



第五章

事件

事件程序碼是任何 applet 或使用者圖形界面的核心。圖形應用程式是屬於事件驅動（event-driven）：必須一直等到使用者移動滑鼠、按下按鈕或鍵入值，它們才會有動作。事件驅動程式是一個環繞著其事件程序（event-processing）的結構化模組，因此，確實地了解事件處理的構造對於好的程式設計是非常重要的。

很不幸地，Java 事件處理模組在 Java 1.0 和 Java 1.1 之間有著些許的改變。Java 1.0 模組是屬於簡單型的模組，適合於撰寫基本的 applet。然而，它有一些缺點，且無法配合複雜界面的需求。雖然 Java 1.0 的事件模組是截然不同於 Java 1.1，但是若要在一些以 Java 1.0 為基礎的 Web 瀏覽器上執行的 applet，你仍然需要用到它。Java 1.1 事件模組解決很多 Java 1.0 模組的問題，但是如果它不是供新內部類別特色使用時，仍然會衍生出許多麻煩。本章節涵蓋這兩種版本同時也提供一些範例來加以說明。

Java 1.0 事件模組

在 Java 1.0 中，所有的事件皆是以 `java.awt.Event` 的類別為代表。這個類別有許多描述事件的實體變數，在這些變數中，`id` 是用來指定事件的型態、`Event` 定義一些可能的 `id` 欄位值的常數、`target` 欄位指定產生事件的物件（通常為 `java.awt.Component`）或事件發生在哪一個物件上（即事件的來源）。至於其它欄位的使用與否，則是依據事件的型態而定。例如，當 `id` 是 `BUTTON_EVENT` 時，就會定義 `x` 和 `y` 欄位，但 `id` 若是 `ACTION_EVENT` 時則不定義。`arg` 欄位提供額外的型態相關（`type-dependent`）資料。

Java 1.0 事件首先會被指派（`dispatch`）到它們發生的那個 `Component` 的 `handleEvent()` 方法，這個方法的預設執行程序會先檢查 `Event` 物件的 `id` 欄位，並指派最常使用的事件型態給各種的特定型態（`type-specific`）方法，如表 5-1 所示。

表 5-1: Java 1.0 `Component` 的事件程序方法

<code>action()</code>	<code>lostFocus()</code>	<code>mouseExit()</code>
<code>gotFocus()</code>	<code>mouseDown()</code>	<code>mouseMove()</code>
<code>keyDown()</code>	<code>mouseDrag()</code>	<code>mouseUp()</code>
<code>keyUp()</code>	<code>mouseEnter()</code>	

列在表 5-1 的方法皆是由 `Component` 類別所定義。Java 1.0 事件模組中最主要的特性之一就是你必須改寫這些方法，以便處理事件。這意味著你必須建立一個子類別定義自訂事件處理的行為，舉例來說，當我們撰寫 `applet` 時我們實際上做了什麼事情。然而，請注意並非所有的事件型態皆是由 `handleEvent()` 指派給較特定的方法。例如，如果你對 `LIST_SELECT` 或 `WINDOW_ICONIFY` 事件有興趣的話，那麼你就需要改寫 `handleEvent()`，而非其它較特定的方法。如果你這樣做，通常應該會引發 `super.handleEvent()`，以便繼續以預定方式指派其它型態的事件。

`handleEvent()` 和所有的特定型態方法皆會傳回 `boolean` 值。如果事件處理方法傳回 `false`，它意味著該事件沒有被處理，所以它應該被傳送到目前元件的容器（`container`），由容器來判斷是否想要處理它。反過來說，如果方法是傳回 `true`，即意味著該事件已處理且不再需要去處理了。

容器階層放棄未處理過 (unhandled) 的事件這個事實是很重要的。這意謂著我們可以改寫 applet 內的 `action()` 方法，以便處理位在 applet 內的按鈕所產生的 `ACTION_EVENT` 事件。如果事件並沒有和它們一樣的增加，那我們就需要針對每一個要加入到介面的按鈕，建立一個自訂 `Button` 的子類別！

若程式中有使用到 Java 1.0 的事件模組，那麼通常會在高階的元件中處理事件。例如，在一個 applet 中，你可以改寫你所建立的 `Applet` 子類別的 `handleEvent()`，或一些其它的特定型態的方法。或者，在一個獨立程式中，你可以子類別化 `Frame` 以便提供事件處理方法的定義。當程式顯示一個對話框時，它會子類別化 `Dialog` 來定義方法。至於複雜的介面，因為位於階層頂端上容器的事件處理方法可以會變得冗長以及複雜化，因此當你撰寫它們時需要特別的細心。

元件和它們的事件

在 Java 1.0 模組中，沒有實際上的方法可以得知何種事件的型態是由何種的 GUI 元件所產生，你只能透過文件來查詢該資料。此外，不同的元件使用不同的 `Event` 物件的欄位，並且利用該物件的 `arg` 欄位傳遞不同的值。表 5-2 列出每一個 AWT 元件，和它所產生的事件型態。表格的第一欄是用來指定元件的型態和事件的型態。事件型態就是儲存在 `Event` 的 `id` 欄位上的常數。

從第二欄一直到第六欄是用來指出是否有對所給定的事件設定 `when` (事件發生的時間)、`x` (滑鼠的 `x` 軸座標)、`y` (滑鼠的 `y` 軸座標)、`key` (被按下的鍵盤值) 和 `modifiers` (被按下的輔助鍵) 等欄位。如果在這一欄上出現一個點，代表這個事件會對相對應的欄位設定一個值。第七欄則是用來解釋發生什麼事情時會觸發該事件，和該 `Event` 物件的 `arg` 欄位的值為何。

表 5-2 中，事件元件型態如果是 `Component` 的話，可適用於所有的 `java.awt.Component` 子類別；如果事件的元件型態是 `Windows` 的話，則適用於 `Windows` 的子類別 `Dialog` 和 `Frame`。

表 5-2 : AWT 元件和它們所產生的 Java 1.0 事件

元件 事件型態(id)	when x y key mods	事件意義 參數(型態:數值)
Button ACTION_EVENT		使用者按下按鈕 String: 按鈕的文字標籤
Checkbox ACTION_EVENT		使用者按下檢查盒 Boolean: 新檢查盒狀態
Choice ACTION_EVENT		使用者選擇某個項目 String: 選擇項目的文字標籤
Component Got_FOCUS		取得輸入焦點 (沒有傳遞任何參數)
Component KEY_ACTION		使用者按下某個功能鍵 (沒有傳遞任何參數) key 包含鍵常數
Component KEY_ACTION_RELEASE		使用者放開某個功能鍵 (沒有傳遞任何參數) key 包含 ASCII 鍵值
Component KEY_PRESS		使用者按下某個鍵 (沒有傳遞任何參數) key 包含 ASCII 鍵值
Component KEY_RELEASE		使用者放開某個鍵 (沒有傳遞任何參數) key 包含 ASCII 鍵值
Component LOST_FOCUS		失去輸入焦點 (沒有傳遞任何參數)
Component MOUSE_ENTER		滑鼠進入某個元件 (沒有傳遞任何參數)
Component MOUSE_EXIT		滑鼠離開某個元件 (沒有傳遞任何參數)
Component MOUSE_DOWN		使用者按下滑鼠鍵 (沒有傳遞任何參數)
Component MOUSE_UP		使用者放開滑鼠鍵 (沒有傳遞任何參數)
Component MOUSE_MOVE		使用者移動滑鼠 (沒有傳遞任何參數)

元件 事件型態(id)	when	x	y	key	mods	事件意義 參數(型態:數值)	(續)
Component MOUSE_D R A G						使用者拖曳滑鼠 (沒有傳遞任何參數)	
List A C T I O N _ E V E N T						使用者雙擊某個項目 String: 選取項目的標籤	
List L I S T _ S E L E C T						使用者選取某個項目 Integer: 選取項目的索引值	
List L I S T _ D E S E L E C T						使用者取消某個項目 String: 取消選取項目的索引值	
MenuItem A C T I O N _ E V E N T						使用者選取某個項目 String: 選取項目的標籤	
Scrollbar S C R O L L _ L I N E _ U P						使用者請求捲動 Integer: 捲動的目的位置	
Scrollbar S C R O L L _ L I N E _ D O W N						使用者請求捲動 Integer: 捲動的目的位置	
Scrollbar S C R O L L _ P A G E _ U P						使用者請求捲動 Integer: 捲動的目的位置	
Scrollbar S C R O L L _ P A G E _ D O W N						使用者請求捲動 Integer: 捲動的目的位置	
Scrollbar S C R O L L _ A B S O L U T E						使用者請求捲動 Integer: 捲動的目的位置	
TextField A C T I O N _ E V E N T						使用者按下 < Enter > 鍵 String: 使用者輸入的文字	
Window W I N D O W _ D E S T R O Y						視窗被關閉 (沒有傳遞任何參數)	
Window W I N D O W _ I C O N I F Y						視窗被最小化 (沒有傳遞任何參數)	
Window W I N D O W _ D E I C O N I F Y						視窗從最小化狀態還原至原狀 (沒有傳遞任何參數)	
Window W I N D O W _ M O V E D						視窗被移動 (沒有傳遞任何參數)	

鍵與修飾子常數

Event 類別包含兩個和鍵盤按鍵有關的欄位，其中一個是 key (鍵) 欄位，當鍵盤事件發生時，這個欄位會填入相關的按鍵值；另一個則是 modifiers (修飾子) 欄位，列出目前對鍵和滑鼠事件生效的鍵盤輔助鍵。

四個修飾子常數 (modifier constant) 是由 Event 類別所定義，如表 5-3 所示。你可以利用 Event 方法：shiftDown()、controlDown()、metaDown() 來檢查前三個修飾子所給定的事件。

表 5-3：Java 鍵盤修飾子

修飾子常數	含意
Event.SHIFT_MASK	Shift 鍵被按下 (或 Caps Lock 處於開啟狀態)
Event.CTRL_MASK	Ctrl 鍵被按下
Event.META_MASK	Meta 鍵被按下
Event.ALT_MASK	Alt 鍵被按下

當 KEY_PRESS 或 KEY_RELEASE 事件發生時，它意謂著使用者按下某個鍵，而該鍵可以是一個列印文字、控制文字或非列印文字，並且擁有一個標準的 ASCII 值 - Enter (ASCII 10 或 \n)、Tab (ASCII 9 或 \t)、Esc (ASCII 27)、Backspace (ASCII 8) 或 Delete (ASCII 127)。在這種情況下，事件中的 key 欄位值只是這些鍵被按下或放開的 ASCII 值。

當 KEY_ACTION 或 KEY_ACTION_RELEASE 事件發生時，它意謂著使用者按下了某個沒有 ASCII 表示法的功能鍵。Event 定義了好幾個功能鍵常數，如表 5-4 所示。

表 5-4：Java 功能鍵常數

Key 常數	含意
Event.HOME	Home 鍵
Event.END	End 鍵
Event.PGUP	Page Up 鍵

Key 常數	含意	(續)
Event.PGDN	Page Down 鍵	
Event.UP	Up 箭頭	
Event.DOWN	Down 箭頭	
Event.LEFT	Left 箭頭	
Event.RIGHT	Right 箭頭	
Event.F1 至 Event.F12	功能鍵 F1 到 F12	

滑鼠鍵

為了能夠維持跨平台作業，Java 只認得單一的滑鼠鍵 - Event 事件沒有任何的 mouseButton 欄位可以指出在多鍵滑鼠上的哪一個按鍵被按下。而在支援雙鍵或三鍵的滑鼠上，由右鍵和中間鍵所產生的按下滑鼠鍵、拖曳滑鼠和放開滑鼠鍵等事件，就如同按住輔助鍵（像是 Ctrl、Alt）的效果一樣，如表 5-5 所示。

表 5-5：滑鼠鍵修飾子

滑鼠鍵	Event.modifiers 欄位的旗標設定
左鍵	沒有任何設定
右鍵	Event.META_MASK
中間鍵	Event.ALT_MASK

利用鍵盤修飾子來指出被按下的滑鼠鍵，既可以保持單鍵滑鼠的平台相容性，又允許程式在支援多鍵滑鼠的平台上使用右鍵和中間鍵。舉個來說，假設你想要撰寫一個程式，可以讓使用者透過滑鼠的移動控制，使用兩種不同的顏色畫線，那麼你可以用下列的方式設計程式：如果 modifier 欄位沒有任何設定值的話，使用主要的顏色畫線；若 modifier 欄位有 META_MASK 設定的話，就使用次要的顏色畫線。透過這種方式，如果使用者是使用雙鍵或三鍵的滑鼠時，則使用者可以很容易地利用滑鼠的右鍵和左鍵畫兩種顏色的線修；若是使用單鍵時，則使用者可以搭配滑鼠和 Meta 鍵畫出次要顏色的線條。

在 Java 1.0 中繪圖

範例 5-1 示範一個使用 Java 1.0 事件模組的 applet，它改寫 `mouseDown()` 和 `mouseDrag()` 方法以便能夠讓使用者可以利用滑鼠來繪圖。改寫 `keyDown()` 方法以便當它測試到 C 鍵時可以清除螢幕畫面，並改寫 `action()` 以便當使用者按下 [Clear] 按鈕時可以清除螢幕畫面。我們已經在本書的其它地方看過類似的 applet，因此就不再顯示其結果。

範例 5-1：Scribble1.java

```
import java.applet.*;
import java.awt.*;

/** 這個簡單的 applet 是使用 Java 1.0 的事件處理模組 */
public class Scribble extends Applet {
    private int lastx, lasty;      // 儲存上一次滑鼠的座標
    Button clear_button;          // [Clear] 按鈕
    Graphics g;                   // 用來繪圖的 Graphics 物件

    /** 初始化按鈕和 Graphics 物件 */
    public void init() {
        clear_button = new Button("Clear");
        this.add(clear_button);
        g = this.getGraphics();
    }

    /** 回應滑鼠的單擊 */
    public boolean mouseDown(Event e, int x, int y) {
        lastx = x; lasty = y;
        return true;
    }

    /** 回應滑鼠的拖曳 */
    public boolean mouseDrag(Event e, int x, int y) {
        g.setColor(Color.black);
        g.drawLine(lastx, lasty, x, y);
        lastx = x; lasty = y;
        return true;
    }
}
```

```
/** 回應鍵的按下 **/  
public boolean keyDown(Event e, int key) {  
    if ((e.id == Event.KEY_PRESS && (key == 'c'))) {  
        clear();  
        return true;  
    }  
    else return false;  
}  
/** 回應按下按鈕**/  
public boolean action(Event e, Object arg) {  
    if (e.target == clear_button) {  
        clear();  
        return true;  
    }  
    else return false;  
}  
/** 清除螢幕畫面的方法 **/  
public void clear() {  
    g.setColor(this.getBackground());  
    g.fillRect(0, 0, bounds().width, bound().height);  
}  
}
```

