



Mobile Information Device Programming (5)

Lecturer: Alireza Mousavi
School of Engineering & Design
www.brunel.ac.uk/~emstaam



MIDP User Interface (UI) Architecture

Issues that are main concerns of programming UI in mobile devices are:

- The screen size of mobile devices
- The text entry capabilities of a MIDP device



UI in J2ME

The UI architecture that is defined in J2SE is not suitable for MIDP:

- The AWT and Swing libraries are designed for computers and not for handheld devices – e.g. features like horizontal scrolling
- The AWT and Swing Libraries are too large for the limited computational and processing resources of a hand-held device



MIDP UI Component Model

- The MIDP UI components are defined in the:
javax.microedition.lcdui
- The MIDP UI components comprise the majority of classes in this package
- Understanding the organisation of these UI components is essential in developing MIDP applications



Example

In the `startApp()` method you may have:

```
display = Display.getDisplay(this);  
// Reference to a display object  
display.setCurrent(form);  
// Make this form the currently displayed entity
```



What Happens when an AMS launches a MIDlet?

- It instantiates the ***Display*** class
- It associates the ***Display*** class with the MIDlet instance
- ***Display*** can show 1 displayable at a time and we have no control how they are displayed (device dependent)



MIDP High-Level & Low-Level API

MIDP UI architecture defines two levels of API:

- High-Level UI API
- Low-Level UI API

In order to develop High-Level and Low-Level UIs, MIDP UI architecture defines all classes, interfaces and exceptions.



High-Level & Low Level UI API

"The two levels of UI API levels provide you with the option to choose the correct interface detail for your applications" (SUN Microsystems)



High-Level UI API

- This API provides an abstract and portable interface
- It is used by business applications
- This interface provides general UI controls and event-handling mechanisms (SUN Microsystems)
- It provides less device reliant coding for UI developments (portability issues)



Low-Level API

- Provides little GUI abstraction
- It is designed for drawing capabilities
- It contains the Canvas class and Graphics class
- It provides access to specified keystrokes
- The low-level access is more device specific (less portability)



The *Display* class

- It manages the display for implementation
- The *Display* object represents device display
- There is only one Display instance in a MIDlet
- You cannot instantiate a Display object (why?)
- You need to call the method:

Example: `display = Display.getDisplay (this);` //refers to the running MIDlet [static]

- `DestroyApp(boolean)` method destroys the display



The *Displayable* class

- Is the abstract class that the *Display* object presents
- This class is a super class for all things that can be placed on a display
- The class defines all the methods that ensure consistency and functionality across all subclasses
- The *Displayable* has two subclasses:
 - Screen: high-level UI
 - Canvas: low-level UI



Event Handling

- Discussion about classes that facilitate event processing
- We will first discuss the high-level (Screen) UI and later the low-level (Canvas)



What is Event Handling

Event Handling is recognising :

- that an event i.e. button pressed, has been triggered and
- the action required e.g. help message appears



Steps to successfully manage an event (Muchow 2002)

- The hardware should realise that a button has been pressed or released (event trigger),
- The AMS needs to be notified of the event,
- The AMS will send the message to the MIDlet containing information about the event, and
- The MIDlet will implement the code e.g. display a message



CommandListener and ItemStateListener

Before a MIDlet can recognise a message from the AMS about an event, it must set up a "*listener*"

There are two *listener* interfaces in MIDP:

- CommandListener → method: `commandAction()`
- ItemStateListener → method: `ItemStateChanged()`



Command Objects

A **Command** is an object that holds information about an event.

Three steps to implement a command

1. Create the Command
2. Add the command to the environment (Form, Text Box, etc.
3. Add a “listener” to the environment



Example – Command Object

Define a form, add command and listener

```
private Form    myForm; // reference to a form
private Command cmExit; // reference to a command to exit the MIDlet
...
myForm = new Form("Welcome"); // the form object
cmExit = new Command("Exit", Command.EXIT, 1); // Command object
...
myForm.addCommand(cmExit); // Add command to the form
myForm.setCommandListener(this); // Listen for events
...
public void commandAction(Command c, Displayable s){ // CommandAction method
    if(Command == cmExit) {
        DestroyApp(true);
        notifyDestroyed( );
    }
}
```



Item Objects

An **Item** is a component that can be added to form. It is another event-handling tool.

