J2ME 101, Part 1: Introduction to MIDP's high-level user interface

Skill Level: Introductory

John Muchow
Author

20 Nov 2003

This is the first installment of a comprehensive four-part introduction to Java 2 Micro Edition (J2ME) and the Mobile Information Device Profile (MIDP). The series will consist of two tutorials and two companion articles. In this first tutorial, you will learn about the essential components of J2ME, with a primary focus on MIDP. The focus is on MIDP's high-level user interface, with a step-by-step introduction to the components that facilitate the main interaction between the user and the device display.

Section 1. Getting started

What is this tutorial about?

This is first installment in a four-part comprehensive introduction -- two tutorials and two articles -- to Java 2 Micro Edition (J2ME) and the Mobile Information Device Profile (MIDP). In this tutorial you will learn about the essential components of J2ME, with a primary focus on MIDP. We'll focus here on MIDP's high-level user interface, with a step-by-step introduction to the components that facilitate the main interaction between the user and the device display. These components include `TextField`, `Gauge`, and `DateField`, among others. You will also learn about event-handling techniques for the high-level API, using both `Command` and `Item` objects.

In the second part of the tutorial you will learn how to work with MIDP's low-level user interface. We'll walk through exercises that include creating and drawing on a canvas, working with fonts, and drawing shapes. As part of this discussion you will become familiar with all the J2ME components that work directly with the device display.

Two companion articles round out this introduction to J2ME. The first article will

introduce you to the Record Management System (RMS), which is the persistent storage environment within MIDP. The second article will cover networking support for J2ME. You'll learn about the Generic Connection Framework, opening network connections and communicating with remote systems, talking with Java Servlets, and a few other goodies.

Portions of this tutorial are used with permission from the book Core J2ME Technology and MIDP *by John W. Muchow, published by Sun Microsystems Press and Prentice Hall. Copyright 2002 Sun Microsystems Inc.*

Should I take this tutorial?

This tutorial is intended for experienced Java programmers who would like to learn how to develop mobile applications using J2ME. The code examples are not particularly complex, but it is assumed that you understand how classes are created, inherited, and instantiated within the Java platform. You will also benefit from a working knowledge of using and creating Java Archive Files (JARs). See Resources for more information on using JARs.

Software and installation requirements

To complete this tutorial you will need to install JDK version 1.4 or greater, along with the J2ME Wireless Toolkit (WTK). The WTK download contains an IDE for creating Java applications, commonly called *MIDlets*, as well as the libraries required for creating them.

The WTK is contained within a single executable file. Run this file to begin the installation. It is recommended that you use the default installation directory. If you choose another directory, make sure that the path you select does *not* include any spaces.

Section 2. Introduction to J2ME and MIDP

J2ME and MIDP overview

This section is an overview of many of the essential components that will let you create and run applications on J2ME, focusing specifically on the Mobile Information Device Profile (MIDP). Each of the components discussed in this section will be utilized in the tutorial. Here you will learn the basics of how each one is put together and interacts with the other technologies or components to aid in the development of mobile device applications on the Java platform.

Components of J2ME

Given the size and complexity of J2SE, there was a need to create a Java implementation targeted at devices with limited memory and/or processing power. J2ME is intended to fill this gap, providing a platform and development language familiar to millions of programmers.

Even among "micro" devices there are significant variations in capability. For example, a typical personal digital assistant (PDA) has a much larger screen, more memory, and a faster processor than a mobile phone. To support these differences, J2ME introduced the concept of configurations and profiles. Basically, a *configuration* outlines a set of device requirements (memory and connectivity, among others). A *profile* is the API that sits on top of a given configuration, providing the specific features and capabilities for a range of devices.

In this tutorial we will work with the Connected Limited Device Configuration (CLDC) and the MIDP. We'll begin by learning more about configurations and profiles, as well as the other foundational components of J2ME, in the next few panels.

Configurations

A J2ME *configuration* defines a Java platform for a range of devices. Each configuration encompasses the features available in the Java language as well as the core libraries. A J2ME application can be built to meet the requirements of one of two device configurations: the Connected Device Configuration (CDC) or the Connected, Limited Device Configuration (CLDC). A device that implements the CDC has the following characteristics:

- 512 kilobytes (minimum) memory for running Java programs
- 256 kilobytes (minimum) for run time memory allocation
- Network connectivity, possibly persistent, and high-bandwidth

And here are the typical characteristics of a device that implements the CLDC:

- 128 KB of memory for running Java programs
- 32 KB of memory for run time memory allocation
- A limited user interface
- Runs on battery power
- Wireless network connection, low bandwidth

All of the example applications we develop for this tutorial will be for devices of the CLDC type.

Profiles

A J2ME *profile* is an extension of a configuration. As noted earlier, it defines the libraries available to a developer writing applications for a specific device type.

The MIDP extends the CLDC. MIDP defines APIs for user interface components, input and event handling, persistent storage, and networking and timers -- all with consideration for the screen and memory limitations of mobile devices within the CLDC configuration.

We will use MIDP to develop all of the example applications used in this tutorial.

MIDP's high-level and low-level APIs

MIDP offers a high-level API and a low-level API for user interface development. We will focus on the high-level API in this first half of the tutorial. MIDP's high-level interface is used to build common user-interface components such as `Form` s, `TextBox` es, and `Gauge` s, among others. The high-level API handles most component functionality for you, such as drawing each component on the screen.

The low-level API offers more flexibility than the high-level one, but is also more demanding to work with. The low-level interface and its components will be introduced in the second part of this tutorial.

MIDlets

A Java application that is built on top of the CLDC and MIDP is known as a *MIDlet*. A *MIDlet suite* consists of one or more MIDlets packaged together as a JAR. The MIDlets we develop together in this tutorial will help you to learn about the various high-level components of J2ME and MIDP.

The application manager

The application manager is the software on a mobile device that is responsible for installing, running, and removing MIDlets. We'll run each of the applications we develop together in this tutorial, giving you the opportunity to learn firsthand how the application manager works.

J2ME and MIDP summary

In this section you learned about the components that will let you create and run applications on the J2ME platform. J2ME is a subset of the Java platform designed specifically for the development of mobile device applications. Applications built on the J2ME platform are developed for a particular device configuration and device

profile. In this tutorial we will work with example programs developed for the CLDC and its extension, MIDP.

Section 3. Event handling

Event handling overview

In this section you will learn about event handling for MIDP's high-level interface. High-level MIDP events are divided into two categories: `Command` and `Item` events. In the simplest sense, `Command` events are triggered by keypresses on the device, whereas `Item` events are the result of visual components changing on the display. In this section you'll learn how both event types are processed.

Command objects

When an event occurs on a mobile device, a `Command` object holds information about that event. This information includes the type of command executed, the label of the command, and its priority. In J2ME, commands are commonly represented with soft-buttons on the device. Figure 1 shows two `Command` objects, one with the label "Exit" and one with label "View."

Figure 1. Command objects and soft-buttons

If there are too many commands to be shown on the display, a device will create a
menu to hold multiple commands. Figure 2 shows how this might look.

Figure 2. Command objects and a menu

Event processing with Command objects

The only MIDP components that can manage commands are `Form`, `TextBox`, `List`, and `Canvas`. You'll learn about each of these components as we progress through the tutorial.

The basic steps to process events with a `Command` object are as follows:

1. Create a `Command` object.

2. Add the `Command` to a `Form`, `TextBox`, `List`, or `Canvas`.

3. Create a listener.

Upon detection of an event, the listener will generate a call to the method `commandAction()`. Within this method you can determine which command generated the event and process it accordingly, as shown in the next panel.

Command processing example

The following code block shows an event processing procedure using a `Command` object.

```
private Form fmMain;                // Form
private Command cmExit;             // Command to exit the MIDlet
      ...
fmMain = new Form("Core J2ME");    // Create Form and give it a title

// Create Command object, with label, type and priority
cmExit = new Command("Exit", Command.EXIT, 1);
...
fmMain.addCommand(cmExit);         // Add Command to Form
fmMain.setCommandListener(this);   // Listen for Form events


...

public void commandAction(Command c, Displayable s)
{
  if (c == cmExit)
  {
    destroyApp(true);
    notifyDestroyed();
  }
}
```

Item objects

It is also possible to process events using `Item` objects. Several `Item` s are predefined in MIDP for processing particular event types. For example, the `DateField` item allows the user to select the date and time that will display on the screen, and the `TextField` item allows a user to enter a series of alpha-numeric and special characters.

Event processing with Item objects

With the exception of `StringItem`, `Spacer`, and `ImageItem`, each `Item` has the ability to recognize events. As with a `Command` object, you must first create a listener before events will be acknowledged. When a change occurs on a given item -- for example, the text of a `TextField` component is updated -- an event is generated.

When a change occurs in any `Item` component, the method `itemStateChanged()` will be called. Within this method you can determine which `Item` generated the event.

A note about device implementations

Device implementations may differ in their handling of `Item` events. MIDP does not specify when an `Item` event must be acknowledged, so `itemStateChange()` may not automatically be called for every event on an `Item`. For example, if the year value of a `DateField` component is updated, the method `itemStateChange()` may not be called the moment the change is detected. The device may not acknowledge this event until the user moves to another component on the display.

Item processing example

The following code block shows simple event processing for a `DateFieldItem` object.

```
private Form fmMain;                  // Form
private DateField dfToday;            // DateField item
...
fmMain = new Form("Core J2ME");                      // Create Form object
dfToday = new DateField("Today:", DateField.DATE);   // Create DateField
...
fmMain.append(dfToday);              // Add DateField to Form
fmMain.setItemStateListener(this);   // Listen for Form events

...

public void itemStateChanged(Item item)
{
  // If the datefield initiated this event
  if (item == dfToday)
    ...
}
```

A MIDlet for event processing

Our first MIDlet will demonstrate both `Command` and `Item` event processing. For each MIDlet we develop in this tutorial, we'll follow the same typical J2ME development procedure, as follows:

1. Create the project.

2. Write the source code.

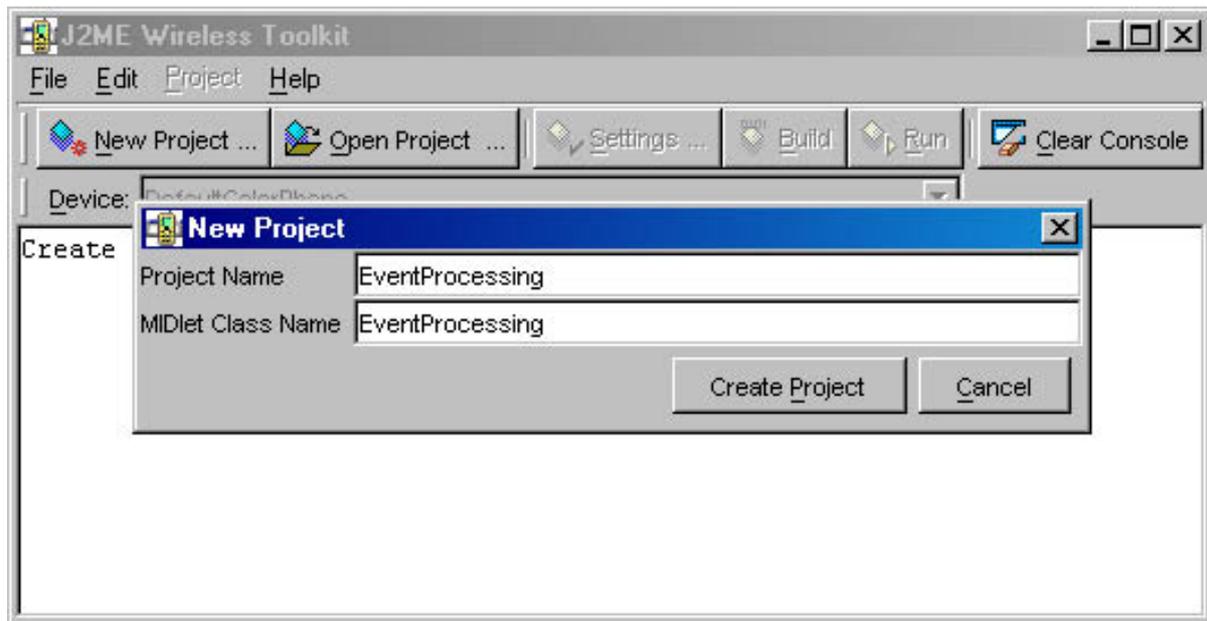3. Compile and preverify the code.

4. Run the MIDlet.

Now, let's get to it.

