CHAPTER 8

MIDP Application Model

Due to strict memory constraints and the requirement to support application interaction and data sharing within related applications, the Mobile Information Device Profile does not support the familiar Applet model introduced by Java™ 2 Platform, Standard Edition (J2SE™). Rather, MIDP introduces a new application model that was designed to augment the CLDC application model and to allow multiple Java applications to share data and run concurrently on the KVM.

In order to understand the MIDP application model, it is useful to first examine the limitations of the CLDC application model.

8.1 Limitations of the CLDC Application Model

The CLDC application model (see Section 5.2, "CLDC Application Model") is intentionally very simple. In the *CLDC Specification*, the term *Java application* refers to a collection of class files that contain a unique launch point identified by the method `public static void main(String[] args)`. While this model is general and is familiar to J2SE programmers, it is not suitable for environments in which a graphical user interface is needed, or in which there is a need to share data between multiple Java applications—especially in light of the security constraints defined by the CLDC security model (see Section 4.2, "Security").

The limitations of the CLDC application model arise primarily from the fact that having more than one application running in the absence of the full Java 2 Standard Edition security model could pose security risks. These security risks include the sharing of *external resources* (such as persistent storage) and *static fields of classes.* For instance, consider a situation in which a virtual machine conforming to CLDC is running two applications: `Application1` and `Application2`. Suppose that `Application1` creates a persistently stored object[1] named `Application1_data.rms`. If the application model does not prevent applications from

accessing data and resources created by other applications, `Application2` could now legitimately open and read `Application1_data.rms`. Similarly, if applications were allowed to share classes, then data stored in the static fields of classes of `Application1` could be accessed by `Application2`.

Several possible solutions to these issues were examined by the CLDC and MIDP expert groups. For example, one proposed solution was to make the KVM support multiple "logical virtual machines." In this solution, several logically isolated Java virtual machines would run inside the same physical virtual machine, each sharing the non-writable portion of their memory space (that is, code and read-only memory locations). Each logical virtual machine would have a separate object heap, and each virtual machine would have a separate "root" for its name space.[2]

Using the persistent storage example above, when `Application1` creates `Application1_data.rms`, the virtual machine would create it in a place that `Application2` could not access. Furthermore, since the static fields of classes would reside in separate object heaps, `Application1` and `Application2` could not interact via these shared resources.

While this idea and other proposed solutions partially or totally solved the security issues, there was still a fundamental problem: without the ability for applications to interact, designing suites of applications that work together would be impossible. Even though this restriction would not be too severe for CLDC, which does not define any APIs for accessing shared resources, it would be overly limiting for MIDP devices, in which users expect some level of data sharing and interaction. The MIDP application model was created to address this need.

Table 8.1 Classes in package `javax.microedition.midlet`

Classes in `javax.microedition.midlet`	**Description**
`MIDlet`	Superclass of all MIDP applications. Extended by a MIDlet to allow the system to start, stop, and destroy it.
`MIDletStateChangeException`	Thrown when the application cannot make the change requested.

1. This is a hypothetical example, since CLDC does not provide any file or database APIs.
2. For the purpose of this discussion, the term *name space* denotes a logical path by which a given object can be found in the system. For example, an RMS database might have a logical path "`/my_midlet/storage1.rms`".

8.2 MIDlets

In MIDP, the basic unit of execution is a *MIDlet*. A MIDlet is a class that extends the class `javax.microedition.MIDlet` (see Table 8.1).

As an example of programming with the MIDP application model, consider the following program that implements one of the simplest MIDlets possible: the canonical "Hello World" application:

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

public class HelloWorld extends MIDlet implements CommandListener {
    private Command exitCommand;
    private TextBox tb;

    public HelloWorld() {
        exitCommand = new Command("Exit", Command.EXIT, 1);
        tb = new TextBox("Hello MIDlet", "Hello, World!", 15, 0);
        tb.addCommand(exitCommand);
        tb.setCommandListener(this);
    }

    protected void startApp() {
        Display.getDisplay(this).setCurrent(tb);
    }

    protected void pauseApp() {}
    protected void destroyApp(boolean u) {}

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

The output of this simple MIDlet, when run on a MIDP device emulator, is shown in the figure on the left. In this figure, the MIDlet's `startApp` method has just completed its execution. Hitting the upper-rightmost button (the *Exit* command) on the phone's keypad would invoke the `commandAction` method and run the `destroyApp` and `notifyDestroyed` code.

