J2ME 101, Part 2: Introduction to MIDP's low-level UI

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Getting started

What is this tutorial about?

This is the second 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). The first tutorial (see Resources on page 51) served as an introduction to MIDP, with particular attention to the components that comprise MIDP's high-level interface. We also discussed event-handling techniques on the high-level API and walked through a series of exercises designed to teach you, hands-on, about developing applications on the J2ME platform.

In this second tutorial, you will learn about the most important components of MIDP's low-level interface. We'll discuss the basics of creating and working with the `Canvas` and `Graphics` classes. The `Canvas` class is the backdrop of the low-level API, used to create and define the display area for your application. The `Graphics` class lets you draw shapes and images on the display area, or screen, as well as select different fonts for your text display. As with Part 1, we'll introduce each component and then build a MIDlet to demonstrate its capabilities. The tutorial will conclude with a brief overview of the Game API, which was introduced with MIDP 2.0.

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.*

Should I take this tutorial?

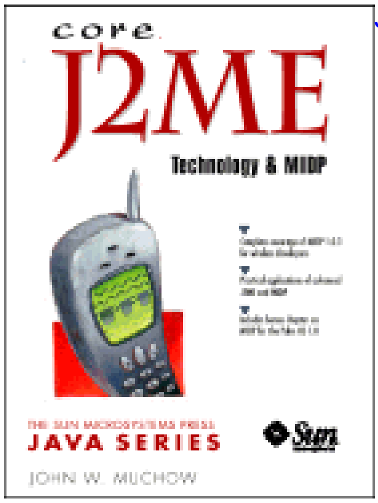
This tutorial is intended for experienced Java programmers who would like to learn how to develop mobile applications using J2ME. The code examples are not particularly complex, but it is assumed that you understand how classes are created, inherited, and instantiated within the Java platform. You will also benefit from a working knowledge of using and creating Java Archive Files (JARs). See Resources on page 51 for more information on using JARs.

Software and installation requirements

To complete this tutorial you will need to install *JDK version 1.4* (http://java.sun.com/products/jdk/1.4) or greater, along with the *J2ME Wireless Toolkit* (http://java.sun.com/products/j2mewtoolkit) (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.

About the author



John Muchow (mailto:John@CoreJ2ME.com) , author of *Core J2ME Technology and MIDP* (Prentice Hall, 2002), is a freelance technical writer and developer. Visit *Core J2ME* (http://www.CoreJ2ME.com/) for additional source code, articles, and developer resources. Send John *e-mail* (mailto:john@corej2me.com) for additional information about writing or development projects.

Section 2. The low-level API

Overview

Although the high-level API provides a complete set of components for building an application user interface, the high-level components do not provide a means to draw directly onto the device display. Without this flexibility, we are greatly limited as to the types of applications we can create. For example, most mobile game developers rely on the ability to draw lines and shapes as an integral part of the development process.

If the high-level API is the workhorse of MIDP, then the low-level API is the one we turn to when we need to let our imaginations run wild. In the sections that follow, you'll learn about all of the essential components of this most powerful API.

Components of the low-level API

`Canvas` and `Graphics` are the two classes at the heart of the low-level API. As you will soon discover, they actually work hand-in-hand. `Canvas` forms the backdrop for writing onto the device display as well as managing related events. `Graphics` provides the actual tools for drawing, such as `drawRoundRect()` and `drawString()`.

In the next section, you'll learn all the essentials of working with the `Canvas` class.

Section 3. The Canvas class

Overview

The `Canvas` class provides the backdrop for creating a custom (or *low-level*) user interface. The bulk of the methods in this class are used to handle events and draw images and text onto the device display. We'll cover the following topics in this section:

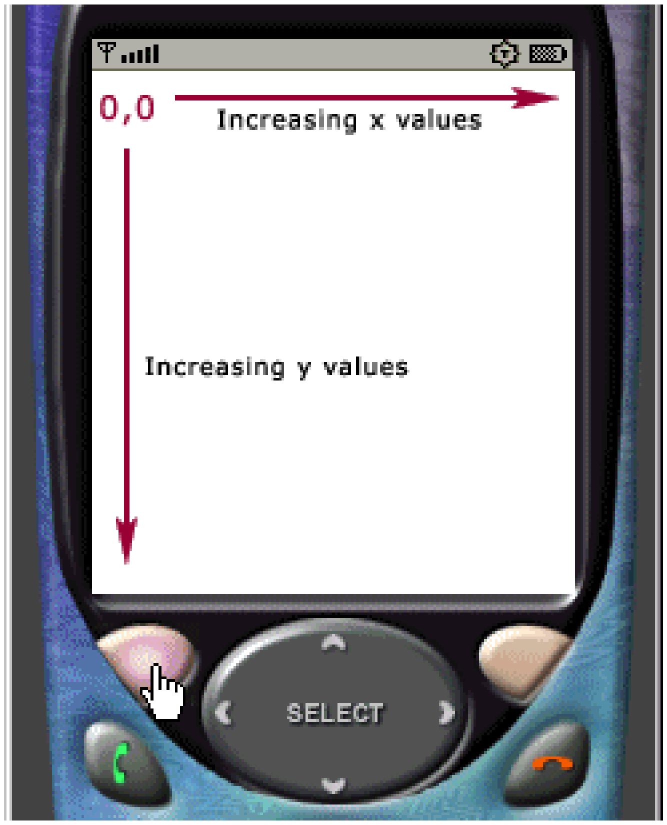
- The coordinate system
- Creating a `Canvas` object
- Painting on the `Canvas`
- Command events
- Keycode events
- Game action events
- Pointer events

We'll create two MIDlets that demonstrate the capabilities of the `Canvas` class. The first one, KeyMapping, will show how to capture, recognize, and process keycode- and game-related events. The second one, ScratchPad, MIDlet, will demonstrate how to manipulate pointer events to create a simple line-drawing program.

The coordinate system

Our first objective is to become familiar with the coordinate system for working with the device display. The coordinate system for the `Canvas` class originates in the upper-left corner of the display. x values increase moving to the right; y values increase going down. When drawing, the pen thickness is one pixel in width and height. The coordinate system is illustrated in Figure 1.

Figure 1. The coordinate system



Following are the methods to determine the width and height of the canvas:

```
int getWidth()    Canvas width
int getHeight ()  Canvas height
```

The height and width of the `Canvas` also represent the entire drawing area available to the device display. In other words, you do not specify the size of the canvas you would like; the software on an MIDP device will always return the maximum drawing area available for a given device.

Creating a Canvas

The first step to working with `Canvas` class is to createa class that extends it, as shown in the partial code listing below:

```
class TestCanvas extends Canvas implements CommandListener
{
  private Command cmdExit;
  ...

   display = Display.getDisplay(this);
```

```
      cmdExit = new Command("Exit", Command.EXIT, 1);
      addCommand(cmdExit);
      setCommandListener(this);
      ...

      protected void paint(Graphics g)
      {
        // Draw onto the canvas
        ...
      }
   }

   TestCanvas canvas = new TestCanvas(this);
```

Painting on the Canvas

The `Canvas` class method `paint()` lets you draw shapes, display images, write text, and otherwise create what will be shown on the device display. The code listing below shows how you might clear a device display to present a white screen:

```
   protected void paint(Graphics g)
   {
     // Set background color to white
      g.setColor(255, 255, 255);

      // Fill the entire canvas
     g.fillRect(0, 0, getWidth(), getHeight());
   }
```

As you can see, we use a reference to a `Graphics` object inside the `paint()` method to do the actual drawing. We'll have a closer look at the `Graphics` methods in upcoming panels.

Command events

As with the `Form`, `List`, and `TextBox` components discussed in Part 1 (see Resources on page 51) of this tutorial, a`Canvas` can process `Command` events. We process `Canvas Commands` the same way we would those on any other component:

1. Create an instance of a `Command` object.

2. Add the `Command` to the display object (`Canvas`, `Form`, `List`, or `TextBox`).

3. Create a `Command` listener to capture events.

4. Write a `commandAction()` method to process events.

TestCanvas implements CommandListener

The following code listing shows how the `TestCanvas` class might implement a `CommandListener`:

```
class TestCanvas extends Canvas implements CommandListener
{
  private Command cmdExit;
  ...

  display = Display.getDisplay(this);
  cmdExit = new Command("Exit", Command.EXIT, 1);
  addCommand(cmdExit);
  setCommandListener(this);
  ...

  protected void paint(Graphics g)
  {
    // Draw onto the canvas
    ...
  }

  public void commandAction(Command c, Displayable d)
  {
    if (c == cmdExit)
      ...
  }
}
```

Keycodes

In addition to soft-keys for processing commands, a `Canvas` object has access to 12 keycodes. The codes that are guaranteed to be available on any MIDP device are shown below:

```
KEY_NUM0
KEY_NUM1
KEY_NUM2
KEY_NUM3
KEY_NUM4
KEY_NUM5
KEY_NUM6
KEY_NUM7
KEY_NUM8
KEY_NUM9
KEY_STAR
KEY_POUND
```

The five methods for working with keycodes are as follows:

```
void keyPressed(int keyCode)
void keyReleased(int keyCode)
void keyRepeated(int keyCode)
boolean hasRepeatEvents()
String getKeyName(int keyCode)
```

Game actions

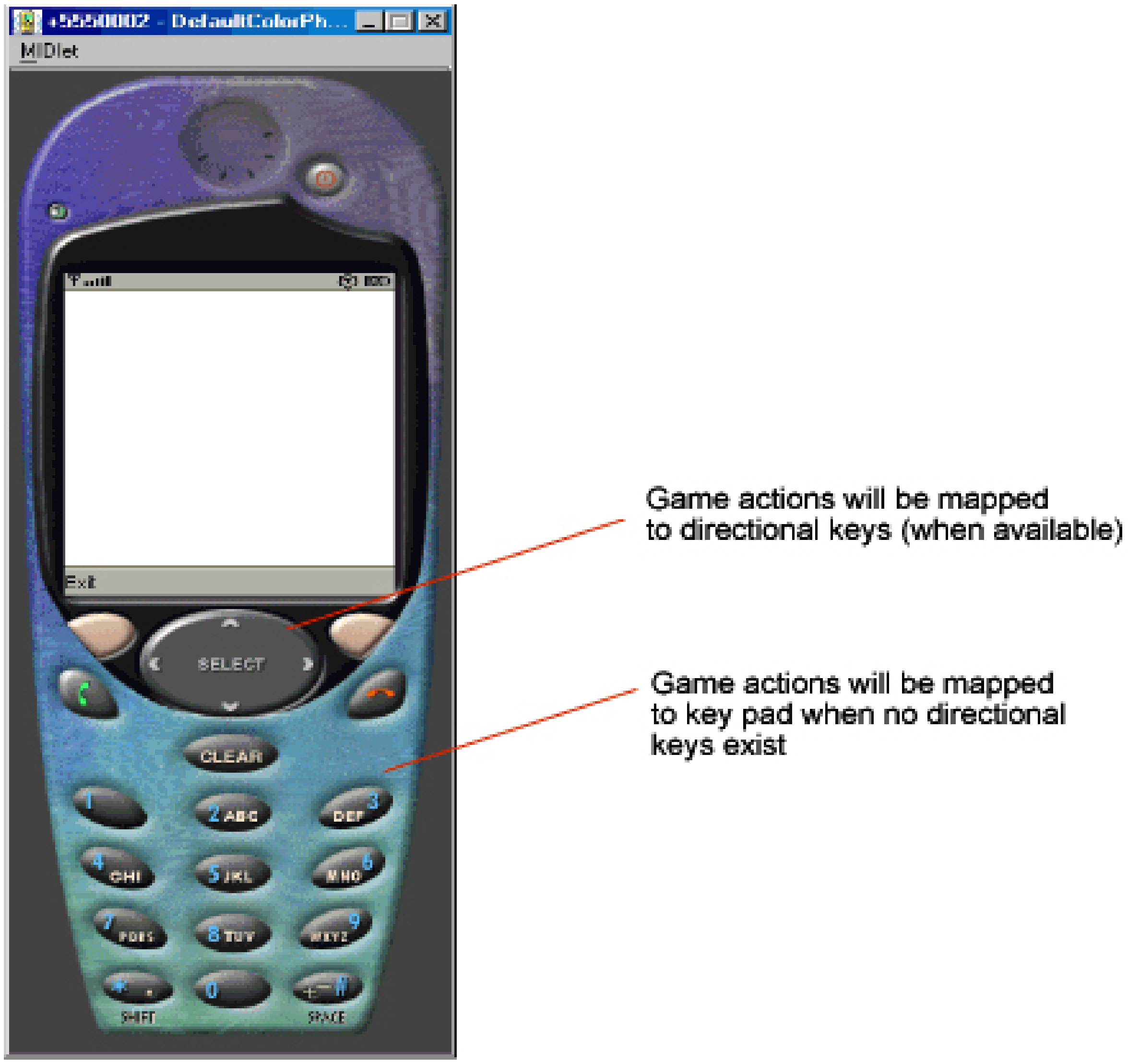
MIDP is often used to create game applications on the Java platform. The following constants have been defined for handling game-related events in MIDP:

```
UP
DOWN
LEFT
RIGHT
FIRE
GAME_A
GAME_B
GAME_C
GAME_D
```

