程式碼實作學習

Java Steps: Learning Java Step by Step

Release 1.0

董少桓、林彥宏

September 30, 2011

CONTENTS

CHAPTER

ONE

簡介

1.1 Java 程式語言的特色

Java 程式語言起源於 1991 年，Green Team 軟體開發團隊用它來開發 Star 7 機器上的應用程式，當時設計此語言的 James Gosling 因為看見窗外的「橡樹（oak）」，決定將新語言命名為 Oak 。但是由於工程師們喜歡邊喝咖啡邊討論，隨後又將名稱改為 Java（一種咖啡的名稱），這個名稱就一直沿用到現在。

Java 原本是為了控制冰箱、冷氣、微波爐等家電用品而設計的程式語言。由於家電用品相當多樣，因此 Java 選用了一個與傳統的程式語言不一樣的執行模式：

- 傳統的程式語言在編譯後會產生 machince code（機器碼），然後直接在硬體上執行；
- Java 在編譯後則會產生 Byte Code 並間接的在 Java Virtual Machine（JVM）上執行。這個 JVM（Java 虛擬機器）其實是一個軟體，其功用是解譯並執行 Byte Code，而 JVM 仍然是在硬體上執行。

因為 JVM 是軟體，所以 Java 也有跨平台的特性：只要為不同的處理器或作業系統設計其專屬的 JVM，Java 的程式便可以不需改寫的在這些處理器或作業系統上執行。這便是「Write once, runs everywhere 或一次編譯、到處執行」的由來。

Java 也支援物件導向程式設計（Object-Oriented Programming），所謂「物件」，簡單的說有「屬性」也有「方法」，例如冷氣機的「屬性」可以包括：「開關」及「溫度」；而「方法」則可以包括：「開機」、「關機」及「設定溫度」等。為了讓程式設計師，可以比較容易的使用物件撰寫模擬、控制與應用電腦本身（如滑鼠與鍵盤等也是物件）和我們生活周遭的物件的程式，因此便有研究人員發明了支援物件導向程式設計的語言。

Java 的設計搭上了全球資訊網的順風車，原因是 Java 的設計團隊可以寫一個能夠在瀏覽器中執行的 JVM，而讓 Java 的程式可以透過網路下載至瀏覽器中執行。這個「網路」＋「物件導向」的特性讓 Java 瞬間爆紅。

除了跨平台、物件導向、可透過網路動態的載入及執行程式等功能之外，Java 還支援多執行緒、例外狀態處理與自動記憶體回收的功能：

- 多執行緒讓一個程式可以執行數個工作；
- 例外狀態處理讓處理例外的程式碼也能夠物件化；
- 自動記憶體回收則讓程式設計師免除了使用低階的指標（pointers）來設計資料結構及管理記憶體的負擔。這個特色成了 C 語言程式設計師的福音，因為它可以為程式設計師減少許多不容易 debug 的錯誤。

1.2 安裝 JDK

在編譯與執行 Java 程式前，你的電腦必須先安裝 JDK（Java Development Kit）：

- 可以從這裡下載新版的 JDK http://java.sun.com/javase/downloads/index.jsp

在安裝完成後，也需要完成 path 及 classpath 的設定：

```
path=C:\Program Files\Java\jdk1.6.0\bin;....
classpath=.;C:\Program Files\Java\jdk1.6.0\lib;....
```

請注意：以上路徑中的 jdk1.6.0 會因版本的不同而異。此外，在設定 classpath 時要特別注意在＝號的右邊要輸入這個 「.」 。這個點的意義是目前的目錄（current directory），也是執行 Java 程式時用來搜尋執行檔的目錄。

1.3 編譯及執行 Java 程式

有兩種方式可以編譯及執行一個 Java 程式。第一種是使用程式開發環境（program development environment），例如： Eclipse ；另一種則是使用一般的程式編輯器。以下是使用「記事本」寫 Java 程式時所需要進行的三個步驟：

1. 使用「記事本」輸入以下的程式並將檔案命名為 EnglishExam.java （注意：附檔名必須是.java 而不是.txt， **而這個檔案的主檔名必須與 public class 後面的 EnglishExam 相同** ）：

```
public class EnglishExam {
        public static void main(String argv[]) {
                System.out.println("Your score is 97.");
        }
}
```

2. 執行「命令提示字元」並將目錄切換至儲存 EnglishExam.java 的目錄，然後執行：

```
javac EnglishExam.java
```

執行這個 javac 指令就是執行 Java 的編譯器（compiler），其結果是在同樣的目錄產生一個 EnglishExam.class 的 byte code 檔。

3. 上面的步驟如果有編譯錯誤則繼續修改程式。如果沒有編譯錯誤則可以執行：

```
java EnglishExam
```

執行後 Your score is 97。便會顯示在螢幕上。

在開發 Java 程式的過程中，有可能發生編譯錯誤（compile-time error）。這時便需要再次的使用編輯器修改錯誤，直到沒有任何的編譯錯誤為止。編譯完畢之後，在程式執行時也有可能發生 run-time error。同樣的，這時也需要使用編輯器修改、編譯、執行、除錯，直到沒有錯誤為止。

一般的 Java 程式都是由一或多個類別（class）所組成，**其中的一個類別至少要有一個命名為 public static void main 的方法（method），而這個程式就是由 main 開始執行**。（透過網路瀏覽器執行的 Java applet 不適用此規則。）

上例中的 public class EnglishExam 是指定 EnglishExam 這個類別是 public 是公用的，也就是可以被程式中其他的類別引用。而 public static void main(String argv[]) 的意義是：

1. public：指定 main 為一個可以被其他類別使用的 public method；
2. static：指定 main 為一個類別方法（static method），一個類別方法隸屬於一個 class；
3. void：代表 main 執行完畢後回傳的型態，因為 main 沒有回傳任何數值，因此它的回傳型態是 void；
4. String argv[]：指這個方法的輸入參數是 argv[] 而 String 則是它的型態。main 的輸入參數 String argv[] 可以在執行一個 Java 程式時將字串（String）資料輸入這個程式。例如在編譯以下的程式之後：

```
public class HelloJava {
        public static void main(String argv[]) {
                System.out.println("Hello " + argv[0] + argv[1]);
        }
}
```

以「命令提示字元」執行：

```
java HelloJava Basic C++
```

便會呼叫 System.out.println 並輸出：

```
Hello Basic C++
```

這個程式的 argv[] 代表 argv 這個變數是一個陣列，而 argv[0]、argv[1] 則取用 argv 內第 0、1 個儲存格的內容。

Java 程式中 **用大刮號 { } 標示的 Block（區塊）** 是用來組織程式層次關係的語法。

例如上例的程式就有兩個區塊，一組用來標示 class 的區塊，另一組則用來標示 main 的區域。區塊中可以包含其他的區塊，在撰寫程式時也應注意要把區塊的內容往右縮排。一組用來標示類別的區塊內，可以有數個變數與方法。而一組用來標示方法的區塊內可以有一或多句以「;」結束的程式碼。這些程式碼共同構成了這個方法的 body。

為 Java 程式中使用的名字命名，有一個不成文的規定： **類別名稱的第一個字母要用大寫 。 方法或變數的第一個字母則是小寫** ，若有數個字合併時則 **後續的字的第一個字母也習慣用大寫** 。

動手練習：修正程式碼的錯誤

CHAPTER

TWO

JAVA 的變數、方法與型態

一個電腦程式基本上是由兩部分組成：

1. 資料：這部分的程式碼要為所處理的資料命名、指定其型態、並存放於記憶體中；
2. 處理：這部分的程式碼要使用撰寫在方法（method）中的運算式、條件判斷式、迴圈，來存取資料、計算結果、然後輸出。

這兩部分的內容（程式碼），不但是機器要知道如何執行，更要讓人（程式設計師）容易寫，也容易讀。而讓程式碼易寫、易讀的最基本的方法，就像收拾家中的衣櫃一般：按照種類與性質，分別放置整齊。

Java 語言整理程式碼的最基本的單元稱做「類別」。一個類別可以有儲存資料的「變數」與負責處理資料的「方法」。

「類別」可以提供兩種整理程式碼的方式：一種是以程式的邏輯結構當作分類的方式（俗稱的結構化程式設計）；另一種則是以物件的種類來歸類。而有的時候，一個類別也可以同時提供這兩種方式。

以程式的邏輯結構（例如將迴圈的邏輯放入一個方法中，或將一個判斷分數高低的邏輯放入一個方法中）來分類時，所使用到的變數稱為類別變數或 static field，而所用到的方法稱為類別方法或 static method。稱做 static（靜態）的原因是，這類的變數與方法是在程式開始執行時便在記憶體中產生了，而且它們的壽命一直到程式結束時才結束。

以物件來歸類的話，則是將類別內的程式碼，看成是產生這種類別的物件的「規格」。由於只是規格，所以只有在使用這個類別製造物件時，所對應的記憶體才會產生。而這種在程式執行時「動態」產生的物件實例中的變數與方法，稱為實例變數（instance variable），與實例方法（instance method）。

類別方法或實例方法，也需要自己有儲存資料的地方，而在類別方法或實例方法中儲存資料的變數，都叫區域變數（local variable），意思是在一個方法所屬的區域內才可以使用的變數。

Java 如何分區呢？主要是使用大刮號，例如：

```
{
        {
                ...
        }
        {
                ...
        }
        ...
}
```

程式語言的句子與一般說話的語言一樣是由基本的字詞（names）組合而成。Java 為這些字詞命名的規定是一個字詞可以包含一個或多個英文字母、數字、_ 及 $ 所組成的字元，而第一個字元不可以是數字。

Java 語言有 public, class, void, if, while, for 等 保留字 。除了開始認識這些保留字的意義與用法之外，程式設計師所要學習的第一件事，就是為儲存資料的 **變數** 與執行處理的 **方法** 命名。

變數（variable） 是程式中的一種字詞。一個變數有一個 **名字（name）** 、一個 **資料值（value）** 、一塊儲存資料值的 **記憶體** 以及這個資料值的 **型態（type）** （如 int、double 等）。由於一個變數的型態，定義了這個變數的值所需要的記憶體的大小，所以一個 Java 程式在編譯時，就可以知道如何為這些變數，在執行時，配置適當大小的記憶體空間，以存放這些變數的值。

整的來說，Java 有三種變數：

1. **區域變數（local variable）** ：宣告在方法內或參數部分的變數；
2. **類別變數（class variable or static field）** ：在一個類別中以 static 宣告的變數；
3. **實例變數（instance variable or non-static field）** ：在一個類別中沒有使用 static 宣告的變數。

Java 也有兩種方法：

1. **類別方法（class method or static method）** ：這種方法以 static 宣告。呼叫的方式是 C.m(...)，其中 C 是類別名稱，m 是方法名稱，... 則是 0 至多個傳入的參數。
2. **實例方法（instance method or non-static method）** ：這種方法不以 static 宣告，隸屬於一個類別所產生的實例。呼叫的方式是 o.m(...)，其中 o 是這個類別或其子類別的實例，而 m 是其方法名稱，... 則是 0 至多個傳入的參數。

Java 之所以有種類這麼多的變數與方法，是因為 Java 同時支援結構化（例如：C 與 Basic）與物件導向兩種普遍使用的程式設計方式。撰寫結構化程式時需要使用類別變數與類別方法。類別變數在概念上與結構化程式語言的全域變數（global variable）一

致；而類別方法在概念上則與結構化程式語言的函式（function）或程序（procedure）一致。實例變數、實例方法，則屬物件導向程式設計的功能。一般的 Java 程式可以同時使用結構化與物件導向並存的方式設計程式。

這份講義介紹 Java 結構化程式設計的語法及語意，另一份講義《Java 物件導向程式設計》則介紹 Java 物件導向程式設計的功能。

此外，Java 變數的型態也有兩大類：

1. **primitive type** ，包括： int、double、boolean、char 等。
2. **reference type ，包括：**

    (a) 類別型態：經由類別（class）的宣告而得到。如果 Car 是一個類別，而 aCar 是一個這個類別的變數，則 Car 便是 aCar 的型態（type）。之所以稱為 reference type，是因為 aCar 這個變數在記憶體中的位置，實際上是存著指向（reference）一個 Car 實例的地址。

    (b) 介面型態：經由介面（interface）的宣告而得到。

    (c) 陣列（array）型態。

    (d) enum 型態：一種特別的類別宣告方式，用於宣告月份、一週的七天等。

CHAPTER

THREE

區域變數與基本資料型態

區域變數是一個方法的參數或是宣告在一個方法的區塊中。以下是宣告區域變數的幾個範例，其中 int 代表整數，而 double 代表倍精準浮點數，宣告的意義是告訴編譯器一個變數的型態是什麼：

```
public class EnglishExam {
        public static void main(String argv[]) {
                // argv 是傳入參數，同時也是一種區域變數

                // 宣告三個區域變數
                int vocabulary;
                int grammar;
                int listening;
                ...
        }
}
```

多個變數的宣告，可以合併在一行：

```
public class EnglishExam {
        public static void main(String argv[]) {
                // 將以上三個區域變數，合併在一行
                int vocabulary, grammar, listening;
                ...
        }
}
```

宣告變數時，也可以同時指定數值：

```
public class EnglishExam {
        public static void main(String argv[]) {
                // 在宣告時也將數值存入這三個區域變數中
                int vocabulary = 24;
                int grammar = 26;
                int listening = 33;
                ...
```

```
        }
}
```

```
public class EnglishExam {
        public static void main(String argv[]) {
                // 也可以這樣寫：
                double vocabulary = 22.5, grammar = 25.4, listening = 32.0;
                ...
        }
}
```

有了變數之後，可以使用設值運算符號（=），將數值存入區域變數中，如果一個區域變數尚未被存入數值，則其預設值（default value）會被存入，而數字的預設值是 0。例如：

```
public class EnglishExam {
        public static void main(String argv[]) {
                int vocabulary, grammar, listening;
                int score;

                vocabulary = 22;
                grammar = 26;
                score = vocabulary + grammar + listening;

                System.out.print("The score of the exam is ");
                System.out.println(score);
                // listening 的預設值是 0, 所以印出 48
        }
}
```

以上程式碼執行的結果為：

```
The score of the exam is 48
```

Java 的註解是以 **//** 或 **/* */** 表示，例如：

```
// 這是註解
/*
   這也是註解
   這還是註解
*/
```

CHAPTER

FOUR

螢幕輸出及鍵盤輸入

螢幕輸出有幾種方式。第一種是前面章節已經使用過的 System.out.print 及 System.out.println。這兩種方法的的差別是前者沒有換行，而後者有換行。如果有數個資料需要一起印出時，則可以使用 + 進行串接。例如：

```
public class Show {
        public static void main(String argv[]) {
                int design, acting;

                design = 3;
                acting = 5;
                System.out.println( "Design is " + design + "and acting is " + acting );
        }
}
```

第二種方式是在 J2SDK5 之後才支援 [1] 。這個方式與 C 語言的 printf 功能類似。例如：

```
System.out.printf("Today is %s, %d.\n", "January", 18);
// %s 的位置替換成 January 這個 String
// %d 的位置替換成 18 這個整數
// \n 表示換行符號
```

顯示：

```
Today is January, 18
```

例如：

```
double score = 92.345
System.out.printf("My score is %.2f.\n", score);
// %.2f 的意義是小數點以下取兩位，並四捨五入。
System.out.printf("My score is %6.2f.%n", score);
// %6.2f 的意義是：包括小數點共 6 位，小數點以下取兩位，
// 並四捨五入。所以 9 的左邊多空一格。
```