Create the project

We begin by creating a new project in the J2ME Wireless Toolkit, as follows:

1. Click **New Project**.

2. Enter the project name and MIDlet class name, as shown in Figure 3.

3. Click **Create Project** to complete this step.

Figure 3. Creating the EventProcessing project

Write the code

Next, you should copy and paste the following code into a text editor.

```
/*--------------------------------------------------
* EventProcessing.java
*---------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class EventProcessing extends MIDlet implements
ItemStateListener, CommandListener
{
    private Display display;     // Reference to Display object for this MIDlet
    private Form fmMain;         // Main Form
    private Command cmExit;      // Command to exit the MIDlet
    private TextField tfText;    // TextField component (item)

    public EventProcessing()
    {
      display = Display.getDisplay(this);

      // Create the date and populate with current date
      tfText = new TextField("First Name:", "", 10, TextField.ANY);

      cmExit = new Command("Exit", Command.EXIT, 1);

      // Create the Form, add Command and DateField
      // listen for events from Command and DateField
      fmMain = new Form("Event Processing Example");
      fmMain.addCommand(cmExit);
      fmMain.append(tfText);
      fmMain.setCommandListener(this);     // Capture Command events (cmExit)
      fmMain.setItemStateListener(this);   // Capture Item events (dfDate)
    }

    // Called by application manager to start the MIDlet.
    public void startApp()
    {
      display.setCurrent(fmMain);
    }
```

```
      public void pauseApp()
      { }

      public void destroyApp(boolean unconditional)
      { }

      public void commandAction(Command c, Displayable s)
      {
        System.out.println("Inside commandAction()");

        if (c == cmExit)
        {
          destroyApp(false);
          notifyDestroyed();
        }
      }

      public void itemStateChanged(Item item)
      {
        System.out.println("Inside itemStateChanged()");
      }
}
```

Save, compile, and preverify

When you create a new project, the WTK builds the proper directory structure for you. So, in this example, the WTK has created the *C:\WTK20\apps\EventProcessing* directory and all the necessary subdirectories. Save your Java source file as EventProcessing.java in the *src* directory, as shown in Figure 4. (Note that the drive and WTK directory will vary depending on where you have installed the toolkit.)
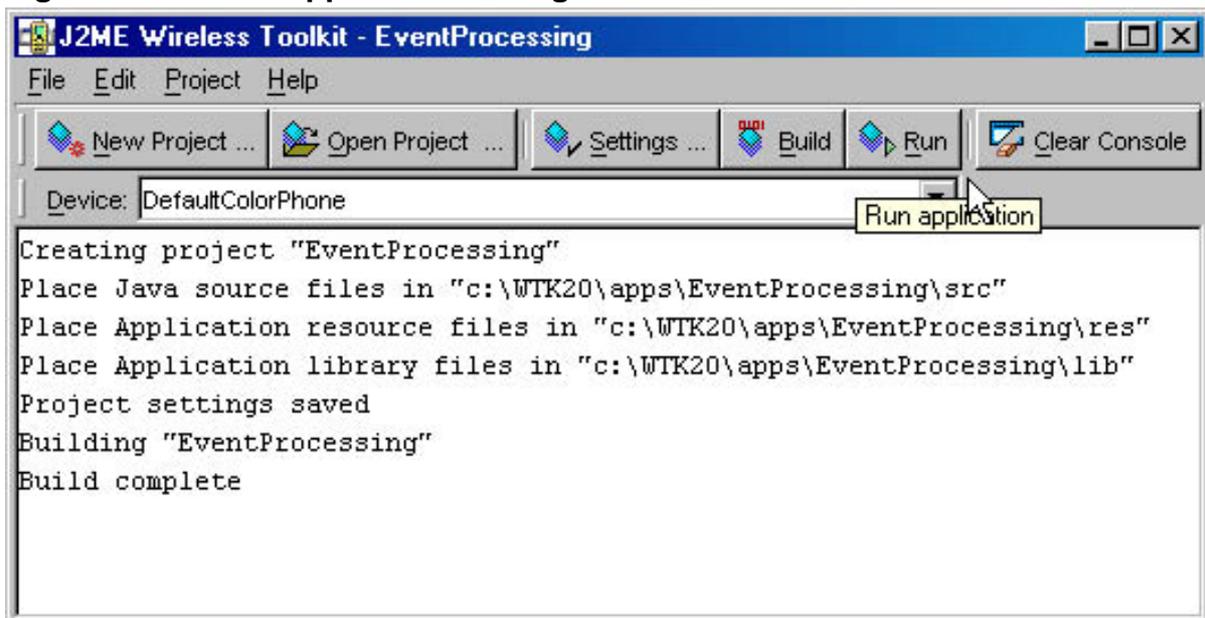
Figure 4. Save the EventProcessing code
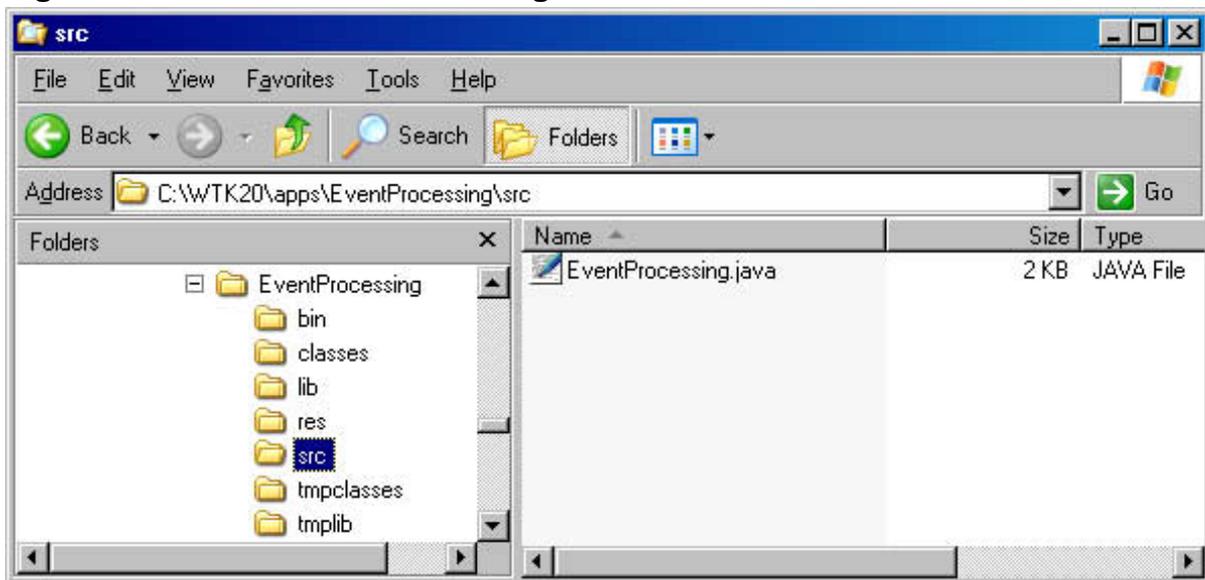
Next, click **Build** to compile, preverify, and package the MIDlet, as shown in Figure 5.

Figure 5. Build the EventProcessing project
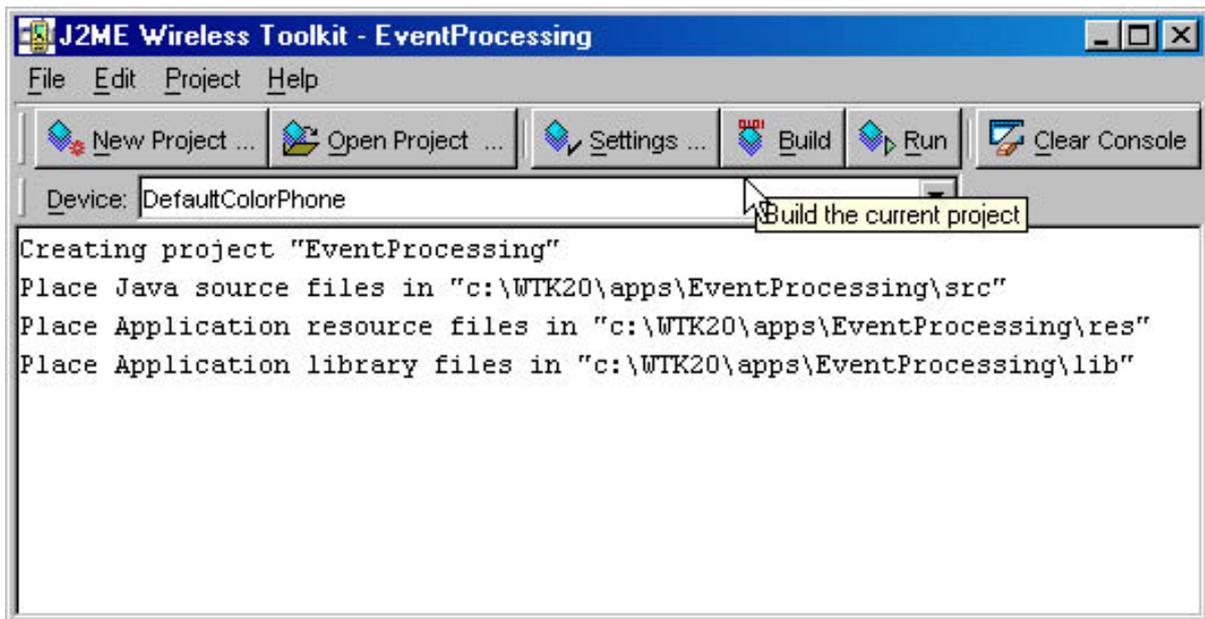
And finally, click **Run** to start the application manager.

**Figure 6. Start the application manager**

A note about preverification

If you come from a background of writing code with J2SE, then preverification may not be familiar to you. Given the memory limitations on mobile devices, preverification was devised to limit the amount of overhead to run applications on a mobile device. Rather than verifying code solely at run time, MIDlets go through a preverification step during the development cycle, which reduces the resource and memory consumption necessary when starting a MIDlet.

Start the MIDlet

To start the EventProcessing MIDlet, click the **Launch** soft-button as shown in the left screenshot of Figure 7.

Figure 7. Run the EventProcessing MIDlet
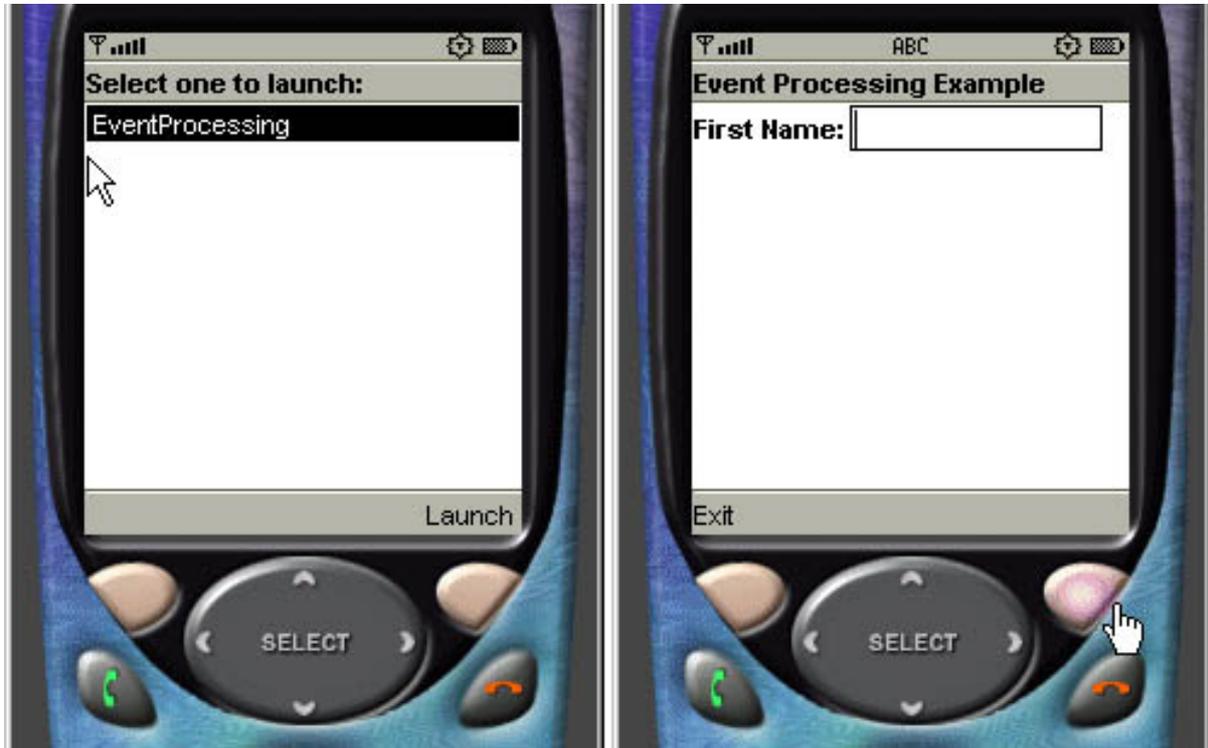
After the MIDlet is running, the `Form` and `TextBox` will be displayed as shown in the right-hand screenshot of Figure 7.

Detecting Item events

Figure 8 shows a string of characters entered into a `TextBox`.

Figure 8. Item events

As each character is input, an `Item` event is generated. Looking back at the code in the `itemStateChanged()` method, we print a message to the WTK console for each event, as shown below:

```
public void itemStateChanged(Item item)
{
  System.out.println("Inside itemStateChanged()");
}
```

Figure 9 shows the console output for an `Item` event.

**Figure 9. Console output for an Item event**

```
J2ME Wireless Toolkit - EventProcessing                    _ □ X
File   Edit   Project   Help                         ▷

  🔷 New Project ...  │  📂 Open Project  ...  │  🔷 Settings ...  │  💠 Build  │  🔷 Run  │  📝 Clear Console

  Device: DefaultColorPhone                                         ▼

Inside itemStateChanged()
Inside itemStateChanged()
Inside itemStateChanged()
Inside itemStateChanged()
```
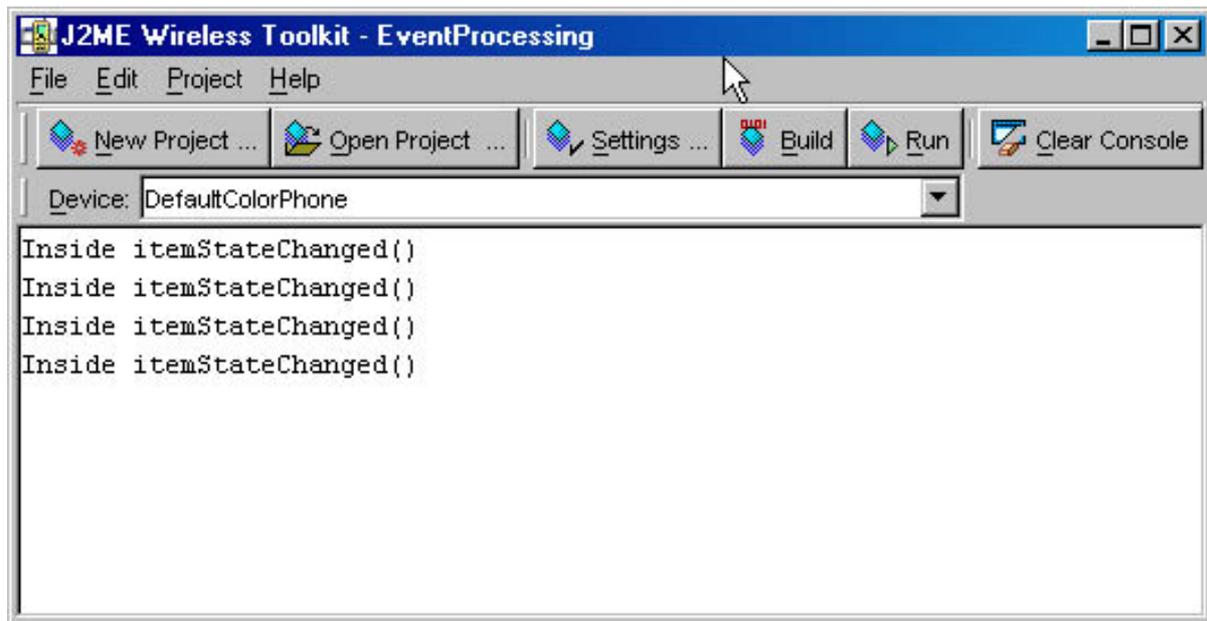
Detecting Command events

When we choose to exit the MIDlet by pressing the soft-button with the **Exit** label, a
Command event will be generated. Here's the code for processing Command s:

```
public void commandAction(Command c, Displayable s)
{
  System.out.println("Inside commandAction()");

  if (c == cmExit)
  {
    destroyApp(false);
    notifyDestroyed();
  }
}
```

For each Command event, the commandAction() method is called. In Figure 10 we
can see the console message printed when the Exit command is selected.

