

BRIEF INTRODUCTION TO MIDP PROGRAMMING

Version 1.0

18 Nov 02

Table of Contents

1. INTRODUCTION	4
1.1 PURPOSE	4
1.2 REFERENCES	4
2. MIDLET BASICS.....	5
2.1 MIDLET	5
2.2 MIDLET LIFECYCLE	5
2.3 MIDP USER INTERFACE APIs.....	6
2.4 MIDLET SUITES AND APPLICATION DESCRIPTORS.....	7
3. A FIRST EXAMPLE: HELLOWORLDMIDLET	9
3.1 DESIGN	9
3.2 HELLOWORLDMIDLET.JAVA.....	9
3.3 HELLOSCREEN.JAVA	10
4. A SECOND EXAMPLE: BOUNCINGTEXTMIDLET	12
4.1 DESIGN	12
4.1.1 Overview	12
4.1.2 MIDlet: BouncingTextMIDlet	12
4.1.3 First Screen: TextInputScreen	12
4.1.4 Second Screen: BouncingTextCanvas.....	13
4.2 BOUNCINGTEXTMIDLET.JAVA	14
4.3 TEXTINPUTSCREEN.JAVA.....	16
4.4 BOUNCINGTEXTCANVAS.JAVA.....	17

Change history

13 Mar 02	Version 0.9.3	Document published in Forum Nokia.
18 Nov 02	Version 1.0	Document updated.

Disclaimer:

The information in this document is provided "as is," with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

The phone UI images shown in this document are for illustrative purposes and do not represent any real device.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc.

Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

License:

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

1. INTRODUCTION

1.1 Purpose

The J2ME™ Mobile Information Device Profile (MIDP) and Connected Limited Device Configuration (CLDC) are defined in **[MIDPSPEC]** and **[CLDCSPEC]**. These specifications define the foundation for writing MIDP applications, which are called MIDlets.

The following document introduces MIDP programming through two simple example MIDlets.

1.2 References

- | | |
|-----------------|---|
| MIDPSPEC | <i>Mobile Information Device (JSR-37), JSR Specification, Java 2 Platform, Micro Edition, 1.0a, 15 Dec 2000, http://java.sun.com/products/midp/</i> |
| CLDCSPEC | <i>Connected, Limited Device Configuration (JSR-30), JSR Specification, Java 2 Platform, Micro Edition, Version 1.0, 19 May 2000, http://java.sun.com/products/cldc/</i> |

2. MIDLET BASICS

2.1 MIDlet

A MIDlet is an application written for the Java 2 Micro Edition (J2ME) Mobile Information Device Profile (MIDP).

The MIDP specification [**MIDPSPEC**] defines certain minimum hardware, software, networking, and application management requirements. It also specifies standard system APIs for devices that support MIDP. MIDP is intended for use on connected devices that have a limited CPU, memory, keyboard, and display capabilities, such as cell phones and pager devices. The MIDP specification also defines support for HTTP client networking capabilities.

MIDP applications are co-resident with other applications and functions in the Mobile Information Device (MID). The MIDlet lifecycle helps MIDlets coexist with other applications on a Mobile Information Device. This document focuses on developing MIDP Applications (MIDlets), which are shown in boldface in the following figure.

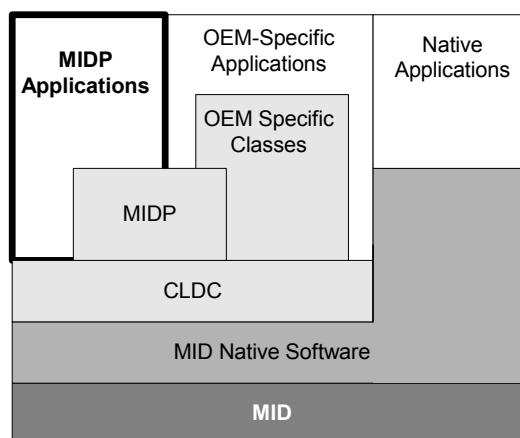


Figure 1: High-level architecture of MIDP applications

There are several basic concepts to understand when designing and implementing a MIDP application. Among these are a MIDlet's lifecycle, user interface (using high-level or low-level APIs), command handling, deployment, and related application management.

A MIDlet programmer also often needs an understanding of timers, persistent storage, and networking support. However, these features are beyond the scope of this document.