[1] System.out.printf(), http://www.java2s.com/Code/JavaAPI/java.lang/System.out.printf.htm

顯示：

```
My score is 92.35.
My score is  92.35.
```

鍵盤輸入則可以透過 [2] 。例如：

```
// 使用時先載入 Scanner 所屬的 package
import java.util.*;  // * 的意義是 java.util 內所有的類別

// 定義物件：
Scanner scanner = new Scanner(System.in);

// 輸入字串：
String name = scanner.nextLine();

// 輸入整數：
int score = scanner.nextInt();

// 輸入 double
double height = scanner.nextDouble();

// 輸入 float
float weight = scanner.nextFloat();
```

以下是一個完整的範例：

```
import java.util.*;

public class EnglishExam {
        public static void main(String argv[]) {
                int vocab, grammar, listen, score;

                Scanner scanner = new Scanner(System.in);
                String name = scanner.nextLine();
                vocab = scanner.nextInt();
                grammar = scanner.nextInt();
                listen = scanner.nextInt();
                score = vocab + grammar + listen;
                System.out.printf("The total score of %s is %d.%n", name, score);
        }
}
```

[2] java.util.Scanner, http://www.java2s.com/Code/JavaAPI/java.util/Scanner.htm

CHAPTER

FIVE

算術運算式

使用算術運算式要注意的是運算的優先次序。例如：先乘除、後加減。如果不記得運算的優先次序，那麼最簡便的方法是使用（）也就是指定內層的刮號先執行。以下是算術運算式的幾個範例：

使用算術運算式另一項要注意的是型態的轉換，也就是在一個算術式中同時有整數與浮點數時，Java 會將整數先轉換成浮點數然後再進行運算。

例如：

```
int i = (3 + 4) /  **3**  // 兩個整數相除，結果仍是整數，小數部分捨棄。傳回 2
double f;
f = (3 + 4) /  **3.0**    // 7 轉型成浮點數，然後與浮點數 3.0 相除。傳回 2.333...
System.out.println(i);
System.out.println(f);
```

印出：

```
2
2.3333333333333335
```

如果需要指定轉換的型態，則可以使用以下的方式：

至於 i、f 的原來的值還是不變。

如同一般的算術運算式，設值運算式（=）也會傳回值。例如：

```
x = 3          // 傳回 3
y = (x = 3)    // 因為 x = 3 傳回 3, 所以 y 的值也設成 3
```

自動將資料範圍較小的型態轉為資料範圍較大的型態，稱為自動轉型 (promotion)。Java 資料型態範圍之大小次序為：

```
byte < short < int < long < float < double
```

以下的例子會出現 possible loss of precision 的錯誤：

這個錯誤是因為 a 會自動轉型成較大的 long 再跟 b 相加，但是 int 型態的 c 放不下 long 的資料。此時就要透過強制轉型 (casting)，將 a 轉型成 int：

```
int c = (int)(a + b);
```

當資料型態由小轉為大時，會自動轉型；當資料型態由大轉為小時，則需強迫轉型。

CHAPTER

SIX

類別變數與類別方法

非物件導向程式語言（例如：C），程式設計師主要是使用函式（function、全域變數與區域變數將一個大的程式分割成幾個小的部份，以簡化程式的撰寫。這些觀念在 Java 中仍然可以使用，而使用的方式是透過類別變數與類別方法。

舉例而言，如果要為英文檢定考試寫一個計算成績的程式，那麼這個程式應該有一個計算成績的方法。例如：

```
public class EnglishExam {
        public static int englishScore(int v, int g, int l) {
                return v + g + l;
        }
        public static void main(String argv[]) {
                System.out.print("The score of the exam is ");
                System.out.println(englishScore(24, 27, 32));
        }
}
```

執行結果：

```
The score of the exam is 83
```

public static int englishScore(int v, int g, int l) 定義了 englishScore 這個類別方法，這個方法有三個命名為 v, g, l 的輸入參數，它們的型態都是 int。這個方法的輸出型態也是 int，也就是會使用 return 傳出一個 int 的值 (v + g + l)。Java 使用 static 這個保留字來定義類別方法。因為這種方法是靜態的，也就是在程式執行時，呼叫這個方法的程式碼，一定都會執行相同的方法。

在 main 中的 System.out.println(englishScore(24, 27, 32)) 將 24, 27, 32 傳入 englishScore 中，並依序成為 v, g, l 三個輸入參數的值，而這三個數相加的結果 83 會繼續的傳入 System.out.println，然後顯示在螢幕上。一個方法的輸入參數也是那個方法的區域變數。所以 v, g, l 三個輸入參數也是 englishScore 的區域變數。

除了直接將數值傳入方法中以外，還可以將變數或其他也有傳回值的式子，寫在方法

呼叫中傳入的位置。例如：

```
public class EnglishExam {
        public static int englishScore(int v, int g, int l) {
                return v + g + l;
        }
        public static void main(String argv[]) {
                int a = 3, b = 4;
                System.out.print("The score of the exam is ");
                System.out.println(englishScore( **a, b, a + b** ));
        }
}
```

執行結果：

```
The score of the exam is 14
```

Java 會在得到 a, b, a + b 的數值後，才將 3, 4, 7 傳入 englishScore 中。也就是先得到數值再傳入，然後 v, g, l 便使用傳入的數值生成三個區域變數。這個特性稱為 **call-by-value 或傳值呼叫** 。v, g, l 三個參數之所以也是區域變數，因為這三個變數的可見範圍（scope）只包含 englishScore 的區塊。

如果一個方法沒有傳回值，那麼這個方法的輸出型態便是 void。例如：

```
public class EnglishExam {
        public static  **void displayScore** (int v, int g, int l) {
                System.out.print("The score of the exam is ");
                System.out.println(v + g + l);
        }

        public static void main(String argv[]) {
                displayScore(24, 27, 32);
        }
}
```

執行結果：

```
The score of the exam is 83
```

displayScore 這個方法將字串顯示在螢幕上，不需要傳回值，因此它的輸出型態是宣告成 void，而 main 的輸出型態也是 void。

不同的類別中也可以定義同名的方法。這個功能稱做 **overloading** 。而 Java 是以 **類別名稱. 方法名稱 (0 或多個參數)；** 呼叫宣告在不同類別的類別方法。例如：

```
public class Exam {
        public static void main(String argv[]) {
                int voc = 3, grammar = 7, listen = 8;
                System.out.print("The score of the english exam is ");
                System.out.println(EnglishExam.displayScore(voc, grammar, listen));
```

```
                System.out.print("The score of simple english exam is ");
                System.out.println(SimpleEnglishExam.displayScore(voc, grammar, listen));
        }
}

class EnglishExam {
        public static int displayScore(int v, int g, int l) {
                return v + g + l;
        }
}

class SimpleEnglishExam {
        public static int displayScore(int v, int g, int l) {
                return v + g + 0;
        }
}
```

執行結果：

```
The score of the english exam is 18
The score of simple english exam is 10
```

一個類別中也可以有同名的方法，但是他們必須有不同的輸出入型態。例如：

```
public class Exam {
        public static void main(String argv[]) {
                int voc = 3, grammar = 7, listen = 8;
                System.out.print("The int score of the exam is ");
                System.out.println( **EnglishExam.displayScore(3, 7, 8)** );
                System.out.print("The double score of the exam is ");
                System.out.println( **EnglishExam.displayScore(3.0, 8.0, 7.0)** );
        }
}

class EnglishExam {
        public static int  **displayScore(int v, int g, int l)**  {
                return v + g + l;
        }
        public static double  **displayScore(double v, double g, double l)**  {
                return v + g + l;
        }
}
```

執行結果：

```
The int score of the exam is 18
The double score of the exam is 18.0
```

另一個 overloading 的例子是：十。十可以用來將數字相加，也可以將字串合併。例如：

```
int a = 4, b = 5;
System.out.print(3 + a + b);
```

執行結果：

```
12
```

例如：

```
.. code-block:: java
```

String a = ``xy", b = ``Z"; System.out.print(``3" + a + b);

執行結果：

```
3xyz
```

使用類別方法在程式中有許多好處：

1. 增加程式碼的再用性：同樣的計算步驟，只需要透過呼叫類別方法便可以重複使用。
2. 讓程式碼的細節，被隱藏在類別方法中：程式設計師在完成類別方法的撰寫後，便只需要知道那個類別方法的輸入、輸出與功用即可，而不用擔心執行的細節。
3. 容易除錯：除錯的過程可以一個類別方法、一個類別方法的進行，容易找出錯誤的根源。
4. 容易擴充類別方法內程式碼的功能：只要在類別方法內擴充其功能，而不用在每次呼叫時都重複的擴充。例如以下的程式碼擴充了成績的計算方式，所有 displayScore 的呼叫的計算結果都同步改變：

除了使用 static 宣告類別方法外，還有也是使用 static 宣告的類別變數。以下是一個在程式中內建三筆考試成績的資料，呼叫 displayScore 計算成績後，將三筆資料加總並存入 total 這個類別變數中的範例：

```
.. code-block:: java
```

public class Exam { public static int total = 0;

public static void main(String argv[]) { total = displayScore(3, 4, 5); // total = 12 total = total + displayScore(4, 5, 6); // total = 27 total = total + displayScore(1, 2, 3); // total = 33 System.out.print(``The total score is "); System.out.println(total);

}

public static int displayScore(int v, int g, int l) { return v + g + l;

}

```
    }
```

執行結果：

```
The total score is 33
```

程式設計師也可以使用不是定義在自己類別中的類別變數，而 Java 是以 **類別名稱. 變數名稱** 使用定義在其他類別中的類別變數。以下便是一種將 total 宣告在另一個類別 EnglishExam 中的寫法是：

```
public class Exam {
        public static void main(String argv[]) {
                EnglishExam.computeScore(3, 4, 5);
                EnglishExam.computeScore(4, 5, 6);
                EnglishExam.computeScore(1, 2, 3);
                System.out.print("The total score is ");
                System.out.println(EnglishExam.total);
        }
}

class EnglishExam {
        public static int total = 0;

        public static void computeScore(int v, int g, int l) {
                total = total + (v + g + l);
        }
}
```

執行結果：

```
The total score is 33
```

以下則是一個為考試成績的計算，加入權重的範例。在這個範例中是以 Exam.wV, Exam.wG, Exam.wL 來使用這三個類別變數：

```
public class Exam {

        public static double wV = 0.3, wG = 0.3, wL = 0.4;

        public static void main(String argv[]) {
                int voc = 3, grammar = 7, listen = 8;
                System.out.print("The score of the english exam is ");
                System.out.println(EnglishExam.displayScore(voc, grammar, listen));
                System.out.print("The score of simple english exam is ");
                System.out.println(SimpleEnglishExam.displayScore(voc, grammar, listen)
        }
}

class EnglishExam {
```

```
        public static double displayScore(int v, int g, int l) {
        return v * Exam.wV + g * Exam.wG + l * Exam.wL;
        }
}

class SimpleEnglishExam {
        public static double displayScore(int v, int g, int l) {
                return v * Exam.wV + g * Exam.wG + 0;
        }
}
```

執行結果：

```
The score of the english exam is 6.2
The score of simple english exam is 3.0
```

類別變數與區域變數，在變數的可用「區域」與存在的「時間」上都不相同。類別變數若是定義為 public，則它的可用區域便包括整個程式，而且在整個程式執行時都存在。區域變數則是在程式執行到一個區塊或方法內時，那個區塊或方法的區域變數才存在，一旦離開那個區塊或方法，便消失了。因此區域變數的可用區域，只在定義該區域變數的區塊或方法內。

以下是一個「計算蛋與水果總價」的程式及其執行過程的動畫：

```
class Market {
        static int sEgg = 5, sFruit = 20;
                static int getMoney(int nEgg, int nFruit) {
                return sEgg * nEgg + sFruit * nFruit;
        }
}

public class Ex1 {
        public static void main(String argv[]) {
                int egg = 20, fruit = 30;
                System.out.print("Money:");
                System.out.println(Market.getMoney(egg, fruit));
        }
}
```

執行結果：

```
Money:700
```

Media:Ex1new.swf 觀看執行過程

CHAPTER

SEVEN

運算式、句子與條件判斷句

運算式（expression）執行完畢會後傳回一個值；句子（statement）在執行之後，則不回傳值。一個運算式與句子在執行時都可能會改變電腦的狀態，例如：更改變數、陣列或檔案的內容。算術運算式已經介紹了，而最簡單的句子，包括：

1. expression statement：在運算式後加一個";" 。
2. block statement：在 { } 內可以有 0 或多個變數宣告或句子。

這兩種句子，也都已經看過範例。這個單元將介紹關係運算式、邏輯運算式及 if、switch 等句子。

7.1 關係運算式

以下是在寫關係運算式（relational expression）時常常會使用到的關係運算子（relational operator），整理如下：

- 需注意關係運算式使用的 ``==" ，和寫設值運算式（assignment expression）的 ``=" 是不一樣的。

7.2 邏輯運算式

Java 常用的邏輯運算子（logical operators）有三個，分別是 && (AND)、|| (OR)、! (NOT)。邏輯運算式使用邏輯運算子以連結二個或以上的關係運算式。

- && (AND)

&& 是「AND（且）」運算：左右二邊「都」為 true，結果為 true；其餘情況其結果都為 false。

如果左邊為 false，則系統就不會去執行右邊；因為結果必為 false；既然左邊已經為 false，不管右邊結果為何，結果一定為 false。

- || (OR)

|| 是「OR（或）」運算，左右二邊只要有一邊為 true，結果為 true；只有左右二邊都為 false，結果才為 false。

如果左邊為 true，則系統就不會去執行右邊；因為結果必為 true；即然左邊已經為 true，不管右邊結果為何，結果一定為 true。

- ！(NOT)

! 是「NOT（相反）」運算，其結果為! 右邊的相反。

7.3 運算子的優先順序

以下是在在寫運算式時常常會使用到的運算子的優先順序，整理如下：

優先順序	運算子
1	**(** (左括號) **)** (右括號) **++** (左遞增) **--** (左遞減)
2	**+** (正) **-** (負) **!** (NOT) **++** (右遞增) **--** (右遞減)
3	***** (乘) **/** (除) **%** (取餘數)
4	**+** (加) **-** (減)
5	**<** (小於) **<=** (小於等於) **>** (大於) **>=** (大於等於)
6	**==** (等於) **!=** (不等於)
7	**&&** (AND)
8	**||** (OR)
9	**?:** (條件判斷)

