J2ME Chapter 1. Scrollable Messages

The ScrollMBDemo MIDlet is a test rig for a CustomItem subclass called ScrollableMessagesBox. ScrollableMessagesBox displays a list of messages, and allows the user to scroll up and down through them. Since the messages box is a CustomItem, it can be easily integrated into forms alongside other items.

Figure 1 shows ScrollMBDemo in action.

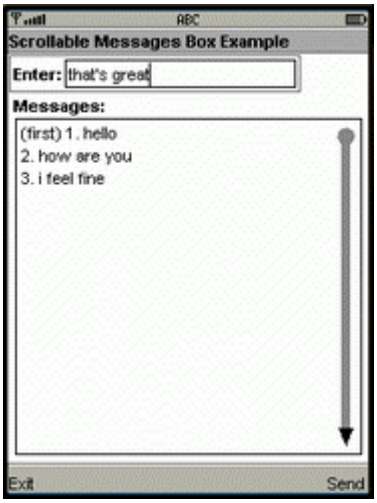

Figure 1. ScrollMBDemo with Some Messages.

The user types a message into the "Enter:" textfield at the top of the form, presses the "Send" command, and the message is added to the messages box. The messages are automatically numbered, starting from 1.

Once the box has filled up, the messages start scrolling upwards, so the latest messages remain visible (see Figure 2).

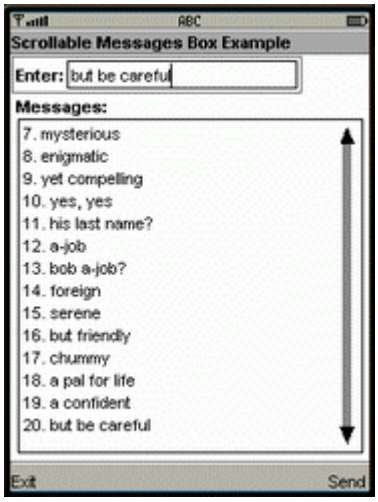

Figure 2. ScrollMBDemo with Many Messages.

Two commands, "Up" and "Down", appear in the commands menu when the user tabs into the messages box. The commands permit the user to scroll up and down through the messages list. Scrolling is also possible by pressing the up and down keys (which are mapped to the '2' and '8' keys in Sun's WTK). Fast scrolling is available by holding down the up and down keys.

Figure 3 shows the messages list from Figure 2 after the user has scrolled up to the beginning of the list.

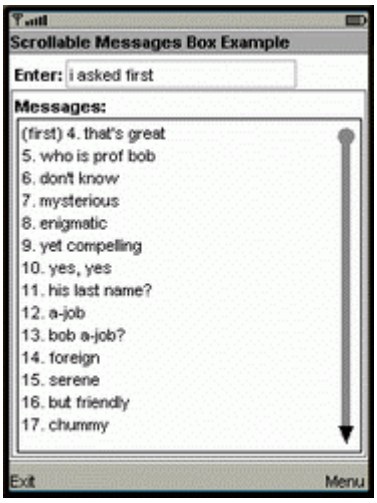

Figure 3. Scrolling upwards in ScrollMBDemo.

The box only stores a limited number of messages (roughly 1.5 times the number of lines that can be drawn in the box). When this limit is reached, older messages are discarded to make room for new ones. This is why message number 4 is labeled with "(first)" in Figure 3: the older messages (1-3) have been deleted.

When scrolling is available in the up direction, an arrow is drawn at the top of the scrollbar on the right of the box (see Figure 2). Scrolling upwards is disabled when the first message is displayed at the top of the box. The arrow changes to a circle (as in Figure 3), and the "Up" command is removed from the commands menu.

When downwards scrolling is enabled, an arrow appears at the bottom of the scrollbar (as in Figure 3). Scrolling downwards is switched off when the current message is positioned at the very top of the box. The downwards arrow is replaced by a circle, and the "Down" command is removed from the commands menu.

A drawback of ScrollableMessagesBox is that scrolling is controlled by commands and keys only; it's not possible to click on an arrow, or drag in the bar, to trigger text movement.

The ScrollableMessagesBox class is used in several of our J2ME network examples, as a convenient way of displaying the lengthy communication between a user and other participants in the system. [Note: these examples are not yet available online (August 2005). Sorry.]

