Developing a 2D game in Java ME - Part 2

This article explains what user interface (UI) elements are available for a MIDlet and how they are used in creating a user interface for the Arkanoid clone. This is the second in a series of articles that cover all the basics of developing an application for mobile devices using Java ME, learning the main libraries, classes, and methods available in Java ME.

Introduction

24 Aug 2008

Since the interaction with the user is the most important issue in any mobile application due to the size of the screen, you need to understand the basics of this side of MIDlets. This article explains which user interface (UI) elements are available on a MIDlet and how they are used in creating a user interface for the Arkanoid clone.

Any user interaction is done through a UI element. In fact, in the Hello World MIDlet in part one, the Alert element was used to show a message on the screen. This message was actually shown on the screen with the help of another UI element called Display.

Let's start with a discussion of the overall architecture of the UI elements.

Architecture

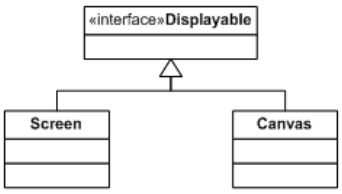
MIDP 2.0 provides UI classes in one package, javax.microedition.lcdui. There are several conflicting sources what "lcdui"-acronym stands for, but one definition is "liquid crystal display user interface" (LCD UI). To show a UI element on a device screen, you must use a class that implements the Displayable interface. A displayable class can, for example, have a title, a ticker, and certain commands associated with it.

The Display class manages what is displayed on the screen. The static method `getDisplay(MIDlet midlet)` gives you access to the Display of your MIDlet. Then you can use the method `setCurrent(Displayable element)` to choose what to display. Only one element can be displayed at a time. See the code from the previous article:

```
Display.getDisplay(this).setCurrent(alert);
```

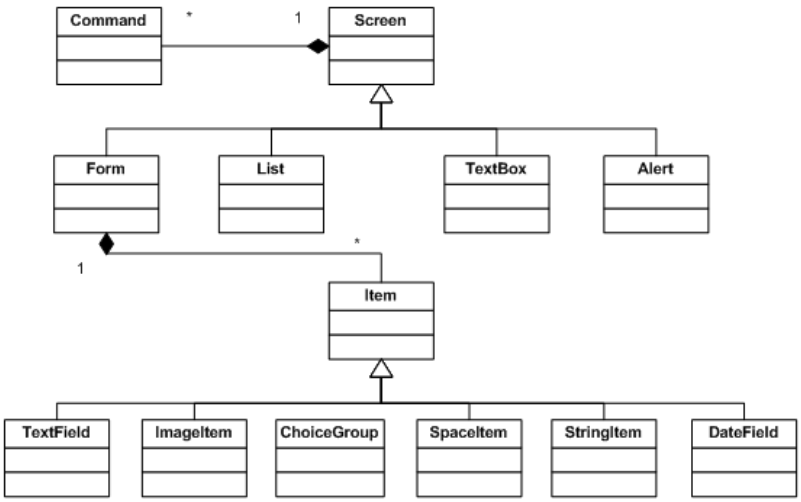
The Displayable classes of MIDP 2.0's javax.microedition.lcdui package can be divided into high-level and low-level groups.

The high-level group is implemented through the Screen class and the low-level group through the Canvas class. Both classes and their subclasses implement the Displayable interface, as in the following figure:



High-level user interface

The classes of the high-level group are perfect for developing MIDlets that target the maximum number of devices because these classes do not provide exact control over their display. The high-level classes are heavily abstracted to provide minimum control over their look and feel, which is left for the device on which they are deployed to manage, according to its capabilities. See the class diagram of these classes:



As you can see, there are quite a few classes that provide user interface elements. Let's analyze each of them.

Command

A MIDlet interacts with a user through commands. A command is the equivalent of a menu item in a normal application, and it can only be associated with a displayable UI element. The Displayable class allows the user to attach a command to it by using the method `addCommand(Command command)`. A displayable UI element can have multiple commands associated with it.

