9

MIDlet 的事件處理

Versioning，注意，是把 version 這個字當動詞用，也是軟體公司賺大錢的技倆。

- 前言
- MIDlet 程式結構
- MIDlet 的生命週期
- MIDlet 的基本構造
- 實體機器上的輸入介面
- MIDlet 事件處理
- 總結

前　言

從本章開始，我們將要學習整個 MIDlet 的生命週期，試著和 Java Application Manager 打交道，這可以讓您各了解 MIDlet 的生與死。

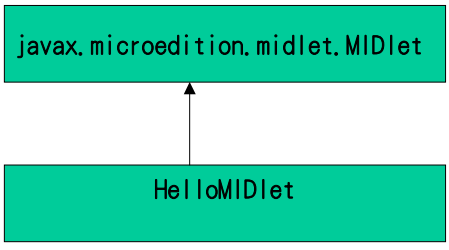
同時，我們也會討論 MIDlet 的運作核心 - 事件處理。因為唯有了解整個程式運作的來龍去脈，才能掌控所有的細節，我們也才有本錢駕馭像 JBuilder MobileSet 這種功能強大的開發工具。

還沒讀通本章的朋友，請先不要直接使用 JBuilder MobileSet 來開發您的 MIDlet，因為這只會讓您更加疑惑。

MIDlet 程式結構

如果各位曾經撰寫過 Java Applet 或是 Java Servlet，一定知道要製作 Java Applet，就必須繼承自 java.applet 這個類別，要製作 Java Servlet，則程式必須繼承自 javax.servlet.http.HttpServlet 這個類別。同理，要撰寫能夠在手機上頭執行的 Java MIDlet 也必須要繼承自 javax.microedition.midlet.MIDlet 類別，如下圖所示：

MIDlet的繼承體系



javax.microedition.midlet.MIDlet 類別中定義了三個抽象方法（abstract）他們分別是：

startApp()　➞ 至運作狀態。

pauseApp()　➞ 至停止狀態。

destoryApp()➞ 至消滅狀態。

也就是說，所有我們撰寫的 MIDlet 都必須實做這三個方法，因此一個 Java MIDlet 的程式外殼至少要如下：

```
import javax.microedition.midlet.*;
public class HelloMIDlet extends MIDlet
{
   public HelloMIDlet()
   {
      //建構式
   }
   public void startApp()
   {
   }
   public void pauseApp()
   {
   }
   public void destroyApp(boolean unconditional)
   {
   }
}
```

根據 MIDP 規格，MIDlet 中不應該有

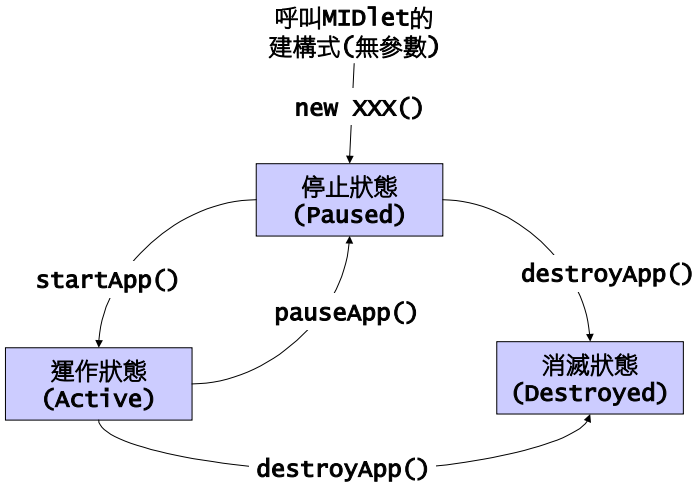
public static void main(Straing[] args)

這個方法。如果有的話，Java Application Manager（JAM）會忽略不管。

一旦 MIDlet 被 JAM 載入之後，首先會先呼叫 MIDlet 中沒有參數的建構式以進行初始化的工作，熟悉 Java 語法的朋友一定知道，如果您沒有在程式中加入任何建構式，編譯器會自動幫您加入一個預設建構式（default constructor），但是如果您撰寫了自己的建構式（有任何參數），那麼編譯器將不會自動幫您加上預設建構式，因此，萬一您有必要撰寫有參數的建構式，別忘了再手動加上一個沒有參數的建構式，否則 MIDlet 無法正確地初始化。

MIDlet 的生命週期

當 MIDlet 成功地初始化之後，就開始展開了它的生命週期。底下這張圖，描述一個 MIDlet 的生命週期，MIDlet 的生命週期完全由 Java Application Manager 控制，也就是說，當 MIDlet 要從一個狀態變成另外一個狀態時，Java Application Manager 會呼叫對應的函式，如果狀態轉換時發生錯誤，那麼 Java Application Manager 會丟出 MIDletStateChangeException 例外。

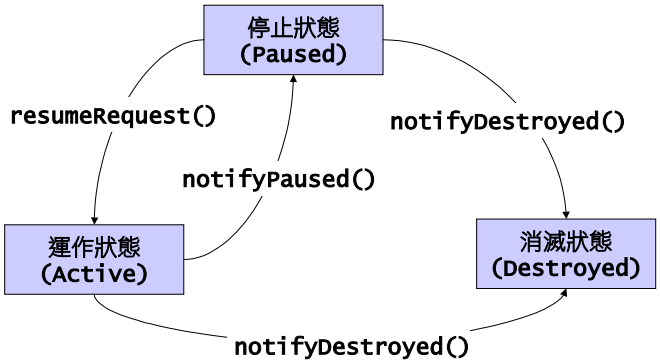


看過上面這張圖之後，我們先給大家一個概念，那就是：只有當 Java Application Manager 認爲 MIDlet 的狀態必須改變時，才會呼叫圖中的相關函式，這些函式呼叫成功之後，Java Application Manager 才會改變 MIDlet 的狀態。因此，如果 MIDlet 自己叫用這些函式，並不會發生錯誤，但是也不會造成狀態的轉換，只是一個單純的函式呼叫而已。