Typically, these values are mapped to the device's directional arrows, but not all mobile devices have these. If a device lacks directional arrows, game actions will be mapped to the keypad (for example, LEFT may be mapped to 2, RIGHT mapped to 5,and so on).

Figure 2 shows how game actions could be mapped on a mobile device based on the availability of directional keys.

Figure 2. Mapping game actions on a mobile device


+5550002 - DefaultColorPh...
MIDlet
Exit
SELECT
CLEAR
Game actions will be mapped to directional keys (when available)
Game actions will be mapped to key pad when no directional keys exist

Detecting game actions

The following code sample shows one way to detect various game actions and call the appropriate methods based on the action selected.

```
protected void keyPressed(int keyCode)
{
  switch (getGameAction(keyCode))
  {
      case Canvas.FIRE:
        shoot();
        break;

      case Canvas.RIGHT:
        goRight();
        break;
       ...
  }
}
```

Another option would be to create a reference for each game action during initialization, then look for the appropriate key at runtime, as shown below:

```
  // Initialization
  keyFire =  getKeyCode(FIRE);
  keyRight = getKeyCode(RIGHT);
  keyLeft = getKeyCode(LEFT);
  ...

  // Runtime
  protected void keyPressed(int keyCode)
  {
    if (keyCode == keyFire)
      shoot();
    else if (keyCode == keyRight)
      goRight()
    ...
  }
```

The KeyMapping MIDlet

We'll build a simple MIDlet to demonstrate some of the `Canvas` class functionality we've discussed so far. The `KeyMapping` MIDlet will use the method `getKeyName()` to display the name of the keycode or game action that has been selected on a device.

You should recall the MIDlet development process from Part 1 of this tutorial, but we'll review it for completeness:

1. Create the project.
2. Write the source code.
3. Compile and preverify the code.
4. Run the MIDlet.

Now, let's build the KeyMapping MIDlet.

Create the project

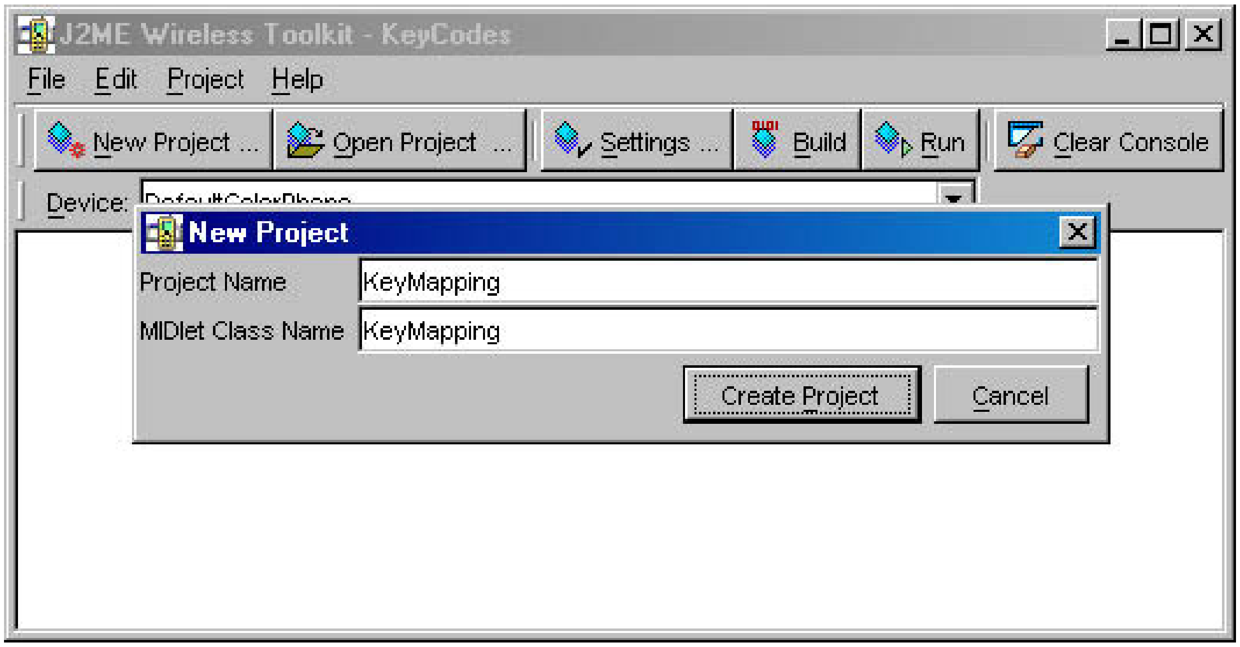
We'll begin by creating a new project in the WTK, as follows:

1. Click **New Project**.
2. Enter the project name and MIDlet class name.

3. Click **Create Project**.

Figure 3 shows the steps to create the KeyMapping project in the WTK.

Figure 3. Creating the KeyMapping project in the WTK



Project Name: KeyMapping
MIDlet Class Name: KeyMapping

Write the code

Study the KeyMapping.java code sample below, and then paste it into a text editor.

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

public class KeyMapping extends MIDlet
{
      private Display  display;       // The display
      private KeyCodeCanvas canvas;   // Canvas

      public KeyMapping()
      {
        display = Display.getDisplay(this);
        canvas  = new KeyCodeCanvas(this);
      }

      protected void startApp()
      {
        display.setCurrent( canvas );
      }

      protected void pauseApp()
      { }

      protected void destroyApp( boolean unconditional )
      { }

      public void exitMIDlet()
      {
        destroyApp(true);
        notifyDestroyed();
      }
}

/*--------------------------------------------------
* Class KeyCodeCanvas
*-------------------------------------------------*/
class KeyCodeCanvas extends Canvas implements CommandListener
{
      private Command cmExit;           // Exit midlet
      private String keyText = null;    // Key code text
      private KeyCodes midlet;

      /*--------------------------------------------------
      * Constructor
      *-------------------------------------------------*/
      public KeyCodeCanvas(KeyCodes midlet)
      {
        this.midlet = midlet;

        // Create exit command and listen for events
        cmExit = new Command("Exit", Command.EXIT, 1);
        addCommand(cmExit);
        setCommandListener(this);
      }

      /*--------------------------------------------------
      * Paint the text representing the key code
      *-------------------------------------------------*/
      protected void paint(Graphics g)
      {
        // Clear the background (to white)
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, getWidth(), getHeight());

        // Set color and draw text
        if (keyText != null)
        {
          // Draw with black pen
          g.setColor(0, 0, 0);

          // Center text
          g.drawString(keyText, getWidth()/2, getHeight()/2,
          Graphics.TOP | Graphics.HCENTER);
        }
```

```
      }

      /*--------------------------------------------------
      * Command event handling
      *-------------------------------------------------*/
      public void commandAction(Command c, Displayable d)
      {
        if (c == cmExit)
          midlet.exitMIDlet();
      }

      /*--------------------------------------------------
      * Key code event handling
      *-------------------------------------------------*/
      protected void keyPressed(int keyCode)
      {
        keyText = getKeyName(keyCode);
        repaint();
      }
}
```

Notes about the code

Two sections of the KeyMapping.java source deserve mention before we move on. First, the method `keyPressed()` saves the name of the key that triggered the event and requests the device to repaint the display, as shown below:

```
protected void keyPressed(int keyCode)
{
  keyText = getKeyName(keyCode);
  repaint();
}
```

Second, when the `repaint` request is processed, the `paint()` method will be called. Inside `paint()` we clear the display, select a black pen for writing to the display, and write the text string we saved previously onto the center of the display, as shown below:

```
protected void paint(Graphics g)
{
  // Clear the background (to white)
  g.setColor(255, 255, 255);
  g.fillRect(0, 0, getWidth(), getHeight());

  // Set color and draw text
  if (keyText != null)
  {
    // Draw with black pen
    g.setColor(0, 0, 0);

