



第二章

物件、類別與介面

本章範例重點放在 Java 的物件導向本質上，這樣做主要是為了跟《Java in a Nutshell》的第三章相連貫。本章提供一些編寫 Java 程式時所必須了解的物件導向概念與語法。下面幾個段落很快地介紹 Java 物件導向技術。

「物件」(object) 是資料值 (data value) 或「欄位」(field) 的集合，加上用在資料上的「方法」(method)。物件的資料型態稱為「類別」(class)，我們可以說物件是類別的「實例」(instance)。類別負責定出物件內每個欄位的型態 (type)，並且提供方法處理實例內的資料。物件的產生是利用 new 運算子 (operator)，這個運算子會呼叫類別的「建構式」(constructor) 方法把新物件初始化。物件的欄位和方法是透過 . 運算子存取與呼叫的。

處理物件欄位的方法一般都稱為實例方法 (instance method)。實例方法跟我們在第一章 Java 的基本概念 看到的「靜態方法」或「類別方法」不同。在第一章中，類別方法被宣告成 static，它們適用於類別本身，而非類別的個別實例。類別中的欄位也可宣告為 static，使它們成為類別欄位 (class field) 而非實例欄位 (instance field)。雖然每個物件都會有每個實例欄位的副本，可是類別欄位的副本卻只有一份，由類別的所有實例所共用。

類別的欄位與方法都對有不同的可見性 (visibility) 等級，包括公開 (public)、私有 (private) 與保護 (protected)。不同的可見性等級，限制不同背景環境 (context) 下所能用到的欄位與方法。每個類別都會有一個超類別 (super class)，類別會「繼承」超類別上的欄位與方法。繼承其它類別的類別稱為子類別 (subclass)。Java 裡面的類別會構成一個類別階層 (class hierarchy)。java.lang.Object 是這個階層的根 (root)。換句話說，Object 是其它 Java 類別的最上層超類別。

介面 (interface) 是 Java 的一種概念，它和類別一樣可以定義方法，但是不提供任何方法的實作。類別可以提供某個介面中所有方法的實作來「實作」介面。

Rect 類別

範例 2-1 示範一個繪製矩形的類別。Rect 類別的每個實例都會有四個欄位：x1、y1、x2 與 y2，它們負責定義矩形的四個角座標。Rect 類別裡面也定義了一些處理這些座標的方法。

請注意 toString() 方法，它改寫 java.lang.Object (Rect 所隱含的超類別) 的 toString()。toString() 會產生一個代表 Rect 物件的 String。之後，你會發現這個方法對輸出 Rect 的值非常有用。

範例 2-1：Rect.java

```
package com.davidflanagan.examples.classes;
/**
 * 這個類別會顯示一個矩形。它的欄位分別代表矩形的四個角座標。
 * 它的方法定義在 Rect 物件上執行的運算。
 */
public class Rect {
    // 這些是類別的資料欄位
    public int x1, y1, x2, y2;

    /**
     * 這是類別的主要建構式。它用引數設定新物件每個欄位的初始值。
     * 類別通常會跟建構式同名，而且建構式不必宣告回傳值。
     */
}
```

```
    **/  
public Rect(int x1, int y1, int x2, int y2) {  
    this.x1 = x1;  
    this.y1 = y1;  
    this.x2 = x2;  
    this.y2 = y2;  
}  
  
/**  
 * 這是另一個建構式。它的實作部份是靠上面建構式完成的。  
 **/  
public Rect(int width, int height) { this(0,0,width,height); }  
  
/** 另一個建構式 **/  
public Rect() { this(0,0,0,0); }  
  
/** 根據指定值，移動矩形 **/  
public void move(int deltax, int deltay) {  
    x1 += deltax; x2 += deltax;  
    y1 += deltay; y2 += deltay;  
}  
  
/** 檢查指定點是不是在矩形裡面 **/  
public boolean isInside(int x, int y) {  
    return ((x >= x1) && (x <= x2) && (y >= y1) && (y <= y2));  
}  
  
/**  
 * 傳回另一個矩形跟這個矩形之間的聯集 (union)。  
 * 即，傳回涵蓋這兩個矩形的最小矩形。  
 **/  
public Rect union(Rect r) {  
    return new Rect((this.x1 < r.x1) ? this.x1 : r.x1,  
                    (this.y1 < r.y1) ? this.y1 : r.y1,  
                    (this.x2 > r.x2) ? this.x2 : r.x2,  
                    (this.y2 > r.y2) ? this.y2 : r.y2);  
}
```

範例 2-1 : Rect.java (續)

```
/**
 * 傳回另一個矩形跟這個矩形之間的交集 ( intersection )
 * 即，傳回它們的重疊處。
 */
public Rect intersection(Rect r) {
    Rect result = new Rect((this.x1 > r.x1) ? this.x1 : r.x1,
        (this.y1 > r.y1) ? this.y1 : r.y1,
        (this.x2 < r.x2) ? this.x2 : r.x2,
        (this.y2 < r.y2) ? this.y2 : r.y2);
    if (result.x1 > result.x2) { result.x1 = result.x2 = 0; }
    if (result.y1 > result.y2) { result.y1 = result.y2 = 0; }
    return result;
}

/**
 * 這是超類別 Object 的方法。
 * 改寫此方法使 Rect 物件能夠轉成字串、利用 + 運算子連結成字串，
 * 並傳給跟 System.out.println() 類似的方法。
 */
public String toString() {
    return "[" + x1 + "," + y1 + "; " + x2 + "," + y2 + "];";
}
}
```