The Command class holds information about the command. This information is encapsulated in four properties: short label, optional long label, command type, and priority. In the Hello World application the command was created by providing these values in its constructor:

```
// adds a command to exit the MIDlet
comExit = new Command("Exit", Command.EXIT, 1);
```

Note that commands are immutable once they have been created.

By specifying the command type, you can let the device running the MIDlet map any predefined keys on the device to the command itself. For example, a command with the type OK is mapped to the device's OK key. The rest of the types are BACK, CANCEL, EXIT, HELP, ITEM, SCREEN, and STOP. The SCREEN type relates to an application-defined command for the current screen. Both SCREEN and ITEM will probably never have any device-mapped keys. In order to receive feedback from the user, you need to listen for commands. This is done by implementing the CommandListener Interface.

In the Hello World MIDlet the CommandListener Interface was implemented through the method `commandAction()`.

```
alert.addCommand(comExit);
// adds a listener to the form
alert.setCommandListener(this);
[...]
public void commandAction(Command cmd, Displayable display) {
  if (cmd == comExit) {
    exit();
  }
}
```

As you can see, the commandAction method receives two parameters: the Command that is executed and the Displayble that is currently shown.

Alert

The Hello World MIDlet used an alert. This element represents a screen that shows data to the user and waits for a certain period of time before proceeding to the next Displayable. An alert can contain a text string and an image. The intended use of Alert is to inform the user about errors and other exceptions.

TextBox

The TextBox class is a Screen that allows the user to enter and edit text. This element can be configured to adapt to your needs. You can restrict the maximum number of characters that a user is allowed to enter into a textbox. You can also constrain the text that is accepted by the textbox, and modify its display using bitwise flags defined in the TextField class. There are six constraint settings for restricting content: ANY, EMAILADDR, NUMERIC, PHONENUMBER, URL, and DECIMAL. ANY allows all kinds of text to be entered, while the others restrict the content according to their names. Similarly, there are six constraint settings that affect the display: PASSWORD, UNEDITABLE, SENSITIVE, NON_PREDICTIVE, INITIAL_CAPS_WORD, and INITIAL_CAPS_SENTENCE. For example, to only accept e-mail addresses in a textbox, you need to set the TextField.EMAILADDR flag using the method `setConstraints()`. To protect this field, you need to combine it with the TextField.UNEDITABLE flag. This is done with a bitwise OR operation between these two flags: `setConstraints(TextField.EMAILADDR | TextField.UNEDITABLE);`.

The contents of a textbox can be set with a couple of methods. Use `setString(String text)` to set the contents with a String value, and `insert(String text, int position)` to insert text in a certain position.

List

A List contains a list of choices. When a List is present on the display, the user can interact with it by selecting elements and possibly by traversing and scrolling among them.

The List can be configured to:

- Choice.EXCLUSIVE - only one element can be selected
- Choice.MULTIPLE - multiple elements can be selected
- Choice.IMPLICIT - the currently hightlighted element is selected

Form

A Form is a Screen that contains an arbitrary mixture of items. In general, any subclass of the Item class may be contained within a form. The implementation handles layout, traversal, and scrolling. The entire contents of the Form scroll together.

There are eight Item types that can be added to a form.

