Build your stock with J2ME

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

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

Section 1. Introduction

J2ME tutorial overview

In this tutorial, we will build a typical Java 2 Platform, Micro Edition (J2ME) application, called *UniStocks,* based on CLDC (Connected Limited Device Configuration) and MIDP (Mobile Information Device Profile) APIs. As we build the application, we'll cover the following topics:

- MIDlet basics
- MIDP high-level user interface design
- MIDP low-level user interface design
- Record management system (RMS)
- J2ME networking and multithreading
- Server-side design
- Application optimization and deployment
- Overcoming J2ME limitations

About the author

Jackwind Li Guojie has been writing software professionally for many years. As leader of the Jackwind Group, he provides software consulting and training servies in the Asia-Pacific area. Currently, he is also pursuing research on soft computing at Nanyang Technological Unversity, Singapore. You can contact Jackwind at *jackliguojie@hotmail.com.*

About UniStocks

UniStocks is a stock application that enables the user to access and manage information of any stock -- anywhere, anytime.

Like any stock application on your PC or on the Web, UniStocks lets the user:

- Add stocks (store stock symbols on a phone)
- Delete stock(s) (remove stock symbols)
- View live information of selected stocks, such as current high price, low price, volume, etc.
- View charts of historical data (one month, three months, six months, one year), price, volume, and so forth.

UniStocks is based on a client-server architecture. The server will provide all required stock information, such as live data and historical data.

Figures 1 through 3 show the main menu; the downloading status, and the stock historical

chart, respectively.

Figure 1: UniStocks main menu

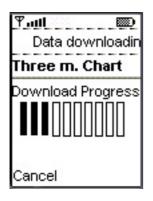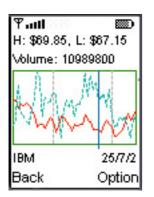Figure 2: UniStocks download status

Figure 3: UniStocks historical chart

Section 2. Getting started

Choose your development tools

Few IDE tools are available for J2ME. You should already be familiar with the J2ME Wireless Toolkit (WTK). WTK lets you compile, package, deploy, and execute J2ME applications. WTK is not a real IDE, because it lacks important features like editing and debugging. However, it is easy to use, which is appealing to many developers.

Other tools, such as IBM VisualAge Micro Edition and Borland JBuilder Mobile Set 2.0, are extensions of mature IDEs. They provide wizards and other tools to help you create J2ME applications.

You should choose the right tools according to your needs. (See Resources on page 39 for IDE links.) For this project, we'll use the text editor Emacs with WTK 1.04.

Code the MIDlet

The J2ME platform consists of a set of layers, on top of which lies MIDP. We develop J2ME applications on top of MIDP; thus, the applications are called *MIDlets*. Every J2ME application must extend the `MIDlet` class so the application management software can control it.

Here is a blueprint of our MIDlet:

```
public class UniStock extends MIDlet implements CommandListener
{
  Display    display;

  private    List    menu;
  private    Command commandSelect;
  private    Command commandExit;
  ...

  public UniStock() {   // The constructor.
     ...
     // Data initialization.

     // Read saved data from RMS.

     // Create UI components and the first screen (menu).
  }

  public void startApp() {  // Enter the active state.
     // Display the first screen.
     display.setCurrent(menu);
  }

  public void pauseApp() {
  }

  public void destroyApp(boolean unconditional) {
     ...
     // Clean up data streams, network, etc.
     ...
```

```
  }

  public void commandAction(Command c, Displayable s) {
     notifyDestroyed();
  }

  // Other customized methods.
  ...
}
```

When the application management software invokes the `startApp()`, `pauseApp()`, or `destroyApp()` method, the MIDlet's state changes. For example, when `pauseApp()` is invoked, the MIDlet changes from an active to a paused state.

Because those methods signal state changes, they need to be lightweight in order to return quickly. As you can see in the above code listing, we put most of the initialization process in `<init>` and the constructor, rather than in `startApp()`.

Warning: During application execution, `startApp()` and/or `pauseApp()` might be called several times as the state changes. You should be careful never to place any initialization code inside those two methods.
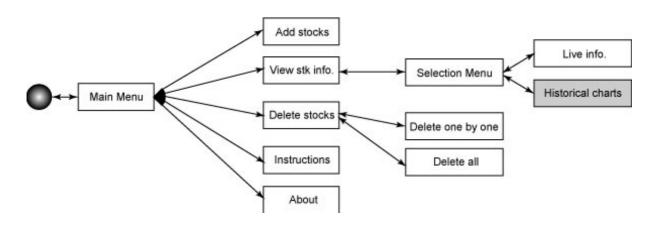
Section 3. High-level user interface design

General design overview

GUI APIs are defined in MIDP rather than CLDC. UniStocks will use both high-level user interface APIs (such as `Alert`, `Form`, and exclusive components like `Command`), as well as low-level ones (such as `Canvas`).

`Screen` is the super class of all high-level interface APIs. Figure 4 shows a screen map of UniStocks. Note that "Historical charts," with the gray background, uses the low-level API to draw charts to the screen. The screen map does not show the intermediate-level screens, such as alerts and error reporting screens.

Figure 4: UniStocks screen map

Avoid splash screens

What about a cool splash screen? I strongly recommend you don't display a splash screen. Small devices have limited processor power. Displaying a splash screen will significantly delay your application launch process. In addition, it will increase your final distribution file size. (The limit of jar file size for some phones is as low as 30K!)

If you really want to display a splash screen, display it only when the user first launches the MIDlet. Users will become frustrated if they must wait for your splash screen to display every time.

In this application, we use a simple "About" alert to show the application's nature and license information.

Screen navigation: The tree model

While developing my company's first few J2ME applications, my development team and I found that it was difficult to navigate among screens. MIDP only provides the `Display` class for one-screen display management. After some brainstorming, we created the tree model shown in Figure 5, which is easily understood and adopted in J2ME development.
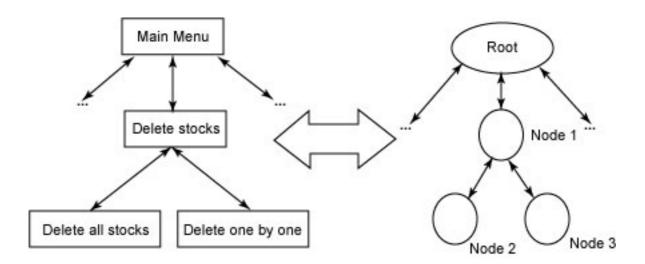
Figure 5: This image shows that a screen map is a typical tree.

As Figure 5 illustrates, our UniStocks screen map is actually a bidirectional tree. Each screen in the map is a node, and the main menu is the root node.

