

- 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](#)

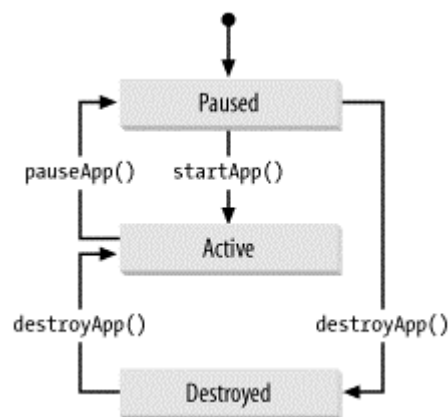
[Chapter 9](#)

## Chapter 4. Working with MIDlets

MIDlets are very simple to implement. All MIDlets must extend the `javax.microedition.midlet.MIDlet` abstract class and implement certain methods in that class. The `MIDlet` abstract class provides the basic functionality required in all MIDlets. A MIDlet runs in a controlled environment and therefore must implement certain methods that allow the *application manager* (which installs and runs the MIDlet) to control the behavior of the MIDlet. These methods are known as *life cycle methods*, since they reflect various states in which a MIDlet can be.

You'll recall from the previous chapter that a MIDlet can be in one of three states: paused, active, or destroyed. The state chart in [Figure 4-1](#) shows the possible state transitions of a MIDlet, this time with the addition of the methods that the Java Manager will call inside the MIDlet code during those transitions.

**Figure 4-1. MIDlet state transitions**



Here, the `javax.microedition.midlet.MIDlet` abstract class defines three life cycle methods that are called during the state transitions: `pauseApp()`, `startApp()`, and `destroyApp()`. These three methods were present in the example we developed in [Chapter 1](#). The responsibilities for these three life cycle methods are as follows.

```
public void startApp()
```

This method indicates that the MIDlet is moving from a paused state to an active state. Here, the MIDlet will typically initialize any objects that are required while the MIDlet is active, and set the current display.

```
public void pauseApp()
```

This method is called when the MIDlet is moving from an active state to a paused state. This means that it will pause any threads that are currently active, as well as optionally setting the next display to be shown when the MIDlet is re-activated. Data can be persisted, if necessary, and retrieved later when the MIDlet is activated again.

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

This method indicates that the MIDlet is moving to the destroyed state. It should free or close all resources that have been acquired during the life of the

MIDlet. In addition, the method should persist any data that it wishes to save for future use.

It is important to note that `startApp()` can be called more than once. In addition to being called when the MIDlet is first activated to move the MIDlet from the paused state to the active state, it can also be called if the MIDlet has been paused during execution and wishes to again return to the active state.

## 4.1 The Application Manager

The application manager, sometimes called the Application Management System (AMS) or MIDlet management software, is software that is preinstalled on a MIDP device and that functions as an operating environment. For example, on a Motorola i85s, the Java Apps menu item will start the application manager, which immediately shows the Java logo and the words "Mobile Information Device Profile Compatible" and then displays a menu of the MIDlet suites that have been installed on the phone.

However, the application manager must do more than simply show a menu of the MIDlet suites that are installed. According to the MIDP specification, the application manager must be able to:

- Retrieve a MIDlet suite from somewhere, possibly through a serial connection, infrared connection, or across a wireless connection to the Internet
- Install a MIDlet suite on the MIDP device
- Perform version management on MIDlet suites that are installed
- Launch a MIDlet from a MIDlet suite and provide an operating environment for the KVM, as well as any system, MIDP, and CLDC classes
- Delete a previously installed MIDlet suite

As a MIDlet programmer, you typically won't need to be concerned with the internals of the application manager running on the device—it's unique to each device. However, some insight into its responsibilities is important when designing MIDP applications. In this case, the MIDlet life cycle methods can be called by the application manager to control the MIDlet state:

- When the user launches a MIDlet, the application manager creates a new instance of the MIDlet class by calling its zero-argument constructor. This typically performs the one-time initialization. Once this is done, the MIDlet will be placed in a paused state. However, if any exception occurs during the instantiation of the MIDlet class, the application manager will move the class to the destroyed state.
- After the MIDlet has been placed in the paused state, the application manager calls `startApp()` to transition it to the active state.
- The application manager can then call `pauseApp()` to move it from the active state to the paused state, either via a request from the program itself or from the operating environment.
- `destroyApp()` can be called by the application manager to transition the MIDlet to the destroyed state. The `destroyApp()` method takes a `boolean` argument to indicate if the MIDlet should clean up or not.

Example 4-1 shows a MIDlet skeleton class that implements the life cycle methods of the `javax.microedition.midlet.MIDlet` class.

#### Example 4-1. MIDlet skeleton

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

public class MyMIDlet extends MIDlet {

 public MyMIDlet() {
 // constructor
 }

 public void startApp() {
 // entering active state
 }

 public void pauseApp() {
 // entering paused state
 }

 public void destroyApp(boolean unconditional) {
 // entering destroyed state
 }
}
```

Believe it or not, this class is all you need to create a MIDlet. The only thing we should reiterate is our earlier warning that `startApp( )` can be called more than once. Hence, if you have any one-time initialization that you wish to perform for your MIDlet, be sure that it is placed in the constructor of the MIDlet object and not in the `startApp( )` method.

Earlier, we mentioned that a MIDlet could change its own state if needed. The `javax.microedition.midlet.MIDlet` abstract class provides three methods that can be called by a MIDlet to control its own state transitions:

```
public void notifyPause()
```

A MIDlet may call this method to pause itself. It can be called while in the active state, to inform the Java Application Manager that the MIDlet has entered the paused state.

```
public void resumeRequest()
```

A MIDlet may call this method to express interest in entering the active state. The application manager also calls this method to determine which MIDlet to activate, then it calls its `startApp( )` method.

```
public void notifyDestroyed()
```

A MIDlet calls this method to destroy itself. It can be called while in the active state or the paused state, to indicate to the application manager that it has entered the destroyed state. Note that the application manager will not call

`destroyApp( )`. Consequently, the MIDlet manages the release of its resources.

## 4.2 Creating MIDlets

Now that you're familiar with MIDlet states and the application manager, let's create another MIDlet. As you've probably guessed by now, this involves the following five steps:

1. Write the MIDlet.
2. Compile the MIDlet's source code.
3. Preverify the MIDlet's class file.
4. Package the application in a JAR file.
5. Create a JAD file.

Let's review each of these steps. First, we'll look at the command-line technique that was shown in [Chapter 1](#). Then, we'll introduce the KToolbar application, which comes with the J2ME Wireless Toolkit and which can make our lives much easier.

### 4.2.1 Write the MIDlet

The first step in the development life cycle is to write the MIDlet. [Example 4-2](#) shows a simple MIDlet, `PaymentMIDlet`. This MIDlet creates a `List` object of type `EXCLUSIVE` (that is, only one option can be selected at a time), and adds three methods of payments to it. It displays a list of options for the user to select a method of payment.

#### Example 4-2. Sample MIDlet

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

public class PaymentMIDlet extends MIDlet {

 // The display for this MIDlet
 private Display display;

 // List to display payment methods
 List method = null;

 public PaymentMIDlet() {
 method = new List("Method of Payment",
 Choice.EXCLUSIVE);
 }

 public void startApp() {
 display = Display.getDisplay(this);
 method.append("Visa", null);
 method.append("MasterCard", null);
 method.append("Amex", null);
 display.setCurrent(method);
 }

 /**
 * 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) {
 }
}

```

### 4.2.2 Compile the Source Code

To compile the source code with the command-line tools of the Java Wireless Toolkit, use the `javac` command. Remember that you should use the `-bootclasspath` option to make sure the source code is compiled against the correct CLDC and MIDP classes.

```

C:\midlets> javac -bootclasspath C:\j2mewtk\lib\midpapi.zip
PaymentMIDlet.java

```

This command produces the `PaymentMidlet.class` file in the current directory. This is a slightly simplified version of the command we used in [Chapter 1](#), which puts the resulting class file in a temporary directory.

### 4.2.3 Preverify the Class File

The next step is to preverify the class file using the `preverify` command:

```

C:\j2mewtk\bin> preverify -classpath
C:\midlets;C:\j2mewtk\lib\midpapi.zip
PaymentMIDlet

```

Again, a slightly different approach. This command creates an *output* subdirectory in the current directory and writes a new file `PaymentMIDlet.class`. This is the preverified class that the KVM can run with its modified class verifier.

### 4.2.4 Package the Application in a JAR File

In order to enable dynamic downloading of MIDP applications, the application must be packaged in a JAR file. To create a JAR file, use the `jar` command:

```

C:\midlets> jar cvf payment.jar PaymentMidlet.class

```

### 4.2.5 Create a JAD File

A JAD file is necessary if you want to run a CLDC-compliant application. [Example 4-3](#) shows a sample JAD file for the payment MIDlet.