2.2 MIDlet Lifecycle

The Application Management Software (AMS) is the environment in which a MIDlet is installed, started, stopped, and uninstalled. The AMS is also sometimes referred to as the Java Application Manager (JAM). The AMS creates each new MIDlet instance, and controls its state by directing a MIDlet to start, pause, or destroy itself. The AMS is described in more detail in the MIDP specification [**MIDPSPEC**].

A MIDlet may be in one of the following states: Paused, Active, or Destroyed. These reflect how the AMS may control the lifecycle of a MIDlet.

When a MIDlet is first created and initialized, it is in the Paused state. If an exception occurs in the MIDlet's constructor, the MIDlet immediately enters the Destroyed state and is discarded. It may enter the Paused state from the Active state via completion of the MIDlet's `pauseApp()` method, or if calling `startApp()` from the Paused state throws an exception.

The MIDlet enters the Active state from the Paused state when the MIDlet method `startApp()` call has completed and the MIDlet is functioning normally.

The MIDlet may enter the Destroyed state on the completion of the method `destroyApp(boolean unconditional)`. This method releases all held resources and performs any necessary cleanup so that the MIDlet can be garbage collected. If the `unconditional` argument is `true`, the MIDlet will always enter the Destroyed state. If it is `false` and a `MIDletStateChangeException` is thrown, the MIDlet will not enter the Destroyed state. Appropriate exception handling may be needed in this case.

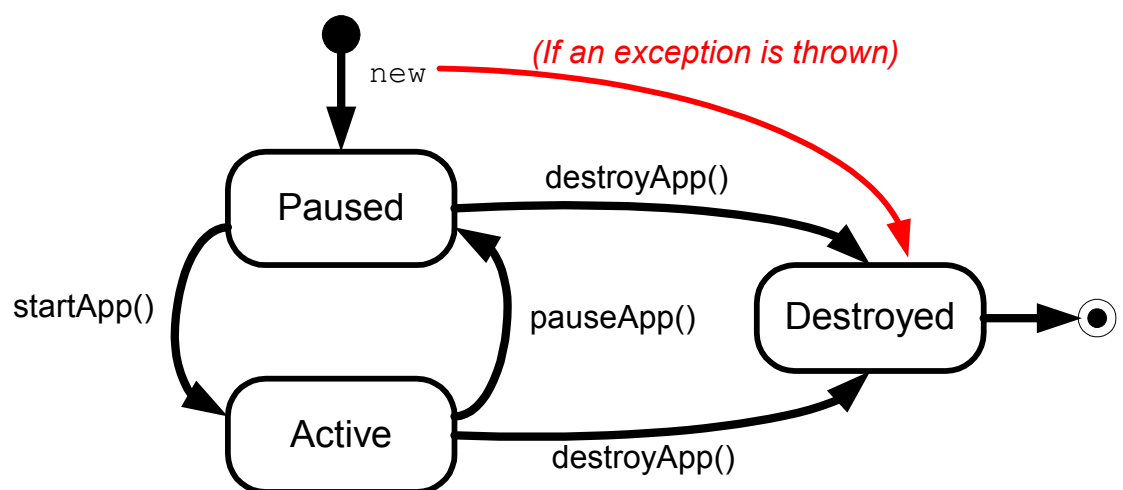


Figure 2: The states of the MIDlet lifecycle

A MIDlet programmer is mainly concerned with writing code that is executed during the Active state, and in the cleanup code needed for the MIDlet to enter the Paused or Destroyed states.

2.3 MIDP User Interface APIs

The MIDP User Interface (UI) has a high-level and a low-level API. The low-level API is based on use of the abstract class `Canvas`. The high-level API's classes `Alert`, `Form`, `List`, and `TextBox` are extensions of the abstract class `Screen`.

High-level API classes are designed to provide abstractions and components that are highly portable across different MIDs, as the actual implementation takes care of aspects such as drawing, font characteristics, navigation, scrolling, etc. The particular device's implementation of these classes performs the adaptation to its hardware and native UI look and feel.

The low-level API's `Canvas` class allows applications to have more direct control of the UI by permitting greater control of what is drawn on the display, and reception of low-level keyboard events. It is the application programmer's responsibility to ensure portability across MIDs with different characteristics (e.g., display size, color vs. black and white, and different keyboard types).