In a tree structure like this, we can use the navigation techniques Depth-First-Search and Breadth-First-Search. Further, the implementation will be easy.

Tree model implementation

A typical node implementation is as follows:

```
class Node {
  Node    parent;
  Vector  children;
  boolean isRoot;

  ...
}
```

Similarly, we implemented the screen as a tree node:

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

/**
 * SubForm: A template of all subforms, 'node' in a tree.
 *
 * @version          $2.1 2002-JAN-15$
 * @author           JackWind Li Guojie (http://www.JackWind.net)
 */

public class SubForm extends Form implements CommandListener
{
  Command      backCommand;       // Back to the parent screen.
  UniStock     midlet;                    // The MIDlet.
```

```
  Displayable  parent;             // The parent screen.

  /**
   * Constructor - pass in the midlet object.
   */
  public SubForm(String title, UniStock midlet, Displayable parent) {
    super(title);
    this.midlet = midlet;
    this.parent = parent;

    backCommand = new Command("Back", Command.BACK, 1);

    addCommand(backCommand);
    setCommandListener(this);
  }

  /**
   * commandListener: Override this one in subclass!
   * Call this method in subclass: super.commandAction(c, s)
   */
  public void commandAction(Command c, Displayable s) {
    if(c == backCommand) {
      if(parent != null)
        midlet.display.setCurrent(parent);
    }
  }

}
```

We don't keep a children list in the node because we usually create new child screens on the fly. (Of course, you can keep a children list if you don't want to create child screens every time.) When the user presses the Back command, the system simply displays its parent screen. The child might make some changes on the parent screen, and then display its parent screen after the Back button is pressed.

Using this tree model, we can easily create user-friendly J2ME applications. As an alternative, you can look into another navigation model, called a *stack-based framework,* described by John Muchow in *Core J2ME Technology and MIDP*. (See Resources on page 39 .)

A sample screen

The following code list is a simplified version of our "View Stock Details" form implementation. The class `FormView` extends the tree node implementation `SubForm`. `FormView` adds its own customized commands, methods, and so on. It also overrides the `commandAction()` method for its customized command event-handling routine:

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

/**
 * Form: Display view stock form.
 *
 * @version                   1.0           2002-JUL-07
 * @author         JackWind Li Guojie (http://www.JackWind.net)
 */
```

```
public class FormView extends SubForm
{
  ChoiceGroup choiceStocks;
  Command     commandLive;
  Command     commandOneMonth;
  Command     commandThreeMonth;
  Command     commandSixMonth;
  Command     commandOneYear;

  int mode;    // 1 - Live info.
               // 2 - One month.
               // 3 - Three months.
               // 4 - Six months.
               // 5 - One year.

  Stock         s;   // Selected stock.


  StockLive     sl;
  StockHistory  sh;

  public FormView(String title, UniStock midlet, Displayable parent) {
    super(title, midlet, parent);

    commandLive = new Command("Live Info", Command.SCREEN, 1);
    commandOneMonth = new Command("One m. chart", Command.SCREEN, 1);
    commandThreeMonth = new Command("Three m. Chart", Command.SCREEN,
1);
    commandSixMonth = new Command("Six m. Chart", Command.SCREEN, 1);
    commandOneYear = new Command("One yr. Chart", Command.SCREEN, 1);

    addCommand(commandLive);
    addCommand(commandOneMonth);
    addCommand(commandThreeMonth);
    addCommand(commandSixMonth);
    addCommand(commandOneYear);

    choiceStocks = new ChoiceGroup("Select a stock: ",
Choice.EXCLUSIVE);
       for(int i=0; i<midlet.stocks.size(); i++) {
         if(UniStock.DEBUG)
           UniStock.debug("Loading #" + i);
         Stock s = (Stock)midlet.stocks.elementAt(i);
         Exchange e = (Exchange)midlet.exchanges.elementAt((int)s.ex);
         choiceStocks.append( s.code + " [" + e.code + "]", null);
       }

       append(choiceStocks);

  }

  public void commandAction(Command c, Displayable ds) {
    super.commandAction(c, ds);

    if(c == commandLive || c == commandOneMonth || c==
commandThreeMonth
                        || c == commandSixMonth || c == commandOneYear)
{
       if(c == commandLive)
         mode = 1;
       else if(c == commandOneMonth)
         mode = 2;
       else if(c == commandThreeMonth)
```

```
      mode = 3;
    else if(c == commandSixMonth)
      mode = 4;
    else if(c == commandOneYear)
      mode = 5;

    if(choiceStocks == null || choiceStocks.getSelectedIndex() == -1)
{
        midlet.reportError("Nothing selected to view!");
        s = null;
        return;
    }else{
      s =

(Stock)(midlet.stocks.elementAt(choiceStocks.getSelectedIndex()));
    }

    Download dl = new Download(this, midlet);

    NextForm nextForm = new NextForm(c.getLabel(), midlet, this, dl);
    midlet.display.setCurrent(nextForm);

    Thread t = new Thread(dl);
    dl.registerListener(nextForm);
    t.start();
  }

  }


}
```

Concepts behind MVC and user interface delegation

The classic Model-View-Controller (MVC) design pattern was introduced in the SmallTalk programming environment. The *model* maintains the data state in the application domain. The *view* represents the model in graphical or nongraphical form. The *controller* receives the external input and interprets it in order to update the model or view. However, sometimes separating the view from the controller is difficult. Instead, developers combine them, calling it a *representation*. This modified version of MVC is often called *user interface delegation*.

Why use MVC or user interface delegation? With MVC and UI delegation, you can adapt your application painlessly. In a J2ME environment, MVC lets you do modular component testing. You can fully test the business logic code before mixing it with the GUI part.

Using MVC/UI delegation

The following two code listings both try to add a new `Stock` object; however, they use different approaches. The first one uses good MVC design. It clearly separates the presentation from the model. Alternatively, the second one stuffs everything inside the method.