In this example, there are a few key things that are common to all MIDlets, no matter how simple or complex the applications are. First, HelloWorld extends the class `javax.microedition.midlet.MIDlet`.[3]

Like every Java class, a MIDlet can have a constructor. In the MIDP application model, the system (see Section 8.4, "MIDP System Software") calls the public no-argument constructor of a MIDlet exactly once to instantiate the MIDlet. The functionality that needs to be defined in the constructor depends on how the MIDlet is written, but in general, any operations that must be performed only once when the application is launched should be placed in the constructor.

If no such functionality is required by the MIDlet, then there is no need to provide a constructor. Care should be taken in the constructor to catch any exceptions and handle them gracefully, since the behavior of an uncaught exception at the MIDlet level is undefined.

Class `javax.microedition.midlet.MIDlet` defines three abstract methods, `startApp`, `pauseApp`, and `destroyApp`, which must be defined by all MIDlets. The `startApp` method is generally used for starting or restarting a MIDlet. This method may be called by the system under different circumstances (see Section

[3] Note that the MIDlet also implements the `CommandListener` interface. This interface, along with the `TextBox`, `Command`, and `Display` classes, is part of the `javax.microedition.lcdui` package that is discussed in Chapter 9, "MIDP User Interface Libraries."

8.2.1, "MIDlet states"), but its purpose is to acquire or re-acquire resources needed by the MIDlet and to prepare the MIDlet to handle events such as user input and timers. Note that the `startApp` method may be called more than once—once to start execution for the first time, and again for every time the MIDlet is "resumed"; therefore, a `startApp` method should be written so that it can be called under these different circumstances.

The `startApp` method can "fail" in two ways: transient and non-transient. A transient failure is one that might have to do with a certain point in time (such as a lack of resources, network connectivity, and so forth). Since this type of error may not be fatal, a MIDlet can tell the system that it would like to try and initialize later by throwing a `javax.microedition.midlet.MidletStateChangeException`. A non-transient failure is one that is caused by any other unexpected and uncaught runtime error. In this case, if the runtime error is not caught by the `startApp` method, it is propagated back to the system that will destroy the MIDlet immediately and call the MIDlet's `destroyApp` method.

A robust `startApp` method should distinguish between transient and non-transient exceptions and respond accordingly, as illustrated in the following code fragment:

```
void startApp() throws MIDletStateChangeException {
    HttpConnection c = null;
    int status = -1;
    try {
        c = (HttpConnection)Connector.open(...);
        status = c.getResponseCode();
        ...
    } catch (IOException ioe) {
        // Transient failure: could not connect to the network
        throw new MIDletStateChangeException(
                    "Network not available");
    } catch (Exception e) {
        // Non-transient failure: can either
        // catch this and exit here, or let the
        // exception propagate back to the system
        destroyApp(true);
        notifyDestroyed();
    }
}
```

The `pauseApp` method is called by the system to ask a MIDlet to "pause." The precise definition of what pause means is discussed in Section 8.2.1, "MIDlet

states," but in general, the `pauseApp` method works in conjunction with the `startApp` method to release as many resources as possible so that there is more memory and/or resources available to other MIDlets or native applications. The use of the `pauseApp` method is illustrated in the following code fragment:

```
boolean firstTime= false;
int[] runTimeArray = null;
...
void startApp() {
    runTimeArray = new int[200];
    if (firstTime) {
        // first time, initialize array with some values, etc.
        ...
        firstTime = false;
    } else {
        // coming from a paused state, read data
        // saved from a RecordStore and put into
        // runTimeArray
        ...
    }
}

void pauseApp() {
    // open a RecordStore and save runTimeArray values;
    // set runTimeArray to null so that it is a candidate
    // for garbage collection.
    ...
    runTimeArray = null;
}
```

