11

MIDP 圖形處理

所謂真正的藝術，就是允許別人有欣賞的自由。

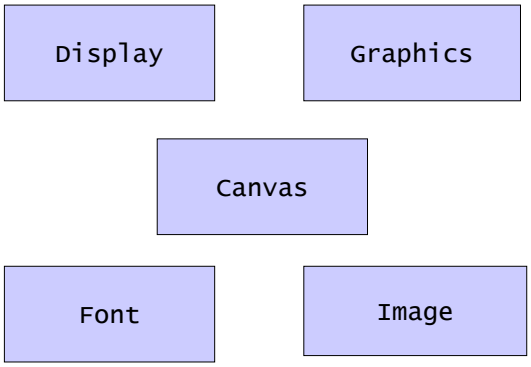
前 言

圖形處理在 MIDP 之中屬於比較低階的技巧。當高階使用者介面元件無法滿足您的需求，而我們又急需其他使用者介面時，就必需動用到低階事件處理，並大量配合圖形處理來完成屬於我們自己的使用者介面。如果您想要撰寫行動通訊裝置上的 Game，就更需要了解圖形處理。

在本章中，我們只概略說明 MIDP 之中與圖形處理相關的議題，讓大家對 MIDP 處理圖形的能力概略的認識。

MIDP 圖形處理類別函式庫

MIDP 1.0 所提供的圖形處理類別函式庫都放在 javax.microedition.lcdui 這個 package 裡頭，其相關類別如下圖所示：



Display 類別 ▼

其中 Display 物件在每個 MIDlet 之中只有一個，屬於用 Singleton Pattern 所包裝起來的類別，我們一定要使用 getDisplay 來取得。Display 就如同一個畫面管理員一般，我們只要呼叫它的 setCurrent()函式，並傳入一個 Displayable 類別的子類別作為參數，就可以設定當時顯示液晶螢幕上的畫面。或者您也可以使用 getCurrent()取得目前正在作用的 Displayable 物件。

Display 也提供了一些函式，可以用來取得該裝置顯示器的相關資訊，比方說：

boolean isColor() → 查詢顯示螢幕是否為彩色，如果是彩色就傳回 true，灰階匯是單色就傳回 false。

int numColors() → 傳回顯示螢幕所支援的色彩數，一般的單色螢幕會傳回 2，即黑色和白色兩種顏色。

在使用圖形處理類別函式庫的時候，請不要假設螢幕大小，因為不同裝置螢幕的大小很可能差異甚大。

從前面的章節中，我們可以得知 Displayable 的子類別共有兩個，一個是 Screen 類別，當我們討論圖形使用者介面的時候已經討論的非常詳細，所以本章所要探討的都是圍繞在其第二個子類別，也就是 Canvas 身上。

Canvas 類別

當 MIDP 所提供的圖形使用者介面元件無法滿足我們的需要時，就是 Canvas 物件上場的時候了。在之前我們曾經說過，當我們使用 Canvas 的時候，除了只能夠和高階的 Command 物件做互動之外，其他所有的低階處理事件都要我們自己一手包辦，請看底下範例：

SimpleCanvas.java

```
import java.io.*;
import javax.microedition.rms.* ;
public class SimpleCanvas extends MIDlet
{
    private Display display;

    public SimpleCanvas()
    {
        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
{
     public void paint(Graphics g)
     {
          g.setColor(255,255,255) ;
          g.fillRect(0,0,getWidth(),getHeight()) ;
```

```
        g.setColor(0,0,0) ;
        g.drawString("Hello",10,10,0) ;
    }
}
```

【執行結果】

從上述範例我們可以看出，每當 Display 的 setCurrent()函式設定了 Canvas 的子類別作為參數的時候，系統會自動呼叫該類別的 paint()函式，並傳入 Graphics 物件當作參數。因此我們必須在 paint()之中一我們的需要寫好程式，以供顯示之用。有關 Graphics 類別的使用，在稍後會有詳細的討論，在此我們先使用它幾個簡單的函式。

一般來說，我們的 paint()函式要做的第一件事情就是清空螢幕，因為系統並不會自動幫我們清掉前面一個畫面，但是 MIDP 並沒有提供任何清除螢幕的函式，所以在本範例之中，我們用

```
g.setColor(255,255,255) ;
```

先將繪圖色設成白色，函後呼叫

```
g.fillRect(0,0,getWidth(),getHeight()) ;
```

將整個螢幕塗成白色。造出所謂的背景色（Background Color）這裡我們用了 Canvas 類別的 getWitch()與 getHeight()取得了螢幕的長度和寬度，因為前面我們曾說過，不同的機器有不同的解析度，所以我們不能在寫程式的時候假設螢幕的長度和寬度固定。

