導讀

這份導讀可讓你更了解如何使用本書。

新舊版差異

本書使用的 IDE 換成了 Eclipse，理由之一是之前使用的 NetBeans IDE，在移交 Apache 基金會後做了許多調整，有些功能在使用上不符合本書需求，理由之二是可銜接我的另一本書《Servlet&JSP 技術手冊-從 Servlet 到 Spring Boot》，該書亦使用 Eclipse IDE。

就 JDK10 至 14 的新特性，當然也列入了本書之中，像是 JDK10 的 `var` 宣告，JDK11 的 Single-File Source-Code 與 `var` 增強，JDK14 的 `switch` expression 等。

過去書中會針對多年才釋出一次的 JDK 重大版本，整理出新功能索引，由於 JDK 從 Java 9 開始，以半年為一個釋出週期（參考第 1 章），每次釋出都會包含部分新特性，書中整理出新功能索引的意義已經不大，就不再列出了。

雖然本書是基於 Oracle JDK 製作範例，然而現今 JDK 來源多元化，你應該好好閱讀第 1 章，瞭解有哪些 JDK 可以選擇。

自 Java SE 12 開始，有些 Java 新功能未正式定案前，為了取得開發者的意見回饋，會以預覽形式發佈；由於預覽功能未來仍可能變動規格，本書不會說明預覽功能。

至於書本的內容，照慣例整本都重新審閱了一次；範例可以使用 `var` 簡化變數的部分，基本上就會使用，然而若覺得保留型態，可以突顯當時主題想強調的觀念，就不會使用 `var`；4.4 談字串時，加入了更多 Unicode 的說明，這是因為身為開發者，必須對 Unicode 有一定程度的認識；15.3 增加了更多規則表示式（Regular expression）的介紹，其中也談到了規則表示式對 Unicode 的支援。

第 16 章談 JDBC 時，連線的資料庫換成了 H2，這是個純 Java 實現的資料庫，功能比 SQLite 多，而且也是《Servlet&JSP 技術手冊-從 Servlet 到 Spring Boot》使用的資料庫，亦為 Spring 推薦使用的資料庫之一。

字型

本書內文中與程式碼相關的文字，使用等寬字型來加以呈現，以與一般名詞做區別。例如 JDK 是一般名詞，而 `String` 為程式碼相關文字，使用了等寬字型。

程式範例

你可以在以下網址下載本書的範例：

- http://books.gotop.com.tw/v_ACL059300

本書許多的範例示範，都使用完整程式實作來展現，當你看到以下程式碼示範時：

ClassObject Guess.java

```
package cc.openhome;

import java.util.Scanner; ◄── ❶告訴編譯器接下來想偷懶
import static java.lang.System.out;

public class Guess {
    public static void main(String[] args) {
        var console = new Scanner(System.in);  ◄── ❷建立 Scanner 實例
        var number = (int) (Math.random() * 10);
        var guess = -1;

        do {
            out.print("猜數字（0 ~ 9）:");
            guess = console.nextInt(); ◄── ❸取得下一個整數
        } while(guess != number);

        out.println("猜中了...XD");
    }
}
```

範例開始的左邊名稱為 ClassObject，表示可在範例檔的 **samples** 資料夾中各章節資料夾，找到對應的 ClassObject 專案，而右邊名稱為 Guess.java，表示可在專案中找到 Guess.java 檔案。若程式碼出現標號與提示文字，表示後續的內文中，會有對應於標號及提示的更詳細說明。

原則上，建議每個專案範例都親手動作撰寫，但若因教學時間或實作時間上的考量，本書有建議進行的練習，如果在範例旁發現有個  圖示，例如：

Game1 SwordsMan.java

```
package cc.openhome;

public class SwordsMan extends Role {
    public void fight() {
        System.out.println("揮劍攻擊");
    }
}
```

表示建議範例動手實作，而且在範例檔的 labs 資料夾中，會有練習專案的基礎，可以開啟專案後，完成專案中遺漏或必須補齊的程式碼設定。

若使用以下的程式碼呈現，表示它是完整的程式內容，但不是專案的一部分，主要用來展現完整內容如何撰寫：

```
public class Hello {
     public static void main(String[] args) {
         System.out.println("Hello!World!");
     }
}
```

如果使用以下的方式呈現，表示它是程式片段，主要展現程式撰寫時需要特別注意的部分：

```
var swordsMan = new SwordsMan();
...略
out.printf("劍士 (%s, %d, %d)%n", swordsMan.getName(),
        swordsMan.getLevel(), swordsMan.getBlood());
var magician = new Magician();
...略
out.printf("魔法師 (%s, %d, %d)%n", magician.getName(),
        magician.getLevel(), magician.getBlood());
```