上表雖然不易記憶，然而，由於 ``(" 與")" 的優先順序是最優先，所以在程式的設計過程中，我們可以藉由使用左、右括號來指定運算式的優先順序。

立即練習：

```
a = 48
b = 32
c = 55

d = a>b?a+b++*--c:a-b--*++c
```

1. 請問 d 的值是多少？

上面的程式碼，讀者可能需要查閱運算子優先次序，才能計算輸出的值是多少，雖然對電腦來說，可以很快就按照規則計算出正確的數字，但畢竟程式碼是由「人」撰寫

和維護，要記住這些瑣碎的規則可是一點都不容易。

請再嘗試以下的練習：

1. 在上一個練習的程式碼加上 ( ) 括號，讓你自己可以很容易看懂計算的優先次序。

7.4 Java 條件判斷句的種類

Java 的條件判斷句（statement）可以分成以下 3 種：

1. if...
2. if... else...
3. switch

7.4.1 if 條件判斷句

Java 語言 if (...) {...} 條件判斷句的流程圖及語法如下：

Image If-1.png

```
1 if (/* 條件判斷 */) {
2   //條件判斷成立時執行的程式碼
3 }
```

說明：

1. 如果第 1 行的條件判斷成立（值 == true），則會執行第 2 ~ 4 行的程式碼（圖形藍色的路徑）。而如果條件判斷不成立（值 == false），則會跳過第 2 ~ 4 行程式碼不執行（圖形黑色 False 的路徑）。
2. 如果第 1 行的條件判斷成立時，執行的程式碼只有一行，則區塊符號可以省略不寫。如果超過一行，其區塊符號 { } 不可以省略。

Example 1：請設計一個 Java 程式，讓使用者自行輸入一個成績，判斷輸入的成績是否「大於或等於」60 ，如果是就輸出「成績及格！」。

Image If-2.png

```
1 import java.util.Scanner;
2 class if_loop {
3   public static void main(String[] args) {
4     if_condition();
5   }
6
```

```
7     public static void if_condition() {
8       System.out.print(" 請輸入成績:");
9       Scanner scanner = new Scanner(System.in);
10      int grade = scanner.nextInt();
11
12      if (grade >= 60) {
13          System.out.println(" 成績及格!");
14      }
15    }
16  }
```

執行結果：

```
請輸入成績:60
成績及格!
```

說明：

- 輸入的 grade 為 60 ，條件判斷的結果為 true（60 >= 60），會執行條件判斷成立區塊內的程式碼，輸出「成績及格!」（圖形藍色的路徑）。
- 因為範例中條件判斷成立時，執行的程式碼只有一行：System.out.println(``成績及格!"); 所以區塊符號可以省略不寫。

7.4.2 if...else...條件判斷句

Java 語言 if (...) {...} else {...} 條件判斷句的語法如下：

Image If-else-1.png

```
if (條件判斷)
{
  條件判斷成立時執行的程式碼;
  ....
}
else
{
  條件判斷不成立時執行的程式碼;
  ....
}
```

說明：

- 如果第 1 行的條件判斷成立（值 == true），則執行第 2 ~ 5 行區塊符號內的程式碼（圖形藍色的路徑）。
- 如果第 1 行的條件判斷不成立（值 ==false），則跳過第 2 ~ 5 行的程式碼，而執行第 7 ~ 10 行區塊符號內的程式碼（圖形紅色的路徑）。

- 如果 if 條件判斷成立或不成立時，執行的程式碼只有一行，則該區塊符號可以省略不寫。而如果超過一行，則區塊符號不可以省略。
- if 和 else 是彼此互斥的關係，二個條件區塊在程式執行的過程中，只會選擇一個條件區塊去執行。

Example 2：請設計一個 Java 程式，讓使用者輸入一個成績，判斷該輸入的成績是否「大於或等於」60 ，如果是就輸出「成績及格!」；如果不是則輸出「成績不及格!」。

Image If-else-2.png

```
import java.util.Scanner;

class if_else_loop {
  public static void main(String[] args) {
    if_else_condition();
    if_else_condition();
  }

  public static void if_else_condition() {
    System.out.print(" 請輸入成績:");
    Scanner scanner = new Scanner(System.in);
    int grade = scanner.nextInt();

    if (grade >= 60) {
      System.out.println(" 成績及格!");
    }
    else {
      System.out.println(" 成績不及格!");
    }
  }
}
```

執行結果：

```
請輸入成績:88
成績及格!
請輸入成績:59
成績不及格!
```

說明：

- 第一次輸入的 grade 為 88 ，其條件判斷的結果為 True（88 >= 60），執行 if 成立區塊內的程式碼，輸出「成績及格!」（圖形藍色的路徑）。
- 第二次輸入的 grade 為 59 ，其條件判斷的結果為 False（59 >= 60），執行 else 區塊內的程式碼，輸出「成績不及格!」（圖形紅色的路徑）。
- 因為 if 條件判斷成立和不成立時，執行的程式碼都只有一行，所以二者的區塊符

號都可以省略不寫。

7.4.3 數個 if 的條件判斷

幾個接續在一起的 if 句子，可以寫成：if (...) {...} else if (...) {...} else {...}：

[[Image:If-elseif-else-1.png]]

```
1  if (條件判斷 1) {
2    條件判斷 1 成立時執行的程式碼;
3    ....
4  }
5  else if (條件判斷 2) {
6    條件判斷 2 成立時執行的程式碼;
7    ....
8  }
9  :
10 :
11 else {
12   上述條件判斷都不成立時執行的程式碼;
13   ....
14 }
```

說明：

- 如果在第 1 行的條件判斷 1 成立（值 == true），則執行第 2 ~ 5 行區塊符號內的程式碼（圖形藍色的路徑），並略過其餘的程式碼。
- 如果在第 6 行的條件判斷 2 成立（值 == true），則執行第 7 ~ 10 行區塊符號內的程式碼（圖形綠色的路徑），並略過其餘的程式碼。
- 如果所有的條件判斷都不成立（值 == false），則跳過第 1 ~ 12 行的程式碼，而執行第 14 ~ 17 行區塊符號內的程式碼 (圖形紅色的路徑) 。
- if 、else if 和 else 是彼此互斥的關係，所有條件區塊在程式執行的過程中，只會選擇一個條件區塊去執行。

Example 3：請設計一個 Java 程式，讓使用者自行輸入一個成績，判斷該輸入的成績是屬於 A, B, C, D 或 E 。

[[Image:If-elseif-else-2.png]]

```
1  import java.util.Scanner;
2  class if_elseif_else_loop {
3      public static void main(String[] args) {
4          if_elseif_else_condition();
5          if_elseif_else_condition();
6          if_elseif_else_condition();
```

```
        if_elseif_else_condition();
        if_elseif_else_condition();
    }

     public static void if_elseif_else_condition() {
        System.out.print(" 請輸入成績:");
        Scanner scanner = new Scanner(System.in);
        int grade = scanner.nextInt();

        if (grade >= 90)
            System.out.println(" 成績為 A");
        else if (grade >= 80)
            System.out.println(" 成績為 B");
        else if (grade >= 70)
            System.out.println(" 成績為 C");
        else if (grade >= 60)
            System.out.println(" 成績為 D");
            else
            System.out.println(" 成績為 E");
    }
}
```

執行結果：

```
請輸入成績:90
成績為 A
請輸入成績:89
成績為 B
請輸入成績:70
成績為 C
請輸入成績:60
成績為 D
請輸入成績:59
成績為 E
```

說明：

- 第一次輸入的 grade 為 90 ，符合條件判斷 1，輸出 ``成績為 A" (圖形藍色的路徑) 。
- 第二次輸入的 grade 為 89 ，符合條件判斷 2，輸出 ``成績為 B" (圖形綠色的路徑) 。
- 第三次輸入的 grade 為 70 ，符合條件判斷 3，輸出 ``成績為 C" (圖形粉紅色的路徑) 。
- 第四次輸入的 grade 為 60 ，符合條件判斷 4，輸出 ``成績為 D" (圖形淺藍色的路徑) 。

- 第五次輸入的 grade 為 59 ，都不符合上面的條件判斷，會執行 else 區塊內的程式碼，輸出 ``成績為 E" (圖形紅色的路徑) 。
- 因為 if 條件判斷、所有 else if 條件判斷或 else 成立時，執行的程式碼都只有一行，所以區塊符號都可以省略不寫。

7.4.4 switch 與 break

breake

- break：在程式執行時，遇到 break，會跳過目前執行區塊後的程式碼，並跳出目前的區塊。

switch

當需要對一個 int、short、char、byte 或是 enum 型態值做多種不同的判斷時，可以使用 switch ，以下是 switch 的語法：

[[Image:Switch-1.png]]

```
1  switch (變數或運算式) {
2      case 值 1:
3          符合值 1 執行的程式碼;
4          ....
5          break;
6      case 值 2:
7          符合值 2 執行的程式碼;
8          ....
9          break;
10        :
11        :
12     case 值 n:
13         符合值 n 執行的程式碼;
14         ....
15         break;
16     default:
17         都不符合上述值執行的程式碼;
18         ....
19 }
```

說明：

- 第 1 行：Switch 後面括號內的程式碼，可以是變數（例如：grade）或是運算式，然而其型態必需是 int、short、char、byte 或是 enum 型態（圖形藍色菱形的部

份）。

- 第 2 、6 、12 行：用以判斷 switch 後面括號內的變數或運算式的值是否符合值 1（第 2 行；圖形藍色的路徑）、值 2（第 6 行；圖形綠色的路徑）、值 n（第 12 行；圖形粉紅色的路徑）的條件判斷。值 1、值 2、值 n 必須是常數（compile-time constant）。
- 第 (3~4) 、(7~8) 、(13~14) 行：如果符合值 1（圖形藍色的路徑）、值 2（圖形綠色的路徑）、值 n（圖形粉紅色的路徑）時會執行的程式碼。
- 第 16 行：如果都不符合 case 值 1 到 case 值 n 的情況下，則會去執行 default 區塊內（第 17、18 行；圖形紅色的路徑）的程式碼，然後離開整個 swtich 的條件判斷。
- 第 5 、9 、15 行： **當程式執行遇到 break 敘述，會結束 swtich 的執行** 。
- 在 Java 語言中， **switch 條件判斷的 case 的值只能是單一的常數值（compile-time constant），不可以是範圍值（例如：>=90）** 。

Example 4：請使用 switch 來設計一個 Java 程式，讓使用者自行輸入一個成績，判斷該輸入的成績是介於哪一個區間之內。

[[Image:Switch-2.png]]

```
1  import java.util.Scanner;
2  class Switch_Statement {
3      public static void main(String[] args) {
4          switch_statement();
5          switch_statement();
6              switch_statement();
7          switch_statement();
8          switch_statement();
9              switch_statement();
10     }
11
12         public static void switch_statement() {
13         System.out.print(" 請輸入成績:");
14         Scanner scanner = new Scanner(System.in);
15         int grade = scanner.nextInt();
16             grade = (int)grade / 10;
17
18             switch (grade) {
19                     case 10:
20                     case 9:
21                             System.out.println("90~100");
22                             break;
23                     case 8:
```

```
                        System.out.println("80~89");
                        break;
                case 7:
                        System.out.println("70~100");
                        break;
                case 6:
                        System.out.println("60~69");
                        break;
                default:
                        System.out.println("0~59");
            }
        }
}
```

執行結果：

```
請輸入成績:100
90~100
請輸入成績:90
90~100
請輸入成績:89
80~89
請輸入成績:74
70~59
請輸入成績:60
60~69
請輸入成績:59
0~59
```

說明：

- 第一次輸入的 grade 為 100，符合 case 10 的條件判斷，但 case 10 區塊內沒有 break 敍述（圖形藍色的路徑）會繼續往下執行到 case 9 區塊內的程式碼，輸出 ``90~100"（圖形綠色的路徑），之後遇到 break 敍述跳離開整個 swtich。
- 第二次輸入的 grade 為 90，符合 case 9 的條件判斷，會執行 case 9 區塊內的程式碼，輸出 ``90~100"（圖形綠色的路徑），之後遇到 break 敍述跳離開整個 swtich 。
- 第三次輸入的 grade 為 89，符合 case 8 的條件判斷，會執行 case 8 區塊內的程式碼，輸出 ``80~89"（圖形粉紅色的路徑），之後遇到 break 敍述跳離開整個 swtich。
- 第四次輸入的 grade 為 74，符合 case 7 的條件判斷，會執行 case 7 區塊內的程式碼，輸出 ``70~79"（圖形淺藍色的路徑），之後遇到 break 敍述跳離開整個 swtich。
- 第五次輸入的 grade 為 60，符合 case 6 的條件判斷，會執行 case 6 區塊內的程式碼，輸出 ``60~69"（圖形橘色的路徑），之後遇到 break 敍述跳離開整個 swtich。
- 第六次輸入的 grade 為 59，都不符合上面 case 的條件判斷，會執行 default 區塊

內的程式碼，輸出 ``0~59"（圖形紅色的路徑），之後離開 swtich。

7.4.5 巢狀條件判斷

巢狀條件判斷（nested-if）：是指在一個 if 裡面還有 if 。

Example 5：請寫一個判斷使用者輸入的數是否是 2 或 3 或 6 的倍數，或者都不是他們倍數的 java 程式。

[[Image:nested_if.pngllink=]]

```
import java.util.Scanner;
public class Times {
        public static Scanner keyboard = new Scanner(System.in);

        public static void main(String[] args) {
                times();
                times();
                times();
                times();
        }

        public static void times() {
                System.out.print(" 請輸入一個整數：");
                int num = keyboard.nextInt();

                if ((num % 2) == 0) {
                        if ((num % 3) == 0) {
                                System.out.println(num + " 是 2、3、6 的倍數。");
                        }
                        else {
                                System.out.println(num + " 是 2 的倍數。");
                        }
                }
                else {
                        if ((num % 3) == 0) {
                                System.out.println(num + " 是 3 的倍數。");
                        }
                        else {
                                System.out.println(num + " 都不是 2、3、6 的倍數。");
                        }
                }
        }
}
```

輸出結果：

```
請輸入一個整數：12
12 是 2、3、6 的倍數。
請輸入一個整數：4
4 是 2 的倍數。
請輸入一個整數：9
9 是 3 的倍數。
請輸入一個整數：11
11 都不是 2、3、6 的倍數。
```

說明：

- 當輸入的 num 為 12 時，第一層（外層）可被 2 整除（執行 if 區塊），再往第二層（內層）可被 3 整除（執行 if 區塊），輸出 ``12 是 2、3、6 的倍數。"（圖形藍色的路徑）。
- 當輸入的 num 為 4 時，第一層（外層）可被 2 整除（執行 if 區塊），再往第二層（內層）不可被 3 整除（執行 else 區塊），輸出 ``4 是 2 的倍數。"（圖形粉紅色的路徑）。
- 當輸入的 num 為 9 時，第一層（外層）不可被 2 整除（執行 else 區塊），再往第二層（內層）可被 3 整除（執行 if 區塊內），輸出 ``9 是 3 的倍數。"（圖形綠色的路徑）。
- 當輸入的 num 為 11 時，第一層（外層）不可被 2 整除（執行 else 區塊），再往第二層（內層）不可被 3 整除（執行 else 區塊），輸出 ``11 都不是 2、3、6 的倍數。"（圖形紅色的路徑）。

7.4.6 ？：條件判斷式

條件判斷? 條件判斷成立時執行的程式碼: 條件判斷不成立時執行的程式碼;

if 與?：的差別在於 if 不傳回值；而?：可以傳回值（見以下範例）

Example 6：請設計一個 Java 程式，可以判斷成績是否及格。

```
if (grade >= 60)
        message = " 成績及格!";
else
        message = " 成績不及格!";
```

上面 Example 6 可以改寫成下面只有一行的?：條件判斷式。

```
message = (grade >=60) ? " 成績及格!" : " 成績不及格!";

message 的值是上例執行 ? : 運算式之後傳回的結果。
```

CHAPTER

EIGHT

迴圈與遞迴

8.1 迴圈

我們在撰寫程式時，常常需要用到迴圈。像是要讓使用者輸入 10 位學生的成績，如果沒有使用迴圈，就必須將輸入成績的程式碼寫 10 次，而這 10 次的程式碼卻完全一樣。如果使用迴圈，便只需要寫 1 次就可以了。使用迴圈讓這輸入成績的程式碼可以連續執行 10 次，以達到相同的效果，因此可以讓程式設計的更精簡、有彈性。

Java 用來寫迴圈的句子有以下三個：

1. for
2. while
3. do-while

8.1.1 for-loop

for-loop 的語法如下：

```
for (控制變數的初始值設定; 控制變數的條件判斷; 控制變數值的改變) {
  符合控制變數的條件判斷時，執行的程式碼區塊;
  ....
}
```

說明：

- for-loop 主要由四個部份所組成，分別是「控制變數的初始值設定」、「控制變數的條件判斷」、「控制變數值的改變」與「符合控制變數的條件判斷時，執行的程式碼區塊」。
- 「控制變數的初始值設定」：只有在迴圈第一次執行時才會執行，其目的是用來設定控制迴圈變數（loop control variable）初始值的設定。例如：int i = 0; 表示 for

迴圈一開始執行時，宣告一個 int 的控制變數，其值為 0。

- 「控制變數的條件判斷」：每一次執行 { } 內的程式碼區塊前，會先執行「控制變數的條件判斷」。如果符合該條件判斷，才會去執行迴圈內的程式碼；不符合便結束迴圈的執行。例如：i < 5; 表示如果此時迴圈的控制變數 i 的值小於 5 才會去執行迴圈內的程式碼，否則便離開迴圈。
- 「控制變數值的改變」：每一次 for 執行完在 { } 內的程式碼區塊之後，會執行的部份。它會改變迴圈控制變數的值。例如：i++; 表示每一次迴圈執行完之後，將變數 i 的值加 1。
- 「符合控制變數的條件判斷時，執行的程式碼區塊」（第 2 行的 { 至第 5 行的 }）：符合控制變數的條件判斷時，會執行的程式碼。例如：System.out.println(i); 表示將迴圈的控制變數 i 的值輸出。

Example 1：請利用 for-loop 來設計一個程式，讓迴圈執行 5 次，每次執行時都輸出迴圈執行到第幾次。

```
1 public class for_loop {
2   public static void main(String[] args) {
3     for (int i=1; i<=5; i++)
4       System.out.println("for 迴圈執行第" + i + " 次。");
5   }
6 }
```

執行結果：

```
1 for 迴圈執行第 1 次。
2 for 迴圈執行第 2 次。
3 for 迴圈執行第 3 次。
4 for 迴圈執行第 4 次。
5 for 迴圈執行第 5 次。
```

- 由於區塊內的程式碼只有一個句子，所以 { } 可以省略不寫。

Example 2：請利用 for-loop 搭配 if 條件判斷來設計一個 Java 程式，將 1 到 10 中的偶數輸出。

```
1 public class for_loop2 {
2   public static void main(String[] args) {
3     for (int i=1; i<=10; i++)
4       if (i % 2 == 0)
5         System.out.println("1 到 10 的偶數有：" + i);
6   }
7 }
```

輸出結果：

```
1 到 10 的偶數有：2
1 到 10 的偶數有：4
1 到 10 的偶數有：6
1 到 10 的偶數有：8
1 到 10 的偶數有：10
```

8.1.2 while-loop

while-loop 是前測式的迴圈，其語法如下：

```
//控制變數的初始值設定;

while (控制變數的條件判斷) {
  //符合控制變數的條件判斷時，執行的程式碼區塊;
  //....
  //控制變數值的改變;
}
```

說明：

- while-loop 和 for-loop 同樣由四個部分所組成，分別是「控制變數的初始值設定」、「控制變數的條件判斷」、「符合控制變數的條件判斷時，執行的程式碼區塊」與「控制變數值的改變」。
- while-loop 是「前測式迴圈」，因為控制變數的條件判斷位於迴圈一開始的地方，因此 while-loop 執行次數可以為 0 次，也就是控制變數的條件判斷在一開始便不符合。

Example 3：請利用 while-loop 設計一個 Java 程式，讓迴圈執行 5 次，每次執行時都輸出迴圈執行到第幾次。

```
public class while_loop {
    public static void main(String[] args) {
       int i = 1;
       while (i <= 5) {
           System.out.println("while 迴圈執行第" + i + " 次。");
           i++;
       }
   }
}
```

執行結果：

```
while 迴圈執行第 1 次。
while 迴圈執行第 2 次。
while 迴圈執行第 3 次。
```

```
while 迴圈執行第 4 次。
while 迴圈執行第 5 次。
```

Example 4：請利用 while-loop 設計一個 Java 程式，這個程式一直讓使用者從鍵盤輸入數字，直到輸入 -1 時為止。這個程式輸出這些數字的和。

```
import java.util.Scanner;
public class Sentinel {
 static Scanner keyboard = new Scanner(System.in);
 static int i = -1;

 public static void main(String[] args) {
     int total = 0;
     int s = 0;

      System.out.print("Please enter score or enter -1 to stop: ");
     s = keyboard.nextInt();**
     while (s != i) {
         total = total + s;
          **System.out.print("Please enter score or enter -1 to stop: ");
         s = keyboard.nextInt();
     }
     System.out.println("The sum of scores: "+ total);
 }
}
```

輸出結果：

```
Please enter score or enter -1 to stop: 3
Please enter score or enter -1 to stop: 5
Please enter score or enter -1 to stop: 7
Please enter score or enter -1 to stop: 9
Please enter score or enter -1 to stop: -1
The sum of scores: 24
```

- 這個程式的特性是輸入數字的程式碼，在進入迴圈前及迴圈尾各出現一次。
- 這種迴圈叫旗標控制迴圈（sentinel-controlled loop）。

8.1.3 do-while

do-while loop 是一種後測式迴圈，其語法如下：

說明：

- do-while loop 和 for-loop、while-loop 同樣由四個部分所組成，分別是「控制變數的初始值設定」、「控制變數的條件判斷」、「符合控制變數的條件判斷時，執行的程

式碼區塊」與「控制變數值的改變」。

- do-while loop 是「後測式迴圈」，因為控制變數的條件判斷位於最後面。
- do-while loop 最少會執行 1 次。因為一開始會先執行 do 區塊內的程式碼，之後才會進行 while 之後控制變數的條件判斷。
- 特別留意在最一行 while 控制變數的條件判斷後面，需加上「;」以表示 do-while loop 的結束。

Example 5：請利用 do-while loop 設計一個 Java 程式，讓迴圈執行 5 次，每次執行時都輸出迴圈執行第幾次。

```
public class do_while_loop {
    public static void main(String[] args) {
     **int i = 1;
    do {
        System.out.println("do while 迴圈執行第" + i + " 次。");
        i++;
    } while (i <= 5);**
   }
}
```

執行結果：

```
do while 迴圈執行第 1 次。
do while 迴圈執行第 2 次。
do while 迴圈執行第 3 次。
do while 迴圈執行第 4 次。
do while 迴圈執行第 5 次。
```

Example 6：請利用 do-while loop 搭配 if 條件判斷來設計一個 Java 程式，將 1 到 10 中間的偶數輸出。

```
public class do_while_loop2 {
    public static void main(String[] args) {
     **int i = 1;
    do {
        if (i % 2 == 0)
            System.out.println("1 到 10 的偶數有：" + i);
        i++;
    } while (i <= 10);**
   }
}
```

執行結果：

```
1 到 10 的偶數有：2
1 到 10 的偶數有：4
1 到 10 的偶數有：6
```

```
1 到 10 的偶數有：8
1 到 10 的偶數有：10
```

8.1.4 breake & continue

====breake====

- break：可以離開目前的 switch、for、while、do while 的區塊，並跳離至區塊後的下一行程式碼。在 switch 中主要用來離開；而在 for、while 與 do while 迴圈中，主要用於中斷目前迴圈的執行。

Example 7：下面是一個使用 break 敍述的 Java 程式，觀察程式碼及程式執行的過程。

執行結果：

```
迴圈執行第 1 次。
迴圈執行第 2 次。
迴圈執行第 3 次。
迴圈執行第 4 次。
```

- 此迴圈只會執行 4 次，因為當 i==5 時就會執行 break; 述敍而離開迴圈。

8.1.5 continue

- continue：作用與 break 敍述類似，主要使用於 for、while、do while 迴圈，所不同的是 break 敍述會結束迴圈區塊的執行，而 continue 只會結束目前執行中區塊的程式碼，並跳回迴圈區塊的開頭繼續下一迴圈，而不是離開迴圈。

Example 8：下面是一個使用 continue 敍述的 Java 程式，觀察程式碼及程式執行的過程。

```
public class continue1 {
    public static void main(String[] args) {
       for (int i=1; i<=10; i++) {
           if (i == 5)
               continue;
           System.out.println(" 迴圈執行第" + i + " 次。");
       }
   }
}
```

執行結果：

```
迴圈執行第 1 次。
迴圈執行第 2 次。
迴圈執行第 3 次。
```

```
迴圈執行第 4 次。
迴圈執行第 6 次。
迴圈執行第 7 次。
迴圈執行第 8 次。
迴圈執行第 9 次。
迴圈執行第 10 次。
```

- 此迴圈會執行 1 ~ 4 加 6 ~ 9 次，當 i==5 時，會執行 continue; 述敍而跳離此次迴圈，然後從區塊開頭執行下一次迴圈，所以 i==5 並沒有被執行。

8.1.6 巢狀迴圈

- 巢狀迴圈（nested loop）是指一個迴圈裡面還存在一個以上的迴圈。
- 巢狀迴圈執行的總次數為每一個迴圈執行次數的乘積。例如：一個輸出九九乘法表程式的執行次數，就是外層的 8 次乘以內層的 9 次，因此得到 8 x 9 = 72 次。
- 巢狀迴圈裡面的每一個迴圈都不可以與其它迴圈重疊，只能是彼此包含的關係。

Example 9：請使用巢狀迴圈設計一個輸出九九乘法表的程式，並計算巢狀迴圈總共執行了幾次。

```
public class NineNineTable {
    public static void main(String[] args) {
       int count = 0;
       for (int i=2; i<=9; i++) {
           for (int j=1; j<=9; j++) {
               System.out.printf("%d*%d=%2d ", i, j, i*j);
               count++;
           }
           System.out.printf("\n");
       }
       System.out.printf(" 巢狀迴圈總共執行了%d 次 \n", count);
   }
}
```

執行結果：

```
2*1= 2 2*2= 4 2*3= 6 2*4= 8 2*5=10 2*6=12 2*7=14 2*8=16 2*9=18
3*1= 3 3*2= 6 3*3= 9 3*4=12 3*5=15 3*6=18 3*7=21 3*8=24 3*9=27
4*1= 4 4*2= 8 4*3=12 4*4=16 4*5=20 4*6=24 4*7=28 4*8=32 4*9=36
5*1= 5 5*2=10 5*3=15 5*4=20 5*5=25 5*6=30 5*7=35 5*8=40 5*9=45
6*1= 6 6*2=12 6*3=18 6*4=24 6*5=30 6*6=36 6*7=42 6*8=48 6*9=54
7*1= 7 7*2=14 7*3=21 7*4=28 7*5=35 7*6=42 7*7=49 7*8=56 7*9=63
8*1= 8 8*2=16 8*3=24 8*4=32 8*5=40 8*6=48 8*7=56 8*8=64 8*9=72
9*1= 9 9*2=18 9*3=27 9*4=36 9*5=45 9*6=54 9*7=63 9*8=72 9*9=81
巢狀迴圈總共執行了 72 次
```

Example 10：請使用巢狀迴圈設計一個 Java 程式，這個程式持續輸入兩個整數 m, n，利用 while-loop 計算 m 的 n 次方，直到輸入 999 為止。

```
import java.util.Scanner;
public class PowerOf {
    public static Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        int n, m;

        System.out.println("Input: ");
        m = input.nextInt();

        while (m != 999) {
            n = input.nextInt();

            System.out.print(m + "^" + n + "=");

            int result = 1;
            while (n > 0) {
                result *= m;
                n--;
            }

            System.out.println(result);

            System.out.println("Input: ");
            m = input.nextInt();
        }
    }
}
```

------Result--------

```
Input:
3
4
3^4=81
Input:
4
5
4^5=1024
Input:
5
6
5^6=15625
Input:
999
```

說明：

- 外層的 while 迴圈控制程式執行到輸入 999 才結束。
- 內層的 while 迴圈計算 m 的 n 次方。

8.1.7 for-loop、while-loop 與 do-while-loop 的相互轉換

- for-loop、while-loop、do-while-loop 可以彼此相互轉換，前面曾經介紹這三種迴圈主要由 4 個部分所組成，我們可以藉由搬動這 4 個部分的程式碼來完成迴圈的相互轉換。
- for-loop -> while-loop 可以遵從下面三個步驟：

1. 將「控制變數的初始值設定」從 for-loop 拉到 while-loop 前面。
2. 將「控制變數的條件判斷」保留在 while-loop 的「控制變數的條件判斷」內。
3. 將「控制變數值的改變」拉到 while-loop 的「符合控制變數的條件判斷時，執行的程式碼區塊」內的最後一行。

- while-loop -> do-while-loop ：

1. 將 while-loop 的「控制變數的條件判斷」拉到 do-while-loop 最後一行的「「控制變數的條件判斷」」內即可完成。

Example 11：請分別利用 for 、while 與 do-while 設計計算 1 加到 100 的程式。

```
public class Loop_Transfer1 {
    public static void main(String[] args) {
        // for-loop
        System.out.println("for 迴圈 1 加到 100 為" + for_loop());

        // while-loop
        System.out.println("while 迴圈 1 加到 100 為" + while_loop());

        // do-while-loop
         System.out.println("do-while 迴圈 1 加到 100 為" + do_while_loop());
    }

    public static int for_loop() {
        int sum = 0;
        for(int i=1; i<=100; i++)
            sum += i;
        return sum;
    }

    public static int while_loop() {
```

```
        int sum = 0;
        int i = 1;
        while (i <= 100) {
            sum += i;
            i++;
        }
        return sum;
    }

    public static int do_while_loop() {
        int sum = 0;
        i = 1;
        do {
            sum += i;
            i++;
        } while (i <= 100);
        return sum;
    }
}
```

執行結果：

```
for 迴圈 1 加到 100 為 5050 while 迴圈 1 加到 100 為 5050 do-while 迴圈 1 加到 100 為 5050
```

Example 12：請分別利用 for、while 與 do-while 設計計算 1 到 100 中間 3 的倍數的總合。

```
public class Loop_Transfer2 {
    public static void main(String[] args) {
        // for-loop
        System.out.println("for 迴圈 1 到 100 間 3 的倍數總合為" + for_loop());

        // while-loop
        System.out.println("while 迴圈 1 到 100 間 3 的倍數總合為" + while_loop());

        // do-while-loop
        System.out.println("do-while 迴圈 1 到 100 間 3 的倍數總合為" + do_while_loop());
    }

    public static int for_loop() {
        int count = 0;
        for (int i=1; i <=100; i++)
            if (i % 3 == 0)
                count += i;
        return count;
    }
```

```
20
21      public static int while_loop() {
22          int count = 0;
23          int i = 1;
24          while (i <= 100) {
25              if (i % 3 == 0)
26                  count += i;
27              i++;
28          }
29          return count;
30      }
31
32      public static int do_while_loop() {
33          int count = 0;
34          int i = 1;
35          do {
36              if (i % 3 == 0)
37                  count += i;
38              i++;
39          } while (i <= 100);
40          return count;
41      }
42  }
```