The item class contains a number of subclasses ([see slide 11](#))



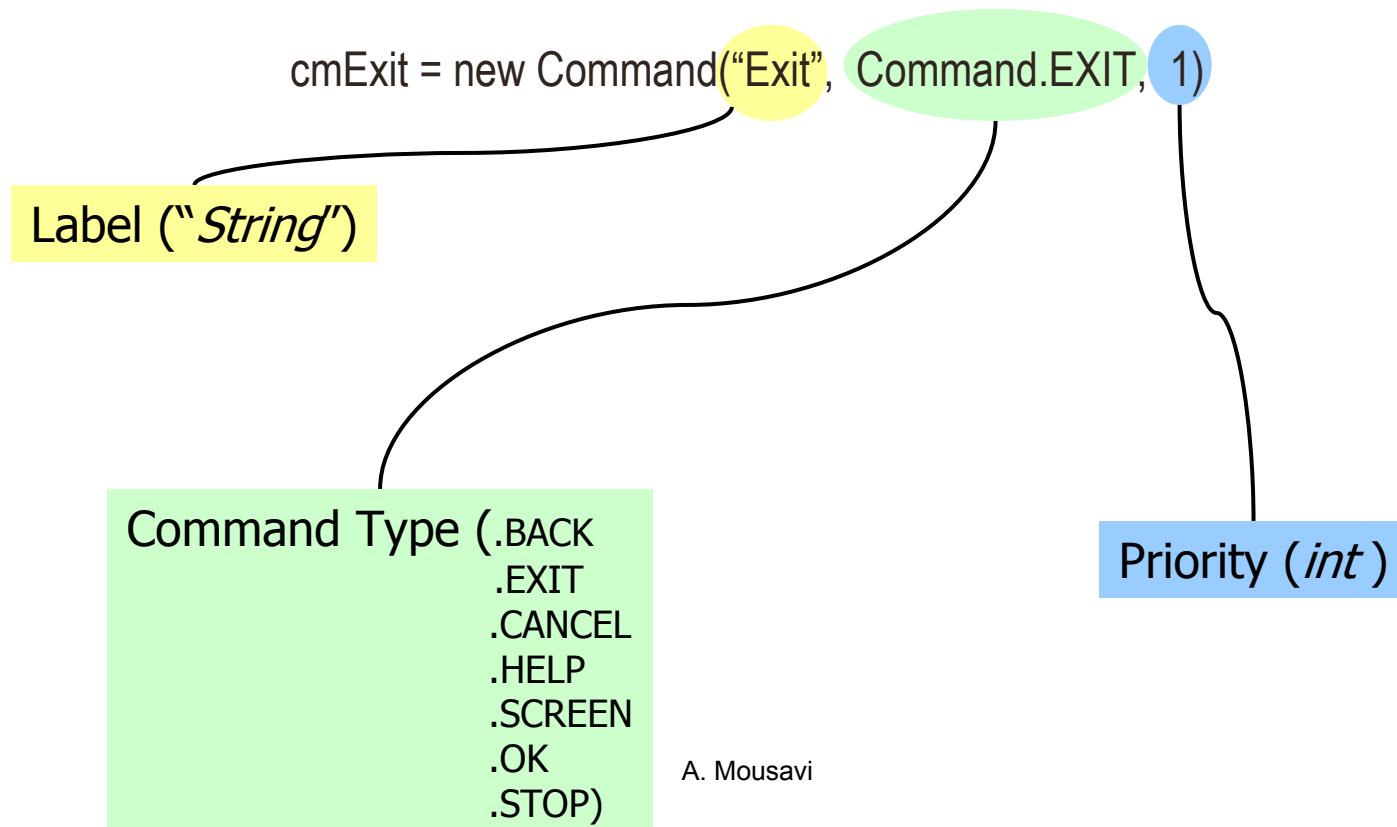
Example Item Object

```
private Form fmMain; // reference to a form
private tfPhone; // reference to a text field
...
fmMain = new Form("Phone No."); // Form object
tfPhone = new TextField("Phone Number: ", " ", 10, TextField.PHONENUMBER); // TextField
...
fmMain.append(tfPhone); // add tfPhone to the form
fmMain.setItemStateListener(this); // listen to the events
...
public void itemStateChanged(Item item) {
    if(item == tfPhone) { // if the TextField has initiated this event ...
        ...
    }
}
```



Command & Command Listener

A command object holds three parameters:





Exercise

Create an application with an Exit Command.