接著我們重新呼叫

```
g.setColor(0,0,0) ;
```

將繪圖色設成白色，然後畫上字串，這時的繪圖色就成了所謂的前景色（Foreground Color）

最後我們再使用

```
g.drawString("Hello",10,10,0) ;
```

將字串畫在螢幕上。

以上就是 Canvas 類別的使用方式，當我們使用它時，我們要考慮的經常只有 paint()函式裡面該如何實作而已。

Image 類別

Image 類別是我們在處理圖形時常常會用的類別，如果根據它的產生方式，我們可以細分成可修改（mutable）和不可修改（immutable）兩種。要辨別一個 Image 物件是可修改還是不可修

改，您可以呼叫 Image 物件的 isMutable()函式得知。我們可以使用 getWidth()與 getHeight()取得該 Image 物件的寬度與高度。

不可修改的 Image 物件

所謂不可修改的 Image 物件，其產生方式是藉由讀取影像檔、從網路接收、或者從 resource bundle 中取得，一旦他們被產生之後，其內容就無法再更改。如果您還有印象的話，您會發現前面章節談到 Alert、ImageItem、以及 Choice、ChoiceGroup 這些元件的時候，他們各自有函式可以接受 Image 物件作為參數，如果您仔細看其說明文件，文件一定會特別告知他們要的是不可修改的 Image 物件。

要產生不可修改的 Image 物件，方法有三種:

1. 從影像檔讀取 ➞ 根據 MIDP 的規定，所有實做 MIDP 的廠商至少要提供讀取 PNG (Portable Network Graphics) 影像檔的功能。有些廠商會支援其他如 GIF 影像檔的功能，但是不建議使用，因為很可能讓您的 MIDlet 無法跨平台。
2. 由 Byte 陣列建立 ➞ 我們可以經由從網路或 resource bundle 裡的文字檔讀入一連串的 byte 陣列，然後用此陣列產生不可修改的 Image 物件。
3. 從其他 Image 物件（可修改或不可修改皆可）來產生。

底下範例中我們將針對第一種方法討論：

ImageEX.java

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

public class ImageEx extends MIDlet implements CommandListener
{
    private Command    cmdExit;
    private ImageCanvas canvas;

    public ImageEx()
    {
        cmdExit = new Command("Exit", Command.SCREEN, 2);
        canvas  = new ImageCanvas();
    }

    public void startApp()
    {
        canvas.addCommand(cmdExit);
        canvas.setCommandListener(this);
        Display.getDisplay(this).setCurrent(canvas);
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean unconditional)
    {
    }


    public void commandAction(Command command, Displayable
screen)
    {

        if (command == cmdExit)
        {
```

```
            notifyDestroyed();
        }
    }
}
ImageCanvas.java
import javax.microedition.lcdui.*;

public class ImageCanvas extends Canvas
{
    public void paint(Graphics g)
    {
      try
      {
         Image image = Image.createImage("/sun.png");
         g.drawImage(image, 0, 0, g.TOP|g.LEFT);
      }catch(Exception e)
      {
         System.out.println(e.getMessage()) ;
      }
    }
}
```

【執行結果】

在此範例之中，我們使用

```
Image image = Image.createImage("/sun.png");
```

從 MIDlet Suite 之中讀取名為 sun.png 的檔案，其內容如為：

然後利用

```
g.drawImage(image, 0, 0, g.TOP|g.LEFT);
```

畫出此 Image 物件。

第二種建立不可修改 Image 物件的方法是利用其函式：

```
createImage(byte[] imagedata, int imageOffset, int
imageLength)
```

第三種方法則是利用函式:

```
createImage(Image source)
```

就可以從既有可修改或不可修改的 Image 物件取得一份不可修改的拷貝。

可修改的 Image 物件

建立一個可修改的 Image 物件非常簡單，只要呼叫 Image 物件的靜態函式。

```
createImage(int width,int height)
```

即可建立一個可修改的 Image 物件。事實上，可修改的 Image 和 Double Buffering 的技術息息相關，可修改的 Image 物件實際上就是一個供人在背景繪圖的 off screen。因此在建立可修改的 Image 物件前，您應該先呼叫 Canvas 類別的 isDoubleBuffered()函式來確定您的機器是否支援 Double Buffering 技術，如果該函式傳回 false，那麼您就不該試圖建立可修改的 Image 物件。

一旦您取得了可修改的 Image 物件，我們就可以呼叫 Image 類別的 getGraphics()取得代表 off screen 的 Graphics 物件，在 off screen 繪圖並不會影響到正常的畫面（on screen)。最後，我們可以利用 on screen（由 paint 函式傳入的 Graphics 物件）的 drawImage()函式繪出可修改 Image 物件的內容。

Font 類別與文字繪製

字型

當我們需要在螢幕上匯出文字時，我們常常需要一些有關字型的相關資料，這時就需要 Font 類別的輔助。通常，我們會使用