    // Center text
```

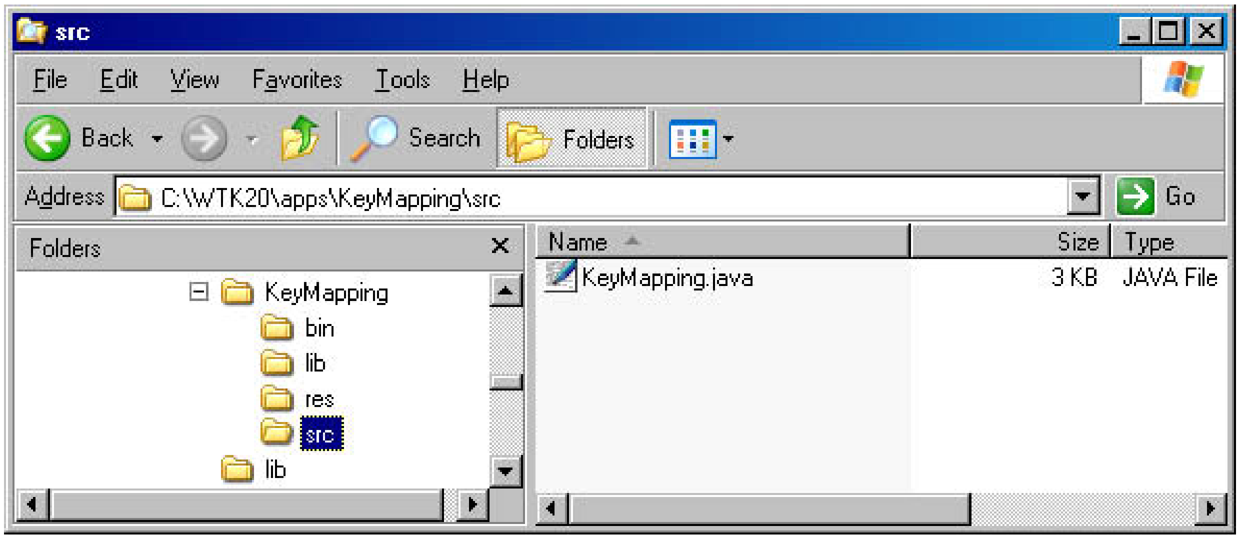
```
      g.drawString(keyText, getWidth()/2, getHeight()/2,
      Graphics.TOP | Graphics.HCENTER);
    }
  }
```

We'll cover the specifics of working with the `Graphics` *object, including the methods shown in the above code example, in the next section.*

Save, compile, and preverify

The next step is to save, compile, and preverify the code. You will recall that when you create a new project the WTK builds the proper directory structure for you. In this example, the WTK has created the *C:\WTK20\apps\KeyMapping* directory and all the necessary subdirectories. Save your Java source file as KeyMapping.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 KeyMapping code

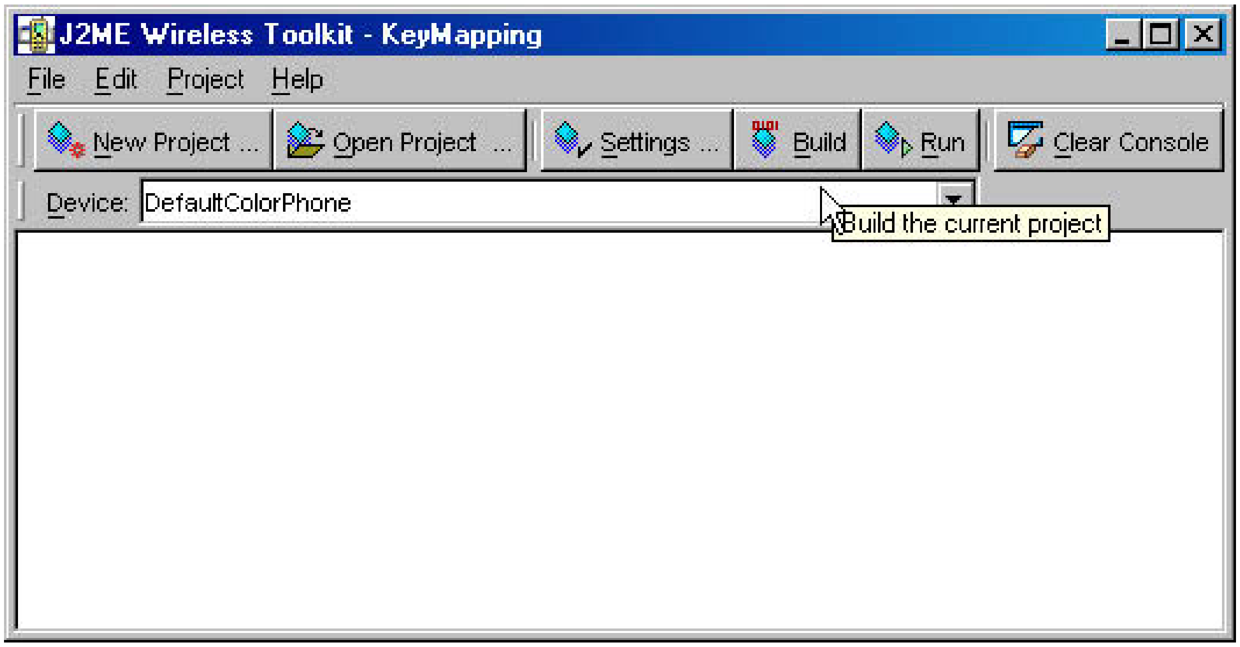


C:\WTK20\apps\KeyMapping\src

Name	Size	Type
KeyMapping.java	3 KB	JAVA File

Next, click **Build** to compile, preverify, and package the MIDlet, as shown in Figure 5.

Figure 5. Build the KeyMapping project

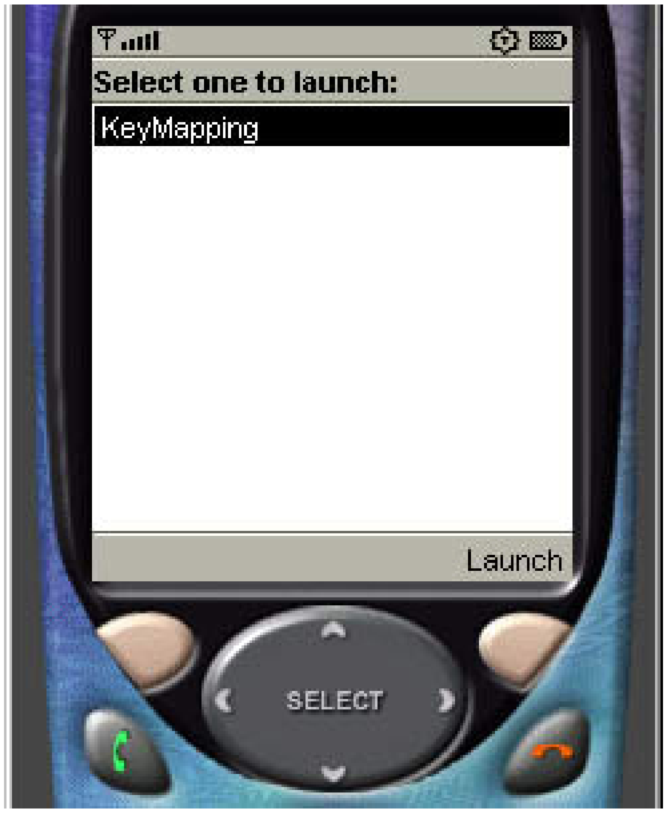


And finally, click **Run** to start the Application Manager.

Start the MIDlet and view its output

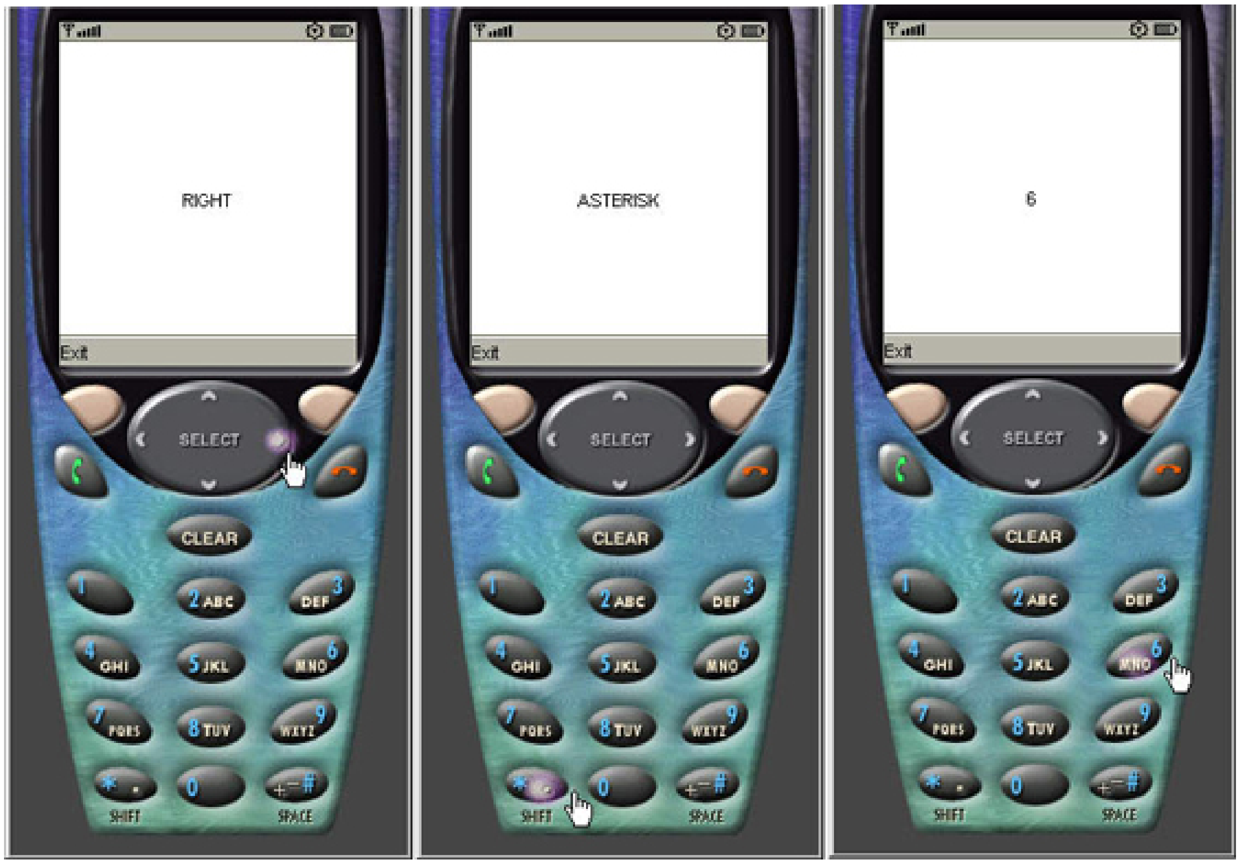
To start the KeyMapping MIDlet, click the **Launch** soft-button as shown in Figure 6.

Figure 6. Run the KeyMapping MIDlet



For each key code and game action entered you'll see the corresponding name centered on the display, as shown in Figure 7.

Figure 7. Output of the KeyMapping MIDlet



Pointer events

The final topic we'll cover in this section is how to manage pointer events in a `Canvas`. These events are designed to facilitate interaction with pointer devices such as a touch pad or mouse. We use the following methods to handle pointer events:

```
boolean hasPointerEvents()
boolean hasPointerMotionEvents()
void pointerPressed(int x, int y)
void pointerReleased(int x, int y)
void pointerDragged(int x, int y)
```

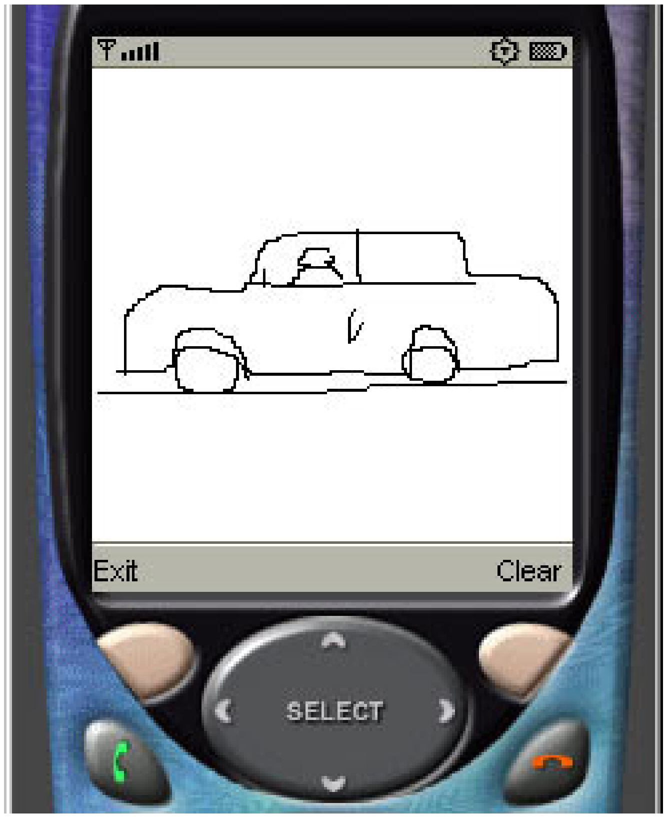
Most of the methods should be self-explanatory. The method `hasPointerMotionEvents()` returns a boolean value indicating whether or not a device supports the concept of "click and drag."

The ScratchPad MIDlet

We'll write a short MIDlet to get you started with pointer events. The ScratchPad

MIDlet is a very simple drawing program. The program recognizes when the pointer (mouse) is clicked down. As the pointer is dragged around the display we'll call the `paint()` method to draw a line from the previous pointer location to the current one. We'll also include a soft-button to allow the user to clear the display. The ScratchPad MIDlet is shown in Figure 8.

Figure 8. The ScratchPad MIDlet



Support setup

Not all devices support pointer events. For example, a typical PDA includes a *stylus* for pointing and dragging on the device display. A mobile phone on the other hand, does not include a pointing device. In order for the WTK to recognize pointer events (clicks of your mouse on the device emulator), some additional configuration is in order. You'll need to locate the properties file for the device emulator type you are using in the WTK. For example, when using the default color phone emulator, the file you are looking for is: *DefaultColorPhone.properties*. Change the `touch_screen` attribute to `true`, as shown below:

```
touch_screen=true
```

With this change in place, when running this MIDlet, your mouse will act as a *style* inside WTK, allowing you to click and drag on the display.

All property files are located in the directory *x:\WTK20\wtklib\devices*.

Building the MIDlet

Here are the steps to build the ScratchPad MIDlet:

1. Create a new project with the name **ScratchPad**.
2. Copy and paste the ScratchPad source code into a text editor.
3. Save each source file in the *\apps\ScratchPad\src* of your WTK installation.
4. Build and run the project.

ScratchPad source

Below is the source code for the ScratchPad MIDlet. After you've completed the steps to build and run the MIDlet on your own, you should be able to view ScratchPad's output in your WTK device emulator.

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

public class ScratchPad extends MIDlet
{
  private Display  display;         // Display object
  private ScratchPadCanvas canvas;  // Canvas

  public ScratchPad()
  {
    display = Display.getDisplay(this);
    canvas  = new ScratchPadCanvas(this);
  }

  protected void startApp()
  {
    display.setCurrent( canvas );
  }

  protected void pauseApp()
  { }

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

```
    public void exitMIDlet()
    {
      destroyApp(true);
      notifyDestroyed();
    }
  }

  /*--------------------------------------------------
  * Class ScratchPadCanvas
  *
  * Pointer event handling
  *-------------------------------------------------*/
  class ScratchPadCanvas extends Canvas implements CommandListener
  {
    private Command cmExit;   // Exit midlet
    private Command cmClear;  // Clear display
    private int startx = 0,   // Where pointer was clicked
                starty = 0,
                currentx = 0, // Current location
                currenty = 0;
    private ScratchPad midlet;
    private boolean clearDisplay = true;

    /*--------------------------------------------------
    * Constructor
    *-------------------------------------------------*/
    public ScratchPadCanvas(ScratchPad midlet)
    {
      this.midlet = midlet;

      // Create exit command and listen for events
      cmExit = new Command("Exit", Command.EXIT, 1);
      cmClear = new Command("Clear", Command.SCREEN, 1);
      addCommand(cmExit);
      addCommand(cmClear);
      setCommandListener(this);
    }

    /*--------------------------------------------------
    * Draw line based on start and ending points
    *-------------------------------------------------*/
    protected void paint(Graphics g)
    {
      // Clear the background (to white)
      if (clearDisplay)
      {
        g.setColor(255, 255, 255);
        g.fillRect(0, 0, getWidth(), getHeight());

        clearDisplay = false;
        startx = currentx = starty = currenty = 0;

        return;
      }

      // Draw with black pen
      g.setColor(0, 0, 0);
      // Draw line
      g.drawLine(startx, starty, currentx, currenty);

      // New starting point is the current position
      startx = currentx;
      starty = currenty;
    }

    /*--------------------------------------------------
    * Command event handling
    *-------------------------------------------------*/
    public void commandAction(Command c, Displayable d)
    {
      if (c == cmExit)
        midlet.exitMIDlet();
      else if (c == cmClear)
      {
        clearDisplay = true;
        repaint();
      }
    }

    /*--------------------------------------------------
    * Pointer pressed
    *-------------------------------------------------*/
    protected void pointerPressed(int x, int y)
    {
      startx = x;
      starty = y;
    }

    /*--------------------------------------------------
    * Pointer moved
    *-------------------------------------------------*/
    protected void pointerDragged(int x, int y)
    {
      currentx = x;
      currenty = y;
      repaint();
    }
  }
```

Summary

The `Canvas` creates the foundation for working directly with the device display. We began this section with an overview of the coordinate system, followed by a discussion on detecting and processing events. We then developed two MIDlets. Together, the example MIDlets demonstrate how you can create a simple line-drawing program by capturing pointer events and drawing onto the display.

Section 4. The Graphics class

Overview

We use `Graphics` objects to draw onto a `Canvas`. In this section you'll learn about each of the following aspects of working with the `Graphics` class:

- Color support
- Stroke style
- Drawing shapes
- Working with fonts
- Anchor points
- Drawing text
- Drawing images
- Additional `Graphics` methods

We'll create three MIDlets together in this section. The DrawShapes MIDlet will allow you to practice drawing arcs and rectangles. The FontViewer MIDlet will introduce you to the basics of working with fonts and anchor points. And the DrawImage MIDlet will demonstrate some of the drawing capabilities of the `Graphics` class.

We'll close the section with a brief overview of two `Graphics` methods that are beyond the scope of this tutorial.

A note about the code examples

As we call various methods in the `Graphics` class we will always include a reference to the appropriate `Graphics` object. For example, in the code below, it is clear where the variable *g* is obtained:

```
protected void paint(Graphics g)
{
  // Clear the background (to white)
  g.setColor(255, 255, 255);
  g.fillRect(0, 0, 20, 20);
  ...
}
```

For brevity, small code examples in this section may look as follows:

```
g.setColor(255, 255, 255);

or
```

```
g.fillRect(0, 0, 20, 20);
```

So, whenever you see a reference to `g.someMethod()`, you can safely assume that *g* refers to a valid `Graphics` object.

Color support

The `Display` object, introduced in Part 1 of this tutorial, will be an important component of the examples that follow in this section, so a quick review is in order. A MIDlet has one instance of a `Display` object. This object is used to obtain information about the current display -- such as the availability of color support -- and includes methods for requesting that objects (`Forms`, `Textboxes`, etc.) be displayed. The `Display` object is essentially the manager of the device display, controlling what is shown on the device. The following two `Display` methods are of interest for our current discussion:

```
boolean isColor()
int numColors()
```

The first method indicates if the device supports color. If yes, the second method can be called to determine the number of colors supported.

Color support methods

The following eight methods provide support for getting colors and setting your color preferences:

```
void setColor(int RGB)
void setColor(int red, int green, int blue)
int getColor()
int getBlueComponent()
int getGreenComponent()
int getRedComponent()
void setGrayScale(int value)
int getGrayScale()
```

Notice that you can specify the color in one of two ways: you can assign an integer to represent all three colors values (red, green, and blue, with eight bits allocated to each color), or you can use separate parameters for each color. When using one value to hold the color, red occupies the uppermost eight bits, green the middle, and blue the lower.

Here's how you might set the color using one integer value:

```
int red   = 0,
    green = 128,
    blue  = 255;

...

g.setColor((red << 16) | (green << 8) | blue);
```

And here's how you can set the same color using three separate parameters:

```
g.setColor(red, green, blue);
```

Stroke style

You can choose the stroke style when drawing lines, arcs, and rectangles on the device display. Here are the methods available for setting the stroke style:

```
int getStrokeStyle()
void setStrokeStyle(int style)
```

The two stroke styles defined in the `Graphics` class are `DOTTED` and `SOLID`, as shown below:

```
g.setStrokeStyle(Graphics.DOTTED);

or

g.setStrokeStyle(Graphics.SOLID);
```

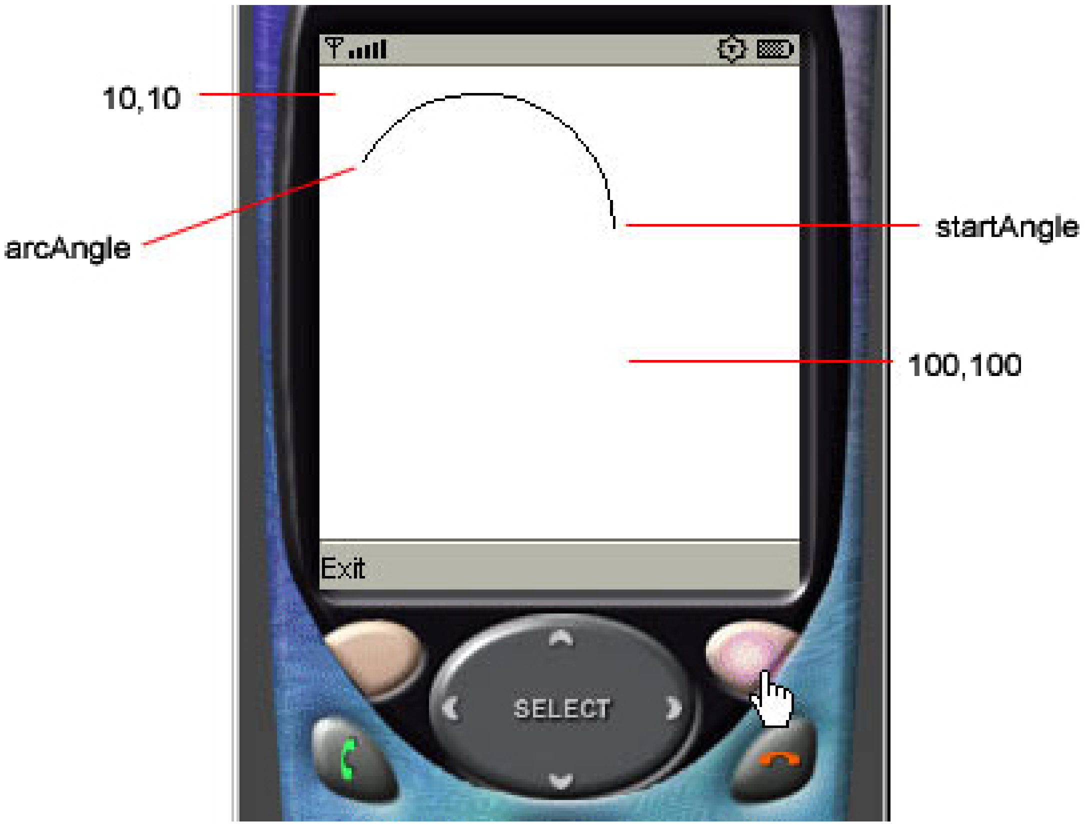
Drawing arcs

When drawing an arc, you can draw it as an outline or request that it be filled. You start by specifying the outside dimensions of an imaginary box (also known as the *bounding box*) in which it will be drawn. The *start angle* determines where the drawing of the arc will begin, with 0 being at 3 o'clock. Positive values go counter-clockwise. The *arc angle* specifies how many degrees to draw from the start angle, going in a counter-clockwise direction. To better understand what all this means, take a look at the following code:

```
g.drawArc(10, 10, 100, 100, 0, 150);
```

The code example requests an arc be drawn with a bounding box located at 10,10, that extends to 100, 100. The start angle is 0, and the arc angle is 150. Figure 9 shows the resulting arc.

Figure 9. An arc drawn to specification



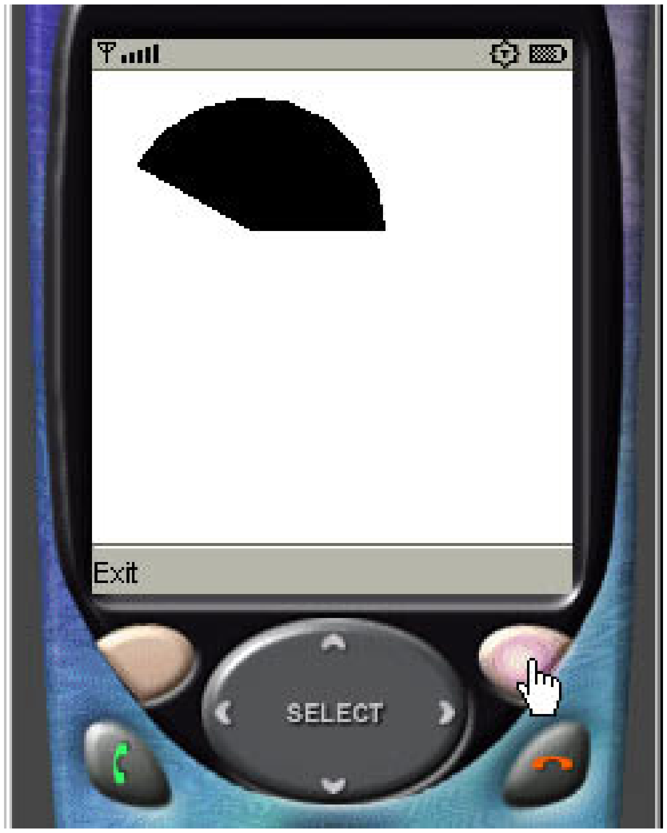
Methods for drawing an arc

Here are the methods for drawing an arc:

```
void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

The arc in Figure 10 is the same arc as in Figure 9; however, it has been constructed using the `fillArc()` method.