The last method that a MIDlet must implement is the `destroyApp` method. This method is called by the system when the MIDlet is about to be destroyed; it can also be called indirectly (that is, by calling `notifyDestroyed`) by the MIDlet itself in order to clean up before exiting. In either case, the `destroyApp` method should be written so that it performs all the necessary clean up operations to release all the resources (that is, close the graphical user interface components, network connections, database records) that the application had allocated during its execution. The following code fragment provides a simple example:

```
RecordStore rs = null;
...
```

```
void startApp() {
    ...
    rs = RecordStore.openRecordStore(...);
    ...
}
void destroyApp(boolean u) {
    if (rs != null) {
        // make sure all records are written to storage
        // and close the record store
        ...
    }
}
```

Taken as a whole, the aforementioned methods—`startApp`, `pauseApp`, and `destroyApp`—represent a state machine with each method responsible for entry or exit to a given state. The next section describes these MIDlet states and the rules that govern the transitions between the different states of a MIDlet.

8.2.1 MIDlet states

During the lifetime of a MIDlet, it may be in one of three distinct states, with well-defined rules that govern the transitions between these states (see Figure 8.1):

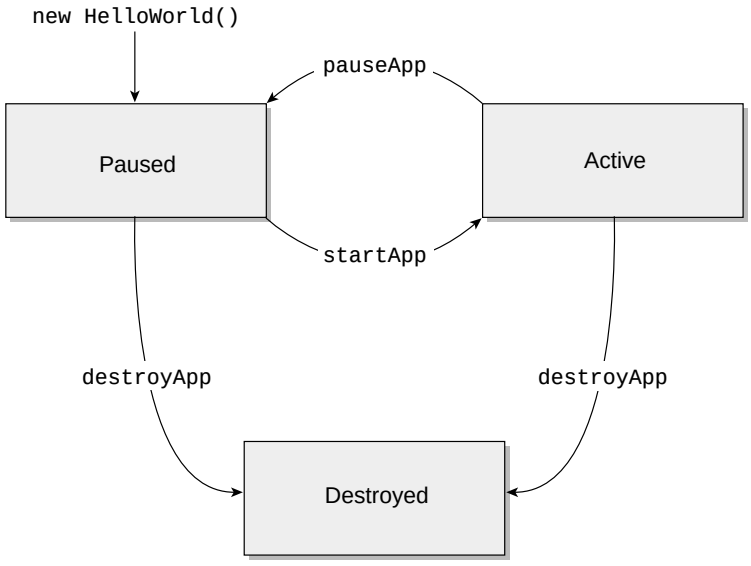

Figure 8.1 MIDlet states and state transitions

- *Paused:* A MIDlet is in the *Paused* state when it has just been started and has not yet entered its `startApp` method. It can also be in the *Paused* state as a result of the `pauseApp` or the `notifyPaused` methods (see below). When in the *Paused* state, a MIDlet should hold as few resources as possible. A MIDlet in the *Paused* state can receive asynchronous notifications, for example, from a timer firing (see Section 12.1, "Timer Support").

  Since CLDC does not provide guaranteed real-time behavior, the system is not required to use the *Paused* state to manage the interaction of the real-time portions of the phone and the Java environment. Consider, for example, the case where the virtual machine is running on a cellular phone, and the phone receives a call. In this scenario, the real-time operating system on the phone might suspend the virtual machine altogether rather than cycle the MIDlets to the *Paused* state.

- *Active*: A MIDlet is in the *Active* state upon entry to its `startApp` method. In addition, a MIDlet can transition from the *Paused* state to the *Active* state as the result of the `resumeRequest` method. While in the *Active* state, a MIDlet can allocate and hold all necessary resources for optimal execution.