```
Font.getDefaultFont()
```

取得代表系統預設所使用字型的 Font 物件。或者您也可以自行使用

```
Font.getFont()
```

來取得代表特定字型的物件。

getFont()共有三個參數，他們分別是外觀（face）、樣式（style)、以及尺寸（size)。他們分別有各種選項：

外觀 → **Graphics.FACE_MONOSPACE**
Graphics.FACE_PROPORTIONAL
Graphics.FACE_SYSTEM

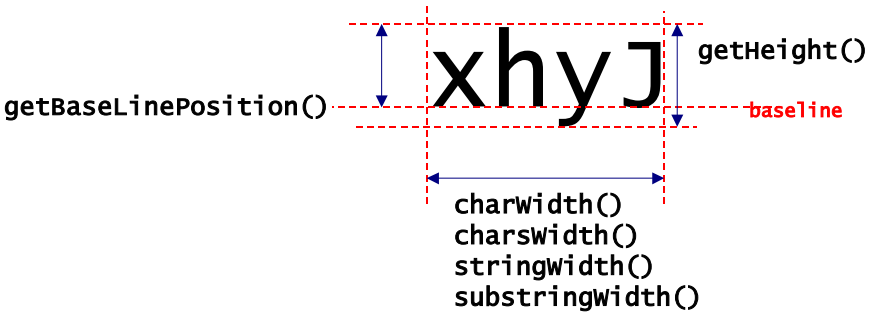
樣式 → **Graphics.STYLE_BOLD**
Graphics.STYLE_ITALIC
Graphics.STYLE_PLAIN
Graphics.STYLE_UNDERLINED

尺寸 → **Graphics.SIZE_LARGE**
Graphics.SIZE_MEDIUM
Graphics.SIZE_SMALL

但是最實際程式之中，我們最常使用的是 Graphics 類別的getFont()函式取得當時顯示在畫面上的螢幕所使用的字型。同理，我們也可以使用 Graphics 類別的 setFont()函式設定所使用的字型。

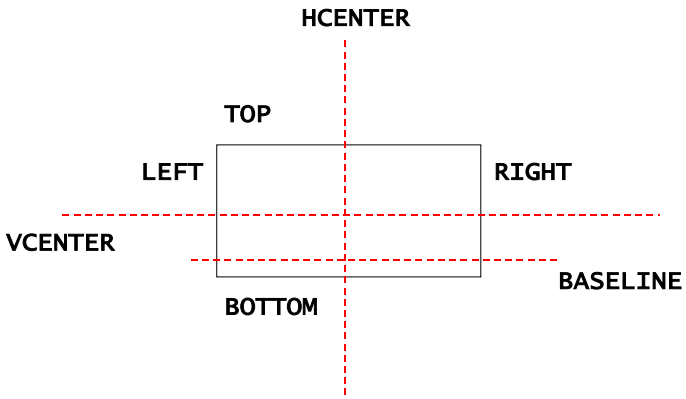
當我們取得代表字型的物件之後，我們就可以利用 getFace()函式取得其外觀、getStyle()取得其樣式、getSize()取得其尺寸。而樣式的部分我們也可以改用 isBold()、isItalic()、isPlain()、isUnderlined()函式來取得相關資訊，這是因為樣式是可以合成的，一個字型的樣式很可能同時是黑體與斜體的組合。

Font 類別除了可以幫我們取得字型的外觀、樣式與尺寸之外，還能幫助我們取得該字型在螢幕上的相關屬性，請參考下圖：



定位點

定位點的定義共有 7 種，分別是 Graphics.TOP、Graphics.BOTTOM、Graphics.LEFT、Graphics.RIGHT、Graphics.HCENTER、Graphics.VCENTER、Graphics.BASELINE。他們對文字和圖形的繪製都具有意義。他們意義如下圖所示：



這幾種定義可以有意義地組合。舉例來說，如果我們選擇使用 TOP 加上 LEFT，則繪製文字時，我們會使用函式：