### Example 4-3. A sample JAD file

```
MIDlet-1: payment,,PaymentMIDlet
MIDlet-Name: Payment
MIDlet-Version: 1.0
MIDlet-Vendor: ORA
MIDlet-Jar-URL: payment.jar
MIDlet-Jar-Size: 961
```

Once you have the JAD file, you can test your application using the MIDP emulator using the `emulator` command of the Java Wireless Toolkit, as shown here:

```
C:\j2mewtk\bin> emulator -Xdescriptor:C;\midlets\payment.jad
```

If all goes well, activate the MIDlet and you will see output similar to [Figure 4-2](#).

**Figure 4-2. Running the payment MIDlet**



If your MIDP application consists of multiple MIDlets, they can all be in one JAR file as a MIDlet suite. However, you would need to specify them in the JAD file using the `MIDlet-n` entry, where *n* is the number of the MIDlet. Consider the JAD file in [Example 4-4](#), with three hypothetical MIDlets.

### Example 4-4. Three hypothetical MIDlets

```
MIDlet-1: Buy, , BuyMidlet
MIDlet-2: Sell, , SellMidlet
MIDlet-3: Trade, , TradeMidlet
MIDlet-Name: Trading
MIDlet-Version: 1.0
MIDlet-Vendor: ORA
MIDlet-Jar-URL: trade.jar
MIDlet-Jar-Size: 2961
```

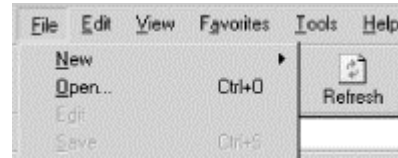
If you run this JAD file, you would see something similar to [Figure 4-3](#).

**Figure 4-3. MIDlet suite**



A MIDP application may consist of multiple MIDlets, as shown in [Figure 4-3](#). Similarly, a desktop application consists of menus and options, as shown in [Figure 4-4](#).

**Figure 4-4. Desktop application**

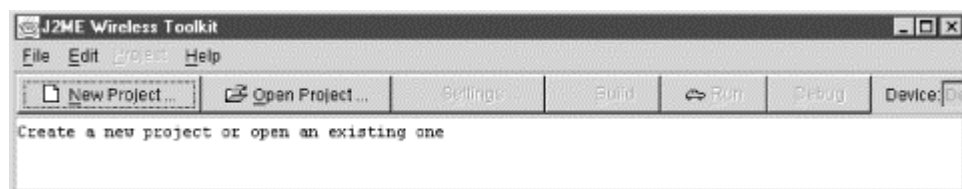


#### 4.2.6 Simplifying the Development

You have now seen how to compile, preverify, create JAR and JAD files, and run MIDlets from the command line. This is fine if you want to understand what's happening behind the scenes. However, there is an alternative. An integrated development environment, such as the J2ME Wireless Toolkit, can be used to simplify the development and deployment of MIDlets. The J2ME Wireless Toolkit comes with an application called KToolbar. The following steps show how to use the KToolbar to set up a simple MIDlet, develop the application, package it, and run it.

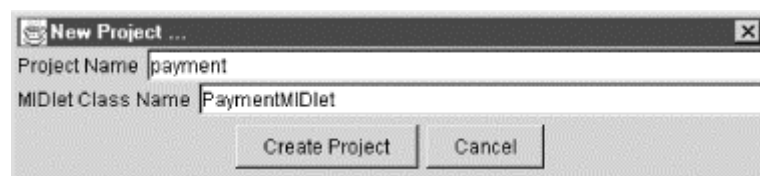
1. In Microsoft Windows, choose Start → Programs → J2ME Wireless Toolkit → KToolbar to start the development environment. [Figure 4-5](#) shows the resulting KToolbar screen.

**Figure 4-5. KToolbar screen**



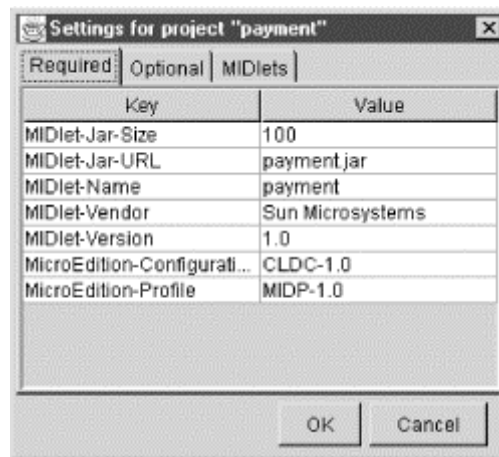
2. Click on the New Project button to create a new project called `payment`, and call the MIDlet class `PaymentMIDlet`, as shown in [Figure 4-6](#).

**Figure 4-6. New project**



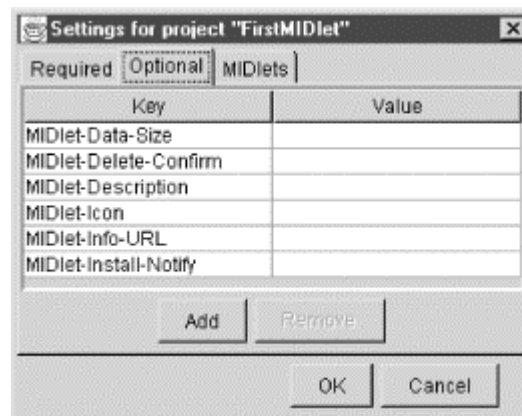
3. Once you click on Create Project in [Figure 4-6](#), you will get a setting project window, as shown in [Figure 4-7](#). This window allows you to modify the MIDlet attributes. All the required attributes are shown in [Figure 4-7](#).

**Figure 4-7. Required attributes**



- If you click on the Optional tab, you will get a window with all the optional attributes, which are shown in [Figure 4-8](#).

**Figure 4-8. Optional attributes**



- Once you click OK, you will get the original KToolbar screen with information to indicate where to save your source and resource files. Assuming the Wireless Toolkit is installed in the directory *C:\J2MEWTK*, then you will be told to save your Java source files in *C:\J2MEWTK\apps\payment\src* and your resource files (e.g., icons) in *C:\J2MEWTK\apps\payment\res*.

Now, use your favorite text editor and write `PaymentMIDlet`, or simply copy the source from [Example 4-2](#). Then, save it in the required location and click on the Build button to compile it. Note that the KToolbar application performs all the steps of compiling the application, preverifying the classes, compressing them into a JAR file, and creating a corresponding JAD file for you. All you have to do is to click the Run button to run it. Then you can test your MIDlet using a default phone, Motorola's i85s, or a Palm OS, as shown in [Figure 4-9](#).

**Figure 4-9. Select a testing device (upper right corner of KToolbar)**



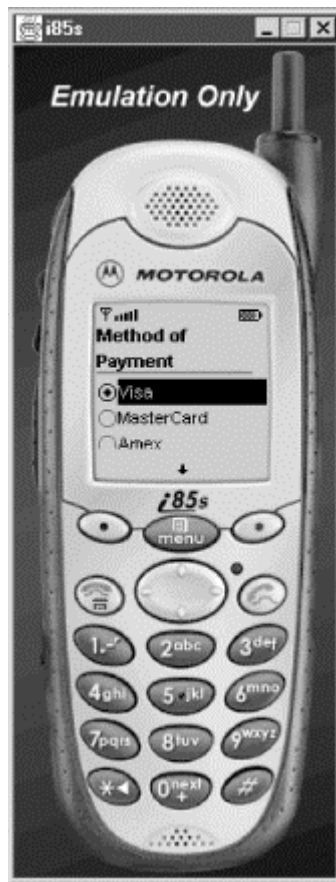
Choose your favorite testing device to test the MIDlet. For example, [Figure 4-10](#) shows the `PaymentMIDlet` running in a default gray phone device.

**Figure 4-10. PaymentMIDlet on the default phone**



[Figure 4-11](#) shows the `PaymentMIDlet` running on Motorola's i85s device.

**Figure 4-11. PaymentMIDlet on the Motorola i85s**

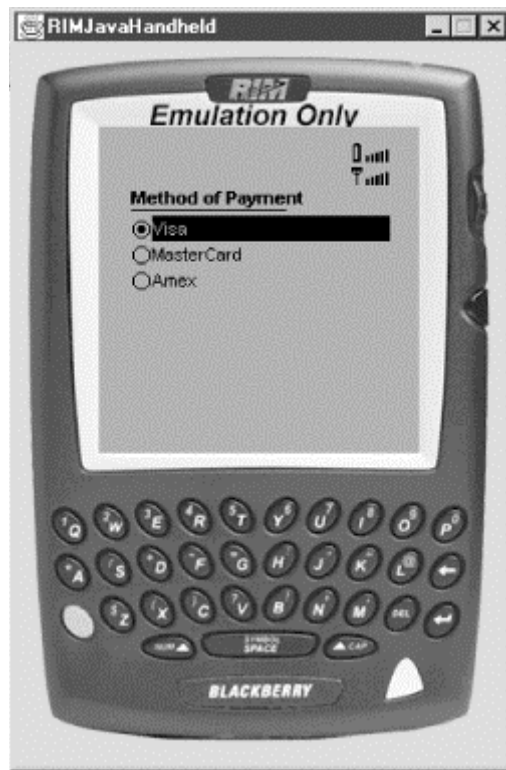


[Figure 4-12](#) shows the same application running on a Palm Pilot and [Figure 4-13](#) shows the `PaymentMIDlet` application running on RIM's BlackBerry. [Chapter 9](#) discusses how to install the Java Application Manager on a real Palm OS device and how to convert existing MIDlets into PRC executable files for handheld devices running Palm OS 3.5 or higher.