- *Destroyed:* A MIDlet is in the *Destroyed* state when it has returned from the `destroyApp` or `notifyDestroyed` methods. Once a MIDlet enters the *Destroyed* state, it cannot reenter any other state.

The *Destroyed* state replaces the normal convention of an application calling the `System.exit` method to terminate. A MIDlet cannot call the `System.exit` method, since doing so will throw a `java.lang.SecurityException` (see Section 12.4, "Exiting a MIDlet.")

The MIDlet state transitions can be triggered either by the MIDP system itself or by the application programmer. Below we summarize how the MIDP system can initiate these state transitions. Note that all these method calls are synchronous in that the state changes are not complete until the method returns to the caller.

- `startApp`: This method is called by the system to move a MIDlet into the *Active* state for the first time, and also when a MIDlet is being resumed from the *Paused* state. When a MIDlet enters its `startApp` method, it is in the *Active* state. An important note: Beginning MIDlet programmers are often tempted to treat `startApp` as equivalent to the `main` method, and thus combine MIDlet initialization and main-line processing in the `startApp` method. In general this is not a good idea, since the `startApp` method may be called more than once: one time initially, and then once per each transition from *Paused* to *Active* state.

- `pauseApp`: During the normal operation of a mobile device, the system might run into situations where it is necessary to suspend or pause some of the MIDlets on the device. The most common example is when a device is running low on memory. In order to reclaim memory, the system may call the `pauseApp` method of all the *Active* MIDlets. When the `pauseApp` method of a MIDlet is called, the MIDlet should release as much of its resources as possible and become quiescent. Note that the system does not actually try to force any specific behavior on a MIDlet in a *Paused* state, so it is possible for an ill-behaved MIDlet to ignore this request. However, if such a situation occurs, the system may forcibly terminate the MIDlet or even the virtual machine itself, especially in a low-memory situation. In addition, note that while in the *Paused* state the MIDlet can still receive asynchronous events such as timer events.
- `destroyApp`: This is the normal way the system terminates a MIDlet. The `destroyApp` method has one boolean parameter that indicates whether or not the request is unconditional. If the request is not unconditional (that is, the boolean parameter is *`false`*), then the MIDlet may request a "stay of execution" by throwing a `MIDletStateChangeException`. In this case, if the system is able to honor the request, the MIDlet may continue in its current state (*Paused* or *Active*). If, on the other hand, the request is unconditional, then the MIDlet should give up its resource, save any persistent data it might be caching, and return. Upon return from this method, the MIDlet enters the *Destroyed* state and can be reclaimed by the system.

While the MIDP system is the primary instigator of MIDlet state changes, a MIDlet application programmer can also request state changes via the following methods:

- `resumeRequest`: This method can be called by a *Paused* MIDlet to indicate that it wishes to reenter the *Active* state. The primary scenario for this call is when a *Paused* MIDlet handles a timer expiration and needs to resume processing.
- `notifyPaused`: This method is provided to allow a MIDlet to signal to the system that it has voluntarily entered the *Paused* state (that is, it has released its resources and is now quiescent). An example use case for this call is a time-based MIDlet that sets timers and has nothing to do until those timers expire.
- `notifyDestroyed`: This method can be called by the MIDlet to tell the system that the MIDlet has released all its resources and has saved any cached data to persistent storage. Note that the `destroyApp` method of a MIDlet will *not* be called as a result of invoking this method.

8.3 MIDlet Suites

One of the central goals for the MIDP application model is to provide support for the controlled sharing of data and resources between multiple, possibly simultaneously running MIDlets. This means that the security issues discussed in the beginning of this chapter need to be addressed. To accomplish this, the *MIDP Specification* requires that in order for MIDlets to interact and share data, they must be placed into a single JAR file. This collection of MIDlets encapsulated in a JAR file is referred to as a *MIDlet suite*. MIDlets within a MIDlet suite share a common name space (for persistent storage), runtime object heap, and static fields in classes. In order to preserve the security and the original intent of the MIDlet suite provider, the MIDlets, classes, and individual files within the MIDlet suite cannot be installed, updated, or removed individually—they must be manipulated as a whole. In other words, the basic unit of application installation, updating, and removal in MIDP is a MIDlet suite.