測試 Rect 類別

範例 2-2 是一個叫做 RectTest 的獨立程式，我們用它測試範例 2-1 的 Rect 類別。請注意產生新 Rect 物件時用的 new 關鍵字與 Rect() 建構式的用法。該程式利用 . 運算子呼叫 Rect 物件的方法並存取它的欄位。當 Rect 利用字串的連結運算子 + 產生秀給使用者看的字串時，這個測試程式會隱含用到 Rect 的 toString() 方法。

範例 2-2 : RectTest.java

```

package com.davidflanagan.examples.classes;
/** 這個類別示範 Rect 類別的用法 */
public class RectTest {
    public static void main(String[] args) {
        Rect r1 = new Rect( 1, 1, 4, 4); // 產生 Rect 物件
        Rect r2 = new Rect( 2, 3, 5, 6);
        Rect u = r1.union(r2);          // 呼叫 Rect 的方法
        Rect i = r2.intersection(r1);

        if (u.isInside(r2.x1, r2.y1)) // 以 Rect 的欄位為引數並呼叫 Rect 的方法
            System.out.println("(" + r2.x1 + ", " + r2.y1 + ") is inside the union");

        // 底下兩行會間接呼叫 Rect.toString() 方法
        System.out.println(r1 + " union " + r2 + " = " + u);
        System.out.println(r1 + " intersect " + r2 + " = " + i);
    }
}

```

Rect 的子類別

範例 2-3 是 Rect 類別的子類別 DrawableRect，它繼承 Rect 的欄位與方法，並且新增自己的方法 draw()（這個方法會用特定的 java.awt.Graphics 物件畫出矩形，Graphics 物件將於第十一章詳加介紹）。DrawableRect 也定義了自己的建構式，這個建構式只是將其引數傳給相對應的 Rect 建構式。請注意關鍵字 extends 的用法，它說明 Rect 是 DrawableRect 的超類別。

範例 2-3 : DrawableRect.java

```

package com.davidflanagan.examples.classes;
/**
 * 這是一個可以在螢幕上繪圖的 Rect 子類別。
 * 它繼承所有 Rect 的欄位與方法。
 * 並且利用 java.awt.Graphics 物件繪圖。
 */

```

範例 2-3 : DrawableRect.java (續)

```
public class DrawableRect extends Rect {
    /** DrawableRect 的建構式會呼叫 Rect() 的建構式 */
    public DrawableRect(int x1, int y1, int x2, int y2) { super(x1,y1,x2,y2); }

    /** 這是 DrawableRect 定義的新方法 */
    public void draw(java.awt.Graphics g) {
        g.drawRect(x1, y1, (x2 - x1), (y2 - y1));
    }
}
```

另一個子類別

範例 2-4 示範另一個子類別 ColoredRect，它是 DrawableRect 的子類別，也就是 Rect 的子子類別 (sub-subclass)。它繼承 DrawableRect 與 Rect (當然還包括隱含的超類別 Object) 的欄位與方法。ColoredRect 新增兩個欄位，分別指定繪製時矩形的框線顏色與填入顏色 (這兩個欄位的型態是 java.awt.Color，將於第十一章介紹)，同時也定義一個新的建構式把這些欄位初始化。最後，ColoredRect 會改寫 DrawableRect 類別的 draw() 方法。ColoredRect 把 draw() 定義成以指定顏色繪製矩形，而不是用 DrawableRect 的預設顏色繪製。

範例 2-4 : ColoredRect.java

```
package com.davidflanagan.examples.classes;
import java.awt.*;

/**
 * 這個類別是 DrawableRect 的子類別，它新增繪製矩形用的顏色。
 */
public class ColoredRect extends DrawableRect {
    // 這些是此類別所定義的新欄位。
    // 而 x1、y1、x2、y2 則是從超超類別 (super-superclass) Rect 繼承而來。
    protected Color border, fill;
```

```
/**
 * 這個建構式先用 super() 呼叫超類別的建構式，
 * 然後執行一些本身的初始化動作。
 */
public ColoredRect(int x1, int y1, int x2, int y2, Color border, Color fill) {
    super(x1, y1, x2, y2);
    this.border = border;
    this.fill = fill;
}

/**
 * 這個方法會改寫超類別的 draw()，所以它可以用指定顏色繪製矩形。
 */
public void draw(Graphics g) {
    g.setColor(fill);
    g.fillRect(x1, y1, (x2 - x1), (y2 - y1));
    g.setColor(border);
    g.drawRect(x1, y1, (x2 - x1), (y2 - y1));
}
}
```

複數

範例 2-5 示範複數 (complex number) 的類別定義。你可以從代數學來看複數，複數是由實數與虛數構成。虛數 i 是 -1 的平方根。ComplexNumber 類別定義兩個 double 欄位，分別代表複數的實數部分與虛數部分。這些欄位都宣告為 private，也就是只能在這個類別的內部使用。因為無法從外面存取這些欄位，所以這個類別定義了二個存取方法：real() 與 imaginary() 以傳回欄位的值。把欄位宣告成私有的並定義存取方法這樣做稱為「封裝」(encapsulation)。封裝是對類別使用者隱藏類別的實作，也就是說，在不影響使用者的情況下，可以改變實作。

請注意，除了建構式之外，ComplexNumber 類別並沒有定義任何方法來設定欄位值。換句話說，一旦產生 ComplexNumber 物件，它所代表的值就無法改變，這種性質稱為「不可變性」(immutability)；有時候，把物件設計成有不可變性是非常有用的。

`ComplexNumber` 定義兩個 `add()` 方法與兩個 `multiply()` 方法以執行複數的加法與乘法。每個方法各有兩種版本：一個是實例方法，另一個是類別或靜態方法。以 `add()` 來說。實例方法會把 `ComplexNumber` 類別目前的實例中的值加到另一個指定的 `ComplexNumber` 物件上。因為類別方法沒有目前的實例，所以它會相加兩個指定 `ComplexNumber` 物件的值。呼叫實例方法必須透過類別的實例，就像這樣：

```
ComplexNumber sum = a.add(b);
```

而呼叫類別方法則是透過類別本身而非實例：

```
ComplexNumber sum = ComplexNumber.add(a, b);
```

範例 2-5：ComplexNumber.java

```
package com.davidflanagan.examples.classes;
/** 這是處理複數的類別，裡面定義複數的運算方法。 */
public class ComplexNumber {
    // 實例變數。每個 ComplexNumber 物件有兩個 double 值 (x 和 y)
    // 因為它們是私有的，所以只能在類別內部使用。
    // 但是可以透過下面的 real() 與 imaginary() 存取。
    private double x, y;

    /** 此建構式會把 x 和 y 變數初始化 */
    public ComplexNumber(double real, double imaginary) {
        this.x = real;
        this.y = imaginary;
    }

    /**
     * 存取方法。傳回複數的實數部份。
     * 請注意沒有 setReal() 方法設定實數部份的值。
     * 這意謂 ComplexNumber 類別具有不可變性。
     */
    public double real() { return x; }

    /** 存取方法。傳回複數的虛數部分。 */
    public double imaginary() { return y; }
```

```
/** 計算複數的大小 */
public double magnitude() { return Math.sqrt(x*x + y*y); }

/**
 * 此方法負責把 ComplexNumber 轉成字串。
 * 改寫 Object 的方法，讓複數可以轉成有意義的字串，
 * 以便使用 System.out.println() 與相關方法輸出複數。
 */
public String toString() { return "(" + x + "," + y + ")"; }

/**
 * 這是靜態類別方法。它會把兩個複數相加，然後以新的複數傳回結果。
 * 因為它被宣告成靜態的，所以裡面不能用「目前實例」或「this」物件。
 * 可以這樣用它：
 * ComplexNumber c = ComplexNumber.add(a, b);
 */
public static ComplexNumber add(ComplexNumber a, ComplexNumber b) {
    return new ComplexNumber(a.x + b.x, a.y + b.y);
}

/**
 * 這個方法跟上面的方法有相同名稱，不過它是非靜態的實例方法。
 * 它將指定複數加到目前的複數上。可以這樣用它：
 * ComplexNumber c = a.add(b);
 */
public ComplexNumber add(ComplexNumber a) {
    return new ComplexNumber(this.x + a.x, this.y + a.y);
}

/** 複數相乘的靜態類別方法 */
public static ComplexNumber multiply(ComplexNumber a, ComplexNumber b) {
    return new ComplexNumber(a.x*b.x - a.y*b.y, a.x*b.y + a.y* b.x);
}

/** 複數相乘的實例方法 */
public ComplexNumber multiply(ComplexNumber a) {
    return new ComplexNumber(x*a.x - y*a.y, x*a.y + y*a.x);
}
}
```

產生虛擬隨機亂數

到目前為止，我們定義過的類別都代表真實世界中的物件（複數是抽象的數學概念，不過還是可以把它視為真實世界中的物件）。然而，在某些情況下，你可能需要建立不代表某種物件或抽象概念的類別。在範例 2-6 中，定義了一個可以產生虛擬隨機亂數（pseudo-random number）的類別，就是這種類別。

`Randomizer` 類別並不是為了實作某種類型的物件而定義的（【譯註】：從後面這幾句話中，作者解釋需要有 `Randomizer` 物件的原因）。然而，產生虛擬隨機亂數的這個簡單演算法，卻需要用狀態變數 `seed` 儲存亂數種子。因為我們需要追蹤某些狀態，所以不能像第一章的 `Factorial.factorial()` 方法一樣（【譯註】：`factorial()` 方法是類別方法），把 `Randomizer` 類別的 `random()` 方法定義成靜態方法。如果某個方法需要把前後兩次呼叫時的狀態儲存起來，通常我們必須把這個方法變成物件的實例方法，並且在物件裡面放必要的狀態。不過，這個物件卻無法在真實世界或抽象概念中找到相對應的東西。

這裡的 `Randomizer` 類別有一個實例變數 `seed`，來儲存虛擬隨機亂數的必要狀態，而 `Randomizer` 的其它欄位則宣告成 `static` 與 `final`，成為 Java 的常數。換句話說，每個 `static final` 欄位都會有跟類別相關的欄位，而且值也不會改變。