------Result--------

```
for 迴圈 1 到 100 間 3 的倍數總合為 1683
while 迴圈 1 到 100 間 3 的倍數總合為 1683
do-while 迴圈 1 到 100 間 3 的倍數總合為 1683
```

8.2 遞迴

- 遞迴是一個相當獨特，對訓練邏輯思考很有助益的程式設計方式，為了能循序漸進的闡述遞迴程式的撰寫，在本節中，我們將以 String 資料型態當作程式的輸入資料，並以 1 來表示 boolean 的 true；而以 0 來表示 boolean 的 false。
- 我們需要使用 Integer.parseInt(<String Variable>) 將 String 型態的資料轉換為 int 型態的資料。其轉換過後的值如果 == 1，則代表 true；如果!= 1，則代表 fasle。

:[註] Java 的每一個 primitive type 都有一個與之對應的 reference type 或類別，例如：int 與 Integer。parseInt 則是 Integer 的一個類別方法。Java 這麼設計的原因，會在《Java 物件導向程式設計》中的「泛型與 Collection」部分說明。

8.2.1 簡易的遞迴程序

以下是呼叫 and2 的範例：

```
System.out.println(and2("11")); //=> true
System.out.println(and2("10")); //=> false
System.out.println(and2("01")); //=> false
System.out.println(and2("00")); //=> false
```

and2 的特性是，它的參數中的兩個資料值都要是 1 結果才是 true，要不然結果就是 false。

現在我們試著將 and2 的功能擴充，並將擴充後的方法命名為 allTrue。而 allTrue 可以判斷一個 string 中的值是否都是 true，這個 string 可以有不止兩個資料值。以下是呼叫 allTrue 的範例：

```
System.out.println(allTrue(""));       => true  輸入參數中沒有一個 false，所以結果是 true。
System.out.println(allTrue("111"));    => true
System.out.println(allTrue("1111"));   => true
System.out.println(allTrue("11010")); => false
```

以下是 allTrue 的寫法：

```
public class Recursive {
    public static void main(String args[]) {
        System.out.println(allTrue(""));       // => true  輸入參數中沒有一個 false 的值，所
        System.out.println(allTrue("111"));    // => true
        System.out.println(allTrue("1111"));   // => true
        System.out.println(allTrue("11010"));  // => false
    }

    public static boolean allTrue(String str) {
        if (str.equals(""))
            return true;
        else if (Integer.parseInt(str.substring(0, 1)) == 1 )
            return allTrue(str.substring(1));
        else
            return false;
    }
}
```

- Integer.parseInt()：將 () 裡的 string 由 String 型態轉換為 int 型態，以提供 if 來判斷其為 true(== 1) 或 false(!= 1)。
- str.substring(0, 1)：取出 str 字串變數中的第 0 個位置（index）至第 1 個位置（index）間的資料。（例如："1234" => ``1"）
- str.substring(1)：取出 str 字串變數中第 1 個位置以後的資料。（例如："1234" =>

``234"）

allTrue(``111") ; 的執行過程可 trace 如下：

```
   allTrue("111")
=> allTrue("11")
=> allTrue("1")
=> allTrue("")
=> true
```

allTrue(``1101") ; 的執行過程可 trace 如下：

```
   allTrue("1101")
=> allTrue("101")
=> allTrue("01")
=> false
```

觀察 allTrue 這個程序以及它的執行過程，我們發現：

1. allTrue 內有三個執行的可能（if 條件判斷的三個條件情況）。
2. 其中一行呼叫 allTrue 自己。這個呼叫自己的遞迴呼叫，傳入少了頭一筆資料值的 String(str.substring(1))。另外兩行沒有遞迴呼叫，而程式執行到這兩行之後便傳回 true 或 false，然後結束。
3. if 條件判斷式，控制程式執行哪個式子。
4. allTrue 這個程式的輸出入型態是：String -> boolean，也就是傳入一個 String、傳出一個布林值的意思。

一般而言，遞迴程式都有類似的模式：

1. 有終止條件（termination condition）如：str.equals(``")、n == 0。
2. 有遞迴呼叫，而遞迴呼叫所傳入的資料一定會趨近終止條件（例如使用 substring(1) 使 string 愈來愈縮短）。

這很合理，因為這樣程式才能結束。

這個單元的練習都是 String -> boolean 型態的程序。我們還會在後續的單元，陸續的介紹遞迴程式的其他型式。另外，在撰寫程式的過程中，最簡單的除錯工具是把變數的值印出來。System.out.println(<expression>) 與 System.out.println() 是可以顯示一個式子的傳回值與換行的兩個程序：

執行以上的程式碼會列印與傳回以下的內容：

```
str=111
str=11
str=1
str=
true
```

前面 4 行與最後 1 行分別顯示了每次呼叫 allTrue 的輸入參數的值與最後的執行結果。

8.2.2 累計答案值的遞迴程序

上一單元我們練習了幾個傳回 true 或 false 的程序。這些題目的答案不需要經過累計，例如：累加、累乘、或累積成字串的過程。這個單元我們將練習需要經過累計才能計算出答案的遞迴程序。

假設我們需要撰寫一個能夠計算一個 string 中有幾個 true 的程序。我們將這個程序命名為 countTrue：

```
countTrue("");     => 0
countTrue("1101"); => 3
```

很顯然的，在程序執行的過程中如果遇到 1(true) 則需要將 1 加入結果之中。寫法如下：

```
public class Recursive {
    public static void main(String args[]) {
        System.out.println(countTrue(""));       // => 0  輸入參數中沒有一個 false 的值，所以
        System.out.println(countTrue("1"));      // => 1
        System.out.println(countTrue("01"));     // => 1
        System.out.println(countTrue("101"));    // => 2
        System.out.println(countTrue("1101"));   // => 3
    }

    public static int countTrue(String str) {
        if (str.equals(""))
            return 0;
        else if (Integer.parseInt(str.substring(0, 1)) == 1)  // 1(true)
            return 1 + countTrue(str.substring(1));
        else                                                  // 0(false)
            return countTrue(str.substring(1));
    }
}
```

我們可以追蹤這個程式是怎麼執行的：:: countTrue(``"); ``" 使 str.equals(``") 成立因此傳回 0 => 0

CountTrue(``1"); ``1" 使 (Integer.parseInt(str.substring(0, 1)) == 1) 成立因此傳回 1

=> (1 + countTrue(``");) => (1 + 0) => 1

countTrue(``01"); ``01" 使 else 成立因此傳回 1

=> countTrue(``1"); => 1

countTrue(``101"); ``101" 使 (Integer.parseInt(str.substring(0, 1)) == 1) 成立因此傳回 2

=> (1 + countTrue(``01"); => (1 + 1) => 2

接下來，我們可以追蹤將一個有四個資料值的 string 傳入 countTrue 的執行狀況：

```
   countTrue("1101");
=> (1 + countTrue("101");)
=> (1 + (1 + countTrue("01");))
=> (1 + (1 + countTrue("1");))
=> (1 + (1 + (1 + countTrue(""));))
=> (1 + (1 + (1 + 0)))
=> 3
```

另外，countTrue 的型態是：String -> int 。

8.2.3 將遞迴程序轉換成迴圈程序

1. 階乘（factorial）是一個常見的遞迴程序的範例。如果「!」代表階乘的符號，則

```
0! = 1
N! = N * (N - 1)!
```

2. 加總（sum）是計算一個 string 變數內數字和的範例。

```
例如：sum("1234")  =>  10
```

在這一個單元，我們將以階乘（factorial）與加總（Sum）兩個程式為範例，說明如何將一個遞迴程式逐步的改寫成迴圈程式。

內涵式遞迴

階乘函數可以用 Java 寫成如下的程式碼，我們將之命名為 factorial：

```
public class  Recursive {
    public static void main(String args[]) {
        System.out.println(factorial(3));
    }

    public static int factorial(int n) {
        if (n == 0)
            return 1;
        else
            return n * factorial(n - 1);
    }
}
```

Trace

```
   factorial(3)
=> return n * factorial(n - 1);   // n=3
=> return 3 * factorial(2);
=> return 3 * (n * factorial(n - 1));   // n=2
=> return 3 * (2 * factorial(1));
=> return 3 * (2 * (n * factorial(n - 1)));   // n=1
=> return 3 * (2 * (1 * factorial(0)));
=> return 3 * (2 * (1 * 1));   // n=0
=> return 3 * (2 * (1 * 1));
=> 6
```

觀察以上的執行狀況，呼叫 factorial 的遞迴呼叫是內涵 (embedded) 在乘法算式 (n * ...) 之內。而程式的執行結果是在最後一個遞迴呼叫 factorial(0) 完成後才依序累乘 (3 * (2 * (1 * 1))) 而得的。這種類型的遞迴程序稱為內涵式遞迴（embedded recursion）。

加總涵式 sum 的寫法如下：

```
public class Recursive {
    public static void main(String args[]) {
        System.out.println(sum("1234"));
    }

    public static int sum(String str) {
        if (str.equals(""))
            return 0;
        else
            return  Integer.parseInt(str.substring(0, 1)) + sum(str.substring(1));
    }
}
```

Trace

```
   Sum("1234")
=> return Integer.parseInt(str.substring(0, 1)) + sum(str.substring(1));   // str="1234"
=> return 1 + sum("234");
=> return 1 + (Integer.parseInt(str.substring(0, 1)) + sum(str.substring(1)));   // str=
=> return 1 + (2 + sum("34"));
=> return 1 + (2 + (Integer.parseInt(str.substring(0, 1)) + sum(str.substring(1))));   /
=> return 1 + (2 + (3 + sum("4")));
=> return 1 + (2 + (3 + (Integer.parseInt(str.substring(0, 1)) + sum(str.substring(1)))))
=> return 1 + (2 + (3 + (4 + sum(""))));
=> return 1 + (2 + (3 + (4 + (Integer.parseInt(str.substring(0, 1)) + sum(str.substring(
=> return 1 + (2 + (3 + (4 + 0)));
=> 10
```

觀察以上的執行狀況，呼叫 sum 的遞迴呼叫也是內涵 (embedded) 在加總算式 (Integer.parseInt(str.substring(0, 1)) + ...) 之內。而程式的執行結果是在最後一個遞迴呼叫

sum(``") 完成後才依序累加 (1 + (2 + (3 + (4 + 0)))) 而得的。這是內涵式遞迴的另一個例子。

尾端式遞迴

第二種形式的遞迴程序是尾端遞迴（tail recursion）。尾端遞迴的特色是遞迴呼叫沒有內涵在任何一個尚未執行完成的式子內。以下面的例子而言，呼叫 tail-fac 的遞迴呼叫雖然是在 if 之內，但是該 if 的條件判斷式已經執行完畢，所以是尾端遞迴。

Trace

```
   tail-fac(3, 1)
=> tail-fac(2, 3)
=> tail-fac(1, 6)
=> tail-fac(0, 6)
=> 6
```

以上程序的執行狀況是「平的」。尾端遞迴在執行的過程中不會像內涵式遞迴累積一層又一層如同樓梯般的式子，原因是前面遞迴呼叫時所產生的區域變數區內的變數值，並不需要被保留，因此下一次遞迴呼叫的區域變數區可以直接的覆蓋上一次遞迴呼叫時所使用的區域變數區。

將內涵式遞迴程序轉換成尾端遞迴程序的技巧是增加傳入的參數。以 fac2 來說我們增加了一個能夠儲存累乘值的參數 m。增加這個參數後便不需要以累積乘法算式的方式來計算階乘的值了。

而尾端遞迴的加總函式 sum，也可以經由增加一個參數及改變遞迴的回傳值得到：

```
public class  Recursive {
    public static void main(String args[]) {
        System.out.println(tail-sum("1234", 0));
    }

    public static int tail-sum(String str, int n) {
        if (str.equals(""))
            return n;
        else
            return  tail-sum(str.substring(1), (n + Integer.parseInt(str.substring(0, 1)
    }
}
```

Trace

```
   tail-sum("1234", 0)
=> tail-sum("234", 1)
=> tail-sum("34", 3)
=> tail-sum("4", 6)
```

```
=> tail-sum("", 10)
=> 10
```

迴圈

第三種計算階乘的方式是使用一般程式語言常常使用的「迴圈」。一個內涵式遞迴程序在轉換成尾端式遞迴程序後，便可以改寫成迴圈程式了。

轉換的方式可以分成下面三個步驟：

1. 將尾端式遞迴的終止條件內的條件判斷拉到迴圈的條件判斷內，並將它轉換成相反（Not）的條件判斷（加上! 或!=）。
2. 將尾端式遞迴的參數拉到迴圈內並改寫成設值運算式，但是要注意的是設值運算式的前後次序。
3. 將尾端式遞迴的終止條件成立時的回傳值拉到迴圈的後面，讓程序結束後將結果值回傳回去。

使用 whlile-loop 將尾端式遞迴轉換成一般的程式語言常常使用的「迴圈」。

```
public class Recursive {
    public static void main(String args[]){
        System.out.println(fac3(3, 1));
    }

    public static int fac3(int n, int m) {
        while (n != 0) {
            m = m * n;
            n = n - 1;
            // 注意：上兩行次序不可顛倒。
        }
        return m;
    }
}
```

Trace

```
   fac3(3, 1)
=> 6
```

```
public class Recursive {
    public static void main(String args[]){
        System.out.println(sum3("1234", 0));
    }

    public static int sum3(String str, int n) {
        while (!str.equals("")) {
```

```
            n = n + Integer.parseInt(str.substring(0, 1));
            str = str.substring(1);
            // 注意：上兩行次序不可顛倒
        }
        return n;
    }
}
```

Trace

```
   sum3("1234", 0)
=> 10
```

同樣也可以使用 Java 語言的 for Loop 來將尾端式遞迴轉換成一般的程式語言常常使用的「迴圈」。

```
public class Recursive {
    public static void main(String args[]){
        System.out.println(fac3(3, 1));
    }

    public static int fac3(int n, int m) {
        for (int i=m; i<=n; i++)
            m = m * i;
        return r;
    }
}
```

Trace

```
   fac(3, 1)
=> 6
```

```
public class Recursive {
    public static void main(String args[]){
        System.out.println(sum3("1234", 0));
    }

    public static int sum3(String str, int n) {
        for (; !str.equals(""); str = str.substring(1))
            n = n + Integer.parseInt(str.substring(0, 1));
        return n;
    }
}
```

Trace

```
   sum3("1234", 0)
=> 10
```

CHAPTER

NINE

實例、實例變數與實例方法

Java 是一個物件導向程式語言。物件導向的基本觀念是讓程式可以描述、建構及處理真實世界中所看到的物件並設計它們之間的層次關係。例如：Exam 可以有 EnglishExam、ChineseExam 兩個副屬的種類。而一次 EnglishExam 又可以有任意數量的考生應考，其中每一份考卷中都可以有考生姓名與成績的資料。這時 Exam、EnglishExam 及 ChineseExam 便很適合設計成類別，而一份英文考試的考卷便可以用 EnglishExam 這個類別所生成的實例（instance or object）來代表。