Java 1.0 事件的細節

範例 5-2 的 applet 是一個可以用來處理所有發生在 applet 內的使用者輸入事件，以及顯示事件的細節，主要是滑鼠和鍵盤事件。該程式沒有定義任何的 GUI 元件，所以它沒有處理由那些元件所產生出來的較高階的「語意」(semantic) 事件。

這個範例很有趣，因為它示範了如何解譯修飾子，以及如何了解各種鍵事件的型態。如果你找到你自己所撰寫的複雜事件處理的程式碼，那麼你或許會想要在看完這個範例之後再去模組化它。

範例 5-2: EventTester1.java

```
import java.applet.*;
import java.awt.*;
import java.util.*;

/** 這個 applet 會列出有關 Java 1.0 事件的細節 **/
public class EventTester1 extends Applet {
    // 處理滑鼠事件
    public boolean mouseDown(Event e, int x, int y) {
        showLine(mods(e.modifiers) + "Mouse Down: [" + x + ", " + y + "]);
        return true;
    }
    public boolean mouseUp(Event e, int x, int y) {
        showLine(mods(e.modifiers) + "Mouse Up: [" + x + ", " + y + "]);
        return true;
    }
    public boolean mouseDrag(Event e, int x, int y) {
        showLine(mods(e.modifiers) + "Mouse Drag: [" + x + ", " + y + "]);
        return true;
    }
    public boolean mouseMove(Event e, int x, int y) {
        showLine(mods(e.modifiers) + "Mouse Move: [" + x + ", " + y + "]);
        return true;
    }
    public boolean mouseEnter(Event e, int x, int y) {
        showLine("Mosue Enter: [" + x + ", " + y + "]);
        return true;
    }
    public boolean mouseExit(Event e, int x, int y) {
        showLine("Mosue Exit: [" + x + ", " + y + "]);
        return true;
    }

    // 處理焦點事件
    public boolean gotFocus(Event e, Object what) {
        showLine("Got Focus");
        return true;
    }
}
```

```
}
public boolean lostFocus(event e, Object what) {
    showLine("Lost focus");
    return true;
}

// 處理按下鍵 (key down) 和放開鍵 (key up) 事件
// 鍵事件有兩種型態，因此會較為難懂
public boolean keyDown(Event e, int key) {
    int flags = e.modifiers;
    if (e.id == Event.KEY_PRESS)           // 一般鍵
        showLine("Key Down: " + mods(flags) + key_name(e));
    else if (e.id == Event.KEY_ACTION)     // 功能鍵
        showLine("Function Key Down: " + mods(flags) + function_key_name(key));
    return true;
}

public boolean keyUp(Event e, int key) {
    int flags = e.modifiers;
    if (e.id == Event.KEY_RELEASE)        // 一般鍵
        showLine("Key Up: " + mods(flags) + key_name(e));
    else if (e.id == Event.KEY_ACTION_RELEASE) // 功能鍵
        showLine("Function Key Up: " + mods(flags) + function_key_name(key));
    return true;
}

// 其餘的方法是用來幫助我們整理各種修飾子和鍵

// 傳回目前的輔助鍵清單
private String mods(int flags) {
    String s = "[ ";
    if (flags == 0) return "";
    if ((flags & Event.SHIFT_MASK) != 0) s += "Shift ";
    if ((flags & Event.CTRL_MASK) != 0) s += "Control ";
    if ((flags & Event.META_MASK) != 0) s += "Meta ";
    if ((flags & Event.ALT_MASK) != 0) s += "Alt ";
    s += " ] ";
    return s;
}
```

範例 5-2: EventTester1.java (續)

```
// 傳回一般鍵 (非功能鍵) 的名稱
private String key_name(Event e) {
    char c = (char) e.key;
    if (e.controlDown()) {          // 如果有設定 CTRL 旗標, 就去處理控制字元
        if (c < ' ') {
            c += '@';
            return "^" + c;
        }
    }
    else {                          // 如果沒有設定 CTRL 旗標, 即確認 ASCII 控制
        switch (c) {                // 字元含有特殊的含意
            case '\n': return "Return";
            case '\t': return "Tab";
            case '\033': return "Escape";
            case '\010': return "Backspace";
        }
    }
    // 處理剩下的可能情形
    if (c == '\177') return "Delete";
    else if (c == ' ') return "Space";
    else return String.valueOf(c);
}

// 傳回功能鍵的名稱, 與定義在 Event 類別內的常數作比較
private String function_key_name(int key) {
    switch(key) {
        case Event.HOME: return "Home";   case Event.END: return "End";
        case Event.PGUP: return "Page Up"; case Event.PGDN: return "Page Down";
        case Event.UP: return "Up";       case Event.DOWN: return "Down";
        case Event.LEFT: return "Left";   case Event.RIGHT: return "Right";
        case Event.F1: return "F1";       case Event.F2: return "F2";
        case Event.F3: return "F3";       case Event.F4: return "F4";
        case Event.F5: return "F5";       case Event.F6: return "F6";
        case Event.F7: return "F7";       case Event.F8: return "F8";
        case Event.F9: return "F9";       case Event.F10: return "F10";
        case Event.F11: return "F11";     case Event.F12: return "F12";
    }
}
```

```
    }
    return "Unknown Function Key";
}

/** 顯示在視窗上的線條清單 */
protected Vector lines = new Vector();
/** 新增一個新線條到線條清單內，並重新顯示 */
protected void showLine(String s) {
    if (lines.size() == 20) lines.removeElementAt(0);
    lines.addElement(s);
    repaint();
}
/** 這個方法是用來重畫視窗上的文字 */
public void paint(Graphics g) {
    for(int i = 0; i < lines.size(); i++)
        g.drawString((String)lines.elementAt(i), 20, i*16 + 50);
}
}
```

Java 1.1 事件模組

Java 1.1 事件模組是由 AWT 和 JavaBeans API 兩者所共用。在這個模式之下，不同的事件類別是由不同的 Java 類別所代表，每一個事件皆是 `java.util.EventObject` 的子類別，而 AWT 事件，則是 `java.awt.AWTEvent` 的子類別。為了方便起見，各種不同型態的 AWT 事件皆是放置在新的 `java.awt.event` 包裝內，如 `MouseEvent` 和 `ActionEvent`。

每個事件皆可利用 `getSource()` 取得一個來源物件，且每個 AWT 事件可以利用 `getID()` 取得一個型態值 (type value)，這個值是用來區別由相同事件類別所代表的各種不同的事件型態。例如，`FocusEvent` 有兩種可能的型態：`FocusEvent.FOCUS_GAINED` 和 `FocusEvent.FOCUS_LOST`。事件子類別包括任何有關於特定事件型態的資料值，例如，`MouseEvent` 擁有 `getX()`、`getY()` 和 `getClickCount()` 方法，同時它也繼承來自它處的 `getModifiers()` 和 `getWhen()` 方法。