```
g.drawString("Hello",10,10,Graphics.TOP|Graphics|LEFT) ;
```

繪製圖形時，我們會使用函式：

```
g.drawImage(image, 0, 0, g.TOP|g.LEFT);
```

這時畫面上的結果為：

文字

圖形

注 意

不管是 drawString()或是 drawImage()其第二與第三個三述所指定的座標指的是定位點所參考的起始位址。以上述結果為例，我們指定(0,0)為定位點參考起始位置，然後又選擇的 TOP 與 LEFT 作為定位點，代表(0,0)這個座標為字串或圖形繪製在螢幕上時左上角的點。

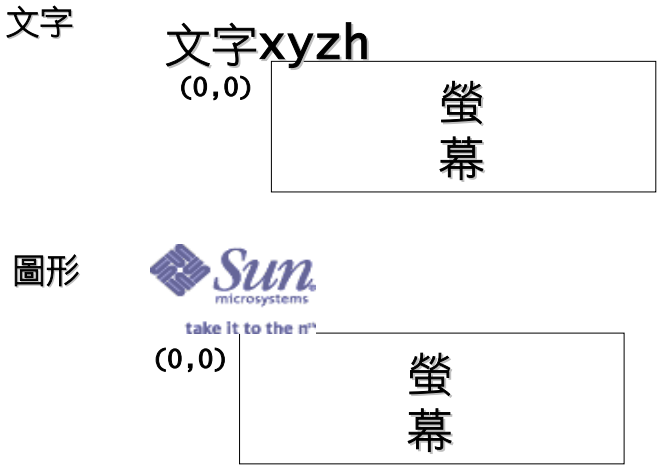
再舉個例子，如果我們選擇使用 BUTTOM 加上 HCENTER，則繪製文字時，我們會使用函式：

```
g.drawString("Hello",10,10,Graphics.BOTTOM|Graphics.HCENTE
R) ;
```

繪製圖形時，我們會使用函式：

```
g.drawImage(image, 0, 0, g.BOTTOM |g.HCENTER);
```

這時畫面上的結果為：



上面函式呼叫的意思是說，(0,0)為字串或圖形繪出在螢幕上時中下方的位置。如果您使用了這樣的函式呼叫，螢幕上將無法看到任何東西，應為都畫到螢幕外了。

由此我們可以歸納出，如果您使用的函式為：