一個類別可以使用 **new** 指令來產生實例。一個 Java 的程式可以由許多個有層次關係的類別構成，這個層次關係描述這些類別間的 **繼承關係** 。這一類的例子在真實世界中不勝枚舉，舉例來說：如果「哺乳類」是一個類別，那麼「人類」則可以是「哺乳類」的一個 **「子類別」** （subclass or subtype），相對的「哺乳類」也可以是「人類」的 **「父類別」** （superclass or supertype）。而某一個人「張三」便是「人類」的一個 **「實例」** 。

以「張三」這個實例來說，他的身高、體重、性別等資料都可以存放在這個實例的「實例變數」中，而「張三」能「跑」能「跳」，則可以是它的「實例方法」。簡單的說：「類別（class）」可以用 new 來產生「實例（instance）」，而「實例」可以包含「實例變數」與「實例方法」。其中「實例變數」負責資料的儲存，而「實例方法」則負責資料的處理：

Image:image001.jpg

註：Java 原文將「實例變數」稱為 non-static field；而將「實例方法」稱為 non-static method。為了與中文的譯名一致，這份講義將以 instance variable 與 instance method 稱之。同樣的，這份講義亦將以 class method 代替原文的 static method，以 class variable 代替原文的 static variable。

9.1 何謂實例（Instance）

實例（instance）是由 new 所產生的某個類別的實作，也稱為物件（object）。一個 class 可以生成多個實例，而且每個實例都擁有一份自己的實例變數。以下圖為例：class A 宣告了三個實例變數 x，y 與 z，而每個實例都為這三個實例變數配置了記憶體以儲存他們的值。

Image:newImage005.jpg

「實例變數」及「實例方法」與用於設計結構化程式的「類別變數」及「類別方法」的主要差異是：

1. 類別變數或類別方法以 static 起始，他們在程式開始執行時即實質的存在，所以不需要產生實例，即可使用。
2. 實例變數或實例方法沒有 static 起始，他們一定要等到實例產生之後才有實質的存在，因此需要實例，才可以使用。

在後續的圖例中，我們將使用「淡的顏色」來表示「沒有記憶體、沒有實質存在的變數或方法」，並以「鮮明的顏色」來表示「有記憶體、有實質存在的變數或方法」以區別它們的差異：

Image:Picture.jpg

以下是一個藉由呼叫實例方法將實例變數的值傳回並輸出的範例：

```
class Book {
  String name = "Sun";
  String getName() {
    return name;
  }
}

public class Ex2 {
  public static void main(String argv[]) {
    Book b1 = new Book();
    Book b2 = new Book();
    System.out.println("Name of b1:" + b1.getName());
    System.out.println("Name of b2:" + b2.getName());
  }
}
```

執行結果：

```
Name of b1:Sun
Name of b2:Sun
```

Media:Ex2.swfl 觀看執行過程及詳細解說]]

9.2 Class Method 與 Instance Method 的差異

接下來的這個範例，說明了類別方法與實例方法的差異：

```
class A {
  static String c1() {
    return "class method 不需 new 就可使用";
  }
  String c2() {
    return "instance method 需 new 才可使用";
  }
}
public class Ex3 {
  public static void main(String argv[]) {
    System.out.println("= 呼叫 class method=");
    System.out.println(A.c1());
    System.out.println("= 呼叫 instance method=");
    //System.out.println(A.c2()); =>error
    A a = new A();
    System.out.println(a.c2());
  }
}
```

執行結果：

```
= 呼叫類別方法 =
class method 不需 new 就可使用
= 呼叫實例方法 =
instance method 需 new 才可使用
```

Media:Ex3.swfl 觀看執行過程及詳細解說]]

9.3 Class Variable 與 Instance Variable 的差異

這個範例繼續說明了類別變數與實例變數之間的差異:

```
class A { static int a = 0; int b = 5; int getA() {
        return a;
    } void setA(int value) {
        a = value;
    } int getB() {
        return b;
```

```
    } void setB(int value) {
        b = value;
    }
}
public class Ex4 {
    public static void main(String argv[]) { A obj1 = new A(); A obj2 = new
        A(); obj1.setA(1); obj1.setB(20); obj2.setA(2); obj2.setB(30); Sys-
        tem.out.println(``a of obj1:" + obj1.getA()); System.out.println(``a of obj2:"
        + obj2.getA()); System.out.println(``b of obj1:" + obj1.getB()); Sys-
        tem.out.println(``b of obj2:" + obj2.getB());
    }
}
```

執行結果：

```
a of obj1:2
a of obj2:2
b of obj1:20
b of obj2:30
```

Media:Ex4.swfl 觀看執行過程

由於 a 是類別變數，只有一個儲存值，而這個值是共享的；b 則是實例變數，所以每個實例都有一個 b 的儲存值，而它們的值也可以不一樣。

類別變數、區域變數與實例變數的不同之處，透過它們在 Java 程式內的 **可見範圍（scope）** 與佔用記憶體的 **起始時間與釋放時間（extent）** 的不同，可以更明確的區分出來：

1. **類別變數** ：每一個 class 都有一個命名空間（name space），因此兩個 class 若有同名的變數、方法也不用擔心彼此衝突。這便好像將一個程式的記憶體劃分成好幾個區域，而每一個 class 都配屬了一個區域。 **隸屬於某一 class 的類別變數** 的有效時間是從程式開始執行，一直到程式結束：

- extent：從程式開始執行，一直到程式結束；
- scope：以 public、private、protect 宣告類別變數的可見範圍。

2. **區域變數** ：於方法執行時才存在，方法執行完畢後便消失。這種變數的值是存放在記憶體中稱為堆疊（stack）的一塊區域上。

- extent：從方法執行時開始，到方法結束時為止。

- scope：該變數所屬，由 { } 區隔的區塊（block）中。

3. **實例變數** ：實例變數是用來的儲存實例中資料的變數，實例（instance）的另一個名稱是物件（object）。例如在一個處理學生資料的程式中，某一學生的資料，可以存放在 name、id、score 等幾個 **隸屬於這個物件的實例變數** 中。實例是存放在記憶體中稱為堆積堆（heap）的一塊區域上。

- extent：從一個實例被造（new），直到它的記憶體被回收（garbage collect）時為止。
- scope：以 public、private、protect 宣告實例變數的可見範圍。

9.4 實例方法中的 this 是什麼？

這個單元將會以數個範例，將一個類別方法，轉換成作用相同的實例方法，以更深入的闡述在實例方法中常常用到的 this 這個保留字的意義。

在下面的範例中 ee 是一個用 new 生成的 EnglishExam 物件，這個物件的三個實例變數分別被設值成 3, 4, 5，然後傳入 englishScore 這個類別方法中：

```
class EnglishExam { public int vocab, grammar, listen; public static int englishScore(int
    v, int g, int l) {
        return v + g + l;
    }
}
public class Demo {
    public static void main(String argv[]) { EnglishExam ee = new EnglishExam();
        ee.vocab = 3; ee.grammar = 4; ee.listen = 5; System.out.print(``The score
        of the exam is "); System.out.println(EnglishExam.englishScore(ee.vocab,
        ee.grammar, ee.listen));
    }
}
```

執行結果：

```
1 The score of the exam is 12
```

englishScore 雖然是一個類別方方法，然而傳入這個方法的三個參數，實際上是 ee 的三個實例變數的值。那何必這麼麻煩？直接將 ee 傳入 englishScore 不是更簡潔嗎？以下即是將上例改寫後的版本：

```
class EnglishExam {
    public int vocab, grammar, listen; public static int score(EnglishExam THIS) { re-
        turn THIS.vocab + THIS.grammar + THIS.listen;
    }
}
public class Demo {
    public static void main(String argv[]) { EnglishExam ee = new EnglishExam();
        ee.vocab = 3; ee.grammar = 4; ee.listen = 5; System.out.print(``The score
        of the exam is "); System.out.println(EnglishExam.score(ee));
    }
}
```

這個範例直接將 ee 傳入 score（原名是 englishScoer）中，而 score 仍然是一個類別方法。在這個例子中我們使用：

```
EnglishExam.score(ee);
```

來呼叫 score 這個方法。如果我們將以上的呼叫改寫成：

```
ee.score();
```

並將 score 這個類別方法，改寫成實例方法:

```
class EnglishExam {
  public int vocab, grammar, listen;
  public int score() {
    return vocab + grammar + listen;
  }
}

public class Demo {
  public static void main(String argv[]) {
    EnglishExam ee = new EnglishExam();
    ee.vocab = 3; ee.grammar = 4; ee.listen = 5;
    System.out.print("The score of the exam is ");
    System.out.println(ee.score());
  }
}
```

那麼執行的結果仍然是：

```
The score of the exam is 12
```

由此可見，一個 instance method 其實有一個隱藏的參數。以上例而言是 ee，而任何在 score 方法中所用到的變數，如果不是區域變數或是類別變數，便會被當成是該隱藏性參數（ee）的實例變數。

如果我們需要支援 ChineseExam，那麼上面的程式可以進一步改寫如下：

```
class EnglishExam {
  public int vocab, grammar, listen;
  public int score() {
    return vocab + grammar + listen;
  }
}
class ChineseExam {
  public int word, sentence, composition;
  public int score() {
    return word + sentence + composition;
  }
}
public class Demo {
  public static void main(String argv[]) {
    EnglishExam ee = new EnglishExam();
    ee.vocab = 3; ee.grammar = 4; ee.listen = 5;
    ChineseExam cc = new ChineseExam();
    cc.word = 2; cc.sentence = 3; cc.composition = 4;
    System.out.print("The score of the English exam is ");
    System.out.println(ee.score());
    System.out.print("The score of the Chinese exam is ");
    System.out.println(cc.score());
  }
}
```

前面所提到的隱藏參數，可以使用 this 這個保留字來抓到。因此，上例也可以改寫成：

```
class EnglishExam {
  public int vocab, grammar, listen;
  public int score() {
    return this.vocab + this.grammar + this.listen;
  }
}
class ChineseExam {
  public int word, sentence, composition;
  public int score() {
    return this.word + this.sentence + this.composition;
  }
}
public class Demo {
  public static void main(String argv[]) {
    EnglishExam ee = new EnglishExam();
```

```
    ee.vocab = 3; ee.grammar = 4; ee.listen = 5;
    ChineseExam cc = new ChineseExam();
    cc.word = 2; cc.sentence = 3; cc.composition = 4;
    System.out.print("The score of the English exam is ");
    System.out.println(ee.score());
    System.out.print("The score of the Chinese exam is ");
    System.out.println(cc.score());
  }
}
```

this 的值是當使用一個實例呼叫一個實例方法（例如：ee.score()）時所傳入的隱藏性參數，而這個隱藏性參數便是那個實例（即：ee）的地址。因此，如果你在一個實例方法內更改一個實例變數的值時，那個實例變數的值便永遠被更改。

Java 可以使用 this 也可以不使用 this 來存取實例的變數或方法。在後續的單元，我們為了說明的需要，有時會使用 this，有時不使用 this，來存取實例變數或呼叫實例方法。

實例方法也可以有其他參數，例如，以下便是將權重傳入並計算分數的例子：

```
class EnglishExam {
  public int vocab, grammar, listen;
  public double score(double wb, double wg, double wl) {
    return wb * vocab + wg * grammar + wl * listen;
  }
}
public class Demo {
  public static void main(String argv[]) {
    EnglishExam ee = new EnglishExam();
    ee.vocab = 3; ee.grammar = 4; ee.listen = 5;
    System.out.print("The score of the English exam is ");
    System.out.println(ee.score(0.2, 0.3, 0.5));
  }
}
```

在前面的單元中我們曾經使用 int, double 等型態的變數，這種變數稱為 primitive type。而 ee 的型態則是 EnglishExam，這種型態稱為 reference type。一個 reference type 變數的初值是 null，代表沒有任何實例的地址被設成它的值。Java 的字串屬於 String 類別，而字串也是 reference type 的一種。一個變數如果是 primitive type，則它的地址中所存放的是數值本身。如果一個變數是 reference type，則這個變數的地址內所存放的是一個指向一個實例的地址。這個被指向的實例是放在 Java 自動化記憶體管理區內，如果這個實例沒有被任何其他變數直接或間接的透過其他實例指到，那麼 Java 的 garbage collector 便會在執行記憶體回收動作時將該實例的記憶體自動回收。

由於 Java 是一個「傳值呼叫」（call-by-value）的程式語言，所以當一個方法被呼叫時，是變數的值被傳入方法中。由於存放在 reference type 變數中的值，實際上是地址，所以是地址被傳入被呼叫的方法內。因此便會發生在程式執行中，同時有數個變數指向

同一個被 reference 的實例。而其中任何一個方法更改了那個實例的實例變數的值時，其他使用這個實例的方法也會看到被改變的新值。

CHAPTER

TEN

實例的產生與封裝

在前面的章節中，我們曾經使用 new 來產生一個 EnglishExam 的實例，然後個別的將這個實例的實例變數初始化。例如：

```
EnglishExam ee = new EnglishExam();
ee.vocab = 3;
ee.grammar = 4;
ee.listen = 5;
```

另一個達到同樣目的的方式是在類別內直接的初始化實例變數的值。例如：

```
class EnglishExam {
  public int vocab = 6,
    grammar = 7,
    listen = 8;
  public int score() {
    return vocab + grammar + listen;
  }
}
```

如果一個實例變數沒有初始化，那麼它的值是便會根據它的型態，而設定成該型態的 default value。

Java 也支援使用一個類別的 **constructor（建構子）** 將實例變數的值初始化。建構子與 class 同名，並且不需要定義 return type。呼叫建構子之後會製造一個物件並將那個物件回傳。例如：

```
class EnglishExam {
  public int vocab, grammar, listen;
  EnglishExam() {
    vocab = 6;
    grammar = 7;
    listen = 8;
  }
  public int score() {
    return vocab + grammar + listen;
```

```
  }
}

public class Exem {
  public static void main(String args[]) {
    EnglishExam ee = new EnglishExam();
    System.out.println("The score of the exam is: " + ee.score());
  }
}
```

執行結果:

```
The score of the exam is: 21
```

建構子也可以使用傳入的參數，將實例變數的值初始化。例如：

```
class EnglishExam {
  public int vocab, grammar, listen;
  EnglishExam() {
    vocab = 6; grammar = 7; listen = 8;
  }
  EnglishExam(int v, int g, int l) {
    vocab = v; grammar = g; listen = l;
  }
  public int score() {
    return vocab + grammar + listen;
  }
}

public class Exam {
  public static void main(String args[]) {
    EnglishExam ee = new EnglishExam();
    System.out.println("The score of the first exam is: " + ee.score());
    ee = new EnglishExam(7, 8, 9);
    System.out.println("The score of the second exam is: " + ee.score());
  }
}
```

這時的執行結果是:

The score of the first exam is: 21 The score of the second exam is: 24

以上的程式有一個沒有參數的建構子，及一個三個參數的建構子。如果一個類別內沒有定義建構子，那麼 Java 會自動提供一個沒有參數的建構子給這個類別。

Media:Ex9.swfl 觀看一個建構子範例的執行與解說

10.1 Getter, Setter 及 Data Abstraction

假設有一個 Exam 類別，而這個類別有一個 minute 的實例變數，而 e 則是一個 Exam 的實例。那麼使用 e.minute 便可以直接的取用它的值，然而 minute 卻必須宣告成 public。另一個存取實例變數的方式是使用 getter 及 setter 方法，這時程式設計師可以將 minute 宣告成 private，並選擇性的將 getter 及 setter 設定成所需要的存取權限。例如：

```
public class Exam {
  private int minutes;
  public Exam() {
    minutes = 80;
  }
  public int getMinutes() {
    return minutes;
  }
}
```