範例 2-6：Randomizer.java

```
package com.davidflanagan.examples.classes;
/**
 * 這個類別裡面定義取得虛擬隨機亂數的方法。
 * 此外，還定義了這些方法會用到的狀態變數。
 */
public class Randomizer {
    // 所有的 static final 欄位都是常數
    static final int m = 233280;
    static final int a = 9301;
    static final int c = 49297;

    // 這個狀態變數是由每個 Randomizer 實例維護的
    long seed = 1;
```

```
/**
 * Randomizer 的建構式。
 * 必須傳入初始值或「種子」，以設定虛擬隨機性 (pseudo-randomness)。
 */
public Randomizer(long seed) { this.seed = seed; }

/**
 * 這個方法用非常簡單的演算法計算介於 0 與 1 之間的虛擬隨機亂數。
 * 事實上，由 Math.random() 與 java.util.Random 所算出的隨機亂數其隨機性比較好。
 */
public float randomFloat() {
    seed = (seed * a + c) % m;
    return (float)seed / (float)m;
}

/**
 * 這個方法會產生介於 0 與最大指定值之間的虛擬隨機亂數整數。
 * 它會用到上面的 randomFloat()。
 */
public int randomInt(int max) {
    return Math.round(max * randomFloat());
}

/**
 * 這個巢狀類別是一個簡單的測試程式：它會輸出 10 個整數亂數。
 * 請注意，Randomizer 物件如何用目前的系統時間當隨機種子。
 */
public static class Test {
    public static void main(String[] args) {
        Randomizer r = new Randomizer(new java.util.Date().getTime());
        for(int i = 0; i < 10; i++) System.out.println(r.randomInt(100));
    }
}
}
```

範例 2-6 介紹一個很重要的特性。Randomizer 類別裡面定義一個靜態內部類別 Test。Randomizer.Test 內含 main()，因此它是可以測試 Randomizer 的獨立程式。當你編譯 Randomizer.java 檔後，會得到兩個類別檔：Randomizer.class 與 Randomizer\$Test.class。執行這個巢狀類別 Randomizer.Test 需要一點技巧。你可以這麼做：

```
% java Randomizer.Test
```

然而，目前的 Java SDK 版本並無法把類別名稱 Randomizer.Test 正確對映到類別檔案 Randomizer\$Test.class。因此，如果要執行這個測試程式，你必須利用 \$ 符號取代類別名稱的 . 符號來呼叫 Java 直譯器：

```
% java Randomizer$Test
```

如果你是使用 UNIX 系統，因為 \$ 符號已經有特定意義，所以必須在前面加 \ 以跳脫其特殊意義。在這類系統上，你必須鍵入：

```
% java Randomizer\Test
```

或：

```
% java 'Randomizer$Test'
```

計算統計值

範例 2-7 示範一個類別，它會算出一組數字的簡單統計值。當我們把某個數字傳給 addDatum() 方法時，Averager 類別會馬上更新它的內部狀態。因此，其它方法就可以很輕鬆傳回到目前為止的數字平均值與標準差。跟 Randomizer 一樣，Averager 並不代表任何實際物件或抽象概念。儘管如此，Averager 裡面維護了一些狀態（例如 Averager 裡面的一些私有欄位），並且有操作這些狀態的方法，因此它也實作成一般的類別。

編註 若 Randomizer.java 中有 package 宣告，執行時須指定路徑，例如：

```
% java com.davidflanagan.examples.classes.Randomizer$Test.
```

跟範例 2-6 一樣，範例 2-7 定義了一個包含 `main()` 的巢狀類別 `Test` 以實作 `Averager` 的測試程式。

範例 2-7：Averager.java

```
package com.davidflanagan.examples.classes;
/**
 * 這個類別用來計算一群數字的平均值。
 */
public class Averager {
    // 利用私有欄位存放目前狀態
    private int n = 0;
    private double sum = 0.0, sumOfSquares = 0.0;

    /**
     * 這個方法會把一筆新資料加到平均值裡。
     */
    public void addDatum(double x) {
        n++;
        sum += x;
        sumOfSquares += x * x;
    }

    /** 這個方法會傳回所有傳到 addDatum() 之數字的平均值。 */
    public double getAverage() { return sum/n; }

    /** 這個方法會回傳所有傳到 addDatum() 之數字的標準差。 */
    public double getStandardDeviation() {
        return Math.sqrt(((sumOfSquares - sum*sum/n)/n));
    }

    /** 這個方法會傳回所有傳到 addDatum() 之數字的數量。 */
    public double getNum() { return n; }

    /** 這個方法會傳回所有傳到 addDatum() 之數字的總合。 */
    public double getSum() { return sum; }

    /** 這個方法會傳回所有傳到 addDatum() 之數字的平方總合 */
}
```

範例 2-7 : Averager.java (續)

```
public double getSumOfSquares() { return sumOfSquares; }

/** 這個方法會重新把 Averager 物件狀態設定成初始值 */
public void reset() { n = 0; sum = 0.0; sumOfSquares = 0.0; }

/**
 * 這個巢狀類別是一個簡單的測試程式，
 * 用來檢查程式碼是否能夠正常運作。
 */
public static class Test {
    public static void main(String args[]) {
        Averager a = new Averager();
        for(int i = 1; i <= 100; i++) a.addDatum(i);
        System.out.println("Average: " + a.getAverage());
        System.out.println("Standard Deviation: " + a.getStandardDeviation());
        System.out.println("N: " + a.getNum());
        System.out.println("Sum: " + a.getSum());
        System.out.println("Sum of squares: " + a.getSumOfSquares());
    }
}
}
```