**Figure 4-12. PaymentMIDlet on Palm OS**



**Figure 4-13. PaymentMIDlet running on RIM's BlackBerry**



#### 4.2.7 Deploying MIDlets

As of this writing, deploying MIDlets is still an experimental process. However, the Java application manager that comes with the MIDP reference implementation now provides some clues about how we can deploy MIDlets to various devices. Specifically, MIDlets can be installed in two ways:

- Using a data cable or other dedicated physical connection from a local host computer
- Using a network to which the device is intermittently connected

The first method works well with PDAs, which are often used with a host computer, with which the PDAs frequently synchronize their data. For example, the MIDP for Palm implementation, which is discussed in [Chapter 9](#), is a good example of this; its application manager allows MIDlet suites to be installed from a host PC during the synchronization process.

The second method is more popular when installing MIDlets on cell phones and other wireless devices. With these devices, the most likely delivery transport is the wireless network itself. The process of deploying MIDlet suites over a network is referred to as *over-the-air (OTA) provisioning*. OTA provisioning is not yet part of the MIDP specification, but it is likely to become the dominant mechanism for distributing MIDlets and will probably be included in the formal specification soon.

##### 4.2.7.1 Deploying OTA

As of this writing, OTA provisioning is just starting to be used with J2ME devices such as the Motorola i85s/i50x series of cell phones. OTA provisioning allows MIDlet

providers to install their MIDlet suites via web servers that provide hypertext links. This allows you to download MIDlet suites to a cell phone via a WAP or Internet microbrowser. Here is a brief description of how this process works.

First, to deploy a MIDlet from a web server, you need to reconfigure your web server by adding a new MIME type:

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

How to add the MIME type depends on what server you are running. For example, if you're running Apache Tomcat, you would add a new MIME type by adding a new entry in the *web.xml* server configuration file, as follows:

```
<mime-mapping>
 <extension>jad</extension>
 <mime-type>text/vnd.sun.j2me.app-descriptor</mime-type>
</mime-mapping>
```

You would then use the following type of procedure to install a MIDlet suite from a web page:

1. Click on a link, which will probably request a file with a JAD extension, such as the following:

```
Click here to install the MIDlet suite
```

2. The server will then send the *MyApp.jad* file to the phone with the MIME type set to `text/vnd.sun.j2me.app-descriptor`, as described earlier. Recall that the JAD file must contain the `MIDlet-Jar-URL` and `MIDlet-Jar-Size` properties, which tell the device where to download the MIDlet suite, as well as the suite's size in bytes.
3. The Java application manager on the phone will then ask if you want to install the MIDlet into the phone, assuming that the phone has the resources to run the MIDlet (i.e., that there's enough space on the device to hold the MIDlet suite).
4. If you answer yes, the entire JAR file will be downloaded from the server, using the properties specified in the JAD file.

Once the MIDlet is downloaded, it will be installed the first time you try to use it. A downloaded MIDlet stays on the device until you remove it (unlike Java applets).

#### **4.2.7.2 Deploying to the Motorola i50x/i85s**

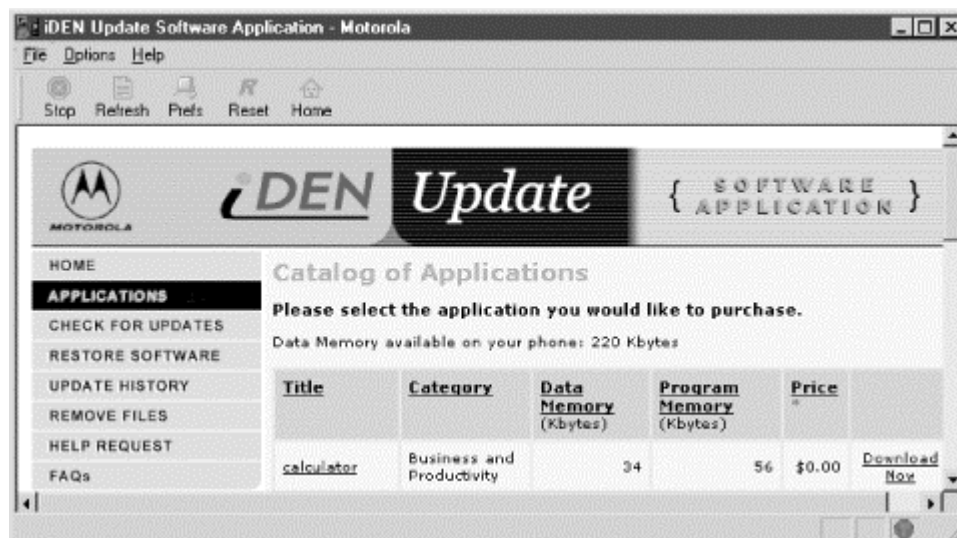
You can also download J2ME applications to a Motorola/Nextel i50x or i85s device from your desktop through a data cable. This cable does not come with the phone itself, but can be ordered online from Nextel. The iDEN update software can then be downloaded from the iDEN development site (<http://www.motorola.com/idendev>).

In addition, you can also purchase a data cable that comes with a CD-ROM containing the iDEN update software from Nextel from this site. Obtaining the software may involve authorization from your carrier, which can take between one and five days.

Once you are granted authorization, however, you can install applications on up to five individual phones. The following paragraphs describe how to use the Motorola iDEN update software to download a J2ME MIDlet to your phone.

After you have obtained the update software, start it up and choose the J2ME Developers tab on the far left. This will result in a screen similar to that in [Figure 4-14](#). From here, you can choose a JAD file to download the application into your phone through the data cable. Note that the JAD file and the JAR file must reside in the same directory and must have the same filename (excluding the extension).

**Figure 4-14. Motorola iDEN update software**



For the most part, downloading an application to the phone is easy. However, the Motorola i85s and i50x phones will perform a number of checks to ensure the integrity of the application while installing it. You should observe the following rules to ensure that the phone will install the application.

The JAD file downloaded to the i85s or i50x must contain at least the following entries, which are case-sensitive:

```
MIDlet-Name:
MIDlet-Version:
MIDlet-Vendor:
MIDlet-Jar-Size:
MIDlet-Jar-URL:
```

It can also contain the following optional entries:

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

In addition, the JAD file can contain any other MIDlet-specific information that does not begin with the letters "MIDlet-".

Remember from [Chapter 3](#) that the JAR file *must* contain a manifest with at least the following information, which must be identical to the data in the JAD file:

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

If you do not include this information in the manifest, the phone will respond with a "Descriptor Error" when it is attempting to install the application. If this happens, simply press the Menu button while the MIDlet is selected and remove it from the system.

Here are some other things to note when downloading to the Motorola i85s or i50x:

- The JAD file is case-sensitive.
- The maximum file length for both the JAD and the JAR file is 16 characters, which includes the four characters for the extension (e.g., *.JAD* or *.JAR*).
- The byte size of the JAR file must be accurately stated in the JAD file.
- Each of the attributes in the JAD and JAR file manifests must have a value associated with it. You cannot leave an attribute value blank.
- Classes which are instantiated using the `Class.forName( )` method must be identified in the JAD file using the attribute: `iDEN-Install-Class-n:`, where `n` is a positive integer. The class name is listed afterward without the `.class` extension.

[Example 4-5](#) shows the manifest information that we would be using if we wanted to download the HelloMidlet application from [Chapter 1](#) to the Motorola i85s.

Remember that the manifest must contain the three specified attributes (`MIDlet-Name`, `MIDlet-Version`, and `MIDlet-Vendor`) and that they must be identical to the values in the JAD file. If they differ, the phone will not install the MIDlet. We have also included the MIDlet class identification information and the profile and configuration version numbers, which we recommend that you include in your MIDlet manifests as well.

#### **Example 4-5. Manifest.mf**

```
MIDlet-Name: HelloMidlet
MIDlet-Vendor: ORA
MIDlet-Version: 1.0.0
MIDlet-1: HelloMidlet,,HelloMidlet
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
```

At this point, let's create a compressed JAR file of the classes that make up the MIDlet. With the manifest and the preverified class in the same directory, enter the following command:

```
>jar cvfm HelloMidlet.jar manifest.mf HelloMidlet.class
```

Once that is completed, you'll need to create the JAD file. [Example 4-6](#) shows the JAD file for our HelloMidlet application. Note that we had to change the value of the `MIDlet-Jar-Size` attribute to match the size, in bytes, of the JAR file that we just

created. In this case, it turned out to be 954 bytes with the additional manifest information.

