Although this document is written so that it slightly resembles a chapter of a book, this does not belong to my Java book *A Natural Introduction to Computer Programming in Java*. This document is additional material which you might use after you know the basics of programming in Java.

CHAPTER 18

MIDLETS – JAVA PROGRAMS FOR MOBILE DEVICES

Sun Microsystem provides a Java platform that is commonly installed in mobile phones. This Java platform is called Java Micro Edition (ME). The Java platform that we use in personal computers is Java Standard Edition (SE). The Java Micro Edition differs from the Standard Edition so that it is a lighter version of Java. This means, for example, that the number of standard Java classes is smaller in the Java Micro Edition.

In this chapter we shall learn to make some basic programs that run on devices that provide the Java Micro Edition. In practice this means that you can run these programs in many types of mobile phones. The Java applications that run on mobile devices are called midlets. The first three letters in this word refer to the acronym of Mobile Interconnected Device, MID. Mobile phones are devices that are interconnected via a network.

Although mobile phones are computers that contain processors, they differ from traditional computers in that their displays are small, their keyboards have only numerical and some special keys, and usually they lack a pointing device such as a mouse. These limimations must be taken into account when programs such as midlets are designed for mobile phones.

In this chapter we shall study the basic structure of Java midlets. As Java midlets provide a graphical user interface (GUI), it is beneficial if you are familiar with Java Standard Edition GUI programming. Java midlets are constructed by utilizing standard Java classes. This means, for example, that all midlets are derived from a standard class named MIDlet. We will learn how to build midlet user interfaces with standard classes, how to draw and show images on the display, and how to use threads in midlet programs.

2007-08-28 This file was created.
2008-06-05 Last modification.

The page size of this document is A4, but you can print this on a letter size page.

HelloSimpleMIDlet.java – A program that says "Hello"

Midlets – Java programs that can run on mobile phones and other small devices – can be constructed so that a new application-specific midlet class is derived from a standard class named **`MIDlet`**. In this case, the name of the new class is **`HelloSimpleMIDlet`**.

In these example programs, we always have this line which creates a **`Display`** object which represents the physical display of the device (e.g. a mobile phone) in which the program will be executed.

```
//  HelloSimpleMIDlet.java (c) Kari Laitinen

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

public class HelloSimpleMIDlet extends  MIDlet
{
   Display  display_of_this_midlet  =  Display.getDisplay( this ) ;

   Form  form_for_string_item  =  new  Form( "THIS IS THE TITLE OF A Form" ) ;

   public HelloSimpleMIDlet()
   {
      StringItem text_to_be_shown  =
                  new  StringItem( "", "Hello. I am a simple Java midlet.") ;

      form_for_string_item.append( text_to_be_shown ) ;
   }

   protected void startApp() throws MIDletStateChangeException
   {
      display_of_this_midlet.setCurrent( form_for_string_item ) ;
   }

   protected void pauseApp()
   {
   }

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

Every midlet must have methods named **`startApp()`**, **`pauseApp()`**, and **`destroyApp()`**. The program execution system invokes these methods when a midlet application starts executing, when it is paused, or when it is entirely destroyed. The **`startApp()`** method, for example, makes the **`Form`** object as the currect display content when this program starts executing.

HelloSimpleMIDlet.java - 1. A midlet that shows text inside a StringItem object.

The text "Hello. I am a simple Java midlet" that is shown on the display is stored inside a **`StringItem`** object. **`StringItem`** objects represent items that contain text that can be seen, but not modified, by the user. The **`StringItem`** object is attached to a **`Form`** object with the **`append()`** method. A **`Form`** object can contain items that are shown on the screen.

HelloSimpleMIDlet.java - X. The midlet is being executed in an emulator.

GraphicsDemoMIDlet.java – drawing methods demonstrated

When we want to construct a midlet which draws graphical objects such as lines and rectangles to the display, we need to derive a new canvas class from the standard **Canvas** class. Inside this new class we write a method named **paint()** which can use the drawing methods provided in class **Graphics**. In this program the name of the new canvas class is **GraphicsDemoCanvas**.

Canvas means in traditional sense a piece of strong cloth on which an artist can create a painting. In Java **Canvas** is a class from which you can derive new classes which represent drawing surfaces. Into a class that is derived from the standard **Canvas** class you must write a method named **paint()** which will take care of the actual drawing and painting activities. Method **paint()** will be called automatically when the program is being executed, and it will receive a reference to a **Graphics** object as a parameter. Methods of class **Graphics** can be used to perform the actual drawing activities.

```
//  GraphicsDemoMIDlet.java

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

class GraphicsDemoCanvas extends Canvas
{
   int  canvas_width   =  getWidth() ;
   int  canvas_height  =  getHeight() ;

   public void paint( Graphics graphics )
   {
      graphics.setColor( 255, 255, 255 ) ;  // white color clears the canvas
      graphics.fillRect( 0, 0, canvas_width, canvas_height ) ;

      graphics.setColor( 0, 0, 0 ) ;  // black color is used for drawing

      graphics.drawString( "Canvas size is "  +  canvas_width
                       +  " x "  +  canvas_height, 20, 20,
                          Graphics.TOP | Graphics.LEFT ) ;

      //  Drawing a horizontal line into the middle of canvas area.
      graphics.drawLine( 0, canvas_height / 2,
                         canvas_width, canvas_height / 2 ) ;

      graphics.fillRect( 20, 70, 100, 40 ) ;

      graphics.fillArc(  20, 170, 100, 80,  45, 270 ) ;
      graphics.drawArc( 100, 170, 100, 80, 315,  90 ) ;
   }
}
```

At the beginning of the **paint()** method, white color is set as the current drawing color. The numerical values 255, 255, and 255 describe the red, blue, and green components in white color. When a large filled rectangle is drawn onto the display, the display is cleared of possible older drawings.

GraphicsDemoMIDlet.java - 1: A midlet that demonstrates methods of class Graphics.

The **MIDlet**-based class of this program is rather simple as the drawing activities are carried out in the **Canvas**-based class.

Here, an object of class **GraphicsDemoCanvas** is created, and it is set as the current display contents in the **startApp()** method which is called automatically when this midlet starts executing.