對話框

本書會出現以下的對話框：

提示 »» 針對課程中提到的觀念，提供一些額外資源或思考方向，暫時忽略這些提示對課程進行沒有影響，然而有時間的話，針對這些提示多做思考或討論是有幫助的。

注意 »» 針對課程中提到的觀念，以對話框方式特別呈現出必須注意的使用方式、陷阱或避開問題的方法，看到這個對話框時請集中精神閱讀。

附錄

範例檔案中包括本書中全部範例，提供 Eclipse 範例專案，附錄 A 說明如何使用這些範例專案。

聯繫作者

若有堪誤回報等相關書籍問題，可透過網站與作者聯繫：

- openhome.cc

chapter

4 認識物件

學習目標

- 區分基本型態與類別型態
- 瞭解物件與參考的關係
- 從包裹器認識物件
- 以物件觀點看待陣列
- 認識字串的特性
- 知道如何查詢 API 文件

4.1 類別與實例

Java 有基本型態與類別型態兩個型態系統，第 3 章談過基本型態，本章要來談類別型態，使用 Java 撰寫程式幾乎都在使用**物件（Object）**，要產生物件必須先定義**類別（Class）**，類別是物件的設計圖，物件是某個類別的**實例（Instance）**。

不賣弄術語了，讓我們開始吧！....

4.1.1 定義類別

正式說明如何使用 Java 定義類別前，先來看看，如果要設計衣服是如何進行？有個衣服的設計圖，上頭定義了衣服的款式與顏色、尺寸，你會根據設計圖製作出實際的衣服，每件衣服都是同一款式，但擁有自己的顏色與尺寸，你會為每件衣服別上名牌，這個名牌只能別在同款式的衣服上。

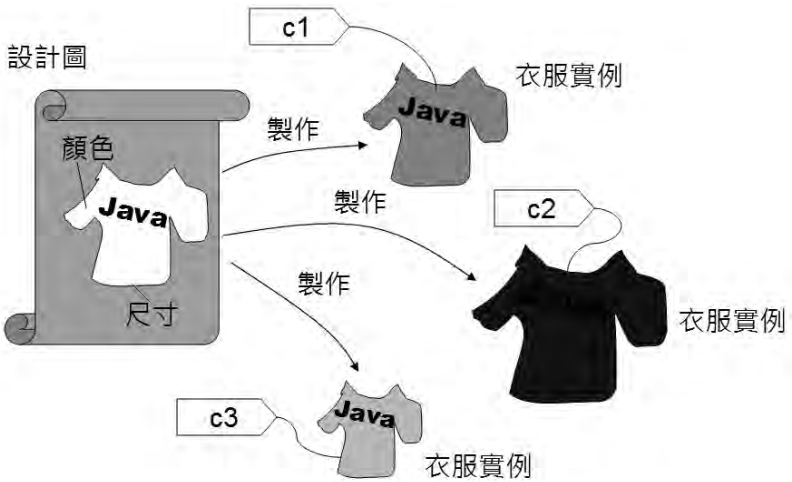

圖 4.1 設計圖、製作、實例與款式名牌

如果今天要設計的就是有關服飾設計的軟體，如何使用 Java 撰寫呢？可以在程式中定義類別，這相當於上圖中衣服的設計圖：

```
class Clothes {
    String color;
    char size;
}
```

類別定義時使用 **class** 關鍵字，名稱使用 Clothes，相當為衣服設計圖取名為 Clothes，衣服的顏色用字串表示，也就是 color 變數，可儲存"red"、"black"、"blue"等值，衣服尺寸會是'S'、'M'、'L'，使用 char 型態宣告變數。若要在程式中，利用 Clothes 類別作為設計圖，建立衣服實例，要使用 **new** 關鍵字。例如：

```
new Clothes();
```

在物件術語中，這叫作新建一個**物件**，更精確地說，是新建 Clothes 的**實例**，若要有個名牌，專門綁到這個物件上，可以如下宣告：

```
Clothes c1;
```

在 Java 的術語中，這叫宣告**參考名稱（Reference name）、參考變數（Reference variable）或直接叫參考（Reference）**，當然，c1 本質上就是個變數。若要將 c1 綁到新建的物件上，可以使用=指定，以 Java 術語來說，稱 c1 **參考（refer）**至新建物件。例如：

```
Clothes c1 = new Clothes();
```