#### **Example 4-6. HelloMidlet.jad**

```
MIDlet-1: HelloMidlet,,HelloMidlet
MIDlet-Jar-Size: 954
MIDlet-Jar-URL: http://www.oreilly.com/
MIDlet-Name: HelloMidlet
MIDlet-Vendor: ORA
MIDlet-Version: 1.0.0
MIDlet-Description: A sample application
```

Now we're ready to go. Again, be sure that the JAD file and the JAR file have the same name and reside in the same directory. Then use the iDEN software tools to download the application to your phone. It should only take a few seconds once you've chosen the target JAD file. After the download has completed, start the Java Application Manager on the phone (Java Apps under the Main Menu) and select the HelloMidlet application. Press the soft button to install it. You are now installing your first Java MIDlet on a real device. If everything goes okay, you can run your program after it completes the installation and verification steps.

## **Chapter 5. MIDP GUI Programming**

User interface requirements for handheld devices are different from those for desktop computers. For example, the display size of handheld devices is smaller, and input devices do not always include pointing tools such as a mouse or pen input. For these reasons, you cannot follow the same user-interface programming guidelines for applications running on handheld devices that you can on desktop computers.

The CLDC itself does not define any GUI functionality. Instead, the official GUI classes for the J2ME are included in profiles such as the MIDP and are defined by the Java Community Process (JCP). You'll note that the GUI classes included in the MIDP are not based on the Abstract Window Toolkit (AWT). That seems like a major issue, which brings us to the following question.

### **5.1 Why Not Reuse the AWT?**

After a great deal of consideration, the MIDP Expert Group decided not to subset the existing AWT and Project Swing classes for the following reasons:

- AWT is designed for desktop computers and optimized for these machines.
- AWT assumes certain user interaction models. The component set of the AWT is designed to work with a pointing device such as a mouse; however, many handheld devices, such as cell phones, have only a keypad for user input.
- AWT has a rich feature set, and includes support for functionality that is not found or is impractical to implement on handheld devices. For example, the AWT has extensive support for window management, such as resizing overlapping windows. However, the limited display size of handheld devices makes resizing a window impractical. Therefore, the window and layout managers within the AWT are not required for handheld devices.

- When a user interacts with an AWT-based application, event objects are created dynamically. These objects exist only until each associated event is processed by the application or system, at which time the object becomes eligible for garbage collection. The limited CPU and memory of handheld devices, however, cannot handle the burden.

## 5.2 The MIDP GUI APIs

Because of the issues outlined earlier, the MIDP contains its own abbreviated GUI, which is much different from AWT. The MIDP GUI consists of both high-level and low-level APIs, each with their own set of events. This chapter discusses and shows examples of using objects from both the high-level and low-level APIs. Handling events from APIs, however, is deferred to the next chapter.

The high-level API is designed for applications where portability between mobile information devices is important. To achieve portability, the API employs a high-level abstraction and gives you little control over its look and feel. For example, you cannot define the visual appearance (shape, color, or font) of the high-level components. Most interactions with the components are encapsulated by the implementation; the application will not be aware of them. Consequently, the underlying implementation does the necessary adaptation to the device's hardware and native user interface style. Classes that implement the high-level API all inherit the `javax.microedition.lcdui.Screen` class.

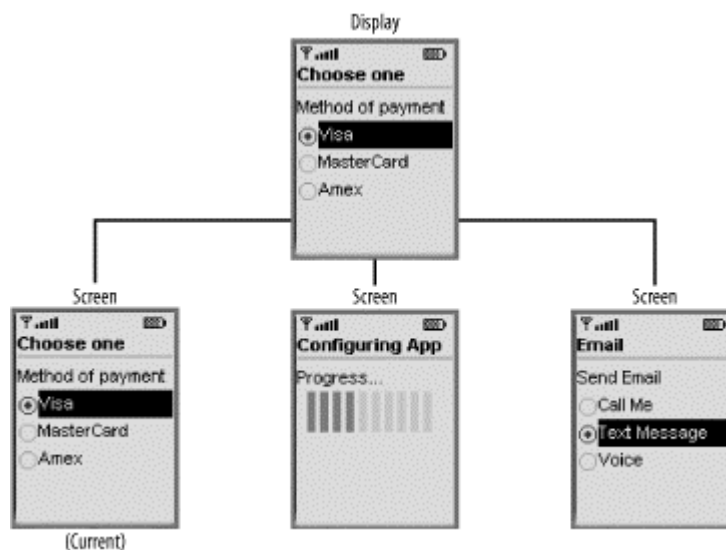
The low-level API provides little abstraction. It is designed for applications that need precise placement and control of graphic elements, as well as access to low-level input events. This API gives the application full control over what is being drawn on the display. The `javax.microedition.lcdui.Canvas` and `javax.microedition.lcdui.Graphics` classes implement the low-level API. However, we should point out that MIDlets that access the low-level API are not guaranteed to be portable, because this API provides mechanisms to access details that are specific to a particular device.

### 5.2.1 The MIDP GUI Model

Here's how the MIDP GUI model works, in a nutshell. In order to show something on a MIDP device, you'll need to obtain the device's *display*, which is represented by the `javax.microedition.lcdui.Display` class. The `Display` class is the one and only display manager that is instantiated for each active MIDlet and provides methods to retrieve information about the device's display capabilities.

Obtaining the device's display is easy. However, this object by itself isn't very interesting. Instead, the more interesting abstraction is the *screen*, which encapsulates and organizes graphics objects and coordinates user input through the device. Screens are represented by the `javax.microedition.lcdui.Screen` object and are shown by the `Display` object by calling its `setCurrent( )` method. There can be several screens in an application, but only one screen at a time can be visible (or *current*) in a display, and the user can traverse only through the items on that screen. [Figure 5-1](#) shows the one-to-many relationship between the display and its screens.

**Figure 5-1. Relationship between display and screens**



There are three types of screens in the MIDP GUI:

- Screens that entirely encapsulate a complex user interface component, such as a `List` or `TextBox` component (the `List` component is shown in [Figure 5-8](#) and the `TextBox` component is shown in [Figure 5-5](#)). The structure of these screens is predefined, and the application cannot add other components to these screens.
- Generic screens that use a `Form` component. The application can add text, images, and a simple set of related UI components to the form, which acts as a container.
- Screens used within the context of the low-level API, such as a subclass of the `Canvas` or `Graphics` class.

## 5.2.2 The `lcdui` Package

All MIDP GUI classes are contained in the `javax.microedition.lcdui` package. This package contains three interfaces and twenty-one classes, as shown in [Table 5-1](#) and [Table 5-2](#).

**Table 5-1. `lcdui` interfaces**

Interface	Description
<code>Choice</code>	Defines an API for a user interface component that implements a selection from a predefined number of choices
<code>CommandListener</code>	Used by applications that need to receive high-level events from implementations
<code>ItemStateListener</code>	Used by applications that need to receive events that indicate changes in the internal state of the interactive items

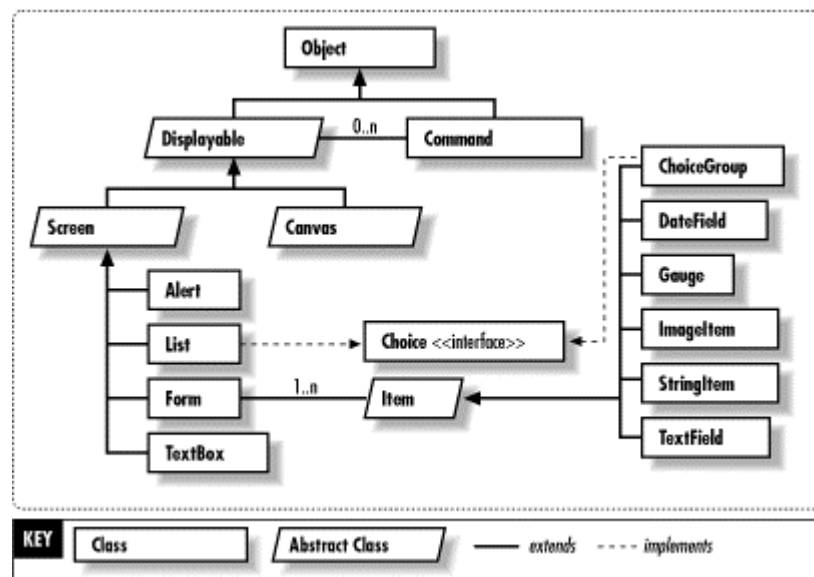
**Table 5-2. `lcdui` classes**

Class	Description
<code>Alert</code>	A screen that shows data to the user and waits for a certain period of time before proceeding to the next screen.
<code>AlertType</code>	A utility class that indicates the nature of the alert.