```
g.drawString("Hello",x,y,g.TOP|g.LEFT) ;
```

跟我們使用

```
g.drawString("Hello",x + stringWidth()/2,y + getHeight,g.
BOTTOM|g.HCENTER) ;
```

兩者的意義是相同。

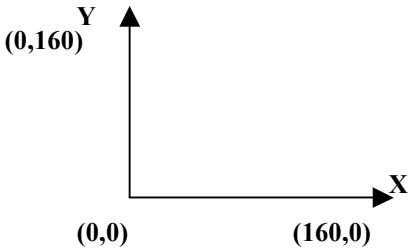
Graphics 類別 ▼

Graphics 物件的取得方式有兩種，第一種是藉由 paint()函式所傳入的參數，這是我們一般最常取得 Graphics 物件的方式。另外一種方式則是利用 Image 類別的 getGraphics()函式來取得。不過他們的不同在於，paint()函式所傳入的是代表當時螢幕的 Graphics 物件，只要取得它，我們就能任意使用其相關功能在螢幕上繪圖。而利用 Image 類別的 getGraphics()函式所取得的 Graphics 物件，是代表在該 Image 上繪圖的 off screen，所以任何對它的呼叫都不會影響到螢幕上的內容。利用 Image 類別的 getGraphics()函式取得 Graphics 物件已經屬於 Double Buffering 技巧的範圍，在本章中我們不予討論。

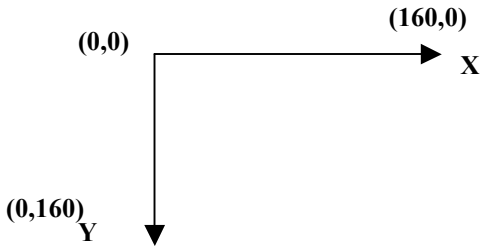
Graphics 物件所使用的座標系統

在使用繪圖函式前，請先注意 MIDP 中 X 座標與 Y 座標的定義方式。

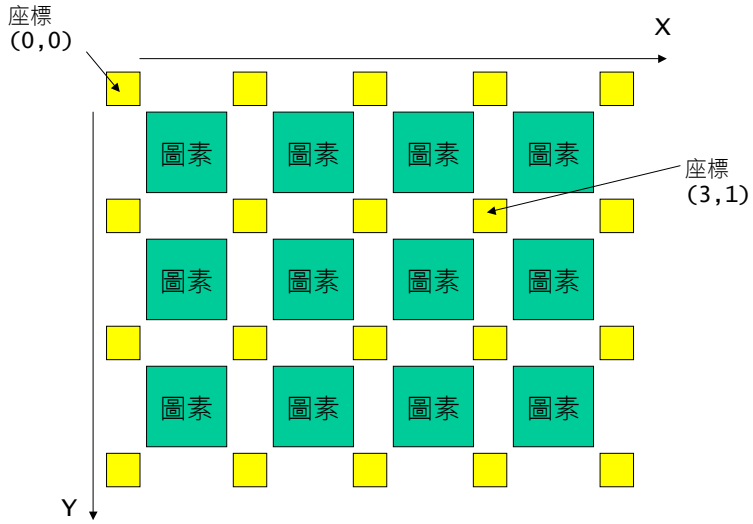
傳統的笛卡耳座標其原點在左下角，向右 X 座標值遞增，向上 Y 座標值遞增。



但是我們在手機的螢幕上做圖時，Y 座標遞減的方向正好相反，變成向右 X 座標值遞增，向下 Y 座標值遞增。



更重要的一點是，我們在所有圖形相關函式之中所使用的座標所代表的並非圖素本身，而是指圖素和圖素之間的空格所構成的座標，如下圖所示：



所以一般我們所說的座標(3, 1)並非指位於(3, 1)這個圖素，而是指圖素(2, 0)、(2, 1)、(3, 0)、(3, 1)所包圍的這個部分。

也正因為座標指的並非圖素本身，所以會造成在繪製圖型和填滿區塊時有所差異，這兩者的不同我們將在稍後詳細說明。

顏色設定

我們可以隨時利用 Graphics 物件的 getColor()取得當時所使用的顏色，其傳回值是一個整數值，內容為 0x00RRGGBB，也就是說，最後第 0~7 位元代表藍色、8~15 代表藍色，16~23 代表紅色。這個整數值的結構也可以當作 Graphics 物件 setColor (int RGB) 函式的參數。

或者，我們可以使用 getBlueComponent()、getGreenComponent()、getRedComponent()分別取得目前顏色設定的藍、綠、紅之設定值。

並使用 Graphics 物件 setColor（int red,int green,int blue）函式，傳入您所希望的藍、綠、紅之顏色設定值來設定顏色。

另外，我們可以使用 getGrayScale()函式取得目前所作用的灰階色階數，使用這個函式的時候請特別注意，如果您已經使用了相對應的 setGrayScale()來設定灰階色階數，那麼 getGrayScale()函式只是單純地傳回設定值。但是如果您沒有設定灰階色階數，那麼呼叫 getGrayScale()函式的時候，會導致系統利用目前作用色的 R、G、B 值來做大量運算，並求得最接近的灰階色階數。

注　意

在 MIDP 之中並沒有提供設定前景色和背景色的函式，因此我們必須自己處理前景色和後景色的問題。

文字和圖形的繪製

有關文字的繪製，是使用 drawChar()、drawChars()以及 drawString()這三個函式來完成。我們已經在前面討論的 Font 類別的部分向大家詳細說明。

而關於圖形的繪製，則是使用 drawImage()函式來完成。這個部分我們也在前面討論 Image 類別的時候做了詳細的介紹。

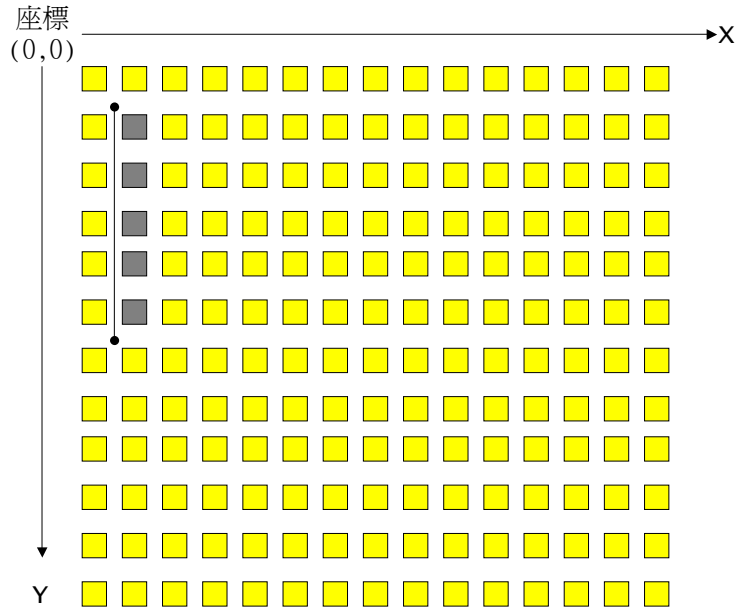
圖形的繪製和畫面填充

◆ 畫線

我們可以使用 Graphics 類別的 drawLine()函式繪製線段。DrawLine 的四個參數分別是起點 X 座標，起點 Y 座標、終點 X 座標、終點 Y 座標。舉例來說，如果我們函式呼叫為：