這時在其他的類別中若有一個 Exam 的實例 e，便可以使用 getMinutes 取出 e.minutes 的值：

```
e.getMinutes()
```

使用 getter 方法的好處之一是，能夠很容易的在取用實例變數的值時，增加協助偵錯的程式碼：

```
public class Exam {
  private int minutes;
  public Exam() {
    minutes = 80;
  }
  public int getMinutes() {
    System.out.println("Accessing minutes...");
    return minutes;
  }
}
```

setter 方法的作用也類似：

```
public class Exam {
  private int minutes;
    public Exam() {
    minutes = 80;
  }
  public int getMinutes() {
    return minutes;
  }
  public void setMinutes(int m) {
```

```
10      System.out.println("Setting minutes...");
11      minutes = m;
12    }
13  }
```

然而 setter 方法不需要傳回值。因此 setMinutes 的傳回值的型態是 void。

getter 與 setter 方法的另一個功用是，模擬不必真實的存在，但是卻可以透過運算而得到的實例變數。例如：

```
1  public class Exam {
2    private int minutes;
3    public Exam() {
4      minutes = 80;
5    }
6    public int getMinutes() {
7      return minutes;
8    }
9    public void setMinutes(int m) {
10     minutes = m;
11   }
12   public int getHours() {
13     return minutes/ 60.0
14   }
15   public void setHours(double h) {
16     minutes = (int)(h * 60);
17   }
18 }
```

使用以上的 getHours 及 setHours 使得 Exam 好像多了 hours 這個其實並不存在的實例變數一般。

使用 getter 及 setter 方法，可以在改變實例內資料的儲存方式後，不用修改其他使用此資料的程式碼，讓程式易於維護。例如：hours 比 minutes 更常用時，便可以將上面的範例，更改成以 hours 來儲存，以增加程式的效率：

```
1  public class Exam {
2    private double hours;
3    public Exam() {
4      hours = 1.5;
5    }
6    public int getMinutes() {
7      return (int)(hours * 60);
8    }
9    public void setMinutes(int m) {
10     hours = m / 60.0;
11   }
12   public double getHours() {
```

```
13      return hours;
14    }
15    public void setHours(double h) {
16      hours = h;
17    }
18  }
```

這時程式碼雖然做了更改，但是並不影響程式的其他部份。如果沒有使用 getter 與 setter，類似的更動，便需要將程式碼中所有存取 minutes 的地方全部做修改。例如：

```
1  x.minutes             需要更改成      (int)(x.hours * 60)
2  (x.minutes / 60.0)    則需要更改成    x.hours
```

使用 getter 及 setter 方法，間接的存取實例變數的程式設計方式稱為「資料抽象化」(data abstraction)。應用資料抽象化的方式寫程式，有以下的好處：

1. 程式碼容易再利用
2. 程式碼容易瞭解
3. 易於增加類別的功能
4. 易於改進資料的儲存方式

CHAPTER

ELEVEN

繼承、抽象類別

假設我們要將前面的 EnglishExam 及 ChineseExam 增加一個 minutes 的實例變數。那麼一個不好的方式是將這個實例變數分別的放入這兩個類別中：

```
class EnglishExam {
  public int minutes;
  public int vocab, grammar, listen;
  public int getMinutes() {
    return minutes;
  }
  public int score() {
    return vocab + grammar + listen;
  }
}
class ChineseExam {
  public int minutes;
  public int word, sentence, composition;
  public int getMinutes() {
    return minutes;
  }
  public int score() {
    return word + sentence + composition;
  }
}
public class Demo {
  public static void main(String argv[]) {
    EnglishExam ee = new EnglishExam();
    ee.minutes = 75;
    ee.vocab = 3; ee.grammar = 4; ee.listen = 5;
    ChineseExam cc = new ChineseExam();
    ee.minutes = 75;
    cc.word = 2; cc.sentence = 3; cc.composition = 4;
    System.out.print("The score of the English exam is ");
    System.out.println(ee.score());
```

```
31      System.out.println("English exam takes " + ee.getMinutes() + "minutes.");
32      System.out.print("The score of the Chinese exam is ");
33      System.out.println(cc.score());
34      System.out.println("Chinese exam takes " + cc.getMinutes() + "minutes.");
35    }
36  }
```

不好的原因是 EnglishExam 及 ChineseExam 中有許多 **重複的程式碼** 。而這些重複的程式碼會增加開發的成本及維護的負擔。所幸 Java 容許我們將類別設計成階層式的結構，而這個階層的結構可以自然的將不同類別間的繼承關係表達出來。因此，我們可以設計一個 Exam 類別，並將 minutes 宣告在 Exam 內；再由 EnglishExam 及 ChineseExam 繼承 Exam 即可。Java 使用 **extends** 這個保留字讓一個類別繼承另一個類別。

```
1   class Exam {
2     public int minutes;
3     public Exam() {
4           System.out.println("Calling Exam()...");
5           minutes = 75;
6     }
7     public int getMinutes() {
8           return minutes;
9     }
10  }
11
12  class EnglishExam extends Exam {
13    public int vocab, grammar, listen;
14    public int score() {
15          return vocab + grammar + listen;
16    }
17  }
18
19  class ChineseExam extends Exam {
20    public int word, sentence, composition;
21    public int score() {
22          return word + sentence + composition;
23    }
24  }
25
26  public class Demo {
27    public static void main(String argv[]) {
28          EnglishExam ee = new EnglishExam();
29          ee.vocab = 3; ee.grammar = 4; ee.listen = 5;
30          ChineseExam cc = new ChineseExam();
31          cc.word = 2; cc.sentence = 3; cc.composition = 4;
32          System.out.print("The score of the English exam is ");
33          System.out.println(ee.score());
```

```
34          System.out.println("English exam takes " + ee.getMinutes() + "minutes.");
35          System.out.print("The score of the Chinese exam is ");
36          System.out.println(cc.score());
37          System.out.println("Chinese exam takes " + cc.getMinutes() + "minutes.");
38    }
39  }
```

這時 Exam 是 EnglishExam 及 ChineseExam 的父類別，而 EnglishExam 及 ChineseExam 是 Exam 的子類別。Exam 也有一個父類別，這個類別是 Java 內建的 Object 類別。一個 Java 程式內所有的類別都直接或間接的繼承了 Object 類別。Exam 也可以寫成：

```
1  class Exam extends Object {
2    // ...
3  }
```

除了減少重複不必要的程式碼以外，類別的繼承還有以下兩個好處：

1. 讓父類別的程式碼可以在完全除錯後，才被子類別繼承。這樣可以讓程式的偵錯更為容易。
2. 可以購買軟體廠商已經開發好的類別，再透過繼承擴充其功能。

一個子類別的實例，含有自己類別的實例變數與方法，以及所有父類別的實例變數與方法。例如：ee 這個實例便有自己定義的 vocab, grammar, listen 及繼承而來的 minutes 四個實例變數及 score 及 getMinutes 兩個方法。

通常將實例變數與實例方法放置於父類別時，需要滿足以下兩個條件：

1. 可以減少重複的程式碼。
2. 父類別的實例變數或實例方法對子類別有用處。例如：Exam 內的 minutes 便對 ChineseExam 及 EnglishExam 有用。

當類別間有繼承關係時，建構子的呼叫順序是先執行父類別的建構子。例如：

```
1  class Exam {
2    public int minutes;
3    public Exam() {
4          System.out.println("Calling Exam()...");
5          minutes = 75;
6    }
7    public int getMinutes() {
8          return minutes;
9    }
10 }
11
12 class EnglishExam extends Exam {
13   public int vocab, grammar, listen;
14   public EnglishExam() {
```

```
        System.out.println("Calling EnglishExam()...");
        vocab = 7; grammar = 7; listen = 7;
  }
  public int score() {
        return vocab + grammar + listen;
  }
}

class ChineseExam extends Exam{
  public int word, sentence, composition;
  public ChineseExam() {
        System.out.println("Calling ChineseExam()...");
        word = 7; sentence = 7; composition = 7;
  }
  public int score() {
        return word + sentence + composition;
  }
}

public class Demo {
  public static void main(String argv[]) {
        EnglishExam ee = new EnglishExam();
        ChineseExam cc = new ChineseExam();
  }
}
```

會得到以下的執行結果：

```
Calling Exam()...
Calling EnglishExam()...
Calling Exam()...
Calling ChineseExam()...
```

如果我們需要將 EnglishExam 更加的細分。例如：增加一個 GREEnglishExam 類別，而這個類別的特性是 listen 實例變數的值是 0：

```
class Exam {
  public int minutes;
  public Exam() {
        minutes = 75;
  }
  public int getMinutes() {
        return minutes;
  }
}

class EnglishExam extends Exam {
  public int vocab, grammar, listen;
```

```
  public EnglishExam() {
        vocab = 7; grammar = 7; listen = 7;
  }
  public int score() {
        return vocab + grammar + listen;
  }
}

class GREEnglishExam extends EnglishExam {
  public int score() {
        return vocab + grammar + 0;
  }
}

class ChineseExam extends Exam{
  public int word, sentence, composition;
  public ChineseExam() {
        word = 7; sentence = 7; composition = 7;
  }
  public int score() {
        return word + sentence + composition;
  }
}

public class Demo {
  public static void main(String argv[]) {
        EnglishExam ee = new EnglishExam();
        GREEnglishExam gre = new GREEnglishExam();
        System.out.println("English exam score is " + ee.score());
        System.out.println("GRE English exam score is " + gre.score());
  }
}
```

這時執行的結果會得到：

```
English exam score is 21
GRE English exam score is 14
```

呼叫 gre.score() 得到 14 的原因是，GREEnglishExam 的 score 方法遮蔽了 EnglishExam 的 score 方法，而名稱相同的方法有以下兩種關係：

1. Overloading：是指參數輸入的個數或類別不同，但是卻同名的方法。
2. Shadowing：這是指數個方法同名，而參數的個數與型態也相同，但是卻分別的定義在不同的類別的方法。

以上例而言 gre.score() 會呼叫 GREEnglishExam 的 score 方法，而 ee.score() 則會呼叫 EnglishExam 的 score 方法。在執行時呼叫幾個同名的方法的哪一個，是根據實例所屬

的類別，例如：gre 的類別是 GREEnglishExam 所以 gre.score() 會呼叫 GREEnglishExam 的 score 方法。如果 GREEnglishExam 沒有定義被呼叫的方法，例如：gre.getMinutes()，這時則會呼叫其父類別的方法，如果父類別中也沒有定義，則會呼叫祖父類別的方法，依此類推。所以 gre.getMinutes() 會呼叫到 Exam 的 getMinutes 方法。

11.1 private 與 protected 變數與方法

在討論 getter, setter 方法時，我們談到資料抽象化的好處。但是如果那個實例變數本身（例如：minutes）仍然是定義成 public 那麼將失去強制性，也就是其他的程式設計師仍然可以繞過 getter 與 setter 方法而直接的存取 minutes。

private 與 protected 兩個保留字可以設定某個變數或方法的存取範圍。private 變數或方法的存取範圍為自己的類別內，而 protected 變數或方法則包括自己的類別、子類別及所屬的 package 內。至於一個 package 則是由數個為了完成某種功能的類別所組合而成。例如：一個類別 C 若屬於一個 package p，則 C 的程式碼必須以

```
package p;
```

起始，而且也必須存放在一個命名為 p 的目錄中。一個 Java 的檔案，如果要使用 C 或 p 所提供的功能，則可以用下兩種方式，輸入 C 或 p 內所有非 private 的名字：

```
import p.C;
import p.*;
```

一個 package 也可以使用

```
jar vcf p.jar p
```

指令將其中的.class 檔包裹起來，放在 jdk...jrelibext 的目錄中供其他程式使用。

private 與 protected 的變數與方法可以有許多交替使用的可能。例如：

```
class Exam {
  public Exam() {
        minutes = 75;
  }
  public int getMinutes() {
        return minutes;
  }
  public void setMinutes(int m) {
        minutes = m;
  }
  private int minutes;
}
```

以上的範例將 minutes 宣告為 private 而將 getMinutes, setMinutes 宣告為 public。這時只有

Exam 能直接存取 minutes，而程式中所有其他的類別都可以透過 getMinutes, setMinutes 間接的存取 minutes。Java 的習慣是將 private 的變數或方法宣告在 public 的變數或方法的下方。一個類別的 public 變數或方法，形成了這個類別對其他類別的 public interface 或公開的介面。

```
class Exam {
  public Exam() {
        minutes = 75;
  }
  public int getMinutes() {
        return minutes;
  }
  private int minutes;
}
```

而這個範例，則讓實例在初始化時便將 minutes 設值為 75，由於沒有 setMinutes 而 minutes 又是 private，所以其他類別的程式碼將無法更動 minutes 的值。

```
class Exam {
  public Exam() {
        minutes = 75;
  }
  public int getMinutes() {
        return minutes;
  }
  protected int minutes;
}
```

這個範例則容許 Exam 的子類別及與 Exam 位於同樣 package 內的類別直接存取 minutes。

```
class Exam {
  public Exam() {
        minutes = 75;
  }
  protected int getMinutes() {
        return minutes;
  }
  protected void setMinutes(int m) {
        minutes = m;
  }
  private int minutes;
}
```

而這個範例則只有 Exam 能直接存取 minutes，同時也只有 Exam 的子類別及位於相同 package 中的類別能夠透過 getMinutes 及 setMinutes 間接的存取 minutes。

11.2 建構子之間的呼叫

如果在建構一個 EnglishExam 實例時要同時傳入四個參數值給 vocab, grammar, listen, minutes 四個實例變數，並將其初始化。而且也要能夠只傳入三個參數值給 vocab, grammar, listen，那麼一種寫這些建構子的方式是：

```
class EnglishExam extends Exam {
  public int vocab, grammar, listen;
  public EnglishExam() {
       vocab = 6; grammar = 6; listen = 6;
  }
  public EnglishExam(int v, int g, int l) {
        vocab = v; grammar = g; listen = l;
  }
  public EnglishExam(int v, int g, int l, int m) {
        vocab = v; grammar = g;  listen = l;
        minutes = m;
  }
  public int score() {
        return vocab + grammar + listen;
  }
}
```

這個寫法有一個缺點就是

```
vocab = v; grammar = g; listen = l;
```

出現兩次。避免這些重複程式碼的方法是使用 this(v, g, l) 去呼叫那個三個參數的建構子：

```
.. code-block:: java

    class EnglishExam extends Exam { public int vocab,  grammar,  listen;  public En-
        glishExam() {

            vocab = 6; grammar = 6; listen = 6;

        } public EnglishExam(int v, int g, int l) {

            vocab = v; grammar = g; listen = l;

        } public EnglishExam(int v, int g, int l, int m) {

            this(v, g, l); minutes = m;

        } public int score() {

            return vocab + grammar + listen;

        }
```

```
    }
```

在建構子中使用 this(...) 指的是呼叫另一個，在相同的類別中，參數的個數與型態皆相同的建構子。要注意的是在建構子中使用 this(...)，一定要寫在建構子的第一行。