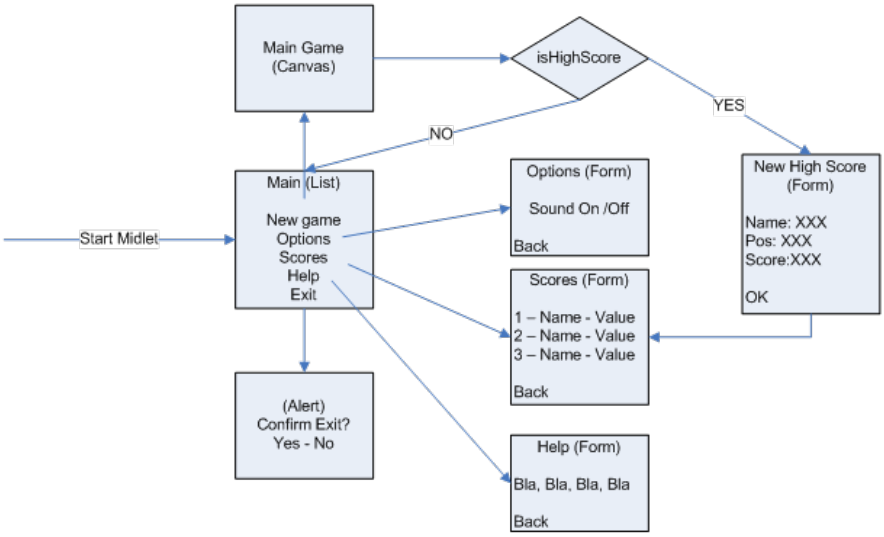
- **StringItem** is a label that cannot be modified by the user. This item may contain a title and text, both of which may be null to allow it to act as a placeholder. The Form class provides a shortcut for adding a StringItem without a title: `append(String text)`
- **DateField** allows the user to enter a date/time in one of three formats: DATE, TIME, or DATE_TIME.
- **TextField** is similar to a TextBox, which was described earlier.
- **ChoiceGroup** is similar to a List.
- **Spacer** is used for positioning UI elements by putting some space between them. This element is an invisible UI element and it can be set to a

particular size.

- **Gauge** is used to simulate a progress bar. However, in addition to this progress bar look, control can also be used in an interactive mode by the user. For example, if you want to show the user a volume control, a gauge can be used to show an interactive knob.
- **ImageItem** holds an image. Like the StringItem, the Form class provides a shortcut method for adding an image: `append(Image image)`. Images are discussed in a later section.
- **CustomItem** is an abstract class that allows the creation of subclasses that have their own appearances, their own interactivity, and their own notification mechanisms. If you need a UI element that is different from the supplied elements, you can subclass CustomItem to create a control that can be added to a form.

User Interface examples

In order to learn how to use all these classes, let's create a simple interface for the Arkanoid game. The following screens are implemented:



For each of these game screens we are going to create a `init[ScreenName]` method that will initialize the screen and return the created Displayable element.

For the Main Menu we will use the List component to show the main options. Check the following code:

```
public Displayable initMainForm() {
  if (mainForm == null) {
    // creates a implicit List where the current element is
    // the selected
    mainForm = new List("Menu", List.IMPLICIT);
    // append list options
    mainForm.append("New Game", null);
    mainForm.append("Options", null);
    mainForm.append("Scores", null);
    mainForm.append("Help", null);
    mainForm.append("Exit", null);

    // adds a select Command
    comSelect = new Command("Select", Command.ITEM, 1);
    mainForm.setSelectCommand(comSelect);

    // adds a listener to the form
    mainForm.setCommandListener(this);
  }

  return mainForm;
}
```

For the settings menu we choose a Form element, and add a Choice Group for Sound Options (On-Off) to it.

```
public Displayable initSettingsForm() {
  // check if already created
  if (settingsForm == null) {
    settingsForm = new Form("Settings");
    settingsForm.addCommand(initBackCommand());
    settingsForm.setCommandListener(this);
    // creates a choice Group for sound options
    soundChoice = new ChoiceGroup("Sound", List.EXCLUSIVE);
    soundChoice.append("On", null);
    soundChoice.append("Off", null);
    // appends the choice to the form
    settingsForm.append(soundChoice);
  }

  return settingsForm;
}
```

For the help screen we selected a simple Form with a static message:

```
public Displayable initHelpForm() {
    if (helpForm == null) {
      helpForm = new Form("Help");
      helpForm
          .append("Use cursors to move your pad, don't let "+
              "the ball go by you, hit all the bricks!");
      helpForm.setCommandListener(this);
      helpForm.addCommand(initBackCommand());
    }
```

```
    return helpForm;
  }
```