```
g.drawLine(1,1,1,6)
```

則實際繪製的線段如下圖所示：

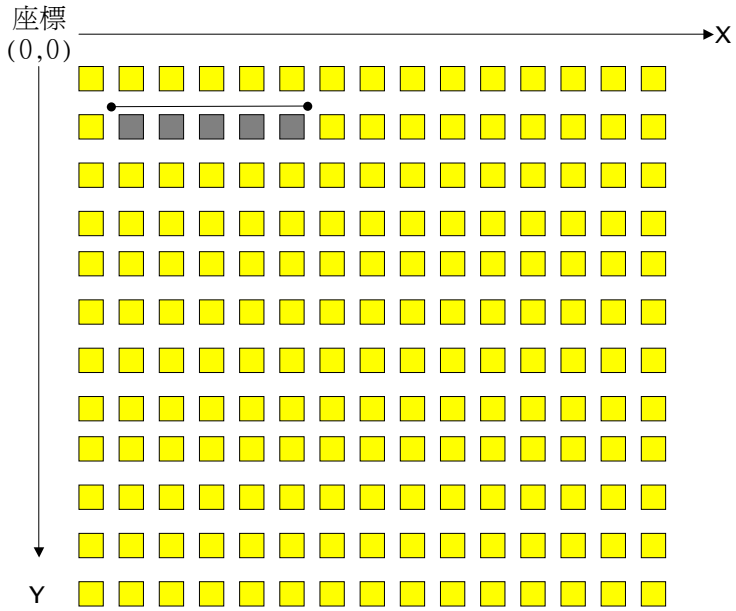


我們可以發現座標右邊的畫素都會被填滿。

如果我們函式呼叫為：

```
g.drawLine(1,1,6,1)
```

則實際繪製的線段如下圖所示：



我們可以發現座標下方的畫素都會被填滿。

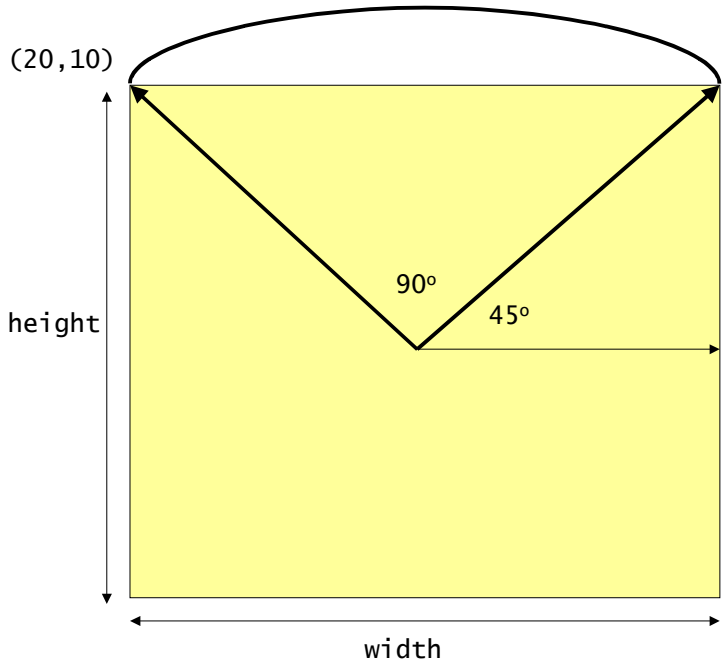
◆ 畫弧形

我們可以使用 Graphics 類別的 drawArc()函式繪製弧形。drawArc 共有 6 個參數，它們分別是:前四個決定弧形所在的矩形範圍，第五個決定起始角度，第六個參數則決定弧形本身所涵蓋的角度。

如果我們函式呼叫為：