ScrollableMessagesBox is also a reasonably-sized example of how to utilize CustomItem. It illustrates how to specify an item's size, draw to its canvas, catch key presses, and respond to item-specific commands. Two aspects of CustomItem which

aren't present are off-screen editing and traversal, but I'll talk about those briefly at the end of the chapter.

1. Class Diagrams for ScrollMBDemo

Figure 4 shows the class diagrams for the ScrollMBDemo application. The class names and public/protected methods are shown.

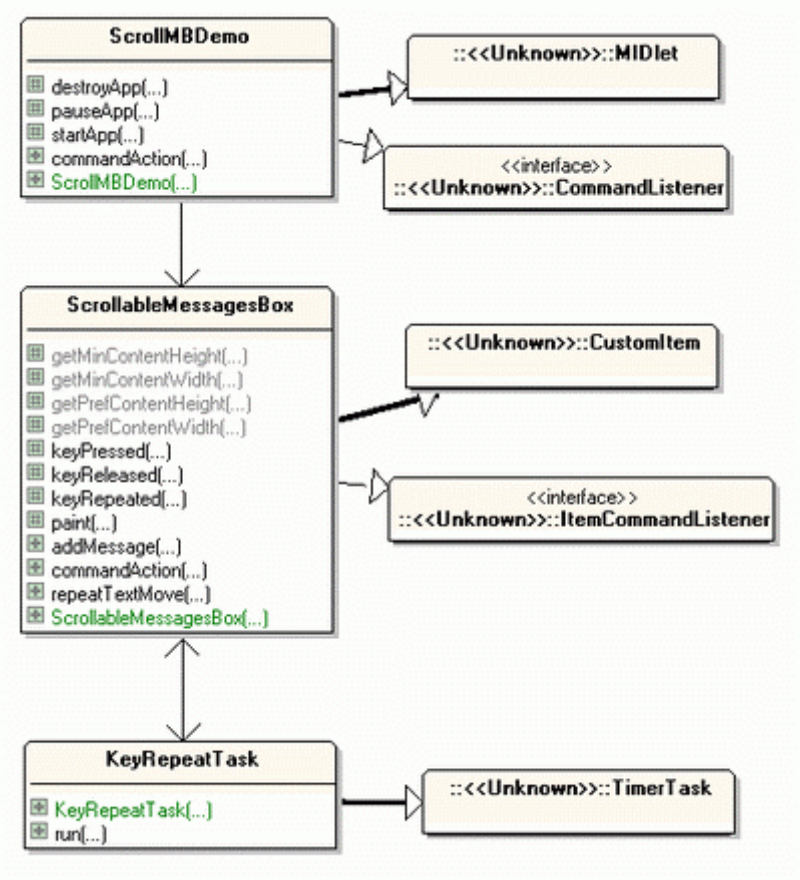

Figure 4. Class Diagrams for ScrollMBDemo.

The ScrollMBDemo MIDlet sets up a form containing a textfield for entering messages and a ScrollableMessagesBox for displaying them. KeyRepeatTask provides key repetition to speed up scrolling. KeyRepeatTask is unnecessary if the platform natively supports key repetition, an issue I'll discuss in section 3.5.

2. Testing ScrollableMessagesBox

ScrollMBDemo is a test harness for ScrollableMessagesBox's functionality. It starts by creating a textfield and a ScrollableMessagesBox instance, scroller:

```
// globals
private Display display;
```



```
private Form form;
private TextField msgTF;
private ScrollableMessagesBox scroller;
private Command exitCmd, sendCmd;


public ScrollMBDemo()
{
  form = new Form("Scrollable Messages Box Example");

  // get the form's dimensions
  int width = form.getWidth();
  int height = form.getHeight();

  msgTF = new TextField("Enter:", null, 15, 0);
  int msgTFHeight = msgTF.getPreferredHeight();
       // height of textfield

  scroller = new ScrollableMessagesBox("Messages:",
                                       width, height-msgTFHeight-20);
       // 20 is for the "Messages:" title and a bit of space

  form.append(msgTF);
  form.append(scroller);

  exitCmd = new Command("Exit", Command.EXIT, 1);
  sendCmd = new Command("Send", Command.SCREEN, 1);
  form.addCommand(exitCmd);
  form.addCommand(sendCmd);

  form.setCommandListener(this);
}  // end of ScrollMBDemo()
```

The only unusual aspect of ScollMBDemo() is the calculation of the ScrollableMessagesBox's width and height, which are passed to its constructor:

```
scroller = new ScrollableMessagesBox("Messages:",
                              width, height-msgTFHeight-20);
```

The width and height values are from the form, and msgTFHeight is the height of the textfield. The '20' is a guess, representing the probable height of CustomItem's label (the "Messages:" string).

The dimensions are used to size ScrollableMessagesBox's drawable area, which excludes its label. The box will be as wide as the form, and have a height that spans all the form below the input text field and the box's label.

2.1. Adding a Message to the Messages Box

commandAction() connects the "Send" command to ScrollableMessagesBox's addMessage() method:

```
public void commandAction(Command c, Displayable d)
{
  if (c == exitCmd)
    destroyApp(true);
```



```
  else if (c == sendCmd) {
    String msg = msgTF.getString();
    if ((msg != null) && (!msg.equals("")))
      scroller.addMessage(msg);
      // pass the message to ScrollableMessagesBox
  }
} // end of commandAction()
```

The message is added to the end of the list, and automatically prefixed with a number.

3. The ScrollableMessagesBox Class

An important aspect of ScrollableMessagesBox is the link between the data structure it uses to store messages (the lines[] array) and the display of those messages. This is shown in Figure 5.

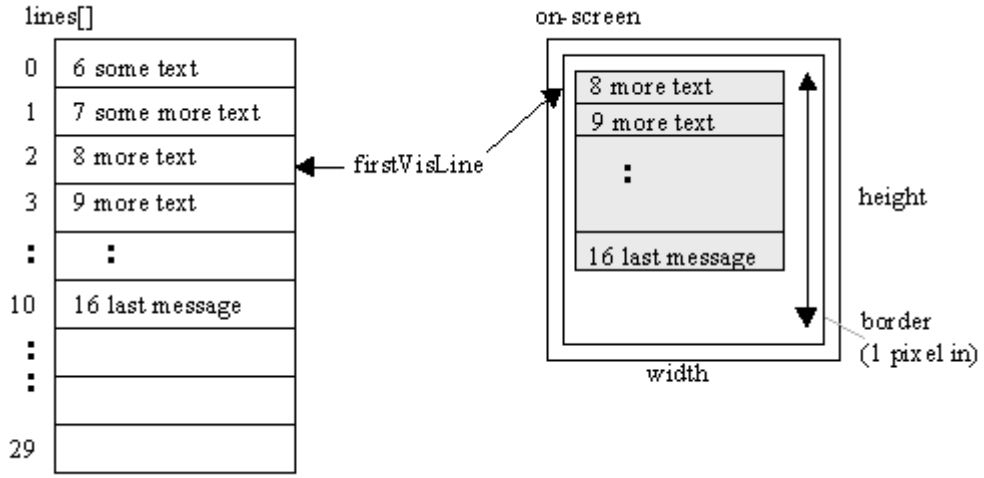

Figure 5. Message Storage and On-screen Appearance.

firstVisLine stores the index of the line that's currently visible at the top of the box. As the messages list is scrolled up or down, this variable's value is changed accordingly.

lines[] is initialized in ScrollableMessagesBox(), with its size determined by the on-screen height of the box.

```
// globals
// offsets for writing text
private static final int XOFFSET = 5;
private static final int YOFFSET = 3;

private static final float LINES_FACTOR = 1.5f;
  /* the factor multiplied to the number of box lines
     to get the number of lines stored in lines[] */


private int width, height;  // of the scrolling box
private int fontHeight;
```



```
private int maxVisLines;    // max no. of visible lines

private String[] lines;     // stores the message lines
private int maxLines;       // max no. of stored lines
private int numLines = 0;   // current number of stored lines

private int firstVisLine = 0;
   /* index of the line that's currently visible at the
      top of the box */


public ScrollableMessagesBox(String title, int w, int h)
{
  super(title);
  width = w;
  height = h;

  Font f = Font.getDefaultFont();
  fontHeight = f.getHeight();

  maxVisLines = (height-(YOFFSET+1))/fontHeight;

  // initialize the lines[] array
  maxLines = (int)(LINES_FACTOR * maxVisLines);
  lines = new String[maxLines];

  // other initialization code, explained later...
}
```

The maximum number of lines that can be drawn inside the box is:

```
  maxVisLines = (height-(YOFFSET+1))/fontHeight;
```

YOFFSET is the offset from the top of the box to where the first line is drawn (see Figure 6). The '1' ensures that the drawing range is inside the 1 pixel border at the bottom of the box.