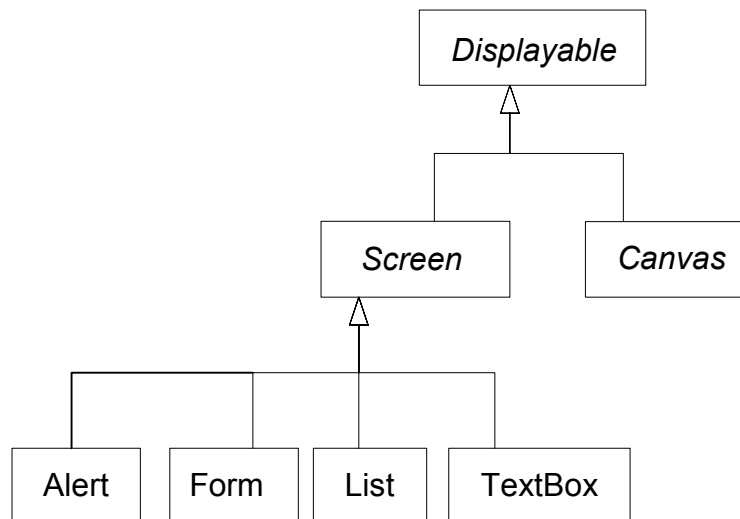


Figure 3: Class diagram of the MIDP screen types

2.4 MIDlet Suites and Application Descriptors

Application Management Software in the MID provides an environment for installing, updating, starting, stopping, and uninstalling MIDlets.

One or more MIDlets are packaged in a MIDlet suite JAR file for use by the application management software. MIDlets in the same suite share the same execution environment (virtual machine) and can interact with one another.

Each MIDlet suite JAR file may have an associated Application Descriptor used to describe its contents. The application descriptor MIME type is `text/vnd.sun.j2me.app-descriptor`. The file extension of an application descriptor file must be `jad`. The application management software uses the descriptor to manage each MIDlet—that is, to verify that a MIDlet is suited to execution on the mobile information device before loading the MIDlet suite JAR file.

The application descriptor is also used to hold configurable application properties of a MIDlet (see method `getAppProperty(String key)`). These properties can be modified in the JAD file without modifying the associated JAR file. The MIDP specification [MIDPSPEC] provides detailed information on the use of the application descriptor and mandatory or optional attributes.

An Example: The two example MIDlets included in this document were packaged in a single MIDlet suite. The JAD file for them (`helloExamples.jad`) defines the packaging of the two MIDlets (`HelloWorldMIDlet` and `BouncingTextMIDlet`), several mandatory and optional JAD attributes, and one application property "BouncingText-InitString" used by the

BouncingTextMIDlet. This property can be modified in the JAD file, changing the default text string used by the MIDlet without modifying the associated JAR file.

```
MIDlet-1: HelloWorld, , example.hello.HelloWorldMIDlet
MIDlet-2: BouncingText, , example.hello.BouncingTextMIDlet
MIDlet-Jar-Size: 9222
MIDlet-Jar-URL: HelloExamples.jar
MIDlet-Name: HelloExamples
MIDlet-Vendor: Forum Nokia
MIDlet-Version: 1.0.2
BouncingText-InitString: MIDP is fun !
```

Figure 4: HelloExamples.jad

When the application manager loads the MIDlet suite, it allows the user to choose which MIDlet to run, as shown in the following screen snapshot.



Figure 5: Choosing a MIDlet from the MIDlet suite

For more detailed information on MIDP, please see [\[MIDPSPEC\]](#).

3. A FIRST EXAMPLE: HELLOWORLDMIDLET

3.1 Design

The following is an example of a simple MIDlet that displays a "Hello World" message to the screen and handles an Exit command.

This example illustrates the:

- MIDlet lifecycle
- high-level user Interface (using class `TextBox`)
- command handling

Class `HelloWorldMIDlet` extends class `javax.microedition.midlet.MIDlet`. It uses a `HelloScreen` object to print the message and handle an Exit command.

Class `HelloScreen` extends the MIDP high-level API class `TextBox`. A `TextBox` user interface component has three visual areas: a title, a string contents area, and a command area (e.g., the Exit command in the figure below). Class `HelloScreen` implements the `CommandListener` interface used to receive user input from the command area.