If we want to add another attribute to the `Stock` class -- for example, company name -- the `Stock` constructor needs one more parameter. We also need to check whether the user tries

to add certain restricted stocks. For the MVC code, we simply modify the `addStock()` method in `UniStock`. For the second listing, we must modify every code snippet that contains the code for creating and/or adding stocks, which can be tedious:

```
// Called by certain events (controller), such as user input, etc.
public void process() {
  if(stockOK) {
    // Add stock here ....
    if(nextForm != null) {
      if( midlet.addStock(textCode.getString(),
                      (byte)choiceExchanges.getSelectedIndex(), temp) )
{
        nextForm.setStatus(true, "Stock added.");
      }else{
        nextForm.setStatus(false, "Stock found, but could not be
added.");
      }
    }

    ...
  }
}
```

```
// Called by certain events (controller), like user input, etc.
public void process() {
  if(stockOK) {
    boolean added = false;
    // Add stock here ...
    if(nextForm != null) {

      // Create a new stock.
      Stock s = new         Stock(textCode.getString(),
                        (byte)choiceExchanges.getSelectedIndex());

      // Check for duplication.
      if(midlet.stocks.indexOf(s) != -1) {
        debug("Stock already in records!");
      }else{
        midlet.stocks.add(s);
        added = true;
      }

      if(added) {
        nextForm.setStatus(true, "Stock added.");
      }else{
        nextForm.setStatus(false, "Stock found, but could not be
added.");
      }
    }

    ...

  }
}
```

What's wrong with Alert?

Even for moderately sized applications, you need to use `Alert` to notify the user of any action. However, occasionally you may encounter some problems; for example, the user might be confused by the improper use of Alert:

```
// After the 'Delete' command is pressed ...
Alert alert = new Alert("Information",
                "Are you sure you want to delete all the data? ", null, null);
alert.setTimeout(Alert.FOREVER);
display.setCurrent(alert);

delete_all_data();
System.out.println("Data deleted");
...
```

Some books use code similar to the above code to show their readers how to use `Alert`. However, that code is *wrong*. When you press the Delete command, the above code will run. During the execution, you might find that "Data deleted" is printed immediately after the `Alert` displays (as shown here in Figure 6). If you press the Delete command unintentionally, you cannot cancel or roll back deletion because the data has already been deleted before you noticed it.

Figure 6: Deletion alert

`Alert` is misused in the above code. According to the MIDP Java API documentation, "The intended use of `Alert` is to inform the user about errors and other exceptional conditions." Therefore, we need a dialog here. In the next panel, I will present a flexible, reusable `Dialog` class.

Solution: The Dialog class

Please note that `Dialog` is not a standard class in CLDC or MIDP. The developers at JackWind Group created the `Dialog` class shown in the following listing:

```
/**
 * FileName:    Dialog.java
 * Version:     1.0
 * Create on:   2001-JUN-01
 *
 * All rights reserved by JackWind Group. (www.JackWind.net)
```

```
 */

import javax.microedition.lcdui.*;

/**
 * Dialog: A class simulating dialog for UI.
 *
 * @version                    1.0 2001-JUN-01
 * @author                         JackWind Group (http://www.JackWind.net)
 */

class DialogListener {
  public void onOK() {}
  public void onYES() {}
  public void onNO() {}
  public void onCANCEL() {}
  public void onCONFIRM() {}
  // RETRY.
  // ABORT.
}


public class Dialog extends Form implements CommandListener{
  Display          display;
  DialogListener   dll;

  Displayable        parent;

  public static final int OK          = 1;

  public static final int YES         = 2;
  public static final int NO          = 4;
  public static final int CANCEL      = 8;
  public static final int CONFIRM     = 16;

  Command cmOK;
  Command cmYES;
  Command cmNO;
  Command cmCANCEL;

  Command cmCONFIRM;

  StringItem   text;

 /**
  * The constructor.
  *@param title     Form title, must not be null
  * @param text      Form text, must not be null
  * @param mode      Control commands. int > 0
  * @param MIDlet    Our midlet, must not be null
  * @param DialogListener dll, can be null
  * @param parent    The parent form, must not be null
  */
  public Dialog(String title, String text, int mode, MIDlet midlet,
                                                      DialogListener dll
    super(title);
    this.dll = dll;
    this.text = new StringItem(null, text);
    this.display = Display.getDisplay(midlet);
    this.parent = parent;

    if( (mode & OK) != 0) {
      cmOK = new Command("OK", Command.SCREEN, 1);
      addCommand(cmOK);
```

```
      }

      if( (mode & YES) != 0) {
        cmYES = new Command("Yes", Command.SCREEN, 1);
        addCommand(cmYES);
      }

      if( (mode & NO) != 0) {
        cmNO = new Command("No", Command.SCREEN, 1);
        addCommand(cmNO);
      }

      if( (mode & CANCEL) != 0) {
        cmCANCEL = new Command("Cancel", Command.SCREEN, 1);
        addCommand(cmCANCEL);
      }

      if( (mode & CONFIRM) != 0) {
        cmCONFIRM = new Command("Confirm", Command.SCREEN, 1);
        addCommand(cmCONFIRM);
      }

      append(text);
      setCommandListener(this);
    }

    public void commandAction(Command c, Displayable s) {
      if(dll != null) {
        if(c == cmOK)
          dll.onOK();
        else if(c == cmYES)
          dll.onYES();
        else if(c == cmNO)
          dll.onNO();
        else if(c == cmCANCEL)
          dll.onCANCEL();
        else if(c == cmCONFIRM)
          dll.onCONFIRM();
      }

      midlet.display.setCurrent(parent);

    }

}
```

Using Dialog

Using our `Dialog` class, we can rewrite the code from the "What's Wrong with Alert?" section:

```
Dialog dl = new Dialog ( "Confirmation,"
        "Are you sure you want to delete all data?",
        Dialog.CONFIRM | Dialog.CANCEL,
        midlet,

        new DialogListener() {             // Anonymous inner class.
```

```
        public void onCONFIRM() {
           delete_all_data();
           System.out.println("Data deleted");
        }
        // If cancelled, do nothing.
     },
     this
);



display.setCurrent(dl);
```

Now, when you press the Delete command, you will see the screen shown here in Figure 7. You can confirm the deletion or simply cancel this action. Similarly, you can use this `Dialog` to let the user answer simple questions with YES, NO, OK, and so on.

To create a new `Dialog`, we need a `DialogListener` to act in response to user input. In our implementation, `DialogListener` is a class, not an interface. In this way, you simply override any method necessary without implementing all the methods. In the above code, we use an anonymous inner class as `DialogListener`, and override the `onCONFIRM()` method.

Figure 7: Confirmation dialog

Section 4. Low-level user interface design

What's next

In this section, we will build a `Canvas` to display stock price and volume charts.

After the user selects a stock symbol and historical period, the application will retrieve historical data from the server. (I will discuss networking later.) Getting the necessary data, the application will draw the actual charts onto the canvas (see on page below).