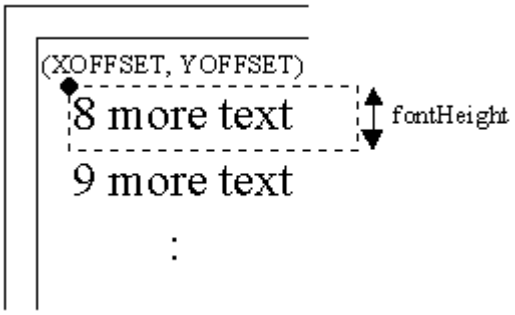

Figure 6. The Drawing Coordinates for the Message Lines.

Each line occupies fontHeight pixels in the vertical direction, so dividing the drawing range by fontHeight produces the number of visible lines.

The actual size of lines[] is a constant multiple of maxVisLines, so the array can store more lines than can be shown on the screen:

```
  maxLines = (int)(LINES_FACTOR * maxVisLines);
  lines = new String[maxLines];
```


3.1. Are the Arrows Showing?

The arrows at the top and bottom of the scrollbar switch to circles when scrolling up or down is disabled. The status of the arrows is stored in two global booleans, upArrowShowing and downArrowShowing, which are initialized in the constructor:

```
// globals
private boolean upArrowShowing, downArrowShowing;

// in the constructor
upArrowShowing = false;
downArrowShowing = false;
```

At start-up, both arrows are disabled because there are no messages in the box. The scrollbar will be drawn with circles at both ends.

3.2. Item Commands

A CustomItem may have its own commands, which appear in the commands menu when the user traverses into the item (typically by tabbing, or by pressing the down arrow key). Item-specific commands are a useful way of supplying additional functionality when it's relevant.

ScrollableMessagesBox has two commands, "Up" and "Down", which are added to the menu when scrolling is enabled. For example, the "Up" command is offered only when the user can scroll upwards in the box; it disappears from the menu when the user has scrolled upwards so the first message is visible at the top of the box.

The commands are initialized in the constructor, and ScrollableMessagesBox is specified as the item command listener. This means that the processing of the commands is self-contained inside the CustomItem.

```
// globals
private Command upCmd, downCmd;

// in the constructor
// create commands, but don't show them yet
upCmd = new Command("Up", Command.ITEM, 1);
downCmd = new Command("Down", Command.ITEM, 1);

setItemCommandListener(this);
  // ScrollableMessagesBox implements ItemCommandListener
```

commandAction() links the commands to two text movement methods, textMovesUp () and textMovesDown(), which I'll describe in sections 3.6 and 3.7.

```
public void commandAction(Command c, Item i)
// use commands to move the text
{ if (c == upCmd)
    textMovesDown();
  else if (c == downCmd)
    textMovesUp();
```



```
}
```

It's worth noting that the "Up" command moves the text down, while "Down" moves it up.

3.3. User Input Modes

The CustomItem class supports both keyboard and pointer input. The possible input types include key pressing, releasing, and repetition, and pointer pressing, releasing, and dragging. Key repetition occurs when a key is pressed down for a long period.

The actual range of supported input mechanisms depends on the device, which can be queried by calling CustomItem.getInteractionModes(). The result is an integer whose bits contain the relevant information. For example, the following methods check if key release and pointer release are supported:

```
private boolean hasKeyRelease()
{  return ((getInteractionModes() & KEY_RELEASE) != 0);  }

private boolean hasPointerRelease()
{  return ((getInteractionModes() & POINTER_RELEASE) != 0);  }
```

A complete list of interaction modes can be found in the CustomItem documentation.