A MIDlet suite can be characterized more precisely by its *packaging* and its *runtime environment*. These characteristics are discussed in more detail below.

8.3.1 MIDlet suite packaging

A MIDlet suite is encapsulated within a JAR file. A MIDlet suite provider is responsible for creating a JAR file that includes the appropriate components for the target user, device, network, and locale. For example, since the *CLDC Specification* does not include the full internationalization and localization support provided by Java 2 Standard Edition, a MIDlet suite provider must tailor the JAR file components to include the necessary additional resources (strings, images, and so forth) for a particular locale.

The contents of the MIDlet suite's JAR file include the following components:

- The class files implementing the MIDlet(s)
- Any resource files used by the MIDlet(s): for example, icon or image files, and so forth.
- A manifest describing the JAR contents

All the files needed by the MIDlet(s) are placed in the JAR file using the standard structure based on mapping the fully qualified class names to directory and file names within the JAR file. Each period is converted to a forward slash, '/'. For class files, the `.class` extension is appended.

The JAR manifest provides a means to encode information about the contents of the JAR file.[4] In particular, the JAR manifest specification provides for *name-value pairs* which MIDP uses to encode *MIDlet attributes*. These attributes can be retrieved by a MIDlet with the method `MIDlet.getAppProperty`. Note that MIDlet attribute names are case sensitive. Also note that all the attribute names that start with "`MIDlet-`" are reserved for use by the MIDP expert group to define MIDP-specific MIDlet attributes. MIDlet suite developers can define their own MIDlet attributes, provided that the new attributes do not start with the reserved "`MIDlet-`" prefix. The *MIDP Specification* currently defines the MIDlet attributes shown in Table 8.2.

Of the predefined attributes in Table 8.2, the following must be in the JAR manifest (the other attributes in the table may be optionally included):

- *`MIDlet-Name`*
- *`MIDlet-Version`*
- *`MIDlet-Vendor`*
- `A MIDlet-<`*`n`*`>` for each MIDlet
- *`MicroEdition-Profile`*
- *`MicroEdition-Configuration`*

As an example, the following shows a JAR manifest for a hypothetical MIDlet suite of card games provided by Motorola. The suite contains two MIDlets: `Solitaire` and `JacksWild`:

```
MIDlet-Name: CardGames
MIDlet-Version: 1.1.9
MIDlet-Vendor: Motorola
MIDlet-1: Solitaire, /Solitaire.png, com.motorola.Solitaire
MIDlet-2: JacksWild, /JacksWild.png, com.motorola.JacksWild
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
```

[4] JAR file format specifications are available at `http://java.sun.com/products/jdk/1.2/docs/guide/jar/index.html`. Refer to the JDK JAR and manifest documentation for syntax and related details.

Table 8.2 MIDlet attributes

Attribute Name	Attribute Value Description
`MIDlet-Name`	The name of the MIDlet suite that identifies the MIDlets to the user.
`MIDlet-Version`	The version number of the MIDlet suite. The format is *`major.minor.micro`* as described in the JDK Product Versioning Specification (see `http://java.sun.com/products/jdk/1.2/docs/guide/versioning/spec/VersioningSpecification.html`). It can be used by the system for installation and upgrade uses, as well as communication with the user. Default value is 0.0.0.
`MIDlet-Vendor`	The name of the MIDlet suite provider.
`MIDlet-Icon`	The name of a PNG file to be used as the icon to identify the MIDlet suite to the user.
`MIDlet-Description`	A short description of the MIDlet suite.
`MIDlet-Info-URL`	A URL for information further describing the MIDlet suite.
`MIDlet-`*`<n>`*	The *`name`*, *`icon`*, and *`class`* of the n[th] MIDlet in the JAR file separated by a comma. The lowest value of *`<n>`* must be 1 and consecutive ordinals must be used. *`Name`* is used to identify this MIDlet to the user. *`Icon`* is the name of an image (PNG) within the JAR that the system should use for the icon of the n[th] MIDlet. *`Class`* is the name of the `MIDlet` class for the n[th] MIDlet.
`MIDlet-Jar-URL`	The URL from which the JAR file was loaded.
`MIDlet-Jar-Size`	The size of the JAR file in bytes.
`MIDlet-Data-Size`	The minimum number of bytes of persistent data required by the MIDlet. The default is zero.
`MicroEdition-Profile`	The J2ME profile required, using the same format and value as the system property `microedition.profiles` (for example "`MIDP-1.0`").
`MicroEdition-Configuration`	The J2ME Configuration required using the same format and value as the system property `microedition.configuration` (for example "`CLDC-1.0`").