可以將程式語法如下圖表示，直接對照圖 4.1，就可以瞭解類別與實例的區別，以及 class、new、=等語法的使用：

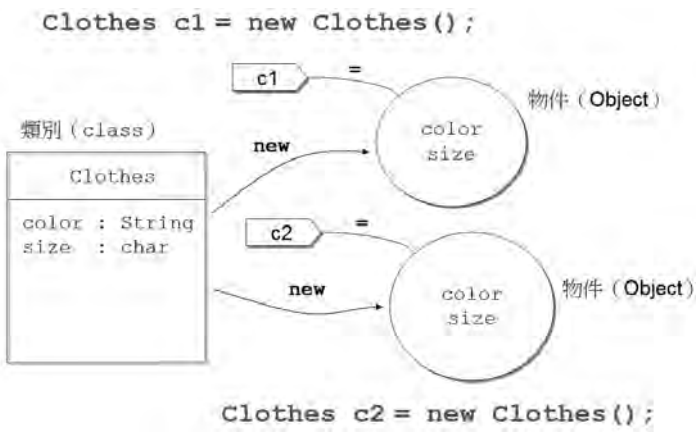

圖 4.2　class、new、=等語法對照

提示》》 物件（Object）與實例（Instance），在 Java 中幾乎是等義的名詞，本書中就視為相同意義，會交相使用這兩個名詞。

`Clothes c1 = new Clothes()`的寫法中，出現了兩次 `Clothes`，這是重複的資訊，Java SE 10 以後，第 3 章談過，從 Java SE 10 開始，若編譯器可以從前後文推斷出**區域變數**型態，可以使用 `var` 宣告變數，不用明確指定變數型態，也就是在函式中的話可以寫為：

```
var clothes = new Clothes();
```

在 `Clothes` 類別中，定義了 `color` 與 `size` 兩個變數，以 Java 術語來說，叫作定義兩個值域（Field）成員，或是定義兩個資料成員，這表示每個新建的 `Clothes` 實例，可以擁有個別的 `color` 與 `size` 值。例如：

ClassObject Field.java

```
package cc.openhome;

class Clothes {                          ❶ 定義 Clothes 類別
    String color;
    char size;
}

public class Field {
    public static void main(String[] args) {
        var sun = new Clothes();
        var spring = new Clothes();      ❷ 建立 Clothes 實例

        sun.color = "red";
        sun.size = 'S';
        spring.color = "green";          ❸ 爲個別物件的資料成員指定值
        spring.size = 'M';

        System.out.printf("sun (%s, %c)%n", sun.color, sun.size);          ❹ 顯示個別物件的資料成員值
        System.out.printf("spring (%s, %c)%n", spring.color, spring.size);
    }
}
```

在這個 Field.java 中，定義了兩個類別，一個是公開（`public`）的 `Field` 類別，因此檔案主檔名必須是 Field，另一個是非公開的 `Clothes`❶，回憶一下第 2 章，一個原始碼中可以有多個類別定義，但只能有一個是公開類別，且檔案主檔名必須與公開類別名稱相同。

提示 »» 只要有一個類別定義，編譯器就會產生一個.class 檔案。上例中，實際上會產生 Field.class 與 Clothes.class 兩個檔案。

程式中建立了兩個 Clothes 實例，並分別宣告了 sun 與 spring 名稱來參考❷，接著要求 JVM，將 sun 參考的物件之 color 與 size 分別指定為"red"與'S'，將 spring 的 color 與 size 分別指定為"green"與'M'❸。最後分別顯示 sun、spring 的資料成員值❹。

執行結果如下，可以看到 sun 與 spring 擁有各自的資料成員：

```
sun (red, S)
spring (green, M)
```

方才的範例中，為個別物件指定資料成員值的程式流程是類似的，若想在建立物件時，一併進行某個初始流程，像是指定資料成員值，可以定義**建構式（Constructor）**，建構式是與類別名稱同名的**方法（Method）**，直接來看範例比較清楚：

ClassObject_Field2.java

```
package cc.openhome;

class Clothes2 {
    String color;
    char size;

    Clothes2(String color, char size) {  ❶定義建構式
        this.color = color;  ❷ color 參數的值指定給這個物件的 color 成員
        this.size = size;
    }
}

public class Field2 {
    public static void main(String[] args) {
        var sun = new Clothes2("red", 'S');      ❸使用指定建構式建立物件
        var spring = new Clothes2("green", 'M');

        System.out.printf("sun (%s, %c)%n", sun.color, sun.size);
        System.out.printf("spring (%s, %c)%n", spring.color, spring.size);
    }
}
```

在這個例子中，定義新建物件時，必須傳入兩個引數，分別指定給字串型態的 color 參數（Parameter）與 char 型態的 size 參數❶，而建構式中，由於 color 參數與資料成員 color 同名，不可以直接寫 color = color，這會將 color 參數的值又指定給 color 參數，你必須使用 **this** 表示，將 color 參數的值指定給這個物件（this）的 color 成員。