從圖中我們可以看出 startApp()很可能不光只有被呼叫一次而已，而是每次從停止狀態重新回到運作狀態的時候都會被 Java Application Manager 呼叫。所以只需要被初始化一次的變數就不適合在 startApp()之中做初始化，請改用建構式做初始化動作。

其實 MIDlet 自己本身也可以控制自己的狀態，但是不是自己改變自己的狀態，而是先自己呼叫上述相對應的狀態改變函式，然後發出訊息通知 Java Application Manager，請它來幫我們改變

MIDlet 的狀態，方法如下圖所示：



對照以上這兩張圖，舉個例子大家可能會比較清楚:假設今天如果是 MIDlet 主動要將 MIDlet 的狀態由運作狀態變成停止狀態，那麼我們光是直接呼叫 pauseApp()函式，只會執行 pauseApp()之中的程式碼而已，卻不會改變 MIDlet 的狀態，因此在呼叫 pauseApp()之後，MIDlet 要另外呼叫 notifyPaused()以通知 Java Application Manager，Java Application Manager 收到通知之後，才會讓 MIDlet 進入停止狀態。從這裡我們可以看出 startApp()、pauseApp()以及 destroyApp()並非控制 MIDlet 生命週期的函式，他們只是一個提供我們初始化資源、釋放資源的地方而已。

注 意

根據 MIDP 1.0 規格書中所說，即使 MIDlet 處於停止狀態，但是它仍然可以處理非同步事件（Asynchronous Event），比方說 Timer 的事件或是其他回呼函式（callback）。

從以上幾張圖中可以看出，只要 MIDlet 進入了毀滅狀態，就無法再回頭。但是根據規格，我們無法在 MIDlet 之中直接呼叫 System.exit()或 Runtime.exit()來結束程式的執行，如果您這樣做的話會引發 java.lang.SecurityException 例外。MIDlet 一定要自己先呼叫 destroyApp()，然後再呼叫 notifyDestroyed()，請 Java Application Manager 幫我們將 MIDlet 轉換到毀滅狀態，然後結束 MIDlet 的運作。單單 MIDlet 自己呼叫 destroyApp()根本無濟於事，作用就如同單純的函式呼叫一般，完全不會改變 MIDlet 的狀態，這點請大家務必注意。

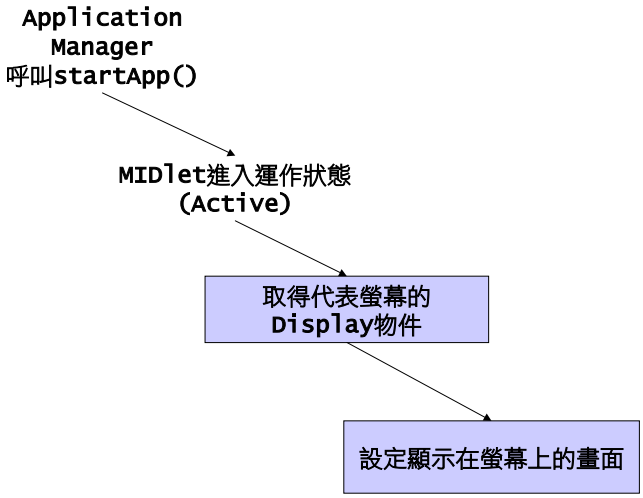
注　意

在我們之後的範例程式之中，我們常常只呼叫 notifyDestroyed()，而沒有呼叫 destroyApp()，這是因為我們的範例通常沒有釋放資源的需求，所以可以不用呼叫。但是如果是正規的程式，建議您記得呼叫 destroyApp()比較好。

destroyApp()有個布林值作為參數，根據 MIDP 的規格，如果使用者傳入 true，那麼 MIDlet 不管如何應該無條件釋放所有資源，然後讓 Java Application Manager 結束 MIDlet 的運作。如果使用者傳入 false，而 MIDlet 不希望結束執行，那麼它可以藉由丟出 MIDletStateChangeException 例外告知呼叫它的人：“我還不想被消滅”。

MIDlet 的基本構造

一般來說，當我們的 MIDlet 開始運作狀態時，我們應該做下列事項：



要取得代表螢幕的 Display 物件，我們會使用

```
Display.getDisplay(this)
```

來取得代表該 MIDlet 顯示幕的 Display 物件。從 Java Application Manager 呼叫 startApp()到呼叫 destoryApp()這段時間內，不管何時呼叫 Display.getDisplay(this)，取得的都是同一份 Display 物件的參考。