Figure 6: The `HelloScreen` UI component (derived from `TextBox`)

The functionality of the `HelloWorldMIDlet` is very simple. The `startApp()` method sets the MIDlet's current `Display` to a `HelloScreen` object if a current display does not already exist. If the `HelloScreen` receives an Exit command from the user interface, it calls the `exitRequested()` method of the parent `HelloWorldMIDlet`, which handles the MIDlet state transition to the `Destroyed` state. The MIDlet `pauseApp()` and `destroyApp()` methods are empty because this example is so simple.

3.2 HelloWorldMIDlet.java

```
package example.hello;

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

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

public class HelloWorldMIDlet
    extends MIDlet
{
    public HelloWorldMIDlet()
    {
    }

    public void startApp()
    {
        Displayable current = Display.getDisplay(this).getCurrent();
        if(current == null)
        {
            HelloScreen helloScreen = new HelloScreen(this, "Hello
World.");
            Display.getDisplay(this).setCurrent(helloScreen);
        }
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean b)
    {
    }

    // A convenience method for exiting
    void exitRequested()
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
```

3.3 HelloScreen.java

```
package example.hello;

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

```
class HelloScreen
    extends TextBox
    implements CommandListener
{
    private final HelloWorldMIDlet midlet;
    private final Command exitCommand;

    HelloScreen(HelloWorldMIDlet midlet, String string)
    {
        super("HelloWorldMIDlet", string, 256, 0);
        this.midlet = midlet;
        exitCommand = new Command("Exit", Command.EXIT, 1);
        addCommand(exitCommand);
        setCommandListener(this);
    }

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

4. A SECOND EXAMPLE: BOUNCINGTEXTMIDLET

4.1 Design

4.1.1 Overview

The UI of the previous example was fairly static. MIDP is also well suited to dynamic graphical UIs. This second example presents a simple MIDlet that displays a text message whose position changes randomly over time. The user can enter the text message, and a default text message can be set as an application property.

The example illustrates:

- basic MIDlet lifecycle
- use of an application property from the JAD application descriptor file
- use of high-level and low-level user interface components
- simple transitions between multiple display screens
- simple command handling
- handling of display properties such as screen size and color support, when using the low-level API class `Canvas`

The screen snapshots in Figure 7 show the use of the MIDlet.

4.1.2 MIDlet: BouncingTextMIDlet

The MIDlet constructor creates the initial `TextInputScreen`, and sets the current display to that in method `startApp()`. The constructor also sets the default text string from the application property `BouncingText-InitString` set in the application descriptor file. In this example, the default text is "MIDP is fun!".

The MIDlet also provides a method for the state transition from `TextInputScreen` to `BouncingTextCanvas`. It is used by `TextInputScreen` to handle its `Start` command.

A separate thread is used for the bouncing-text canvas animation (see methods `BouncingTextCanvas.start()` and `BouncingTextMIDlet.textInputScreenDone()`).

The thread is an instance of `BouncingTextCanvas` (which implements the `Runnable` interface). This thread must be stopped or destroyed as appropriate, when the MIDlet is to enter the `Paused` or `Destroyed` state. This is handled in the `pauseApp()` and `destroyApp()` MIDlet lifecycle methods.

4.1.3 First Screen: TextInputScreen

The leftmost screen shows the initial `TextInputScreen`. The default text, from the application property `BouncingText-InitString`, is used to initialize the

`TextInputString` screen. The string can be modified in the `TextInputScreen` by the user (as in Figure 8).

The Start command is handled by `TextInputScreen` and causes a transition to the `BouncingTextCanvas` screen. The second and fourth screens of Figure 7 show that screen. The Exit command is also handled, and calls the `exitRequested()` method of its parent `BouncingTextMIDlet`.