ScrollableMessagesBox doesn't use pointer interaction, and assumes that key presses and releases are available. However, it does check for key repetition support:

```
private boolean hasKeyRepeats()
// are key repeats supported?
{  return ((getInteractionModes() & KEY_REPEAT) != 0);  }
```

3.4. Dealing with Key Presses

A key press automatically generates a call to CustomItem.keyPressed(), which is overridden in ScrollableMessagesBox:

```
// globals
private boolean upKeyPressed = false;
private boolean downKeyPressed = false;


protected void keyPressed(int keyCode)
{
  int gameAct = getGameAction(keyCode);
  if ((gameAct == Canvas.UP) && upArrowShowing) {
    textMovesDown();
    upKeyPressed = true;
  }
  else if ((gameAct == Canvas.DOWN) && downArrowShowing) {
    textMovesUp();
    downKeyPressed = true;
  }
```



```
}
```

Only the 'up' and 'down' keys have any effect, and only if scrolling is currently enabled in that particular direction. As part of my home-grown implementation of key repetition (explained in the next section), the upKeyPressed and downKeyPressed booleans record which direction key was pressed.

Releasing a key triggers a call to CustomItem.keyReleased(), which is overridden as:

```
protected void keyReleased(int keyCode)
{
  int gameAct = getGameAction(keyCode);
  if ((gameAct == Canvas.UP) && upKeyPressed)
    upKeyPressed = false;
  else if ((gameAct == Canvas.DOWN) && downKeyPressed)
    downKeyPressed = false;
}
```

keyReleased()'s only job is to reset the upKeyPressed and downKeyPressed booleans.

3.5. Dealing with Key Repetition

If the system supports key repetition, then CustomItem.keyRepeated() is called repeatedly while a key is held down. Its version in ScrollableMessagesBox is very similar to keyPressed():

```
protected void keyRepeated(int keyCode)
{
  int gameAct = getGameAction(keyCode);
  if ((gameAct == Canvas.UP) && upArrowShowing)
    textMovesDown();
  else if ((gameAct == Canvas.DOWN) && downArrowShowing)
    textMovesUp();
}
```

Since key repetition isn't offered by every device, ScrollableMessagesBox() checks for its support by calling setKeyRepetition():

```
// globals
// key repeating timer and task
private Timer timer;
private KeyRepeatTask keyRepeatTask;


private void setKeyRepetition()
{ if(!hasKeyRepeats())  // key repeat not supported
    // start a timer to carry out key repeats
    keyRepeatTask = new KeyRepeatTask(this);
    timer = new Timer();
    timer.schedule(keyRepeatTask, 0, KEY_DELAY);
  }
}
```


If key repetition isn't available then a timer is employed to periodically run a KeyRepeatTask object.

Unfortunately, there's a problem (at least with Sun's WTK 2.2 on Windows 98). On that platform, hasKeyRepeats() returns true, indicating that key repetition is supported by the emulator. However, no matter how long I press a key, CustomItem.keyRepeated() is never called.

My solution was to comment out the hasKeyRepeats() test in setKeyRepetition(), and *always* use the timer task to generate repetitions.

The KeyRepeatTask class is very short.

```
public class KeyRepeatTask extends TimerTask
{
  private ScrollableMessagesBox scroller;

  public KeyRepeatTask(ScrollableMessagesBox smb)
  {  scroller = smb;  }

  public void run()
  { scroller.repeatTextMove(); }

} // end of KeyRepeatTask class
```

Its only task is to call repeatTextMove() back in ScrollableMessagesBox:

```
public void repeatTextMove()
{ if (upKeyPressed && upArrowShowing)
    textMovesDown();
  else if (downKeyPressed && downArrowShowing)
    textMovesUp();
}
```

repeatTextMove() repeats a text move depending on the currently pressed key and on scrolling availability.

3.6. Moving the Messages Up

The messages are moved *up* either by the "Down" command being selected (processed by commandAction()) or the 'down' key being pressed (dealt with by keyPressed() and repeatTextMove()). This idea is illustrated in Figure 7.

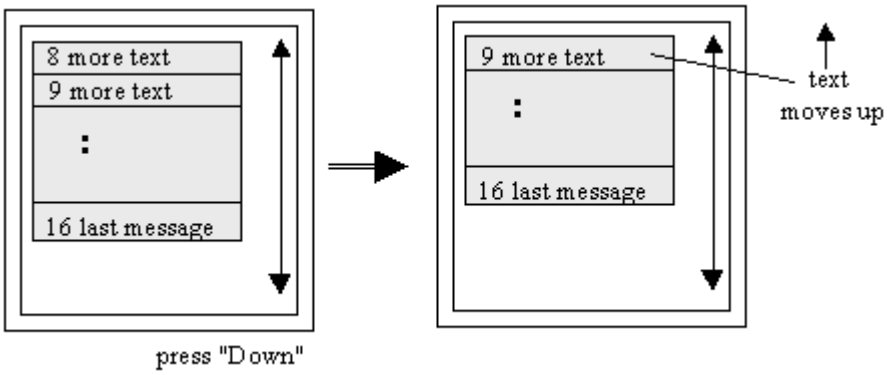

Figure 7. Press "Down" to Move the Messages Up.