In addition to the JAR file, the MIDP provides for a separate and optional[5] file called the *application descriptor.* The application descriptor allows the system to verify that the associated MIDlet suite is suited to the device before loading the full JAR file of the MIDlet suite. It also allows MIDlet attributes to be supplied to the MIDlet suite without modifying the JAR file.

The application descriptor is a file that has a MIME type of `text/vnd.sun.j2me.app-descriptor`, and file extension of `.jad`, and contents described by the following BNF syntax:

```
appldesc: *attrline
attrline: attrname ":" WSP attrvalue WSP newline
attrname: 1*<any Unicode char except CTLs or separators>
attrvalue: *valuechar | valuechar *(valuechar | WSP) valuechar
valuechar: <any valid Unicode character, excluding CTLs and WSP>
newline: CR LF | LF
CR = <Unicode carriage return (0x000D)>
LF = <Unicode linefeed (0x000a)>
WSP: 1*( SP | HT )
SP = <Unicode space (0x0020)>
HT = <Unicode horizontal-tab (0x0009)>
CTL = <Unicode characters 0x0000 - 0x001F and 0x007F>
separators = "(" | ")" | "<" | ">" | "@"
           | "," | ";" | ":" | "'" | <">
           | "/" | "[" | "]" | "?" | "="
```

If the application descriptor is present, then it must contain the following predefined attributes (again, the other attributes in Table 8.2 are optional in the application descriptor):

- *MIDlet-Name*
- *MIDlet-Version*
- *MIDlet-Vendor*
- *MIDlet-Jar-URL*
- *MIDlet-Jar-Size*

A rudimentary form of version control between the JAR file and the application descriptor is ensured by requiring that the attribute values for *MIDlet-Name*,

---

[5] Note: The application descriptor is optional for distributors of MIDlet suites (such as cellular carriers and so forth). However, all MIDP-compliant implementations must accept an application descriptor.

MIDlet-Version, and *MIDlet-Vendor* be identical in the JAR manifest and the application descriptor. If they are not, then the system will assume a version mismatch, and the MIDlet suite will not be installed. All other MIDlet attributes names may also be duplicated, but their values can differ. In this case, the value from the application descriptor will override the value from the JAR manifest.

8.3.2 MIDlet suite execution environment

The *MIDP Specification* defines the environment in which MIDlets within a suite execute. This environment is shared by all MIDlets within a MIDlet suite, and any MIDlet can interact with other MIDlets packaged in a suite. The runtime environment is logically composed of name spaces, as depicted in Figure 8.2. These name spaces determine how a MIDlet accesses an entity within that name space.