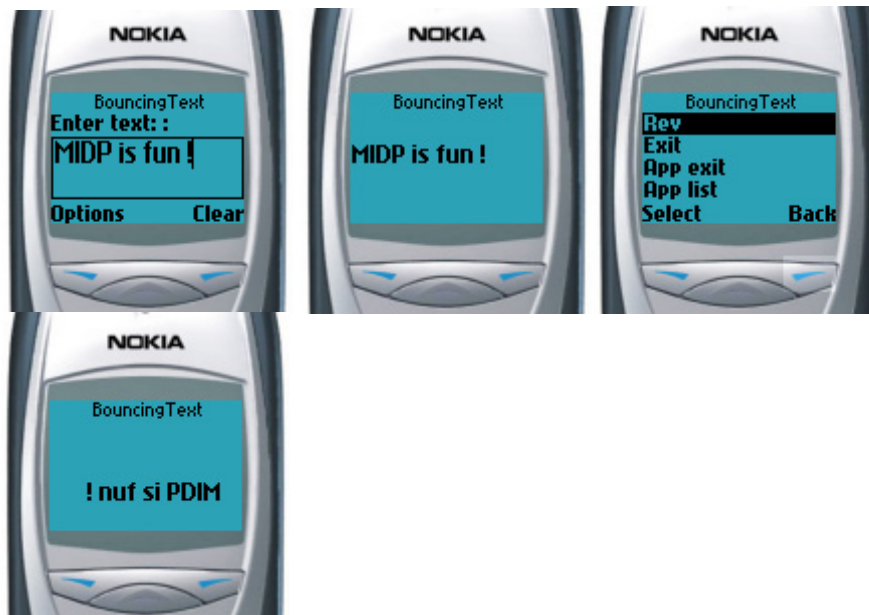


Figure 7: BouncingTextMIDlet User Interface

4.1.4 Second Screen: BouncingTextCanvas

The two rightmost screen snapshots are of the `BouncingTextCanvas` screen, taken at different times. The `BouncingTextCanvas` has the following functions:

- The text string changes position over time.
- The string will always be displayed within the available screen space if possible (unless the string is too long). The string will not "bounce" off-screen.
- A Rev command is handled by the `BouncingTextCanvas` to cause a reversal of the displayed text string.
- An Exit command is also handled, and calls the `exitRequested()` method of the `BouncingTextMIDlet`.

The `BouncingTextCanvas` application code is aware of properties such as display size and whether or not the display is black and white or supports color. It uses these display properties to decide the canvas size, where text is drawn on the canvas, and for the canvas background color and text foreground color.

On a black-and-white device, the background is white and the text is black, while on devices with color displays these are chosen randomly each time the `startApp()` method is called. An example of the use of the MIDlet on a color display is shown in Figure 8. Note that the user has modified the default text and changed it to be "XYZ."

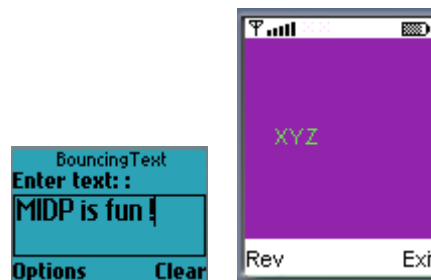


Figure 8: MIDlet application code should take account of Display size and color support

A string is drawn on a canvas starting from some $\{x, y\}$ anchor position, which is near the string's top left corner. In this example, the bouncing text message should not bounce outside the displayable area of the canvas. A maximum anchor position is calculated based on the height and width of the canvas' displayable area, and the string's height and width when displayed. To avoid attempts to display any part of the string outside the canvas' displayable area, the anchor position is chosen between $\{0, 0\}$ and a maximum anchor position every time the next animation is performed.