Java 1.1 事件處理模組是以事件傾聽者 (event listener) 的概念為基礎，物件在接收事件方面，是扮演著一個事件傾聽者的角色。產生事件 (稱為來源事件，event source) 的物件會將當事件發生時希望得到通知的傾聽者列一個清單，並提供方法讓這些傾聽者可以將它們從這份物件名單上移除或新增。當來源事件物件產生一個事件時 (或是當來源事件物件上有使用者輸入事件發生時)，來源事件就會通知傾聽者物件有事件發生了。

來源事件會藉由引發事件傾聽者的方法來通知事件傾聽者，並傳遞一個事件物件 (Event Object 的子類別的實體) 給它。為了使來源得以引發傾聽者的方法，所有的傾聽者都必須實作必需的方法。這是確保藉由要求所有事件的事件傾聽者能對某特定事件型態實作一個相對應的介面。例如，ActionEvent 事件的事件傾聽者必須實作 ActionListener 介面。java.awt.event 包裝為每個它所定義的事件型態定義了一個事件傾聽者介面。(事實上，對 MouseEvent 物件而言，它定義兩個傾聽者的介面：MouseListener 和 MouseMotionListener。) 所有的事件傾聽者介面本身都會衍生 (extend) java.util.EventListener。這個介面並沒有定義任何的方法，而是做為一個標記 (marker) 介面，清楚地以其身份確認所有的事件傾聽者。

一個事件傾聽者介面可能定義超過一個以上的方法。例如，MouseEvent 事件類別就代表好幾個不同的滑鼠事件型態，如按下按鈕事件、放開按鈕事件，以及不同的事件型態能讓對應的事件傾聽者內的不同方法被引發。依照慣例，會傳遞一個引數給事件傾聽者的方法，而該引數就是一種型態的事件物件。這個事件物件應該包括程式對應到事件所需的所有資料。表 5-6 列出定義在 java.awt.event 內的事件型態、相對應的傾聽者、和由每個傾聽者介面所定義的方法。

表 5-6：Java 1.1 事件型態、傾聽者與傾聽者方法

事件類別	傾聽者介面	傾聽者方法
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
ComponentEvent	ComponentListener	componentHidden() componentMoved() componentResized() componentShown()
ContainerEvent	ContainerListener	componentAdded() componentRemoved()

事件類別	傾聽者介面	傾聽者方法	(續)
FocusEvent	FocusListener	focusGained() focusLost()	
ItemEvent	ItemListener	itemStateChanged()	
KeyEvent	KeyListener	keyPressed() keyReleased() keyTyped()	
MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()	
	MouseMotionListener	mouseDragged() mouseMoved()	
TextEvent	TextListener	textValueChanged()	
WindowEvent	WindowListener	windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowIconified() windowOpened()	

針對這些內含一個以上方法的事件傾聽者介面，`java.awt.event` 定義一個簡單的「轉接者」(adapter) 介面，為介面的每個方法提供一個空的主體 (body)。當你只對一個或二個定義的方法感到興趣時，那麼通常子類別化轉接者類別會比實作介面來得容易些。如果你子類別化該轉接者，你只要改寫感興趣的方法即可，但是如果你是直接實作介面，那麼你就必須要定義所有的方法，也就是說你必須提供空的主體給所有不感興趣的方法。這些事先定義的 no-op (不動作) 轉接者類別和它們所實作的介面有相同的名稱，只是將 `Listener` 更改成 `Adapter`：`MouseAdapter`、`WindowAdapter` 等。

一旦你實作一個傾聽者介面，或子類別化一個轉接者類別，你必須將自己的新類別予以實體化 (instantiate)，以便定義一個個別的事件傾聽者物件，然後利用適當的來源事件登錄該傾聽者。以 AWT 程式來說，來源事件總是某種 AWT 元件。事件傾聽者登錄方法是依據標準的命名規則：如果來源事件產生型態為 X 的事件，那

麼它會有一個名為 `addXListener()` 的方法來新增事件傾聽者，和一個名為 `removeXListener()` 的方法來移除傾聽者。Java 1.1 事件模組的其中一項特色就是：很容易就能決定一個元件所能產生的事件型態 - 只需搜尋事件傾聽者登錄方法。例如，藉由檢查 `Button` 物件的 API，你就可以決定它所產生的 `ActionEvent` 事件。表 5-7 列出 AWT 元件和它們所產生的事件。

表 5-7：AWT 元件和它們所產生的 Java 1.1 事件

元件	所產生的事件	含意
<code>Button</code>	<code>ActionEvent</code>	使用者按下按鈕
<code>Checkbox</code>	<code>ItemEvent</code>	使用者點選或取消點選某項目
<code>CheckboxMenuItem</code>	<code>ItemEvent</code>	使用者點選或取消點選某項目
<code>Choice</code>	<code>ItemEvent</code>	使用者點選或取消點選某項目
<code>Component</code>	<code>ComponentEvent</code> <code>FocusEvent</code> <code>KeyEvent</code> <code>MouseEvent</code>	元件移動、改變大小、隱藏、顯示 元件取得或失去焦點 使用者按下或放開某鍵 使用者按下或放開滑鼠鍵、滑鼠進入或離開元件、使用者移動或拖曳滑鼠。注意： <code>MouseEvent</code> 有兩個相對應的傾聽者： <code>MouseListener</code> 和 <code>MouseMotionListener</code> 。
<code>Container</code>	<code>ContainerEvent</code>	對容器新增或移除元件
<code>List</code>	<code>ActionEvent</code> <code>ItemEvent</code>	使用者雙擊列示的項目 使用者點選或取消點選某項目
<code>MenuItem</code>	<code>ActionEvent</code>	使用者點選功能表上的選單
<code>Scrollbar</code>	<code>AdjustmentEvent</code>	使用者移動捲軸列
<code>TextComponent</code>	<code>TextEvent</code>	使用者更改文字內容
<code>TextField</code>	<code>ActionEvent</code>	使用者完成編輯文字
<code>Window</code>	<code>WindowEvent</code>	視窗的開啟、關閉、最小化、還原成原狀、或關閉請求

在 Java 1.1 中繪圖

Java 1.1 事件模組是相當地有彈性，而且接下來你可以利用一些不同的方式來建構你的事件處理程式碼。範例 5-3 將示範第一個技巧。再重述一次，這個範例就是先前的基本 `Scribble` applet，只是這一次我們是使用 Java 1.1 事件模組。該 applet 版

本實作本身的 `MouseListener` 和 `MouseMotionListener` 介面，並利用它本身所擁有的 `addMouseListener()` 和 `addMouseMotionListener()` 方法來登錄。

範例 5-3 : Scribble2.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/** 這個 applet 利用 Java 1.1 事件處理模組 */
public class Scribble2 extends Applet
    implements MouseListener, MouseMotionListener {
    private int last_x, last_y;

    public void init() {
        // 當滑鼠和滑鼠動作事件發生，就告訴這個元件要去通知哪個
        // MouseListener 和 MouseMotionListener 物件。因為我們是
        // 實作本身的介面，所以呼叫本身的方法。
        this.addMouseListener(this);
        this.addMouseMotionListener(this);
    }

    // MouseListener 介面的一個方法。在使用者按下滑鼠鍵時引發。
    public void mousePressed(MouseEvent e) {
        last_x = e.getX();
        last_y = e.getY();
    }