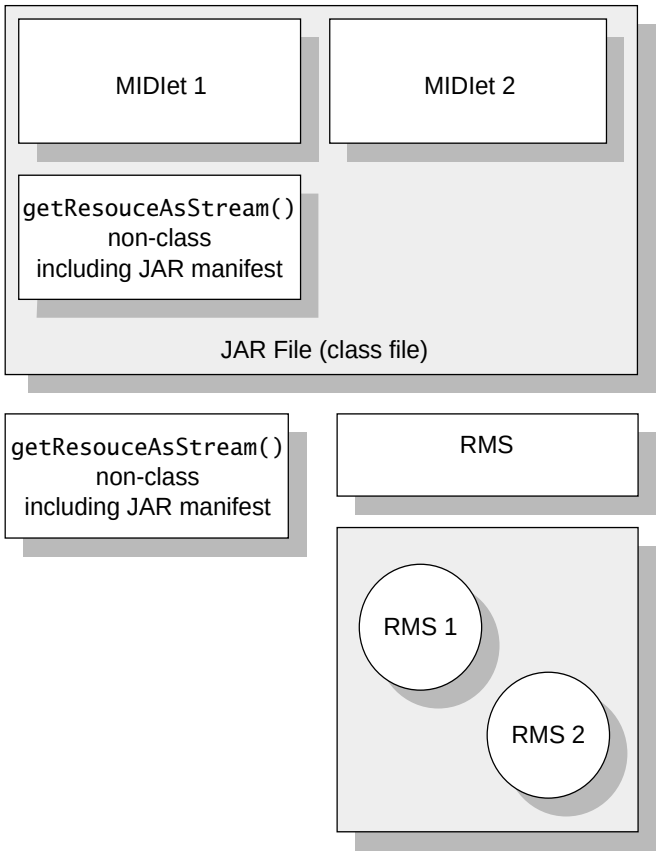

Figure 8.2 MIDlet suite name spaces

The logical name spaces in a MIDlet's runtime environment are as follows:

- Classes and native code that implement the CLDC and MIDP. This is the only name space that is shared by all MIDlet suites on the device.
- Classes within the MIDlet suite's JAR file.
- All non-class files in the MIDlet suite's JAR file, such as icon or image files, and the JAR manifest. These files are accessible via the method `java.lang.Class.getResourceAsStream`.
- The contents of the Application Descriptor File. Accessible via the method `javax.microedition.midlet.MIDlet.getAppProperty`. (Note that the JAR manifest is also accessible via this method.)
- A separate name space for RMS record stores (see Chapter 11, "MIDP Persistence Libraries").

All the classes needed by a MIDlet within a suite must be in the JAR file or in the CLDC and MIDP libraries. A MIDlet may load and invoke methods from any class in the JAR file, in the MIDP libraries, or in the CLDC libraries. All of the classes within these three scopes are shared in the execution environment, along with a single heap containing the objects created by MIDlets, MIDP libraries, and CLDC libraries. The usual locking and synchronization primitives of the Java programming language can be used where necessary to avoid concurrency problems.

The class files of the MIDlet are only available for execution and can neither be read as resources nor extracted for re-use, since the underlying CLDC implementation may store and interpret the contents of the JAR file in any manner suitable for the device.

The non-class files within the JAR file, including the JAR manifest, can be accessed using the method `java.lang.Class.getResourceAsStream`. The parameter to this method, a `String` object, represents a path that is interpreted in the following manner: If it begins with a '/', the search for the resource begins at the "root" of the JAR file; however, if it does not begin with a '/', the resource is searched for along a path relative to the class instance retrieving the resource. For example, if a JAR file has the following contents as shown in this JAR manifest:

```
META-INF/
META-INF/MANIFEST.MF
examples/
examples/HelloWorld.class
```

then, in the following code fragment, both `getResourceAsStream` calls will open an `InputStream` to the manifest file.

```
Class C = Class.forName("examples.HelloWorld");
InputStream s1 = C.getResourceAsStream("/META-INF/MANIFEST.MF");
InputStream s2 = C.getResourceAsStream("../META-INF/MANIFEST.MF");
```

8.4 MIDP System Software

In the preceding sections, frequent references were made to the piece of software called the "system," "MIDP system" or the "execution environment." For example, in Table 8.1, the description of the `MIDlet` class includes the following text: "Extended by a MIDlet to allow the *system* to start, stop, and destroy it." The *MIDP Specification* calls this software the *application management software*, or the AMS. In some documents, application management software is also referred to as the *Java Application Manager*, or the JAM. The two terms (AMS and JAM) are equivalent.