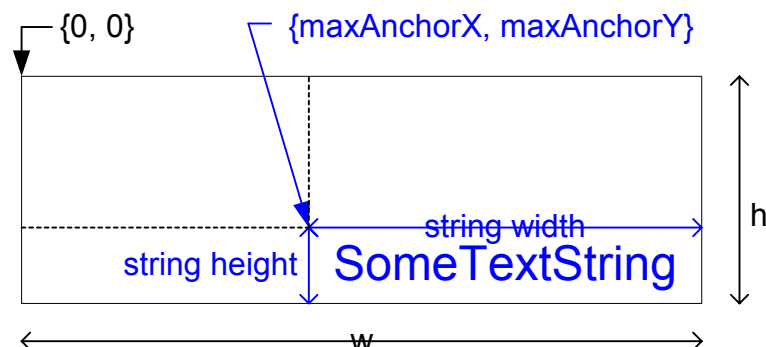


Figure 9: Displaying the string within the canvas' displayable area

4.2 BouncingTextMIDlet.java

```
package example.hello;

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

public class BouncingTextMIDlet
```

```
extends MIDlet
{
    private final TextInputScreen textInputScreen;
    private BouncingTextCanvas bouncingTextCanvas = null;

    public BouncingTextMIDlet()
    {
        String str = getAppProperty("BouncingText-InitString");
        if (str == null)
        {
            str = "Hello World."; // default
        }
        textInputScreen = new TextInputScreen(this, str);
    }

    public void startApp()
    {
        Displayable current = Display.getDisplay(this).getCurrent();
        if (current == null)
        {
            Display.getDisplay(this).setCurrent(textInputScreen);
        }
        else
        {
            Display.getDisplay(this).setCurrent(current);
            if (current == bouncingTextCanvas)
            {
                // restart animation after pauseApp
                bouncingTextCanvas.startAnimation();
            }
        }
    }

    public void pauseApp()
    {
        if (bouncingTextCanvas != null)
        {
            bouncingTextCanvas.stopAnimation();
        }
    }

    public void destroyApp(boolean b)
    {
        if (bouncingTextCanvas != null)
    }
```

```
        {
            bouncingTextCanvas.stopAnimation();
        }
    }

    // A convenience method for exiting
    void exitRequested()
    {
        destroyApp(false);
        notifyDestroyed();
    }

    // Handle the display transition from the initial
    // 'TextInputScreen' state to a 'BouncingTextCanvas' state
    void textInputScreenDone(String str)
    {
        bouncingTextCanvas =
            new BouncingTextCanvas(this, Display.getDisplay(this), str);
        bouncingTextCanvas.startAnimation();

        Display.getDisplay(this).setCurrent(bouncingTextCanvas);
    }
}
```

4.3 TextInputScreen.java

```
package example.hello;

import javax.microedition.lcdui.*;

class TextInputScreen
    extends TextBox
    implements CommandListener
{
    private final BouncingTextMIDlet midlet;
    private final Command exitCommand;
    private final Command startCommand;

    TextInputScreen(BouncingTextMIDlet midlet, String string)
    {
        super("Enter text: ", string, 256, TextField.ANY);
        this.midlet = midlet;

        // Command area
    }
}
```

```

        exitCommand = new Command("Exit", Command.EXIT, 1);
        addCommand(exitCommand);
        startCommand = new Command("Start", Command.SCREEN, 1);
        addCommand(startCommand);
        setCommandListener(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        if (c == exitCommand)
        {
            midlet.exitRequested();
        }
        else if (c == startCommand)
        {
            midlet.textInputScreenDone(getString());
        }
    }
}

```

4.4 BouncingTextCanvas.java

```

package example.hello;

import java.util.Random;
import javax.microedition.lcdui.*;

class BouncingTextCanvas
    extends Canvas
    implements CommandListener, Runnable
{
    private final BouncingTextMIDlet midlet; // parent MIDlet
    private final Display display;           // parent MIDlet's Display
    private final Command exitCommand;       // display 'Exit' command
    private final Command reverseCommand;    // display 'Reverse'
    command
    private final BouncingText text;         // text string to print

    private int bgRGB;                       // background color
    private int textRGB;                     // text color
    private volatile Thread animationThread = null; // animation thread

    BouncingTextCanvas(BouncingTextMIDlet midlet, Display display,
                       String string)
    {

```

```
this.midlet = midlet;
this.display = display;

// Setup the command handling
setCommandListener(this);
reverseCommand = new Command("Rev", Command.SCREEN, 1);
exitCommand = new Command("Exit", Command.EXIT, 1);
addCommand(exitCommand);
addCommand(reverseCommand);

// Initialize the bouncing text message
//
// Create a new 'bouncing text' string whose color is 'textRGB'.
// Its initial position is at x=0 and
// y="middle of the canvas". the canvas width and height
// are also used by a BouncingText object, as it manages
// its own position update within the canvas displayable area
// {x,y} = {0,0} to {w,h}.

int ch = getHeight(); // canvas height
int cw = getWidth();  // canvas width
initColors();          // Choose fg and background colors ...
                      // ... based on display capabilities.
text = new BouncingText(string, textRGB, 0, ch/2, cw, ch);
}

void startAnimation()
{
    animationThread = new Thread(this);
    animationThread.start();
}

void stopAnimation()
{
    animationThread = null;
}

// While this thread has not been stopped, it will perform a
// tick() action approximately every 'millis_per_tick' milliseconds
// (as close to that period as possible).
public void run()
{
    int millis_per_tick = 100; // msec
```

```

Thread currentThread = Thread.currentThread();

try
{
    // This ends when animationThread is set to null, or when
    // it is subsequently set to a new Thread. Either way, the
    // current thread should terminate.
    while (currentThread == animationThread)
    {
        long startTime = System.currentTimeMillis();
        tick();
        repaint(0, 0, getWidth(), getHeight());
        serviceRepaints();
        long elapsedTime = System.currentTimeMillis() -
startTime;
        if (elapsedTime < millis_per_tick)
        {
            synchronized(this)
            {
                wait(millis_per_tick - elapsedTime);
            }
        }
        else
        {
            currentThread.yield();
        }
    }
}
catch(InterruptedException e)
{
}
}

//Draw the background and the bouncing text string
public void paint(Graphics g)
{
    g.setColor(bgRGB);
    g.fillRect(0, 0, getWidth(), getHeight());
    text.draw(g);
}

// Handle the UI commands
public void commandAction(Command c, Displayable d)
{
    if (c == exitCommand)
    {