Moving the text up by one line is handled by incrementing the firstVisLine integer. For the Figure 7 example, it would start with the value 8, and be changed to 9.

There are a few other aspects to consider: the move request should be ignored when the last line is already at the top of the screen, and the up and down arrows may need to be changed. These tasks are carried out by textMovesUp():

```
private void textMovesUp()
{
  if (firstVisLine < numLines-1) {   // if not already at end of msgs
    firstVisLine++;
    if (firstVisLine == numLines-1) {  // showing last message at top
      removeCommand(downCmd);     // disable downward scrolling
      downArrowShowing = false;
    }
    if (!upArrowShowing) {
      addCommand(upCmd);          // enable upward scrolling
      upArrowShowing = true;
    }
    repaint();
  }
}  // end of textMovesUp()
```

The call to repaint() at the end of textMovesUp() causes paint() to use the new firstVisLine value, and the current values of downArrowShowing and upArrowShowing, to redraw the messages box.

3.7. Moving the Messages Down

The messages are moved down either by the "Up" command being selected or the 'up' key being pressed. This is illustrated in Figure 8.

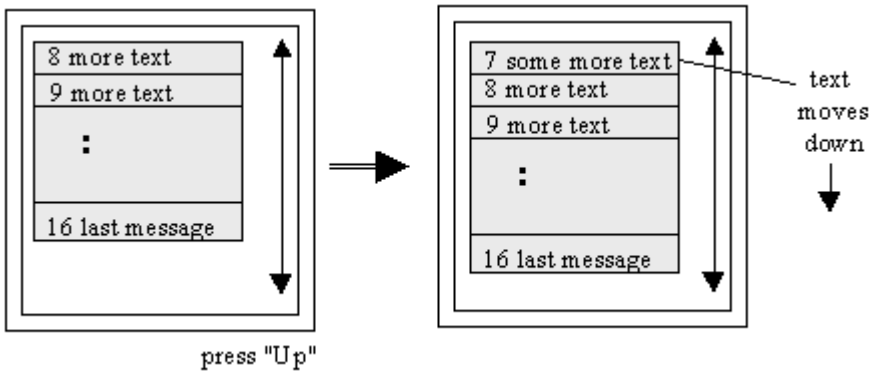

Figure 8. Press "Up" to Move the Messages Down.

The textMovesDown() method is similar in concept to textMovesUp(): it decrements firstVisLine, but only if the first message isn't at the top of the screen. It also adjusts the downArrowShowing and upArrowShowing booleans.

```
private void textMovesDown()
{
  if (firstVisLine > 0) {  // if not already at messages start
    firstVisLine--;
    if (firstVisLine == 0) {  // showing first message at top
      removeCommand(upCmd);   // disable upward scrolling
      upArrowShowing = false;
    }
    if (!downArrowShowing) {
      addCommand(downCmd);    // enable downward scrolling
      downArrowShowing = true;
    }
    repaint();
  }
} // end of textMovesDown()
```

3.8. Painting the Messages Box

The messages box consists of three elements: the background (including the border), the visible messages, and the scrollbar. paint() renders the first two itself, and calls drawScrollBar() for the third task:

```
protected void paint(Graphics g, int w, int h)
{
  // a white background with a black border
  g.setColor(255, 255, 255);   // white background
  g.fillRect(0, 0, w, h);
  g.setColor(0,0,0);           // black border
  g.drawRect(1, 1, w-2, h-2);

  drawScrollBar(g, w, h);
```



```
  int yPos = YOFFSET;

  // calculate the index of the first invisible line
  int invisLine = firstVisLine + maxVisLines;
  int firstInvisLine = (numLines < invisLine) ? numLines : invisLine;

  // write the visible lines onto the canvas
  for (int i = firstVisLine; i < firstInvisLine; i++) {
    if (i == 0)  // this line is the first saved line
      g.drawString("(first) " + lines[i], XOFFSET, yPos,
                                  Graphics.TOP|Graphics.LEFT);
    else
      g.drawString(lines[i], XOFFSET, yPos,
                                  Graphics.TOP|Graphics.LEFT);
    yPos += fontHeight;
  }
} // end of paint()
```

The two integer arguments to paint() (w and h) are the width and height of the drawing area. All the drawing operations should utilize these rather than assume some fixed size. For example, the black border one pixel in from the edges of the drawing area is created using:

```
  g.drawRect(1, 1, w-2, h-2);
```

The drawing of the messages uses the XOFFSET and YOFFSET constants, and positions each line fontHeight pixels below the one above it, as shown in Figure 6.