Figure 10. Console output for a Command event
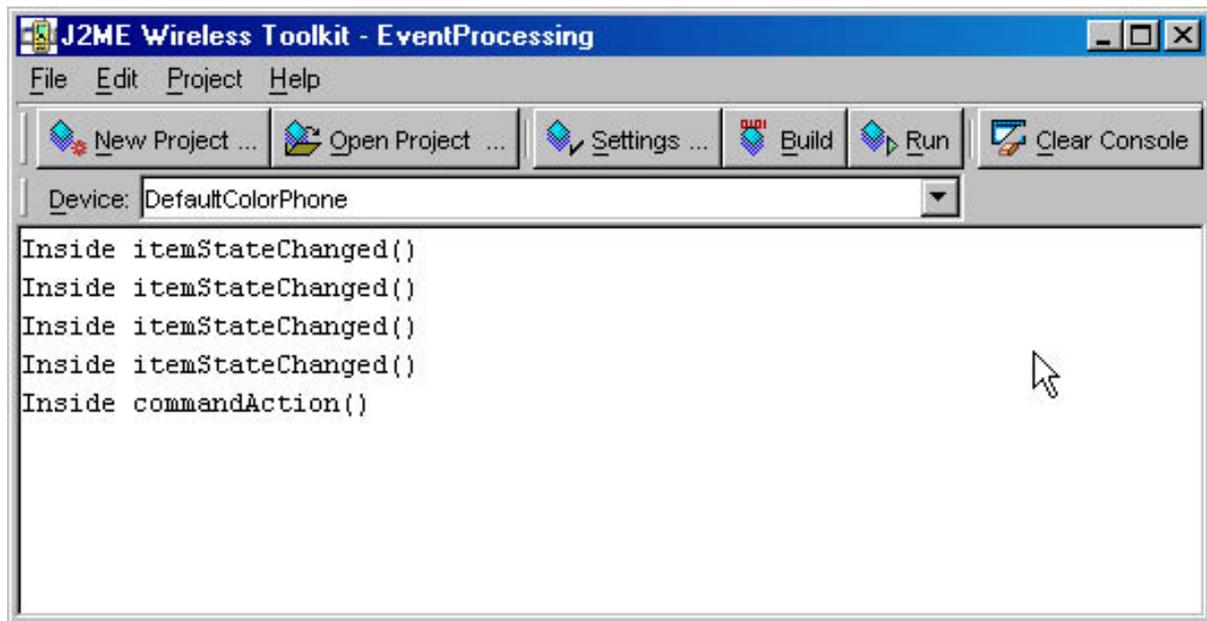
Event handling summary

In this section you've learned about the two event handling techniques for MIDP's high-level API. You've also learned the procedure we will use to create and run every example MIDlet used throughout the remainder of this tutorial. This concludes the discussion of event handling for the first part of the tutorial, although you will learn about event handling for the low-level interface in Part 2.

Section 4. Display, Displayable, and Screens objects

Display, displayable, and screens objects overview

Before looking at each of the high-level components available within MIDP you should be familiar with the objects designed to work with the device display. In this section, you will be introduced to the `Display`, `Displayable`, and `Screen` components, which together comprise MIDP's device display mechanism.

Display object

A MIDlet has one instance of a `Display` object. This object is used to obtain information about the current display -- such as the color support available -- and includes methods for requesting that objects (that is, `Form` s and `Textbox` es) be displayed. The `Display` object is essentially the manager of the device display,

controlling what is shown on the device.

Displayable object

Although there is only one `Display` object per MIDlet, many objects within a MIDlet may be *displayable* -- that is, `Form`s, `TextBox`es, `ChoiceGroup`s, etc.

A `Displayable` object is a component that is visible on a device. MIDP contains two subclasses of `Displayable`: `Screen` and `Canvas`. Following are the class definitions for each:

```
abstract public class Displayable
public abstract class Canvas extends Displayable
public abstract class Screen extends Displayable
```

Figure 11 illustrates the device display hierarchy up to this point.

Figure 11. The current display hierarchy
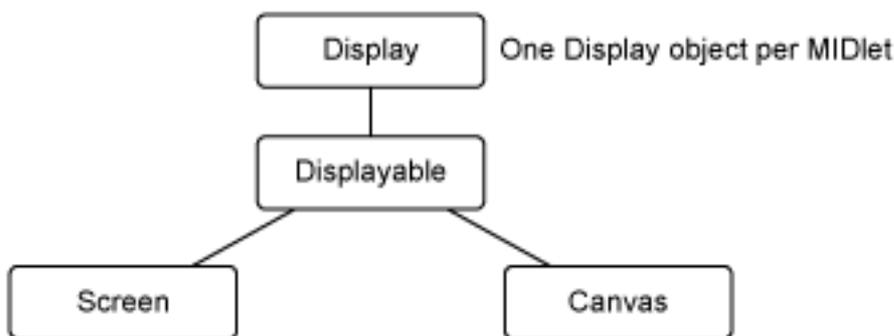
Screen object

A `Screen` object is not something that is visible on the device. Rather, `Screen` is subclassed by high-level components, which the end-user then interacts with on the display.

Figure 12 shows the components that subclass `Screen`. Each has its own unique look and feel and provides a comprehensive set of methods.

Figure 12. Components that subclass Screen
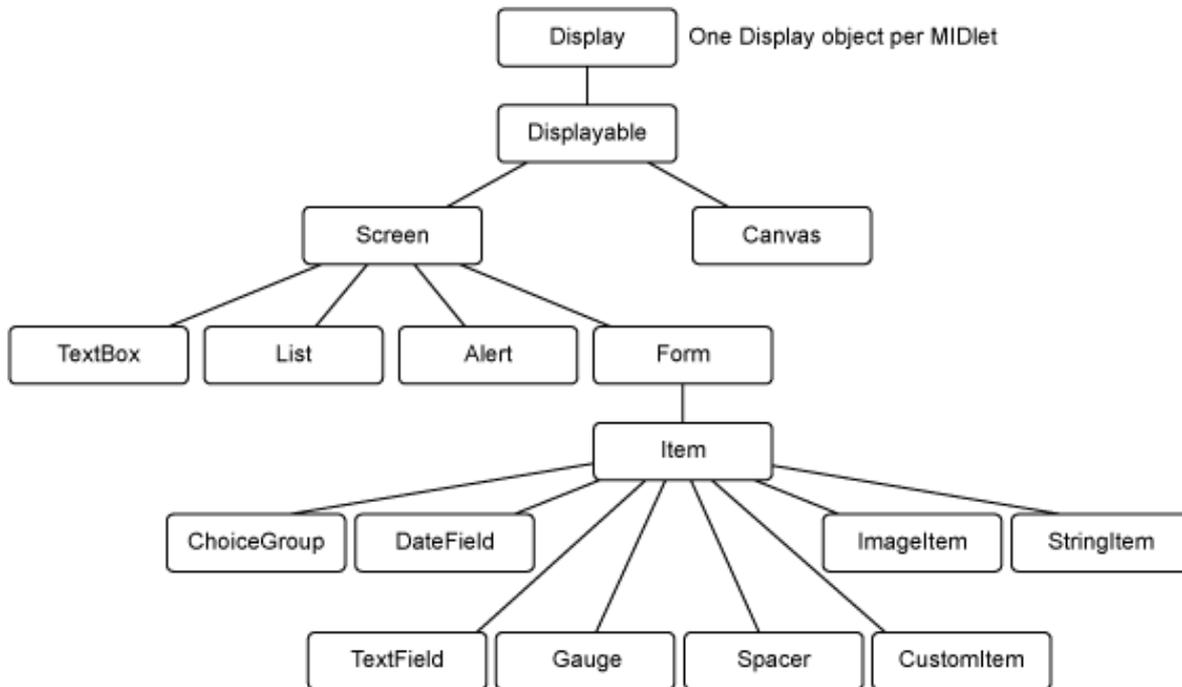
Object hierarchy summary

This section has introduced the object hierarchy for working with the user interface components in MIDP. Each MIDlet consists of one `Display` object and any number of `Displayable` objects. The two subclasses of `Displayable` are `Screen` and `Canvas`. `Screen` supports high-level user interface components, whereas canvas supports low-level ones. You'll learn about `Canvas` in Part 2 of this tutorial.

Section 5. Form and Item components

Form and item components overview

This section will introduce the components that may be shown on a `Form`. A `Form` is essentially a container to hold other components, where each component is a subclass of the `Item` class.
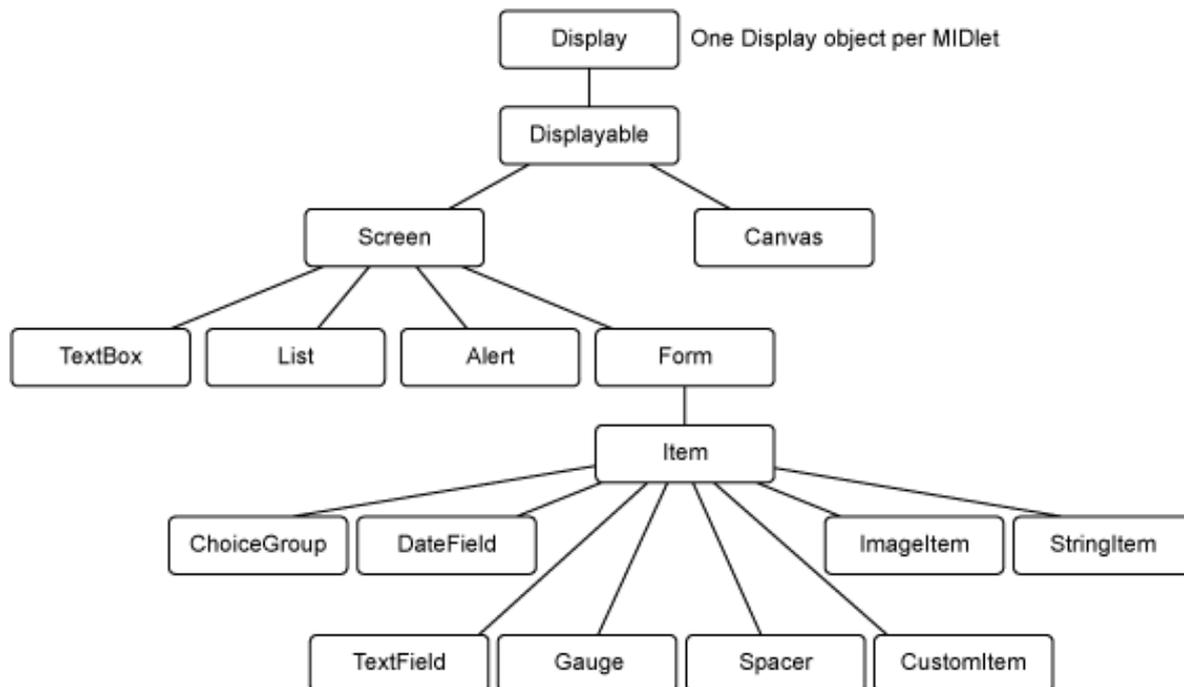
In the sections that follow, we'll look at each of the device-display components that comprise MIDP's high-level API, namely:

- `DateField`

- `Gauge`

- `StringItem`

- `TextField`

- `ChoiceGroup`

- `Spacer`

- `CustomItem`

- `Image` and `ImageItem`

The complete device display component hierarchy is shown in Figure 13.

Figure 13. The complete device display hierarchy

The DateField component

The `DateField` component provides a means to visually manipulate a `Date` object, as defined in `java.util.Date`. When creating a `DateField` object you specify whether the user can edit the date, the time, or both.

The constructors for `DateField` are as follows:

```
DateField(String label, int mode)
DateField(String label, int mode, TimeZone timeZone)
```

Below is a partial code listing for creating a `DateField` object and populating the object with the current date and time.

```
private DateField dfAlarm;

// DateField with label, that allows both date and time to be changed
dfAlarm = new DateField("Set Alarm Time", DateField.DATE_TIME);
dfAlarm.setDate(new Date());
```

The DateFieldTest MIDlet

We'll create a simple DateFieldTest MIDlet to demonstrate the `DateField` component. As you will recall, the steps to create and run the MIDlet are as follows:

1.   Create a new project with the name **DateFieldTest**.

2.   Copy and paste the DateFieldTest source code into a text editor.

3.   Save the source code as DateFieldTest.java in the directory *\apps\DateFieldTest\src* of your WTK installation.

4.   Build and run the project.

DateFieldTest source

Here's the source code for our example DateFieldTest MIDlet:

```
/*-------------------------------------------------
 * DateFieldTest.java
 *-------------------------------------------------*/
import java.util.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.Timer;
import java.util.TimerTask;

public class DateFieldTest extends MIDlet implements ItemStateListener, CommandListener
{
  private Display display;         // Reference to display object
  private Form fmMain;              // Main form
  private Command cmExit;           // Exit MIDlet
  private DateField dfAlarm;        // DateField component

  public DateFieldTest()
  {
    display = Display.getDisplay(this);

    // The main form
    fmMain = new Form("DateField Test");

    // DateField with todays date as a default
    dfAlarm = new DateField("Set Alarm Time", DateField.DATE_TIME);
    dfAlarm.setDate(new Date());

    // All the commands/buttons
    cmExit = new Command("Exit", Command.EXIT, 1);

    // Add to form and listen for events
    fmMain.append(dfAlarm);
    fmMain.addCommand(cmExit);
    fmMain.setCommandListener(this);
```

```
      fmMain.setItemStateListener(this);
   }

  public void startApp ()
  {
    display.setCurrent(fmMain);
  }

  public void pauseApp()
  { }

  public void destroyApp(boolean unconditional)
  { }

  public void itemStateChanged(Item item)
  {
    System.out.println("Date field changed.");
  }

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

DateFieldTest output

Below are two screenshots showing the time and date functions of the DateFieldTest
MIDlet.

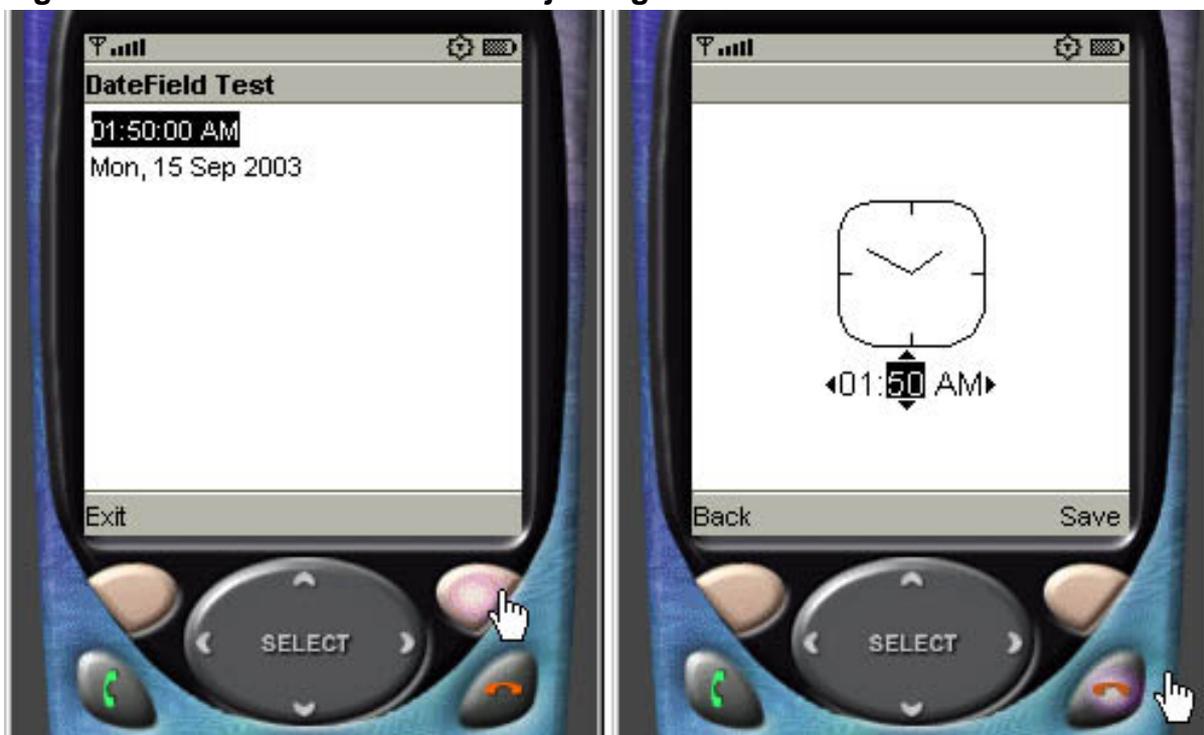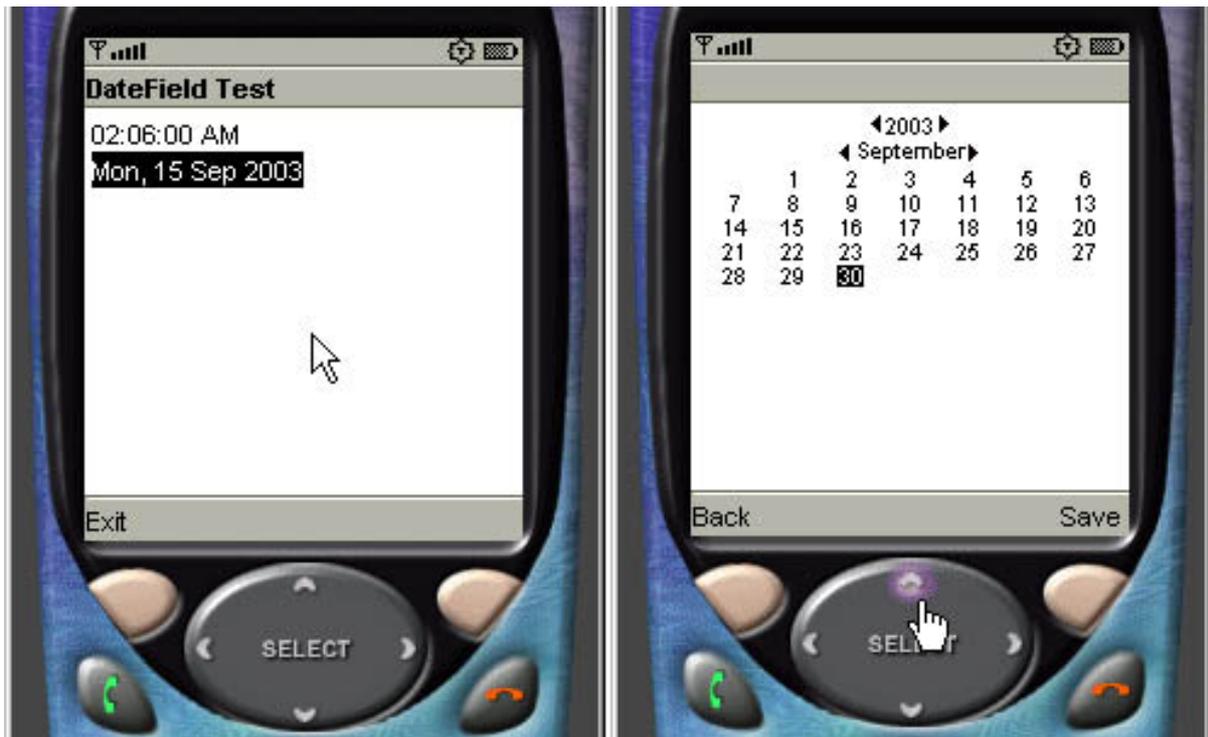**Figure 14. DateFieldTest MIDlet: Adjusting the time**

Figure 15. DateFieldTest MIDlet: Adjusting the date

DateField modes

The DateFieldTest MIDlet allows both the date and the time to be modified by the user. The `DateField` component could also be allocated to allow only the date *or* the time to be adjusted by specifying the proper *mode* parameter. Following are the declarations showing each available *mode* option:

```
DateField("Set Alarm Time", DateField. DATE_TIME );
DateField("Set Alarm Time", DateField. TIME );
DateField("Set Alarm Time", DateField. DATE );
```

The Gauge component

A `Gauge` component displays a progress-meter style interface. There are two types of `Gauge`: *interactive* and *non-interactive*. The former allows the user to make changes to the gauge. The latter requires the developer to update the gauge.

Here's the constructor for the `Gauge` component:

```
Gauge(String label, boolean interactive, int maxValue, int initialValue)
```

And here's a short code block that creates an interactive `Gauge`:

```
private Gauge gaVolume;      // Volume adjustment
```

```
gaVolume = new Gauge("Sound Level", true, 100, 4);
```

The InteractiveGauge MIDlet

Follow these steps to create a MIDlet with an interactive gauge:

1. Create a new project with the name **InteractiveGauge**.

2. Copy and paste the InteractiveGauge source code into a text editor.

3. Save the source code as InteractiveGauge.java in the directory
   \apps\InteractiveGauge\src of your WTK installation.

4. Build and run the project.

InteractiveGauge source

Here's the source code for our InteractiveGauge MIDlet:

```
/*---------------------------------------------------
 * InteractiveGauge.java
 *---------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class InteractiveGauge extends MIDlet implements CommandListener
{
  private Display display;    // Reference to display object
  private Form fmMain;        // The main form
  private Command cmExit;     // Exit the form
  private Gauge gaVolume;     // Volume adjustment

  public InteractiveGauge()
  {
    display = Display.getDisplay(this);

    // Create the gauge and exit command
    gaVolume = new Gauge("Sound Level", true, 50, 4);
    cmExit = new Command("Exit", Command.EXIT, 1);

    // Create form, add commands, listen for events
    fmMain = new Form("");
    fmMain.addCommand(cmExit);
    fmMain.append(gaVolume);
    fmMain.setCommandListener(this);
  }

  // Called by application manager to start the MIDlet.
  public void startApp()
  {
    display.setCurrent(fmMain);
  }

  public void pauseApp()
  { }

  public void destroyApp(boolean unconditional)
  { }
```
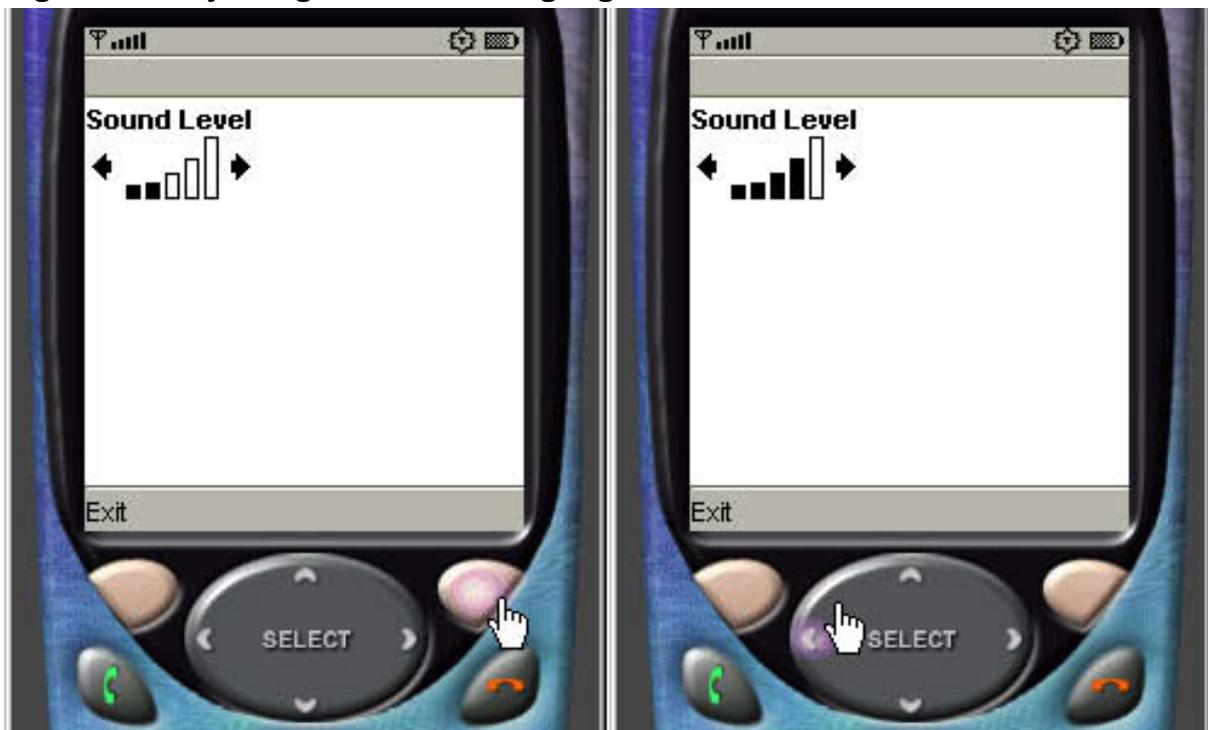
```
  public void commandAction(Command c, Displayable s)
  {
    if (c == cmExit)
    {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}
```

InteractiveGauge output

The screenshot in Figure 16 shows how the display is updated when an interactive gauge is adjusted.

Figure 16. Adjusting an interactive gauge

The StringItem component

A `StringItem` component is used to display a label and/or text string. Because a user cannot change either the label or the text when the application is running, a `StringItem` does not recognize events.

The constructor for `StringItem` is shown below:

```
StringItem(String label, String text)
```

The StringItemTest MIDlet

The StringItemTest MIDlet defines and displays a `StringItem`. The application also incorporates a command labeled **Change** that can programmatically change the label and message. Here are the steps to create a StringItemTest MIDlet:

1. Create a new project with the name **StringItemTest**.

2. Copy and paste the StringItemTest source code into a text editor.

3. Save the source code as StringItemTest.java in the directory *\apps\StringItemTest\src* of your WTK installation.

4. Build and run the project.

StringItemTest source

Here's the source code for our StringItemTest MIDlet:

```
/*---------------------------------------------------
 * StringItemTest.java
 *---------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class StringItemTest extends MIDlet implements CommandListener
{
  private Display display;      // Reference to Display object
  private Form fmMain;          // Main form
  private StringItem siMsg;     // StringItem
  private Command cmChange;     // Change the label and message
  private Command cmExit;       // Exit the MIDlet

  public StringItemTest()
  {
    display = Display.getDisplay(this);

    // Create text message and commands
    siMsg = new StringItem("Website: ", "www.IBM.com");
    cmChange = new Command("Change", Command.SCREEN, 1);
    cmExit = new Command("Exit", Command.EXIT, 1);

    // Create Form, add Command and StringItem, listen for events
    fmMain = new Form("StringItem Test");
    fmMain.addCommand(cmExit);
    fmMain.addCommand(cmChange);
    fmMain.append(siMsg);
    fmMain.setCommandListener(this);
  }

  // Called by application manager to start the MIDlet.
  public void startApp()
  {
    display.setCurrent(fmMain);
  }

  public void pauseApp()
  { }

  public void destroyApp(boolean unconditional)
```

```
    { }

  public void commandAction(Command c, Displayable s)
  {
    if (c == cmChange)
    {
      // Change label
      siMsg.setLabel("Section: ");

      // Change text
      siMsg.setText("developerWorks");

      // Remove the command
      fmMain.removeCommand(cmChange);
    }
    else if (c == cmExit)
    {
      destroyApp(false);
      notifyDestroyed();
    }
  }
}
```

StringItemTest output

As previously mentioned, a `StringItem` does not recognize events, so a user cannot change the label or text. It is, however, possible for you to change both. Figure 17 shows how the label and text can be updated when the **Change** command is selected.

Figure 17. Changing the StringItem label and text

The TextField component

A `Textfield` is analogous to any typical text entry field. You can specify a label, the maximum number of characters, and the type of data you will accept. The `TextField` component also implements a password modifier that masks characters as they are input.

Here is the constructor for `TextField`:

```
TextField(String label, String text, int maxSize, int constraints)
```

The final parameter, *constraints*, is the one we're interested in, because it is our means to specify the type of input allowed in the `TextField`.

TextField constraints

MIDP defines the following constraint parameters for the `TextField` component:

- `ANY` allows any characters.

- `EMAILADDR` allows only valid email addresses.

- `NUMERIC` allows any numeric value.

- `PHONENUMBER` allows only phone numbers.

- `URL` allows only characters that are valid within URL.

- `PASSWORD` masks all characters as they are input.

The TextFieldTest MIDlet

The TextFieldTest MIDlet will demonstrate a `TextField` that obtains and displays a series of characters input by a user. Here's the procedure to create and run the TextFieldTest MIDlet:

1. Create a new project with the name **TextFieldTest**.

2. Copy and paste the TextFieldTest source code into a text editor.

3. Save the source code as TextFieldTest.java in the directory *\apps\TextFieldTest\src* of your WTK installation.

4. Build and run the project.

TextFieldTest source

Here's the source code for our TextFieldTest MIDlet:

```
/*-------------------------------------------------
* TextFieldTest.java
*--------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TextFieldTest extends MIDlet implements CommandListener
{
  private Display display;        // Reference to Display object
  private Form fmMain;            // Main form
  private Command cmTest;         // Get contents of textfield
  private Command cmExit;         // Command to exit the MIDlet
  private TextField tfText;       // Textfield

  public TextFieldTest()
  {
    display = Display.getDisplay(this);

    // Create commands
    cmTest = new Command("Get Contents", Command.SCREEN, 1);
    cmExit = new Command("Exit", Command.EXIT, 1);

    // Textfield for phone number
    tfText = new TextField("Phone:", "", 10, TextField.PHONENUMBER);

    // Create Form, add Commands and textfield, listen for events
    fmMain = new Form("Phone Number");
    fmMain.addCommand(cmExit);
    fmMain.addCommand(cmTest);
    fmMain.append(tfText);
    fmMain.setCommandListener(this);
  }

  // Called by application manager to start the MIDlet.
  public void startApp()
  {
    display.setCurrent(fmMain);
  }

  public void pauseApp()
  { }

  public void destroyApp(boolean unconditional)
  { }

  public void commandAction(Command c, Displayable s)
  {
    if (c == cmTest)
    {
      System.out.println("TextField contains: " + tfText.getString());
    }
    else if (c == cmExit)
    {
     destroyApp(false);
     notifyDestroyed();
    }
  }
}
```

TextFieldTest output

Working with a `TextField` is straightforward. As characters are entered, they are shown in the `TextField`. What is unique about this particular `TextField` is that we have set the *constraint* field to `TextField.PHONENUMBER`. Therefore, only those values that are relevant for phone numbers will be allowed. Figure 18 shows how the WTK device emulator displays a `TextField` component that allows only characters valid in a phone number.

Figure 18. A TextField component with the phonenumber constraint
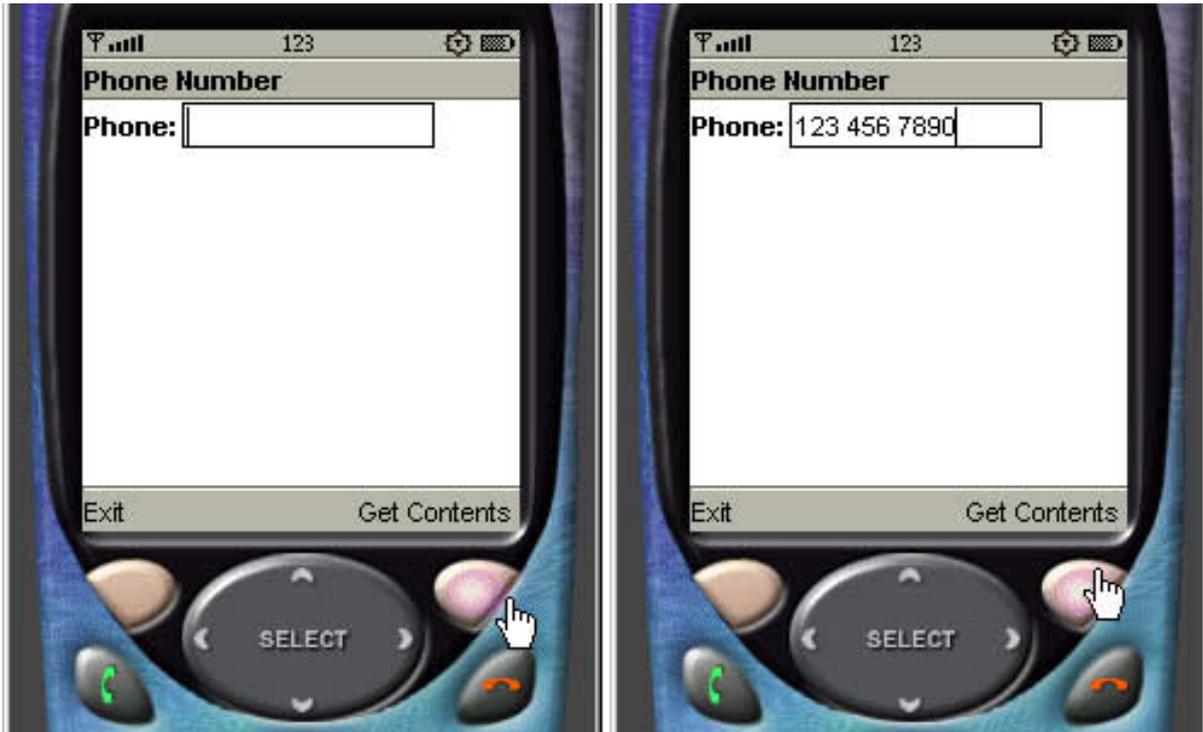
When you select the **Get contents** command the current value of the `TextField` will be printed on the console, as shown in Figure 19.

Figure 19. A TextField content display
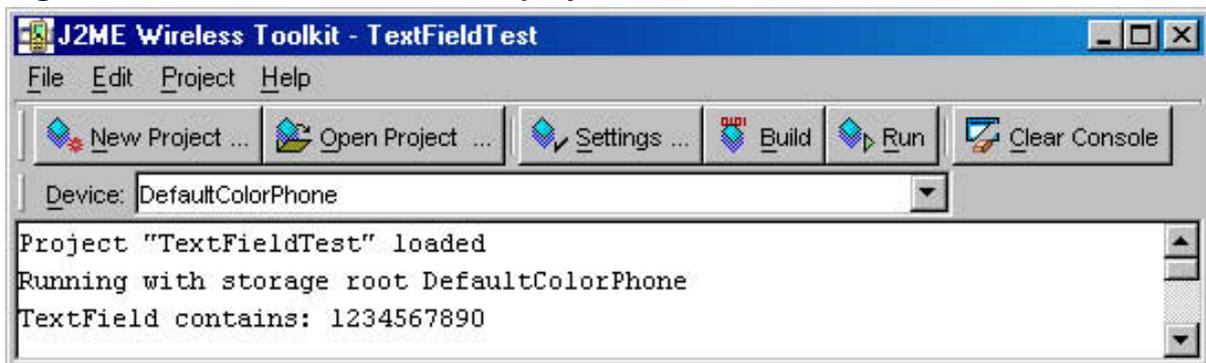
Adding a new constraint

Now let's see what happens when we add a second constraint -- a password modifier -- to our example `TextField`.

Here's the original `TextField`:

```
tfText = new TextField("Phone:", "", 10, TextField.PHONENUMBER);
```

And here's the new one:

J2ME 101, Part 1: Introduction to MIDP's high-level user interface

```
tfText = new TextField("Phone:", "", 10, TextField.PHONENUMBER | TextField.PASSWORD);
```

Save the new source code, build, and run the MIDlet. You'll now see how input is masked as each character is entered, as shown in Figure 20.

Figure 20. A TextField component with a password constraint

The ChoiceGroup component

`ChoiceGroup` components allow a user to select from a predefined list of entries. There are two `ChoiceGroup` formats: *multi-selection*, which are commonly referred to as checkboxes and *exclusive-selection*, which are essentially radio groups. Both formats are shown in Figure 21.

Figure 21. The two formats of a ChoiceGroup component
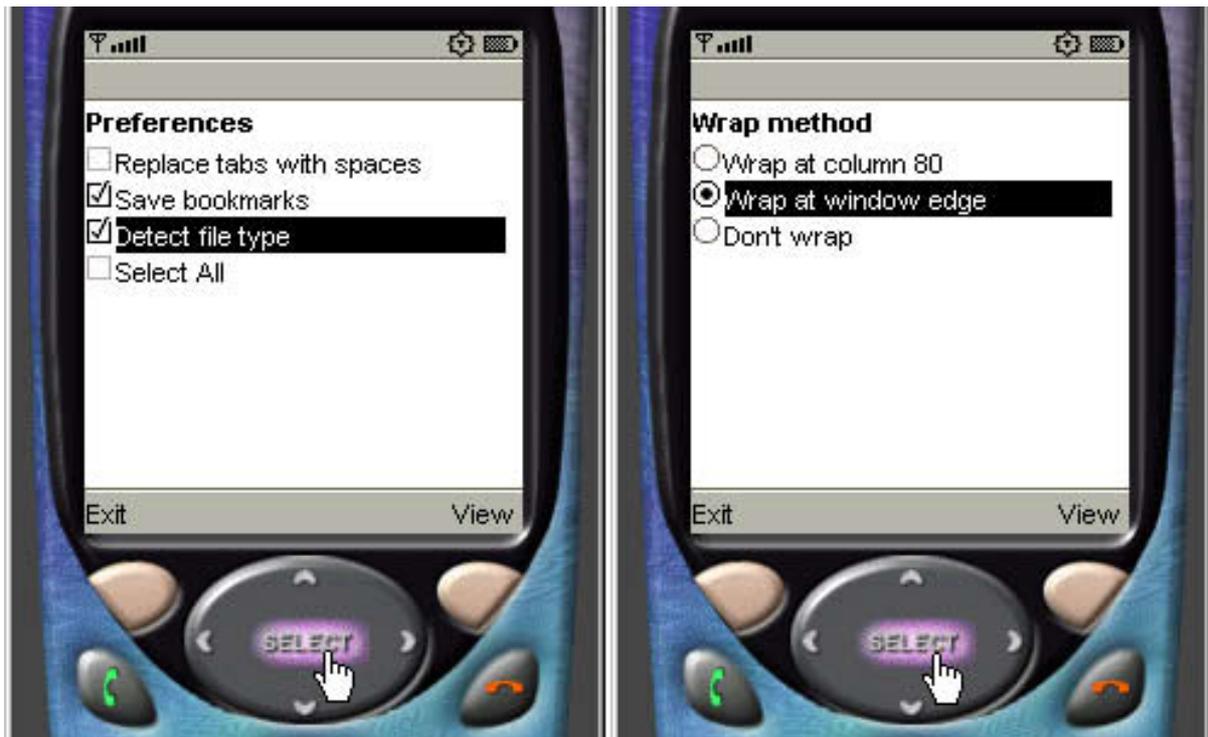
The ChoiceGroupTest MIDlet

Follow these steps to create a multi-selection ChoiceGroupTest MIDlet:

1. Create a new project with the name **ChoiceGroupTest**.

2. Copy and paste the ChoiceGroupTest source code into a text editor.

3. Save the source code as ChoiceGroupTest.java in the directory *\apps\ChoiceGroupTest\src* of your WTK installation.

4. Build and run the project.

ChoiceGroupTest source

Here's the source code for the ChoiceGroupTest MIDlet:

```
/*----------------------------------------------------
 * ChoiceGroupTest.java
 *--------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class ChoiceGroupTest extends MIDlet implements ItemStateListener, CommandListener
{
  private Display display;    // Reference to display object
  private Form fmMain;        // Main form
  private Command cmExit;     // A Command to exit the MIDlet
```

```
      private Command cmView;        // View the choice selected
      private int selectAllIndex;    // Index of the "Select All" option
      private ChoiceGroup cgPrefs;   // Choice Group of preferences

      private int choiceGroupIndex; // Index of choice group on form

      public ChoiceGroupTest()
      {
        display = Display.getDisplay(this);

        // Create a multiple choice group
        cgPrefs = new ChoiceGroup("Preferences", Choice.MULTIPLE);

        // Append options, with no associated images
        cgPrefs.append("Replace tabs with spaces", null);
        cgPrefs.append("Save bookmarks", null);
        cgPrefs.append("Detect file type", null);
        selectAllIndex = cgPrefs.append("Select All", null);

        cmExit = new Command("Exit", Command.EXIT, 1);
        cmView = new Command("View", Command.SCREEN,2);

        // Create Form, add components, listen for events
        fmMain = new Form("");
        choiceGroupIndex = fmMain.append(cgPrefs);
        fmMain.addCommand(cmExit);
        fmMain.addCommand(cmView);
        fmMain.setCommandListener(this);
        fmMain.setItemStateListener(this);
      }

      public void startApp()
      {
        display.setCurrent(fmMain);
      }

      public void pauseApp()
      { }

      public void destroyApp(boolean unconditional)
      { }

      public void commandAction(Command c, Displayable s)
      {
        if (c == cmView)
        {
          boolean selected[] = new boolean[cgPrefs.size()];

          // Fill array indicating whether each element is checked
          cgPrefs.getSelectedFlags(selected);

          for (int i = 0; i < cgPrefs.size(); i++)
            System.out.println(cgPrefs.getString(i) +
    (selected[i] ? ": selected" : ": not selected"));