The user can view price and volume for every historical trading day by pressing the right arrow to find the next day or the left arrow for the previous day. (See information about event processing in subsequent panels) By pressing the up and down arrow, the user can zoom in and zoom out (see Figure 9).
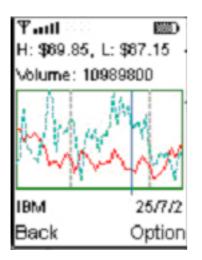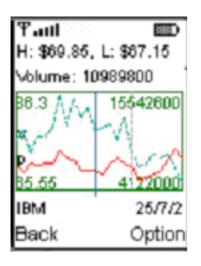
Figure 8: Stock chart canvas

Figure 9: Zoom in and out

Low-level drawing

When creating a `Canvas`, we need to extend the `Canvas` class and override at least its `paint(Graphics g)` method. By overriding the `paint()` method, we can draw the stock chart:

```
protected void paint(Graphics g) {
  // Clear background.
  g.setColor(255, 255, 255);
  g.fillRect(0, 0, screenH, screenV);

  // Draw Strings - Price, volume, etc.
  g.setColor(0, 0, 0);
  g.setFont(font);
  g.drawString("H: $"+UniStock.getString(sr.priceHigh) +
               ", L: $" + UniStock.getString(sr.priceLow),
               1, 1, Graphics.TOP | Graphics.LEFT);
  g.drawString("Volume: " + sr.volumn,
               1, fontH + 2, Graphics.TOP | Graphics.LEFT);

  // Draw the chart.

  for(int i=left+1; i<=right;  i++) {
  // For each visible day (except the first day).

    StockRecord current = (StockRecord)sh.vec.elementAt(i);

    // Draw price chart.
    // Multiplication first, then division to increase accuracy.
    g.setColor(255, 51, 0);            // Set color
    g.setStrokeStyle(Graphics.SOLID);
    g.drawLine(
        startX + (i-1-left)*step,
        startY + Y - (last.priceHigh-priceLowest)*Y/priceBase,
        startX + (i-left)*step,
        startY + Y - (current.priceHigh - priceLowest)*Y/priceBase
    );

    // Draw volume chart.

    last = current;


  } // End for loop.

}
```

The above code is our `paint()` method. Inside the method, we get a reference to the `Graphics` object; thus, we can use it to do the actual drawing. These main drawing methods are available in `Graphics`:

- `drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`
- `drawChar(char character, int x, int y, int anchor)`
- `drawImage(Image img, int x, int y, int anchor)`
- `drawLine(int x1, int y1, int x2, int y2)`

- drawRect(int x, int y, int width, int height)
- drawString(String str, int x, int y, int anchor)
- fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
- fillRect(int x, int y, int width, int height)
- fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

In UniStock, we use drawString() to draw price and volume for the selected day. Then we draw the price and volume chart by concatenating small segments created by drawLine() for each period.

Drawing and filling basics

Small devices have a limited screen or display area. Therefore, you should plan carefully before you start coding. Otherwise, a slight drawing inaccuracy will ruin your user interface. One common problem here is that people are often unclear about how filling and drawing methods work.

The origin of the graphics coordinate system is at the upper left corner, with coordinates increasing down and to the right, as Figures 10 and 11 illustrate. The arguments required by drawing and filling methods define a coordinate path (shown in the gray spots in the figures) instead of pixel positions. This can sometimes be confusing.

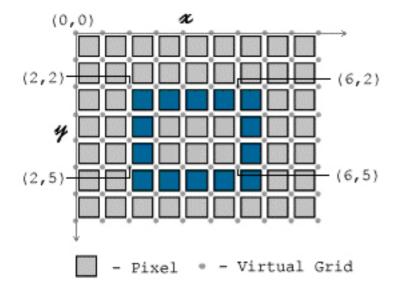Figure 10: Drawing a rectangle -- drawRect (2, 2, 4, 3)

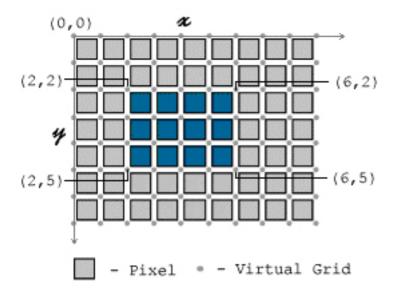Figure 11: Filling a rectangle -- fillRect(2, 2, 4, 3)

Figure 10 is the screen view after invoking the method `drawRect(2, 2, 4, 3)` (which draws a rectangle starting with (2,2), width = 4, height = 3). Notice in Figure 10 that `drawRect()` draws an extra row of pixels on the rectangle's right and bottom sides.

Figure 11's filled rectangle is the result of `fillRect(2, 2, 4, 3)`. Unlike `drawRect()`, `fillRect()` fills the specified path's interior.

Here is the code for drawing a border rectangle for a canvas:

```
g.drawRect(0, 0, getWidth()-1, getHeight()-1);
```

Here is the code for filling a canvas' full screen:

```
g.fillRect(0, 0, getWidth(), getHeight());
```

Low-level event processing

Both `Screen` and `Canvas` are direct subclasses of `Displayable`. We can create commands and add them to `Screen` or `Canvas`, and then set a `CommonListener` for them. This is how high-level event processing works.

But `Canvas` can also handle low-level events. For low-level events -- such as game action, key events, and pointer events -- we don't need to create and register listeners, because `Canvas` has direct methods to handle them:

- `showNotify()`
- `hideNotify()`
- `keyPressed()`
- `keyRepeated()`

- keyReleased()
- pointerPressed()
- pointerDragged()
- pointerReleased()
- paint()

The following is our event-handling routine in `StockCanvas`:

```
protected void keyPressed(int keyCode) {
  switch(getGameAction(keyCode)) {
    case RIGHT:
      cursor ++;
      ...
      repaint();
      break;

    case LEFT:
      cursor --;
      ...

      repaint();
      break;

    case UP:
      zoom(true);
      repaint();
      break;

    case DOWN:
      zoom(false);
      repaint();
      break;

  }// End of switch.
}
```

Once the user presses a key, the `keyPressed()` method will be called with the pressed key code as the only parameter.

Why do we need `getGameAction()` to process the `keyCode`? This way, we can ensure our application's portability. `getGameAction()` will translate a key code into a game action. Those game actions should be available on all J2ME-supported devices. However, a hand phone might have different code settings with a two-way pager. So we need to translate those settings with `getGameAction()`.

Tip: Use game actions, such as UP, RIGHT, and LEFT, to ensure application portability.

Double buffering

Occasionally, you find that canvases flicker during repainting. This flickering is due to the fact that the `Canvas` class must clear the previous screen (background) before it invokes the

paint() method. Erasing Canvas's background results in flickering, which we can eliminate using a well-known technique called *double buffering*.

Double buffering prepares the next Canvas content to display in an offScreen() buffer, and then copies the complete display content to the screen. This way, we avoid erasing and flickering.