It's easy to know which line to start with (the one indexed by firstVisLine), but somewhat more tricky to know when to stop. There are two cases to consider:

- stop when the bottom of the drawing area is reached, or
- stop after the drawing loop writes out the last line in lines[].

The bottom of the drawing area is calculated indirectly by adding the maximum number of visible lines that can be drawn (maxVisLines) to the index of the first visible line (firstVisLine):

```
  int invisLine = firstVisLine + maxVisLines;
```

invisLine is the index of the line that would be drawn off the bottom of the messages box, and so be invisible.

The second case stops the drawing if there are less lines stored in lines[] than can fill the drawing area.

The choice between the two stopping cases is decided by comparing numLines (the number of lines in lines[]) with invisLine to see which is smaller. The result is stored in firstInvisLine:

```
  int firstInvisLine = (numLines < invisLine) ? numLines : invisLine;
```

firstInvisLine is used as the stopping value for the loop that prints the lines.

3.9. Drawing the Scrollbar

The scrollbar is a double-headed vertical arrow drawn close to the right hand side of the messages box. The upwards and downwards arrowheads (triangles) change to circles when it's not possible to scroll the messages up or down.

```
private void drawScrollBar(Graphics g, int w, int h)
{
  // grey vertical bar
  g.setColor(128,128,128);  // grey
  g.fillRect(w-14, 13, 6, h-26);
  g.setColor(0,0,0);  // black

  // upwards head of the bar
  if (!upArrowShowing) {
    g.setColor(128,128,128);  // grey
    g.fillArc(w-17, 7, 12, 12, 0, 360);   // filled circle
    g.setColor(0,0,0);  // black
  }
  else
    g.fillTriangle(w-11,6, w-6,18, w-16,18);

  // downwards head of the bar
  if (!downArrowShowing) {
    g.setColor(128,128,128);  // grey
    g.fillArc(w-17, h-19, 12, 12, 0, 360);   // filled circle
    g.setColor(0,0,0);  // black
  }
  else
    g.fillTriangle(w-11,h-6, w-6,h-18, w-16,h-18);
} // end of drawScrollBar()
```

There's a lot of 'magic' numbers in drawScrollBar(), but all the coordinates are specified relative to the width and height of the canvas (w and h).

The upwards arrow:

```
g.fillTriangle(w-11,6, w-6,18, w-16,18);
```

and the vertical bar:

```
g.fillRect(w-14, 13, 6, h-26);
```

are shown in Figure 9.

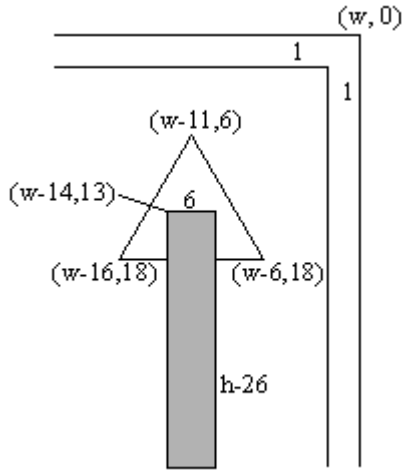

Figure 9. The Coordinates for the Upwards Arrow and Vertical Bar.

3.10. Fixing the Message Box's Size

Every CustomItem must define get methods for the item's minimum and preferred width and height. In ScrollableMessagesBox, they are:

```
protected int getMinContentHeight()
{  return height; }

protected int getMinContentWidth()
{  return width;  }

protected int getPrefContentHeight(int w)
{  return height;  }

protected int getPrefContentWidth(int h)
{  return width;  }
```

The minimum size is the smallest size that the item can handle, while the preferred size specifies the optimal dimensions.

The arguments to getPrefContentHeight() and getPrefContentWidth() are the system's width and height values for the item (or perhaps –1).