        }
        else if (c == cmExit)
        {
          destroyApp(false);
          notifyDestroyed();
        }
      }

      public void itemStateChanged(Item item)
      {
        if (item == cgPrefs)
        {
          // Is "Select all" option checked ?
          if (cgPrefs.isSelected(selectAllIndex))
          {
            // Set all checkboxes to true
            for (int i = 0; i < cgPrefs.size(); i++)
              cgPrefs.setSelectedIndex(i, true);
```
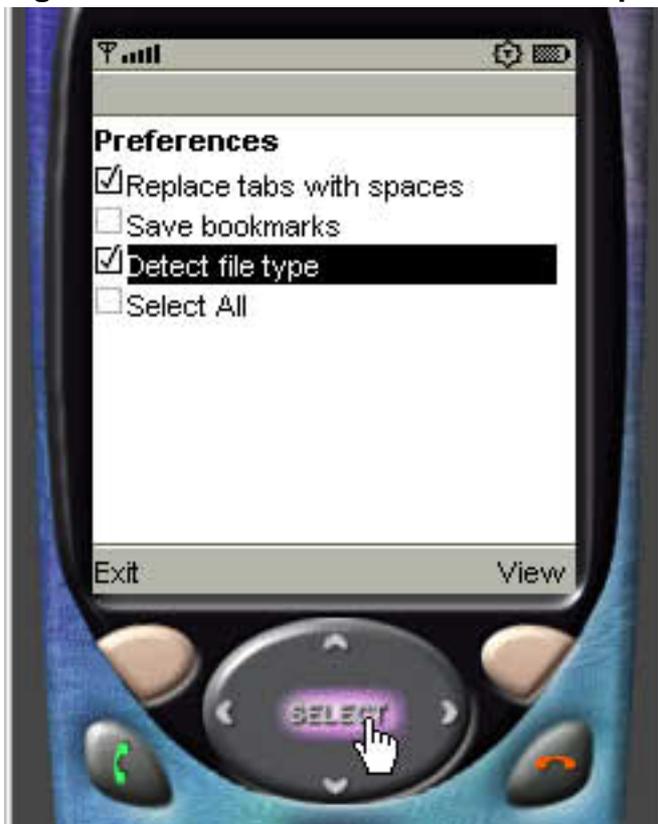
```
            // Remove the check by "Select All"
            cgPrefs.setSelectedIndex(selectAllIndex, false);
        }
      }
    }
  }
}
```

ChoiceGroupTest output

As you select and deselect the entries in the `ChoiceGroup` the display is updated
to reflect your changes. In Figure 22 we see a multi-selection `ChoiceGroup`
display.

Figure 22. A multi-selection ChoiceGroup display

To output the selection status of each entry, click the **View** command, as shown in
Figure 23.

Figure 23. Displaying the selection status on the ChoiceGroup component
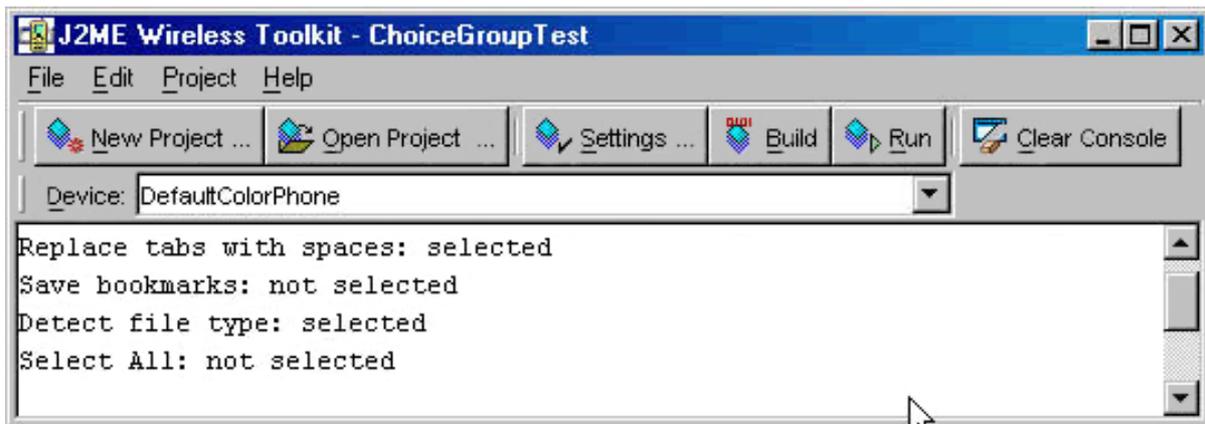
Below is the code to print the selection status of each component to the console. We fill an array of booleans with the selection status and loop through each entry, printing whether or not each one is active.

```
public void commandAction(Command c, Displayable s)
{
  if (c == cmView)
  {
    boolean selected[] = new boolean[cgPrefs.size()];

    // Fill array indicating whether each element is checked
    cgPrefs.getSelectedFlags(selected);

    for (int i = 0; i < cgPrefs.size(); i++)
      System.out.println(cgPrefs.getString(i) +
(selected[i] ? ": selected" : ": not selected"));
  }
  else if (c == cmExit)
  {
   destroyApp(false);
   notifyDestroyed();
  }
}
```

The select-all feature

There is no "select-all" feature built into a multi-selection ChoiceGroup. Supporting such a feature is quite simple, however. First, make note of the index of our select-all entry when creating the ChoiceGroup:

```
// Create a multiple choice group
cgPrefs = new ChoiceGroup("Preferences", Choice.MULTIPLE);

// Append options, with no associated images
cgPrefs.append("Replace tabs with spaces", null);
cgPrefs.append("Save bookmarks", null);
cgPrefs.append("Detect file type", null);
selectAllIndex = cgPrefs.append("Select All", null);
```

When a change in the ChoiceGroup is triggered, the method itemStateChanged() will be called. Inside this method, check to see if the "selectAllIndex " has been chosen. If so, loop through the ChoiceGroup,

setting each entry to "selected," as shown below:

```
public void itemStateChanged(Item item)
{
  if (item == cgPrefs)
  {
    // Is "Select all" option checked ?
    if (cgPrefs.isSelected(selectAllIndex))
    {
      // Set all checkboxes to true
      for (int i = 0; i < cgPrefs.size() - 1; i++)
        cgPrefs.setSelectedIndex(i, true);

      // Remove the check by "Select All"
      cgPrefs.setSelectedIndex(selectAllIndex, false);
    }
  }
}
```

Figure 24. The ChoiceGroup component select-all feature

The Spacer component

Spacer is an non-visible component that facilitates the positioning of other items on the display. You can use Spacer s to provide both vertical and horizontal white space between components, simply by specifying the width and height of each one. Because a Spacer is not a visual component it does not recognize events.

The CustomItem component

The `CustomItem` component lets you create your own `Item` components. As with other `Item` s, this type of component can be added to a `Form` and can recognize and process events.

`CustomItem` s are drawn onto the display with the `paint()` method. As the creator of the component, it is up to you to write the code inside `paint()`. The process of creating a `CustomItem` is no different from working with any other Java platform object that extends another class. The shell of a simple `CustomItem` component is shown below:

```
public class NewItem extends CustomItem
{
    public NewItem(String label)
    {
      super(label);
      ...
    }

    protected void paint(Graphics g, int width, int height)
    {
      ...
    }

    protected int getMinContentHeight()
    {
      ...;
    }

    protected int getMinContentWidth()
    {
      ...
    }

    protected int getPrefContentHeight(int width)
    {
      ...
    }

    protected int getPrefContentWidth(int height)
    {
      ...
    }

...

}
```

While `CustomItem` is one of the most exciting components of the MIDP's high-level interface, it is beyond the scope of this tutorial for us to create one together. See Resources to learn more about the `CustomItem` component.

The ImageItem and Image components

Two classes are used to work with display images: `Image` and `ImageItem`. `Image` is used to create an image object and holds information such as the height and width, and whether or not the image is mutable. `ImageItem` defines how an image will be displayed; that is, whether the image will be centered, to the left, at the top of the screen, etc.

MIDP offers two types of images: immutable and mutable. An *immutable* image

cannot be changed once it has been created. Typically, this type of image is read from a resource such as a file. A *mutable* image is essentially a chunk of memory. It is up to you to create the contents of the image by writing it into the memory block. We'll work with immutable images in the sections that follow. You'll learn about mutable images in Part 2, when we discuss the low-level interface.

Constructors for the Image and ImageItem classes

Following are the constructors for the `Image` and `ImageItem` classes:

```
Image createImage(String name)
Image createImage(Image source)
Image createImage(byte[] imageDate, int imageOffset, int imageLength)
Image createImage(int width, int height)
Image createImage(Image image, int x, int y, int width,
                 int height, int transform)
Image createImage(InputStream stream)
Image createRGBImage(int[] rgb, int width, int height,
                 boolean processAlpha)

ImageItem(String label, Image img, int layout, String altText)
```

Steps for displaying an image

The following code shows how we could create an image from a file, associate it with an `ImageItem` object, and append the image onto a `Form`.

```
Form fmMain = new Form("Images");

...

// Create an image
Image img = Image.createImage("/house.png");

// Append to a form
fmMain.append(new ImageItem(null, img, ImageItem.LAYOUT_CENTER, null));
```

The ImageTest MIDlet

The ImageTest MIDlet will display an image that is read from a file. The file will be stored in the same JAR file that contains the MIDlet. The build-and-run cycle for the ImageTest MIDlet will be slightly different from the one we've used for earlier MIDlets because it will incorporate the use of multiple image files. Here's the process for creating the ImageTest MIDlet:

1.   Create a new project with the name **ImageTest**.

2.   Copy and paste the ImageTest source code into a text editor.

3.   Save the source code as ImageTest.java in the directory

\apps\ImageTest\src of your WTK installation.

4. Create two PNG images, one with the name *image_color.png*, and the other with the name *image_bw.png*. Save these images in the \apps\ImageTest\res directory.

5. Build and run the project.

Note that Portable Network Graphics (PNG) is the only image format required for any MIDP device implementation. See Resources to learn more about PNG.

ImageTest source

Here's the source code for the ImageTest MIDlet:

```
/*-------------------------------------------------
 * ImageTest.java
 *-------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class ImageTest extends MIDlet  implements CommandListener
{
  private Display display;      // Reference to Display object
  private Form fmMain;          // The main form
  private Command cmExit;       // Command to exit the MIDlet

  public ImageTest()
  {
    display = Display.getDisplay(this);

    cmExit = new Command("Exit", Command.EXIT, 1);
    fmMain = new Form("");
    fmMain.addCommand(cmExit);
    fmMain.setCommandListener(this);

    try
    {
      // Read the appropriate image based on color support
      Image im = Image.createImage((display.isColor()) ?
                             "/image_color.png":"/image_bw.png");

      fmMain.append(new ImageItem(null, im, ImageItem.LAYOUT_CENTER, null));

      display.setCurrent(fmMain);
    }
    catch (java.io.IOException e)
    {
      System.err.println("Unable to locate or read .png file");
    }
  }

  public void startApp()
  {
    display.setCurrent(fmMain);
  }

  public void pauseApp()
  {
  }

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

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

Color or grayscale?

You most likely noted that the Image source code contains a provision for displaying the appropriate image based on whether or not a device supports color. When the application is started the appropriate image will be read from the JAR file, as shown below:

```
// Read the appropriate image based on color support
Image im = Image.createImage((display.isColor()) ?
                             "/image_color.png":"/image_bw.png");

// Create ImageItem and append to the form
fmMain.append(new ImageItem(null, im, ImageItem.LAYOUT_CENTER, null));
```
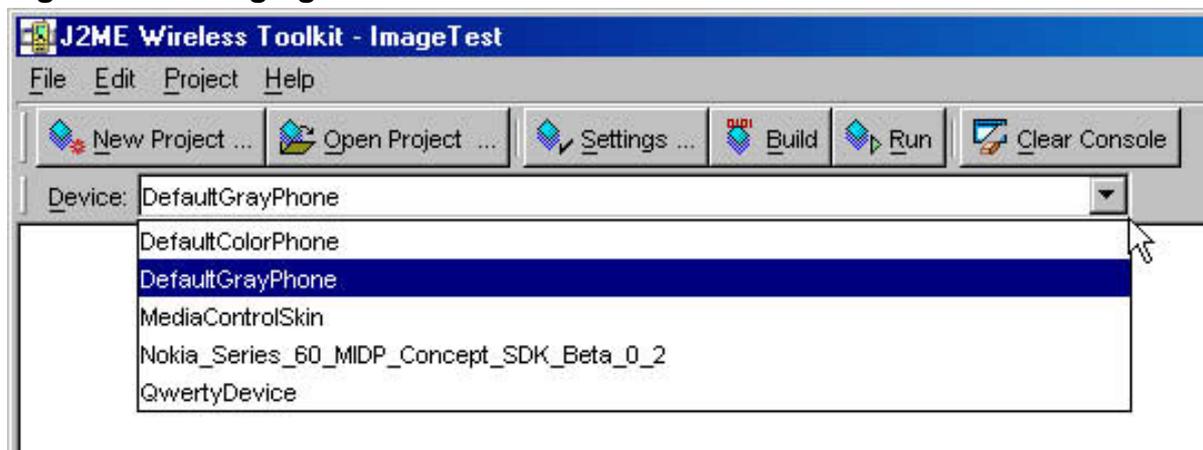
ImageTest output

By default, the WTK emulator uses the `DefaultColorPhone`, and the device will display the *image_color.png* file, as shown in Figure 25.

Figure 25. Displaying an image from a default file
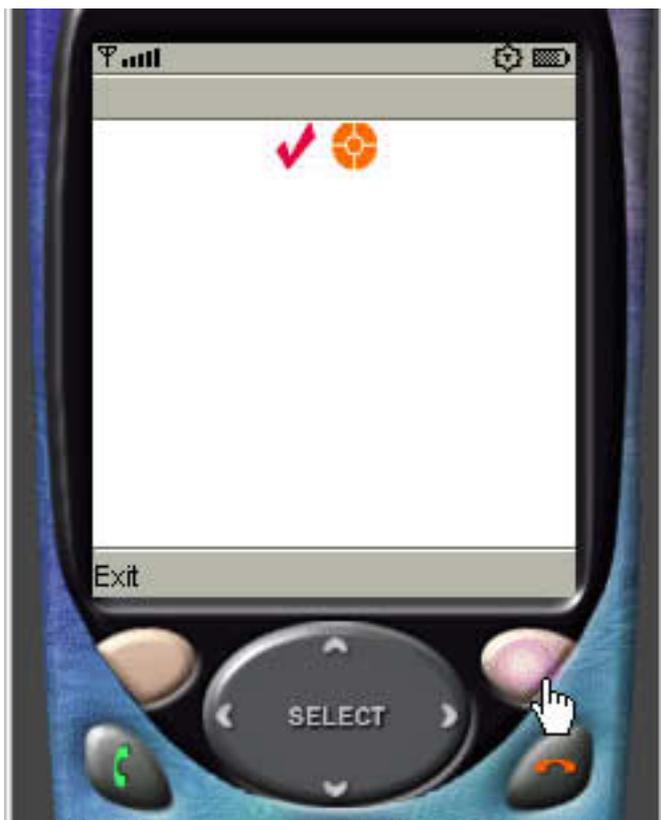
If you change the emulator to use the `DefaultGrayPhone`, as shown in Figure 26, the MIDlet will load *bw_image.png*.

Figure 26. Changing the default emulator

The resulting image is shown in Figure 27.

Figure 27. The resulting black-and-white image

Form summary

In this section you've learned about the various components that can be shown on a
`Form`. Each of the components in this section is a subset of the `Item` class. A `Form`
can contain any number of `Item` objects. The `Form` component scrolls the display to
accommodate multiple components, and also handles event processing for every
`Item` it contains.

Section 6. List, TextBox, Alert, and Ticker components

## List, textBox, alert, and ticker components overview

In this section you will learn about the `List`, `TextBox`, `Alert`, and `Ticker`
components of the MIDP high-level interface. Unlike the `Form` component, which
provides vertical scrolling to accommodate all the components on the display, a
`List`, `TextBox`, or `Alert` object takes up the entire display on its own.