    // MouseMotionListener 介面的一個方法。
    // 在使用者按住滑鼠鍵並拖曳時引發。
    public void mouseDragged(MouseEvent e) {
        Graphics g = this.getGraphics();
        int x = e.getX(), y = e.getY();
        g.drawLine(last_x, last_y, x, y);
        last_x = x; last_y = y;
    }

    // 其它 MouseListener 介面沒有使用的方法
    public void mouseReleased(MouseEvent e) {}
}
```

範例 5-3 : Scribble2.java (續)

```
public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}

// 其它 MouseMotionListener 介面的方法
public void mouseMoved(MouseEvent e) {}
}
```

利用外部類別繪圖

範例 5-4 示範另一個 Scribble applet 的版本，這個版本定義兩個分離式的類別來當作 `MouseListener` 和 `MouseMotionListener` 物件。這些類別定義了事件處理的邏輯，而且 applet 主體縮少成只包含 `init()`，用以建立和登錄傾聽者。注意 `MouseListener` 和 `MouseMotionListener` 類別的用法。將它們予以子類別化會比實作 `MouseListener` 和 `MouseMotionListener` 介面來的容易些。

範例 5-4 : Scribble3.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * 這個 applet 利用外部類別 (external class) 實作 Java 1.1 事件處理模組。
 */
public class Scribble3 extends Applet {
    int last_x;
    int last_y;

    public void init() {
        MouseListener ml = new MyMouseListener(this);
        MouseMotionListener mml = new MyMouseMotionListener(this);
        // 當滑鼠和滑鼠動作事件發生時，告訴這個元件要去通知哪個
        // MouseListener 和 MouseMotionListener 物件。
    }
}
```

```
        this.addMouseListener(ml);
        this.addMouseMotionListener(mml);
    }
}

class MyMouseListener extends MouseAdapter {
    private Scribble3 scribble;
    public MyMouseListener(Scribble3 s) { scribble = s; }
    public void mousePressed(MouseEvent e) {
        scribble.last_x = e.getX();
        scribble.last_y = e.getY();
    }
}

class MyMouseMotionListener extends MouseMotionAdapter {
    private Scribble3 scribble;
    public MyMouseListener(Scribble3 s) { scribble = s; }
    public void mouseDragged(MouseEvent e) {
        Graphics g = scribble.getGraphics();
        int x = e.getX(), y = e.getY();
        g.drawLine(scribble.last_x, scribble.last_y, x, y);
        scribble.last_x = x; scribble.last_y = y;
    }
}
```

利用內部類別繪圖

在我們的 applet 中，範例 5-4 在事件處理問題上並非是一個令人滿意的解法，它將事件處理邏輯放置在分離式類別內 - 該 applet 的外部。這些類別必須被傳遞到一個參考到該 applet 的參照，如此它們才能自其中取用出來。既然我們是使用 Java 1.1 的事件模組，因此有另一種 Java 1.1 新特色的解法：內部類別 (inner class)。範例 5-5 示範當事件傾聽者被以匿名內部類別實作時，這個 applet 的模樣為何。注意這個程式本身非常的簡潔。這大概是使用 Java 1.1 事件模組最普遍的方式，因此你可能會看到很多類似這樣的程式碼。在這種情況下，我們的 applet 只不過是事件處理程式碼，因此幾乎全部的 applet 都變成這兩個內部類別的一部分。

注意我們新增一個特色到這個 applet 內。現在，它包括一個 [Clear] 按鈕，利用這個按鈕登錄 ActionListener 物件；當有適當的事件發生時，它就會清除螢幕上的塗鴉。

範例 5-5 : Scribble4.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * 這個簡單的 applet 是利用匿名內部類別來實作 Java 1.1 事件處理模組。
 */
public class Scribble4 extends Applet {
    int last_x, last_y;

    public void init() {
        // 定義、實體化和登錄一個 MouseListener 物件
        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                last_x = e.getX();
                last_y = e.getY();
            }
        });

        // 定義、實體化和登錄一個 MouseMotionListener 物件
        this.addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                Graphics g = getGraphics();
                int x = e.getX(), y = e.getY();
                g.setColor(Color.black);
                g.drawLine(last_x, last_y, x, y);
                last_x = x; last_y = y;
            }
        });

        // 建立一個清除鈕 [Clear]
```

```
Button b = new Button("Clear");
// 定義、實體化和登錄一個傾聽者以處理按鈕的按下
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {          // 清除螢幕上的繪圖
        Graphics g = getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, getSize().width, getSize().height);
    }
});
// 將這按鈕新增至 applet
this.add(b);
}
```

利用轉接者類別繪圖

範例 5-6 將示範 Scribble 的另一種變革。在這個方式中，某些事件傾聽者物件並沒有處理本身的事件，而只當作傳遞事件給其它物件的「轉接者」。轉接者在 Java 1.1 事件模組中是以兩種方式使用，其中一種用法是描述便利類別（convenience class），例如定義 java.awt.event 內的 MouseAdapter 和 FocusAdapter。然而，在這種方式中，我們是以較一般的意義來使用「轉接者」，引用任何用來接收事件並予以傳遞的小型且簡單的類別，只用來連結 GUI 和其餘的應用程式。轉接者的運作方式就好像是一個實際的轉接器，利用不同種類的連接器將兩條管線連接在一起。

為這個 Scribble.ScribbleActionListener 版本的按鈕處理 ActionEvent 物件的轉接者類別 ScribbleActionListener 被當作一個區域類別實作。

在範例 5-6 中，我們將 Scribble applet 轉換成一個獨立的應用程式，該應用程式提供高階的功能，將它的 GUI 當作一個分離式類別實作。這個用來加強應用程式和介面之間清楚劃分的技巧，是一個很有用的技巧，尤其是對大型的應用程式。

範例 5-6 : Scribble5.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

範例 5-6 : Scribble5.java (續)

```
/** 應用程式類別。處理由 GUI 傳送過來的高階指令 */
public class Scribble5 {
    /** 主要的進入點 (entry point) , 只建立一個這個應用程式類別的實體 */
    public static void main(String[] args) { new Scribble5(); }

    /** 應用程式的建構子 : 建立一個 GUI 類別的實體 */
    public Scribble5() { window = new ScribbleGUI(this); }
    protected Frame window;

    /** 這個應用程式方法是用來處理由 GUI 傳送過來的命令 */
    public void doCommand(String command) {
        if (command.equals("clear")) { // 清除 GUI 視窗
            // 將這個功能放在 GUI 類別內將會更具模組化 (moular) ,
            // 但為了達到示範目的 , 所以我們就將它放在這裡。
            Graphics g = window.getGraphics();
            g.setColor(window.getBackground());
            g.fillRect(0, 0, window.getSize().width, window.getSize().height);
        }
        else if (command.equals("print")) {} // 尚未實作
        else if (command.equals("quit")) {} // 結束應用程式
            window.dispose(); // 關閉 GUI
            System.exit(); // 並離開
        }
    }
}

/** 這個類別為我們的應用程式實作 GUI */
class ScribbleGUI extends Frame {
    int lastx, lasty; // 儲存上一次按下滑鼠的座標位置
    Scribble5 app; // 參照到這個應用程式 , 傳送指令

    /**
     * GUI 建構子執行所有建立 GUI 和設定事件傾聽者的工作。
     * 注意 , 區域 (local) 和匿名 (anonymous) 類別的用法
     */
    public ScribbleGUI(Scribble5 application) {
```