接下來使用 new 建構物件時，只要傳入字串與字元，就可以初始 Clothes 實例的 color 與 size 值，執行結果與上個範例是相同的。

注意 物件導向中有所謂封裝（Encapsulation），不過這邊只是定義類別，離封裝還有很大的距離，第 5 章會詳細談到如何用 Java 來實現更完整的封裝。

4.1.2 使用標準類別

在之前的說明中，瞭解了定義類別之後，可以建立類別的實例進行操作，第 1 章談過，Java SE 提供了標準 API，這些 API 就是由許多類別組成，可以直接取用這些標準類別，省去撰寫程式時重新打造輪子的需求。底下來舉兩個基本的標準類別：java.util.Scanner 與 java.math.BigDecimal。

提示 java.util 與 java.math 套件是歸在 java.base 模組之中，因此不用在模組描述檔中增加任何設定。

使用 java.util.Scanner

目前為止的程式範例都很無聊，變數值都是寫死的，沒有辦法接受使用者的輸入。若要在文字模式下取得使用者輸入，基本上可以使用 **System.in** 物件的 **read()**方法，不過這個方法是以 int 型態傳回讀入的字元編碼。想想，若輸入了一個字元'9'，使用 System.in.read()的話，還得自行將字元'9'轉換為整數 9，真的是不方便。

你可以使用 **java.util.Scanner** 來代勞，底下直接以實際範例來說明：

ClassObject Guess.java

```
package cc.openhome;

import java.util.Scanner;  ◄── ❶告訴編譯器接下來想偷懶

public class Guess {
    public static void main(String[] args) {
        var console = new Scanner(System.in);      ◄── ❷建立 Scanner 實例
        var number = (int) (Math.random() * 10);
        var guess = -1;

        do {
            System.out.print("猜數字（0 ~ 9）:");
```

```
            guess = console.nextInt();  ◀── ❸取得下一個整數
        } while(guess != number);

        System.out.println("猜中了...XD");
    }
}
```

由於不想每次都鍵入 java.util.Scanner，一開始就使用 import 告訴編譯器，之後就只要鍵入 Scanner 就可以了❶。在建立 Scanner 實例時，必須傳入 java.io.InputStream 的實例，第 10 章介紹到輸入輸出串流時會知道，System.in 就是一種 InputStream，可以在建構 Scanner 實例時使用❷。

接下來想要什麼資料，就跟 Scanner 的實例要就可以了，正如其名，範例中的 Scanner 實例會掃描標準輸入，看看使用者有無輸入字元，怎麼掃描你就不用管了，反正是有個 Scanner.java 定義了程式碼做這些事，編譯為 Scanner.class 成為 Java SE 標準 API 供大家使用。

Scanner 的 **nextInt()**方法會看看標準輸入中，有沒有輸入下個字串（以空白或換行為區隔），有的話會嘗試剖析為 int 型態❸，Scanner 對每個基本型態，都有對應的 nextXXX()方法，例如 **nextByte()**、**nextShort()**、**nextLong()**、**nextFloat()**、**nextDouble()**、**nextBoolean()**等，如果想取得下個字串（以空白或換行為區隔），可以使用 **next()**，若想取得使用者輸入的整行文字，就使用 **nextLine()**（以換行為區隔）。

提示 »» 慣例上，套件名稱為 java 開頭的類別，表示標準 API 提供的類別。

使用 java.math.BigDecimal

第 2 章介紹基本型態時留了一個問題給你：1.0 - 0.8 的結果是？答案不是 0.2，而是 0.19999999999999996！為什麼？這是 Java 的臭蟲（Bug）嗎？不！不是的！使用其他程式語言（例如 JavaScript、Python 等）也有可能顯示這個結果。

簡單來說，Java（包括其他程式語言）遵合 IEEE 754 浮點數演算（Floating-point arithmetic）規範，使用分數與指數來表示浮點數。例如 0.5 會使用 1/2 來表示，0.75 會使用 1/2 + 1/4 來表示，0.875 會使用 1/2 + 1/4 + 1/8

來表示，而 0.1 會使用 1/16 + 1/32 + 1/256 + 1/512 +1/4096 + 1/8192 + … 無限循環下去，無法精確表示，因而造成運算上的誤差。

再來舉個例子，你覺得以下程式片段會顯示什麼結果？

```
var a = 0.1;
var b = 0.1;
var c = 0.1;
if((a + b + c) == 0.3) {
    System.out.println("等於 0.3");
}
else {
    System.out.println("不等於 0.3");
}
```

由於浮點數誤差的關係，結果是顯示「不等於 0.3」！類似的例子還很多，結論就是，**如果要求精確度，就要小心使用浮點數，而且別用==比較浮點數運算結果**。

要如何得到更好的精確度呢？可以使用 **java.math.BigDecimal** 類別，以方才的 `1.0 - 0.8` 為例，如何得到 0.2 的結果？直接使用程式來示範：

ClassObject DecimalDemo.java

```
package cc.openhome;

import java.math.BigDecimal;

public class DecimalDemo {
    public static void main(String[] args) {
        var operand1 = new BigDecimal("1.0");
        var operand2 = new BigDecimal("0.8");
        var result = operand1.subtract(operand2);
        System.out.println(result);
    }
}
```

建構 `BigDecimal` 的方法之一是使用字串，`BigDecmial` 在建構時會剖析傳入字串，以預設精度進行接下來的運算，`BigDecimal` 提供有 **add()**、

subtract()、**multiply()**、**divide()**等方法，可以進行加、減、乘、除等運算，這些方法都會傳回代表運算結果的 BigDecimal。

上面這個範例可以顯示出 0.2 的結果，再來看利用 BigDecimal 比較相等的例子：

ClassObject DecimalDemo2.java

```
package cc.openhome;

import java.math.BigDecimal;

public class DecimalDemo2 {
    public static void main(String[] args) {
        var op1 = new BigDecimal("0.1");
        var op2 = new BigDecimal("0.1");
        var op3 = new BigDecimal("0.1");
        var result = new BigDecimal("0.3");

        if(op1.add(op2).add(op3).equals(result)) {
            System.out.println("等於 0.3");
        }
        else {
            System.out.println("不等於 0.3");
        }
    }
}
```

由於 BigDecimal 的 add()等方法，都會傳回代表運算結果的 BigDecmial，因此可以利用傳回的 BigDecimal 再呼叫 add()方法，最後再呼叫 **equals()**比較兩個 BigDecimal 實質上是否相同，也就有了 a.add(b).add(c).equals(result)的寫法。

提示 »» JWorld@TW 的〈[FAQ] 為何 1.0 - 0.8 不是 0.2?〉討論中，可以看到更多浮點數誤差的例子：
www.javaworld.com.tw/jute/post/view?bid=29&id=19197&tpg=1&ppg=1&sty=1&age=0

這邊先簡介了 BigDecimal 的使用，在第 15 章還會有更詳細的說明。

4.1.3 物件指定與相等性

在上一個範例中，比較兩個 BigDecimal 是否相等，是使用 equals()方法而非使用==運算子，為什麼？先前提過，Java 並非完全的物件導向程式語言，在 Java 中有兩大型態系統，基本型態與類別型態，這很令人困擾，若不討論底層記憶體實際運作，初學者就必須區分=與==運算用於基本型態與類別型態的不同。

當=用於基本型態時，是將值複製給變數，==用於基本型態時，是比較兩個變數儲存的值是否相同，這對初學者來說沒有問題，底下的程式片段會顯示兩個 true，因為 a 與 b 儲存的值都是 10，而 a 與 c 儲存的值也都是 10：

```
jshell> var a = 10;
a ==> 10

jshell> var b = 10;
b ==> 10

jshell> var c = 10;
c ==> 10

jshell> a == b;
$4 ==> true

jshell> a == c;
$5 ==> true
```

如果操作物件，=用在指定名稱參考至某個物件，而==是用在比較兩個名稱是否參考同一物件。對初學者來說，通常看不懂這句話是什麼意思，白話來說，**=用在將某個名牌綁到某個物件，而==是用在比較兩個名牌是否綁到同一物件**。來看個範例：

```
jshell> var a = new BigDecimal("0.1");
a ==> 0.1

jshell> var b = new BigDecimal("0.1");
b ==> 0.1

jshell> a == b;
$4 ==> false

jshell> a.equals(b);
$5 ==> true
```

上面的程式片段，建議初學者以繪圖表示，以第一行為例，看到 new 關鍵字，就是建立物件，那就畫個圓圈表示物件，這個物件內含"0.1"，並建立一個名牌 a 綁到新建立的物件，因此第一行與第二行執行後，可用下圖來表示：

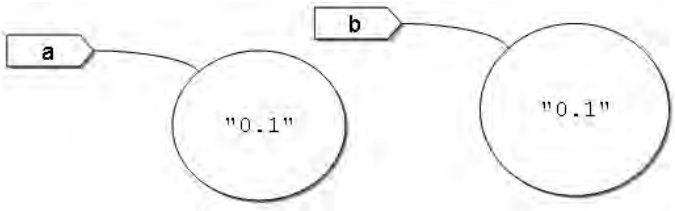

圖 4.3　=用於物件指定的示意圖

程式中使用 a == b，就是在問，a 牌子綁的物件是否就是 b 牌子綁的物件？答案「不是」，也就是 false 的結果，程式中使用 a.equals(b)，就是在問，a 牌子綁的物件與 b 牌子綁的物件，實際上內含值是否相同，因為 a 與 b 綁的物件，內含值都是"0.1"代表的數值，答案「是」，也就是 true 的結果。

再來看一個例子：

```
jshell> var a = new BigDecimal("0.1");
a ==> 0.1

jshell> var b = new BigDecimal("0.1");
b ==> 0.1

jshell> var c = a;
c ==> 0.1

jshell> a == b;
$4 ==> false

jshell> a == c;
$5 ==> true

jshell> a.equals(b);
$6 ==> true
```

這個程式片段若執行至第三行 c = a，表示將 a 牌子綁的物件，也給 c 牌子來綁，用圖表示就是：

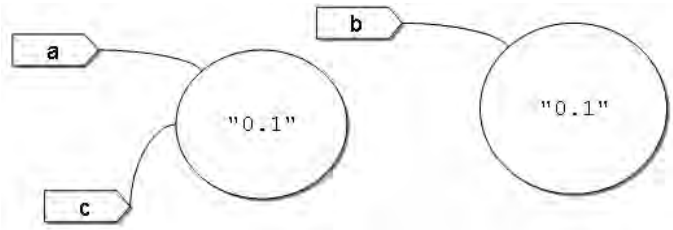

圖 4.4 =用於物件指定的示意圖

若問到 a == b，就是在問 a 與 b 是否綁在同一物件？結果就是 false，問到 a == c，就是在問 a 與 c 是否綁在同一物件？結果就是 true，問到 a.equals(b)，就是在問 a 與 b 綁的物件實際上內含值是否相同？結果就是 true。

==用在物件型態，是比較兩個名稱是否參考同一物件，而!=正好相反，是比較兩個名稱是否沒參考同一物件。實際上，equals()可以自行定義如何比較兩物件的內含值，這將在第 6 章再做說明。

提示 其實從記憶體的實際運作來看，=與==並沒有用在基本型態與物件型態的不同，有興趣的話，可以參考〈我們沒什麼不同〉：openhome.cc/Gossip/JavaEssence/EqualOperator.html

4.2 基本型態包裹器

基本型態 long、int、double、float、boolean 等，在 J2SE 5.0 之前必須親自使用 Long、Integer、Double、Float、Boolean 等類別包裹為物件，才能當作物件來操作，J2SE 5.0 開始支援了自動裝箱（Autoboxing）、拆箱（Unboxing），然而也必要瞭解如何包裹基本型態，這有助於進一步瞭解物件與基本型態的差別。

4.2.1 包裹基本型態

從第 3 章就一直談到，Java 中有兩個型態系統，基本型態與類別型態，使用基本型態目的在於效率，然而更多時候，會使用類別建立實例，因為物件本身可以攜帶更多資訊，若要讓基本型態如同物件般操作，可以使用 **Long**、

Integer、**Double**、**Float**、**Boolean**、**Byte** 等類別建立實例來包裹（Wrap）基本型態。

Long、Integer、Double、Float、Boolean 等類別是所謂的包裹器（Wrapper），正如此名稱所示，這些類別主要目的，是提供物件實例作為「殼」，將基本型態包裹在物件之中，接著就可以直接操作這個物件：

Wrapper IntegerDemo.java

```
package cc.openhome;

public class IntegerDemo {
    public static void main(String[] args) {
        int data1 = 10;
        int data2 = 20;

        var wrapper1 = new Integer(data1);   ←— ❶包裹基本型態
        var wrapper2 = new Integer(data2);

        System.out.println(data1 / 3);    ←— ❷基本型態運算
        System.out.println(wrapper1.doubleValue() / 3);   ←— ❸操作包裹器方法
        System.out.println(wrapper1.compareTo(wrapper2));
    }
}
```

基本型態包裹器都是歸類於 java.lang 套件中，如果要使用 Integer 包裹 int 型態資料，方法之一是用 new 建構 Integer 實例時，傳入 int 型態資料❶。在第 3 章提過，若運算式中都是 int，就只會在 int 空間中做運算，結果會是 int 整數，因此 data1 / 3 就會顯示 3 的結果❷。你可以操作 Integer 的 **doubleValue()**將包裹值以 double 型態傳回，如此就會在 double 空間中做相除，結果就會顯示 3.33333333333... ❸。

Integer 提供 **compareTo()**方法，可與另一個 Integer 物件進行比較，如果包裹值相同就傳回 0，小於 compareTo()傳入物件包裹值就傳回-1，否則就是 1，與==或!=只能比較是否相等或不相等，compareTo()方法傳回更多資訊。

4.2.2　自動裝箱、拆箱

除了使用 new 建構基本型態包裹器之外，J2SE 5.0 以後提供了**自動裝箱（Auto boxing）**功能，可以如下包裹基本型態：

```
Integer number = 10;
```

編譯器會自動判斷是否能進行自動裝箱，在上例中 number 會參考 Integer 實例；同樣的動作可適用於 boolean、byte、short、char、int、long、float、double 等基本型態，分別會使用對應的 Boolean、Byte、Short、Character、Integer、Long、Float 或 Double 包裹基本型態。若使用自動裝箱功能來改寫一下 IntegerDemo 中的程式碼：

```
Integer data1 = 10;
Integer data2 = 20;
System.out.println(data1.doubleValue() / 3);
System.out.println(data1.compareTo(data2));
```

程式看來簡潔許多，data1 與 data2 在運行時會參考 Integer 實例，可以直接進行物件操作。自動裝箱運用的方法還可以如下：

```
int number = 10;
Integer wrapper = number;
```

也可以使用更一般化的 Number 類別來自動裝箱，例如：

```
Number number = 3.14f;
```

3.14f 會先被自動裝箱為 Float，然後指定給 number。

J2SE 5.0 以後可以自動裝箱，也可以**自動拆箱（Auto unboxing）**，自動取出包裹器中的基本形態資訊。例如：

```
Integer wrapper = 10;   // 自動裝箱
int foo = wrapper;      // 自動拆箱
```

wrapper 會參考至 Integer，若被指定給 int 型態的變數 foo，會自動取得包裹的 int 型態再指定給 foo。

在運算時，也可以進行自動裝箱與拆箱，例如：

```
jshell> Integer number = 10;
number ==> 10

jshell> System.out.println(number + 10);
20

jshell> System.out.println(number++);
10
```

上例中會顯示 20 與 10，編譯器會自動自動裝箱與拆箱，也就是 10 會先裝箱，然後在 number + 10 時會先對 number 拆箱，再進行加法運算；number++ 該行也是先對 number 拆箱再進行遞增運算。再來看一個例子：

```
jshell> Boolean foo = true;
foo ==> true

jshell> System.out.println(foo && false);
false
```

同樣地，foo 會參考至 Boolean 實例，在進行&&運算時，會先將 foo 拆箱，再與 false 進行&&運算，結果會顯示 false。

實際上，不建議使用 new 建立基本型態包裹器，從 Java SE 9 開始，基本型態包裹器的建構式都標示為棄用（Deprecated）了，試圖使用 new 建構基本型態包裹器的話會發生編譯警訊：

```
Note: IntegerDemo.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details
```

在使用 javac 時加上-Xlint:deprecation，可以看到更清楚的警訊細節，例如編譯先前的 IntegerDemo.java 的話，會有以下訊息：

```
IntegerDemo.java:8: warning: [deprecation] Integer(int) in Integer has been
deprecated
        var wrapper1 = new Integer(data1);
                           ^
IntegerDemo.java:9: warning: [deprecation] Integer(int) in Integer has been
deprecated
        var wrapper2 = new Integer(data2);
                           ^
2 warnings
```

4.2.3　自動裝箱、拆箱的內幕

自動裝箱與拆箱的功能是**編譯器蜜糖（Compiler sugar）**，也就是編譯器讓你撰寫程式時吃點甜頭，編譯時期會決定是否進行裝箱或拆箱動作。例如：

```
Integer number = 100;
```

在 Oracle 的 JDK 上，編譯之後再予以反組譯會得到以下程式碼：

```
Integer localInteger = Integer.valueOf(100);
```

提示 Java 位元碼格式是公開標準，有位元碼檔案，就可以嘗試使用反組譯程式轉譯為 Java 語法。本書會使用的反組譯程式有 JD（jd.benow.ca）與 JAD（varaneckas.com/jad/），後者雖然沒有在維護了，不過反組譯後的程式碼可看到較多語法蜜糖的實際細節。

使用 `Integer.valueOf()`是為基本型態建立包裹器的方式之一。瞭解編譯器會如何裝箱與拆箱是必要的，例如下面的程式是可以通過編譯的：

```
Integer i = null;
int j = i;
```

然而執行時期會有錯誤，因為編譯器會將之展開為：

```
Object localObject = null;
int i = localObject.intValue();
```