`Ticker` is somewhat different from the other components discussed in this section,
in that it is not a subclass of `Screen`. Rather, it is a variable defined in the `Screen`

class. As a result, any object that subclasses `Screen` can also display a `Ticker`.

Let's quickly review the device display hierarchy to get a feel for where the next set of components sit in relation to the `Form` and `Item` components.

Figure 28. The complete device display hierarchy
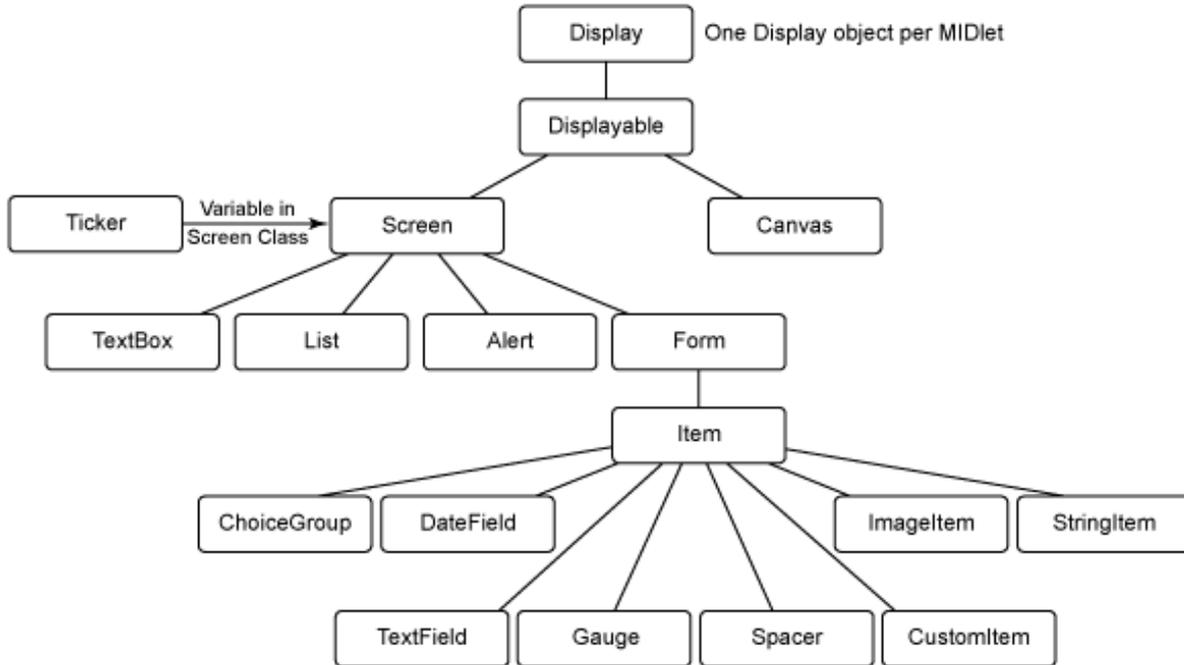
The List component

A `List` contains a series of choices presented in one of three formats. We've seen the multiple-selection and exclusive-selection formats previously, when working with the `ChoiceGroup`. The third format available is implicit. *Implicit lists* are typically used to represent a menu of choices, as the ImplicitList MIDlet will demonstrate.

The ImplicitList MIDlet

Follow these steps to create the ImplicitList MIDlet:

1.   Create a new project with the name **ImplicitList**.

2.   Copy and paste the ImplicitList source code into a text editor.

3.   Save the source code as ImplicitList.java in the directory
      \apps\ImplicitList\src of your WTK installation.

4.   Optionally, you can create three PNG images with the names *next.png*,
      previous.png, and *new.png*. Copy these images into the
      \apps\ImplicitList\res directory.

5. Build and run the project.

ImplicitList source

Here's the source code for our ImplicitList MIDlet:

```java
/*--------------------------------------------------
 * ImplicitList.java
 *--------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class ImplicitList extends MIDlet implements CommandListener
{
  private Display display;    // Reference to Display object
  private List lsDocument;    // Main list
  private Command cmExit;     // Command to exit

  public ImplicitList()
  {
    display = Display.getDisplay(this);

    // Create the Commands
    cmExit = new Command("Exit", Command.EXIT, 1);

    try
    {
      // Create array of image objects
      Image images[] = {Image.createImage("/next.png"),
                        Image.createImage("/previous.png"),
                        Image.createImage("/new.png")};

      // Create array of corresponding string objects
      String options[] = {"Next", "Previous", "New"};

    // Create list using arrays, add commands, listen for events
    lsDocument = new List("Document Option:", List.IMPLICIT, options, images);

    // If you have no images, use this line to create the list
    //      lsDocument = new List("Document Option:", List.IMPLICIT, options, null);

    lsDocument.addCommand(cmExit);
    lsDocument.setCommandListener(this);
  }
  catch (java.io.IOException e)
  {
    System.err.println("Unable to locate or read .png file");
  }
        }

  public void startApp()
  {
    display.setCurrent(lsDocument);
  }

  public void pauseApp()
  {
  }

  public void destroyApp(boolean unconditional)
  {
  }

  public void commandAction(Command c, Displayable s)
  {
    // If an implicit list generated the event
    if (c == List.SELECT_COMMAND)
    {
```

```
        switch (lsDocument.getSelectedIndex())
        {
          case 0:
            System.out.println("Next selected");
            break;

          case 1:
            System.out.println("Previous selected");
            break;

          case 2:
            System.out.println("New selected");
            break;

        }
      }
      else if (c == cmExit)
      {
        destroyApp(false);
        notifyDestroyed();
      }
    }
  }
}
```

ImplicitList output

With this example, we included several images to display alongside each entry in the
list. The code block to allocate the images and labels of each string is shown here:

```
// Create array of image objects
Image images[] = {Image.createImage("/next.png"),
        Image.createImage("/previous.png"),
        Image.createImage("/new.png")};

// Create array of corresponding string objects
String options[] = {"Next", "Previous", "New"};

// Create list using arrays, add commands, listen for events
lsDocument = new List("Document Option:", List.IMPLICIT,
                        options, images);
```

The `ImplicitList` and associated image output are shown in Figure 29.

Figure 29. ImplicitList output
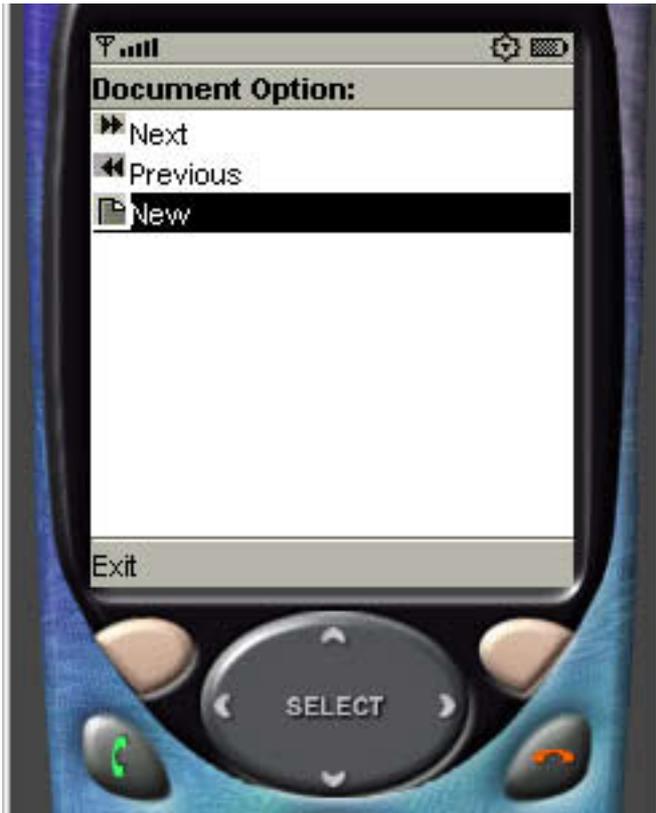
Determining which list entry was selected

As previously mentioned, an implicit list is typically used to represent a menu of choices. We've completed the first step in working with a menu-style interface -- as shown in Figure 29 on the previous section, ImplicitList output, the `ImplicitList` component now displays each entry. We can see how the list is managed internally by reviewing the code inside `commandAction()`.

```
public void commandAction(Command c, Displayable s)
{
  // If an implicit list generated the event
  if (c == List.SELECT_COMMAND)
  {
    switch (lsDocument.getSelectedIndex())
    {
      case 0:
        System.out.println("Next selected");
        break;

      case 1:
        System.out.println("Previous selected");
        break;

      case 2:
        System.out.println("New selected");
        break;

    }
  }
```

J2ME 101, Part 1: Introduction to MIDP's high-level user interface

Whenever an entry is selected, we use a switch statement to determine which one generated the call. For this MIDlet, we simply print a message to the console, as shown in Figure 30.

**Figure 30. ImplicitList console output**
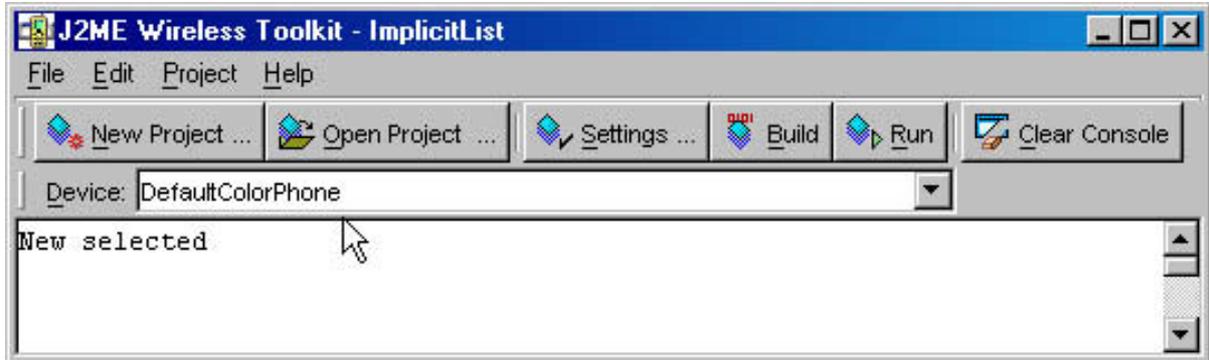
An ImplicitList without images

If you opt not to create or show the PNG images, you can change the code that creates the `ImplicitList` MIDlet by simply specifying `null` for the parameter that references the array of images, as shown below:

```
lsDocument = new List("Document Option:", List.IMPLICIT, options, null);
```

Figure 31 shows the result of this change.

Figure 31. An ImplicitList without the Image
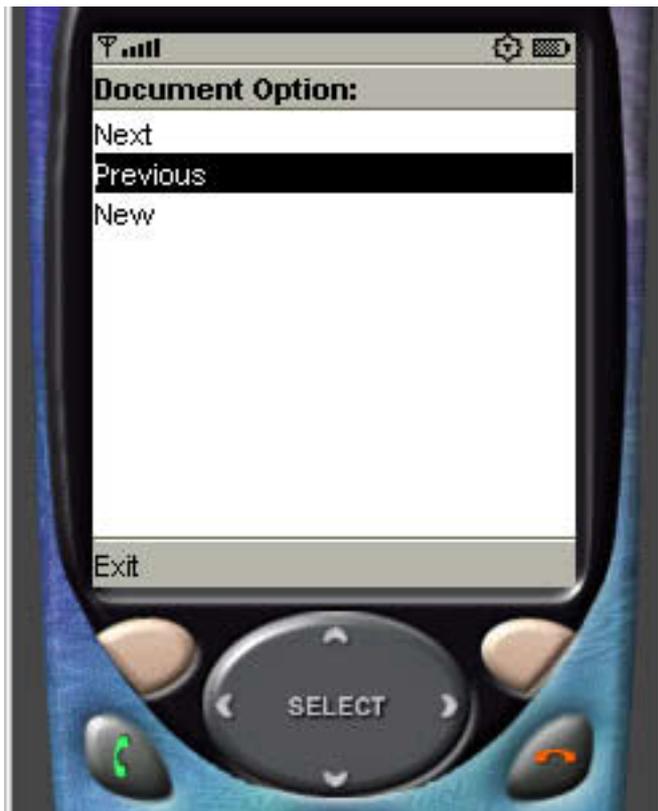
The TextBox component

`TextBox` components are used to allow for multiple-line input. The `TextBox` and `TextField` components share the same constraints (see TextField constraints) for specifying the type of content allowed (for example, `ANY`, `EMAIL`, `URL`, etc.).

Here's the constructor for a `TextBox`:

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

The TextBoxTest MIDlet

Here are the steps for creating a TextBoxTest MIDlet:

1. Create a new project with the name **TextBoxTest**.

2. Copy and paste the TextBoxTest source code into a text editor.

3. Save the source code as TextBoxTest.java in the directory *\apps\TextBoxTest\src* of your WTK installation.

4. Build and run the project.

TextBoxTest source

Here's the source code for the TextBoxTest MIDlet:

```
/*--------------------------------------------------
 * TextBoxTest.java
 *--------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TextBoxTest extends MIDlet implements CommandListener
{
  private Display display;      // Reference to Display object
  private TextBox tbClip;       // Main textbox
  private Command cmExit;       // Command to exit

  public TextBoxTest()
  {
    display = Display.getDisplay(this);

    // Create the Commands. Notice the priorities assigned
    cmExit = new Command("Exit", Command.EXIT, 1);

    tbClip = new TextBox("Textbox Test", "Contents go here..",125, TextField.ANY);
    tbClip.addCommand(cmExit);
    tbClip.setCommandListener(this);

  }

  public void startApp()
  {
    display.setCurrent(tbClip);
  }

  public void pauseApp()
  {
  }

  public void destroyApp(boolean unconditional)
  {
  }

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

TextBoxTest output

The `TextBox` component supports multiple lines of input, as shown in Figure 32.

Figure 32. TextBox component output

Should you need to change the type of content that a user can enter into a `TextBox`, you would simply change the constraint parameter. For example, if you added the declaration shown below only numeric values would be accepted.