The following is our rewritten code for paint():

```
Image offScreen;              // For double buffering.

protected void paint(Graphics g) {
  Graphics ig = null;

  if(isDoubleBuffered()) {  // If the implementation supports it..
    ig = g;
  }else{
    if(offScreen == null)
      offScreen = Image.createImage(screenH, screenV);
    ig = offScreen.getGraphics();
  }

  ig.setColor(255, 255, 255);          // Clear with white background.
  ig.fillRect(0, 0, screenH, screenV);

  ... // Drawing, filling with ig.

  if(isDoubleBuffered())
    g.drawImage(offScreen, 0, 0, Graphics.TOP|Graphics.LEFT);
}
```

If the implementation supports double buffering, we don't need to repeat it. Thus, we must check it before double buffering in the paint() method.

Section 5. Record management system

Store data with RMS

MIDP provides us with the Record Management System (RMS), a records-based persistent storage system. With RMS, you can persistently store data and retrieve it later. In UniStocks, we use RMS to store stock symbols.

`RecordStore`, which consists of a record collection, is the only class in the `javax.microedition.rms` package. A record is a byte array (`byte []`) of data. RMS doesn't support data types for records, so you have to manipulate them yourself.

Here are some RMS facts:
- The naming space for `RecordStore` is controlled at MIDlet-suite granularity
- MIDlets within the same MIDlet suite can share `RecordStore`s
- MIDlets from different MIDlet suites cannot share `RecordStore`s
- When a MIDlet suite is removed, all `RecordStore`s associated with it are removed too.

Warning: RMS does not provide any locking operations. RMS ensures that all individual `RecordStore` operations are atomic, synchronous, and serialized. However, if a MIDlet uses multiple threads to access a `RecordStore`, the MIDlet must coordinate this access, otherwise unintended consequences may result.
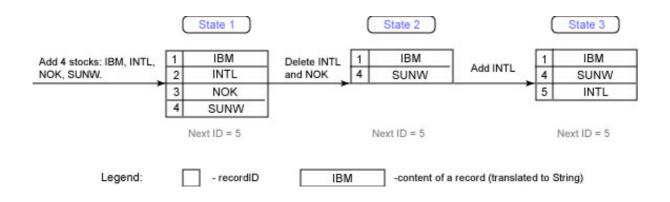

Understanding RecordStore

Records inside a `RecordStore` are uniquely identified by their `recordID`, which is an integer value. The first record that `RecordStore` creates will have a `recordID` equal to 1. Each subsequent record added to `RecordStore` will have a `recordID` one greater than the last added record.

Developers commonly mistake a `RecordStore` to be a `Vector` whose index starts from 1 instead of 0. That is *wrong*, which you may not realize unless you've worked on RMS extensively.

For example, Figure 12 shows the internal state transition of our `RecordStore`. State 2 does not contain any record with a `recordID` equal to 2 or 3. However, its 'Next ID' does not change. As you can see clearly from the state representations below, `RecordStore` is not a `Vector`. In the following panels, you will learn how to correctly add and retrieve records.

Figure 12: The RecordStore's internal state transition

Controlling RecordStores

Open and create a RecordStore:

The code listing below tries to open a `RecordStore`. If the `RecordStore` identified by its name does not exist, RMS will try to create it. The `RecordStore` name should not exceed 32 characters. You should also try to avoid using duplicated names while creating `RecordStore`s. The `openRecordStore()` method throws several exceptions, so we need to manage exception handling:

```
/**
 * Open a record store.
 */
private RecordStore openRecStore(String name) {
  try {
    // Open the record store, create it if it does not exist.
    return RecordStore.openRecordStore(name, true);
  }catch(Exception e) {
    reportError("Fail to open RS: " + name);
    return null;

  }
}
```

Close a RecordStore:

If you don't need to use a `RecordStore` anymore, you can simply close it to release the resources it holds:

```
// Clean up.
try {
  ...
  rsStockList.closeRecordStore();
}catch(Exception e) {
  // reportError("Clean-up error:" + e.toString());
}
```

Erase a RecordStore:

Warning: When you delete a `RecordStore`, you erase its associated records!

```
public void eraseRecordStore() {
  try {
    debug("Deleting record store ...");

    RecordStore.deleteRecordStore(REC_STOCKLIST);
  }catch(Exception e) {
    if(DEBUG)
      debug("Could not delete stores: " + e.toString());
  }
}
```

Create records

As I mentioned earlier, a record is a byte array. However, we usually store data of types
`String`, `int`, and so on in records. Here we can use `DataInputStream` and
`ByteArrayInputStream` to pack data into records:

```
private ByteArrayInputStream    byteArrayInput;
private DataInputStream         dataInput;

byte[]            recData = new byte[200];  // buffer

byteArrayInput = new ByteArrayInputStream(recData);
dataInput = new DataInputStream(byteArrayInput);


...

/**
 * Write a new stock record.
 */

private int writeStock(Stock s) {
  try{
    byteArrayOutput.reset();

    dataOutput.writeUTF(s.code);
    dataOutput.writeByte(s.ex);

    byte[] record = byteArrayOutput.toByteArray();

    // Add to record store.
    return rsStockList.addRecord(record, 0, record.length);

  }catch(Exception e) {
    reportError("Failed to add stock to RMS: " + e.toString());
    if(DEBUG)
      e.printStackTrace();

    return -1;
  }

}
```

In the code listing above, method `rsStockList.addRecord(record, 0, record.length)` has been invoked to create a new record in `RecordStore`. The `recordID` is returned. We can use the following code to check whether or not a new record has been created successfully:

```
/**
 * Add a new stock.
 */
boolean addStock(String code, byte ex, String temp) {
  Stock s = new Stock(code, ex);

  int id = writeStock(s);
    if(id > 0) {
      s.rs_id = id;
      stocks.addElement(s);
      return true;
    }else{
      return false;
    }
}
```

Retrieve data from RMS

In previous panels, I have shown that `RecordStore` behaves differently than a `Vector`. You should never assume that a record with a certain `recordID`, such as 0, always exists. You should use `RecordEnumeration` to retrieve records. The `RecordEnumeration` class represents a bidirectional record store `Record` enumerator. `RecordEnumeration` logically maintains a sequence of `recordID`s of a `RecordStore`'s records. After obtaining `recordID`s from `RecordEnumeration`, we can use this method to retrieve the byte array content:

```
public int getRecord(int recordId, byte[] buffer, int offset)
             throws RecordStoreNotOpenException,
                    InvalidRecordIDException,
                    RecordStoreException
```