The application management software in a MIDP implementation consists of those pieces of software on the device that provide a framework in which MIDlet suites are installed, updated, removed, started, stopped, and in general, managed. Furthermore, the application management software provides MIDlets with the runtime environment discussed in Section 8.3.2, "MIDlet suite execution environment."

8.4.1 Application management functionality

In the *MIDP Specification*, the application management software is presumed to implement a minimum set of functionality listed in Table 8.3.

Before a MIDlet can be launched, the associated JAR file (MIDlet suite) must be retrieved from some source. A device may be able to retrieve MIDlet suites from multiple transport mediums. For example, a device may support retrieval via a serial cable, an IrDA (infrared) port, or a wireless network. The application management software must support a *medium-identification* step in which the retrieval medium can be selected, either automatically or by user input. After selecting the retrieval medium, the application management software can initiate the *negotiation* step, where information about the MIDlet suite

Table 8.3 Typical application management operations

Operation	Description
Retrieval	Retrieves a MIDlet suite from some source. Possible steps include *medium-identification, negotiation,* and *retrieval.*
Installation	Installs a MIDlet suite on the device. Possible steps include *verification* and *transformation.*
Launching	Invokes a MIDlet. Possible steps include *inspection* and *invocation.*
Version management	Allows installed MIDlet suites to be upgraded to newer versions. Possible steps include *inspection* and *version management*.
Removal	Removes a previously installed MIDlet suite. Possible steps include *inspection* and *deletion*.

and the device is exchanged and compared (for example, the application descriptor may be used in this step.) This information can include the device's capability (for example, available memory), the size of the MIDlet, and so forth. Upon verifying that the device can potentially[6] install the MIDlet suite, the *retrieval* step begins. In this step, the device transfers the MIDlet suite to the device.

Once the MIDlet suite has been retrieved, the installation process may begin. A MIDP implementation may need to *verify* that the retrieved MIDlet suite does not violate the device's security policies. For example, a device might enforce some sort of "code signing" mechanism to validate that the retrieved MIDlet suite is from a trusted source (the *MIDP Specification* does not require this, however.) The next step in installation is the *transformation* from the public representation of the MIDlet suite (for example, the compressed JAR file) into some device-specific, internal representation. This transformation may be as simple as storing the JAR file to persistent storage, or it may actually entail preparing the MIDlet to execute directly from non-volatile memory.

After installation, the MIDlets within the MIDlet suite can now be *launched*. Launching a MIDlet means that the user is presented with a selection of installed MIDlets that are gathered by the device performing the *inspection* step. The user may then select one of the MIDlets for the device to run (or *invoke*). Invocation is the point at which the application management software actually starts to run the

6. Potentially, because when the MIDlet suite is internalized for execution on a particular device, it may in fact be too large to fit in non-volatile memory.

MIDlet on the virtual machine. At this point, the methods discussed in Section 8.3.1, "MIDlet suite packaging" are used to control the states of MIDlet.

At some point after installation, a new version of a MIDlet suite may become available. To upgrade to this new version, the application management software must keep track of what MIDlet suites have been installed (*identification*) and their "version number" (*version management)*. Using this information, the older version of a MIDlet suite can be upgraded to the newer version. Again, the *MIDP Specification* does not allow for individual MIDlets to be upgraded—rather, it specifies that the entire MIDlet suite must be upgraded as a unit. This is done to ensure that the original intent of the MIDlet suite provider is not changed, nor its security model compromised.

A related concept is MIDlet suite *removal*. This differs only slightly from the previous step in that after performing *inspection*, the application management software *deletes* the installed MIDlet suite and its related resources. This includes the removal of the records that the application has written to persistent storage via the APIs described in Chapter 11, "MIDP Persistence Libraries."

As mentioned before, the application management software is not specified by the *MIDP Specification*; therefore, all the above discussions are merely possibilities. An application programmer should not expect that every device implements all of the functionality mentioned above.