```
public class GraphicsDemoMIDlet extends MIDlet
{
   Display  display_of_this_midlet =  Display.getDisplay( this ) ;
   GraphicsDemoCanvas  canvas_of_this_midlet =  new  GraphicsDemoCanvas() ;

   protected void startApp() throws MIDletStateChangeException
   {
      display_of_this_midlet.setCurrent( canvas_of_this_midlet ) ;
   }

   protected void pauseApp()
   {
   }

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

The **pauseApp()** and **destroyApp()** methods are often empty in simple programs. The program execution system calls these methods when it wants the midlet application to pause or to terminate.

GraphicsDemoMIDlet.java - 2. The GraphicsDemoMIDlet class.

Methods **`fillArc()`** and **`drawArc()`** are used to produce these two drawings. Methods whose names begin with the word **`fill`** fill the contents of a graphical shape with the current drawing color. These methosds to draw arcs actually can draw much more than their names imply. An arc of 360 degrees is an oval. An oval whose height is the same as its width is a circle.

GraphicsDemoMIDlet.java - X. The midlet is being executed in an emulator.

SumMIDlet – Using TextField objects to input data from the user

This midlet implements the interfaces **CommandListener** and **ItemStateListener** which means that it has the methods **commanAction()** and **itemStateChanged()**.

TextField objects are used to receive two integers from the user. The sum of the two integers will be shown in an uneditable **TextField**. The parameter **TextField.NUMERIC** specifies that only numbers can be written to the text fields.

```
//  SumMIDlet.java (c) Kari Laitinen

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

public class SumMIDlet extends  MIDlet
                  implements CommandListener, ItemStateListener
{
   Display  display_of_this_midlet = Display.getDisplay( this ) ;

   TextField first_integer_text_field   = new TextField( "First integer: ",
                                              "", 8, TextField.NUMERIC ) ;
   TextField second_integer_text_field  = new TextField( "Second integer:",
                                              "", 8, TextField.NUMERIC )  ;
   TextField result_text_field     = new TextField( "Calculated sum:",
                                          "0", 8, TextField.NUMERIC |
                                                  TextField.UNEDITABLE ) ;

   Form  form_of_this_midlet  =  new  Form( "SumMIDlet" ) ;

   Command exit_command  =  new  Command( "EXIT", Command.EXIT, 1 ) ;

   public SumMIDlet()
   {
      first_integer_text_field.setLayout( Item.LAYOUT_CENTER ) ;

      form_of_this_midlet.append( first_integer_text_field ) ;
      form_of_this_midlet.append( second_integer_text_field ) ;
      form_of_this_midlet.append( result_text_field ) ;

      form_of_this_midlet.setItemStateListener( this ) ;

      form_of_this_midlet.addCommand( exit_command ) ;
      form_of_this_midlet.setCommandListener( this ) ;
   }
```

With the **append()** method the **TextField** objects are attached to a **Form** object which will be put on the display in the **startApp()** method.

This line specifies that "this" object will listen to what happens to the objects that are attached to the **Form**. In practice this means that the **itemStateChanged()** method will be called when the texts in the **TextField**s are modified.

SumMIDlet.java - 1: A program that can calculate the sum of two integers.

Method **startApp()** will be called after the constructor of this class is executed. In the beginning the **Form** of this midlet is put visible on the display.

```
protected void startApp() throws MIDletStateChangeException
{
   display_of_this_midlet.setCurrent( form_of_this_midlet ) ;
}

protected void pauseApp()
{
}

protected void destroyApp( boolean unconditional_destruction_required )
{
}

public void commandAction( Command     given_command,
                           Displayable display_content )
{
   if ( given_command  ==  exit_command )
   {
      destroyApp( false ) ;
      notifyDestroyed() ;
   }
}
```

A **Command** object is a possible command that is attached to a so-called soft key of a mobile phone. Usually mobile phones have two soft keys whose actual functionality is determined by **Command** objects. In this program there is only the EXIT command in use, and this method is called automatically when the command is given.

Method **commandAction()** implements the **CommandListener** interface. The two parameters that will be supplied to it tell which command was given, and what was being shown on the display when the command was given.

SumMIDlet.java - 2: The "mandatory" methods and the commandAction() method.

This `itemStateChanged()` method will be called by the program execution system when the text of a `TextField` object is modified. It would be possible to find out which `TextField` object was modified since `item_which_changed_state` references the modified object. (`TextField` is a subclass of class `Item`.) This method does not, however, bother which text field was modified. The sum of the two numbers is calculated always after one of the `TextField` objects is modified.

```
public void itemStateChanged( Item item_which_changed_state )
{
   String first_integer_text   =  first_integer_text_field.getString() ;
   String second_integer_text  =  second_integer_text_field.getString() ;

   if ( first_integer_text.length() ==  0 )
   {
      first_integer_text  =  "0" ;
   }

   if ( second_integer_text.length()  ==  0 )
   {
      second_integer_text  =  "0" ;
   }

   int first_integer   =  Integer.parseInt( first_integer_text ) ;
   int second_integer  =  Integer.parseInt( second_integer_text ) ;

   int sum_of_two_integers  =  first_integer  +  second_integer ;

   String  sum_text  =  "" + sum_of_two_integers ;

   result_text_field.setString( sum_text ) ;
}
}
```

The `TextField` class provides the methods `getString()` and `setString()` which can be used to read and write the texts that are currently stored in the text fields. The `setString()` method is used here to set the result into the third text field.

If the two text fields are empty, i.e., the user has not yet written anything to them, this program acts as if they contained zeroes. The static method `parseInt()` of class `Integer` is used to convert a `String` object to an `int` value. The `parseInt()` method throws an exception if the string is an empty string or it cannot, because of some other reason, convert the string to an `int` value.

SumMIDlet.java - 3. The method that is invoked when a TextField is modified.

Yhteenlaskun tuloksen sisältävä viimeinen tekstikenttä on sellainen että sen arvoa ei midletin käyttäjä voi muuttaa. Tuommoisen tekstikentän teksti näkyy himmeämpänä kuin muiden tekstikenttien tekstit. Tämän tekstikenttäolion luonnissa on käytetty vakiota `TextField.UNEDITABLE` jolla saadaan aikaan ei-editoitava tekstikenttä.

SumMIDlet.java - X. Here the program has calculated the sum of 2233 and 4455.

KeyCodesMIDlet.java – a midlet that reacts to key pressings

In this program all functionality is specified in the class that is derived from the standard **Canvas** class. This class implements the **CommandListener** interface which means that it has the **commandAction()** method to handle Soft Key commands.

```
//  KeyCodesMIDlet.java  (c) Kari Laitinen

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

class KeyCodesCanvas extends Canvas
                     implements CommandListener
{
   String  key_code_as_string  =  "No keys pressed" ;

   int  key_code_numerical  =  0 ;
   int  game_action_code    =  0 ;

   Command select_hexadecimal_printing  =  new Command( "Hexadecimal",
                                                        Command.SCREEN, 1 ) ;
   Command select_decimal_printing      =  new Command( "Decimal",
                                                        Command.SCREEN, 1 ) ;
   Command select_binary_printing       =  new Command( "Binary",
                                                        Command.SCREEN, 1 ) ;

   Command last_given_command  =  select_decimal_printing ;

   public KeyCodesCanvas()
   {
      addCommand( select_hexadecimal_printing ) ;
      addCommand( select_decimal_printing ) ;
      addCommand( select_binary_printing ) ;

      setCommandListener( this ) ;
   }
```

Three **Command** objects are added to "this" canvas. With these commands the user can specify in which numbering system the the key codes are shown on the screen. The three **Command** objects are automatically put into a menu from which the user can select individual commands.

KeyCodesMIDlet.java - 1: Demonstrating the handling of key pressings.

`last_given_command` is a data field in this class. During the execution of this program it points to one of the three `Command` objects. The value of `last_given_command` will be checked in the `paint()` method, and key codes are printed either in decimal, hexadecimal, or binary numbering system.

```
public void commandAction( Command      given_command,
                           Displayable  display_content )
{
   last_given_command  =  given_command ;
}

public void keyPressed( int key_code )
{
   game_action_code = getGameAction( key_code ) ;

   key_code_numerical  =  key_code ;

   key_code_as_string  =  getKeyName( key_code ) ;


   repaint() ;
}
```

By calling the `repaint()` method you can request that the program execution system updates the canvas, i.e., calls the `paint()` method. The `paint()` method will be called always after a key has been pressed down.

When a method named `keyPressed()` is written to a `Canvas`-based class, the method will be called when a key is pressed down while the canvas is visible on the screen. The method receives a key code as a parameter. The received key code can be converted to a so-called game action code or to a string with the `Canvas` methods `getGameAction()` and `getKeyName()`.

By running this program, you can find out that the key code, that is received as a parameter, corresponds to the standard character codes. See documentation of class `Canvas` to find out more information about game action codes.

KeyCodesMIDlet.java - 2: The methods that react to commands and key pressings.

In this program this **paint()** method will be executed always after the user has pressed a key of the mobile phone keyboard.

If the user has selected hexadecimal printing of key codes, the static method **toHexString()** of class **Integer** is used to convert the two **int** values to strings.

```
protected void paint( Graphics graphics )
{
   graphics.setColor( 255, 255, 255 ) ;
   graphics.fillRect( 0, 0, getWidth(), getHeight() ) ;

   graphics.setColor( 0, 0, 0 ) ;

   String game_action_code_to_print  =  "" + game_action_code ;
   String key_code_numerical_to_print  =  "" + key_code_numerical ;

   if ( last_given_command  ==  select_hexadecimal_printing )
   {
      game_action_code_to_print  =
               Integer.toHexString( game_action_code ) + "H" ;
      key_code_numerical_to_print  =
               Integer.toHexString( key_code_numerical ) + "H" ;
   }
   else if ( last_given_command  ==  select_binary_printing )
   {
      game_action_code_to_print  =
               Integer.toBinaryString( game_action_code ) + "B" ;
      key_code_numerical_to_print  =
               Integer.toBinaryString( key_code_numerical ) + "B" ;
   }

   graphics.drawString( "game_action_code:    "  +  game_action_code_to_print,
                        10, 20,
                        Graphics.TOP | Graphics.LEFT ) ;

  graphics.drawString( "key_code_numerical: "  +  key_code_numerical_to_print,
                        10, 40,
                        Graphics.TOP | Graphics.LEFT ) ;

   graphics.drawString( "key_code_as_string:  "  +  key_code_as_string,
                        10, 60,
                        Graphics.TOP | Graphics.LEFT ) ;
}
}
```

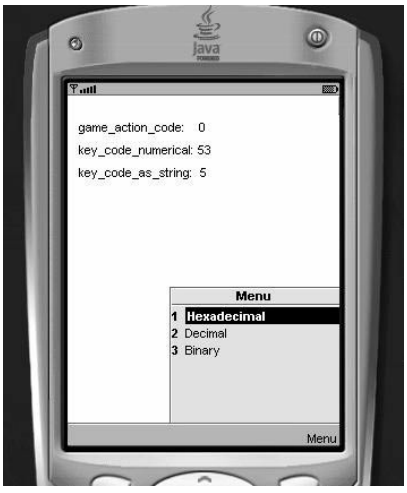
The last parameter for the **drawString()** method specifies how the text is printed in relation to the given point. If you replace the **Graphics.LEFT** with **Graphics.RIGHT**, the text will be printed to the left of the point (10, 60), and it will not be completely visible on the screen.

KeyCodesMIDlet.java - 3: The paint() method in the KeyCodesCanvas class.

An object of type **KeyCodesCanvas** is created first and then it is set visible on the display.

```
public class KeyCodesMIDlet extends MIDlet
{
   Display   display_of_this_midlet  =  Display.getDisplay( this ) ;
   KeyCodesCanvas  key_codes_canvas  =  new KeyCodesCanvas() ;

   protected void startApp() throws MIDletStateChangeException
   {
      display_of_this_midlet.setCurrent( key_codes_canvas ) ;
   }

   protected void pauseApp()
   {
   }

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

KeyCodesMIDlet.java - 4. The short KeyCodesMIDlet class.



KeyCodesMIDlet.java - X. Key 5 has been pressed before the menu is activated.

PictureViewingMIDlet.java – showing images on small screen

This program shows a set of pictures on the screen of the mobile phone. One picture is shown at a time. The user can select another picture with the arrow keys. When Sun Java Wireless Toolkit is used, the picture files must be stored to the **res** folder of the project. The file names of the picture files are stored into an initialized array of type `String[]`. The other array, whose type is `Image[]`, will contain references to `Image` objects that are created in the constructor.