Canvas	The base class for writing applications that need to handle low-level events and to issue graphics calls for drawing to the display.
ChoiceGroup	A group of selectable elements intended to be placed within a Form.
Command	A construct that encapsulates the semantic information of an action.
DateField	An editable component for presenting calendar data and time information that may be placed into a Form.
Display	A utility that represents the manager of the display and input devices of the system.
Displayable	An object that has the capability of being placed on the display.
Font	A utility that represents font and font metrics.
Form	A screen that contains an arbitrary mixture of items (images, text, text fields, or choice groups, for instance).
Gauge	A utility that implements a bar graph display of a value intended for use in a form.
Graphics	A utility that provides a simple two-dimensional geometric rendering capability.
Image	A utility that holds graphical image data.
ImageItem	A utility that provides layout control when Image objects are added to a form or alert.
Item	A superclass for all components that can be added to a Form or Alert.
List	A screen containing a list of choices.
Screen	The superclass of all high-level user interface classes.
StringItem	An item that can contain a String.
TextBox	A screen that allows the user to enter and edit text.
TextField	An editable text component that can be placed into a Form.
Ticker	A ticker-type piece of text that runs continuously across the display. It can be attached to all screen types except Canvas.

The class diagram in [Figure 5-2](#) shows the major classes and the relationships between them.

**Figure 5-2. Class diagram of the major classes in the Lcdui package**



## 5.4 Creating Low-Level GUI Components

In the high-level API, you have no control of what is displayed on the screen and very little freedom to "play" with the components programmatically. The implementation is responsible for selecting the best approach for the device. Some applications, however, such as games, may need more control over what is drawn on the screen. The MIDP `javax.microedition.lcdui` package also provides a low-level API for handling such cases.

In order to directly draw lines, text, and shapes on the screen, you must use the `Canvas` class. The `Canvas` class provides a blank screen on which a MIDlet can draw. For example, let's draw the string "HelloWorld" on the screen. There's a simple way to do this: subclass the `Canvas` class, which is an abstract class that extends `Displayable`, and override the `paint()` method. The resulting class, `MyCanvas`, is shown in [Example 5-1](#).

The implementation of the `paint()` method uses the drawing capabilities of the `javax.microedition.lcdui.Graphics` class. In the `paint()` method, the drawing color is set to red, then a rectangle is drawn in the current color. The methods `getWidth()` and `getHeight()` return the width and height of the `Canvas`, respectively. The next call to `setColor()` sets the drawing color to white; then the string "Hello World!" is drawn in the top left corner of the screen.

#### Example 5-1. Subclassing Canvas

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

public class MyCanvas extends Canvas {
 public void paint(Graphics g) {
 g.setColor(255, 0, 0);
 g.fillRect(0, 0, getWidth(), getHeight());
 g.setColor(255, 255, 255);
 g.drawString("Hello World!", 0, 0, g.TOP | g.LEFT);
 }
}
```

Now, in order to view the `MyCanvas`, it must be instantiated and displayed. Since `Canvas` is a subclass of `Displayable`, it can be displayed the same way any other screen using the `setCurrent()` method. [Example 5-2](#) shows the resulting MIDlet.

#### Example 5-2. Instantiating and displaying MyCanvas

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

public class MyMidlet extends MIDlet {
 public MyMidlet() { // constructor
 }

 public void startApp() {
 Canvas canvas = new MyCanvas();
 Display display = Display.getDisplay(this);
 display.setCurrent(canvas);
 }

 public void pauseApp() {
```

```

 }

 public void destroyApp(boolean unconditional) {
 }
}

```

If you run this in the Wireless Toolkit emulator, you will see something similar to [Figure 5-22](#). Note from [Example 5-1](#) that the colors are set to red and white, but since a grayscale display is being used, the colors are mapped to appropriate shades of black and white. Try switching displays to see which devices give a better feel for the colors.

**Figure 5-22. Drawing "Hello World!" on a Canvas**



### 5.4.1 Drawing Graphics

The (0,0) coordinate represents the upper left corner of the display. The numeric value of the x-coordinate increases from left to right, and the numeric value of the y-coordinate increases from top to bottom. Applications should always check the dimensions of the drawing area by using the following methods of the `Canvas` class:

```

public int getHeight();
public int getWidth();

```

These two methods return the height and width of the displayable area in pixels, respectively.

The drawing model used is called *pixel replacement*. It works by replacing the destination pixel value with the current pixel value specified in the graphics objects being used for rendering. A 24-bit color model is provided with 8 bits each for Red, Green, and Blue (RGB). However, since not all devices support color, colors requested by applications will be mapped into colors available on the devices. A well-written application, however, may check if a device supports color. This can be done using the `isColor()` and `numColors()` methods of the `Display` class, which we covered earlier in the chapter.

The `Graphics` class provides the `setColor()` and `getColor()` methods for setting and getting the color. Unlike the AWT/Swing, however, there is no `setBackground()` and `setForeground()`, so you need to explicitly call `fillRect()`, as shown in [Example 5-1](#). Most of the other methods in the `Graphics` class are self-explanatory and similar to methods in the AWT version of this class. However, let's go over a few of them here to see how they work in the J2ME environment.

### 5.4.2 Double Buffering

The double buffering technique is often used to perform smooth effect animation. In this technique, you do not draw to the display, but instead to a copy of the display (an off-screen buffer) that is maintained in memory. When you are done drawing to the buffer, you then copy the contents of the buffer to the display. The rationale here is that copying the contents of memory to the display is faster than drawing by using primitives.

To implement double buffering, first create a mutable image with the size of the screen:

```
int width = getWidth();
int height = getHeight();
Image buffer = Image.createImage(width, height);
```

Next, obtain a graphics context for the buffer:

```
Graphics gc = buffer.getGraphics();
```

Now, you can draw to the buffer:

```
// animate
// ..
gc.drawRect(20, 20, 25, 30);
```

When you need to copy the buffer to the screen, you can override the `paint( )` method to draw the buffer to the device display:

```
public void paint(Graphics g) {
 g.drawImage(buffer, 0, 0, 0);
}
```



Note that some MIDP implementations are already double-buffered, and therefore this work may not be necessary. To check if the graphics are double-buffered by an implementation, use the `Canvas.isDoubleBuffered( )` method.

### 5.4.3 Threading Issues

The MIDP GUI APIs are thread-safe. In other words, the methods can be called at any time from any thread. The only exception is the `serviceRepaints( )` method of the `Canvas` class, which immediately calls the `paint( )` method to force immediate repainting of the display. This means that if `paint( )` tries to synchronize on any object that is already locked by the application when `serviceRepaints( )` is called, the application will deadlock. To avoid deadlocks, do not lock an object that will be used by the `paint( )` method if `serviceRepaints( )` is involved.

In addition, you can use the `callSerially( )` method of the `Display` class to execute code after all pending repaints are served, as shown in the following segment of code:

```
class TestCanvas extends Canvas implements Runnable {
 void doSomething() {
```

```

 // code fragment 1
 callSerially(this);
 }

 public void run() {
 // code fragment 2
 }
}

```

Here, the object's `run( )` method will be called after the initial call.

#### 5.4.4 Fonts

Fonts cannot be created by applications. Instead, an application requests a font based on attributes (i.e., size, face, style) and the underlying implementation will attempt to return a font that closely resembles the requested font. The `Font` class represents various fonts and metrics. There are three font attributes defined in the `Font` class, and each may have different values, as follows:

##### *Face*

`MONOSPACE, PROPORTIONAL, SYSTEM`

##### *Size*

`SMALL, MEDIUM, LARGE`

##### *Style*

`BOLD, ITALIC, PLAIN, UNDERLINED`

For example, to specify a medium size font, use `Font.SIZE_MEDIUM`, and to specify an italic style, use `Font.STYLE_ITALIC`, and so on. Values for the style attributes may be combined using the OR (`|`) operator; values for the other attributes may not be combined. For example, the value of this style attribute specifies a plain, underlined font:

`STYLE_PLAIN | STYLE_UNDERLINED`

However, the following is illegal:

`SIZE_SMALL | SIZE_MEDIUM`

This is also illegal:

`FACE_SYSTEM | FACE_MONOSPACE`

Each font in the system is actually implemented individually, so in order to obtain an object representing a font, use the `getFont( )` method. This method takes three arguments for the font face, size, and style, respectively. For example, the following snippet of code obtains a `Font` object with the specified face, style, and size attributes:

```
Font font = Font.getFont(FACE_SYSTEM, STYLE_PLAIN, SIZE_MEDIUM);
```

If a matching font does not exist, the implementation will attempt to provide the closest match, which is always a valid `Font` object.

Once a font is obtained, you can use methods from the `Font` class to retrieve information about that font. For example, you can use the methods `getFace()`, `getSize()`, and `getStyle()` to retrieve information about the face, size, and style of the font, respectively.

Let's look at an example. The code in [Example 5-3](#) subclasses the `Canvas` class. In this example, the drawing color is set to white, a rectangle is drawn in the current color, then the drawing color is set to black. The rest of the code draws the system fonts on the device screen, as shown in [Figure 5-23](#).