To enable user to save a new high score we are going to use a Form with a TextField and a Date Field:

```
public Displayable initNewHighScore(int score, int pos) {
    if (newHighScoreForm == null) {
      newHighScoreForm = new Form("New High Score");
      newHighScoreForm.setCommandListener(this);
      // create items
      highScoreName = new TextField("Name", "", 20, TextField.ANY);
      highScoreValue = new StringItem("Score", Integer.toString(score));
      highScorePosition = new StringItem("Position", Integer.toString(pos));
      // create save command
      highScoreSave = new Command("Save", Command.OK, 1);
      // append command and items to screen
      newHighScoreForm.addCommand(highScoreSave);
      newHighScoreForm.append(highScoreName);
      newHighScoreForm.append(highScoreValue);
      newHighScoreForm.append(highScorePosition);
    }
    // update score
    highScoreValue.setText(Integer.toString(score));
    // update pos
    highScorePosition.setText(Integer.toString(pos)+1);
    return newHighScoreForm;
  }
```

The game screen is discussed in the next article. For the time being, let's create a method that emulates the end of the game and use it instead.

```
  public void endGame(int lifes, int score, int time) {
    Displayable nextScreen = initMainForm();
    String message;
    if (lifes == 0) {
      message = "Game Over!!";
    } else {
      message = "You Win!";
    }
    int pos = isHighScore(score);
    if (pos != -1) {
      nextScreen = initNewHighScore(score, pos);
    }

    display(new Alert(message, message, null, AlertType.INFO), nextScreen);
  }
```

Now that all screens have been created, we need to link them in the `commandAction` method. Rewrite the code:

```
public void commandAction(Command cmd, Displayable display) {
  // check what screen is being displayed
  if (display == mainForm) {
    // check what command was used
    if (cmd == comSelect) {
      switch (mainForm.getSelectedIndex()) {
      case (0):
        // At the moment just go directly to the end of the game
        endGame(1, 200, 50);
        break;
      case (1):
        display(initSettingsForm());
        break;
      case (2):
        display(initScoreForm());
        break;
      case (3):
        display(initHelpForm());
        break;
      case (4):
        exit();
        break;
      }
    }
  } else if (display == highScoreForm) {
    if (cmd == comBack) {
      display(initMainForm());
    }
  } else if (display == settingsForm) {
    if (cmd == comBack) {
      soundOn = soundChoice.getSelectedIndex() == 0;
      display(initMainForm());
    }
  } else if (display == helpForm) {
    if (cmd == comBack) {
      display(initMainForm());
    }
  } else if (display == newHighScoreForm) {
    if (cmd == highScoreSave) {
      int pos = Integer.parseInt(highScorePosition.getText())-1;
      // advance all the scores
      for ( int i = scores.length-1; i > pos ; i--){
        scores[i].name  = scores[i-1].name;
        scores[i].value = scores[i-1].value;
        scores[i].when  = scores[i-1].when;
      }
      // insert new score
      scores[pos].name = highScoreName.getString();
      scores[pos].value = Integer.parseInt(highScoreValue.getText());
      scores[pos].when = new Date();
      display(initScoreForm());
    }
  }
}
```

All the menu logic for the MIDlet is specified inside the commandAction. Deciding what to do next depends on which display is shown and which Command was selected. From the main menu, we simply redirect the user to each specific screen. The other screens currently only have a back command. The only exception is the NewHighScores form where a save command already stores the information to a scores array.

You may have noticed the use of the `display()` method. This is a simple helper method to activate a Displayable.

```
public void display(Displayable display) {
  // shows display in the screen
  Display.getDisplay(this).setCurrent(display);
}
```

Now just run your MIDlet and enjoy your first Game Interface. The next article describes how to implement the Game Screen.

Downloads

- Source code
- Binary files

Go To Developing a 2D game in Java ME - Part 3