Figure 10. A filled arc



The DrawShapes MIDlet

The DrawShapes MIDlet will help you become familiar with drawing arcs. We'll also use this MIDlet to learn about drawing rectangles, simply replacing the calls to draw arcs with those to draw rectangles.

Here are the steps to create the DrawShapes MIDlet:

1. Create a new project with the name **DrawShapes**.
2. Copy and paste the DrawShapes source code into a text editor.
3. Save each source file in the *\apps\DrawShapes\src* directory of your WTK installation.
4. Build and run the project.

DrawShapes source

Here's the source code for our `DrawShapes` MIDlet. Run the MIDlet in your WTK device emulator to see the output.

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

public class DrawShapes extends MIDlet
{
  private Display  display;     // The display
  private ShapesCanvas canvas;  // Canvas

  public DrawShapes()
  {
    display = Display.getDisplay(this);
    canvas  = new ShapesCanvas(this);
  }

  protected void startApp()
  {
    display.setCurrent( canvas );
  }

  protected void pauseApp()
  { }

  protected void destroyApp( boolean unconditional )
  { }

  public void exitMIDlet()
  {
    destroyApp(true);
    notifyDestroyed();
  }
}

/*--------------------------------------------------
* Class ShapesCanvas
*
* Draw arcs
*-------------------------------------------------*/
class ShapesCanvas extends Canvas implements CommandListener
{
  private Command cmExit;  // Exit midlet
  private DrawShapes midlet;

  public ShapesCanvas(DrawShapes midlet)
  {
    this.midlet = midlet;

    // Create exit command and listen for events
    cmExit = new Command("Exit", Command.EXIT, 1);
    addCommand(cmExit);
    setCommandListener(this);
  }

  /*--------------------------------------------------
  * Draw shapes
  *-------------------------------------------------*/
  protected void paint(Graphics g)
  {
    // Clear background to white
    g.setColor(255, 255, 255);
    g.fillRect(0, 0, getWidth(), getHeight());

    // Black pen
    g.setColor(0, 0, 0);

    // Start at 3 o'clock and rotate 150 degrees
    g.drawArc(10, 10, 100, 100, 0, 150);

    // Fill the arc
//    g.fillArc(10, 10, 100, 100, 0, 150);

    // Start at 12 o'clock and rotate 150 degrees
//    g.drawArc(10, 10, 100, 100, 90, 150);

    // Change the size of the bounding box
    // Start at 12 o'clock and rotate 150 degrees
//    g.drawArc(15, 45, 30, 70, 90, 150);
  }

  public void commandAction(Command c, Displayable d)
  {
    if (c == cmExit)
      midlet.exitMIDlet();
  }
}
```

To understand better how `startAngle` and `arcAngle` affect the resulting arc, try changing the parameters passed to `drawArc()` and `fillArc()`.

Drawing rectangles

As with arcs, rectangles can be either outlined or filled. In addition, you can specify that a rectangle have either square or rounded corners.

Following are the methods for drawing a rectangle:

```
void drawRect(int x, int y, int width, int height)
void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

void fillRect(int x, int y, int width, int height)
void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)
```

When drawing rounded corners, you specify the horizontal diameter (`arcWidth`) and the vertical diameter (`arcHeight`). These parameters define the sharpness of the arc in each direction. Larger values represent a more gradual arc; smaller ones tighten the curve.

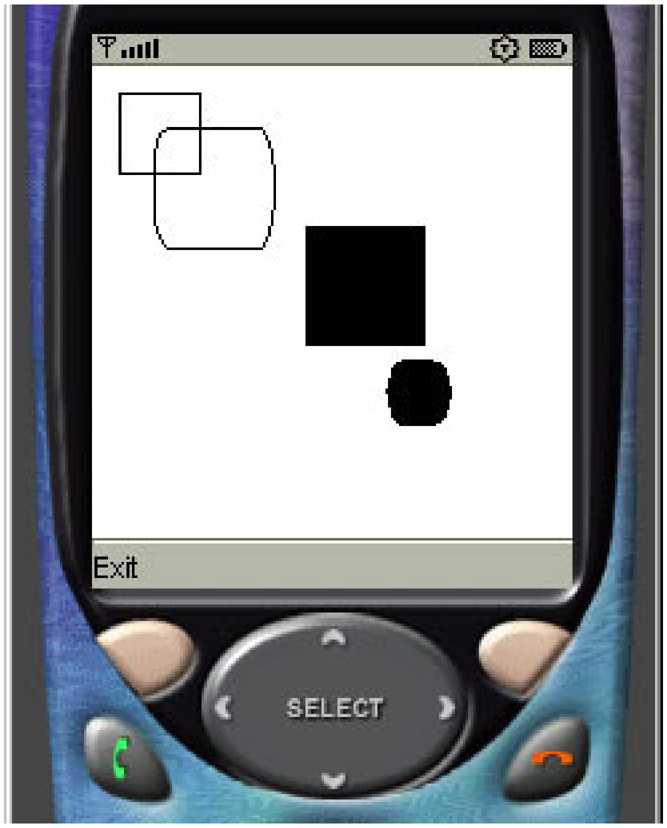
The DrawShapes MIDlet for rectangles

We'll reuse the DrawShapes MIDlet to learn about drawing rectangles. Going back to the DrawShapes source, we simply replace the calls to `drawArc()` with the following:

```
g.drawRect(10, 10, 30, 30);
g.drawRoundRect(23, 23, 45, 45, 10, 35);

g.fillRect(80, 60, 45, 45);
g.fillRoundRect(110, 110, 25, 25, 15, 45);
```

The output for these new specifications is shown in Figure 11.

Figure 11. A couple of rectangles



Working with fonts

It's important that you are familiar with the font support available in MIDP's low-level interface. Following are the methods of the `Font` class:

```
Font getFont(int face, int style, int size)
Font getFont(int fontSpecifier)
Font getDefaultFont()
```

As you can see from the first method, three attributes are associated with a `Font`: *face*, *style*, and *size*. So, for example, you might specify the following `Font` attributes:

```
Font font = Font.getFont(Font.FACE_PROPORTIONAL, Font.STYLE_ITALIC, Font.SIZE_MEDIUM);
```

Your second option is to pass in a font specifier such as `FONT_INPUT_TEXT` or `FONT_STATIC_TEXT`, as shown in the second method above. The former is used to draw system-related content, such as label or button text. The latter refers to text drawn on components in which a user is entering information, such as a `TextField`. Here's an example use of `FONT_INPUT_TEXT`:

```
Font font = Font.getFont(Font.FONT_INPUT_TEXT);
```

The final method for the `Font` class is `getDefaultFont()`, which will return the system-defined font.

Font attributes

The attributes for the `Font` class are listed below:

```
FACE_SYSTEM
FACE_MONOSPACE
FACE_PROPORTIONAL

STYLE_PLAIN
STYLE_BOLD
STYLE_ITALIC
STYLE_UNDERLINED

SIZE_SMALL
SIZE_MEDIUM
SIZE_LARGE
```

Style parameters can be combined using a logical (|) operator, as shown below:

```
Font font = Font.getFont(Font.FACE_PROPORTIONAL,
Font.STYLE_BOLD | Font.STYLE_ITALIC, Font.SIZE_MEDIUM);
```

Getting attributes

After you have a reference to a `Font`, you can query it to determine its attributes. For instance, you might want to find out if a `Font` is bold or italicized, or contains a mono or proportional character set. The code example below shows how to get a variety of `Font` attributes.

```
int getFace()
int getStyle()
int getSize()
boolean isPlain()
boolean isBold()
boolean isItalic()
boolean isUnderlined()
```

Font metrics

The metrics of a font describe measurements such as the font height and the length-in-pixels of a string in a specified font. It is beyond the scope of this tutorial to explain the specifics of each method. However, if you have worked with fonts in J2SE, the primary difference is the limited amount of metric information you have access to. The code sample below shows the limited set of methods available for working with fonts:

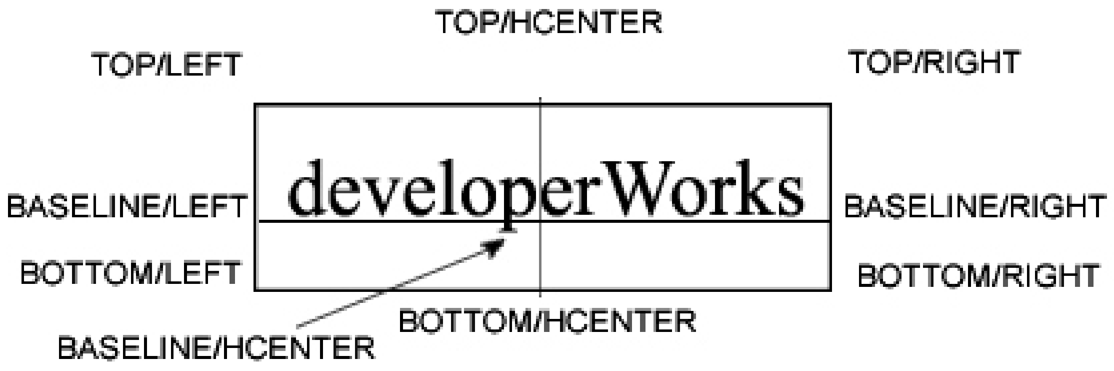
```
int getHeight()
int getBaselinePosition()
int charWidth(char ch)
int charsWidth(char[] ch, int offset, int length)
int stringWidth(String str)
int substringWidth(String str, int offset, int length)
```

Working with anchor points

The last stopping point before drawing text is to introduce the concept of *anchor points*. In addition to specifying the x and y coordinates of where to display a string of characters, anchor points allows you to specify where on the bounding box surrounding the text you wish to place the x and y coordinates.

The easiest way to grasp this idea is to begin by picturing a bounding box around the text string you plan to draw. Figure 12 shows the bounding box around the text string for the word *developerWorks*.

Figure 12. developerWorks bounding box



There are six defined anchor points, three horizontal and three vertical. When specifying the anchor point for drawing text, anchor points are used in pairs, so you must select one horizontal and one vertical point. The available anchor points are listed below:

Horizontal

LEFT

HCENTER

RIGHT

Vertical

TOP

BASELINE

BOTTOM

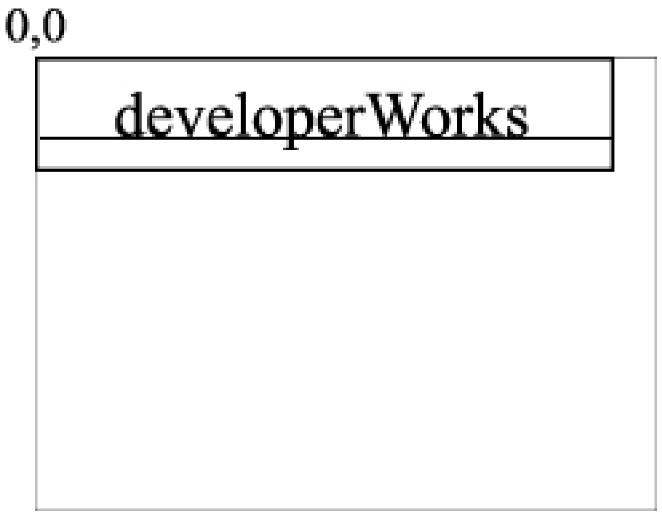
Some working examples

When using anchor points, you are stating which location on the bounding box is located at the specified x and y coordinates. For example, with the code below, we are specifying that the `TOP/LEFT` corner of the bounding box be positioned at 0,0:

```
g.drawString("developerWorks", 0, 0 , Graphics.TOP | Graphics.LEFT);
```

Figure 13 shows the results of this specification. Note that the light gray box represents the device display.

Figure 13. Output of a TOP/LEFT anchor point specification

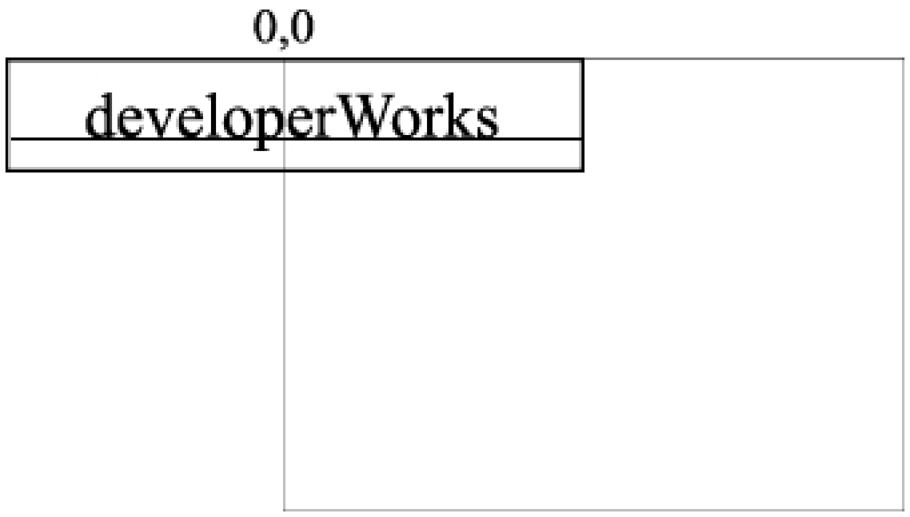


By changing only the anchor point, we can change where the text will be displayed on the device. In the next example, see what happens when we specify that the `TOP/HCENTER` of the bounding box be located at 0,0, as shown below:

```
g.drawString("developerWorks", 0, 0 , Graphics.TOP | Graphics.HCENTER);
```

Here's the output of the new specification.

Figure 14. Output of a TOP/HCENTER anchor point specification



Drawing text

Now that you know about fonts and anchor points, you're ready to learn about the methods for drawing text. The methods are as follows:

```
void drawChar(char character, int x, int y, int anchor)
void drawChars(char[] data, int offset, int length, int x, int y, int anchor)
void drawString(String str, int x, int y, int anchor)
void drawSubstring(String str, int offset, int len, int x, int y, int anchor)
```

A working example

When drawing text, you can either use the default font or specify a particular one. If you want to specify the font, you must do so before you begin drawing the text. For example, the `paint()` method below requests the system type face, in italics, with a size of medium. Note also that the text is centered on the display by specifying the x and y coordinates at the center of the display, and choosing the anchor point that represents the center of the text string (bounding box).

```
protected void paint(Graphics g)
{
  // Get center of display
  int xcenter = getWidth() / 2,
      ycenter = getHeight() / 2;

  // Choose a font
  g.setFont(Font.getFont(Font.FACE_SYSTEM, Font.STYLE_ITALICS, Font.SIZE_MEDIUM));

  // Specify the center of the text (bounding box) using the anchor point
  g.drawString("developerWorks", xcenter, ycenter, Graphics.BASELINE | Graphics.HCENTER);
}
```

The FontViewer MIDlet

The FontViewer MIDlet will combine several of the topics we've covered up to this point. The MIDlet includes a font viewer that allows the user to select various font attributes (face, style, and size) and display a text string with the chosen attributes. We'll use the `TextField` and `ChoiceGroup` components from Part I of this tutorial to create the selection mechanism.

Here are the steps to create the FontViewer MIDlet:

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

2. Copy and paste each of the three source code files below into a text editor.

3. Save each source file in the *\apps\FontViewer\src* directory of your WTK installation.

4. Build and run the project.

FontViewer source

Here's the source code for the FontViewer MIDlet.

```
/*--------------------------------------------------
* FontViewer.java
*
* Show the various font attributes (face, size,
* style) on a Canvas
*-------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class FontViewer extends MIDlet
{
  protected Display  display;     // The display
  protected PrefsForm fmPrefs;    // Form to choose font prefs
  protected FontCanvas cvFont;    // Canvas to display text (in preferred font)

  public FontViewer()
  {
    display = Display.getDisplay(this);
    cvFont = new FontCanvas(this);
    fmPrefs = new PrefsForm("Preferences", this);
  }

  protected void startApp()
  {
    showCanvas();
  }

  protected void showCanvas()
  {
    display.setCurrent(cvFont);
  }

  protected void pauseApp()
  { }

  protected void destroyApp( boolean unconditional )
  { }

  public void exitMIDlet()
  {
    destroyApp(true);
    notifyDestroyed();
  }
}


/*--------------------------------------------------
```

```
* FontCanvas.java
*
* Draw text using specified Font
*-------------------------------------------------*/
import javax.microedition.lcdui.*;

class FontCanvas extends Canvas implements CommandListener
{
  private int face,           // Font face
              style,          //      style
              size;           //      size
  private String text = "developerWorks";   // Text to display in preferred font
  private Command cmExit;     // Exit midlet
  private Command cmPrefs;    // Call the preferences form
  private FontViewer midlet;  // Reference to the main midlet

  public FontCanvas(FontViewer midlet)
  {
    this.midlet = midlet;

    // Create commands  and listen for events
    cmExit = new Command("Exit", Command.EXIT, 1);
    cmPrefs = new Command("Prefs", Command.SCREEN, 2);
    addCommand(cmExit);
    addCommand(cmPrefs);
    setCommandListener(this);
  }

  /*-------------------------------------------------
  * Draw text
  *-------------------------------------------------*/
  protected void paint(Graphics g)
  {
    // Clear the display
    g.setColor(255, 255, 255);      // White pen
    g.fillRect(0, 0, getWidth(), getHeight());
    g.setColor(0, 0, 0);            // Black pen

    // Use the user selected font preferences
           g.setFont(Font.getFont(face, style, size));

    // Draw text at center of display
    g.drawString(text, getWidth() / 2, getHeight() / 2,
    Graphics.BASELINE | Graphics.HCENTER);
  }

  protected void setFace(int face)
  {
    this.face = face;
  }

  protected void setStyle(int style)
  {
    this.style = style;
  }

  protected void setSize(int size)
  {
    this.size = size;
  }

  public void setText(String text)
  {
    this.text = text;
  }

  public int getFace()
  {
    return face;
  }

  public int getStyle()
  {
    return style;
  }

  public int getSize()
  {
    return size;
  }

  public void commandAction(Command c, Displayable d)
  {
    if (c == cmExit)
      midlet.exitMIDlet();
    else if (c == cmPrefs)
      midlet.display.setCurrent(midlet.fmPrefs);
  }
}


/*--------------------------------------------------
* PrefsForm.java
*
* Form that allows user to select font preferences
*-------------------------------------------------*/
import javax.microedition.lcdui.*;

public class PrefsForm extends Form implements
CommandListener, ItemStateListener
{
  private FontViewer midlet;      // Main midlet
  private Command cmBack,         // Back to canvas
                    cmSave;         // Save new font prefs
  protected ChoiceGroup cgFace,   // Font face
                  cgStyle,        // style
                  cgSize;         // size
  protected TextField tfText;     // Text string to display on canvas
                                  // (with the requested font prefs)
              private int face = 0,           // Values for font attributes
              style = 0,          // that are obtained from the choicegroups
                          size = 0;
  private String text = null;     // Value for text string from the textfield
```

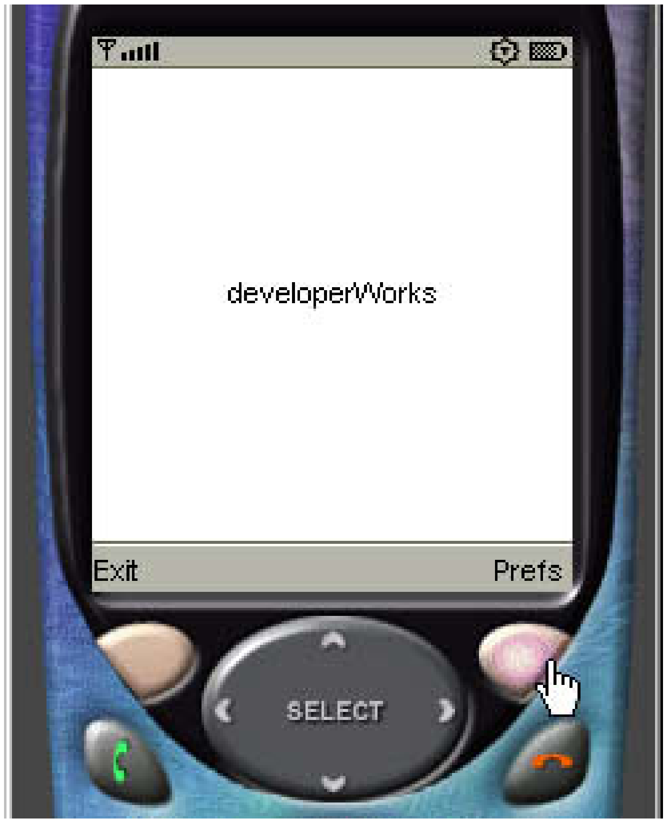
```
public PrefsForm(String title, FontViewer midlet)
{
  // Call the form constructor
  super(title);

  // Save reference to MIDlet so we can
  // access the display manager class
  this.midlet = midlet;

  // Commands
  cmSave = new Command("Save", Command.SCREEN, 1);
  cmBack = new Command("Back", Command.BACK, 2);

    // Exclusive choice group for font face
  cgFace = new ChoiceGroup("Face Options", Choice.EXCLUSIVE);
  cgFace.append("System", null);
  cgFace.append("Monospace", null);
  cgFace.append("Proportional", null);

  // Multiple choice group for font style
  cgStyle = new ChoiceGroup("Style Options", Choice.MULTIPLE);
  cgStyle.append("Plain", null);
  cgStyle.append("Bold", null);
  cgStyle.append("Italic", null);
  cgStyle.append("Underlined", null);