**Figure 5-23. Drawing system fonts on the device screen**



### Example 5-3. Using fonts

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

public class FontCanvas extends Canvas {
 public void paint(Graphics g) {
 g.setColor(0xffffffff);
 g.fillRect(0, 0, getWidth(), getHeight());
 g.setColor(0x000000);

 g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN,
 Font.SIZE_LARGE));
 g.drawString("System Font", 0, 0, g.LEFT | g.TOP);
 g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN,
 Font.SIZE_MEDIUM));
 g.drawString("Medium Size", 0, 15, g.LEFT | g.TOP);
 g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
 Font.SIZE_MEDIUM));
 g.drawString("Bold Style", 0, 30, g.LEFT | g.TOP);
 g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_ITALIC,
 Font.SIZE_MEDIUM));
 g.drawString("Italic Style", 0, 45, g.LEFT | g.TOP);
 g.setFont(Font.getFont(Font.FACE_SYSTEM,
 Font.STYLE_UNDERLINED, Font.SIZE_MEDIUM));
 g.drawString("Underlined Style", 0, 60, g.LEFT | g.TOP);
 }
}
```

Now, we instantiate the `FontCanvas` class and display it, as shown in [Example 5-4](#).

### Example 5-4. Instantiating and displaying the `FontCanvas` class

```

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

public class FontMidlet extends MIDlet {
 public FontMidlet() { // constructor
 }

 public void startApp() {
 Canvas canvas = new FontCanvas();
 Display display = Display.getDisplay(this);
 display.setCurrent(canvas);
 }

 public void pauseApp() {
 }

 public void destroyApp(boolean unconditional) {
 }
}

```

### 5.4.5 Guidelines for GUI Programming for MIDP Devices

As we close this chapter, keep in mind some important guidelines when designing MIDlets with graphical API functionality:

- Be sure to make the MIDlet user interface simple and easy to use. Remember that your applications will likely be used by novice users who probably haven't used a J2ME-enabled phone and may not be familiar with its interface.
- Use the high-level API as much as possible, so that your MIDlets are portable across different handheld devices.
- If your application requires you to use the low-level API, keep to the platform-independent part of the low-level API. This means that your MIDlets should not assume any other keys than those defined in the `Canvas` class. We'll discuss this in more detail in the next chapter.
- MIDlets should never assume any specific screen size; instead, they should query the size of the display and adjust accordingly.
- Entering alphanumeric data through a handheld device can be tedious. If possible, provide a list of choices from which the user can select.

## Chapter 6. MIDP Events

In AWT and Swing, events are generated when a user interacts with an application. For example, if the user selects Save from the File menu, the application is notified of this action and responds to the generated event. The same model holds true for the MIDP. However, as mentioned in the previous chapter, there are two MIDP user interface APIs: high-level and low-level. Therefore, there are two kinds of events: high-level (such as selecting an item from a list) and low-level (such as pressing a key on the device).

This chapter discusses event handling in the MIDP and shows, through examples, how to handle high-level and low-level MIDP events generated by the components of the previous chapter. We start with an explanation of a simple application of events: navigating between screens.

## 6.1 Screen Navigation

A MIDlet developer needs to provide ways for the user to navigate through the different screens that make up the MIDlet. Because we can only show one screen at a time, however, we need to tie a mechanism to each screen that indicates to the MIDlet that the user has completed working with the current `Displayable` screen. We can do this by using the `Command` class, which is part of the `javax.microedition.lcdui` package. Let's take a closer look at the `Command` class now.

### 6.1.1 Commands

Just like a design pattern with the same name, the `Command` class encapsulates the semantic information of an action. Note that it only contains information about a command, not the actual functionality that is executed when a command is activated. Here is the constructor of the `Command` class:

```
public Command(String label, int commandType, int priority);
```

This is the only constructor for the `Command` class. Hence, creating a `Command` object is extremely simple:

```
Command infoCommand = new Command("Info", Command.SCREEN, 2);
```

The `Command` class constructor takes three parameters, and therefore contains the following three lightweight pieces of information: *label*, *command type*, and *priority*.

- The *label* is a string used for the visual representation of the command. For example, the label may appear next to a soft button on the device or as an element in a menu.
- The *command type* element specifies the command's intent. The predefined types are actually static integers in the `Command` class: `BACK`, `CANCEL`, `EXIT`, `HELP`, `ITEM`, `OK`, `SCREEN`, and `STOP`.
- The *priority* value describes the importance of this command relative to other commands on the screen. A priority value of 1 indicates the most important command; higher priority values indicate commands of lesser importance.

Each component that extends `Displayable` (such as `Screen` or `Canvas`) has the following methods available to it:

```
public void addCommand(Command c);
public void removeCommand(Command c);
```

These methods allow you to bind a command to a `Displayable` object. (That's pretty much all of them; recall [Figure 5-2](#).) When the MIDlet executes, the device assigns a visual representation of the command (typically a soft button or menu item) and chooses its placement based on the command type, placing similar commands based on their priority values. Consider the following example, where a `TextBox` object is created along with three commands. The commands are added to the `TextBox` object, and the current screen is then set to be the `TextBox` object:

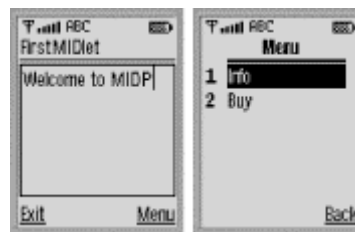
```

Display display = Display.getDisplay(this);
TextBox tb = new TextBox("MIDP", "Welcome to MIDP", 40,
 TextField.ANY);
Command exitCommand = new Command("Exit", Command.SCREEN, 1);
Command infoCommand = new Command("Info", Command.SCREEN, 2);
Command buyCommand = new Command("Buy", Command.SCREEN, 2);
tb.addCommand(exitCommand);
tb.addCommand(infoCommand);
tb.addCommand(buyCommand);
display.setCurrent(tb);

```

Look carefully at what this code displays in [Figure 6-1](#). Here, the application manager maps the `Exit` command to the screen using the soft button on the lower left, but then creates a `Menu` command to hold the `Info` and `Buy` commands. Clicking the right soft button under `Menu` takes you to a screen with a two-button menu: `Info` and `Buy`. This is because the `Info` and `Buy` commands are of lesser priority than the `Exit` command.

**Figure 6-1. Exit, Info, and Buy commands**



The general strategy that the application manager will follow is to assign as many commands with a high priority to as many soft buttons as are available. If there are not enough soft buttons, the implementation will likely group the remaining in a secondary menu that can be selected using a `Menu` soft button, as shown above. However, the exact rules for how each device handles this type of situation are implementation-dependent.

The `Command` class provides only the following three methods for retrieving the type, label, and priority values:

```

public int getCommandType();
public String getLabel();
public int getPriority();

```

Note that there is no way to reset these object properties once they are set in the constructor.



The MIDP UI API lets you set up a screen with no commands, but this is generally not useful because the user cannot move to another screen. It is important to note that the `Command` class can be used with both the high-level and the low-level APIs. Hence, commands can be placed on `Screen` objects as well as `Canvas` objects.

As we mentioned before, the command itself only contains information about a command, not the actual action that happens when a command is activated. The action is defined in a `CommandListener`, which is a *callback* object that is associated with the screen.

### 6.1.2 The `CommandListener` Interface

When a user interacts with a MIDlet, such as by selecting an item in a list or interacting with a `Gauge`, events are generated. Your application is then notified to handle these events through the use of callbacks. Callbacks are actually invocations of programmer-defined methods performed by the underlying application in response to actions taken by a user at runtime.

Callbacks are used in many programming environments, especially in GUI construction kits. For example, the AWT API makes heavy use of callbacks. When a user interacts with a component, for example, the interface code calls back the computational code to respond to the user's action. In some languages such as C/C++, callbacks are implemented by passing a function pointer to another function. The receiving function uses the function pointer to invoke another function when a particular event occurs. Because the Java programming language does not have pointers, however, callbacks are implemented with interfaces. An interface defines a set of methods, but unlike a class, it does not implement their behavior. Instead, you provide interface method implementations for the class that implements the interface.

There are four kinds of user interface callbacks in the MIDP:

- Abstract commands that are part of the high-level API
- Low-level events that represent single-key presses and releases
- Calls to the `paint()` method of a `Canvas` class
- Calls to a `Runnable` object's `run()` method, requested by a call to the `callSerially()` method of the `Display` class

Note that all user-interface callbacks are serialized by the application manager. In other words, they all occur one after another in a single thread of execution, and never at the same time. User interface callbacks are called as soon as the previous callback returns. In addition, the MIDP user interface API is thread-safe and includes a mechanism for event synchronization. An application can use the `callSerially()` method of the `Display` class to execute an operation serially with events.

Both `Screen` and `Canvas` objects can have listeners for commands that are sent when user interaction occurs. For an object to be a listener, it must implement the `CommandListener` interface. You can register a listener by using the `setCommandListener()` method, which is part of the `Displayable` class and is inherited by both `Screen` and `Canvas`. Note that there can only be one `CommandListener` object for each `Displayable` that the MIDlet has created.