```
g.drawArc(20,10,width,height,45,90);
```

則實際繪製的弧形如下圖所示：



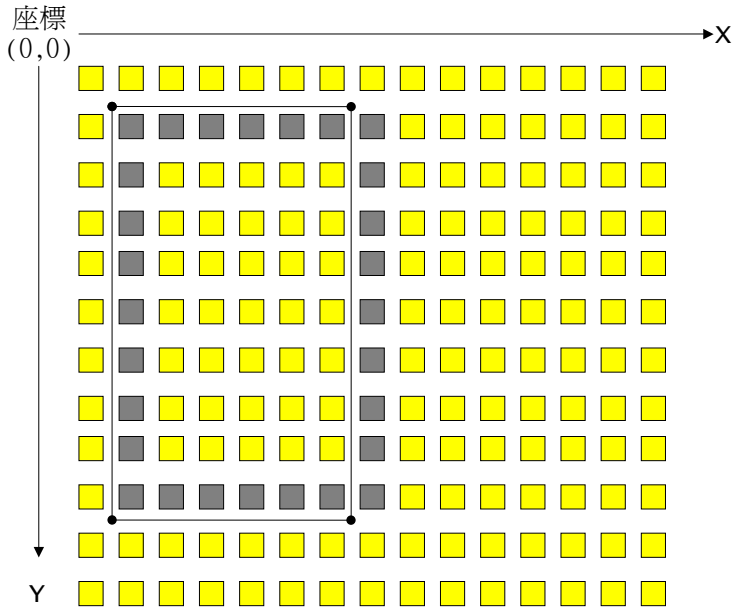
◆ 畫矩形

我們可以使用 Graphics 類別的 drawRect()函式繪製矩形。drawRect 有 4 個參數，分別是起點 X 座標、起點 Y 座標、寬度、長度。

如果我們函式呼叫為：

```
g.drawRect(1,1,6,8)
```

則實際繪製的矩形如下圖所示：



我們可以發現所構成的矩形路徑，其右邊和下方的畫素都被填滿了。

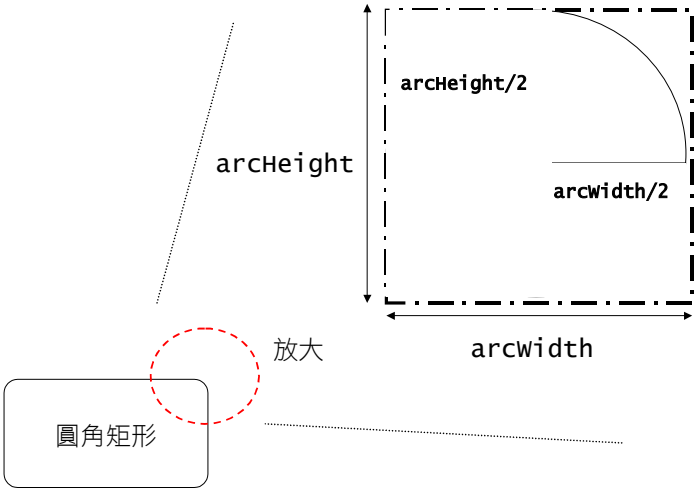
◆ 畫圓角矩形

我們可以使用 Graphics 類別的 drawRoundRect()函式繪製圓角矩形。其實 drawRoundRect()和 drawRect()函式的前四個參數意義相同，唯一的差距只有在最後兩個參數，它們分別是圓角所在矩形的寬度，以及圓角所在舉行的高度。

如果我們函式呼叫為：

```
g.drawRoundRect(1,1,6,8,arcWidth,arcHeight)
```

則實際繪製的圓角矩形，在舉行的部分和使用 drawRect()的結果相同，差別只有在四個直角的樣子不再是直角，而變成圓角。如下圖所示：



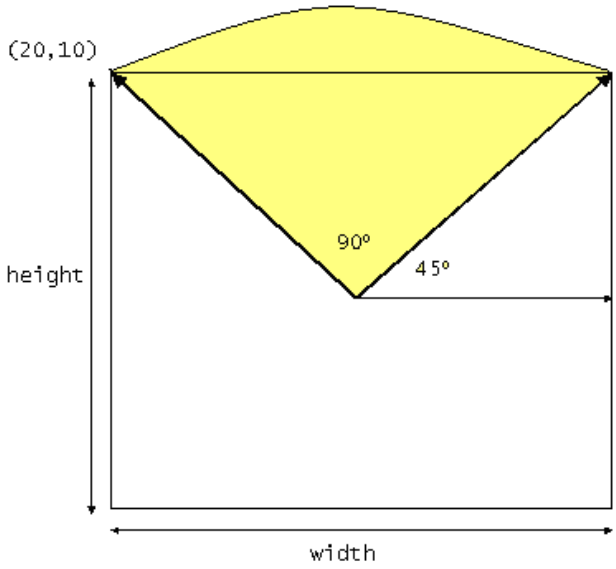
◆ 填充弧形

我們可以使用 Graphics 類別的 fillArc()函式填充弧形。fillArc 共有 6 個參數，它們分別是：前四個決定弧形所在的矩形範圍，第五個決定起始角度，第六個參數則決定弧形本身所涵蓋的角度。

如果我們函式呼叫為：