```
//  PictureViewingMIDlet.java

import java.io.* ;

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

class PictureViewingCanvas extends Canvas
{
   int  index_of_current_picture  =  0 ;

   String[]  picture_file_names  =  { "/marilyn_by_warhol.png",
                                      "/nicole_kidman.png",
                                      "/terminator2.png",
                                      "/kate_winslet.png",
                                      "/scanned_leave.png" } ;

   Image[]   pictures_to_be_shown  =  new  Image[ picture_file_names.length ] ;

   public PictureViewingCanvas()
   {
      for ( int picture_index  =  0 ;
                picture_index  <  picture_file_names.length ;
                picture_index  ++ )
      {
         try
         {
            pictures_to_be_shown[ picture_index ]  =
                   Image.createImage( picture_file_names[ picture_index ] ) ;
         }
         catch ( IOException  caught_io_exception )
         {
            System.out.print( "\n Image object not created ....  "
                           +  picture_file_names[ picture_index ] ) ;
         }
      }
   }
```

For each picture file an object of type `Image` is created. As it is possible that method `createImage()` throws an exception when the picture file cannot be read successfully, the `Image` objects must be created inside a `try-catch` construct.

PictureViewingMIDlet.java - 1: The first part of class PictureViewingCanvas.

An object of type **Image** is drawn to the screen with the **drawImage()** method of class **Graphics**. The image is drawn so that the coordinates (2, 0) refer to its upper left corner.

```
    protected void paint( Graphics graphics )
    {
       graphics.setColor( 255, 255, 255 ) ;  //  white
       graphics.fillRect( 0, 0, getWidth(), getHeight() ) ;

       graphics.drawImage( pictures_to_be_shown[ index_of_current_picture ],
                           2, 0,
                           Graphics.TOP | Graphics.LEFT ) ;
    }


    public void keyPressed( int key_code )
    {
       int game_action_code = getGameAction( key_code ) ;

       switch ( game_action_code )
       {
       case UP:
       case LEFT:

          if ( index_of_current_picture  >  0 )
          {
             index_of_current_picture  --  ;
          }
          else
          {
             index_of_current_picture  =  pictures_to_be_shown.length  -  1 ;
          }

          break ;

       case DOWN:
       case RIGHT:

          if ( index_of_current_picture  <  ( pictures_to_be_shown.length - 1 ))
          {
             index_of_current_picture  ++  ;
          }
          else
          {
             index_of_current_picture  =  0 ;
          }

          break ;
       }

       repaint() ;
    }
}
```

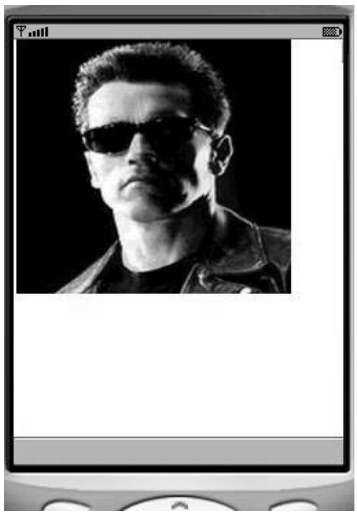
The value of data field **index_of_current_picture** stipulates which picture will be drawn by the **paint()** method. Here, the value of the variable is incremented, or set to zero if it already has reached its maximum allowed value.

PictureViewingMIDlet.java - 2: The rest of class PictureViewingCanvas.

In this program the **MIDlet**-based class, the class that is derived from class **MIDlet**, is short. An object of type **PictureViewingCanvas** is created, and this object is then set as the content of the display.

```
public class PictureViewingMIDlet extends MIDlet
{
   Display  display_of_this_midlet  =  Display.getDisplay( this ) ;
   PictureViewingCanvas  picture_viewing_canvas = new PictureViewingCanvas() ;

   protected void startApp() throws MIDletStateChangeException
   {
      display_of_this_midlet.setCurrent( picture_viewing_canvas ) ;
   }

   protected void pauseApp()
   {
   }

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

PictureViewingMIDlet.java - 3. The MIDlet-based class of the program.



This program is a slightly bad midlet as it does not provide an Exit command. If you run this program on a real phone, you can exit the program when you press the key that normally terminates a phone call.

PictureViewingMIDlet.java - X. The file terminator2.png is being shown here.

MovingBallMIDlet.java – a program that uses a List object

```
//  MovingBallMIDlet.java (c) Kari Laitinen

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

class MovingBallCanvas extends Canvas
                       implements  CommandListener
{
   MIDlet  master_midlet ;
   Display midlet_display ;

   int ball_position_x, ball_position_y ;

   //  The color is specified with a hexadecimal value 0x00RRGGBB
   //  so that each color component (red, green, and blue) can
   //  have value 0 ... 0xFF.

   int  current_color   =    0x00FF0000 ;  //  red is the initial color

   //  The following two initialized arrays must be organized so that
   //  the RGB value of a color has the same index as the name
   //  of the color in question.

   int[] rgb_color_specifications  =  {  0x00FF0000,  0x0000FF00,  0x000000FF,
                                         0x007F0000,  0x00007F00,  0x0000007F,
                                         0x0000FFFF,  0x00FF00FF,  0x00FFFF00,
                                         0x00000000,  0x007F7F7F } ;

   String[] selectable_colors      =  { "Red",       "Green",      "Blue",
                                        "Dark red",  "Dark green", "Dark blue",
                                        "Cyan",      "Magenta",    "Yellow",
                                        "Black",     "Grey" } ;

   List  color_selection_list  =  new  List( "Select Ball Color",
                                             List.IMPLICIT,
                                             selectable_colors, null ) ;

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

   Command  change_color_command  =  new  Command( "Change color",
                                                   Command.SCREEN, 1 ) ;
```

This program displays a ball on the screen. The color of the ball can be changed. The possible ball colors can be selected from a menu that is built by using class **`List`**. When a **`List`** object is created with parameter **`List.IMPLICIT`**, it will be easy to process a selection from the list.

When you create a **`Command`** with parameter **`Command.EXIT`**, the command will be attached to that Soft Key which is the usual Exit key of the phone.