要設定顯示在螢幕上的畫面，我們會使用

```
display.setCurrent（Displayable 類別的子類別實體）
```

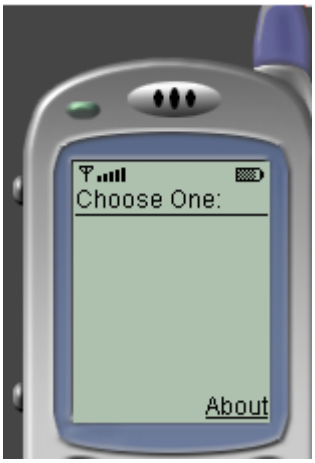
有關 Displayable 類別的子類別我們會在稍後談到，比方說 TextBox 就是一個 Displayable 類別的子類別。因此一個可以顯示畫面的 MIDlet 的程式至少要如下：

```
import javax.microedition.midlet.*;
public class HelloMIDlet extends MIDlet
{
   private Display display;
   public HelloMIDlet()
   {
      ... ...
      display = Display.getDisplay(this);
      ... ...
   }

   public void startApp()
   {
      ... ...
      TextBox t = new TextBox(... ...);
      display.setCurrent(t);
      ... ...
   }

   public void pauseApp()
   {
   }

   public void destroyApp(boolean unconditional)
   {
   }
}
```

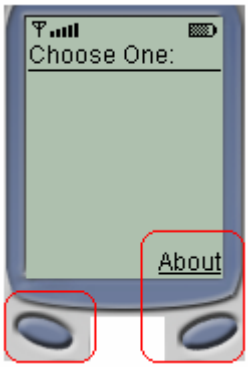
實體機器上的輸入介面

MIDlet 需要透過機器上的輸入裝置和使用者互動，而在不同的機器上，其輸入裝置也會有些許的差異，有些甚至有很大的不同。但是開發 MIDlet 的程式設計師並不用擔心這一點，因為底層的 KVM 會自動幫我們對應。只要您撰寫的是標準的 MIDlet，KVM 會自動讓該機器的使用者使用這台機器上其他軟體一般慣用的方式和 MIDlet 做互動，也會根據該機器實體的螢幕大小來校正 MIDlet 實際上的內容。在 MIDP 參考實作中內附的模擬器，具有下面幾種輸入裝置，分別是：液晶螢幕、軟體按鈕、控制鍵、方向鍵、以及字母數字鍵，底下我們一一討論這些輸入裝置。

- 液晶螢幕（LCD Screen），有些裝置的液晶螢幕有可能具有觸控的功能（Touch Panel）。



◆ 軟體按鈕（Soft Button），用來對應螢幕左下角和右下角的命令，如下圖的 About 命令就對應到右邊的軟體按鈕。



◆ 控制鍵（Control key），一般的用途是用來撥電話與接電話，和 MIDlet 比較沒有直接的關係，不過掛斷電話用的按鈕可以用來終止目前正在執行的 MIDlet，當您按下它時，它會呼叫目前執行的 MIDlet 的 destoryApp()方法，並傳入 true 當作參數（也就是說，MIDlet 沒有任何選擇，一定要結束執行），destoryApp()執行之後，不管 MIDlet 願不願意，Java Application Manager 會立刻將 MIDlet 的狀態改變為毀滅狀態。這種結束 MIDlet 的方式和一般我們按下軟體按鈕所對應的 Exit 命令來結束程式有一些不同，使用軟體鍵盤的話只是產生一個事件，當您的程式攔截到 Exit 所產生的事件時，我們可以改呼叫 destoryApp(false)，這樣 MIDlet 就有比較多的轉還餘地。

◆ 方向鍵（Navigation Key），用來在螢幕上做上下左右的移動，當有選項需要選擇或者玩遊戲的時候特別有用。其中，  用來作為「確認」之用：

◆ 字母數字鍵（Alphanumeric Key），用來輸入字母、數字、以及符號之用，其中  用來切換目前您要輸入的是大寫字母、小寫字母、數字、或是符號，  可以用來刪除輸入的字母、數字、或符號。在台灣發行的手機上，字母數字鍵除了用來輸入字母、數字、和符號之外，上面也會有ㄅㄆㄇㄈ等注音符號，讓我們可以輸入中文字。

注 意

這裡所說明的操作介面是根據 MIDP 參考實作之中的模擬器來討論。當您的 MIDlet 在各種不同的手機或者 PDA 上執行的時候其操作介面很可能有很大的不同。請參閱該機器的使用說明。

MIDlet 事件處理 ▼

接下來我們要討論的是 MIDlet 的事件處理，當 MIDlet 需要和使用者互動的時候，就必須要透過事件處理機制幫助 MIDlet 洞悉使用者的行為。根據 MIDP 1.0 的規格，事件處理分成高階事件處理和低階事件處理。顧名思義，使用高階事件處理機制的程式撰寫起來比較輕鬆，而利用低階事件處理機制所撰寫的程式比較複雜，但是您也可以混合兩者一起用。另外請大家注意，在 MIDP 1.0 規格之中提到，如果您的程式只使用高階事件處理機制，那麼您的 MIDlet 將是可移植的，但是如果採用了低階事件處理機制，那麼將不保證您的程式可以在不同的機器上執行，也不保證會有相同的執行結果。

不管是高階事件處理還是低階事件處理，在 MIDP 中都是透過回呼函式（callback method）來達成。所謂的回呼函式，大家可以想成是一組

事件 ⟷ 處理函式

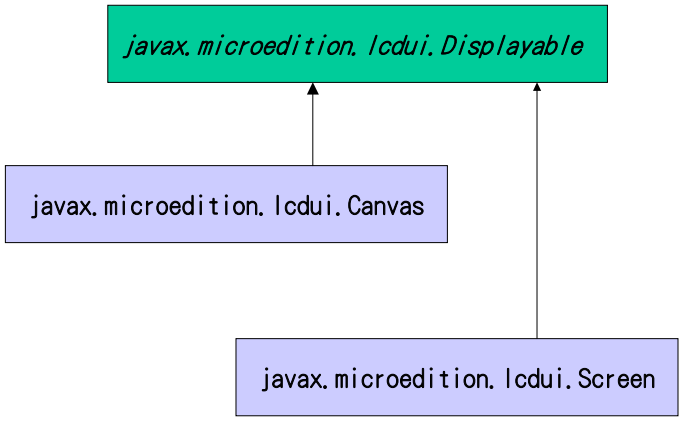
的組合。也就是說，當某個事件發生的時候，就去呼叫特定的函式，並傳給它特定的參數。回呼函式在 C 或是 C++ 之中都是利用函式指標（function pointer）來實現，但是在 Java 中，由於並沒有指標機制，所以回呼函式都是倚賴介面（interface）來完成。

在 MIDP 中，和使用者介面相關的事件 ⟷ 處理函式組合共有四個：

1. 高階事件處理 ➞ 藉由抽象命令（abstract command）來達成。
2. 低階事件處理 ➞ 當時體機器上的按鈕或是 LCD 螢幕被觸動時，就呼叫對應的事件。
3. 繪圖事件處理 ➞ Canvas 類別的 paint()方法會在螢幕需要重繪時被呼叫，並傳入一個 Graphics 物件的參考。
4. 呼叫 Display 類別的 callSerially()方法時會引發繼承 Runnable 介面之類別的 run()方法被呼叫。

在本章中我們只討論 1 和 2，而有關 3 和 4 的部分我們會在往後的章節談到。請注意，在 MIDlet 之中，所有的回呼函式都是依序執行（serialized），也就是說這些對應到某事件的事件處理函式不會同時發生（parallel），就算同時間發生了許多事件，這些相對應的事件處理函式也會一個接著一著執行。此外，所有在 MIDP 之中與使用者介面相關的類別函式庫在多緒（multi-thread）的情況下都能夠安全地使用，他們也包含了事件同步機制。應用程式可以使用 Display 類別的 callSerially()方法，如果就可以用事件處理的方式依序處理我們想做的事情。

其實在 MIDP 1.0 之中，所具有顯示在螢幕之上能力的元件都是來自於 Displayable 這個抽象類別，而 Displayable 又衍生出了 Screen 類別和 Canvas 類別，如下圖所示：



其中，Screen 幾乎用來處理高階事件，而 Canvas 用來處理低階事件。

首先，讓我們來談談較簡單高階事件處理機制。

高階事件處理

在 MIDP 之中，共有兩種高階處理事件介面，它們分別是 CommandListener 以及 ItemStateListener。

首先，我們要先討論 CommandListener 這個高階事件處理介面，它常常與 javax..microedition.lcdui 套件中的 Command 類別一起使用，因為 Command 類別是我們處理高階事件的時候最常使用的物件。

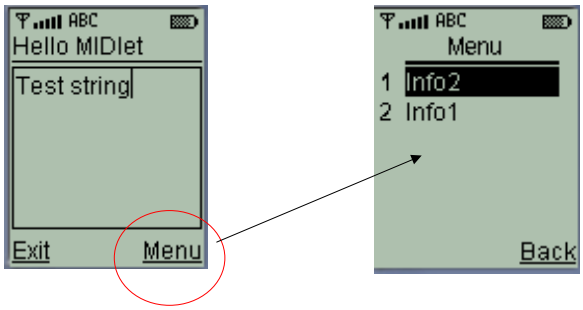
請把 Command 類別想像成我們在一般應用程式之中所使用的系統選單之中的選項（MenuItem）即可。請看以下範例程式：

HLEventMIDlet.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class HLEventMIDlet extends MIDlet implements
CommandListener
{
    private Command exitCommand;
    private Command info1Command;
    private Command info2Command;
    private Display display;

    public HLEventMIDlet()
    {
        display = Display.getDisplay(this);
        exitCommand = new Command("Exit", Command.SCREEN,1);
        info1Command = new Command("Info1", Command.SCREEN,2);
        info2Command = new Command("Info2", Command.SCREEN,2);
    }
    public void startApp()
    {
        TextBox t = new TextBox("Hello MIDlet", "Test string",
256, 0);
        t.setCommandListener(this);
        t.addCommand(exitCommand);
        t.addCommand(info2Command);
        t.addCommand(info1Command);
        display.setCurrent(t);
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    public void commandAction(Command c, Displayable s)
    {
        if (c == exitCommand)
        {
            notifyDestroyed();
        }else if (c == info1Command)
```

```
        {
            notifyDestroyed();
        }
    }
}
```

【執行結果】



在我們的程式中，我們利用 Command 物件生出了三個系統選單選項，它們分別叫做：exitCommand、info1Command、info2Command。Command 的建構式有三個參數。第一個是它們顯示在畫面上的名稱，在本範例中分別為 Exit、Info1、Info2。第二個參數是命令型態，我們都選用 Command.SCREEN 作為參數型態，意思就是說這些參數都是應用程式自訂的選單選項。第三個參數是優先權，我們分別設為 1、2、2，號碼越低代表優先權越高。

產生系統選單選項之後，我們要將它們加入畫面中，於是我們利用 TextBox 從 Displayable 這個抽象類別繼承下來的 addCommand() 函式將這些選項加入畫面之中。這樣一來就會出現您在模擬器上所看到的畫面。可是，手機上的螢幕那麼小，最多只能容納兩個選單選項，那麼，為何 Exit 出現在主畫面中，而 Info1 與 Info2 出現在

Menu 這個系統子選單之中呢？其實這是有一套規格可循的。規則如下：

1. 先比較每個 Command 物件的命令型態，優先順序越高的越先出現。

   在 Command 物件之中共定義了八種命令型態，從優先順序最高到最低列表如下：

   Command.BACK

   Command.CANCEL

   Command.EXIT

   Command.HELP

   Command.ITEM

   Command.OK

   Command.SCREEN

   Command.STOP

2. 如果命令型態相同，就比較其優先權，設定值越低的越先出現。。
3. 如果優先權相同，就以利用 addCommand()加入畫面的先後順序決定。

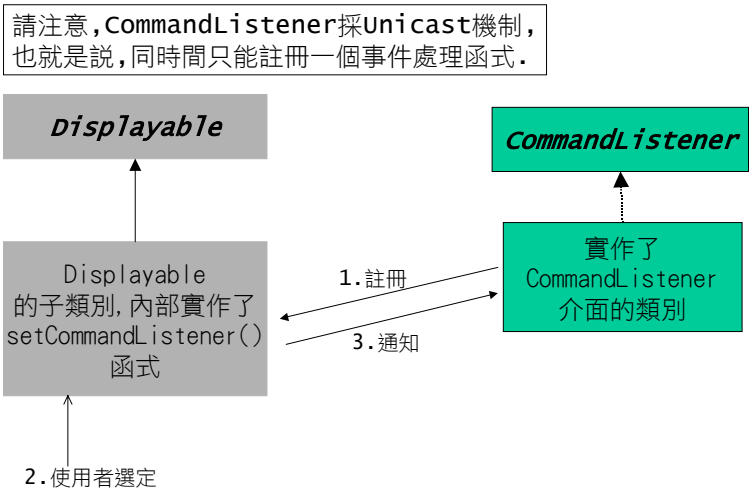
在我們的範例之中，三個選單選項都是 Command.SCREEN 型態，所以就比較三者的優先權，所以您會發現 Exit 出現在主畫面。又因為 Info1 與 Info2 的優先權皆為 2，所以要判斷其加入畫面的先後次序，因此在子選單之中 Info2 比 Info1 還要前面。

當我們將選單選項加入畫面之後，使用者就會利用按鈕點選這些選單選項，這時候我們就要決定這些選項被選定了之後，該做哪些相對應的工作。於是我們利用 TextBox 從 Displayable 這個抽象類別繼承下來的 setCommandListener()函式告訴 TextBox 說，如果這個畫面上如果有任何選單選項被選定，就呼叫實作了 CommandListener 介面的類別之中的 commandAction()函式。這種向事件來源註冊，然後等候通知的事件處理模型在 Java 之中稱為委派模型 (Delegation Model)。

在我們的範例之中，我們使用的程式碼是:

```
t.setCommandListener(this);
```

也就是說 TextBox 的畫面上如果有選單選項被選定，系統就會呼叫 HLEventMIDlet 這個類別（也就是 MIDlet 本身）裡的 commandAction()函式。整體運作方式如下圖所示:



只不過在我們的範例之中，TextBox 放置在 HLEventMIDlet 類別之中，而 HLEventMIDlet 類別又實作了 CommandListener，所以我們才能直接傳遞 this 給 setCommandListener()。

根據上圖，當使用者選定了選單選項之後，就會利用 commandAction()函式告知向它註冊的類別，並將 Command 物件和 Displayable 物件的參考傳遞進來。在我們的範例之中，處理選單選項被選定的事件處理函式如下：

```
public void commandAction(Command c, Displayable s)
{
   if (c == exitCommand)
   {
      notifyDestroyed();
   }else if (c == info1Command)
   {
      notifyDestroyed();
   }
}
```

其中，c 就是被選定的系統選單選項的參考，s 就是發生事件的來源，也就是 TextBox 本身，只不過在此被向上轉型成 Displayable 介面了。在範例之中我們純粹比對 c 是否為之前我們所建立的 Command 物件，根據程式邏輯，如果我們選擇了 Exit 或 Info1，都會關閉程式，如果選擇了 Info2 則不會發生任何事情。

我們也可以利用 Command 物件所提供的 getCommandType()函式取得選單選項的命令型態，或是 getLabel()函式取得該選單選項在螢幕上顯示的文字，或是 getPriority()函式取得其優先權。

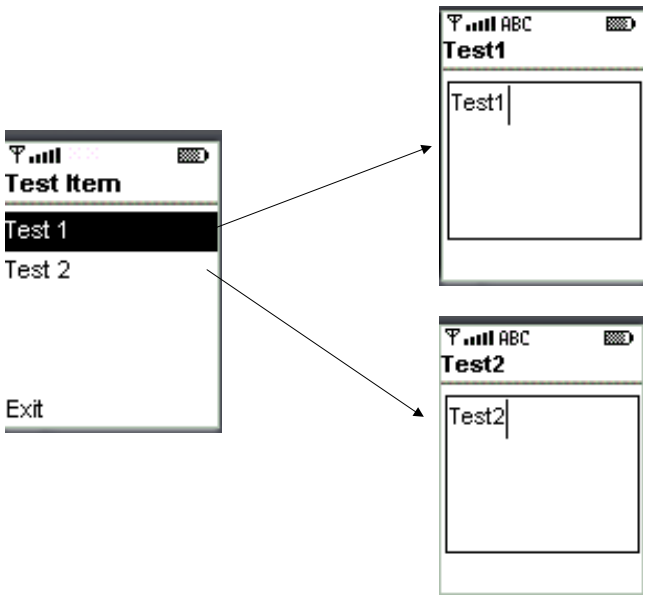
其實 addCommandListener()函式、CommandListener 介面、以及 commandAction()函式三者的關係除了能夠拿來處理系統選單選項之外，也可以拿來處理其他圖形使用者介面元件，如以下範例：

ListEventMIDlet.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class ListEventMIDlet extends MIDlet implements
CommandListener
{
    private Command exitCommand;
    private Display display;
    public ListEventMIDlet()
    {
        display = Display.getDisplay(this);
        exitCommand = new Command("Exit", Command.EXIT,1);
    }
    public void startApp()
    {
        List l = new List("Test Item",Choice.IMPLICIT);
        l.setCommandListener(this);
        l.append("Test 1",null);
        l.append("Test 2",null);
        l.addCommand(exitCommand) ;
        display.setCurrent(l);
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    public void commandAction(Command c, Displayable s)
    {
        if (c == exitCommand)
        {
            notifyDestroyed();
        }else
```

```
        {
            List tmp = (List) s ;
            switch(tmp.getSelectedIndex())
            {
                case 0:
                 display.setCurrent(
new TextBox("Test1","Test1",256,0));
                 break ;
                case 1:
                 display.setCurrent(
new TextBox("Test2","Test2",256,0));
                 break ;
            }
        }
    }
}
```

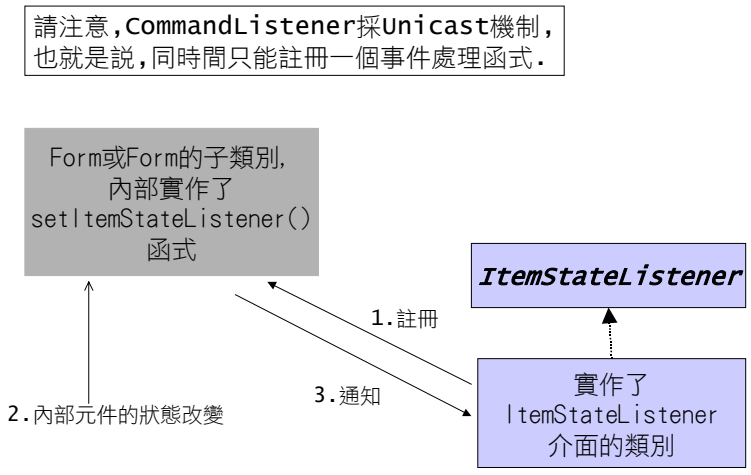
【執行結果】



注　意

當您選擇了 Test 1 或 Test 2 之後，螢幕會切換到 TextBox 所構成的畫面之中，由於我們沒有註冊任何系統選單選項，所以無法離開，請自行加入系統選單選項。或是按下結束程式用的控制鍵來結束程式。

接下來，我們要討論 ItemStateListener 介面。這個介面和 CommandListener 介面的使用方法一樣。當放置於 Form 元件內部的使用者介面元件（Item 物件的子類別，如 Gauge、TextField、DateField 以及 ChoiceGroup）的內部的狀態改變時（一定要和原來的值不同才算改變，如果改變後的值和改變前的值相同，那麼不算狀態改變），該 Form 元件會對所有藉由 setItemStateListener()向它註冊的類別中的 itemStateChanged()函式發出狀態改變的訊息，因為類別因實做了 ItemStateListener 介面，所以一定有 itemStateChanged()函式。整體運作方式如下圖所示：

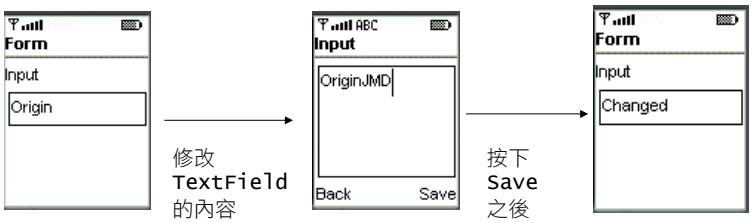


因此，範例程式碼如下：

TFEventMIDlet.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
public class TFEventMIDlet extends MIDlet implements
ItemStateListener
{
    private Display display;
    public TFEventMIDlet()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
        Form f = new Form("Form") ;
        TextField tf = new
        TextField("Input","Origin",25,TextField.ANY) ;
        f.append(tf) ;
        f.setItemStateListener(this) ;
        display.setCurrent(f) ;
    }
    public void pauseApp()
    {
    }

    public void destroyApp(boolean unconditional)
    {
    }
    public void itemStateChanged(Item item)
    {
        TextField tmp = (TextField) item ;
        tmp.setString("Changed") ;
    }
}
```

【執行結果】



注　意

由於我們沒有註冊任何系統選單選項，所以無法離開，請自行加入系統選單選項。或是按下結束程式用的控制鍵來結束程式。

在我們的範例之中，處理狀態改變的事件處理函式如下：

```
public void itemStateChanged(Item item)
{
       TextField tmp = (TextField) item ;
       tmp.setString("Changed") ;
}
```

其中，item 就是狀態改變的那個使用者介面元件的參考。

根據 MIDP1.0 規格書之中說，如果 Form 或其子類別同時要處理 CommandListener 與 ItemStateListener 的話，那麼 itemStateChanged() 會比 commandAction()還要先被呼叫。或者 Form 內部的使用者介面元件被其他項目叫用之前，也會先呼叫 itemStateChanged()。

雖然高階事件處理非常簡單，可是卻讓人有綁手綁腳的感覺，如果要有完全的自由，那麼我們必須處理低階事件才行，因此底下我們將討論低階事件的處理。

低階事件處理

如果您要處理低階事件，或者是要求螢幕繪圖，那麼您就必須使用 Canvas 類別才行。一般要撰寫 Game 的話，我們也會使用 Canvas 類別來製作，除了低階事件提供了處理鍵盤（按鈕）、觸控筆事件的方法之外，也提供了 Game 最需要的圖形顯示功能。

由於 canvas 由 Displayable 繼承而來，所以具有 addCommand() 函式，因此可以和 Command 物件運作，組成高階事件處理機制。與同樣是 Displayable 子類別的 Screen 類別也有相同的特性。但是兩者最主要的差別在於，Screen 類別允許應用程式自己定義許多種 Listener（例如 ItemStateListener），並允許任何實做了該介面的物件向它註冊，但 Canvas 物件卻不允許如此的行為。

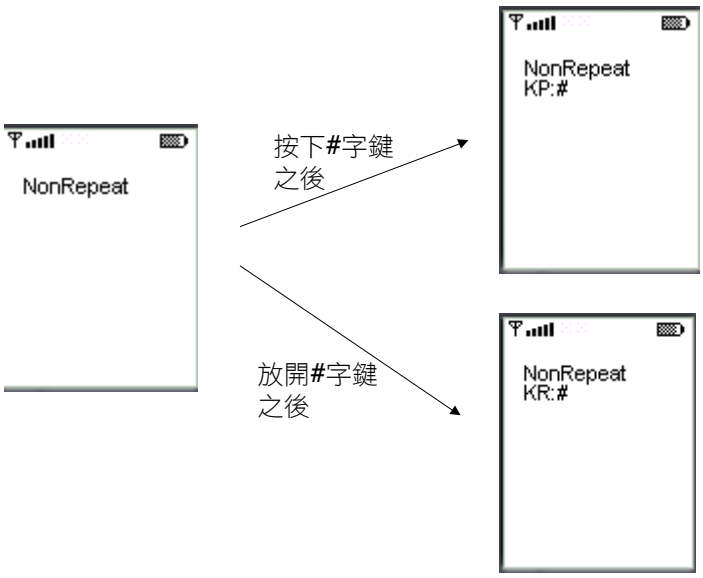
低階事件分成兩種，一種是來自鍵盤（按鈕）的事件，一種是來自觸控筆的事件。我們先討論鍵盤（按鈕）事件的處理。請先看底下範例：

LLEventMIDlet.java

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

public class LLEventMIDlet extends MIDlet
{
    private Display display;

    public LLEventMIDlet()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
     MyCanvas mc = new MyCanvas() ;
```

```
    display.setCurrent(mc) ;
   }
   public void pauseApp()
   {
   }
   public void destroyApp(boolean unconditional)
   {
   }
}

class MyCanvas extends Canvas
{
    String action = "" ;
    public void paint(Graphics g)
    {
        //清除螢幕
        g.setColor(255,255,255) ;
        g.fillRect(0,0,getWidth(),getHeight()) ;
        g.setColor(0,0,0) ;
        //偵測是否支援連打事件
        if(hasRepeatEvents())
        {
            g.drawString("Repeat",10,10,0) ;
        }else
        {
            g.drawString("NonRepeat",10,10,0) ;
        }
        g.drawString(action,10,20,0) ;
    }
    protected void keyPressed(int keyCode)
    {
        action = "KP:" + (char)keyCode ;
        repaint();
    }
    protected void keyReleased(int keyCode)
    {
        action = "KR:" + (char)keyCode ;
        repaint();
    }
    protected void keyRepeated(int keyCode)
    {
        action = "KRe:" + (char)keyCode ;
        repaint();
    }
}
```

【執行結果】



由上述這個範例我們可以知道，當 Canvas 的子類別成為目前正在作用的畫面時，只要按下任何按鈕，會引發 keyPressed()函式，並傳入一個代表該按鈕的整數值，而放開按鈕之後，會引發 keyReleased()函式，並傳入一個代表該按鈕的整數值。在有些機器上，還可以支援連打事件（即一直按著按鈕可以產生連續事件），該事件會引發 keyRepeated()函式，並傳入一個代表該按鈕的整數值，但是這個事件並非每台機器都支援，因此我們必須使用 Canvas 類別之中的 hasRepeatEvents()函式詢問系統是否支援連打事件。

在 MIDP 1.0 規格中，Canvas 類別裡頭定義了幾個常數，他們分別是：KEY_NUM0、KEY_NUM1、KEY_NUM2、KEY_NUM3、KEY_NUM4、

KEY_NUM5 、 KEY_NUM6 、 KEY_NUM7 、 KEY_ NUM8 、 KEY_NUM9 、 KEY_STAR、KEY_POUND 共 11 個，分別代表 0~9 的數字鍵、星號鍵、以及井號鍵。我們可以利用這幾個常數判定鍵盤（按鈕）事件處理函式所傳進來的 keyCode，藉以了解哪個按鈕被按下了，當然，除了這些按鍵之外，其他的按鍵也會有其對應值，您必須自己嘗試找出來，但是為了可以跨平台，建議您僅只使用這些標準的定義鍵。

雖然，根據經驗判斷，如果有個按鍵的排列如下圖：

那麼玩 Game 的時候我們大概可以得知 2、8、4、6 分別代表上、下、左、右，星字鍵可能代表發射、井字鍵可能代表跳躍。可是，並非所有的機器上都會有相同的鍵盤排列方式，也並非一定照我們假設的方式設定按鈕的功能。因此為了 Game 設計師的方便，MIDP 1.0 規格中，Canvas 類別裡頭定義了幾個與 Game 鍵盤代碼相關的常數，他們分別是 UP、DOWN、LEFT、RIGHT、FIRE、GAME_A、GAME_B、GAME_C、GAME_D。這些定義雖然很可能會和之前的定義有所重複，但是因為有了一層抽象性，在移植的時候也就方便多了。

那麼，在程式裡頭該如何處理呢? Canvas 裡頭提供了兩個函式:

1. **getGameAction() →** 傳入 keyCode，函式會回傳所代表的 Game 鍵盤代碼。

   用法如下：

```
public void keyPressed(int keyCode)
{
      switch(getGameAction(keyCode))
      {
          case Canvas.LEFT:
                  moveLeft() ;
                  break ;
          case Canvas.FIRE:
                  fire() ;
                  break ;
          ...略...
      }
}
```

2. **getKeyCode() →** 傳入 Game 鍵盤代碼，函式會回傳所代表的 keyCode。

   用法如下：

```
public void keyPressed(int keyCode)
{
      if(keyCode == getKeyCode(Canvas.LEFT))
      {
          moveLeft() ;
      }else if(keyCode == getKeyCode(Canvas.FIRE))
      {
          fire() ;
      }
      ...略...
}
```

任選任何一種方法，都可以達到跨平台的目的。

接下來，我們要討論觸控筆事件的處理，範例程式如下：

TouchEventMIDlet.java

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

public class TouchEventMIDlet extends MIDlet
{
    private Display display;

    public TouchEventMIDlet()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
     MyCanvas mc = new MyCanvas() ;
     display.setCurrent(mc) ;
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
}

class MyCanvas extends Canvas
{
     String action = "" ;
     public void paint(Graphics g)
     {

          //清除螢幕
          g.setColor(255,255,255) ;
          g.fillRect(0,0,getWidth(),getHeight()) ;
          g.setColor(0,0,0) ;
```

```
        //偵測是否支援連打事件
        if(hasPointerEvents())
        {
            g.drawString("Point",10,10,0) ;
        }else
        {
            g.drawString("NonPoint",10,10,0) ;
        }
        if(hasPointerMotionEvents())
        {
            g.drawString("PointMotion",10,20,0) ;
        }else
        {
            g.drawString("NonPointMotion",10,20,0) ;
        }
        g.drawString(action,10,30,0) ;
    }
    protected void pointerPressed(int x,int y)
    {
        action = "x:" + x + "|" + "y:" + y ;
        repaint();
    }
    protected void pointerReleased(int x,int y)
    {
        action = "x:" + x + "|" + "y:" + y ;
        repaint();
    }
    protected void pointerDragged(int x,int y)
    {
        action = "x:" + x + "|" + "y:" + y ;
        repaint();
    }
}
```

【執行結果】

手機模擬器上的執行結果　　PDA 上的執行結果

由上述這個範例我們可以知道，當 Canvas 的子類別成為目前正在作用的畫面時，只要觸控筆點在觸控螢幕上點選，會引發 pointerPressed()函式，並傳入當時螢幕位置的 X 座標與 Y 座標，而放開觸控筆之後，會引發 pointerReleased()函式，並傳入當時螢幕位置的 X 座標與 Y 座標，如果觸控筆點在觸控螢幕上拖曳，會引發 pointerDragged()函式，並傳入當時螢幕位置的 X 座標與 Y 座標。在有些機器上，還可以支援連打事件（即一直按著按鈕可以產生連續事件），該事件會引發 keyRepeated()函式，並傳入一個代表該按鈕的整數值，但是這個觸控筆事件並非每台機器都支援，因此我們必須使用 Canvas 類別之中的 hasPointerEvents()函式詢問系統是否支援觸控筆事件。同裡，也不是每台機器都支援觸控筆拖曳事件，我們必須使用 Canvas 類別之中的 hasPointerMotionEvents()函式詢問系統是否支援觸控筆拖曳事件。

其他回呼函式(Callback method)

在講述事件處理的開頭，我們曾經提到，在 MIDlet 之中會發生四種事件，其中高階事件和低階事件我們已經完整地討論過了，剩下的兩種:

1. 繪圖事件處理 ➞ Canvas 類別的 paint()方法會在螢幕需要重繪時被呼叫，並傳入一個 Graphics 物件的參考。
2. 呼叫 Display 類別的 callSerially()方法時會引發繼承 Runnable 介面之類別的 run()方法被呼叫。

我們將在往後的章節為大家深入說明。

總　結

我們終於把最重要的高階事件處理和低階事件處理討論結束，有了本章的基礎，我們將可以繼續深入討論圖形使用者介面的深入議題。喘口氣，再接著挑戰 MIDP 的圖形使用者介面程式設計。