```
g.fillArc(20,15,width,height,45,90);
```

則實際繪製的填充弧形如下圖所示：



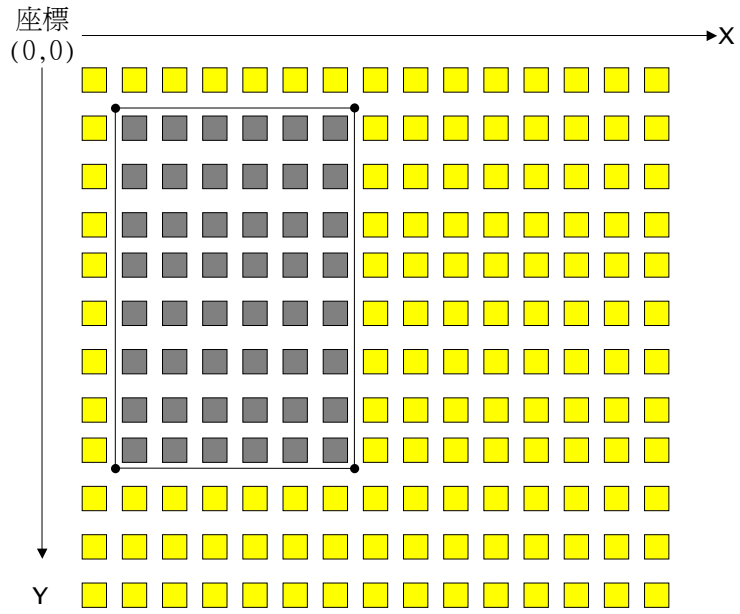
◆ 填充矩形

我們可以使用 Graphics 類別的 fillRect()函式填充矩形。fillRect 有 4 個參數，分別是起點 X 座標、起點 Y 座標、寬度、長度。

如果我們函式呼叫為：

```
g.fillRect(1,1,6,8)
```

則實際繪製的矩形如下圖所示：



我們可以發現只有包含在矩形路經之內的圖素才會被填滿，這和 drawRect()函式的結果有所不同（上下都差一個圖素的大小）。

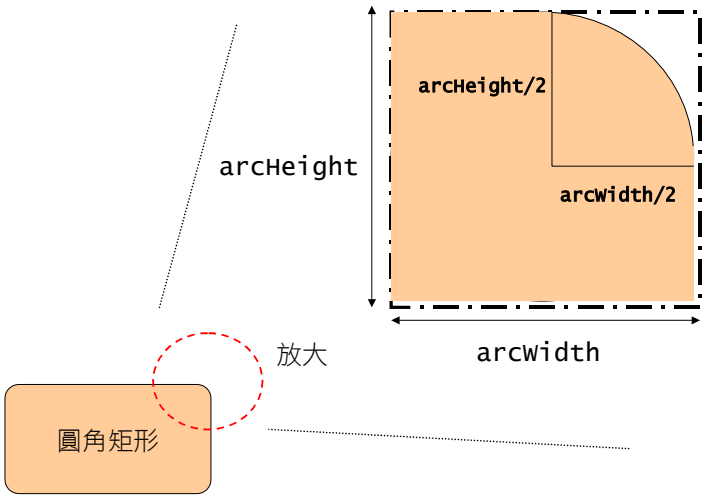
◆ 填充圓角矩形

我們可以使用 Graphics 類別的 fillRoundRect()函式填充圓角矩形。其實 fillRoundRect()和 fillRect()函式的前四個參數意義相同，唯一的差距只有在最後兩個參數，它們分別是圓角所在舉行的寬度，以及圓角所在舉行的高度。

如果我們函式呼叫為：

```
g.fillRoundRect(1,1,6,8,arcWidth,arcHeight)
```

則實際繪製的圓角矩形，在舉行的部分和使用 fillRect()的結果相同，差別只有在四個角的樣子不再是直角，而變成圓角，如下圖所示：



總　結

在本章中，我們為大家介紹了 MIDP 之中所有與繪圖相關的類別，並對這些類別所提供的功能做了使用上的解說。

至於如何巧妙地使用這些基本功能，那又是另外一門學問了，我們將在往後說明手機遊戲設計的時候，再向大家深入解說圖形處理的進階議題。

memo