在 Java 程式碼中，**`null`** 代表一個特殊物件，任何類別宣告的名稱都可以參考至 `null`，表示該名稱沒有參考至任何物件實體，這相當於有個名牌沒有任何人佩戴。在上例中，由於 `i` 沒有參考任何物件，就不可能操作 `intValue()` 方法，就相當於有個名牌沒人佩戴，卻要求戴名牌的人舉手，這是一種錯誤，在 Java 中會出現 **`NullPointerException`** 的錯誤訊息。

編譯器蜜糖提供了方便性，也因此隱藏了一些細節，別只顧著吃糖而忽略了該知道的觀念。來看看，如果如下撰寫，結果會是如何？

```
Integer i1 = 100;
Integer i2 = 100;

if (i1 == i2) {
    System.out.println("i1 == i2");
}
else {
    System.out.println("i1 != i2");
}
```

如果只看 `Integer i1 = 100`，就好像在看 `int i1 = 100`，直接使用==進行比較，有的人會理所當然回答顯示`"i1 == i2"`，那麼底下這個呢？

```
Integer i1 = 200;
Integer i2 = 200;

if (i1 == i2) {
    System.out.println("i1 == i2");
}
```

```
else {
    System.out.println("i1 != i2");
}
```

程式碼只不過將 100 改為 200，執行結果卻顯示了"i1 != i2"，這是為何？先前提過，自動裝箱是編譯器蜜糖，以上例來說，實際上會使用**Integer.valueOf()**來建立 Integer 實例，因此得知道 Integer.valueOf() 到底如何建立 Integer 實例，察查 JDK 資料夾 lib/src.zip 中的 java.base/java/lang 資料夾中的 Integer.java，你會看到 valueOf()的實作內容：

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

這段程式碼簡單來說，就是如果傳入的 int 在 IntegerCache.low 與 IntegerCache.high 之間，嘗試看看快取（Cache）中有沒有包裹過相同的值，如果有就直接傳回，否則就使用 new 建構新的 Integer 實例。IntegerCache.low 預設值是-128，IntegerCache.high 預設值是 127。

提示》》 IntegerCache 是 Integer 類別內部實作中的一個類別，快取會在首次使用到 IntegerCache 類別時建立。

因此若是這個程式碼：

```
Integer i1 = 100;
Integer i2 = 100;
```

第一行程式碼由於 100 在-128 到 127 間，會從快取中傳回 Integer 實例，第二行程式碼執行時，要包裹的同樣是 100，也是從快取中傳回同一 Integer 實例，i1 與 i2 會參考到同一個 Integer 實例，使用==比較就會是 true。如果是這個程式碼：

```
Integer i1 = 200;
Integer i2 = 200;
```

第一行程式碼由於 200 不在-128 到 127 間，直接建立 Integer 實例，第二行程式碼執行時，也是直接建立新的 Integer 實例，i1 與 i2 不會參考到同一個 Integer 實例，使用==比較就會是 false。

IntegerCache.low 預設值是 -128，執行時期無法更改，IntegerCache.high 預設值是 127，可以於啟動 JVM 時，使用系統屬性 **java.lang.Integer.IntegerCache.high** 來指定。例如：

```
> java -Djava.lang.Integer.IntegerCache.high=300 cc.openhome.Demo
```

如上指定之後，Integer.valueOf()就會針對-128 到 300 範圍中建立的包裹器進行快取，而針對先前 i1 與 i2 包裹 200 時，使用==比較的結果，就又顯示 i1 == i2 了。

在 IDE 中，也可以指定 JVM 啟動時可用的一些引數。例如在 Eclipse 中，可以如下操作進行設定：

1. 在想執行的原始碼上按右鍵，執行選單「Run as/Run Configurations…」。
2. 在出現的「Run Configurations」對話方塊中，選擇「Java Application」節點按右鍵，執行選單「New Configuration」。
3. 在新建的組態項目中切換至「Arguments」，在「VM arguments」中輸入「-Djava.lang.Integer.IntegerCache.high=300」，按下「Apply」完成設定。

如上設定之後，每次在該原始碼上按右鍵，執行選單「Run as /Java Application」時，就會套用「VM arguments」的設定。

因此結論就仍是 4.1 談過的，**別使用==或!=來比較兩個物件實質內容值是否相同（因為==與!=是比較物件參考），而要使用 equals()**。例如以下的程式碼：

```
Integer i1 = 200;
Integer i2 = 200;

if (i1.equals(i2)) {
    System.out.println("i1 == i2");
}
else {
    System.out.println("i1 != i2");
}
```

無論實際上 i1 與 i2 包裹的值座落在哪個範圍，只要 i1 與 i2 包裹的值相同，equals()比較的結果就會是 true。