MovingBallMIDlet.java - 1: The data members of class MovingBallCanvas.

```
public MovingBallCanvas( MIDlet   given_master_midlet,
                         Display  given_display )
{
   master_midlet   =  given_master_midlet ;
   midlet_display  =  given_display ;

   ball_position_x  =  getWidth()  / 2  -  20 ;
   ball_position_y  =  getHeight() / 2  -  20 ;

   addCommand( change_color_command ) ;
   addCommand( exit_command ) ;

   setCommandListener( this ) ;
   color_selection_list.setCommandListener( this ) ;
}

public void commandAction( Command     given_command,
                           Displayable display_content )
{
   if ( given_command  ==  change_color_command )
   {
      midlet_display.setCurrent( color_selection_list ) ;
   }
   else if ( given_command  ==  List.SELECT_COMMAND )
   {
      int index_of_selected_color  =
                  color_selection_list.getSelectedIndex() ;

      //  The following assignment statement selects the right color
      //  when the array pointed by rgb_color_specifications is initialized
      //  so that it corresponds to the array containing the selectable
      //  colors.

      current_color = rgb_color_specifications[ index_of_selected_color ] ;

      midlet_display.setCurrent( this ) ;
   }
   else if ( given_command  ==  exit_command )
   {
      //  With the following method call this midlet informs the
      //  runtime system that this method is ready for destruction.
      //  The runtime system does not call the destroyApp() method
      //  before the destruction operation.

      master_midlet.notifyDestroyed() ;
   }
}
```

This statement will be executed after the user has pressed the Soft Key that represents the Change color command. The color selection menu will be the new display content.

`List.SELECT_COMMAND` is a kind of automatic command that is generated when a selection is made on a list that is specified as **`List.IMPLICIT`**. Here we start using a new selected color. The color selection menu is removed from the display as "this" **`Canvas`**-based object is set as display content.

MovingBallMIDlet.java - 2: The second part of class MovingBallCanvas.

A method named **keyPressed()** will be called by the runtime system when this **MovingBallCanvas** object is the display content. The ball coordinates are modified so that the ball appears to move when the arrow keys are pressed.

```
    public void keyPressed( int key_code )
    {
       int game_action_code = getGameAction( key_code ) ;

       switch ( game_action_code )
       {
       case UP:
           ball_position_y  -=  3 ;
           break;

       case DOWN:
           ball_position_y  +=  3 ;
           break;

       case RIGHT:
           ball_position_x  +=  3 ;
           break;

       case LEFT:
           ball_position_x  -=  3 ;
           break;
       }

       repaint() ;
    }
```

Before the ball is drawn with **fillArc()** method, a kind of frame is drawn around the canvas with **drawRect()** method.

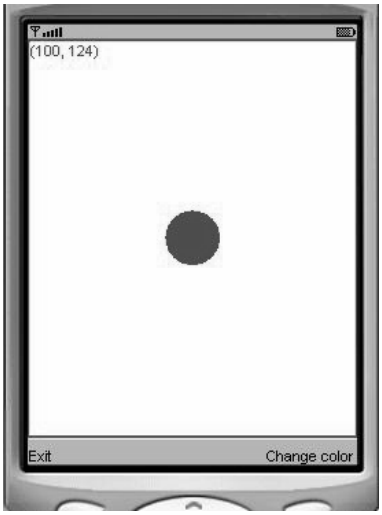
```
    protected void paint( Graphics graphics )
    {
       graphics.setColor( 255, 255, 255 ) ;  //  white
       graphics.fillRect( 0, 0, getWidth(), getHeight() ) ;


       graphics.setColor( current_color ) ;

       graphics.drawRect( 0, 0, getWidth() - 1, getHeight() - 1 ) ;

       graphics.fillArc( ball_position_x, ball_position_y,
                         40, 40, 0, 360 ) ;

       graphics.drawString(  "(" + ball_position_x
                           + ", "  +  ball_position_y  +  ")",
                            2, 0,
                            Graphics.TOP | Graphics.LEFT ) ;
    }
}
```

MovingBallMIDlet.java - 3: The third and last part of class MovingBallCanvas.

When a reference to "this" **MIDlet**-based class is passed as a parameter to the **MovingBallCanvas** constructor, it will be possible inside the **MovingBallCanvas** class to invoke methods for "this" midlet object.

```
public class MovingBallMIDlet extends MIDlet
{
   Display           midlet_display      =  Display.getDisplay( this ) ;
   MovingBallCanvas  moving_ball_canvas  =
                        new MovingBallCanvas( this, midlet_display ) ;

   public MovingBallMIDlet()
   {
   }

   protected void startApp() throws MIDletStateChangeException
   {
      midlet_display.setCurrent( moving_ball_canvas ) ;
   }

   protected void pauseApp()
   {
   }

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

MovingBallMIDlet.java - 4. The actual MIDlet-based class of the program.



The left picture shows the midlet right after it has started executing. When a ball is visible on the screen, it is possible to move it with the arrow keys. When the Soft Key that represents the "Change color" command is pressed, the display will contain a color selection list as shown by the right picture. Note that there are no Soft Key commands attached to the color selection list. A selection can be made with the "Select" key of the phone.

MovingBallMIDlet.java - X. The two possible display contents.

RandomNumbersMIDlet.java – using classes Random ja ChoiceGroup

This program shows how a kind of Settings menu can be created by using standard classes **Form**, **ChoiceGroup**, and **Command**.

ChoiceGroup objects can be attached to a **Form** object. The parameters such as **Choice.EXCLUSIVE** or **Choice.MULTIPLE** specify how individual choices of a group affect other choices in the same group.