```
super("Scribble"); // 建立視窗
app = application; // 儲存應用程式的參照

// 建立三個按鈕
Button clear = new Button("Clear");
Button print = new Button("Print");
Button quit = new Button("Quit");

// 設立一個 LayoutManager, 並新增按鈕到視窗內
this.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 5));
this.add(clear); this.add(print); this.add(quit);

// 這個區域類別是供按鈕的行動傾聽者 (action listener) 使用
class ScribbleActionListener implements ActionListener {
    private String command;
    public ScribbleActionListener(String cmd) { command = cmd; }
    public void actionPerformed(ActionEvent e) {app.doCommand(command); }
}

// 定義將按鈕連接到應用程式的行動傾聽者轉接者
clear.addActionListener(new ScribbleActionListener("clear"));
print.addActionListener(new ScribbleActionListener("print"));
quit.addActionListener(new ScribbleActionListener("quit"));

// 同樣地, 處理視窗關閉請求
this.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) { app.doCommand("quit"); }
});

// 高階動作事件被傳遞至應用程式, 但是我們仍然在這裡處理塗鴉。
// 登錄一個 MouseListener 物件。
this.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        lastx = e.getX(); lasty = e.getY();
    }
});

// 定義、實體化和登錄一個 MouseMotionListener 物件
```

範例 5-6 : Scribble5.java (續)

```
this.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        Graphics g = getGraphics();
        int x = e.getX(), y = e.getY();
        g.setColor(Color.black);
        g.drawLine(lastx, lasty, x, y);
        lastx = x; lasty = y;
    }
});

// 最後，設定視窗的尺寸，並予以顯示
this.setSize(400, 400);
this.show();
}
}
```

Java 1.1 事件模組的內部

我們在上述章節中，看到的以傾聽者為基礎的事件模組，對於建立一個與事先定義的 AWT 元件無關或與 Java beans 無關的 GUI 而言是很理想的。然而，當發展自訂的 AWT 元件時，它就變得有點麻煩了。AWT 元件（非 beans）對這個事件模組提供一個低階的介面，使其更方便使用。

當 AWTEvent 被傳送到元件時，有些預設的處理事項必須在該事件被分配到適當的事件傾聽者之前繼續執行。當你定義一個自訂的元件（以子類別化的方式）時，在它被傳送到傾聽者物件之前，你還有機會可以改寫方法和截取事件。當 AWTEvent 被傳送到元件時，它會被傳遞給 processEvent() 方法。

預設情況之下，processEvent() 僅檢查事件物件的類別，並分配事件給一個特定類別的方法。例如，如果該事件物件是 FocusEvent 的實體，那麼 processEvent() 會將它分配給一個名為 processFocusEvent() 的方法。或者，如果該事件是型態 ActionEvent 的實體，那麼它就會被分配到 processActionEvent()。換句話說，任何的事件型態 XEvent (X 可代換) 都會被分配到一個相對應的 processXEvent()。MouseEvent 事件則是其中的例外，它是依據所發生的滑鼠事件的型態來決定要被分配給 processMouseEvent() 或 processMouseMotionEvent()。

針對任何給定的元件，是由個別的 `processXEvent()` 負責引發所有登錄事件傾聽者物件的適當方法。例如，`processMouseEvent()` 是對每個登錄的 `MouseListener` 物件引發適當的方法。這些定義在 `java.awt.event` 內的方法和事件傾聽者介面之間，有著一對一的對映 (map)。每個 `processXEvent()` 相當於一個 `XListener` 介面。

就如同你所看到的，在 Java 1.0 事件模組和這個 Java 1.1 低階事件模組之間有一個相似點。`processEvent()` 類似 Java 1.0 的 `handleEvent()` 方法，而 `processKeyEvent()` 類似 Java 1.0 的 `keyDown()` 和 `keyUp()` 方法。至於 Java 1.0 模式，有二個等級 (level) 可以讓你攔截事件：你可以改寫它本身的 `processEvent()`，或者你可以依據預設的 `processEvent()` 版本指派以它們類別為基礎的事件，並取代改寫個別的事件方法，例如 `processFocusEvent()` 和 `processActionEvent()`。

還有一個額外的條件使這個低階的 Java 1.1 事件模組運作。為了對元件接收某特殊型態的事件，你必須告訴元件哪個事件型態是你想要接收的。如果你不這樣做，就效率而言，元件不會強求傳送那個事件型態。若是使用事件傾聽者，則登錄傾聽者的行為就足夠通知你想要接收的那個型態事件的元件。可是若是使用低階的模組，你就必須明確地登錄你想要的事件型態。你可以藉由呼叫該元件的 `enableEvents()` 方法來完成這件事，並傳遞一個位元罩遮 (bit mask) 指定你想要的每一個事件型態。位元罩遮的形成是把數個定義在 `AWTEvent` 類別的 `EVENT_MASK` 常數經由 OR 運算出來的結果。

利用低階的事件處理塗鴉

範例 5-7 是 `Scribble applet` 的另一種變化。這個 applet 是利用 Java 1.1 的低階事件處理模組。它改寫特定事件的方法：`processMouseEvent()`、`processMouseMotionEvent()` 和 `processKeyEvent()`。值得注意的是它如何在它的 `init()` 方法中呼叫 `enableEvents()` 來登錄那個型態感興趣的事件。而且，它呼叫 `requestFocus()` 要求取得鍵盤焦點，如此它就能夠接收到鍵事件。同時也注意它將不感興趣的事件傳遞給超類別事件處理的方法。在這個範例中，超類別將不會去使用那些事件，但這仍然是一個很好的練習。

範例 5-7 : Scribble6.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/** 這個簡單的 applet 是利用 Java 1.1 的低階事件處理 **/
public class Scribble6 extends Applet {
    private int lastx, lasty;

    /** 告訴系統我們感興趣的滑鼠事件、滑鼠動作事件和鍵盤事件 **/
    public void init() {
        this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                          AWTEvent.MOUSE_MOTION_EVENT_MASK |
                          AWTEvent.KEY_EVENT_MASK);
        this.requestFocus(); // 要求取得鍵盤焦點，如此我們才能取得鍵事件
    }

    /** 當某型態的滑鼠事件發生時引發 **/
    public void processMouseEvent(MouseEvent e) {
        if (e.getID() == MouseEvent.MOUSE_PRESSED) { // 檢查事件型態
            lastx = e.getX(); lasty = e.getY();
        }
        else super.processMouseEvent(e); // 將未處理的事件傳遞給超類別
    }

    /** 當滑鼠動作事件發生時引發 **/
    public void processMouseMotionEvent(MouseEvent e) {
        if (e.getID() == MouseEvent.MOUSE_DRAGGED) { // 檢查型態
            int x = e.getX(), y = e.getY();
            Graphics g = this.getGraphics();
            g.drawLine(lastx, lasty, x, y);
            lastx = x; lasty = y;
        }
        else super.processMouseMotionEvent(e);
    }

    /** 呼叫鍵事件：當使用者鍵入 c 時，清除螢幕畫面 **/
```

```
public void processKeyEvent(KeyEvent e) {
    if ((e.getID() == KeyEvent.KEY_TYPED) && (e.getKeyChar() == 'c')) {
        Graphics g = this.getGraphics();
        g.setColor(this.getBackground());
        g.fillRect(0, 0, this.getSize().width, this.getSize().height);
    }
    else super.processKeyEvent(e);        // 將未處理的事件傳遞給超類別
}
}
```