In the previous panel, we used `DataInputStream` and `ByteArrayInputStream` to pack data into records. Similarly, we can use `DataOutputStream` and `ByteArrayOutputStream` to retrieve data from records. However, we reverse the order:

```
private ByteArrayOutputStream  byteArrayOutput;
private DataOutputStream       dataOutput;

byteArrayOutput           = new ByteArrayOutputStream();
dataOutput                = new DataOutputStream(byteArrayOutput);

/**
 * Load stock from RMS.
 * Should be called exactly once in the application cycle.
```

```
 */
private void readStock() {
  stocks = new Vector();

  try {
    int total = rsStockList.getNumRecords();
    if(total == 0)  // No record..
      return;

    RecordEnumeration re = rsStockList.enumerateRecords(null, null,
false);

    byteArrayInput.reset();
    for(int i=0; i<total; i++) {
      int id = re.nextRecordId();
      if(DEBUG)
        debug("Reading " + (i+1) + " of total " + total + " id = " +
id);

      rsStockList.getRecord( id, recData, 0);
      Stock s = new Stock (
                    dataInput.readUTF(),     // full name - String
                    dataInput.readByte()     // num - byte
                    );

                    s.rs_id = id; // Keep a copy of recordID.

                    stocks.addElement (s);

                    byteArrayInput.reset();
    }

  }catch(Exception e) {
   if(DEBUG)
     e.printStackTrace();
  }
}
```

Tip: Always keep a copy of `recordID`s after loading data from the records. Those `recordID`s will be useful if you need to modify or delete records later.

Delete and modify records

You can easily delete a record if you know its `recordID`:

```
eraseStock( ((Stock)stocks.elementAt(i)).rs_id )) ;

/*
 * Erase a record.
 * @param i recordID
 */
private boolean eraseStock(int i) {
  try {
    rsStockList.deleteRecord(i);
  }catch(Exception e) {
    reportError(e.toString());
    return false;
  }
```

```
  return true;
}
```

Modifying a record is similar to creating a new record. Instead of creating a new record, we overwrite an existing record. Referring to the code listing in the "Create Records" panel, we change `rsStockList.addRecord(record, 0, record.length);` into `rsStockList.setRecord(recordID, record, 0, record.length)`.

Section 6. J2ME networking

Make the connection

CLDC provides a generic connection framework that we can use to send and retrieve data from the Web. In addition, MIDP provides the `HttpConnection` interface, while the MIDP specification requires that implementations support at least the HTTP 1.1 protocol. Thus, we can use the `HttpConnection` to query a Web server and retrieve stock information for UniStocks.

Using HttpConnection

HTTP is a request-response protocol in which the request parameters must be set before the request is sent. In UniStocks, we use the following code to retrieve a stock's live and historical data:

```
// private void getInfo() throws Exception:

HttpConnection http = null;
InputStream is = null;
boolean ret = false;

// Form URL
if(formAdd != null) {
  URL = baseURL + "?E=" + formAdd.choiceExchanges.getSelectedIndex() +
        "&S=" + formAdd.textCode.getString() + "&T=0";
}else if(formView != null && formView.s != null) {
  // Create view baseURL....
  ...
}

if(UniStock.DEBUG)
  UniStock.debug("Connecting to: " + URL);

try{
  http = (HttpConnection) Connector.open(URL);
  http.setRequestMethod(HttpConnection.GET);
  if(hasListener)

    dl.setProgress(1, 10);

  // Query the server and try to retrieve the response
  is = http.openInputStream();
  if(hasListener)
    dl.setProgress(2, 10);

  String str;                               // Temp buffer.
  int length = (int) http.getLength();
  if(hasListener)
    dl.setProgress(3, 10);

  if(length != -1) { // Length available.
    byte data[] = new byte[length];
    is.read(data);
    str = new String(data);
  }else{          // Length not available.
    ByteArrayOutputStream bs = new ByteArrayOutputStream();
```

```
      int ch;
      while((ch = is.read()) != -1)
        bs.write(ch);

      str = new String(bs.toByteArray());


      bs.close();
    }


    if(UniStock.DEBUG)
      UniStock.debug("Got Data:>" + str + "<");

    // String downloaded.....
    // Process string here.
    ...

    if(hasListener)
      dl.setProgress(10, 10);

    // Alert the user.
    // AlertType.INFO.playSound(midlet.display);

  } catch (IOException e) {
    if(midlet.DEBUG)
      midlet.debug("Downloading error: " + e.toString());
    if(formAdd != null) {
      formAdd.stockOK = false;
    }else if(formView != null) {
    }
  } finally {
    if(formAdd != null)
      formAdd.process();
    if(formView != null)
      ;             // Do something.

    /// Clean up.
    if(is != null)
      is.close();

    if(http != null)
      http.close();

    if(dl != null)
      dl.onFinish();
  }
```

The connection exists in one of three states:

1. **Setup:** In this state, the connection has not been made to the server. `http = (HttpConnection) Connector.open(URL)` only creates an `HttpConnection`, which is not yet connected.
2. **Connected:** After all necessary headers have been set, invoking `http.openInputStream()` will result in a server connection. Request parameters are sent and the response is expected. Once the application downloads, the connection may move to the next state.
3. **Closed:** The connection has been closed and the methods will throw an `IOException` if

called. In our case, we close all the data streams and connections before exiting our
customized download method.

Using threads with HttpConnection

Nearly all J2ME introductory books demonstrate some primitive `HttpConnection`
examples. Without providing visual feedback during a network connection, you leave the
user waiting, staring at a frozen screen (and today's slow wireless networks make the
situation worse). Consumer electronic users normally have much higher expectations than
desktop users.

Therefore, in UniStocks, we use multithreading to tackle this problem. After pressing a key or
command for download, the user will see a screen, as illustrated by on page , and survey
the progress through the gauge. If he changes his mind, he can simply press Cancel and
return to the previous screen. After data downloads, the user will see the chart screen,
shown in Figure 14 (repeated here from Figure 8). This way, everything runs or stops
gracefully. I will show a detailed implementation in the next panel.
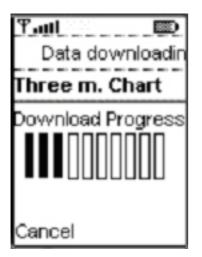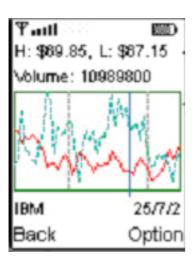
Figure 13: Downloading screen

Figure 14: Stock chart canvas

Tip: Use a gauge rather than an animated picture to display progress.

