Event Handling in Java ME

Version 0.9, Draft

INFORMATION GUIDE

COPYRIGHT

Trademarks and Service Marks

About This Document

This document describes the event handling in Java ME and provides sample code snippet demonstrating handling of high level and low level events.

Scope:

This document is intended for novice Java ME user. Focusing on Java and Java ME is out of scope of this document.

Document History:

Date	Version	Comment
11/10/09	0.9	Draft

Abbreviations:

J2ME	Java 2 Micro Edition.
MIDP	Mobile Information Device Profile
CLDC	Connected Limited Device Configuration
API	Application Programming Interface

Table of Contents

Table of Figures

Introduction

When a user interacts with a MIDlet loaded on a mobile device, some events are generated. For example, if the user selects Exit on device, the application is notified of this action and responds to the generated event, and the application exit. But how does the application get notified of an event, and how does it handle it, this is the focus of this document.

MIDP user interface APIs provide two kinds of events.

- High - Level
- Low - Level

Overview

In Java ME, events represent all activities between the user and the MIDlet. MIDlet communicates these actions to the programs using events. When the user interacts with a program by pressing a command button, the system creates an event representing the action and hands it to the event-handling code within the program. This code determines how to handle the event so that the user gets the appropriate response.

CallBack

When a user interacts with a MIDlet, events are generated and the application is modified to handle and respond to these events. The application is notified of such events through callbacks. Callbacks are invocation of programmer-defined methods that are executed by the application in response to actions taken by a user at run time.

In more detail its normal parameter passing mechanism lets caller provide the callee with information, but provides only meager flow of information in the reverse direction, the single returned result primitive or Object. Further the callee cannot provide further information back to the caller once it has returned. The callback mechanism allows a two-way flow of information and exchange of computing resources between caller and callee.

Consider Form containing command and caller wants to control on command. The callee wants to notify the caller whenever one of the command was pressed, and which one. The callback mechanism, in Java called as Listener mechanism, allows the callee to

notify the caller, the listener, of interesting events, or to use some of the methods the caller has access to.

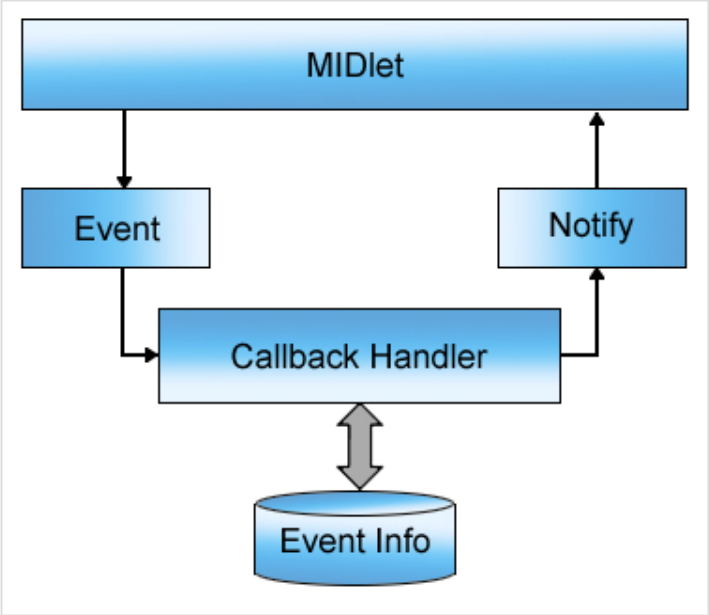

Figure 1: Event Handling in MIDlet

In Java callbacks are implemented using Interface.

There are four kinds of user interface callbacks in MIDP:

- Abstract commands that are part of the high-level API.
- Low-level events that represent single key presses and releases.
- Calls to the *paint()* method of a Canvas class.
- Calls to a Runnable object's run method requested by a call to the callSerially method of the Display class.

High-Level Event Handling

Handling events in the high-level API is based on a listener model.

- *CommandListener* Interface
- *ItemStateListener* Interface

Command:

The *javax.microedition.lcdui.*;* package provides the Command class.

```
Command(String label, int commandType, int priority)
```

Command class constructor encapsulates the semantic information about the command and no action. The actual action happen when the command is activated.

label –The label is a string used for the visual representation of the command.

commandType - The type specifies the command's intent. The defined types are: BACK, CANCEL, EXIT, HELP, ITEM, OK, SCREEN, and STOP.

priority - The priority value describes the importance of this command relative to other commands on the screen. A priority value of 1 indicates the most important command, and higher priority values indicate commands of lesser importance.

When the MIDlet executes, the device chooses the placement of a command based on the command type, and places similar commands based on their priority values.

```
…
exitCommand = new Command("Exit", Command. EXIT, 1);
backCommand = new Command("Back", Command. BACK, 1);
moreCommand = new Command("More", Command.SCREEN, 2);
…
```

The Command class provides the following methods for retrieving the type, label, and priority values. This information is helpful when handling events.

- *int getCommandType()*
- *String getLable()*
- *int getPriority()*

As discussed earlier command only encapsulates the semantic information about the command and no action.The actual action happen when the command is activated.

This is done by:

1. Adding Commands to the displayable object using *addCommand(Command cmd)*;
2. Adding CommandListener to the displayable object using *setCommandListener(CommandListener l)*.

CommandListener Interface

Commands are added to a displayable object with the *addCommand(..)* method. However no action will be performed when a command is activated. To perform an action when a command is activated user needs to register a *CommandListener* with the *setCommandListener(..)* method. *CommandListener* interface has only one method *commandAction(Command c, Displayable d)* that the listener must implement, in order to perform action on command activation.

```
public void commandAction(Command cmd, Displayable dis)
```

cmd - is a command object that identifies the command.

dis - is the displayable object which indentify where the event occurred.

The following code sample explains implementation of CommandListener.

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
public class MainMidlet extends MIDlet implements CommandListener {

        private Display display=null;
        private Command cmd_Exit = null;
        private Form form;

        public MainMidlet()
        {
                display = Display.getDisplay(this);
                form = new Form ("Command  Handling...");
                cmd_Exit = new Command("Exit",Command.EXIT,1);
                form.addCommand(cmd_Exit);
                form.setCommandListener(this);
                display.setCurrent(form);
        }
```

```
        …
        …

        public void commandAction(Command cmd,Displayable dis)
        {
                if(cmd == cmd_Exit)
                {
                        try {
                                destroyApp(true);
                                notifyDestroyed();
                        } catch (MIDletStateChangeException e) {
                                e.printStackTrace();
                        }
                }
        }

}
```

ItemStateListener Interface

Applications use the *ItemStateListener* interface to receive events that indicate changes in the internal state of items within a *Form* screen. This happens when the user does any of the following:

- Adjusts the value of an interactive *Gauge*.
- Enters or modifies the value of a *TextField*.
- Enters a new date or time in a *DateField*.
- Changes the set of selected values in a *ChoiceGroup*.

This interface has only one method that a listener must implement:

```
public void itemStateChanged(Item item)
```

User needs to set ItemStateListener using the *setItemStateListener(ItemStateListener iListener)* method to register a listener.

The following sample code snippet implementing *ItemStateListener* interface

```
…
form.setItemStateListener(this);
display.setCurrent(form);
…
…
public void itemStateChanged(Item item) {

        if(item instanceof TextField){

        String data = textField.getString();
        if(data != null  && data.length() >=10){

        Alert alert = new Alert("TextField Data..",
        data,null,AlertType.CONFIRMATION);
        alert.setTimeout(1000);
        display.setCurrent(alert);

        }
}
…
```

ItemCommandListener

When a command attached to an Item, is invoked, the application is notified by having the *commandAction(Command c , Item item)* method called on the *ItemCommandListener* that had been set on the Item

This interface has only one method that a listener must implement:

```
public void commandAction(Command cmd,Item item)
```

Use the *javax.microedition.lcdui.Item.setItemCommandListener (ItemCommandListener l.)* method to register a listener.

The following sample code snippet implementing *ItemCommandListener* Interface

```
…
textField = new TextField("Name",null,30,TextField.ANY);
dateField = new DateField(null,DateField.DATE_TIME);
```

```
dateField.addCommand(cmd_Date);
dateField.setItemCommandListener(this);
textField.addCommand(cmd_Get);
textField.setItemCommandListener(this);
form.append(dateField);
form.append(textField);
…
…
public void commandAction(Command cmd, Item item) {

                if((cmd == cmd_Get) && (item == textField))              {

                        Alert alert = new
                                Alert("Info",textField.getString(),null,AlertType.INFO);
                        alert.setTimeout(5000);
                        display.setCurrent(alert,form);
                }
                else if(( cmd == cmd_Date)&& (item == dateField)) {

                        Alert alert = new
                              Alert("Info",dateField.getDate().toString(),null,AlertType.INFO);
                        alert.setTimeout(5000);
                        display.setCurrent(alert,form);

                }
        }
…
```

Low-Level Event Handling

Low level events are events associated with the use of *javax.microedition.lcdui.Canvas* class in MIDlet application that handle the low – level input events , Commands or a graphics calls for drawing to the display.

Handling Commands:

Commands can be added to a Canvas in order to map soft-keys, just like High-Level UI, the application must register a listener for commands.

The following sample code snippet implementing *CommandListener* in Canvas.

```
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.Displayable;
import javax.microedition.midlet.MIDletStateChangeException;

public class MainCanvas extends Canvas implements CommandListener{

        private Command cmd_Exit = null;
        private MainMidlet mainMidlet=null;
        public MainCanvas(MainMidlet mid) {
                mainMidlet = mid;
                cmd_Exit = new Command("Exit",Command.EXIT,1);
                addCommand(cmd_Exit);
                setCommandListener(this);
        }

        protected void paint(Graphics g) {

                ...
        }
        public void commandAction(Command cmd,Displayable dis)      {
                if(cmd == cmd_Exit){

                        try {
                                mainMidlet.destroyApp(true);
                                mainMidlet.notifyDestroyed();
                        } catch (MIDletStateChangeException e) {
                                e.printStackTrace();
                        }

                }

        }
}
```

Handling Key Events and Game Action:

Canvas also allows capturing of raw key-press events for most of the keys available on the device. The minimal set of keys required by the MIDP specification to be captured is:

- The digits 0 through 9
- The star or asterisk character (*)
- The pound or hash character (#)

Every key for which events are reported is assigned a key code. If your application needs arrow key and gaming-related events, use game actions instead of key codes.

Following table explain the Standard key codes and Game Action table.

KeyCode	Meaning
KEY_NUM0	Number key 0
KEY_NUM1	Number key 1
KEY_NUM2	Number key 2
KEY_NUM3	Number key 3
KEY_NUM4	Number key 4
KEY_NUM5	Number key 5
KEY_NUM6	Number key 6
KEY_NUM7	Number key 7
KEY_NUM8	Number key 8
KEY_NUM9	Number key 9
KEY_STAR	The star key (*)
KEY_POUND	The pound key (#)
UP	Game action UP
DOWN	Game action DOWN
LEFT	Game action LEFT
RIGHT	Game action RIGHT
FIRE	Game action FIRE
GAME_A	Custom game action A
GAME_B	Custom game action B
GAME_C	Custom game action C
GAME_D	Custom game action D

Event Handling Methods

In order to monitor key event, a subclass of Canvas must override one of these methods. It is important to note that the key codes are likely to be change from platform to platform.

Method	Description
keyPressed(int keyCode)	This event is called when a physical key on the keypad is pressed.
keyReleased(int keyCode)	This event is called when a pressed key is released.
keyRepeated(int keyCode)	This event is called when a key is held down. This event may not be supported on all platforms. Support of this method can be verified with a call to *hasRepeatEvents()*.
pointerPressed(int x,int y)	This event is called when the pointer pressed. Support of this method can be verified with a call to *hasPointerEvents()*
pointerDragged(int x,int y)	This event is called when the pointer dragged. Support of this method can be verified with a call to *hasPointerMotionEvents()*
pointerReleased(int x,int y)	This event is called when the pointer released. Support of this method can be verified with a call to has *hasPointerEvents()*
showNotify()	This event is called when Canvas is visible on display.
hideNotify()	This event is called when Canvas hide form display.
paint(Graphics g)	To renders the Canvas. The application must implement this method in order to paint any graphics.

Key Handling Across Devices

Given that key codes can vary across platform to platform, MIDP provides key mappings so that device implementers can map the raw key events into known constant values. The simplest form of this is the numeric key mappings in the Canvas class. There are 10 constant values corresponding to a 10-digit numeric keypad, plus two more for the "*" and "#" keys.

So if the application need to detect which key is pressed do something like

```
…
protected void keyPressed(int keyCode) {

switch (keyCode){
…
case Canvas.KEY_NUM3:
        System.out.println(getKeyName(keyCode));
break;
...
  }
}
…
```

If the application need to detect which game action key is pressed do something like

```
protected void keyPressed(int keyCode){

int action = 0;
 try {
       action = getGameAction(keyCode);

     } catch (Exception e) {
          e.printStackTrace();
         }
  switch(action){
           case Canvas.DOWN:
                   System.out.println(getKeyName(keyCode));
                  break;
          case Canvas.GAME_A:
                   System.out.println(getKeyName(keyCode));
                    break;
          }
}
```

Following code snippet demonstrate the key code handling on Canvas

```
import javax.microedition.lcdui.Canvas;
import javax.microedition.lcdui.Graphics;
import javax.microedition.lcdui.Font;
```

```
public class MainCanvas extends Canvas {

        private MainMidlet mainMidlet=null;
        private String keyPress = "";
        private int screenWidth;
        private int screenHeight;
        private String KeyValue="Key Press : ";
        private Font newFont;

        public MainCanvas(MainMidlet mid) {
                mainMidlet = mid;
                screenWidth = getWidth();
                screenHeight = getHeight();
                newFont =

Font.getFont(Font.FACE_SYSTEM,Font.STYLE_PLAIN,Font.SIZE_MEDIUM);

        }

        protected void paint(Graphics g) {
                g.setFont(newFont);
                g.setColor(255,255,255);
                g.fillRect(0,0,screenWidth,screenHeight);
                g.setColor(0,0,0);
                g.drawString(KeyValue+keyPress,((screenWidth/2)-
                                        (newFont.stringWidth(KeyValue+keyPress)/2))
                                    ,screenHeight/2,Graphics.LEFT|Graphics.TOP);

        }
        protected void keyPressed(int keyCode)  {

                switch(keyCode) {

                case Canvas.KEY_NUM0:
                        keyPress = getKeyName(keyCode);
                        break;
                case Canvas.KEY_NUM1:
                        keyPress = getKeyName(keyCode);
                        break;
                case Canvas.KEY_NUM2:
                        keyPress = getKeyName(keyCode);
```

```
            break;
        case Canvas.KEY_NUM3:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_NUM4:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_NUM5:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_NUM6:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_NUM7:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_NUM8:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_NUM9:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_POUND:
            keyPress = getKeyName(keyCode);
            break;
        case Canvas.KEY_STAR:
            keyPress = getKeyName(keyCode);
            break;
            default:
                int action = 0;
                try {
                     action = getGameAction(keyCode);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                switch(action){

                case Canvas.DOWN:
                    keyPress =getKeyName(keyCode);
                    break;
                case Canvas.UP:
```

```
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.FIRE:
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.LEFT:
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.RIGHT:
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.GAME_A:
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.GAME_B:
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.GAME_C:
                                keyPress = getKeyName(keyCode);
                                break;
                        case Canvas.GAME_D:
                                keyPress = getKeyName(keyCode);
                                break;
                        default:
                                System.out.println("Not match..");
                        }
                }
                repaint();
        }
}
```