```
tbClip = new TextBox("Textbox Test", "Contents go here..", 125, TextField.NUMERIC);
```

The Alert and AlertType components

An `Alert` is essentially a very scaled-down dialog box. There are two types of `Alert`: *modal*, which displays the dialog until acknowledged by the user, and *non-modal*, which is displayed for a specified number of seconds.

The constructors for an `Alert` are shown below:

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

The `AlertType` component uses sound, rather than a visual cue, to notify the user of an event. For instance, you could program the `AlertType` to play a particular sound to signal a user error.

The `AlertType` component comes with five pre-defined sounds: alarm, confirmation, error, info, and warning.

Looking back at `Alert` constructors, notice that an `Alert` can also include a reference to a `AlertType`. The end result is an `Alert` dialog that is preceded by the sound indicated in the `AlertType`.

The AlertTest MIDLet

The MIDlet we create in this section will demonstrate how the `Alert` and `AlertType` components can work together. We'll start with an exclusive `ChoiceGroup` that contains an entry for each sound that can be broadcast by the `AlertType` component. When you select one of the entries, the `Alert` dialog will declare which entry has been chosen and the `AlertType` component will output the appropriate sound. Here are the steps to create and run the AlertTest MIDlet:

1.   Create a new project with the name **AlertTest**.

2.   Copy and paste the AlertTest source code into a text editor.

3.   Save the source code as AlertTest.java in the directory *\apps\AlertTest\src* of your WTK installation.

4.   Build and run the project.

AlertTest source

Here's the source code for our AlertTest MIDlet:

```
/*--------------------------------------------------
 * AlertTest.java
 *-------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class AlertTest extends MIDlet implements ItemStateListener, CommandListener
{
  private Display display;    // Reference to display object
  private Form fmMain;        // Main form
  private Command cmExit;     // Command to exit the MIDlet
  private ChoiceGroup cgSound; // Choice group

  public AlertTest()
  {
    display = Display.getDisplay(this);

    // Create an exclusive (radio) choice group
    cgSound = new ChoiceGroup("Choose a sound", Choice.EXCLUSIVE);

    // Append options, with no associated images
    cgSound.append("Info", null);
    cgSound.append("Confirmation", null);
    cgSound.append("Warning", null);
    cgSound.append("Alarm", null);
    cgSound.append("Error", null);

    cmExit = new Command("Exit", Command.EXIT, 1);

    // Create Form, add components, listen for events
```

```
      fmMain = new Form("");
      fmMain.append(cgSound);
      fmMain.addCommand(cmExit);
      fmMain.setCommandListener(this);
      fmMain.setItemStateListener(this);
   }

   public void startApp()
   {
     display.setCurrent(fmMain);
   }

   public void pauseApp()
   { }

   public void destroyApp(boolean unconditional)
   { }

   public void commandAction(Command c, Displayable s)
   {
     if (c == cmExit)
           {
             destroyApp(false);
             notifyDestroyed();
           }
   }

   public void itemStateChanged(Item item)
   {
     Alert al = null;

     switch (cgSound.getSelectedIndex())
     {
       case 0:
         al = new Alert("Alert sound", "Info sound",
                        null, AlertType.INFO);
         break;

       case 1:
         al = new Alert("Alert sound", "Confirmation sound",
                        null, AlertType.INFO);
         break;

       case 2:
         al = new Alert("Alert sound", "Warning sound",
                        null, AlertType.INFO);
         break;

       case 3:
         al = new Alert("Alert sound", "Alarm sound",
                        null, AlertType.INFO);
         break;

       case 4:
         al = new Alert("Alert sound", "Error sound",
                        null, AlertType.INFO);
         break;
     }

     if (al != null)
     {
       // Wait for user to acknowledge the alert
       al.setTimeout(Alert.FOREVER);

       // Display alert, show main form when done
       display.setCurrent(al, fmMain);
     }
   }
}
```
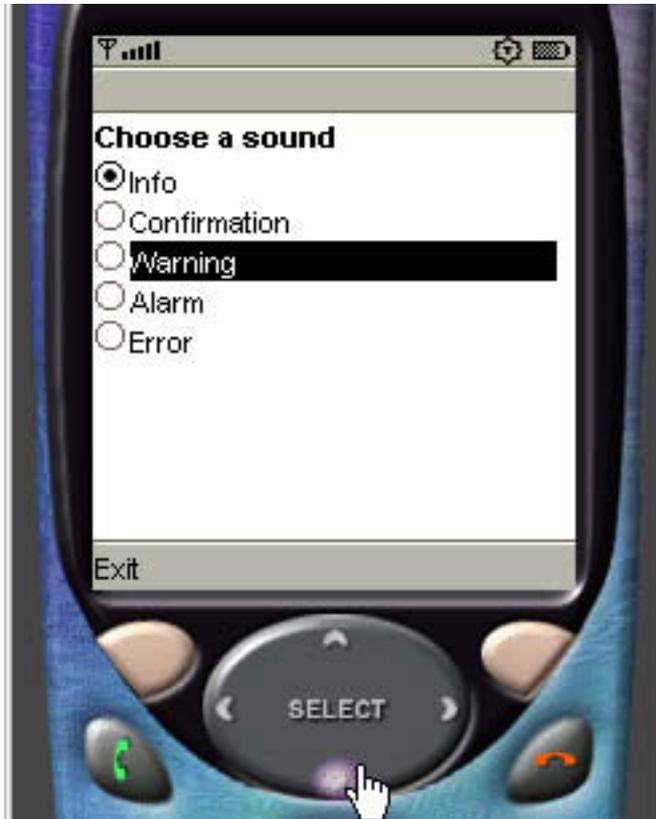
AlertTest output

The main interface for the AlertTest MIDlet is a `ChoiceGroup`, which lists each available `AlertType`, as shown in Figure 33.

Figure 33. A ChoiceGroup for selecting alert types

When you select an entry, an `Alert` dialog informs you which sound has been selected and the appropriate sound is played, as shown in Figure 34.

Figure 34. The ChoiceGroup component displays selection status

Creating a modal Alert

Once again, the code below demonstrates how we can implement the `Alert` and
`AlertType` components together. The code also demonstrates how to create a
modal (versus non-modal) dialog box.

```
al = new Alert("Alert sound", "Error sound", null, AlertType.INFO);
...

// Wait for user to acknowledge the alert
al.setTimeout(Alert.FOREVER);
```

By simply setting the timeout value to `Alert.FOREVER`, you ensure that the alert
will play or display, or both, until acknowledged by the user.

The Ticker component

The `Ticker` component presents a horizontal scrolling banner. The only parameter
you can specify for the `Ticker` is the text message to display. The rate and
direction of the scrolling is determined by the device implementation.

The constructor for Ticker is as follows:

```
Ticker(String str)
```

Referring to the device-display hierarchy one last time in Figure 35, notice that the `Ticker` component is not a subclass of `Screen`. Rather, `Ticker` is a variable in the `Screen` class. This means that a `Ticker` can be attached to any subclass of `Screen`, including an `Alert`.

Figure 35. The complete device display hierarchy
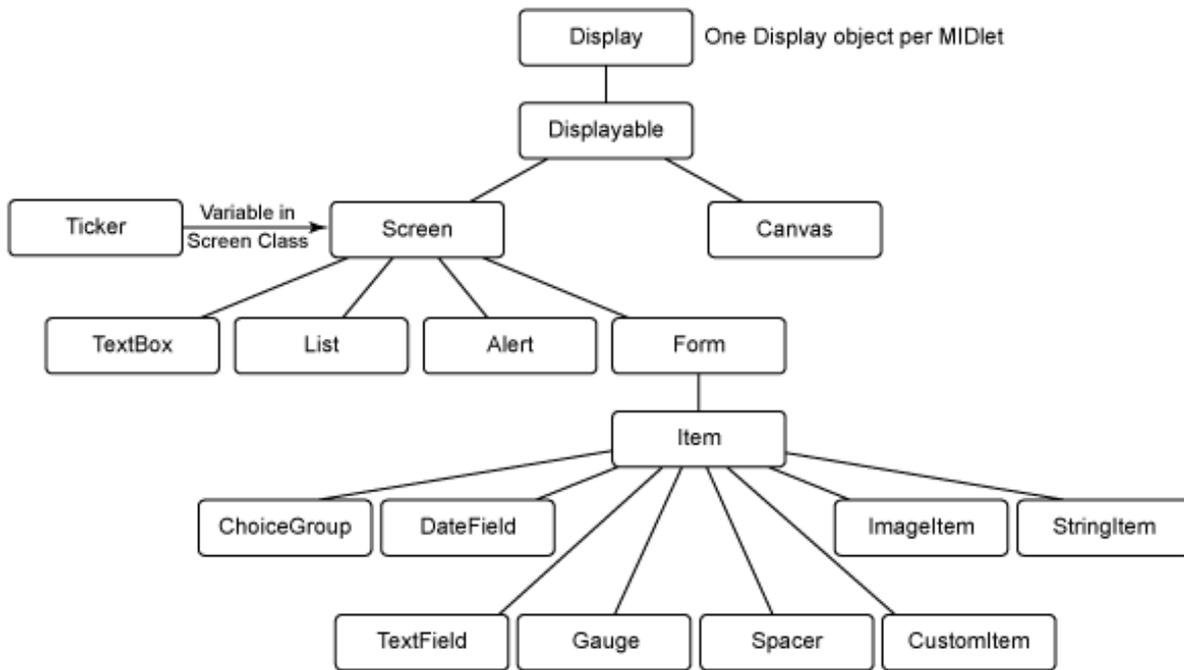
Finally, here is the definition of `Ticker`, as shown inside the `Screen` class:

```
public abstract class Screen extends Displayable
{
   ...
   private Ticker ticker = null;
   ...
}
```

The TickerTest MIDlet

The final MIDlet we will create together for this first half of the tutorial utilizes a `List` component and a `Ticker` component. The `Ticker` scrolls across the top of the display, while the `List` shows a series of products. Here are the steps to create the TickerTest MIDlet:

1.   Create a new project with the name **TickerTest**.

2.   Copy and paste the TickerTest source code into a text editor.

3. Save the source code as TickerTest.java in the directory
    \apps\TickerTest\src of your WTK installation.

4. Build and run the project.

TickerTest source

Here's the source code for our TickerTest MIDlet:

```java
/*--------------------------------------------------
 * TickerTest.java
 *--------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TickerTest extends MIDlet implements CommandListener
{
  private Display display;    // Reference to Display object
  private List lsProducts;    // Products
  private Ticker tkSale;      // Ticker
  private Command cmExit;     // Command to exit the MIDlet

  public TickerTest()
  {
    display = Display.getDisplay(this);

    cmExit = new Command("Exit", Command.SCREEN, 1);

    tkSale = new Ticker("Sale: Real Imitation Cuban Cigars...10 for $10");

    lsProducts = new List("Products", Choice.IMPLICIT);
    lsProducts.append("Wicker Chair", null);
    lsProducts.append("Coffee Table", null);
    lsProducts.addCommand(cmExit);
    lsProducts.setCommandListener(this);
    lsProducts.setTicker(tkSale);
  }

  public void startApp()
  {
    display.setCurrent(lsProducts);
  }

  public void pauseApp()
  {
  }

  public void destroyApp(boolean unconditional)
  {
  }

  public void commandAction(Command c, Displayable s)
  {

    if (c == List.SELECT_COMMAND)
    {
      switch (lsProducts.getSelectedIndex())
      {
        case 0:
          System.out.println("Chair selected");
          break;

        case 1:
          System.out.println("Table selected");
          break;
      }
```

```
    }
    else if (c == cmExit)
    {
      destroyApp(true);
      notifyDestroyed();
    }
  }
}
```

TickerTest output

Below is the TickerTest MIDlet's output. Notice that the scrolling `Ticker` appears at the top of the device display and the `List` appears below.

Figure 36. The TickerTest MIDlet

Screen summary

In this section you've been introduced to the `TextBox`, `List`, and `Alert` components. Each of these components is a subclass of `Screen`. Unlike the `Item` components discussed in the previous section, components that subclass `Screen` can only be displayed on a device display one at a time.

Also introduced in this section was the `Ticker` component, which is not a subclass of `Screen`. Rather, `Ticker` is a variable in the `Screen` class, which can be attached to any subclass of `Screen`.

This completes our discussion of the high-level user interface components within MIDP.

Section 7. Wrap-up

Summary

You've completed the first part of this two-part introduction to J2ME and MIDP. In this first part of the tutorial, you've learned about the user-interface components that comprise MIDP's high-level API, as well as the two event-handling techniques for the API. As part of the tutorial, you've also gained considerable hands-on experience with creating and running MIDlets in the J2ME environment.

At this point, you know enough to start creating your own MIDlets with simple user-interface capabilities. In Part 2 of the tutorial, we'll expand on what you've learned here, with an introduction to MIDP's low-level user interface components. The low-level API component are more complex than the ones discussed here, but they'll also afford you a lot more control over the look-and-feel of your mobile device applications.

Resources

Learn

- Don't miss the other content in the *J2ME 101* series:

    - " J2ME 101, Part 2: Introduction to MIDP's low-level UI" (*developerWorks*, December 2003)

    - " J2ME 101, Part 3: Inside the Record Management System" (*developerWorks*, December 2003)

    - " J2ME 101, Part 4: The Generic Connection Framework" (*developerWorks*, January 2004)

- Also see the IBM Developer Kits for the Java platform page.

- Explore the power of the JAR file format in " JAR files revealed" (*developerWorks*, October 2003).

- If you are new to the Wireless Toolkit, " MIDlet development with the Wireless Toolkit" (*developerWorks*, March 2003) provides an excellent starting point for learning to use it.

- To learn more about the PNG image format, visit the PNG Website.

- Mikko Kontio's " Custom GUI development with MIDP 2.0" (*developerWorks*, May 2003) is a good introduction to the `CustomItem` class.

- The WebSphere Micro Environment provides an end-to-end solution connecting cellular phones, PDAs, and other pervasive devices to e-business.

- The alphaWorks Web Services Toolkit for Mobile Devices provides tools and a run time environments for developing applications that use Web services on small mobile devices, gateway devices, and intelligent controllers.

- Core J2ME (Prentice Hall PTR, 2002) by John W. Muchow is a comprehensive guide to J2ME development. You can also visit the Core J2ME Web site for additional articles, tutorials, and developer resources.

- The *developerWorks* Wireless zone offers a wealth of technical content on pervasive computing.

- You'll find hundreds of articles about every aspect of Java programming in the dW Java technology zone.

- Also see the Java technology zone tutorials page for a complete listing of free Java-related tutorials from *developerWorks*.

Get products and technologies

- Download the JDK version 1.4.1.

- Download the J2ME Wireless Toolkit version 2.0.

About the author

John Muchow
John Muchow is a contributing developerWorks author.