鏈結串列類別

範例 2-8 示範一個類別 `LinkedList`，裡面實作鏈結串列 (linked-list) 資料結構。這個範例也定義一個 `Linkable` 介面。如果一個物件想「鏈結」到 `LinkedList` 時，那麼這個物件的類別就必須實作 `Linkable` 介面。回想一下，用介面定義方法時，事實上並沒有對方法提供任何程式主體。類別必須提供介面中每個方法的實作，並且用 `implements` 關鍵字宣告它有實作這個介面。對任何實作 `Linkable` 的類別來說，它的實例都會被視為 `Linkable` 的實例 (【譯註】：多型)。`LinkedList` 物件會把所有串列內的物件視為 `Linkable` 的實例，不需要知道它們的實際型態為何。

請注意，這個範例是為 Java SDK 1.1 版而寫的，Java SDK 1.2 版裡面有提供一個類似但是不相關的類別：`java.util.LinkedList`。後面這個新的 `LinkedList` 集合類別比範例 2-8 裡面定義的類別還有用，不過範例 2-8 是練習使用介面的好例子。

`LinkedList` 定義一個狀態欄位 `head`，它會指向串列的第一個 `Linkable` 項目。同時這個類別也定義一些新增或移除串列項目的方法。請注意，`Linkable` 介面是在 `LinkedList` 類別裡面的，雖然這是一個非常好用的介面定義方式，但是這不意謂一定要定義巢狀的東西。`Linkable` 可以輕易地定義成一個普通而高階的介面。

`LinkedList` 類別裡面也定義了一個內部類別 `Test`，它是用來測試 `LinkedList` 類別的可獨立執行程式。在這個範例中，內部類別 `Test` 本身又包含了另一個內部類別 `LinkableInteger`。`LinkableInteger` 類別負責實作 `Linkable` 介面，它的實例由測試程式鏈結到串列上。

範例 2-8：LinkedList.java

```
package com.davidflanagan.examples.classes;
/**
 * 此類別實作鏈結串列，串列裡面可放進任何實作 Linkable 介面的物件。
 * 請注意，這裡的方法都是同步的，
 * 所以串列可以安全地被多個執行緒同時使用。
 */
public class LinkedList {
    /** 這個介面定義鏈結物件到鏈結串列的方法 */
    public interface Linkable {
        public Linkable getNext();           // 得到串列的下一個項目
        public void setNext(Linkable node); // 設定串列的下一個項目
    }

    // 這個類別有一個預設建構式：public LinkedList() {}

    /** 這是類別裡面的唯一欄位，用來存放串列開頭 */
    Linkable head;

    /** 傳回串列的第一個節點 (node) */
```

範例 2-8 : LinkedList.java (續)

```
public synchronized Linkable getHead() { return head; }

/** 在串列開頭插入一個節點 */
public synchronized void insertAtHead(Linkable node) {
    node.setNext(head);
    head = node;
}

/** 在串列結尾插入一個節點 */
public synchronized void insertAtTail(Linkable node) {
    if (head == null) head = node;
    else {
        Linkable p, q;
        for(p = head; (q = p.getNext()) != null; p = q);
        p.setNext(node);
    }
}

/** 移除並傳回位於串列開頭的節點 */
public synchronized Linkable removeFromHead() {
    Linkable node = head;
    if (node != null) {
        head = node.getNext();
        node.setNext(null);
    }
    return node;
}

/** 移除並傳回位於串列結尾的節點 */
public synchronized Linkable removeFromTail() {
    if (head == null ) return null;
    Linkable p = head, q = null, next = head.getNext();
    if (next == null) {
        head = null;
        return p;
    }
}
```

```
while((next = p.getNext()) != null) {
    q = p;
    p = next;
}
q.setNext(null);
return p;
}

/**
 * 從串列移除指定節點。
 * 我們利用 equals() 而不用 ==，測試跟指定節點相符的節點。
 */
public synchronized void remove(Linkable node) {
    if (head == null) return;
    if (node.equals(head)) {
        head = head.getNext();
        return;
    }
    Linkable p = head, q = null;
    while((q = p.getNext()) != null) {
        if (node.equals(q)) {
            p.setNext(q.getNext());
            return;
        }
        p = q;
    }
}

/**
 * 這個巢狀類別裡面定義 main() 以測試 LinkedList 類別。
 */
public static class Test {
    /** 這是實作 Linkable 介面的測試類別 */
    static class LinkableInteger implements Linkable {
        int i;          // 存放在節點裡面的資料
        Linkable next; // 指向串列的下個節點
        public LinkableInteger(int i) { this.i = i; } // 建構式
        public Linkable getNext() { return next; } // Linkable 介面的方法
    }
}
```

範例 2-8 : LinkedList.java (續)

```
public void setNext(Linkable node) { next = node; } // Linkable 介面的方法
public String toString() { return i + ""; } // 簡單輸出的方法
public boolean equals(Object o) { // 比較節點的方法
    if (this == o) return true;
    if (!(o instanceof LinkableInteger)) return false;
    if (((LinkableInteger)o).i == this.i) return true;
    return false;
}
}

/**
 * 測試程式：插入一些節點、移除一些節點，然後輸出串列的所有項目。
 * 結果應該會輸出數字：4、6、3、1 與 5。
 */

public static void main(String[] args) {
    LinkedList ll = new LinkedList(); // 產生一個串列
    ll.insertAtHead(new LinkableInteger(1)); // 插入一些項目
    ll.insertAtHead(new LinkableInteger(2));
    ll.insertAtHead(new LinkableInteger(3));
    ll.insertAtHead(new LinkableInteger(4));
    ll.insertAtTail(new LinkableInteger(5));
    ll.insertAtTail(new LinkableInteger(6));
    System.out.println(ll.removeFromHead()); // 移除並輸出一個節點
    System.out.println(ll.removeFromTail()); // 移除並輸出另一個節點
    ll.remove(new LinkableInteger(2)); // 移除另一個節點

    // 輸出串列內容
    for(Linkable l = ll.getHead(); l != null; l = l.getNext())
        System.out.println(l);
}
}
```