  // Exclusive choice group for font size
  cgSize = new ChoiceGroup("Size Options", Choice.EXCLUSIVE);
  cgSize.append("Small", null);
  cgSize.append("Medium", null);
  cgSize.append("Large", null);

  // Textfield for text to display on canvas
  tfText = new TextField("", "developerWorks", 20, TextField.ANY);

  // Create form contents and listen for events
  append(cgFace);
  append(cgStyle);
  append(cgSize);
  append(tfText);
  addCommand(cmSave);
  addCommand(cmBack);
  setCommandListener(this);
  setItemStateListener(this);
}

/*--------------------------------------------------
* Command event handling
*-------------------------------------------------*/
public void commandAction(Command c, Displayable s)
{
  if (c == cmSave)
  {
    // Set the values in canvas class to those selected here
    midlet.cvFont.setFace(face);
    midlet.cvFont.setStyle(style);
    midlet.cvFont.setSize(size);
```

```
      // Make sure we aren't passing a null value
      midlet.cvFont.setText(tfText.getString() !=
      null ? tfText.getString() : "developerWorks");
    }

    // No specific check needed for "Back" command
    // Always return to the canvas at this point
    midlet.showCanvas();
  }

  public void itemStateChanged(Item item)
  {
    if (item == cgFace)
    {
            switch (cgFace.getSelectedIndex())
            {
        case 0:
          face = Font.FACE_SYSTEM;
          break;
        case 1:
          face = Font.FACE_MONOSPACE;
          break;
        case 2:
          face = Font.FACE_PROPORTIONAL;
          break;
      }
    }
    else if (item == cgStyle)
    {
      boolean[] b = new boolean[4];
      cgStyle.getSelectedFlags(b);

      style = 0;

      // STYLE_PLAIN has a value of 0
      //   No need to check for b[0]

      // If bold selected
            if (b[1])
        style = Font.STYLE_BOLD;

      // If italic selected
            if (b[2])
        style |= Font.STYLE_ITALIC;

      // If underlined selected
            if (b[3])
        style |= Font.STYLE_UNDERLINED;
    }
    else if (item == cgSize)
    {
            switch (cgSize.getSelectedIndex())
            {
        case 0:
          size = Font.SIZE_SMALL;
          break;
        case 1:
          size = Font.SIZE_MEDIUM;
          break;
        case 2:
          size = Font.SIZE_LARGE;
          break;
      }
    }
    else if (item == tfText)
    {
      text = tfText.getString();
    }
  }
}
```

FontViewer output

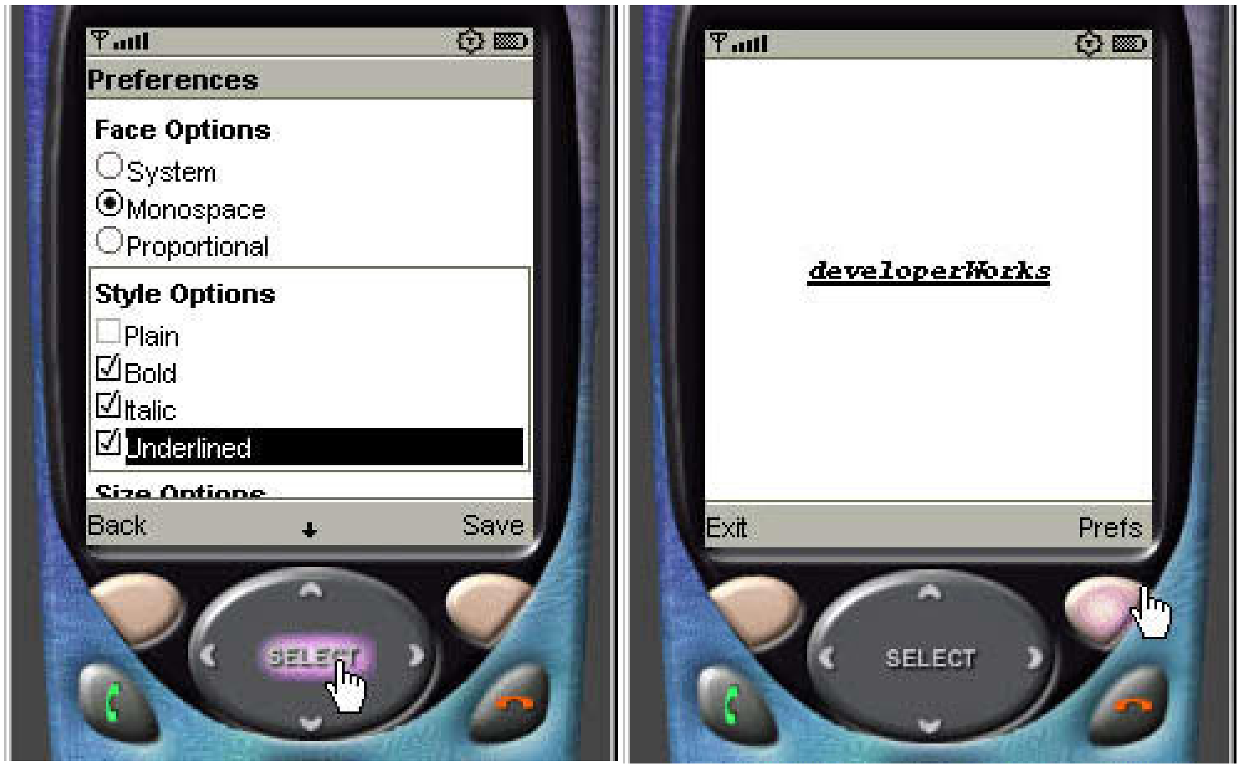
The first screenshot below shows the main user interface of the FontViewer MIDlet. A text string is displayed, and a soft-button labeled **Prefs** allows the user to select the font attributes.

Figure 15. The FontViewer MIDlet


developerWorks

Upon clicking **Prefs** the user is presented with various font settings, as shown in the left screenshot of Figure 16. The right screenshot shows the updated text string.

Figure 16. FontViewer preferences



Preferences

Face Options

○ System
● Monospace
○ Proportional

Style Options

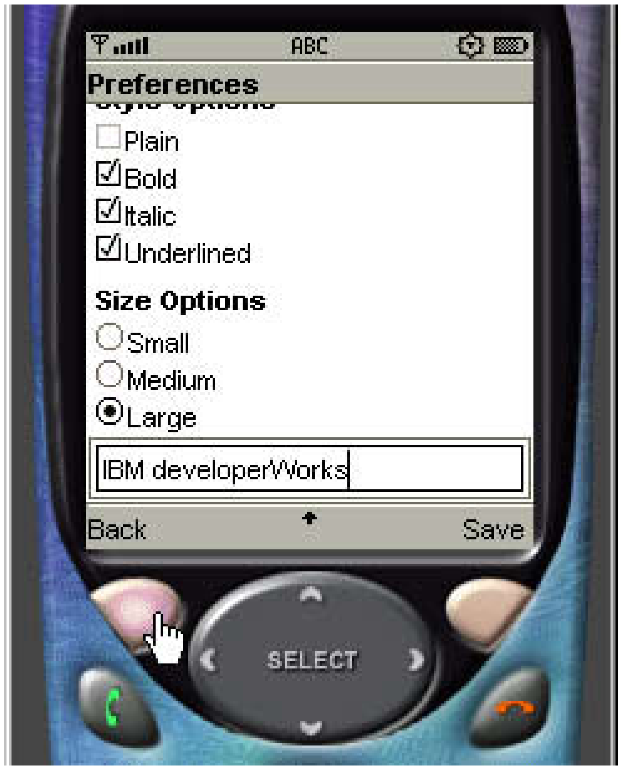
☐ Plain
☑ Bold
☑ Italic
☑ Underlined

Size Options

developerWorks

In the above screenshot you see two`ChoiceGroup`s: an exclusive `ChoiceGroup` for font selection and a multi-selection `ChoiceGroup` for style selection. In addition, if you return to the Preferences and scroll down you'll see two more options. One is an exclusive `ChoiceGroup` to specify the font size. The second is a `TextField` that lets you change the text string that will be displayed with the chosen font attributes, as shown in Figure 17.

Figure 17. FontViewer preferences



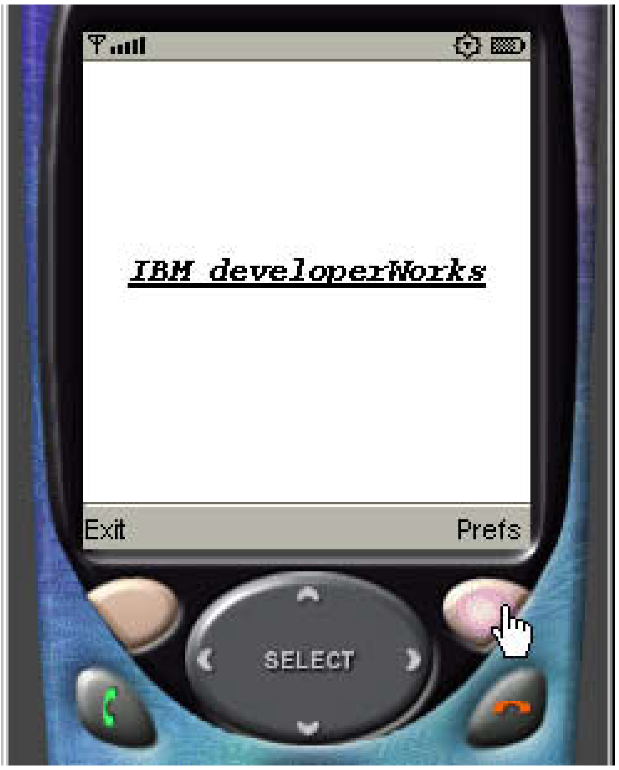
Preferences

☐ Plain
☑ Bold
☑ Italic
☑ Underlined

Size Options

○ Small
○ Medium
● Large

IBM developerWorks



IBM developerWorks

Drawing images

The `Graphics` class includes one method for drawing images:

```
drawImage(Image img, int x, int y, int anchor)
```

We follow the same steps for drawing an image as we do to draw text. For both, we start by establishing the x and y coordinate as well as the anchor points. The anchor list for images is slightly different than the one for text, however. Unlike text, an image has a center point. Therefore, `VCENTER` replaces `BASELINE` when working with images. The image anchor points are shown below:

Horizontal

LEFT

HCENTER

RIGHT

Vertical

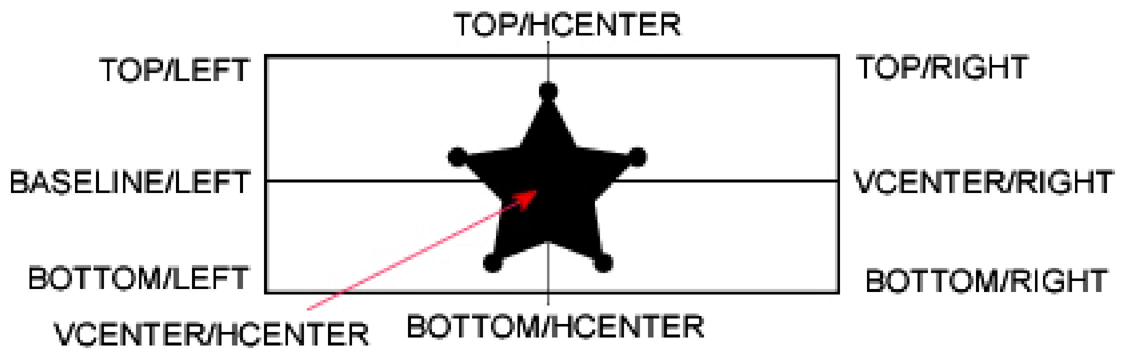
TOP

VCENTER

BOTTOM

Figure 18 shows the location of the anchor points when working with images.

Figure 18. Image anchor points



The DrawImage MIDlet

In Part 1 of this tutorial we created a MIDlet that displayed an image read from a resource (a file). This type of image cannot be altered, and is therefore known as *immutable*. For the `DrawImage` MIDlet, we'll create an image from scratch; that is, we'll allocate memory for the image, obtain reference to a `Graphics` object, and draw the image contents ourselves. This type of image is known as *mutable*.

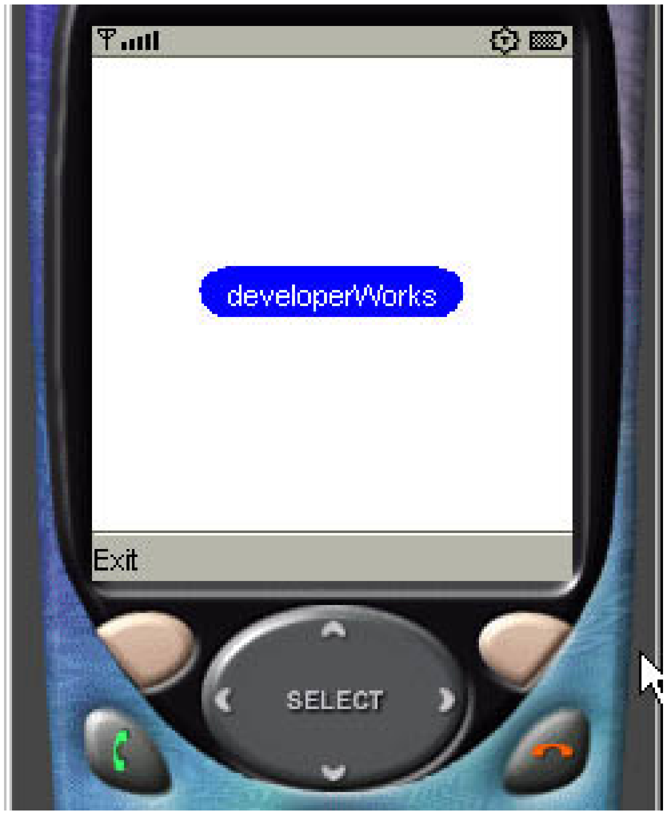
Here are the steps to create the `DrawImage` MIDlet:

1. Create a new project with the name **DrawImage**.
2. Copy and paste the source code below into a text editor.
3. Save the source file in the *\apps\DrawImage\src* directory of your WTK installation.
4. Build and run the project.

Creating a mutable image

Figure 19 shows the image we'll create for this exercise: a rounded-corner rectangle with a blue background. We'll draw the text in white, using the system font, and center the image on the device display.

Figure 19. A mutable image



DrawImage source

Here's the source code for the DrawImage MIDlet. Study it carefully, then run it to view the output for yourself.

```
/*--------------------------------------------------
* DrawImage.java
*
* Draw mutable image on a canvas
*-------------------------------------------------*/
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class DrawImage extends MIDlet
{
  private Display  display;     // The display
  private ImageCanvas canvas;   // Canvas

  public DrawImage()
  {
    display = Display.getDisplay(this);
    canvas  = new ImageCanvas(this);
  }

  protected void startApp()
  {
    display.setCurrent( canvas );
  }

  protected void pauseApp()
  { }

  protected void destroyApp( boolean unconditional )
  { }

  public void exitMIDlet()
  {
    destroyApp(true);
    notifyDestroyed();
  }
}

/*--------------------------------------------------
* Class ImageCanvas
*
* Draw mutable image
*-------------------------------------------------*/
class ImageCanvas extends Canvas implements CommandListener
{
  private Command cmExit;  // Exit midlet
  private DrawImage midlet;
  private Image im = null;
  private String message = "developerWorks";

  public ImageCanvas(DrawImage midlet)
  {
    this.midlet = midlet;

    // Create exit command and listen for events
    cmExit = new Command("Exit", Command.EXIT, 1);
    addCommand(cmExit);
    setCommandListener(this);

    try
    {
      // Create mutable image
      im = Image.createImage(100, 20);

      // Get graphics object to draw onto the image
      Graphics graphics = im.getGraphics();

      // Specify a font face, style and size
      Font font = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_PLAIN, Font.SIZE_MEDIUM);
           graphics.setFont(font);

      // Draw a filled (blue) rectangle, with rounded corners
      graphics.setColor(0, 0, 255);
          graphics.fillRoundRect(0,0, im.getWidth()-1, im.getHeight()-1, 20, 20);
```

```
              // Center text horizontally in the image. Draw text in white
        graphics.setColor(255, 255, 255);

        graphics.drawString(message,
                    (im.getWidth() / 2) - (font.stringWidth(message) / 2),
                    (im.getHeight() / 2) - (font.getHeight() / 2),
                     Graphics.TOP | Graphics.LEFT);
      }
      catch (Exception e)
      {
        System.err.println("Error during image creation");
      }
    }

    /*--------------------------------------------------
    * Draw mutable image
    *-------------------------------------------------*/
    protected void paint(Graphics g)
    {
      // Clear the display
      g.setColor(255, 255, 255);
      g.fillRect(0, 0, getWidth(), getHeight());

      // Center the image on the display
      if (im != null)
        g.drawImage(im, getWidth() / 2, getHeight() / 2,
        Graphics.VCENTER | Graphics.HCENTER);
    }

    public void commandAction(Command c, Displayable d)
    {
      if (c == cmExit)
        midlet.exitMIDlet();
    }
  }
```

Additional Graphics methods

`clip()` and `translate()` are two `Graphics` methods that deserve a brief mention before we close this section. A *clipping region* defines the area of the device display that will be updated during a repaint operation. Following are the clipping methods:

```
void setClip(int x, int y, int width, int height)
void clipRect(int x, int y, int width, int height)
int getClipX()
int getClipY()
int getClipWidth()
int getClipHeight()
```

`translate()` is a method used in relation to the coordinate system. Up to this point, we have assumed that the coordinate 0,0 was always the upper-left corner of the device display. In fact, the coordinate system can be translated so that the origin is in a different x,y location, as shown below:

```
void translate(int x, int y)
int getTranslateX()
int getTranslateY()
```

Both `clip()` and `translate()` are commonly used in graphics-intensive development projects, and are thus somewhat more advanced methods than the others discussed in this tutorial.

Summary

In this section you've learned the fundamentals of working with the `Graphics` class. You now know how to work with colors and fonts, and draw shapes, text, and images. You've also learned a little bit about the technique of clipping, as well as how to translate the coordinate system for greater control of the device display.

From here, obviously, the thing to do is practice what you've learned. To provide incentive, we'll close the tutorial with a brief discussion of one of the most exciting features of MIDP 2.0: the Game API.

Section 5. The Game API

Overview

The Game API, introduced with MIDP 2.0, facilitates the development of games with rich graphical content. The Game API is extensive enough that it could be the subject of an entire tutorial all on its own, so here we'll just get an idea of what the API has to offer. Each of the classes summarized in this section is part of the `javax.microedition.lcdui.game` package.

GameCanvas

Like the now-familiar `Canvas` class, `GameCanvas` provides the foundation for creating a user interface, but in this case one just for games. `GameCanvas` contains a separate off-screen buffer for each class instance and provides a built-in means to query the state of game keys. Here's the `GameCanvas` class:

```
public abstract class GameCanvas extends Canvas
```

Layer and Sprite

`Layer` is an abstract class used to represent a visual object in a game. `Sprite` is a subclass of `Layer`, which is provided to represent an image. Oftentimes a `Sprite` will consist of a series of frames. To perform animation, the frames are shown in sequence to create the effect of a moving image. Transformations, such as rotating or flipping, can also be applied to a `Sprite`. Here are the `Layer` and `Sprite` classes.

```
public abstract class Layer extends Object

public class Sprite extends Layer
```

TiledLayer

A `TiledLayer` is analogous to a spreadsheet, where each cell represents an image. A `TiledLayer` is typically used to represent large visual elements, such as a game background.

```
public class TiledLayer extends Layer
```

LayerManager

To simplify the process of drawing multiple layers in a game, the Game API includes a `LayerManager`. The `LayerManager` maintains an ordered list of `Layers` and determines which areas of each one need to be drawn, and does so in the proper order. The `LayerManager` class is shown below:

```
public class LayerManager extends Object
```

Section 6. Wrap-up and resources

Summary

The *J2ME 101* tutorial series has presented a comprehensive overview of J2ME and MIDP. In the first half of the tutorial you learned about the components of the high-level API. In this second half of the tutorial you've learned about the components of the low-level API. We've covered the basics of working with the `Canvas` and `Graphics` classes. I've introduced the essential methods for each class and together we've built various MIDlets to demonstrate the functionality of each component. We've also briefly discussed the Game API, which is one of the most exciting APIs to work with on the J2ME platform.

The J2ME 101 series will continue with two companion articles. The first article will focus on the important issue of data storage and management in J2ME mobile application development; the second will be about networking support on the J2ME platform. (See Resources on page 51 for links to the articles when they become available.)

You currently know enough about the various components of the J2ME platform and MIDP to begin developing your own simple applications. The upcoming articles will set you on the path to more complex development projects. In the meantime, I suggest you continue to practice what you've learned so far.

Resources

- Don't miss the other content in the *J2ME 101* series:
  - "*J2ME 101, Part 1: Introduction to MIDP's high-level UI*" (*developerWorks*, November 2003)
  - "*J2ME 101, Part 3: Inside the Record Management System*" (*developerWorks*, December 2003)
  - "*J2ME 101, Part 4: The Generic Connection Framework*" (*developerWorks*, January 2004)
- Download the *JDK version 1.4.1* (http://java.sun.com/products/jdk/1.4) .
- Also see the *IBM developer kits for the Java platform* page.
- Download the *J2ME Wireless Toolkit version 2.0* (http://java.sun.com/products/j2mewtoolkit) .

- 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.
- Explore the power of the JAR file format in "*JAR files revealed*" (*developerWorks*, October 2003).
- Continue practicing what you've learned here with the tutorial "*Implementing Push technology with J2ME and MIDP*" (*developerWorks*, June 2003).
- *MicroDevNet* (http://www.microjava.com/) is a good place to read, search, and interact with other J2ME developers.
- Learn about some of the security challenges of MIDP development with "*Securing wireless J2ME*" (*developerWorks*, June 2002).
- The Java section at *Forum Nokia* (http://www.forum.nokia.com/main/1,6566,1_0,00.html) has many whitepapers for those interested in working with J2ME.
- The *KVM mailing list* is an excellent resource for J2ME developers, offering a searchable archive and an active e-mail discussion area.
- Sun Microsystems *CLDC and MIDP wireless forum* lets you post questions and search for J2ME-related information.
- The *WebSphere Micro Environment* (http://www-3.ibm.com/software/wireless/wme/) 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* (http://www.alphaworks.ibm.com/tech/wstkmd) provides tools and a runtime 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* (http://www.corej2me.com) for additional articles, tutorials, and developer resources.
- The *developerWorks* *Wireless zone* (http://www-106.ibm.com/developerworks/wireless) offers a wealth of technical content on pervasive computing.

° You'll find hundreds of articles about every aspect of Java programming in the developerWorks *Java technology zone* (http://www-106.ibm.com/developerworks/java/) .

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

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .