10

MIDP 圖形使用者介面程式設計

相對論的意思不是說凡事都相對，而是說明光的速度是絕對。

渾屯的意思也不是說什麼東西都亂成一團，而是說明亂中有序。

事物的名稱常常會誤導了事物的本質。

前 言

和桌上型電腦比較起來，行動通訊裝置不論在記憶體、操作介面、顯示螢幕、以及電力上都有很明顯的劣勢。在開發 Java 程式的時候，只要想到使用者介面，就會很自然地想到 AWT 或 SWING。但是您將會發現，開發 MIDlet 時所使用的使用者介面元件並非這兩者，而是使用專門針對行動通訊裝置所設計的使用者介面元件。之所以針對行動通訊裝置重新設計使用者介面元件，原因在於：

1. AWT 或 SWING 的針對桌上型電腦的硬體條件做了最佳化。
2. AWT 或 SWING 是針對使用滑鼠作為輸入方式的裝置而設計。但是一般的行動通訊裝置並沒有滑鼠，舉例來說，一般的 PDA 都只有觸控螢幕和簡單的按鈕，而行動電話通常都只有一組按鈕而已。
3. AWT 或 SWING 支援視窗管理的功能，比方說 Layout Manager 的設計。雖然 Layout Manager 可以讓我們的使用者介面元件容易移植，而且在視窗改變大小或是被其他視窗覆蓋時，螢幕上可以有較佳的顯示效果。可是對行動通訊裝置來說，連螢幕上要出現重疊的視窗，或是改變上視窗的大小都不太可能了。
4. AWT 或 SWING 所採用的訊息處理機制會在程式執行的時候產生許多的 Event 物件，這些物件通常是動態產生，而且數量很多，生命週期又短。暫時物件的數量太多，會讓虛擬機器的垃圾蒐集工作變得很沉重，尤其在記憶體和處理器能力有

限的行動通訊裝置上，會造成更沉重的負擔。

因此，為了撰寫出人見仁愛的 MIDlet，我們必須深入了解 MIDP 1.0 規格中所提供的各種圖形使用者介面元件。

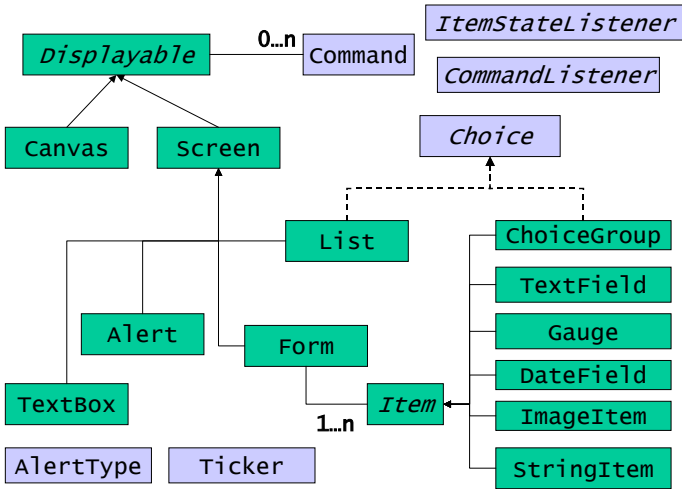
MIDP 使用者介面類別函式庫 ▼

在談到 MIDlet 事件處理的時候，我們曾經說過 MIDlet 的事件處理分為低階事件處理與高階事件處理。根據這兩種不同階層的事件處理機制，我們可以將 MIDP 所提供的使用者介面類別函式庫區分成低階類別函式庫和高階類別函式庫。

如果我們使用高階類別函式庫，就可以讓 MIDlet 所呈現的操作介面和該裝置上一般應用程式所使用的操作介面相同，讓使用者便於操作，但是，我們對於高階圖形使用者介面元件所能做的控制有限，無法隨心所欲地調整他們。也因為如此，使用高階類別函式庫會有較佳的移植性，高階圖形使用者介面元件都是繼承自 Screen 類別。

如果我們使用的是低階類別函式庫，那麼我們可以與裝置上所發生的低階事件互動，也能夠取得螢幕的主控權，做任何自己想做的事情。所以如果一不小心，使用了低階類別函式庫的 MIDlet 在移植上會有一些問題，比方像我們處理鍵盤（按鈕）事件的時候，如果是用了非標準定義的按鈕對應值，在移植上就會有問題，因此在處理上必須小心。低階圖形使用者介面元件都是繼承自 Canvas 類別，並大量地使用 Graphics 類別的圖形處理能力。

MIDP 1.0 所提供的圖形使用者介面類別函式庫都屬於 javax..microedition.lcdui 這個 package 裡頭，其內部圖形使用者介面相關之類別所構成的繼承體系如下圖所示：



從上圖我們可以看出，Displayable 這個抽象類別的子類別可以區分成兩大類：Canvas 與 Screen，其中 Canvas 屬於低圖形使用者介面元件，Screen 屬於高階圖形使用者介面元件。在同一時間，只能有唯一一個 Canvas 或 Screen 類別的子類別出現在螢幕上。

Screen 類別有四個子類別，分別是 Alert、List、TextBox 以及 Form。這四個子類別可以區分為兩類：

1. 封裝了較複雜使用者介面的類別。這類高階圖形使用者介面元件有著事先定義好的固定結構，只能單純地拿來使用，對於其內部的組成結構無法作修改。Alert、List、TextBox 屬於這類型的高階圖形使用者介面元件。

2. 預設沒有任何使用者介面的元件。這類的高階圖形使用者介面元件像是一個容器（Container），可以容納 Item 類別的子類別加入其中，以構成更複雜的圖形使用者介面。只有當事先定義好的高階圖形使用者介面元件不夠用的時候，才會轉而使用這類的高階圖形使用者介面元件。Form 屬於這類型的高階圖形使用者介面元件。

本章中我們只討論高階類別函式庫所提供的元件，低階類別函式庫的相關細節我們會在下一章作深入探討。

List ▼

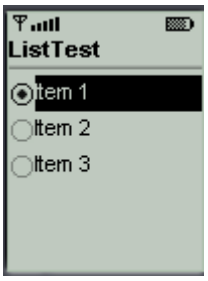
ListTest.java

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

public class ListTest extends MIDlet
{
   private Display display;
   public ListMIDlet()
   {
      display = Display.getDisplay(this);
   }
   public void startApp()
   {
      List l = new List("ListTest",Choice.EXCLUSIVE) ;
      //List l = new List("ListTest",Choice.IMPLICIT) ;
      //List l = new List("ListTest",Choice.MULTIPLE) ;
      l.append("Item 1",null) ;
      l.append("Item 2",null) ;
      l.append("Item 3",null) ;
      display.setCurrent(l) ;
   }
```

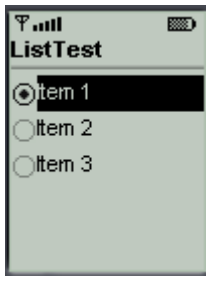
```
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
}
```

【執行結果】

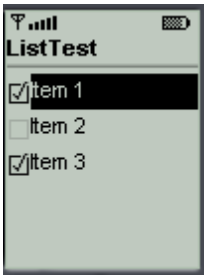
建構式之中使用
Choice.EXCLUSIVE 的結果



建構式之中使用
Choice.IMPLICIT 的結果



建構式之中使用
Choice.MULTIPLE 的結果



從本範例可以看出 Choice.EXCLUSIVE 會造出 Radio Button 的效果（即單選選單），而 Choice.MULTIPLE 會造出 Check Box 的效果

（即多選選選單）。Choice.IMPLICIT 也會造出 Radio Button 的效果，但是它與 Choice.EXCLUSIVE 的差別在於，一旦被 Choice.IMPLICIT 型態的 List 內的選項被選擇之後，它會立刻通知使用 setCommandListener()函式註冊的類別，並呼叫其 commandAction()函式，並在第一個參數傳入 List.SELECT_COMMAND 這個常數。如果您使用的是 Choice.EXCLUSIVE 或 Choice.MULTIPLE，就無法像 Choice.IMPLICIT 型態一般立刻處理事件。

我們可以隨時利用 List 的 append()函式將選項加入 List 之中，append()函式的第一個參數是顯示在螢幕上的文字，第二個則是代表該選項的圖示，如果您不需要圖示，可以將第二個參數設為 null。任何時候我們不需要任何一個選項的時候，可以利用 List 的 delete()刪除特定選項，其參數是欲刪除選項的索引值，請注意，List 之中選項的索引值是以 0 開始。

三種不同的 List 型態分別有各自的處理方式，底下我們一一介紹。

Choice.IMPLICIT 型態的 List 的處理方式如以下範例：

IMPListTest.java

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

public class IMPListTest extends MIDlet implements
CommandListener
{
   private Display display;
   public IMPListTest MIDlet()
   {
```

```
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
        List l = new List("ListTest",Choice.IMPLICIT) ;
        l.append("Item 1",null) ;
        l.append("Item 2",null) ;
        l.append("Item 3",null) ;
        l.setCommandListener(this) ;
        display.setCurrent(l) ;
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }

    public void commandAction(Command c,Displayable s)
    {
        if(c == List.SELECT_COMMAND)
        {
            List tmp = (List) s ;
            int selected = tmp.getSelectedIndex() ;
            System.out.println("Item " + selected + "selected") ;
        }
    }
}
```

前面我們曾說過，Choice.IMPLICIT 型態的 List 會在使用者選擇之後立刻引發事件，並將 List.SELECT_COMMAND 藉由 commandAction()函式的第一個參數傳入。藉由判別 commandAction()函式的第一個參數是否為 List.SELECT_COMMAND，我們可以知道事件是否為 List 所引發。因為這種類型的 List 同時間只有一個選項會被選擇，所以我們利用 List 的 getSelectedIndex()幫我們判定是哪一個選項被選擇。

Choice.EXCLUSIVE 型態的 List 的處理方式如以下範例：

EXCListTest.java

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

public class EXCListTest extends MIDlet implements
CommandListener
{
   private Display display;
   Command commit ;
   public EXCListTest ()
   {
      display = Display.getDisplay(this);
   }
   public void startApp()
   {
     commit = new Command("Commit",Command.SCREEN,1) ;
     List l = new List("ListTest",Choice.EXCLUSIVE) ;
     l.append("Item 1",null) ;
     l.append("Item 2",null) ;
     l.append("Item 3",null) ;
     l.addCommand(commit) ;
     l.setCommandListener(this) ;
     display.setCurrent(l) ;
   }
   public void pauseApp()
   {
   }
   public void destroyApp(boolean unconditional)
   {
   }

   public void commandAction(Command c,Displayable s)
   {
      if(c == commit)
      {
          List tmp = (List) s ;
          int selected = tmp.getSelectedIndex() ;
          System.out.println("Item " + selected + "selected") ;
      }
   }
}
```

Choice.EXCLUSIVE 型態的 List 並不會在使用者選擇之後立刻引發事件，所以通常我們只會藉由系統選單選項來幫助我們。在本範例之中，我們我們利用名為 commit 的系統選單選項，當此選項被使用者選擇之後，因為此類型的 List 同時間只有一個選項會被選擇，所以我們利用 List 的 getSelectedIndex()幫我們判定是哪一個選項被選擇。

Choice.MULTIPLE 型態的 List 的處理方式如以下範例：

MULListTest.java

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

public class HelloMIDlet extends MIDlet implements
CommandListener
{
    private Display display;
    Command commit ;
    public HelloMIDlet()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
      commit = new Command("Commit",Command.SCREEN,1) ;
      List l = new List("ListTest",Choice.MULTIPLE) ;
      l.append("Item 1",null) ;
      l.append("Item 2",null) ;
      l.append("Item 3",null) ;
      l.addCommand(commit) ;
      l.setCommandListener(this) ;
      display.setCurrent(l) ;
    }
    public void pauseApp()
    {
```

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

    public void commandAction(Command c,Displayable s)
    {
      if(c == commit)
      {
           List tmp = (List) s ;
           int num = tmp.size() ;
           for(int i = 0 ; i < num ; i++)
           {
                if(tmp.isSelected(i))
                {
                   System.out.println("Item " + i + "selected") ;
                }
           }
      }
    }
}
```

Choice.MULTIPLE 型態的 List 並不會在使用者選擇之後立刻引發事件，所以通常我們只會藉由系統選單選項來幫助我們。在本範例之中，我們我們利用名為 commit 的系統選單選項，當此選項被使用者選擇之後，因為此類型的 List 同時間可能有好幾個選項會被選擇，所以我們必須先利用 List 的 size()函式告訴我們 List 之中有多少選項，然後再利用 List 的 isSelected()函式幫我們判定該選項是否被選擇。

Alert 與 AlertType

Alert 是一個比較特殊的螢幕型物件（Screen 類別的子類別），當我們利用 Display 類別的 setCurrent()函式將它設為目前顯示在螢幕上的畫面時，它會先發出一段聲音，然後將自己秀在螢幕上，過一段時間之後，它會自動跳回之前的畫面。

因為 Alert 這種與眾不同的特性，所以基本上我們可以把它看做是一般視窗系統上所使用的對話方塊。一般我們印象中的對話方塊會一直顯示在螢幕上，等待使用者按下確定之後，它才會回到原先的畫面。雖然 Alert 在秀出之後隔一段時間之後就會跳回原處，但是我們仍可以利用 Alert 類別的 setTimeout()函式，並傳入 Alert.FOREVER 作為參數，就可以讓 Alert 有類似對話方塊的特性。

請注意，在利用 Display 類別的 setCurrent()函式將它設為目前顯示在螢幕上的畫面時，系統本身一定要存在一個畫面，這樣才能讓 Alert 有地方可以跳回。因此，如果我們在 MIDlet 一啟動直接就將 Alert 設為第一個顯示在螢幕上的畫面的話，會發生錯誤訊息，請特別注意。Alert 的用法如下：

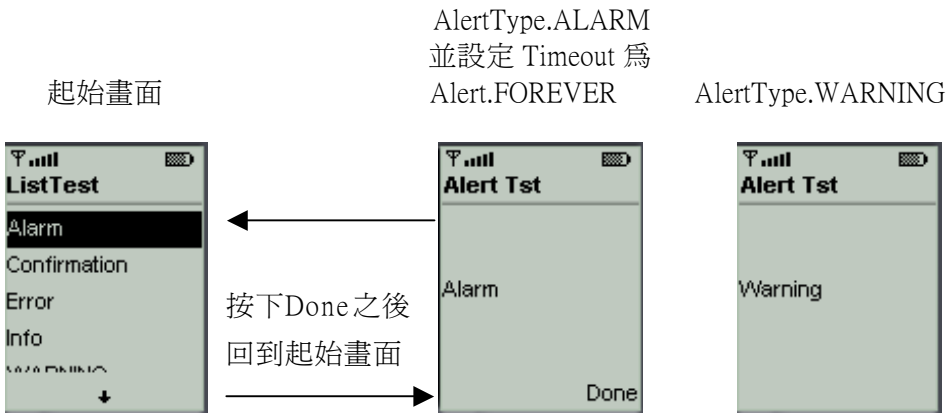
AlertTest.java

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


public class AlertTest extends MIDlet implements CommandListener
{
    private Display display;
    Command commit ;
```

```
public AlertTest()
{
    display = Display.getDisplay(this);
}
public void startApp()
{
  List l = new List("ListTest",Choice.IMPLICIT) ;
  l.append("Alarm",null) ;
  l.append("Confirmation",null) ;
  l.append("Error",null) ;
  l.append("Info",null) ;
  l.append("WARNING",null) ;
  l.setCommandListener(this) ;
  display.setCurrent(l) ;
}
public void pauseApp()
{
}
public void destroyApp(boolean unconditional)
{
}

public void commandAction(Command c,Displayable s)
{
 if(c == List.SELECT_COMMAND)
 {
     Alert al = new Alert("Alert Tst") ;
     List tmp = (List) s ;
     switch(tmp.getSelectedIndex())
     {
          case 0 :
               al.setType(AlertType.ALARM) ;
               al.setString("Alarm") ;
               al.setTimeout(Alert.FOREVER) ;
               display.setCurrent(al) ;
               break ;
          case 1 :
               al.setType(AlertType.CONFIRMATION) ;
               al.setString("Confirmation") ;
               display.setCurrent(al) ;
               break ;
```

```
            case 2 :
                al.setType(AlertType.ERROR) ;
                al.setString("Error") ;
                display.setCurrent(al) ;
                break ;
            case 3 :
                al.setType(AlertType.INFO) ;
                al.setString("Info") ;
                display.setCurrent(al) ;
                break ;
            case 4 :
                al.setType(AlertType.WARNING) ;
                al.setString("Warning") ;
                display.setCurrent(al) ;
                break ;
        }
    }
   }
}
```

【執行結果】



AlertType.CONFIRMATION　AlertType.ERROR　AlertType.INFO

因為 Alert 是螢幕型物件，所以我們可以利用 addCommand()函式在 Alert 裡頭加入系統選單選項，而任何對其事件有興趣的類別也可以利用 addCommandListener()對 Alert 進行註冊動作。

TextBox

TextBox 的作用相當於我們在一般視窗系統之中所使用的 EditBox，不過 TextBox 支援多行輸入，使用範例如下所示：

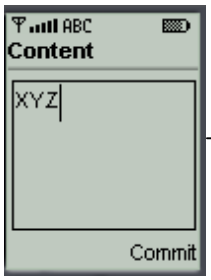
TextBoxTest.java

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

public class TextBoxTest extends MIDlet implements
CommandListener
{
   private Display display;
   Command commit ;
   public TextBoxtest()
   {
      display = Display.getDisplay(this);
   }
   public void startApp()
   {
```

```
      commit = new Command("Commit",Command.SCREEN,1) ;
      TextBox tb = new TextBox("Content","XYZ",6,TextField.ANY) ;
      tb.addCommand(commit) ;
      tb.setCommandListener(this) ;
      display.setCurrent(tb) ;
   }
   public void pauseApp()
   {
   }
   public void destroyApp(boolean unconditional)
   {
   }

   public void commandAction(Command c,Displayable s)
   {
      TextBox tmp = (TextBox)s ;
      System.out.println(tmp.getString()) ;
   }
}
```

【執行結果】

初始畫面



因爲建構式指定最大長度爲 6，所以最多輸入 6 個字

TextBox 的建構式共有四個，第一個是 TextBox 的標題，第二個是 TextBox 的初始內容，第三個是允許輸入字元的最大長度，第四個是型態。

在我們的範例之中，我們將第三個參數設為 6，所以系統只允許我們最多輸入 6 個字。第四個參數我們設為 TextField.ANY，其實還有其他幾種可以選擇，他們的名稱和使用方式如下：

1. TextField.ANY

   允許輸入任何字元或數字。

2. TextField.CONSTRAINT_MASK

   用來和 TextBox 的 getConstraints()函式所傳回的結果做 AND(&)邏輯運算，就可以取得目前的限制設定值。

   請不要在 TextBox 的建構是使用此常數。

3. TextField.EMAILADDR

   允許輸入電子郵件地址，如下圖所示：

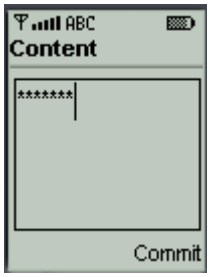
4. TextField.NUMERIC

   只允許輸入數字，如下圖所示：

這可以讓使用者輸入的時候更加方便，不用在數字和英文字母之間切換，而且切換輸入符號的功能也會被除能。

5. TextField.PASSWORD

所有的輸入都會以星號表示，如同輸入密碼一般，如下圖所示：



6. TextField.PHONENUMBER

只允許使用者輸入電話號碼的格式，除了數字之外，還可以輸入星號、加號與井號，如下圖所示：

根據規格書所說，當我們使用 TextField.PHONENUMBER 形式的 TextBox 或 TextField，它會與該裝置上的撥號程式連接，也就是說，通常使用者在此輸入電話之後就可以直接撥通電話。

7. TextField.URL

允許使用者輸入 URL 形式的字串，如下圖所示：

Form 與 StringItem、ImageItem

Form 如果只有單獨出現在螢幕上，其實沒有任何意義，它必須配合 Item 類別的子類別一同運作才有其效果。底下我們介紹 Form 如何與 StringItem、ImageItem 如何一起使用。

StringItem 的作用就如同我們在一般視窗系統之下所認知的 Laebl，其用法如下：

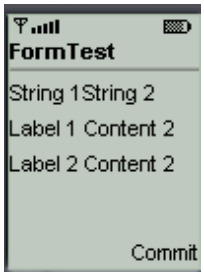
StringItemTest.java

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

public class StringItemTest extends MIDlet implements
CommandListener,ItemStateListener
{
    private Display display;
    Command commit ;
    public StringItemTest ()
    {
       display = Display.getDisplay(this);
    }
    public void startApp()
    {
      commit = new Command("Commit",Command.SCREEN,1) ;
      Form f = new Form("FormTest") ;
      f.append("String 1") ;
      f.append("String 2") ;
      f.append(new StringItem("Label 1 ","Content 2")) ;
      f.append(new StringItem("Label 2 ","Content 2")) ;
      f.addCommand(commit) ;
      f.setCommandListener(this) ;
      f.setItemStateListener(this) ;
      display.setCurrent(f) ;
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }

    public void commandAction(Command c,Displayable s)
    {
Form tmp = (Form) s ;
```

```
        for(int i = 0 ; i < tmp.size() ; i++)
        {
           StringItem si = (StringItem) tmp.get(i) ;
           System.out.println(si.getText()) ;
        }
    }
    public void itemStateChanged(Item item)
    {

    }
}
```

【執行結果】

StrinItem 會自動幫我們做換行的工作



由上述設是我們可以發現，要在 Form 之中加入文字標籤，我們可以使用：

```
append(new StringItem("Label","Text")) ;
```

，也可以使用：

```
append("Text") ;
```

使用單一個字串當作參數的效果等同於呼叫：

```
append(new StringItem(null,"Text")) ;
```

但是不管如何，當我們使用 Form 的 get()函式取出時，一律都是以 StringItem 的形式傳回 (get()的傳回值是 Item 型態，我們必強制轉回成 StringItem)。

ImageItem 的用法與 StringItem 相同。如果我們要在 Form 之中加入圖示，我們可以使用：

```
Append (new ImageItem (標籤文字,Image 物件,位置控制,替代文字)) ;
```

其中位置控制是用來指定圖形顯示時所靠的方向，可以使用的選擇有：

ImageItem.LAYOUT_DEFAULT、ImageItem.LAYOUT_LEFT、ImageItem.LAYOUT_RIGHT、ImageItem.LAYOUT_CENTER、ImageItem.LAYOUT_NEWLINE_BEFORE、ImageItem.LAYOUT_NELINE_AFTER。

而替代文字是在該裝置無法顯示圖片時所用來替代圖片的文字。

我們也可以使用

```
append(new Image()) ;
```

使用 Image 當作參數的效果等同於呼叫：

```
append(new ImageItem(null,Image 物件,ImageItem.LAYOUT_
DEFAULT,null)) ;
```

但是不管用哪個函式插入圖片，當我們使用 Form 的 get()函式取出時，一律都是以 ImageItem 的形式傳回 (get()的傳回值是 Item 型態，我們必強制轉回成 ImageItem)。

Form 與 ChoiceGroup

當 Form 和 ChoiceGroup 一同運作的時候，就如同我們在使用 List 類別一樣，因為 List 和 ChoiceGroup 都實做了 Choice 介面，所以兩者大同小異，很多用法可以回頭參考前面介紹 List 的部分。底下我們介紹 Form 與 ChoiceGroup 如何一起使用。

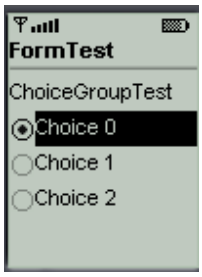
ChoiceGroupTest.java

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

public class ChoiceGroupTest extends MIDlet implements
ItemStateListener
{
    private Display display;
    public ChoiceGroupTest ()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
      Form f = new Form("FormTest") ;
      ChoiceGroup cg =
new ChoiceGroup("ChoiceGroupTest",Choice.EXCLUSIVE);
      cg.append("Choice 0",null) ;
      cg.append("Choice 1",null) ;
      cg.append("Choice 2",null) ;
      f.append(cg) ;
      f.setItemStateListener(this) ;
      display.setCurrent(f) ;
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
```

```
   public void itemStateChanged(Item item)
   {
     ChoiceGroup tmp = (ChoiceGroup)item ;
     System.out.println("Choice " + tmp.getSelectedIndex() +
" selected") ;
   }
}
```

【執行結果】

建構式之中指定型態爲
Choice.EXCLUSIVE



注 意

在前面說明 List 的時候我們曾經說過，List 有三種型態：IMPLICIT、EXCLUSIVE、以及 MULTIPLE。但是在 ChoiceGroup 的建構式中，我們只能使用 EXCLUSIVE 和 MULTIPLE 兩種型態，切莫使用 IMPLICIT 型態，會引發例外情形。

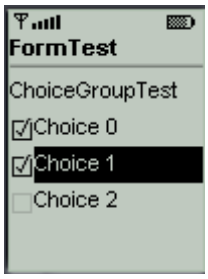
如果我們改用 MULTIPLE 型態的 ChoiceGroup，那麼範例如下所示：

ChoiceGroupTest.java

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

public class ChoiceGroupTest extends MIDlet implements
ItemStateListener
{
   private Display display;
   public ChoiceGroupTest ()
   {
      display = Display.getDisplay(this);
   }
   public void startApp()
   {
     Form f = new Form("FormTest") ;
     ChoiceGroup cg =
new ChoiceGroup("ChoiceGroupTest",Choice.MULTIPLE);
     cg.append("Choice 0",null) ;
     cg.append("Choice 1",null) ;
     cg.append("Choice 2",null) ;
     f.append(cg) ;
     f.setItemStateListener(this) ;
     display.setCurrent(f) ;
   }
   public void pauseApp()
   {
   }
   public void destroyApp(boolean unconditional)
   {
   }

   public void itemStateChanged(Item item)
   {
        ChoiceGroup tmp = (ChoiceGroup)item ;
        for(int i = 0 ; i < tmp.size() ; i ++)
        {
            if(tmp.isSelected(i))
            {
               System.out.println("Choice " + i + " selected") ;
            }
}
   }
}
```

【執行結果】

建構式之中指定型態爲
Choice.MULTIPLE



Form 與 TextField ▼

當 Form 和 TextField 一同運作的時候，就如同我們在使用 TextBox 類別一樣，所以請您參考前面討論 TextBox 的部分。底下我們介紹 Form 與 TextField 如何一起使用。

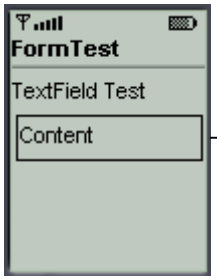
TextFieldTest.java

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

public class TextFieldTest extends MIDlet implements
ItemStateListener
{
    private Display display;
    public TextFieldTest ()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
      Form f = new Form("FormTest") ;
```

```
      TextField tf =
new TextField("TextField Test","Content",10,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 ;
     System.out.println(tmp.getString()) ;
   }
}
```

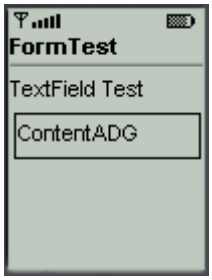
【執行結果】

初始畫面



因爲建構式指定最大長度爲 10，所以最多輸入 10 個字

選擇 Save 之後



TextField 的建構式共有四個，第一個是 TextField 的標題，第二個是 TextField 的初始內容，第三個是允許輸入字元的最大長度，第四個是型態。

在我們的範例之中，我們將第三個參數設為 9，所以系統只允許我們最多輸入 6 個字。第四個參數我們設為 TextField.ANY，其實還有其他幾種可以選擇，請參閱前面討論 TextBox 之處，對於其他幾種型態有詳細的說明。

Form 與 Gauge

Form 如果只有單獨出現在螢幕上，其實沒有任何意義，它必須配合 Item 類別的子類別一同運作才有其效果。底下我們介紹 Form 與 Gauge 如何一起使用。

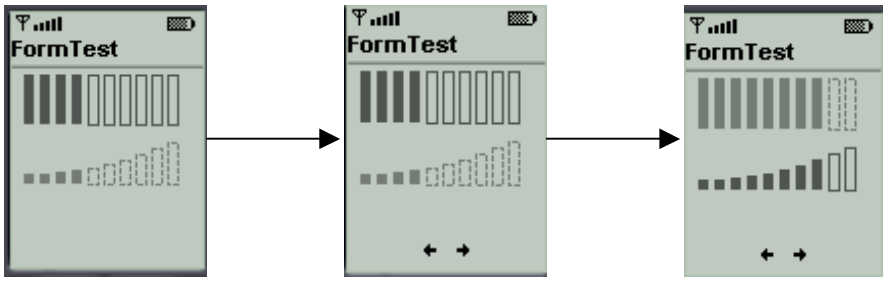
GaugeTest.java

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

public class GaugeTest extends MIDlet implements
ItemStateListener
{
   private Display display;
   Gauge g1 ;
   Gauge g2 ;
   public GaugeTest()
   {
      display = Display.getDisplay(this);
   }
   public void startApp()
   {
     Form f = new Form("FormTest") ;
     g1 = new Gauge("Gauge1",false,100,40);
     g2 = new Gauge("Gauge2",true,100,40);
     f.append(g1) ;
     f.append(g2) ;
     f.setItemStateListener(this) ;
     display.setCurrent(f) ;
   }
   public void pauseApp()
```

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

    public void itemStateChanged(Item item)
    {
       Gauge tmp = (Gauge)item ;
       if(tmp.getLabel().equals("Gauge2"))
       {
            g1.setValue(tmp.getValue()) ;
       }
    }
}
```

【執行結果】

初始畫面　　選擇下面的 Gauge，出現可調整值的箭頭　　調整值



Gauge 的建構式共有四個參數，第一個參數是 Gauge 的標籤名，第二個參數決定它是否可以和使用者互動，以本範例來說，傳入 false 會造成長條狀的 Gauge，我們無法用按鈕改變它的值，必須在程式裡利用 setValue()函式設定其值，反之，如果我們傳入 true，不但可以造成有高低起伏的 Gauge，而且一旦該 Gauge 被選擇之後，還會出現箭頭供我們調整其值。

Form 與 DateField

Form 如果只有單獨出現在螢幕上，其實沒有任何意義，它必須配合 Item 類別的子類別一同運作才有其效果。底下我們介紹 Form 與 DateField 如何一起使用。

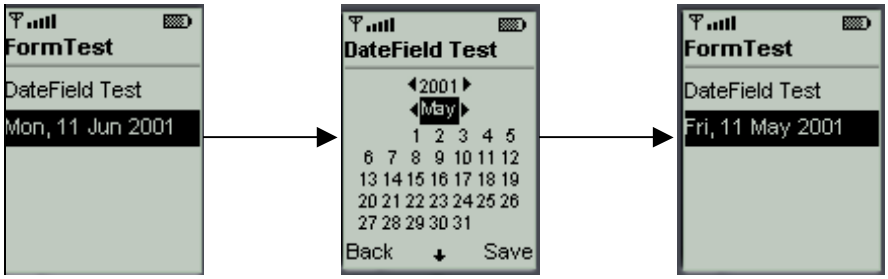
DateFieldTest.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.* ;
public class DateFieldTest extends MIDlet implements
ItemStateListener
{
    private Display display;
    public DateFieldTest ()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
      Form f = new Form("FormTest") ;
      Date now = new Date() ;
      DateField df =
new DateField("DateField Test",DateField.DATE_TIME) ;
      df.setDate(now) ;
      f.append(df) ;
      f.setItemStateListener(this) ;
      display.setCurrent(f) ;
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    public void itemStateChanged(Item item)
    {
```

```
      DateField tmp = (DateField)item ;
      Date d = tmp.getDate() ;
      System.out.println(d.getTime()) ;
   }
}
```

【執行結果】

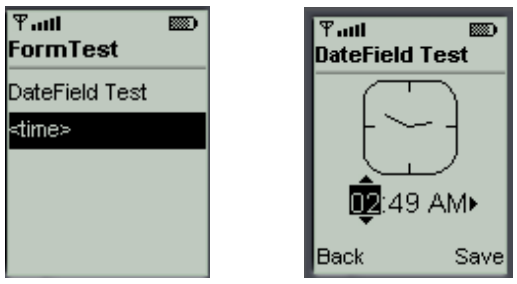
初始畫面　　點選之後　　按下 Save 之後



DateField 的建構式共有三個參數，第一個參數是 DateField 的標籤，第二個是輸入模式，在本範例之中我們是使用 DateField.DATE 形式，另外兩種形式分別是 DateField.TIME 與 DateField.DATE _TIME：

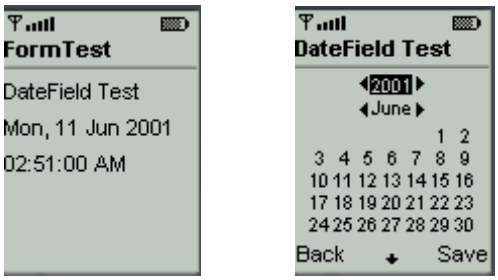
1. DateField.TIME

初始畫面　　設定畫面



2. DateField.DATE_TIME

初始畫面　　設定畫面



任何時候我們都可以利用 DateField 的 setInputMode()函式來改變輸入模式，也可以利用其 getInputMode()取得其輸入模式。藉由呼叫 DateField 的 getDate()函式我們可以取得當時所設定的時間。

Ticker

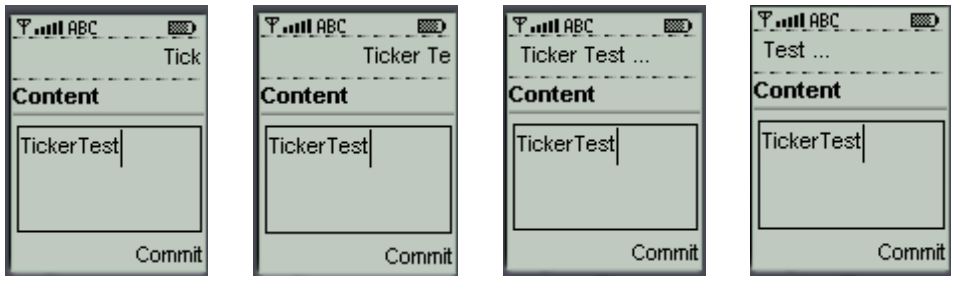
Ticker 是一個類似跑馬燈的類別，除了低階的 Canvas 類別之外，只要是 Screen 類別的子類別都可以加入 Ticker。我們可以利用 Screen 類別中的 setTicker()設定，或者用 getTicker()取出內含的 Ticker 物件。

TickerTest.java

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

public class TickerTest extends MIDlet implements
CommandListener
{
   private Display display;
```

```
    Command commit ;
    public TickerTest()
    {
        display = Display.getDisplay(this);
    }
    public void startApp()
    {
     commit = new Command("Commit",Command.SCREEN,1) ;
     TextBox tb = new
     TextBox("Content","TickerTest",20,TextField.ANY) ;
     tb.setTicker(new Ticker("Ticker Test ...")) ;
     tb.addCommand(commit) ;
     tb.setCommandListener(this) ;
     display.setCurrent(tb) ;
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }

    public void commandAction(Command c,Displayable s)
    {
     TextBox tmp = (TextBox)s ;
     System.out.println(tmp.getString()) ;
    }
}
```

【執行結果】



任何時候我們都可以透過 Ticker 類別的 getString()取得跑馬燈的內容，也可以透過 setString()設定跑馬燈的內容。

總　結

在本章中，我們將所有的高階圖形使用者介面元件做了詳細的討論，相信大家對這些元件的用法都非常熟悉了。

最後提醒各位，即使 MIDP 1.0 提供了如此多的高階圖形使用者介面元件供我們使用，但是當您設計使用者介面的時候請注意底下幾點：

1. 行動通訊裝置不是一般的 PC，它的螢幕大小、記憶體、電力以及運算功能有限，所以請不要設計過於複雜的使用者介面。應該以簡單以及方便使用為主。
2. 如果要讓您的 MIDlet 可以在任何支援 MIDP 的行動通訊裝置上使用，請儘量使用高階圖形使用者介面元件，雖然不同的平台上很可能會有很大的差別，但是至少可以保證能夠執行，並且我們的使用者介面會和該裝置上一般的應用程式在外觀和操作方式上很類似。
3. 在行動通訊裝置，尤其是一般常見的行動電話上，輸入都是透過鍵盤為多，因此在輸入文字（尤其是中文）特別麻煩。我們應該儘量降低讓使用者輸入文字的機會。如果要讓使用者輸入數字的時候，可以儘量利用 TextField.NUMBERIC 讓使用者方便地輸入數字。

4. 請避免使用非標準的高階圖形使用者介面元件，如 kAWT，除了不保證相容之外，由於 kAWT 幾乎是 AWT 的簡化版本，因此在效能上有待商榷。

memo