進階的排序方式

在第一章，我們看到一個數字陣列的排序演算法，這個演算法非常簡單。範例 2-9 中所定義的 `Sorter` 類別則提供一個高效率的排序演算法。`Sorter` 定義了許多 `sort()` 靜態方法，每一個方法都有不同的引數。有些方法是排序方式不同，有些則排序不同物件型態的陣列。這些 `sort()` 最後是用快速排序法（quicksort）排列物件陣列。它們是通用排序方法的變形，最後都會呼叫通用排序方法。

在排列字串方面，`sort()` 會用到 Java 1.1 版的一些國際化特性，尤其是 `java.util.Locale`、`java.text.Collator` 與 `java.text.CollationKey` 這三個類別。在第七章 國際化 將會做更深入的探討。

為了排列物件陣列，`Sorter` 需要比較兩個物件以決定在排序完的陣列中那一個要排在前面。我們在 `Sorter` 裡面為了提供兩種比較方式而定義了兩個巢狀介面：`Comparer` 與 `Comparable`。你可以把 `Comparer` 物件傳給所有 `sort()` 方法以達到排列任意物件的目的。`Comparer` 物件裡面有定義一個比較任意物件的方法 `compare()`。除了使用 `Comparer` 物件之外，我們也可以在需要排序的類別中實作 `Comparable` 介面。在後面的情況下，物件本身具有 `compareTo()` 方法，所以它們可以直接做比較。

請注意在 Java 1.2 版或以後的版本中，`java.util.Arrays` 類別裡面已經定義一些 `sort()` 方法讓我們排列物件或基本型態值陣列。在 `java.util.Collections` 類別裡面也定義了一些 `sort()` 方法，讓我們排列 `java.util.List` 裡面的物件（這些類別都是 Java 1.2 新增的 collection 架構）。這些類別與方法比這裡開發出來的排序方法還要好，不過範例 2-9 仍然是一個有趣而且有用的範例。此外，請注意 `Arrays` 與 `Collections` 裡面的排序方法，它們所使用的 `java.util.Comparator` 與 `java.lang.Comparable` 介面跟範例 2-9 裡面用到的 `Comparer` 與 `Comparable` 介面很像。

本章用範例 2-9 做總結。你應該有注意到它比之前的程式要複雜些，請仔細研究它。這個程式大量使用內部類別，所以研究程式碼之前，請先對內部類別的運作方式有所了解。這個範例的結尾跟前面的範例一樣會用到內部 `Test` 類別，不過其實整個程式到處都用到內部類別與介面。

範例 2-9 : Sorter.java

```
package com.davidflanagan.examples.classes;
// 這些類別支援多地區語言字串的排序
import java.text.Collator;
import java.text.CollationKey;
import java.util.Locale;

/**
 * 此類別定義一些靜態方法，讓 String 或其它物件陣列能夠有效率地排列。
 * 它也定義了兩個介面，替需要排序的物件提供兩種比較方式。
 */
public class Sorter {
    /**
     * 這個介面定義 compare() 方法來比較兩個物件。
     * 欲排列指定型態的物件，必須提供適當的 Comparer 物件，
     * 然後利用裡面的 compare() 方法排列這些物件。
     */
    public static interface Comparer {
        /**
         * 比較物件並利用傳回值指出它們的相對順序：
         * if (a > b) return > 0;
         * if (a == b) return 0;
         * if (a < b) return < 0;
         */
        public int compare(Object a, Object b);
    }

    /**
     * 這是另一個用來排列物件的介面。
     * 如果類別實作這個 Comparable 介面，
     * 就可以呼叫 compareTo() 直接比較該類別的任何兩個實例。
     */
    public static interface Comparable {
        /**
         * 比較物件並利用傳回值指出它們的相對順序：
         * if (this > other) return > 0;
         * if (this == other) return 0;
         */
    }
}
```

```
    * if (this < other) return <0;
    **/
    public int compareTo(Object other);
}

/**
 * 這是內部 Comparer 物件 (利用匿名類別產生),
 * 其功能是比较兩個 ASCII 字串。下面的 sortAscii 方法會用到此物件。
 **/
private static Comparer ascii_comparer = new Comparer() {
    public int compare(Object a, Object b) {
        return ((String)a).compareTo((String)b);
    }
};

/**
 * 這是另一個內部 Comparer 物件, 用來比較兩個 Comparable 物件。
 * 下列的 sort() 會用這個物件當引數而不以任意物件當引數。
 **/
private static Comparer comparable_comparer = new Comparer() {
    public int compare(Object a, Object b) {
        return ((Comparable)a).compareTo(b);
    }
};

/** 由小到大排列 ASCII 字串陣列 **/
public static void sortAscii(String[] a) {
    // 請注意 ascii_comparer 物件的用法
    sort(a, null, 0, a.length-1, true, ascii_comparer);
}

/**
 * 根據引數 up 決定部分 ASCII 字串陣列的排序方式 (由小到大或由大到小)。
 **/
public static void sortAscii(String[] a, int from, int to, boolean up) {
    // 請注意 ascii_comparer 物件的用法
    sort(a, null, from, to, up, ascii_comparer);
}
```