Implementation multithreading with connection

We have two major goals: Show the user connection progress, and give the user control. When the user presses the Download command, the system will create and start a new thread for download. At the same time, the download screen is created and displayed to the user:

```
if(c == commandOneMonth) {
  mode = 1;
  ...

  Download dl = new Download(this, midlet);

  // Download screen.
  NextForm nextForm = new NextForm(c.getLabel(), midlet, this, dl);
  midlet.display.setCurrent(nextForm);

  Thread t = new Thread(dl);
  dl.registerListener(nextForm);
  t.start();
}
```

If the user presses Cancel while downloading, downloading should stop immediately.

```
if(c == commandCancel && (! finished) ) {
  if(download != null)
     download.setInterrupted();

  midlet.display.setCurrent(parent);
}
```

For our implementation, we'll use an interruption technique. In our `getInfo()` method, we

frequently check whether the downloading thread has been interrupted. If a thread has been interrupted, `InterruptedException` is thrown and the `run()` method catches this interruption and the thread exits.

You can use another technique: Close the connection when Cancel is pressed. Once the connection is closed, `Connection` methods will throw an `IOException` if called. However, any streams created from connection are still valid even if the connection has been closed.

My development team and I have found that the first interruption technique performs better.

The Download class

In the last panel, I briefly discussed the techniques for graceful multithreading. The code list below provides a detailed implementation. Here is the code for the `Download` class:

```
public class Download implements Runnable
{
  private UniStock     midlet;
  private String       baseURL;          // Retrieve URL from JAD file.
  private String       URL;

  private boolean      interrupted;

  // For a Class object, only one of the following
  // objects is not null.
  private  FormAdd  formAdd;
  private  FormView formView;

  // We only allow one download listener so far.
  DownloadListener dl;
  boolean hasListener;

  public Download(UniStock midlet) {
    this.midlet = midlet;
    baseURL = midlet.getAppProperty(midlet.URL_KEY);
  }

  ...

  /**
   * Thread starts from here!
   */
  public void run() {
    try {
      getInfo();
    }catch(Exception e) {
      if(midlet.DEBUG)
        midlet.debug("Exception caught: " + e.toString());
    }
  }

  public void setInterrupted() {
    interrupted = true;
    if(midlet.DEBUG)
      midlet.debug("Interrupted!");
  }

  public boolean interrupted() {
```

```
      return interrupted;
  }

  /**
   * Do the real work here.
   */
  private void getInfo() throws Exception {
    HttpConnection http = null;
    InputStream is = null;
    boolean ret = false;

    if(formAdd != null) {
      URL = baseURL + "?E=" +
formAdd.choiceExchanges.getSelectedIndex() +
                    "&S=" + formAdd.textCode.getString() + "&T=0";
    }else if(formView != null && formView.s != null) {
      // Create view baseURL....
      ...
    }

    if(UniStock.DEBUG)
      UniStock.debug("Connecting to: " + URL);

    try{
      http = (HttpConnection) Connector.open(URL);
      http.setRequestMethod(HttpConnection.GET);
      if(hasListener)
        dl.setProgress(1, 10);

      if(interrupted)
        throw new InterruptedException();

      is = http.openInputStream();
      if(hasListener)
        dl.setProgress(2, 10);
      if(UniStock.DEBUG)
        UniStock.debug("Connecting to: " + URL);

      String str;        // Temp buffer.
      int length = (int) http.getLength();
      if(hasListener)
        dl.setProgress(3, 10);

      if(interrupted)
        throw new InterruptedException();

      if(length != -1) { // Length valid.
        byte data[] = new byte[length];
        is.read(data);
        str = new String(data);
      }else{             // Length not available.
        ByteArrayOutputStream bs = new ByteArrayOutputStream();

        int ch;
        while((ch = is.read()) != -1)
                  bs.write(ch);

        str = new String(bs.toByteArray());
          bs.close();
      }

      if(interrupted)
        throw new InterruptedException();
```

```
      if(UniStock.DEBUG)
        UniStock.debug("Got Data:>" + str + "<");

      // String downloaded.....
      // Process data here.
      ...

      if(hasListener)
        dl.setProgress(10, 10);

      // Alert the user.
      // AlertType.INFO.playSound(midlet.display);

    } catch (IOException e) {
      if(midlet.DEBUG)
        midlet.debug("Downloading error: " + e.toString());
      if(formAdd != null) {
        formAdd.stockOK = false;
      }else if(formView != null) {

      }
    } finally {
     if(formAdd != null)
       formAdd.process();
     if(formView != null)
       ;            // Do something.

     /// Clean up.
     if(is != null)
       is.close();
     if(http != null)
       http.close();

     if(dl != null)

       dl.onFinish();
    }

  } // End function getInfo();

}
```

Section 7. Server-side design

Consider the server side

The UniStocks application is based on a client-server architecture. (The MIDlet is the client side.) In previous sections, we have designed the client part. This section focuses on the server side. Before we create a server-side program, we must carefully plan what we should put on the server side, and what we should leave on the client side.

Some advantages of server-side computing are:
- Great processor power
- High-speed connection
- Flexible computing choices (languages, platforms, etc.)

Take advantage of the server side

Now that we know some advantages of server-side programs, how can we utilize them to create our own server-side programs? The goal is to improve the whole application's performance, on the server side and the client side. To help us overcome the limitations of J2ME and small devices, server-side programs can:

- Move the heavy computing part to the server side
- Do floating point computation on the server side
- Preprocess information on the server side to improve performance
- Overcome other J2ME limitations, such as creating pictures
- Reduce upgrade cost: One upgrade on the server side versus an upgrade of every software suite on the client side

Collecting information

The server side provides us with stock information. The best way to get that stock data is from a fast database; we can then send the data to the client. However, we usually don't have such a convenient database to access. Our solution here is to parse the Yahoo! Finance Web pages, which provide stock information from major exchanges, and then forward the formatted data to the client side. You can easily add your regional exchanges in the same way.

You may directly process the Web pages on the client side. However, once the Web pages change their format, you have to update all installed MIDlets. If you later have a faster way to access the stock data, you could simply update the server side only.

Tip: You are not limited to use Java language in server-side programming. Since the client side does not care about the server-side implementation, you can use your favorite script languages, such as Perl, ColdFusion, and so forth.