```
//  RandomNumbersMIDlet.java  (c) Kari Laitinen

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

class RandomNumbersCanvas extends Canvas
                          implements CommandListener
{
   Display  midlet_display ;

   int  generated_random_integer  =  0 ;
   double  generated_random_double  =  0 ;

   Form  settings_form  =  new  Form( "SETTINGS" ) ;

   String[]  integer_ranges  =  { "0 ... 9", "0 ... 99", "0 ... 999" } ;
   ChoiceGroup  integer_range_selection  =  new  ChoiceGroup(
                                            "Range for random integers:",
                                            Choice.EXCLUSIVE,
                                            integer_ranges, null ) ;

   String[]  double_selection_text  =  { "Show only a double value:" } ;
   ChoiceGroup  double_selection  =  new ChoiceGroup( "Generate random double:",
                                                      Choice.MULTIPLE,
                                                      double_selection_text,
                                                      null ) ;
   Command  command_to_make_settings  =
                       new  Command( "Settings", Command.SCREEN, 1 ) ;
   Command  command_to_exit_settings  =
                       new  Command( "Exit settings", Command.SCREEN, 1 ) ;

   public RandomNumbersCanvas( Display given_display )
   {
      midlet_display  =  given_display ;

      settings_form.append( integer_range_selection ) ;
      settings_form.append( double_selection ) ;
      settings_form.addCommand( command_to_exit_settings ) ;
      settings_form.setCommandListener( this ) ;

      addCommand( command_to_make_settings ) ;
      setCommandListener( this ) ;
   }
```

The command that is used to activate the Settings menu is attached to canvas. The command with which we exit the Settings mode is attached to the Settings form.

RandomNumbersMIDlet.java - 1: RandomNumbersCanvas data fields and constructor.

This statement sets "this" object as display content, which means that the settings form is removed from the display and the canvas is brought back.

This midlet generates a random number always after a numerical key has been pressed. First it examines whether the generation of a **double** random number is enabled. If not, it generates a random integer within the selected range.

```
public void commandAction( Command     given_command,
                           Displayable current_display_content )
{
   if ( given_command  ==  command_to_make_settings )
   {
      midlet_display.setCurrent( settings_form ) ;
   }
   else if ( given_command  ==  command_to_exit_settings )
   {
      midlet_display.setCurrent( this ) ;
   }
}

public void keyPressed( int key_code )
{
   if ( key_code  >=  '0'  &&  key_code  <=  '9' )
   {
      Random  random_number_generator  =  new  Random() ;

      if ( double_selection.isSelected( 0 ) )
      {
         generated_random_double  =
                       random_number_generator.nextDouble() ;
      }
      else if ( integer_range_selection.getSelectedIndex()  ==  0 )
      {
         generated_random_integer  =
                     random_number_generator.nextInt( 10 ) ;
      }
      else if ( integer_range_selection.getSelectedIndex()  ==  1 )
      {
         generated_random_integer  =
                     random_number_generator.nextInt( 100 ) ;
      }
      else if ( integer_range_selection.getSelectedIndex()  ==  2 )
      {
         generated_random_integer  =
                     random_number_generator.nextInt( 1000 ) ;
      }
   }

   repaint() ;
}
```

Random numbers can be generated with methods **nextDouble()** and **nextInt()** after a random number generator of type **Random** has been created. **nextDouble()** returns a random **double** value that is greater than or equal to zero and smaller than but not equal to one.

RandomNumbersMIDlet.java - 2: Methods commandAction() and keyPressed().

The **paint()** method prints the random value generated in the **keyPressed()** method. Either the content of data field **generated_random_integer** or **generated_random_double** is printed depending on the settings made. Method **isSelected()** of class **ChoiceGroup** allows us to examine whether a certain choice inside a **ChoiceGroup** is selected. The parameter that is given to the **isSelected()** method is the index of the choice. A zero refers to the first choice. Here we examine a **ChoiceGroup** that contains only a single choice and that is referred to with index value 0.

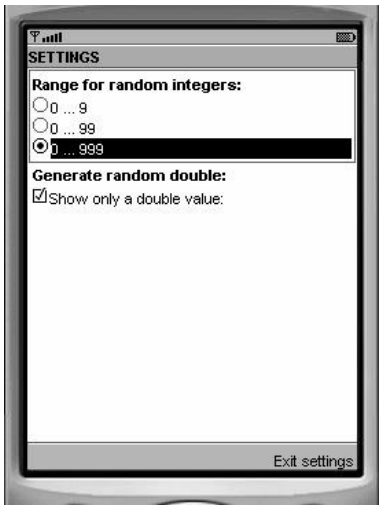
```
   protected void paint( Graphics graphics )
   {
      graphics.setColor( 255, 255, 255 ) ;  //  White color
      graphics.fillRect( 0, 0, getWidth(), getHeight() ) ;

      graphics.setColor( 0, 0, 0 ) ;  //  Black color

      graphics.drawString( "LAST GENERATED RANDOM NUMBER:",
                           10, 20, Graphics.TOP | Graphics.LEFT ) ;

      if ( double_selection.isSelected( 0 ) )
      {
         graphics.drawString( ""  +  generated_random_double,
                              10, 40, Graphics.TOP | Graphics.LEFT ) ;
      }
      else
      {
         graphics.drawString( ""  +  generated_random_integer,
                              10, 40, Graphics.TOP | Graphics.LEFT ) ;
      }
   }
}
```

RandomNumbersMIDlet.java - 3: The last method paint() of class RandomNumbersCanvas.

This class is again short because all functionality of the program is built into the **`Canvas`**-based class. A reference to the **`Display`** object is passed as a parameter to the constructor of **`RandomNumbers-Canvas.`** This way the methods of the canvas class can change the content of the display.

```
public class RandomNumbersMIDlet extends MIDlet
{
   Display  midlet_display  =  Display.getDisplay( this ) ;
   RandomNumbersCanvas  random_numbers_canvas  =
                          new RandomNumbersCanvas( midlet_display ) ;

   protected void startApp() throws MIDletStateChangeException
   {
      midlet_display.setCurrent( random_numbers_canvas ) ;
   }

   protected void pauseApp()
   {
   }

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

RandomNumbersMIDlet.java - 4. The short MIDlet-based class of the program.



The generation of **`double`** random numbers is selected here. The generated **`double`** values are smaller than 1 and larger or equal to 0.

RandomNumbersMIDlet.java - X. The canvas and the Settings menu of the program.

ClockMIDlet.java – a midlet that runs an extra thread

```
//  ClockMIDlet.java  Copyright (c) Kari Laitinen

import java.util.* ;

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


class ClockCanvas extends Canvas
                  implements Runnable
{
   Thread   thread_that_runs_the_clock ;
   boolean  thread_must_be_executed  =  false ;

   Calendar   time_now ;

   int  canvas_width, canvas_height ;

   int  clock_center_point_x, clock_center_point_y  ;

   public ClockCanvas()
   {
      canvas_width  =  getWidth() ;
      canvas_height =  getHeight() ;

      clock_center_point_x  =  canvas_width / 2 ;
      clock_center_point_y  =  canvas_height / 2 + 10 ;

   }

   public synchronized void start_animation_thread()
   {
      if ( thread_that_runs_the_clock  ==  null )
      {
         thread_that_runs_the_clock  =  new  Thread( this ) ;
         thread_must_be_executed  =  true ;
         thread_that_runs_the_clock.start() ;
      }
   }
```

A **Thread** object is created and set to run in parallel with the midlet. The extra thread starts executing automatically after the **start()** method is invoked for the **Thread**. The runtime system calls the **run()** method after the call to the **start()** method is executed. The **run()** method then represents the extra thread. A reference to "this" **ClockCanvas** object is passed as a parameter when the **Thread** object is created. This way the runtime system knows where the **run()** method is located.