範例 2-9 : Sorter.java (續)

```
/** 由小到大排序 ASCII 字串陣列，並且忽略大小寫 */
public static void sortAsciiIgnoreCase(String[] a) {
    sortAsciiIgnoreCase(a, 0, a.length-1, true);
}

/**
 * 排列部分 ASCII 字串陣列，並且忽略大小寫。
 * 如果 up 是 true 就由小到大排列，否則由大到小排列。
 */
public static void sortAsciiIgnoreCase(String[] a, int from, int to,
                                      boolean up) {
    if ((a == null) || (a.length < 2)) return;
    // 產生次要陣列，裡面放所有指定字串的小寫字。
    String b[] = new String[a.length];
    for(int i = 0; i < a.length; i++) b[i] = a[i].toLowerCase();
    // 排列次要陣列的同時，也改變原始陣列裡面相對應字串的順序，
    // 而得到不分大小寫的排序結果。請注意 ascii_comparer 物件的用法。
    sort(b, a, from, to, up, ascii_comparer);
}

/**
 * 採用預設地區性環境 (locale) 的對照順序 (collation order),
 * 它會把字串陣列用由小到大方式排列。
 */
public static void sort(String[] a) {
    sort(a, 0, a.length-1, true, false, null);
}

/**
 * 利用預設地區性環境的對照順序，排列部分字串陣列。
 * 如果 up 是 true，則用由小到大方式排列，否則用由大到小方式排列。
 * 如果 ignorecase 是 true，則忽略大小寫。
 */
public static void sort(String[] a, int from, int to, boolean up, boolean
                        ignorecase) {
    sort(a, from, to, up, ignorecase, null);
}
```

```
}

/**
 * 利用指定地區性環境的對照順序，排序部分字串陣列。
 * 如果 up 是 true，則用由小到大方式排列，否則用由大到小方式排列。
 * 如果 ignorecase 是 true，則忽略大小寫。
 */
public static void sort(String[] a, int from, int to, boolean up, boolean
                        ignorecase, Locale locale) {
    // 判斷是否需要排序
    if ((a == null) || (a.length < 2)) return;

    // 利用 java.text.Collator 物件進行多地區語言的字串比較。
    // 產生一個指定語系或預設語系
    Collator c;
    if (locale == null) c = Collator.getInstance();
    else c = Collator.getInstance(locale);

    // 指定排序時是否考慮大小寫。
    // 請注意：如果你是用 JDK 1.1.1 版（使用預設的 American English 語系），
    // 那麼這個選項可能無法正常運作。
    if (ignorecase) c.setStrength(Collator.SECONDARY);

    // 利用 Collator 物件產生跟字串相對應的 CollationKey 物件陣列。
    // 比較 CollationKey 的速度比 String 還快
    CollationKey[] b = new CollationKey[a.length];
    for(int i = 0; i < a.length; i++) b[i] = c.getCollationKey(a[i]);

    // 利用匿名類別定義 Comparer 物件以比較 CollationKey
    Comparer comp = new Comparer() {
        public int compare(Object a, Object b) {
            return ((CollationKey)a).compareTo((CollationKey)b);
        }
    };

    // 最後，在排列 CollationKey 物件陣列的同時，
    // 也重新排列原始字串陣列。
    sort(b, a, from, to, up, comp);
}
```

範例 2-9 : Sorter.java (續)

```
}

/** 用由小到大方式排列 Comparable 物件陣列 */
public static void sort(Comparable[] a) {
    sort(a, null, 0, a.length-1, true);
}

/**
 * 排序部分 Comparable 物件陣列。
 * 如果 up 是 true, 則用由小到大方式排列, 否則用由大到小方式排列。
 */
public static void sort(Comparable[] a, int from, int to, boolean up) {
    sort(a, null, from, to, up, comparable_comparer);
}

/**
 * 排列部分 Comparable 物件陣列。
 * 如果 up 是 true, 則用由小到大方式排列, 否則用由大到小方式排列。
 * 利用跟陣列 a 完全相同的方式重新排列陣列 b。
 */
public static void sort(Comparable[] a, Object[] b, int from, int to,
                       boolean up) {
    sort(a, b, from, to, up, comparable_comparer);
}

/**
 * 利用 Comparer 物件 c 所定義的比較方式,
 * 以由小到大方式排列任意物件陣列。
 */
public static void sort(Object[] a, Comparer c) {
    sort(a, null, 0, a.length-1, true, c);
}

/**
 * 利用 Comparer 物件 c 所定義的比較方式, 排列部分物件陣列。
 * 如果 up 是 true, 則用由小到大方式排列, 否則用由大到小方式排列。

```

```
    **/  
    public static void sort(Object[] a, int from, int to, boolean up,  
                           Comparer c) {  
        sort(a, null, from, to, up, c);  
    }  
  
    /**  
    * 這是主要的 sort() 常式 (routine),  
    * 它會針對元素 from 與元素 to 之間的陣列元素 a 做快速排序法。  
    * 利用 up 引數決定元素是要由小到大排列 (true) 或由大到小排列 (false),  
    * Comparer 引數 c 用來比較陣列元素。  
    * 利用跟陣列 a 元素完全相同的方式重新排列陣列 b 的元素。  
    **/  
    public static void sort(Object[] a, Object[] b, int from, int to,  
                           boolean up, Comparer c)  
    {  
        // 如果不需要排序的話，則回先呼叫 sort 的程式碼。  
        if ((a == null) || (a.length < 2)) return;  
  
        // 這是基本的快速排序法。  
        // 把不必要的程式碼刪掉雖然可以加快執行速度，  
        // 不過這樣做會讓程式看起來霧煞煞。  
        // 你應該了解程式碼做什麼事，  
        // 不過，不用知道為什麼這樣做可以排列陣列。  
        // 請注意 Comparer 物件中 compare() 方法的用法。  
        int i = from, j = to;  
        Object center = a[(from + to) / 2];  
        do {  
            if (up) { // 由小到大排序  
                while((i < to) && (c.compare(center, a[i]) > 0)) i++;  
                while((j > from) && (c.compare(center, a[j]) < 0)) j--;  
            } else { // 由大到小排序  
                while(( i < to) && (c.compare(center, a[i]) < 0)) i++;  
                while((j > from) && (c.compare(center, a[j]) > 0)) j--;  
            }  
            if (i < j) {  
                Object tmp = a[i]; a[i] = a[j]; a[j] = tmp; // 置換元素  
                if (b != null) { tmp = b[i]; b[i] = b[j]; b[j] = tmp; } // 置換元素  
            }  
        } while (i < j);  
    }  
}
```

範例 2-9 : Sorter.java (續)

```
    }
    if ( i <= j ) { i++; j--; }
} while(i <= j);
if (from < j) sort(a, b, from, j, up, c); // 用遞迴方式排列其餘部分
if ( i < to) sort(a, b, i, to, up, c);
}

/**
 * 這個巢狀類別定義一個測試程式。
 * 示範許多利用 Sorter 類別來排列 ComplexNumber 物件的做法。
 */
public static class Test {
    /**
     * 此 ComplexNumber 子類別實作 Comparable 介面，
     * 並且定義了用來比較複數的 compareTo() 方法。
     * 它是根據複數的大小 (magnitude) 來比較的，
     * 亦即，根據複數跟原點之間的距離。
     */
    static class SortableComplexNumber extends ComplexNumber
        implements Sorter.Comparable {
        public SortableComplexNumber(double x, double y) { super(x, y); }
        public int compareTo(Object other) {
            return sign(this.magnitude() - ((ComplexNumber)other).magnitude());
        }
    }
}

/** 這是測試程式，它會用各種不同方式來排列複數。 */
public static void main(String[] args) {
    // 定義 SortableComplexNumber 物件陣列。用亂數複數設定初值。
    SortableComplexNumber[] a = new SortableComplexNumber[5];
    for(int i = 0; i < a.length; i++)
        a[i] = new SortableComplexNumber(Math.random() *10, Math.random()*10);

    // 利用 SortableComplexNumber 的 compareTo() 排列並輸出結果。
    System.out.println("Sorted by magnitude:");
    Sorter.sort(a);
}
```

```
for(int i = 0; i < a.length; i++) System.out.println(a[i]);

// 利用 Comparer 物件把複數再排一次，
// 其比較方式是比較各自分開的實部與虛部總合。
System.out.println("Sorted by sum of their real and imaginary parts:");
Sorter.sort(a, new Sorter.Comparer() {
    public int compare(Object a, Object b) {
        ComplexNumber i = (ComplexNumber)a, j = (ComplexNumber)b;
        return sign((i.real() + i.imaginary()) -
                    (j.real() + j.imaginary()));
    }
});
for(int i = 0; i < a.length; i++) System.out.println(a[i]);

// 利用 Comparer 物件再排一次。
// 先比較它們的實部，再比較它們的虛部。
System.out.println("Sorted descending by real part, then imaginary:");
Sorter.sort(a, 0, a.length-1, false, new Sorter.Comparer() {
    public int compare(Object a, Object b) {
        ComplexNumber i = (ComplexNumber) a, j = (ComplexNumber) b;
        double result = i.real() - j.real();
        if (result == 0) result = i.imaginary() - j.imaginary();
        return sign(result);
    }
});
for(int i = 0; i < a.length; i++) System.out.println(a[i]);
}

/** 這是 sort 常式裡面會用到的另一個常式 */
public static int sign(double x) {
    if (x > 0) return 1;
    else if (x < 0) return -1;
    else return 0;
}
}
}
```

習題

- 2-1. 寫一個跟 `Rect` 類別類似的 `Circle` 類別。定義 `move()` 與 `isInside()` 方法。圓是由所有距離中心點指定半徑內的點構成，我們可以利用畢氏定理 (Pythagorean theorem) 算出點與中心點之間的距離。另外，定義一個 `boundingBox()` 方法傳回包圍 `Circle` 的最小 `Rect`。另外寫一個簡單的程式來測試所實作的方法。
- 2-2. 寫一個可以顯示個人郵寄地址的類別，它必須有下列欄位：人名、號碼、巷弄、街道名、城市與郵遞區碼。定義一個 `toString()` 方法產生完整的格式化輸出。
- 2-3. 利用 `Sort.Comparer` 或 `Sort.Comparable` 介面，撰寫一個 `Search` 類別的靜態方法 `search()`，針對排序過的物件陣列，執行快速的二元搜尋法。如果在陣列內找到這個物件的話，那麼 `search()` 就傳回陣列的索引值（發現物件的位置）；否則就傳回 `-1`。