Wireless Java: Developing with J2ME, Second Edition

JONATHAN KNUDSEN

Apress™

Wireless Java: Developing with J2ME, Second Edition

ISBN (pbk): 1-59059-077-5
Printed and bound in the United States of America 12345678910

Technical Reviewer: Michael Yuan
Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski
Managing and Production Editor: Grace Wong
Copy Editor: Rebecca Rider
Proofreader: Gregory Teague
Compositor: Diana Van Winkle, Van Winkle Design Group
Indexer: Valerie Perry
Artist and Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.
For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 5

Creating a User Interface

MIDP APPLICATIONS ARE built to run on many different devices without modification. This is particularly difficult in the area of the user interface because devices have screens of all sizes, in grayscale and in color. Furthermore, devices vary widely in their input capabilities, from numeric keypads to alphabetic keyboards, soft keys, and even touch screens. The minimum screen size mandated by MIDP is 96×54 pixels, with at least one bit of color depth.[1] As for input, MIDP is fairly open-ended: devices are expected to have some type of keyboard, or a touch screen, or possibly both.

Given the wide variety of devices that are compliant with MIDP, there are two ways to create applications that work well on all devices:

- *Abstraction*: Specify a user interface in abstract terms, relying on the MIDP implementation to create something concrete. Instead of saying something like, "Display the word 'Next' on the screen above the soft button," you say, "Give me a **Next** command somewhere in this interface."

- *Discovery*: The application learns about the device at runtime and tailors the user interface programmatically. You might, for example, find out how big the device's screen was in order to scale your user interface appropriately.

The MIDP APIs support both methods. Abstraction is the preferred method because it involves less code in your application and more work by the MIDP implementation. In some cases, like games, you need to be more specific about the user interface; these types of applications will discover the capabilities of a device and attempt to tailor their behavior appropriately. MIDP's user-interface APIs are designed so that it's easy to mix the two techniques in the same application.

---

1. Color depth is the number of bits that determine the color of a pixel on the screen. One bit allows for two colors (usually black and white). Four bits allows for 16 colors, which could be different levels of gray or a palette of other colors.

The View from the Top

MIDP contains user-interface classes in the `javax.microedition.lcdui` and `javax.microedition.lcdui.game` packages. The device's display is represented by an instance of the `Display` class, accessed from a factory method, `getDisplay()`. `Display`'s main purpose in life is to keep track of what is currently shown, which is an instance of `Displayable`. If you think of `Display` as an easel, a `Displayable` instance is akin to a canvas on that easel.

MIDlets can change the contents of the display by passing `Displayable` instances to `Display`'s `setCurrent()` method. This is the basic function of a typical MIDlet:

1. Show a `Displayable`.
2. Wait for input.
3. Decide what `Displayable` should be next.
4. Repeat.

`Displayable` has a small family of subclasses that represent various types of user interfaces. Figure 5-1 shows the lineage.

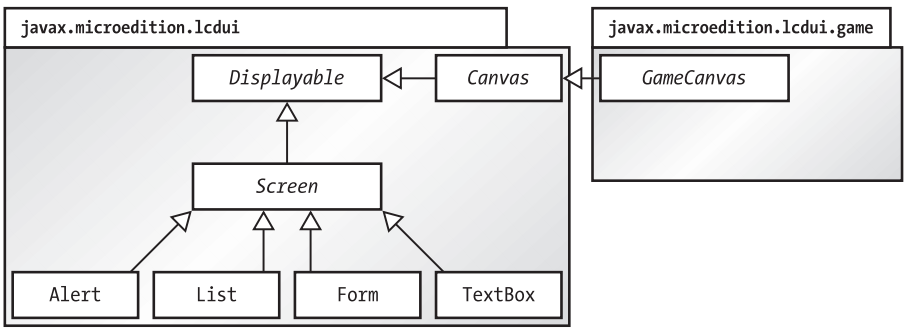


Figure 5-1. Displayables in the `javax.microedition.lcdui` *and* `javax.microedition.lcdui.game` *package*

`Displayable`'s progeny are split between two branches that correspond to the two methods for creating generalized user interfaces, abstraction and discovery. The `Screen` class represents displays that are specified in abstract terms.