利用 processEvent() 繪圖

範例 5-8 是我們在本章節中的最後一個 Scribble applet 的變化，它非常類似範例 5-7，但它是改寫 processEvent()，而非改寫特定事件的方法。其中必須特別注意的是：它將未處理的事件傳遞給超類別的 processEvent() 方法，如此才能藉由該方法正確無誤的指派它們。這個 applet 版本呼叫 enableEvents() 的方式和之前的版本是一樣的。

範例 5-8 : Scribble7.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event;

/** 這個簡單的 applet 是利用 Java 1.1 的低階事件處理 **/
public class Scribble7 extends Applet {
    private int lastx, lasty;

    /** 指定我們所感興趣的事件型態，並要求取得鍵盤焦點 **/
    public void init() {
        this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
            AWTEvent.MOUSE_MOTION_EVENT_MASK |
            AWTEvent.KEY_EVENT_MASK);
        this.requestFocus();    // 請求鍵盤焦點，如此我們才能取得鍵事件
    }

    /** 事件到達時呼叫。將尚未處理的事件傳遞給超類別 **/
```

範例 5-8 : Scribble7.java (續)

```
public void processEvent(AWTEvent e) {
    MouseEvent me;
    Graphics g;
    switch(e.getID()) {
    case MouseEvent.MOUSE_PRESSED:
        me = (MouseEvent)e;
        lastx = me.getX(); lasty = me.getY();
        break;
    case MouseEvent.MOUSE_DRAGGED:
        me = (MouseEvent)e;
        int x = me.getX(), y = me.getY();
        g = this.getGraphics();
        g.drawLine(lastx, lasty, x, y);
        lastx= x; lasty = y;
        break;
    case KeyEvent.KEY_TYPED:
        if (((KeyEvent)e).getKeyChar() == 'c') {
            g = this.getGraphics();
            g.setColor(this.getBackground());
            g.fillRect(0, 0, this.getSize().width, this.getSize().height);
        }
        else super.processEvent(e);
        break;
    default: super.processEvent(e); break;
    }
}
```

Java 1.1 事件的細節

我們利用一個顯示所有事件細節的程式來結束本章節。範例 5-9 非常類似範例 5-2 的 `apple`，除了它是以低階的 Java 1.1 事件模組來取代 Java 1.0 模組，還有它是以獨立應用程式來取代 `apple`。圖 5-1 為該程式的輸出結果。

因為這個應用程式沒有建立任何的 GUI 元件，它只接收像 MouseEvent、KeyEvent 和 WindowEvent 這類的輸入事件。並沒有接收由 AWT 元件所產生的高階「語意」事件，例如 ActionEvent、ItemEvent 和 TextEvent。然而，它仍是一個很有用的範例，因為它示範了如何利用提供在 MouseEvent、KeyEvent 和其它事件類別內的資訊。

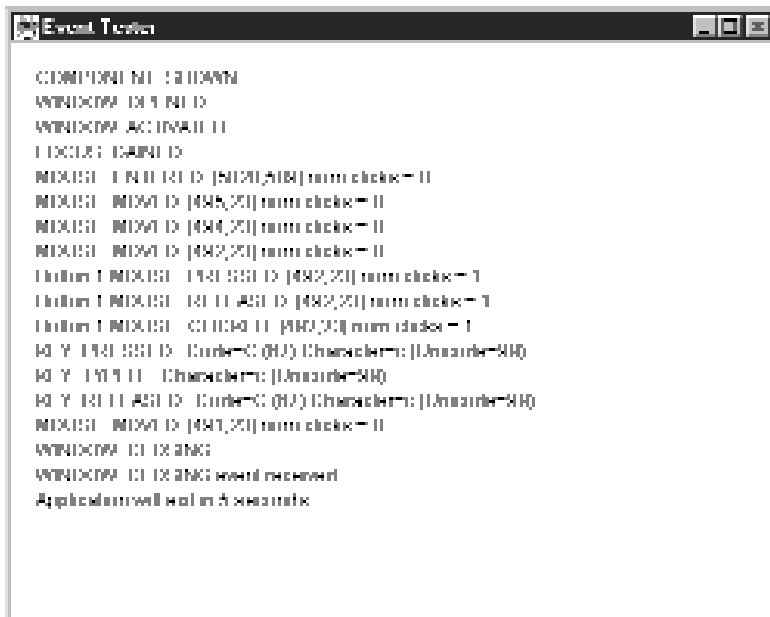


圖 5-1：Java 1.1 事件細節

範例 5-9: EventTester2.java

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/** 這個程式是用來顯示所有發生在其視窗內的事件 */
public class EventTester2 extends Frame
{
    /** main() 方法：建立一個 EventTester 框架，並顯示它 */
  
```

範例 5-9: EventTester2.java (續)

```
public static void main(String[] args) {
    EventTester2 et = new EventTester2();
    et.setSize(500, 400);
    et.show();
}

/** 建構子：登錄我們感興趣的事件型態 */
public EventTester2() {
    super("Event Tester");
    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                     AWTEvent.MOUSE_MOTION_EVENT_MASK |
                     AWTEvent.KEY_EVENT_MASK |
                     AWTEvent.FOCUS_EVENT_MASK |
                     AWTEvent.COMPONENT_EVENT_MASK |
                     AWTEvent.WINDOW_EVENT_MASK);
}

/**
 * 顯示不會引發滑鼠動作的滑鼠事件。
 * 透過下列所定義的 mousemods() 輸出修飾子。
 * 其它的方法會傳回有關滑鼠事件的額外資訊。
 * 利用 showLine() 顯示視窗內的文字行，
 * 它和 paint() 一起定義在這個類別的尾端。
 */
public void processMouseEvent(MouseEvent e) {
    String type = null;
    switch(e.getID()) {
        case MouseEvent.MOUSE_PRESSED:    type = "MOUSE_PRESSED"; break;
        case MouseEvent.MOUSE_RELEASED:   type = "MOUSE_RELEASED"; break;
        case MouseEvent.MOUSE_CLICKED:    type = "MOUSE_CLICKED"; break;
        case MouseEvent.MOUSE_ENTERED:    type = "MOUSE_ENTERED"; break;
        case MouseEvent.MOUSE_EXITED:     type = "MOUSE_EXITED"; break;
    }
    showLine(mousemods(e) + type + " : [" + e.getX() + "," + e.getY() +
            "]" + "num clicks = " + e.getClickCount() +
            (e.isPopupTrigger()? " ; is popup trigger:"));
}
```

```
/**
 * 顯示滑鼠移動和拖曳事件。注意：MouseEvent 是唯一一個能夠
 * 擁有兩個方法、兩個 EventListener 介面和兩個轉接者類別來
 * 處理兩個不同事件種類的事件型態。如同在 init() 所見到的，
 * 滑鼠動作事件必須被請求與其它的滑鼠事件型態分開。
 */
public void processMouseMotionEvent(MouseEvent e) {
    String type = null;
    switch(e.getID()) {
        case MouseEvent.MOUSE_MOVED:    type = "MOUSE_MOVED"; break;
        case MouseEvent.MOUSE_DRAGGED:  type = "MOUSE_MOVED"; break;
    }
    showLine(mousemods(e) + type + ": [" + e.getX() + "," + e.getY() + "] "
        + "num clicks = " + e.getClickCount() + (e.isPopupTrigger()? " "
        + "is popup trigger:"));
}

/**
 * 為 MouseEvent 傳回一個修飾子的字串表示法 (representation)。
 * 注意，在這裡所呼叫的方法皆是繼承自 InputEvent。
 */
protected String mousemods(MouseEvent e) {
    int mods = e.getModifiers();
    String s = "";
    if (e.isShiftDown()) s += "Shift ";
    if (e.isControlDown()) s += "Ctrl ";
    if ((mods & InputEvents.BUTTON1_MASK) != 0) s += "Button 1 ";
    if ((mods & InputEvents.BUTTON2_MASK) != 0) s += "Button 2 ";
    if ((mods & InputEvents.BUTTON3_MASK) != 0) s += "Button 3 ";
    return s;
}

/**
 * 顯示鍵盤的事件。
 * 注意有三種不同的鍵事件型態，
 * 且鍵事件是藉由鍵程式碼或 Unicode 字元輸出 (report)。
 * KEY_PRESSED 和 KEY_RELEASED 事件是針對所有鍵的觸發而產生。
 * 當鍵觸發產生一個 Unicode 字元時，KEY_TYPED 是唯一產生的事件；
 */
```