```
public void setCommandListener(CommandListener c);
```

The `CommandListener` interface is for MIDlets that need to receive high-level events from the implementation. This interface has one method that a listener must implement, which is the `commandAction( )` method.

```
public void commandAction(Command c, Displayable d);
```

The first parameter is a command object that identifies the command (if any) that has been added to `Displayable` with the `addCommand( )` method and invoked. The second parameter is the `Displayable` object where the event occurred.

### 6.1.2.1 Handling simple events

Let's look at a simple example. In [Example 6-1](#), a `List` component is created and filled with the strings "Item1", "Item2", "Item3", and "Item4". The `prepare( )` method is called whenever an item is selected. The `testItem#( )` methods (where # is a number between 1 and 4) each call the `prepare( )` method and set the name of the menu.

#### Example 6-1. Handling high-level events

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

public class EventEx1 extends MIDlet implements CommandListener {

 // display manager
 Display display = null;

 // a menu with items
 List menu = null; // main menu

 // textbox
 TextBox input = null;

 // command
 static final Command backCommand = new Command("Back",
 Command.BACK, 0);
 static final Command mainMenuCommand = new Command("Main",
 Command.SCREEN, 1);
 static final Command exitCommand = new Command("Exit",
 Command.STOP, 2);

 String currentMenu = null;

 // constructor
 public EventEx1() {
 }

 /**
 * Start the MIDlet by creating a list of items and associating
 * the exit command with it.
 */

 public void startApp() throws MIDletStateChangeException {
 display = Display.getDisplay(this);
 menu = new List("Menu Items", Choice.IMPLICIT);
 menu.append("Item1", null);
```

```

 menu.append("Item2", null);
 menu.append("Item3", null);
 menu.append("Item4", null);
 menu.addCommand(exitCommand);
 menu.setCommandListener(this);

 mainMenu();
 }

 public void pauseApp() {
 display = null;
 menu = null;
 input = null;
 }

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

 // main menu
 void mainMenu() {
 display.setCurrent(menu);
 currentMenu = "Main";
 }

 /**
 * a generic method that will be called when any of
 * the items on the list are selected.
 */

 public void prepare() {
 input = new TextBox("Enter some text: ", "", 5,
 TextField.ANY);
 input.addCommand(backCommand);
 input.setCommandListener(this);
 input.setString("");
 display.setCurrent(input);
 }

 /**
 * Test item1.
 */
 public void testItem1() {
 prepare();
 currentMenu = "item1";
 }

 /**
 * Test item2.
 */
 public void testItem2() {
 prepare();
 currentMenu = "item2";
 }

 /**
 * Test item3.
 */
 public void testItem3() {
 prepare();
 }

```

```

 currentMenu = "item3";
 }

 /**
 * Test item4.
 */
 public void testItem4() {
 prepare();
 currentMenu = "item4";
 }

 /**
 * Handle events.
 */
 public void commandAction(Command c, Displayable d) {

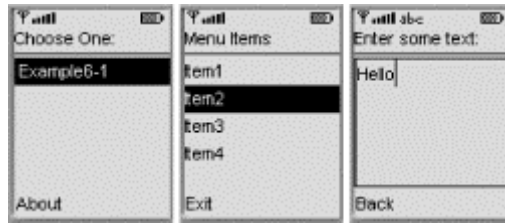
 String label = c.getLabel();

 if (label.equals("Exit")) {
 destroyApp(true);
 } else if (label.equals("Back")) {
 if(currentMenu.equals("item1") ||
 currentMenu.equals("item2") ||
 currentMenu.equals("item3") ||
 currentMenu.equals("item4")) {
 // go back to menu
 mainMenu();
 }
 } else {
 List down = (List)display.getCurrent();
 switch(down.getSelectedIndex()) {
 case 0: testItem1();break;
 case 1: testItem2();break;
 case 2: testItem3();break;
 case 3: testItem4();break;
 }
 }
 }
}

```

The `EventEx1` class implements the `CommandListener` interface by providing an implementation for the `commandAction( )` method. In this implementation, the label of the command passed into the callback method is checked. If the label equals `Exit`, the MIDlet is destroyed. If the label equals `Back` and the current menu is `"Item1"`, `"Item2"`, `"Item3"`, or `"Item4"`, the program goes back to the main menu. Otherwise, the selected item is found and the appropriate method is called. Note that when you have an item list, you can use the `Display.getCurrent( )` method to return the list, and then switch between the items to determine which item is selected. If you run the `EventEx1` MIDlet, you should see output similar to [Figure 6-2](#).

**Figure 6-2. Handling high-level events**



### 6.1.2.2 Creating GUI components and handling events

Now let's look at another example that demonstrates how to create various GUI components and to handle their events. The MIDlet in this example allows you to test lists, forms, choices, gauges, text fields, and text boxes. The `EventEx2` MIDlet shown in [Example 6-2](#) makes use of the following classes, listed in alphabetical order, from the `javax.microedition.lcdui` package: `Alert`, `AlertType`, `Command`, `DateField`, `Display`, `Displayable`, `Form`, `Gauge`, `List`, `TextBox`, `TextField`, and `Ticker`, as well as the `CommandListener` interface.

#### Example 6-2. Constructing and testing GUI components

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

public class EventEx2 extends MIDlet implements CommandListener {
 // display manager
 Display display = null;

 // a menu with items
 List menu = null; // main menu

 // list of choices
 List choose = null;

 // textbox
 TextBox input = null;

 // ticker
 Ticker ticker = new Ticker("Test GUI Components");

 // alerts
 final Alert soundAlert = new Alert("sound Alert");

 // date
 DateField date = new DateField("Today's date: ",
 DateField.DATE);

 // form
 Form form = new Form("Form for Stuff");

 // gauge
 Gauge gauge = new Gauge("Gauge Label", true, 10, 0);

 // text field
 TextField textfield = new TextField("TextField Label", "abc",
 50, 0);

 // command
 static final Command backCommand = new Command("Back",
```

```

 Command.BACK, 0);
static final Command mainMenuCommand = new Command("Main",
 Command.SCREEN, 1);
static final Command exitCommand = new Command("Exit",
 Command.STOP, 2);
String currentMenu = null;

// constructor.
public EventEx2() {
}

/**
 * Start the MIDlet by creating a list of items and associating
 * the exit command with it.
 */

public void startApp() throws MIDletStateChangeException {
 display = Display.getDisplay(this);
 menu = new List("Test Components", Choice.IMPLICIT);
 menu.append("Test TextBox", null);
 menu.append("Test List", null);
 menu.append("Test Alert", null);
 menu.append("Test Date", null);
 menu.append("Test Form", null);
 menu.addCommand(exitCommand);
 menu.setCommandListener(this);
 menu.setTicker(ticker);

 mainMenu();
}

public void pauseApp() {
 display = null;
 choose = null;
 menu = null;
 ticker = null;
 form = null;
 input = null;
 gauge = null;
 textfield = null;
}

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

// main menu
void mainMenu() {
 display.setCurrent(menu);
 currentMenu = "Main";
}

/**
 * Test the TextBox component.
 */
public void testTextBox() {
 input = new TextBox("Enter Some Text:", "", 5,
 TextField.ANY);
 input.setTicker(new Ticker("testTextBox"));
 input.addCommand(backCommand);
 input.setCommandListener(this);
}

```

```

 input.setString("");
 display.setCurrent(input);
 currentMenu = "input";
 }

 /**
 * Test the List component.
 */
 public void testList() {
 choose = new List("Choose Items", Choice.MULTIPLE);
 choose.setTicker(new Ticker("listTest"));
 choose.addCommand(backCommand);
 choose.setCommandListener(this);
 choose.append("Item 1", null);
 choose.append("Item 2", null);
 choose.append("Item 3", null);
 display.setCurrent(choose);
 currentMenu = "list";
 }

 /**
 * Test the Alert component.
 */
 public void testAlert() {
 soundAlert.setType(AlertType.ERROR);
 soundAlert.setString("** ERROR **");
 display.setCurrent(soundAlert);
 }

 /**
 * Test the DateField component.
 */
 public void testDate() {
 java.util.Date now = new java.util.Date();
 date.setDate(now);
 Form f = new Form("Today's date");
 f.append(date);
 f.addCommand(backCommand);
 f.setCommandListener(this);
 display.setCurrent(f);
 currentMenu = "date";
 }

 /**
 * Test the Form component.
 */
 public void testForm() {
 form.append(gauge);
 form.append(textfield);
 form.addCommand(backCommand);
 form.setCommandListener(this);
 display.setCurrent(form);
 currentMenu = "form";
 }

 /**
 * Handle events.
 */
 public void commandAction(Command c, Displayable d) {
 String label = c.getLabel();
 if (label.equals("Exit")) {

```