ClockMIDlet.java - 1: A program that displays a clock that runs.

Methods **start_animation_thread()** and **stop_animation_thread()** are called from the **destroyApp()**, **pauseApp()** and **startApp()** methods of the **ClockMIDlet** class. The extra thread is terminated when the midlet is put to paused state or destroyed altogether.

The extra thread will terminate when **thread_must_be_executed** is assigned the value **false**. This causes the **while** loop inside the **run()** method terminate. The extra thread "dies" when the **run()** method terminates. By calling the **interrupt()** method for the **Thread** object, is is ensured that the extra thread is "awoken to die" in the case it happens to be sleeping.

```
public void stop_animation_thread()
{
   if ( thread_that_runs_the_clock  !=  null )
   {
      thread_must_be_executed  =  false ;
      thread_that_runs_the_clock.interrupt() ;

      thread_that_runs_the_clock  =  null ;
   }
}


public void run()
{
   while ( thread_must_be_executed  ==  true )
   {
      repaint() ;

      try
      {
         Thread.sleep( 1000 ) ;  // Suspend for 1 second.
      }
      catch ( InterruptedException caught_exception )
      {
         // No actions to handle the exception.
      }
   }
}
```

Method **run()** is called automatically after the thread has been created and activated. Method **run()** specifies what the additional thread does. This **run()** method orders the canvas to be repainted, and then it goes to sleep for one second. After each slept second, these activities are repeated. After the **repaint()** method is executed, the runtime system generates a call to the **paint()** method of the canvas. The static **Thread.sleep()** method must be called inside a **try-catch** constructs because it can throw an **InterruptedException**.

ClockMIDlet.java - 2: Methods in class ClockCanvas.

The current time and date of the mobile phone can be found out by creating a **Calendar** object. As this **paint()** method is invoked to draw the clock once in every second, we get an illusion of a clock that runs.

A method named **get()** can be used to receive time-related information from the **Calendar** object. Parameters such as **Calendar.YEAR**, **Calendar.MONTH**, etc. are needed to specify the information that is requested. These parameters are specified in class **Calendar**.

```
public void paint( Graphics graphics )
{
   String[]  days_of_week  =  { "Sun", "Mon", "Tue",
                                "Wed", "Thu", "Fri", "Sat" } ;

   String[]  names_of_months  =  { "Jan", "Feb", "Mar", "Apr",
                                   "May", "Jun", "Jul", "Aug",
                                   "Sep", "Oct", "Nov", "Dec" } ;

   time_now  =  Calendar.getInstance() ;

   int current_year  =  time_now.get( Calendar.YEAR ) ;
   int current_day   =  time_now.get( Calendar.DAY_OF_MONTH ) ;
   int month_index   =  time_now.get( Calendar.MONTH ) ;
   int number_of_day_of_week  =  time_now.get( Calendar.DAY_OF_WEEK ) ;

   String current_month  =  names_of_months[ month_index ] ;

   String current_day_of_week =  days_of_week[ number_of_day_of_week - 1 ] ;

   int current_hours    =  time_now.get( Calendar.HOUR_OF_DAY ) ;
   int current_minutes  =  time_now.get( Calendar.MINUTE ) ;
   int current_seconds  =  time_now.get( Calendar.SECOND ) ;

   graphics.setColor( 255, 255, 255 ) ;  // white
   graphics.fillRect( 0, 0, canvas_width, canvas_height ) ;

   graphics.setColor( 0, 0, 0 ) ;  // black

   graphics.drawString( ""   + current_day_of_week  +
                        " "  + current_month +
                        " "  + current_day   +
                        ", " + current_year,
                        2, 0, Graphics.TOP | Graphics.LEFT ) ;
```

ClockMIDlet.java - 3: The first part of the paint() method in class ClockCanvas.

The clock time is shown also in textual form. These statements ensure that a leading zero is printed before single-digit minute and seconds values. This means that the time "five minutes and three seconds past seven" is written 7:05:03, and not 7:5:3.

These initialized arrays contain coordinates that will be used to determine possible end points for the clock hands. The coordinates are relative to the clock center point.

```
String  minutes_string  =  "00" + current_minutes ;

minutes_string  =   minutes_string.substring(
                            minutes_string.length() - 2,
                            minutes_string.length()  ) ;

String  seconds_string  =  "00" + current_seconds ;

seconds_string  =   seconds_string.substring(
                            seconds_string.length() - 2,
                            seconds_string.length()  ) ;


graphics.drawString( current_hours + ":" + minutes_string  +
                     ":" + seconds_string,
                     2, 12, Graphics.TOP | Graphics.LEFT ) ;


/*  The following coordinates were originally developed for a
    larger clock on a larger display. In this program they are
    divided by 3 in order to get coordinates that are suitable
    for smaller displays. */

int[]  minute_hand_end_points_x  =

  {  0,  11,  21,  31,  41,  50,  59,  67,  74,  81,
    87,  91,  95,  97,  99,
   100,  99,  97,  95,  91,  87,  81,  74,  67,  59,
    50,  41,  31,  21,  11,
     0, -11, -21, -31, -41, -50, -59, -67, -74, -81,
   -87, -91, -95, -97, -99,
  -100, -99, -97, -95, -91, -87, -81, -74, -67, -59,
   -50, -41, -31, -21, -11 } ;

int[]  minute_hand_end_points_y  =

{ -100, -99, -97, -95, -91, -87, -81, -74, -67, -59,
   -50, -41, -31, -21, -11,
     0,  11,  21,  31,  41,  50,  59,  67,  74,  81,
    87,  91,  95,  97,  99,
   100,  99,  97,  95,  91,  87,  81,  74,  67,  59,
    50,  41,  31,  21,  11,
     0, -11, -21, -31, -41, -50, -59, -67, -74, -81,
   -87, -91, -95, -97, -99 } ;
```

ClockMIDlet.java - 4: The paint() method of class ClockCanvas continues.