範例 5-9: EventTester2.java (續)

```
* 這些事件並不會輸出鍵程式碼。
* 如果 isActionKey() 傳回 true, 那麼鍵事件只會輸出一個鍵程式碼,
* 就是因為被按下或放開的鍵沒有相對應的 Unicode 字元。
* 除了透過由 KeyEvent 類別所定義的 VK_constants 來說明鍵程式碼之外,
* 尚可以利用靜態的 getKeyText() 方法將它們轉換成字串,
* 也就是我們在這裡所使用的方式。。
**/
public void processKeyEvent(KeyEvent e) {
    String eventtype, modifiers, code, character;
    switch(e.getID()) {
    case KeyEvent.KEY_PRESSED: eventtype = "KEY_PRESSED"; break;
    case KeyEvent.KEY_RELEASED: eventtype = "KEY_RELEASED"; break;
    case KeyEvent.KEY_TYPED: eventtype = "KEY_TYPED"; break;
    default: eventtype = "UNKNOWN";
    }

    // 將輔助鍵的清單轉換成字串
    modifiers = KeyEvent.getKeyModifiersText(e.getModifiers());

    // 如果有的話, 取得鍵程式碼的字串和數字版本。
    if (e.isActionKey()) character = "";
    else character = "Character=" + e.getKeyChar() + " (Unicode=" +
        ((int)e.getKeyChar() + ")";

    // 顯示全部
    showLine(eventtype + ": " + modifiers + " " + code + " " + character);
}

/**
 * 顯示鍵盤焦點事件。焦點可以永久地取得或失去,
 * 或暫時性從一個元件轉出或轉入 (transfer from/to),
 **/
public void processFocusEvent(FocusEvent e) {
    if (e.getID() == FocusEvent.FOCUS_GAINED)
        showLine("FOCUS_GAINED" + (e.isTemporary()? " (temporary):");
    else
```

```
        showLine("FOCUS_LOST" + (e.isTemporary()? "(temporary)": ""));
    }

    /** 顯示元件事件 **/
    public void processComponentEvent(ComponentEvent e) {
        switch(e.getID()) {
            case ComponentEvent.COMPONENT_MOVED: showLine("COMPONENT_MOVED"); break;
            case ComponentEvent.COMPONENT_RESIZED: showLine("COMPONENT_RESIZED"); break;
            case ComponentEvent.COMPONENT_HIDDEN: showLine("COMPONENT_HIDDEN"); break;
            case ComponentEvent.COMPONENT_SHOW: showLine("COMPONENT_SHOWN"); break;
        }
    }

    /** 顯示視窗事件。注意 WINDOW_CLOSING 的特殊處理 **/
    public void processWindowEvent(WindowEvent e) {
        switch(e.getID()) {
            case WindowEvent.WINDOW_OPENED: showLine("WINDOW_OPENED"); break;
            case WindowEvent.WINDOW_CLOSED: showLine("WINDOW_CLOSED"); break;
            case WindowEvent.WINDOW_CLOSING: showLine("WINDOW_CLOSING"); break;
            case WindowEvent.WINDOW_ICONIFIED: showLine("WINDOW_ICONIFIED"); break;
            case WindowEvent.WINDOW_DEICONIFIED: showLine("WINDOW_DEICONIFIED"); break;
            case WindowEvent.WINDOW_ACTIVATED: showLine("WINDOW_ACTIVATED"); break;
            case WindowEvent.WINDOW_DEACTIVATED: showLine("WINDOW_DEACTIVATED"); break;
        }

        // 如果使用者請求關閉一個視窗，就離開程式。
        // 但是首先顯示一個訊息，強迫它可以被看得見，
        // 並確定使用者有足夠的時間去閱讀它。
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            showLine("WINDOW_CLOSING event received.");
            showLine("Application will exit in 5 seconds");
            update(this.getGraphics());
            try {Thread.sleep(5000);} catch (InterruptedException ie) { ; }
            System.exit(0);
        }
    }

    /** 要顯示在視窗內的行清單 (list of line) **/
    protected Vector lines = new Vector();
```

範例 5-9: EventTester2.java (續)

```
/** 新增一行至行清單內，並重新顯示 */
protected void showLine(String s) {
    if (lines.size() == 20) lines.removeElementAt(0);
    lines.addElement(s);
    repaint();
}

/** 這個方法是用來重畫視窗內的文字 */
public void paint(Graphics g) {
    for(int i = 0; i < lines.size(); i++)
        g.drawString((String)lines.elementAt(i), 20, i*16 + 50);
}
}
```

練習題

- 5-1. 撰寫一個可以回應使用者鍵盤事件的 applet。當使用者按下一個鍵時，它必須在 Web 瀏覽器的狀態列上顯示該鍵。
- 5-2. 撰寫一個可以在它顯示範圍中心顯示一個紅色圓形的 applet。當使用者按下鍵盤上任何的方向鍵時，能夠以很少的像素朝適當的方向移動該圓形。如果使用者在該圓形上方按下滑鼠鍵並拖曳，則根據滑鼠移動方向移動該圓形。使用者可以透過鍵入功能鍵 (F1 到 F10) 來改變該圓形的顏色。
- 5-3. 撰寫一個可以在螢幕上顯示多個圓形的 applet，當使用者在 applet 的背景上按下滑鼠左鍵時，就必須在該滑鼠位置上顯示一個新圓形，當使用者在已存在的圓形上方按下滑鼠左鍵並予以拖曳時，必須能根據滑鼠移動方向移動該圓形。當使用者在圓形上方按下滑鼠右鍵時 (或者是在按下滑鼠右鍵時按下適當的輔助鍵)，從顯示畫面上移除該圓形。為了能完成這項工作，你將必須維護目前所有顯示在畫面上的圓形和其位置清單。利用 `java.util.Vector` 物件來維護這個清單。考慮定義一個巢狀式高階的(即靜態) `Circle` 類別來代表每個圓形的位置，可能的話並畫出每一個圓形。