```

```
        stopAnimation();
        midlet.exitRequested();
    }
    else if (c == reverseCommand)
    {
        text.reverse();
    }
}

// What to do each run() tick
private synchronized void tick()
{
    text.updatePosition();
}

// Choose a random background color on color displays,
// use a white background color on black-and-white displays
private void initColors()
{
    int white = 0xffffffff;
    int black = 0x000000;
    if (display.isColor())
    {
        Random random = new Random();
        textRGB = random.nextInt() & 0xffffffff;
        bgRGB = white - textRGB; // constrasting background color
    }
    else
    {
        textRGB = black;
        bgRGB = white;
    }
}

class BouncingText
{
    private final int w;           // The canvas width
    private final int h;           // The canvas height
    private final int textRGB;     // Color for the string
    private final Random random;   // For generating random numbers

    private String string;         // The string to print
    private int x;                 // The x anchor for the string
    private int y;                 // The y anchor for the string
    private int strWidth = 0;      // The pixel width of the string
```

```
private int strHeight = 0;    // The pixel height of the string
private int xdir = 1;        // 1 or -1 : bounce direction
private int ydir = 1;        // 1 or -1 : bounce direction
```

```
BouncingText(String string, int textRGB,
              int x, int y, int w, int h)
```

```
{
    this.string = string;
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.textRGB = textRGB;
    random = new Random();
}
```

```
// Randomly change the position of the string, the string
// must be completely drawable within the canvas
```

```
private void updatePosition()
```

```
{
    // Choose a random { vx, vy } velocity.
    int vx = random.nextInt() & 0x07; // 0-7 pixels
    int vy = random.nextInt() & 0x07; // 0-7 pixels

    // Note: The methods maxAnchorX() and maxAnchorY()
    //         determine the upper left hand {x, y} coordinates
    //         of the string so that the string can be drawn
    //         within the canvas height and width.
    if ((xdir == 1) && ((x + vx) >= maxAnchorX()))
    {
        xdir = -xdir; // change direction at right edge
    }
    else if ((xdir == -1) && ((x - vx) < 0))
    {
        xdir = -xdir; // change direction at left edge
    }
    int xnew = x + (vx * xdir) ;
    if ((xnew >= 0) && (x < maxAnchorX()))
    {
        x = xnew;
    }

    if ((ydir == 1) && ((y + vy) >= maxAnchorY()))
    {
        ydir = -ydir; // change direction at bottom edge
    }
}
```

```
    }
    else if ((ydir == -1) && ((y - vy) < 0))
    {
        ydir = -ydir; // change direction at top edge
    }
    int ynew = y + (vy * ydir) ;
    if ((ynew >= 0) && (y < maxAnchorY()))
    {
        y = ynew;
    }
}

// Draw the text string at the appropriate position
private void draw(Graphics g)
{
    // If the strHeight is still 0, then initialize the
    // strHeight and strWidth variables used in maxAnchorX()
    // and maxAnchorY(). This assumes that the font represents
    // strings as at least 1 pixel high.
    if (! (strHeight > 0))
    {
        Font f = g.getFont();
        strHeight = f.getHeight();
        strWidth = f.stringWidth(string);
    }

    // draw the string
    g.setColor(textRGB);
    g.drawString(string, x, y, Graphics.TOP|Graphics.LEFT);
}

private int maxAnchorX()
{
    return w - strWidth;
}

private int maxAnchorY()
{
    return h - strHeight;
}

// Reverse the BouncingText string
private void reverse()
{

```

```
// Note: another approach is to store two copies of
//       the string, one forwards and one reversed

char[] carray = string.toCharArray();
for (int old = string.length()-1, nu=0;
     old >=0;
     old--, nu++)
{
    carray[nu] = string.charAt(old);
}
string = new String(carray);
}
}
```