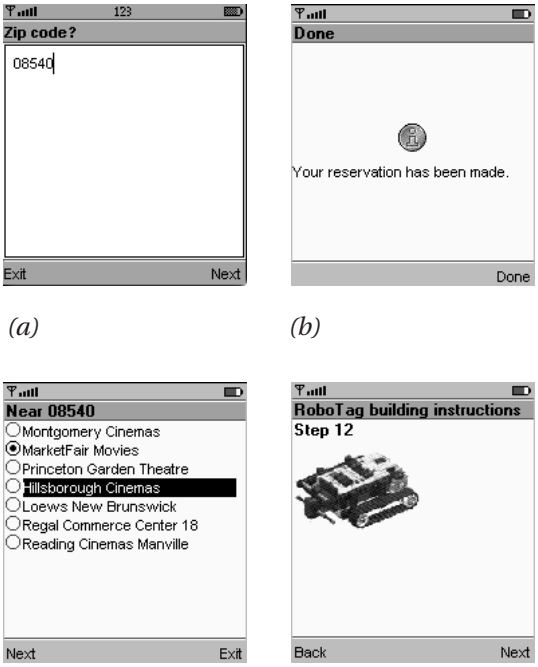


Figure 5-2. The four children of `Screen`*: (a)* `TextBox`*, (b)* `Alert`*, (c)* `List`*, and (d)* `Form`

These screens contain standard user-interface items like combo boxes, lists, menus, and buttons. Four subclasses provide a wide range of functionality, as illustrated in Figure 5-2.

The remainder of this chapter is devoted to explaining the simplest of these four classes: `TextBox` and `Alert`. The next chapter explores the more flexible `List` and `Form`.

For particularly demanding or idiosyncratic displays, you'll have to create a subclass of `Canvas`. Your MIDlet will assume responsibility for most of the drawing, but you get much finer control over what is shown and how user input is handled. `Canvas` supplies methods that allow your MIDlet to learn about its environment—the size of the display, for example, and which kinds of events are supported by the device. User interfaces built on `Canvas` discover the attributes of a device and attempt to create something that looks reasonable. Chapter 10 explains `Canvas`-based user interfaces in detail.

`GameCanvas`, new in MIDP 2.0, provides user interface functionality specifically for game displays. Chapter 11 explains this new API.

Using Display

Display manages a device's screen. You can get a reference to the device's display by supplying a MIDlet reference to the static getDisplay() method. Typically, you'll do this in the startApp() method of a MIDlet:

```
public void startApp() {
  Display d = Display.getDisplay(this);
  // ...
}
```

You may be tempted to call getDisplay() in a MIDlet's constructor, but according to the specification, getDisplay() can only be called after the beginning of the MIDlet's startApp() method.

Once you've got a reference to a device's Display, you'll just need to create something to show (an instance of Displayable) and pass it to one of Display's setCurrent() methods:

```
public void setCurrent(Displayable next)
public void setCurrent(Alert alert, Displayable nextDisplayable)
```

The second version is used when you want to show a temporary message (an Alert) followed by something else. I'll talk more about Alerts at the end of this chapter.

Display's getCurrent() method returns a reference to what's currently being shown. Note that a MIDlet may return a valid object from getCurrent() even if it is not visible to the user. This could happen on a device running multiple MIDlets simultaneously, for example. Note that the Displayable interface has a method called isShown() that indicates whether the given object is actually being shown on the device screen.

You can also query a Display to determine its capabilities, which is helpful for applications that need to adapt themselves to different types of displays. The numColors() method returns the number of distinct colors supported by this device, while the isColor() method tells whether the device supports color or grayscale. A Display for a device supporting 16 levels of gray, for example, would return false from isColor() and 16 from numColors(). In MIDP 2.0, you can also find out whether the device supports transparency by calling numAlphaLevels(), which returns the number of transparency levels. The minimum return value is two, indicating that image pixels with full transparency and full opacity are supported. Return values greater than two indicate that alpha blending is supported. In MIDP 2.0, Display contains two additional pairs of methods. The first methods, getColor() and getBorderStyle(), are used for finding out colors and line styles

from the system user interface scheme. These methods are useful for drawing custom items, a topic that is covered in Chapter 7. The other method pair, `flashBacklight()` and `vibrate()`, invoke the corresponding features of the device. These are more fully discussed in Chapter 11.

Event Handling with Commands

`Displayable`, the parent of all screen displays, supports a very flexible user interface concept, the command. A *command* is something the user can invoke—you can think of it as a button. Like a button, it has a title, like "OK" or "Cancel," and your application can respond appropriately when the user invokes the command. The premise is that you want a command to be available to the user, but you don't really care how it is shown on the screen or exactly how the user invokes it—keypad button, soft button, touch screen, whatever.

Every `Displayable` keeps a list of its `Commands`. You can add and remove `Commands` using the following methods:

```
public void addCommand(Command cmd)
public void removeCommand(Command cmd)
```

Creating Commands

In MIDP, commands are represented by instances of the `Command` class. To create a `Command`, just supply a name, a type, and a priority. The name is usually shown on the screen. The type can be used to signify a commonly used command. It should be one of the values defined in the `Command` class. Table 5-1 shows the type values and their meanings.

Table 5-1. Command Types

NAME	MEANING
OK	Confirms a selection.
CANCEL	Cancels pending changes.
BACK	Moves the user back to a previous screen.
STOP	Stops a running operation.
HELP	Shows application instructions.
SCREEN	Generic type for specific application commands.

To create a standard **OK** command, for example, you would do this:

```
Command c = new Command("OK", Command.OK, 0);
```

To create a command specific to your application, you might do this:

```
Command c = new Command("Launch", Command.SCREEN, 0);
```

It's up to the MIDP implementation to figure out how to show the commands. In the Sun J2ME Wireless Toolkit emulator, commands are assigned to the two soft buttons. A *soft button* is a button on the device keypad with no predefined function. A soft button can serve a different purpose at different times. If there are more commands than there are soft buttons, the commands that don't fit will be grouped into a menu that is assigned to one of the soft buttons.

A simple priority scheme determines who wins when there are more commands than available screen space. Every command has a priority that indicates how hard the display system should try to show the command. Lower numbers indicate a higher priority. If you add a command with priority 0, then several more with priority 1, the priority 0 command will show up on the screen directly. The other commands will most likely end up in a secondary menu.

MIDP 2.0 adds support for long labels on commands. The MIDP implementation decides which label it will use based on the available screen space and the size of the labels. You can create a command with a short and long label like this:

```
Command c = new Command("Run", "Run simulation", Command.SCREEN, 0);
```

The `Command` class provides `getLabel()`, `getLongLabel()`, and `getCommandType()` methods for retrieving information about commands.

Responding to Commands

By themselves, `Commands` aren't very exciting. They'll show up on the screen, but nothing happens automatically when a user invokes a command. An object called a *listener* is notified when the user invokes any command in a `Displayable`. This follows the basic form of the JavaBeans event model; a `Displayable` is a *unicast event source*. A `Displayable` fires off an event every time the user invokes one of its `Commands`.

The listener is an object that implements the `CommandListener` interface. To register the listener with a `Displayable`, use the following method:

```
public void setListener(CommandListener l)
```

`Displayable` is a unicast event source because it can only have one listener object. (*Multicast* event sources can have multiple listeners and use an `add…` method for adding listeners rather than a `set…` method.)

Implementing a `CommandListener` is a matter of defining a single method:

```
public void commandAction(Command c, Displayable s)
```

When a command is invoked, the `Displayable` that contains it calls the `commandAction()` method of the registered listener.

TIP *Event listeners should not perform lengthy processing inside the event-handling thread. The system uses its own thread to call* `commandAction()` *in response to user input. If your implementation of* `commandAction()` *does any heavy thinking, it will tie up the system's event-handling thread. If you have anything complicated to do, use your own thread.*

A Simple Example

By way of illustration, consider the following class:

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

public class Commander extends MIDlet {
  public void startApp() {
    Displayable d = new TextBox("TextBox", "Commander", 20, TextField.ANY);
    Command c = new Command("Exit", Command.EXIT, 0);
    d.addCommand(c);
    d.setCommandListener(new CommandListener() {
      public void commandAction(Command c, Displayable s) {
        notifyDestroyed();
      }
    } );

    Display.getDisplay(this).setCurrent(d);
  }

  public void pauseApp() { }

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

This MIDlet creates a `TextBox`, which is a kind of `Displayable`, and adds a single command to it. The listener is created as an anonymous inner subclass. In Sun's toolkit, this MIDlet appears as shown in Figure 5-3.

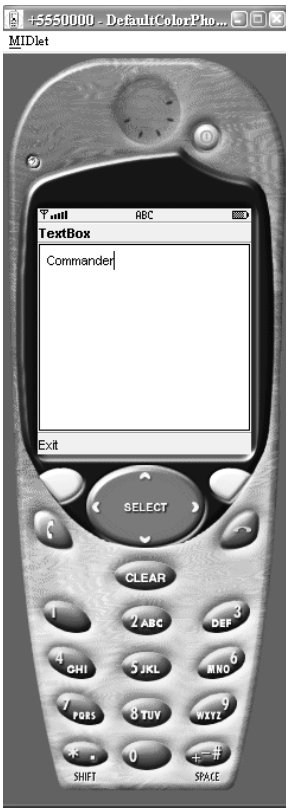
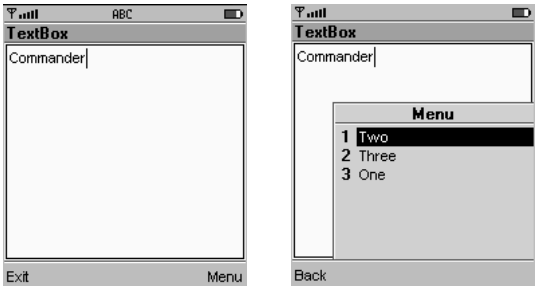


Figure 5-3. A simple MIDlet with a single command, **Exit**

Figure 5-3 shows the **Exit** command being mapped to one of the MIDP simulator's soft buttons. If you add another command to this MIDlet, it will be mapped to the other soft button. If you continue adding commands, the ones that don't fit on the screen will be put into an off-screen menu. For example, a screen with four commands shows up in the MIDP simulator as illustrated in Figure 5-4a.

If you press the soft button for **Menu**, you'll see the remainder of the commands as shown in Figure 5-4b. Menu items can now be selected by pressing a number or using the arrow keys for navigation. In the example shown in Figure 5-4, the **Exit** command is given a higher priority (lower number) than the other commands, which insures that it appears directly on the screen. The other commands, with a lower priority, are relegated to the command menu.



(a) *(b)*

Figure 5-4. This MIDlet has more commands than the device has soft buttons. Invoking the (a) system-generated **Menu** *command brings up the (b) remaining commands.*

Screens and Tickers

The remainder of this chapter and all of Chapter 6 are devoted to `Screen` and its subclasses, which is the left branch of the hierarchy shown in Figure 5-1. `Screen` is the base class for all classes that represent generalized user interfaces.

`Canvas`, by contrast, is a base class for specialized interfaces, such as those for games. `Canvas` will be fully covered later, in Chapter 10.

In the coming sections, we'll explore each of `Screen`'s child classes. Here, I'll briefly describe what all `Screens` have in common: a title and a ticker. The *title* is just what you expect: a string that appears at the top of the screen. A *ticker* is simply a bit of text that scrolls across the top of a `Screen`; it is named after old-fashioned stock tickers.

In MIDP 2.0, the four methods I'm about to describe are moved from `Screen` to `Displayable`. Thus, MIDP 2.0 extends the concept of title and ticker to all `Displayables`, not just `Screens`. In MIDP 2.0, the `Screen` class has no methods.

The title is a text string displayed at the top of the screen. As you saw in Figure 5-3, the title of the screen is "TextBox." Subclasses of `Screen` have constructors that set the title, but the title may also be accessed using the following methods:

```
public void setTitle(String newTitle)
public String getTitle()
```

The ticker is just as easy to access:

```
public void setTicker(Ticker newTicker)
public Ticker getTicker()
```

The `Ticker` class is a simple wrapper for a string. To add a ticker to a screen, you would do something like this:

```
// Displayable d = ...
Ticker ticker = new Ticker("This is the ticker message!");
d.setTicker(ticker);
```

Figure 5-5 shows a ticker in action.

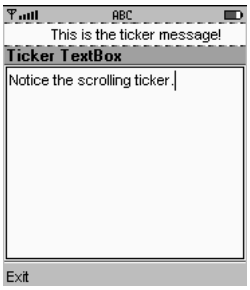


Figure 5-5. A ticker scrolls across the top of a screen.

TextBox, the Simplest Screen

The simplest type of screen is the `TextBox`, which you've already seen in action. `TextBox` allows the user to enter a string. Keep in mind that on a garden-variety MIDP device, text input is a tedious process. Many devices only have a numeric keypad, so entering a single character is a matter of one, two, or three button presses. A good MIDlet requires minimal user input.

That said, your MIDlet may need some kind of input—perhaps a zip code, or a short name, or some kind of password. In these cases, you'll probably want to use a `TextBox`.

A `TextBox` is created by specifying four parameters:

```
public TextBox(String title, String text, int maxSize, int constraints)
```

The title is used as the screen title, while `text` and `maxSize` determine the initial text and maximum size of the text box. Finally, `constraints` can be used to restrict the user's input. Constants from the `TextField` class are used to specify the type of input required:

- `ANY` allows any type of input.
- `NUMERIC` restricts the input to integers.

- `DECIMAL` (new in MIDP 2.0) allows numbers with fractional parts.
- `PHONENUMBER` requires a telephone number.
- `EMAILADDR` input must be an e-mail address.
- `URL` input must be a web address.

It's up to the implementation to determine how these constraints are enforced. The toolkit emulators simply don't allow invalid input; for example, a `NUMERIC` `TextBox` doesn't allow you to enter alphabetic characters.

The constraints above may be combined with the flags listed below. Constraints limit the behavior of users, while flags define the behavior of the `TextBox`. All the flags except `PASSWORD` are new in MIDP 2.0.

- `PASSWORD` characters are not shown when entered; generally, they are represented by asterisks.
- `UNEDITABLE` indicates text that cannot be edited.
- `SENSITIVE` is used to flag text that the implementation should not store. Some input schemes store input from the user for later use in autocompletion. This flag indicates that the text is off-limits and should not be saved or cached.
- `NON_PREDICTIVE` indicates that you are expecting the user to enter text that any text-predicting input scheme will probably not be able to guess. For example, if you're expecting the user to enter an order number like Z51002S, you would use this flag to tell the input scheme to not bother trying to predict the input.
- `INITIAL_CAPS_WORD` is used for input where each word should be capitalized.
- `INITIAL_CAPS_SENTENCE` indicates input where the first character of each sentence should be capitalized.

If you don't want the `TextBox` to perform any validation, use `ANY` or its numerical equivalent, 0, for the `constraints` parameter in the constructor.

The flags may be combined with any of the other constraints using the OR operator. For example, to create a `TextBox` that constrains input to an e-mail address but keeps the entered data hidden, you would do something like this:

```
Displayable d = new TextBox("Email", "", 64,
        TextField.EMAILADDR | TextField.PASSWORD);
```

If you think about it, though, `PASSWORD` is probably more trouble than it's worth. The point of `PASSWORD` fields, at least on desktop machines, is to keep someone walking past your computer screen from seeing your secret password. For every character you enter, the password field shows an asterisk or some other symbol. As you type your secret password, all that shows up on the screen is a line of asterisks. On mobile phones and other small devices, this is less of a concern because the screens are smaller and much more difficult to read than a typical desktop monitor.

Furthermore, the difficulty of entering data on a small device means that it will be hard to correctly enter passwords if you are typing blind. Mobile phones, for example, typically require you to press keys several times to enter a single letter. On Sun's toolkit emulator, pressing the '7' key twice enters the letter 'Q.' On a real device, you would have to enter a password "gandalf" with the following sequence of key presses: 4, 2, 6, 6, 3, 2, 5, 5, 5, 3, 3, 3. Without visual feedback, it would be extremely easy to make a mistake when entering a password. ("Did I press the 5 key two times or three times?") The J2ME Wireless Toolkit emulator shows the current character but previously typed characters are shown as asterisks. Good passwords typically have mixed case, numbers, and possibly punctuation; these would be hard to enter correctly.

In MIDP 2.0 applications, password fields (whether or not they use the `PASSWORD` flag) should be protected with the `SENSITIVE` flag so that the password doesn't show up in any system dictionaries or pop up unexpectedly when the user is entering other text.

MIDP 2.0 includes a single new method in the `TextBox` class called `setInitialInputMode(String characterSubset)`. This method is used to suggest to the implementation what input mode would be best suited to the expected text. You can only suggest the input mode, and you have no way of knowing whether the implementation has honored the request. The string passed to the method can be one of the constants from the J2SE `java.lang.Character.UnicodeBlock` class, prepended with "UCB_". For example, you might pass "UCB_BASIC_LATIN" or "UCB_KATAKANA" to this method. You can also use input subsets defined by `java.awt.im.InputSubset` by prepending them with "IS_". For example, "IS_LATIN" or "IS_KANJI" would be valid. Finally, MIDP 2.0 also defines the character subsets "MIDP_UPPERCASE_LATIN" and "MIDP_LOWERCASE_LATIN".

The input mode is complementary to the text constraints and flags. You might specify `ANY` for the constraints, then call `setInitialInputMode("MIDP_LOWERCASE_LATIN")` to request that the implementation begin by allowing lowercase input. This doesn't prevent the user from changing the input mode, it just starts things off on the right foot.

Using Alerts

An *alert* is an informative message shown to the user. In the MIDP universe, there are two flavors of alert:

- A *timed* alert is shown for a certain amount of time, typically just a few seconds. It displays an informative message that does not need to be acknowledged, like "Your transaction is complete," or "I can't do that right now, Dave."

- A *modal* alert stays up until the user dismisses it. Modal alerts are useful when you need to offer the user a choice of actions. You might display a message like "Are you ready to book these tickets?" and offer **Yes** and **No** commands as options.

MIDP alerts can have an associated icon, like a stop sign or question mark. Alerts may even have an associated sound, although this depends on the implementation. MIDP alerts are very much the same concept as modal dialogs in windowing systems like MacOS and Windows. Figure 5-6 shows a typical `Alert`.

Figure 5-6. Alerts are similar to modal dialogs in a desktop windowing system.

Alerts are represented by instances of the javax.microedition.lcdui.Alert class, which offers the following constructors:

```
public Alert()
public Alert(String title, String alertText, Image alertImage, AlertType alertType)
```

Any or all of the parameters in the second constructor may be null. (Don't worry about the Image class right now; I'll discuss it in the next chapter in the section on Lists.)

By default, timed Alerts are created using a default timeout value; you can find out the default value by calling getDefaultTimeout(). To change the Alert's timeout, call setTimeout() with the timeout value in milliseconds. A special value, FOREVER, may be used to indicate that the Alert is modal.

You could create a simple timed Alert with the following code:

```
Alert alert = new Alert("Sorry", "I'm sorry, Dave...", null, null);
```

To explicitly set the timeout value to five seconds, you could do this:

```
alert.setTimeout(5000);
```

If, instead, you wanted a modal alert, you would use the special value FOREVER:

```
alert.setTimeout(Alert.FOREVER);
```

The MIDP implementation will automatically supply a way to dismiss a modal alert. Sun's reference implementation, for example, provides a **Done** command mapped to a soft button. MIDP 2.0 exposes this command as the static member DISMISS_COMMAND, allowing you to register your own command listener and explicitly recognize this command. You can add your own commands to an Alert using the usual addCommand() method. The first time you call addCommand(), the system's dismiss command is removed.

The default behavior for Alerts automatically advances to the next screen when the Alert is dismissed or times out. You can specify the next screen by passing it and the Alert to the two-argument setCurrent() method in Display. If you call the regular one-argument setCurrent() method, the previous screen is restored when the Alert is dismissed. Alert types serve as hints to the underlying MIDP implementation. The implementation may use the alert type to decide what kind of sound to play when the alert is shown. The AlertType class provides five types, accessed as static member variables: ALARM, CONFIRMATION, ERROR, INFO, and WARNING.

MIDP 2.0 adds an indicator to an `Alert`. By default, no indicator is present, but you can add one by passing a `Gauge` to `Alert`'s `setIndicator()` method. (`Gauge` is presented in the next chapter in the section on `Forms`.) The indicator is handy for showing progress in a network connection or a long computation.

The following example, `TwoAlerts`, shows both types of alert. It features a main `TextBox` that is displayed when the MIDlet begins. Two commands, **Go** and **About**, provide access to the alerts. The **Go** command shows a timed alert that contains a message about a fictitious network error. The **About** command displays a modal alert that could contain copyright information. A third command, **Exit**, provides a way to exit the MIDlet. Keep in mind that all three commands may not fit on the screen; some of them may be accessible from a secondary menu.

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

public class TwoAlerts
    extends MIDlet
    implements CommandListener {
  private Display mDisplay;

  private TextBox mTextBox;
  private Alert mTimedAlert;
  private Alert mModalAlert;

  private Command mAboutCommand, mGoCommand, mExitCommand;

  public TwoAlerts() {
    mAboutCommand = new Command("About", Command.SCREEN, 1);
    mGoCommand = new Command("Go", Command.SCREEN, 1);
    mExitCommand = new Command("Exit", Command.EXIT, 2);

    mTextBox = new TextBox("TwoAlerts", "", 32, TextField.ANY);
    mTextBox.addCommand(mAboutCommand);
    mTextBox.addCommand(mGoCommand);
    mTextBox.addCommand(mExitCommand);
    mTextBox.setCommandListener(this);

    mTimedAlert = new Alert("Network error",
        "A network error occurred. Please try again.",
        null,
        AlertType.INFO);
```

```
    mModalAlert = new Alert("About TwoAlerts",
        "TwoAlerts is a simple MIDlet that demonstrates the use of Alerts.",
        null,
        AlertType.INFO);
    mModalAlert.setTimeout(Alert.FOREVER);
  }

  public void startApp() {
    mDisplay = Display.getDisplay(this);

    mDisplay.setCurrent(mTextBox);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {}

  public void commandAction(Command c, Displayable s) {
    if (c == mAboutCommand)
      mDisplay.setCurrent(mModalAlert);
    else if (c == mGoCommand)
      mDisplay.setCurrent(mTimedAlert, mTextBox);
    else if (c == mExitCommand)
      notifyDestroyed();
  }
}
```

Summary

MIDP's main user-interface classes are based on abstractions that can be adapted to devices that have different display and input capabilities. Several varieties of prepackaged screen classes make it easy to create a user interface. Screens have a title and an optional ticker. Most importantly, screens can contain `Command`s, which the implementation makes available to the user. Your application can respond to commands by acting as a listener object. This chapter described `TextBox`, a screen for accepting user input, and `Alert`, a simple screen for displaying information. In the next chapter, we'll get into the more complex `List` and `Form` classes.