For the hour hand we have different coodinates as it is shorter than the other clock hands.

```
int[]  hour_hand_end_points_x  =

  {  0,   7,  13,  19,  24,  30,  35,  40,  44,  48,
    52,  55,  57,  58,  59,
    60,  59,  58,  57,  55,  52,  48,  44,  40,  35,
    30,  24,  19,  13,   7,
     0,  -7, -13, -19, -24, -30, -35, -40, -44, -48,
   -52, -55, -57, -58, -59,
   -60, -59, -58, -57, -55, -52, -48, -44, -40, -35,
   -30, -24, -19, -13,  -7 } ;

int[]  hour_hand_end_points_y  =

 { -60, -59, -58, -57, -55, -52, -48, -44, -40, -35,
   -30, -24, -19, -13,  -7,
     0,   7,  13,  19,  24,  30,  35,  40,  44,  48,
    52,  55,  57,  58,  59,
    60,  59,  58,  57,  55,  52,  48,  44,  40,  35,
    30,  24,  19,  13,   7,
     0,  -7, -13, -19, -24, -30, -35, -40, -44, -48,
   -52, -55, -57, -58, -59 } ;

// Let's print an 8-point dot in the center of the clock.

graphics.fillArc( clock_center_point_x - 4,
                  clock_center_point_y - 4, 8, 8, 0, 360 ) ;

// The following loop prints dots on the clock circle.

int minute_index  =  0 ;

while ( minute_index  <  60 )
{
   graphics.fillArc(

      clock_center_point_x +
      minute_hand_end_points_x[ minute_index ] / 3  - 2,

      clock_center_point_y +
     minute_hand_end_points_y[ minute_index ] / 3  - 2, 4, 4, 0, 360 ) ;

   minute_index  =  minute_index  +  5  ;
}
```

As the above arrays of clockface coordinates were originally developed for a larger clock, the coordinates are here divided by 3 in order to make them suitable for a small clock on mobile phone display.

ClockMIDlet.java - 5: More of the paint() method of class ClockCanvas.

With these statements it is decided where end of the hour hand should be on the circle that has 60 possible positions. Whether the current time is before noon or after noon, and how many minutes of the current hour have elapsed, affect the positioning of the hour hand.

The clock hands are drawn with the **`drawLine()`** method. To draw the hour hand, we use hour coordinates. Minute hand is drawn with minute coordinates.

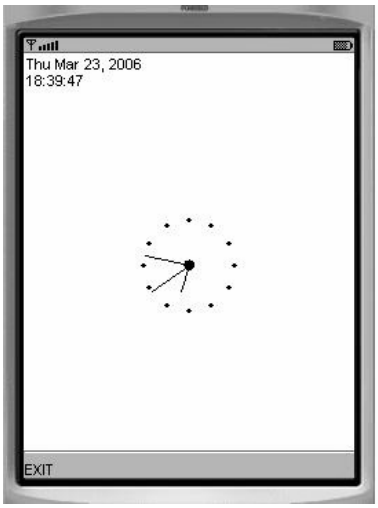
```
        int  hour_index  ;

        if ( current_hours  >=  12 )
        {
           hour_index  =  current_hours  -  12 ;
        }
        else
        {
           hour_index  =  current_hours ;
        }

        //  Remember that we have 60 minutes in every hour,
        //  but not 60 hours in a day.

        hour_index  =  hour_index  *  5  +  current_minutes  /  12 ;

        //  Let's draw the hour hand of the clock.

        graphics.drawLine(  clock_center_point_x,
                            clock_center_point_y,

                            clock_center_point_x  +
                                hour_hand_end_points_x[ hour_index ] / 3,
                            clock_center_point_y  +
                                hour_hand_end_points_y[ hour_index ] / 3 ) ;

        //  The minute and second hands are longer than the hour hand.
        //  Therefore we use different coordinates to print them.

        graphics.drawLine(  clock_center_point_x,
                            clock_center_point_y,

                            clock_center_point_x  +
                                minute_hand_end_points_x[ current_minutes ] / 3,
                            clock_center_point_y  +
                              minute_hand_end_points_y[ current_minutes ] / 3 );

        graphics.drawLine(  clock_center_point_x,
                            clock_center_point_y,

                            clock_center_point_x  +
                                minute_hand_end_points_x[ current_seconds ] / 3,
                            clock_center_point_y  +
                              minute_hand_end_points_y[ current_seconds ] / 3 );

    }
}
```

ClockMIDlet.java - 6: The last part of class ClockCanvas.

```
public class ClockMIDlet extends MIDlet
                         implements CommandListener
{
   Display      midlet_display  =  Display.getDisplay( this ) ;
   ClockCanvas  clock_canvas    =  new  ClockCanvas() ;

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

   protected void startApp() throws MIDletStateChangeException
   {
      midlet_display.setCurrent( clock_canvas ) ;
      clock_canvas.start_animation_thread() ;

      clock_canvas.addCommand( exit_command ) ;
      clock_canvas.setCommandListener( this ) ;
   }

   protected void pauseApp()
   {
      clock_canvas.stop_animation_thread() ;
   }

   protected void destroyApp( boolean unconditional_destruction_required )
   {
      clock_canvas.stop_animation_thread() ;
   }

   public void commandAction( Command      given_command,
                              Displayable  display_content )
   {
      if ( given_command  ==  exit_command )
      {
         destroyApp( false ) ;
         notifyDestroyed() ;
      }
   }
}
```

Also in the case of this midlet most of the program functionality is programmed inside the **`Canvas`**-based **`ClockCanvas`** class. Inside this **`ClockMIDlet`** class we activate and deactivate the extra thread that runs the clock. The methods **`start_animation_thread()`** and **`stop_animation_thread()`**, which start and stop the extra thread are inside the **`ClockCanvas`** class.

ClockMIDlet.java - 7. The MIDlet-based class of the program.



Here the clock is rather small when compared to the display size. To make the clock larger, you could divide the clockface coordinates, for instance, by 2 instead of 3.

ClockMIDlet.java - X. The midlet is being exexuted on March 23, 2006 at 6:39:47 p.m.