The system will *usually* employ the values returned by the 'preferred' methods to set the size of the item, but may choose other numbers. The chosen sizes are reported to the item by the system calling CustomItem.sizeChanged() (which is not used in ScrollableMessagesBox). sizeChanged() is also called whenever the item is resized by the system. A size recalculation can be triggered from within CustomItem by calling invalidate().

4. Other CustomItem Features

There are two important CustomItem features which aren't illustrated by ScrollableMessagesBox:

- traversal;
- off-screen editing.

4.1. Traversal

Different platforms offer varying support for traversal inside a CustomItem. For instance, a system may not provide it at all, or only in the vertical direction, or perhaps in both the vertical and horizontal directions.

A platform's capabilities are determined by calling getInteractionModes(), and checking for the presence of the TRAVERSE_HORIZONTAL and TRAVERSE_VERTICAL bits, or the NONE bit. For instance:

```
int interactionMode = getInteractionModes();
boolean supportsHoriz =
  ((interactionMode & CustomItem.TRAVERSE_HORIZONTAL) != 0);
boolean supportsVert =
  ((interactionMode & CustomItem.TRAVERSE_VERTICAL) != 0);
```

When traversal is supported, the system will signal its use by calling CustomItem's traverse() method in two situations:

- when a traversal enters the item for the first time;
- when a traversal moves through the item.

The traverse() method must be overridden if the CustomItem is to support internal traversal. Its prototype is:

```
protected boolean traverse(int direction,
                   int viewportWidth,int viewportHeight,
                   int[] visRect_inout);
```

When a traversal initially enters the CustomItem, traverse() should return true if the item supports internal traversal, false otherwise. The default implementation of traverse() returns false.

For a traversal within the item, traverse() should return true if the traversal will remain inside the item after this call, or false if the traversal will leave.

As the traversal leaves the item, the system calls CustomItem.traverseOut().

When traverse() is called, the traversal direction is passed in as its first argument: the value can be Canvas.DOWN, Canvas.UP, Canvas.LEFT, Canvas.RIGHT, or NONE (if the system can't assign a direction).

The second and third arguments (viewportWidth and viewportHeight) specify the largest area of the item that is likely to be visible at any given time. This 'viewport'

dimension is only relevant if the item is larger than the form, and scrolling is needed to view different parts of the item.

The fourth argument is employed to pass rectangle information into, *and* out of, the method. The input rectangle representing the region of the item that's currently visible. The output rectangle should be the new visible region after the traversal operation has been applied. This argument is only a concern if the CustomItem is larger than the enclosing form.

traverse() Examples

Sun's WTK 2.2 includes a CustomItem example, as part of its UIDemo demo: a 5x3 table supporting vertical and horizontal traversal. The source code can be found in <WTK_HOME>\apps\UIDemo\src\customitem\Table.java.

All the table is visible inside the form, so only traverse()'s direction argument is utilized in the method's body. The parts of the table affected by a traversal are redrawn by traverse() calling repaint(x,y,w,h). The method finishes by setting the visRect_inout[] array, which is actually unnecessary since all the item can be seen without scrolling.

Mikko Kontio's article, *Custom GUI Development with MIDP 2.0*, available at http://www-128.ibm.com/developerworks/wireless/library/wi-developui/, describes a simpler traversal example, based around a two-element CustomItem which emulates MIDP's ChoiceGroup. The traverse() method is similar to the one in UIDemo, since all the item is visible in the form, making the second, third, and fourth arguments of traverse() irrelevant. The main simplification is that the method only provides vertical traversal.

4.2. Off-screen Editing

Off-screen editing is a coding technique used in a CustomItem when the user needs to input or edit complex information. The basic idea is to have the item create a temporary TextBox or Canvas object which acts as an editing screen.

The CustomItem calls Display.setCurrent() to make the editing screen active. When the edit is completed, the screen passes the data back to the CustomItem via a public callback method, then calls Display.setCurrentItem() to return execution to the CustomItem.

The callback method in CustomItem will typically call repaint() to freshen up the display with the new data, or perhaps notifyStateChanged() to tell an ItemStateListener that things have changed.

Sun's WTK 2.2 CustomItem example in UIDemo (mentioned in the previous section) utilizes this off-screen editing approach. When a user wants to add text to one of the table's cells, the "Edit" command creates a TextBox instance to read in the data.