Processing and delivery

If you supply a valid stock symbol, the Yahoo! Finance server will return the following live data:

```
Symbol Current Price, Date, Time,     Change Open  High  Low    Volume
"IBM", 76.50,  "8/15/2002", "4:00pm", +1.58, 75.40,76.71,74.60, 9269600
```

If the MIDlet requests a certain stock's live information, it must supply the stock's symbol to query the server. The server then queries the Yahoo! Finance server to get information. The server should return the following information to the MIDlet:

- Status: Success or failure (1 for success, 0 for failure)
- Any data if applicable

Why do you need a status byte? When you surf the Web, you may experience the following problems:

- HTTP gateway timeout
- Web server too busy
- HTTP internal error
- Script running timeout

Even if the Web server is O.K. and your scripts are correct, the user might still encounter the above problems due to multiple gateways between the client and the Web server. Using a status byte, we can check whether or not the data received by the client is valid. If the data is not valid, we simply discard it. By doing this, we also reduce parsing errors. Here is the sample output (live data):

```
[Status * last price * price change * open price * high * low * volume]
 1      * 73200        *1020            *73750          *73990 *73070*4586500*
```

Similarly, the server could retrieve the historical data from Yahoo! Finance and send it to the client. Here is the sample output (historical data):

```
[status * year-month-day  price high  price low  volume * ]
 1      * 2-9-6            73990       73070       4586500* ...
```

Section 8. Overcome J2ME limitations

Floating point computation

J2ME does not support floating point computation. In UniStocks, we need floating point support for stock prices and so forth. To solve this problem, we use the following technique: When making floating point computations, we multiply the floating point numbers by 1000, trim them to integers, and then we calculate. After the computation, we might get a result like 12200, which stands for stock price 12.2. To display it correctly, we process a string using the `getString()` method in the `UniStock` class:

```
/**
 * getString: Form a String from a big integer.
 * We assume the integer has been multiplied by 1000.
 */
public static final String getString(int i, boolean trimZero, boolean
isPositive) {
  if(isPositive && i < 0) {
    return " N/A";
  }

  if(i==0) return "0";
  boolean neg = (i > 0 ?  false : true);
  String str= Integer.toString(i);
  if(neg) {
    i = -i;             // Make it positive first.
    str = str.substring(1);
  }

  if(i < 10) {          // Like 9.
    str = "0.00" + str;
  }else if(i < 100) {   // Like 98.
    str = "0.0" + str;
  }else if(i < 1000) { // Like 450.
    str = "0." + str;
  }else{                // Like 10900.
    str = Integer.toString(i);
    str = str.substring(0, str.length()-3) + "." +
str.substring(str.length()-3);
  }

  if(neg)
    str = "-" + str;

  if(trimZero) {        // Trim extra zeros
    int dotP = str.indexOf('.');
    int pos = -1;
    if(dotP < 0)        // if no '.' is found.
      return str;

    for(pos = str.length()-1; pos >= dotP; pos --)
      if(str.charAt(pos) != '0')
        break;

    if(pos == dotP)
    // If nothing is left behind '.', remove '.' too.
      pos -= 1;

    return str.substring(0, pos+1);
  }
```

```
    return str;

}
```

Hardcoding technique

My development team and I developed a shooting game recently. During development, my colleagues complained that J2ME has no sin, cos, or tg methods. Without knowing those sin and cos values, shooting would be uncontrollable. Finally, we identified that we only had 15 different angles, so we hardcoded their sin and cos values into the program. You could use this hardcoding technique to solve similar problems.

The MIDlet suite application descriptor file

The application management software needs the JAD file to obtain information needed to manage resources during MIDlet execution.

If you use the WTK, you can see your JAD file's attributes by pressing the Settings button. The first tab displays the required attributes, some of which you need to modify:

- **MIDlet-Name:** The name of MIDlet suite
- **MIDlet-Vendor:** The default value is Sun Microsystems; change it to your company name
- **MIDlet-Version:** The default value is 1.0; change it to the application's version

In UniStocks, we create a user-defined attribute: INFO_PROVIDER_URL (key). This attribute specifies the stock information provider's URL (value): http://www.jackwind.net/resources/pj2me/StockInfoProvider.php. In our program, we use the following code to retrieve this value:

```
static final String URL_KEY = "INFO_PROVIDER_URL";

public Download(UniStock midlet) {
  this.midlet = midlet;
  baseURL = midlet.getAppProperty(midlet.URL_KEY);
}
```

If you installed the StockInfoProvider servlet on your Web server, you may want to set this URL pointing to the servlet's exact address.

Section 9. Wrap up

What we covered

In this tutorial, we have covered almost every aspect of J2ME by developing a typical application -- UniStocks.

You received great hands-on experience in MIDLet basics, high-level user interface design, low-level user interface design, persistent storage mechanisms, J2ME networking, and server-side design, among other areas. In addition, I presented various frameworks and techniques that you can use to develop your own J2ME applications with ease and confidence.

Source/binary code configuration

1. Download `pj2me.zip` from *http://www.jackwind.net/resources/pj2me*.
2. Unzip `pj2me.zip` to a directory. Read carefully the license in `license.txt`. The `UniStocks` folder contains the MIDlet client source code and resources; the `jackwind` folder contains server-side code (the servlet).
3. Run UniStocks. Copy the `UniStocks` folder to WTK's `apps` directory, and UniStocks should appear in your project list of WTK ('Open Project ...'). Open UniStocks and run it. The default stock provider is http://www.jackwind.net/resources/pj2me/StockInfoProvider.php, so you should configure the proxy setting before connecting to our server.
4. Set up your own stock provider (optional). First configure your proxy host and proxy port in file `StockInfoProvider.java` (inside the `jackwind` folder). Compile source file and install the servlet. For details, please refer to `README.txt` under `jackwind/WEB-INF`. Set the UniStocks JAD file's user-defined attribute `INFO_PROVIDER_URL` to your servlet URL; for example:
   `http://127.0.0.1:8080/jackwind/servlet/StockInfoProvider`

Resources

- Download the UniStocks binary and source from *http://www.jackwind.net*.
- *Download* the Java 2 Platform, Micro Edition, Wireless Toolkit.
- Create your J2ME application with *IBM VisualAge Micro Edition*.
- Check out *Borland JBuilder Mobile Set 2.0* as an IDE alternative.
- *Sun One Studio 4.0* EA Mobile Edition is a free, feature-rich IDE.
- Visit Sun's wireless developer *homepage*.
- *View* JSR 30: J2ME Connected, Limited Device Configuration.
- *View* JSR 37: Mobile Information Device Profile for the J2ME Platform.
- To know more about multithreading with Java, read *Java Thread Programming,* David M. Geary (Sams, 1999).
- To know more about patterns, read *Design Patterns: Elements of Reusable Object-Oriented Software,* Richard Helm, et al. (Addison-Wesley, 1994).
- A must-read for every user interface designer: *The Design of Everyday Things,* Donald

Norm (MIT Press, 1998).

- For more information on the stack-based framework, read John Muchow's *Core J2ME Technology and MIDP* (Prentice Hall/Sun Microsystems Press, 2001).
- For more information, visit *Wireless zone* on *IBM developerWorks*.
- Visit the author's Web site *http://www.jackwind.net* for more J2ME resources and services.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

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

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.