```

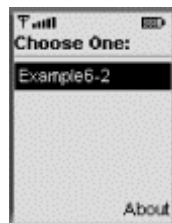
 destroyApp(true);
 } else if (label.equals("Back")) {
 if(currentMenu.equals("list") ||
 currentMenu.equals("input") ||
 currentMenu.equals("date") ||
 currentMenu.equals("form")) {
 // go back to menu
 mainMenu();
 }
 } else {
 List down = (List)display.getCurrent();
 switch(down.getSelectedIndex()) {
 case 0: testTextBox();break;
 case 1: testList();break;
 case 2: testAlert();break;
 case 3: testDate();break;
 case 4: testForm();break;
 }
 }
}
}
}

```

To test the `EventEx2` MIDlet using the J2ME Wireless Toolkit, do the following:

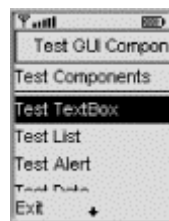
1. Create a new project (call it [Example 6-2](#)) and a MIDlet class (call it `EventEx2`), copy the code to the appropriate location, and compile it.
2. Run the `EventEx2` MIDlet in the emulator.
3. You should see the name of the project ([Example 6-2](#)) in the application manager, as shown in [Figure 6-3](#).

**Figure 6-3. Project Example 6-2 MIDlet**



4. Activate the MIDlet.
5. As the MIDlet runs, you see a menu with the following options: Test TextBox, Test List, Test Alert, Test Date, and Test Form, as shown in [Figure 6-4](#).

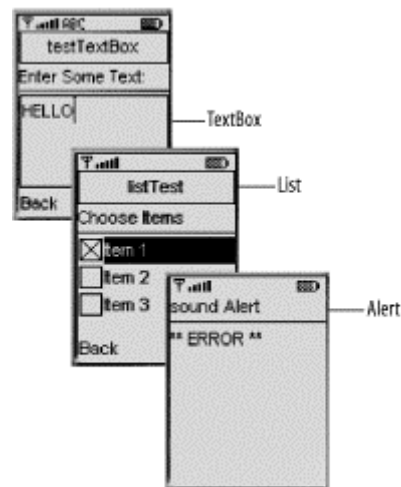
**Figure 6-4. EventEx2 MIDlet**



6. Choose a test to perform.

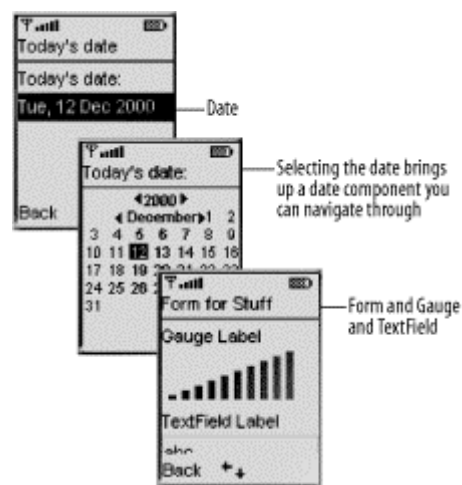
Tests from the `EventEx2` MIDlet are shown in [Figure 6-5](#).

**Figure 6-5. The TextBox, List, and Alert tests**



If you have a soundcard, you will hear a warning sound with the alert. The remaining tests from the `EventEx2` MIDlet are shown in [Figure 6-2](#).

**Figure 6-6. DateField, Calendar, and Form, with Gauge and TextField tests**



### 6.1.3 The `ItemStateListener` Interface

Applications use the `ItemStateListener` interface to receive events that indicate changes in the internal state of items within a `Form` screen. Items within a `Form` screen may be changed when the user performs any of the following actions:

- Adjusts the value of an interactive `Gauge`.
- Enters or modifies the values of a `TextField`.
- Enters a new date or time in a `DateField`.
- Changes the set of selected values in a `ChoiceGroup`.

This interface has only one method that a listener must implement:

```
public void itemStateChanged(Item item);
```

You can use the `setItemStateListener( )` method of the `Form` class to register a listener for these conditions, as shown in the next section.

### 6.1.3.1 Changing the date

In [Example 6-3](#), a `DateField` object is created and added to a form. When you click on the date, you can change it by navigating through the calendar. When the date is changed, a message appears.

#### Example 6-3. Implementing the `ItemStateListener` interface

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

public class EventEx3 extends MIDlet {
 Display display;

 public EventEx3() {
 display = Display.getDisplay(this);
 }

 public void destroyApp (boolean unconditional) {
 notifyDestroyed();
 System.out.println("App destroyed ");
 }

 public void pauseApp () {
 display = null;
 System.out.println("App paused.");
 }

 public void startApp () {
 Form form = new Form("Change Date");

 ItemStateListener listener = new ItemStateListener() {
 java.util.Calendar cal = java.util.Calendar.
 getInstance(java.util.TimeZone.getDefault());

 public void itemStateChanged(Item item) {
 cal.setTime(((DateField)item).getDate());
 System.out.println("\nDate has changed");
 }
 };

 // register for events
 form.setItemStateListener(listener);

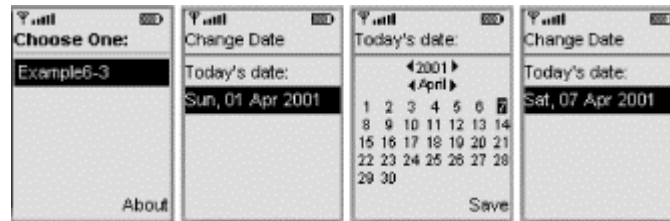
 // get today's date
 java.util.Date now = new java.util.Date();
 DateField dateItem = new DateField("Today's date:",
 DateField.DATE);
 dateItem.setDate(now);

 // add date to the Form screen
 form.append(dateItem);
 display.setCurrent(form);
 }
}
```

```
}
```

Here, the `ItemStateListener` interface is implemented as an anonymous inner class by providing an implementation to the `itemStateChanged()` method. If you run the `EventEx3` MIDlet, you should see output similar to [Figure 6-7](#).

**Figure 6-7. Implementing the `ItemStateListener` interface**



## 6.2 Handling Low-Level Events

If you use the `Canvas` class to write applications to access low-level input events or to issue graphics calls for drawing to the display, you must handle low-level events. Game applications are likely to use the `Canvas` class because it provides methods to handle game actions and key events. The key events are reported with respect to keycodes that are directly bound to concrete keys on the device.

The `Canvas` class, which is a subclass of `Displayable`, allows the application to register a listener for commands, but it requires applications to subclass the listener first. Also, while screens allow the application to define a listener and register it with an instance of the `Screen` class, the `Canvas` class does not allow this, because several listener interfaces need to be created, one for each kind of event.

### 6.2.1 Key Events

Every key for which events are reported is assigned a keycode. The MIDP defines the following keycodes in the `Canvas` class:

`KEY_NUM0`

The keycode for key 0

`KEY_NUM1`

The keycode for key 1

`KEY_NUM2`

The keycode for key 2

`KEY_NUM3`

The keycode for key 3

KEY\_NUM4

The keycode for key 4

KEY\_NUM5

The keycode for key 5

KEY\_NUM6

The keycode for key 6

KEY\_NUM7

The keycode for key 7

KEY\_NUM8

The keycode for key 8

KEY\_NUM9

The keycode for key 9

KEY\_STAR

The keycode for the star key "\*"

KEY\_POUND

The keycode for the pound key "#"

As you probably guessed, these are the keys 0..9, \*, and #. Other keys might exist on some devices, but for portability, applications should use only the standard keycodes. The `getKeyName()` method is used to retrieve an informative string for a key.

### 6.2.2 Game Actions

If your application needs arrow keys and gaming-related events, use game actions instead of keycodes. `Canvas` defines the following constants as well:

UP

A constant for the `UP` game action

DOWN

A constant for the `DOWN` game action

LEFT

A constant for the `LEFT` game action

`RIGHT`

A constant for the `RIGHT` game action

`FIRE`

A constant for the `FIRE` game action

`GAME_A`

A constant for the general purpose "A" game action

`GAME_B`

A constant for the general purpose "B" game action

`GAME_C`

A constant for the general purpose "C" game action

`GAME_D`

A constant for the general purpose "D" game action

While each keycode is mapped to one game action, a game action can be associated with more than one keycode. The translation between the two is done with the `getKeyCode()` and `getGameAction()` methods.



If your application uses game actions and you want the application to be portable, you should translate key events into game actions with the `getGameAction()` method and test the result. For example, the game actions `UP`, `DOWN`, `LEFT`, and `RIGHT` can be mapped differently on different devices. The `getGameAction()` method returns the `RIGHT` game action, for example, when the user presses the key that is a natural right on the device.

### 6.2.3 Event Delivery Methods

The following methods of the `Canvas` class are available for handling low-level events:

```
protected void keyPressed(int keyCode);
protected void keyReleased(int keyCode);
protected void keyRepeated(int keyCode);
protected void pointerPressed(int x, int y);
protected void pointerDragged(int x, int y);
protected void pointerReleased(int x, int y);
```