另一種可能是當 EnglishExam 與 ChineseExam 都需要呼叫一個參數的建構子將 minutes 初始化。一種不好的寫法是：

```
class EnglishExam extends Exam {
  // ...
  public EnglishExam(int m) {
    minutes = m;
  }
  // ...
}
class ChineseExam extends Exam {
  // ...
  public ChineseExam(int m) {
    minutes = m;
  }
  // ...
}
```

這時的

```
minutes = m;
```

也是同樣的重複在兩個類別中。改良的寫法是使用 super(m) 去呼叫定義在 Exam 類別內的一個參數的建構子：

```
class Exam {
  public int getMinutes() {
        return minutes;
  }
  public Exam() {
        minutes = 75;
  }
  public Exam(int m) {
        minutes = m;
  }
  private int minutes;
}
class EnglishExam extends Exam {
  ...
  public EnglishExam(int m) {
        super(m);
  }
  ...
}
```

```
class ChineseExam extends Exam {
  ...
  public ChineseExam(int m) {
        super(m);
  }
  ...
}
```

11.3 實例方法間的呼叫

如果你需要擴充 Exam、EnglishExam 與 ChineseExam，使它們能夠產出一份，包括考試時間、分數的考試資料。例如：

```
Chinese exam score: 88 Exam time: 75 minutes
English exam score: 76 Exam time: 60 minutes
```

第一種完成這個程式的寫法是為 EnglishExam 與 ChineseExam 都提供一個 report 方法：

```
class Exam {
  public int minutes;
  Exam(int m) {
        minutes = m;
  }
  public int getMinutes() {
        return minutes;
  }
}

class EnglishExam extends Exam {
  public int vocab, grammar, listen;
  EnglishExam(int v, int g, int l) {
        super(60);
        vocab = v;
        grammar = g;
        listen = l;
  }
  public int score() {
        return vocab + grammar + listen;
  }
  public void report() {
        System.out.println("English exam score: " + score() +
                           " Exam time: " + getMinutes() + " minutes");
  }
}

class ChineseExam extends Exam{
```

```
29    public int word, sentence, composition;
30    ChineseExam(int w, int s, int c) {
31          super(75);
32          word = w;
33          sentence = s;
34          composition = c;
35    }
36    public int score() {
37          return word + sentence + composition;
38    }
39    public void report() {
40          System.out.println("Chinese exam score: " + score() +
41                                  " Exam time: " + getMinutes() + " minutes");
42    }
43  }
44
45  public class Demo {
46    public static void main(String argv[]) {
47          ChineseExam cc = new ChineseExam(35, 35, 18);
48          cc.report();
49          EnglishExam ee = new EnglishExam(30, 20, 26);
50          ee.report();
51    }
52  }
```

你也可以使用 this 來呼叫方法，所上例中的 report 可以改寫如下:

```
1   public void report() {
2     System.out.println("English exam score: " + this.score() +
3                   " Exam time: " + this.getMinutes() + " minutes");
4   }
```

由於 getMinutes 方法是 Exam 提供給它的子類別的方法。所以也可以用 super 來呼叫這個方法：

```
1   public void report() {
2     System.out.println("English exam score: " + this.score() +
3                   " Exam time: " + super.getMinutes() + " minutes");
4   }
```

super 與 this 這兩個保留字，同時用在建構子中與方法的呼叫，但是意義完全不同。用於建構子中的呼叫時，super 與 this 是用來呼叫其他的建構子；用於方法的呼叫時，super 是指呼叫自己或祖先中同名且參數一致的方法；而 this 甚至有可能呼叫到子孫類別中同名且參數一致的方法。由於 this 是在方法呼叫時才傳入的隱藏性參數，指的是實例本身，並不一定是指 this 這個字所在的類別，所以要看 this 呼叫時的實例為何，才能知道是那個方法被呼叫。

另一個寫這個程式的方式是將 report 拆開並分別放在父類別與子類別中：

```
class Exam {
  public int minutes;
  Exam(int m) {
        minutes = m;
  }
  public int getMinutes() {
        return minutes;
  }
  public void report() {
        System.out.println(" Exam time: " + this.getMinutes() + " minutes");
  }
}

class EnglishExam extends Exam {
  public int vocab, grammar, listen;
  EnglishExam(int v, int g, int l) {
        super(60);
        vocab = v;
        grammar = g;
        listen = l;
  }
  public int score() {
        return vocab + grammar + listen;
  }
  public void report() {
        System.out.println("English exam score: " + this.score());
        super.report();
  }
}

class ChineseExam extends Exam{
  public int word, sentence, composition;
  ChineseExam(int w, int s, int c) {
        super(75);
        word = w;
        sentence = s;
        composition = c;
  }
  public int score() {
        return word + sentence + composition;
  }
  public void report() {
        System.out.println("Chinese exam score: " + this.score());
        super.report();
  }
```

```
}

public class Demo {
  public static void main(String argv[]) {
        ChineseExam cc = new ChineseExam(35, 35, 18);
        cc.report();
        EnglishExam ee = new EnglishExam(30, 20, 26);
        ee.report();
  }
}
```

當使用 super 呼叫一個方法（report）時，Java 會忽略定義在目前類別（EnglishExam 或 ChineseExam）的同名的 report 方法，而會從目前類別的父類別（Exam）開始往上搜尋，找到後，便呼叫那個方法。以上例而言，就是定義在 Exam 中的 report 方法。

11.4 抽象類別

還有一個寫這個程式的方式是使用抽象類別（abstract class）。抽象類別的功用是為它的子類別們提供共用的變數與方法。在抽象類別中可以宣告抽象方法（abstract method），抽象方法沒有程式碼的具體定義，它只有方法的名稱及型態的宣告。在一個抽象類別中宣告一個抽象方法，便必需要在這個抽象類別的直接子類別（direct subclass）定義與這個抽象方法同名、同型態的方法，要不然會發生編譯錯誤。

如果要在 Exam 的子類別強迫定義 score 方法，並且只用 Exam 的 report 印出 score、time 及 examName 的資料。那麼這個程式可以改寫成以下這個沒有重複程式碼的版本：

```
abstract class Exam {
  private int minutes;
  private String examName;
  Exam(String n, int m) {
        examName = n;
        minutes = m;
  }
  public int getMinutes() {
        return minutes;
  }
  public String getExamName() {
        return examName;
  }
  public void report() {
        System.out.println(this.getExamName() + "score: " +
        this.score() + " Exam time: " + this.getMinutes() + " minutes");
  }
  abstract int score();
```

```
}

class EnglishExam extends Exam {
  public int vocab, grammar, listen;
  EnglishExam(int v, int g, int l, String n) {
        super(n, 60);
        vocab = v;
        grammar = g;
        listen = l;
  }
  public int score() {
        return vocab + grammar + listen;
  }
}

class ChineseExam extends Exam {
  public int word, sentence, composition;
  ChineseExam(int w, int s, int c, String n) {
        super(n, 75);
        word = w;
        sentence = s;
        composition = c;
  }
  public int score() {
        return word + sentence + composition;
  }
}

public class Demo {
  public static void main(String argv[]) {
        Exam ex;
        ex = new ChineseExam(35, 35, 18, "Chinese exam");
        ex.report();
        ex = new EnglishExam(30, 20, 26, "English exam");
        ex.report();
  }
}
```

在 Exam 類別內的 report 方法所用到的 this 可以有兩種可能的實例與之對應：ChineseExam 的實例與 EnglishExam 的實例。而呼叫在 main 中的 ex.report() 會繼續呼叫到 this.score()，這時實際上被呼叫的 score 方法有兩種可能：

1. 呼叫到定義在 ChineseExam 中的 score，如果 ex 的值是一個 ChineseExam 的實例。
2. 呼叫到定義在 EnglishExam 中的 score，如果 ex 的值是一個 EnglishExam 的實例。

因此 this 的意義不是指出現 this 這個字的類別（Exam），而是指用於呼叫方法（score）

的實例，這個實例就是 this。以上例而言是 ex，而 ex 的值可以是一個 ChineseExam 的實例，也可以是一個 EnglishExam 的實例，而 ChineseExam 與 EnglishExam 都是 Exam 的子類別，所以 this.score() 實際上 **可能呼叫到自己的子孫所定義的方法** 。

一個抽象類別的主要功用是為它的子類別提供共用的變數與方法。抽象類別內可以宣告抽象方法。但是抽象類別不能產生實例，所以以下的程式碼會產生編譯錯誤：

```
Exam ex = new Exam();
```

抽象類別的抽象方法強制其直接子類別必須定義同名、同型態的實體方法，而 Java 的編譯器可以檢查這個規定是否被達成，因此可以減輕程式設計師自行檢查某個類別是否符合設計需求的負擔。

雖然抽象類別不能產生實例，但是卻可以用於宣告實例變數的型態。例如：

```
public class Demo {
  public static void main(String argv[]) {
        Exam ex;
        ex = new ChineseExam(35, 35, 18, "Chinese exam");
        ex.report();
        ex = new EnglishExam(30, 20, 26, "English exam");
        ex.report();
  }
}
```

ex 這個變數的型態是 Exam，因為 ChineseExam 與 EnglishExam 都是一種 Exam，所以 ex 可以儲存 ChineseExam 的實例，也可以儲存 EnglishExam 的實例，因為 ChineseExam 與 EnglishExam 不但是 Exam 的子類別，也是 Exam 的 **子型態（subtype）** 。這便是 ChineseExam 與 Exam 有 **is-a** 的關係，也就是一個 ChineseExam 是一個（is-a）Exam；而一個 EnglishExam 也是一個 Exam。

但是型態被宣告為 Exam 的變數，只能用於呼叫宣告在 Exam 內的方法。假設 ChineseExam 內有一個 getWord 的方法，但是在 Exam 中沒有這個方法，那麼 ex.getWord() 將會產生編譯錯誤，因為 ex 實例的型態沒有這個方法。

如果確實有需要呼叫 getWord 方法，則可使用轉型的方式進行呼叫。例如：

```
public class Demo {
  public static void main(String argv[]) {
        Exam ex;
        ex = new ChineseExam(35, 35, 18, "Chinese exam");
        System.out.println("Word score: " + (ChineseExam)ex.getWord();
  }
}
```

抽象類別有一個特性，它不可能是在一個類別繼承結構的最下層。如果需要，程式設計師可以宣告一個最下層的類別為 final，一個 final 的類別不能被其他類別繼承。

11.5 物件導向程式的設計原則

如同前面的範例所展示的，一個問題可能有好幾種不同的解法。到底哪一種方法比較好？在什麼狀況用哪種解法呢？以下是設計物件導向程式的幾個參考原則：

1. 類別的結構與實際一致。
2. 高層次的類別歸納一般化的屬性（general properties），例如：Exam 或生物；而低層次類別的屬性則比較特殊化（specialize），例如：EnglishExam 或人類。
3. 沒有重複的程式碼。
4. 直接存取常用的資訊，避免反覆的透過計算得到。
5. 將其他類別不需要用的或不需要知道的變數與方法以 private 或 protected 隱藏起來。
6. is-a 與 has-a 的適當設計：以前例而言，ChineseExam 與 EnglishExam 都是一種（is-a）Exam。所以這時將它們設計成父類別與子類別的關係便很正確。但是，一個考生（a Student）的資料卻不能因為所有的 Exam 都有考生，而將 Student 設計成 EnglishExam 與 ChineseExam 的父類別。然而，將 Student 設計成一個單一的類別，並在 Exam 中宣告一個能夠存放考生實例的實例變數，卻很恰當。因為一份考卷 has-a 考生。這種在類別中宣告實例變數，以儲存其他類別所產生的實例，稱為（**has-a** ）的關係。

11.6 圖形與動畫的範例

到目前為止，我們已經闡述了四種呼叫 Java 方法的方式，這四種方式是：

1. 呼叫類別方法
2. 以實例來呼叫實例方法
3. 以 this 來呼叫實例方法
4. 以 super 來呼叫實例方法

在這四種方式中最單純的是呼叫類別方法，因為呼叫一個類別方法可以藉由看靜態（static）的程式碼即可確定是哪一個類別方法被呼叫到。

以 super 來呼叫實例方法也很單純，其判斷的方式是以程式碼的繼承關係來決定。例如在 C 類別中呼叫 super.m() 則可以從 C 的父類別開始依序向上（祖先們）搜尋，而搜尋到的第一個 m 即為被呼叫的方法。

「以實例來呼叫實例方法」及「以 this 來呼叫實例方法」則不能單單的看靜態的程式碼，而必須要依照程式在動態執行時所使用的實例是哪一個類別的實例才能決定。為了能明確說明，Java 決定哪一個實例方法被執行的的機制（這個機制稱為動態搜尋或 dynamic lookup），在這一節我們使用了幾個能夠動態顯現的圖形與動畫，並將實例以一層包含一層的方式及將類別以樹狀階層圖的方式對照，來明確的解說程式執行時的實例或 this 到底是哪一個類別的實例，以決定是那個方法被執行。

如果一個程式有以下幾個類別：

```
Class A{}
Class B extends A{}
Class C extends A{}
Class D extends B{}
Class E extends B{}
```

那麼它們之間的繼承與所產生的實例有以下的關係。其中繼承是以樹狀階層圖表示，而實例則以對應的多層次的同心圓來表示，同心圓的最內層對應最 super 的類別，而同心圓的最外層則對應這個實例所屬的類別：

Image:inheritance.jpg

以下是另一個圖形與動畫的範例。這個範例說明了父類別之物件不可以使用子類別之方法，但子類別之物件可以使用父類別之方法。程式碼如下：

```
class Make_counter {
        int count;
        void add1() {
                count = count + 1;
        }
        void add3() {
                add1();
                add1();
                add1();
        }
        int getCount() {
                return count;
        }
}

class Make_counter2 extends Make_counter {
        void add2() {
                add1();
                add1();
        }
}

public class Ex6 {
```

```
        public static void main(String argv[]) {
                Make_counter2 c2 = new Make_counter2();
                c2.add2();
                c2.add3();
                System.out.println("count:" + c2.getCount());
                Make_counter c1 = new Make_counter();
                // c1.add2(); => error
        }
}
```

執行結果：

```
count:5
```

Media:Ex6.swfl 觀看執行動畫

11.7 Super, this, abstract class 的動畫範例

以下的這個範例使用有層次的實例與動畫，說明 this, super 的特性:

```
class Father {
        String a = "father";
}
class Son extends Father {
        String a = "son";
        String get_null() {
                return a;
        }
        String get_this() {
                return this.a;
        }
        String get_super() {
                return super.a;
        }
}
public class Ex7 {
        public static void main(String argv[]) {
                Son s = new Son();
                System.out.println("a:" + s.get_null());
                System.out.println("this.a:" + s.get_this());
                System.out.println("super.a:" + s.get_super());
        }
}
```

執行結果:

```
a:son
this.a:son
super.a:father
```

這個範例中的 s 是一個 Son 的實例，而 Son 的父類別是 Father，因此 s 實例有兩層，外層是宣告於 Son 中的實例變數與方法，而內層是宣告於 Father 的實例變數與方法。get_this 方法中的 this 即是 s；而 get_super 中的 super 指的是 s 的內層相對於 Father 的部分。

Media:Ex7.swfl 觀看執行動畫

以下這個範例說明類別間有同名而且型態也相同的方法（overwrite）的特性:

```
class GrandParent {
        String eyes() {
                return "blue";
        }
}
class Parent extends GrandParent {
        String eyes() {
                return "green";
        }
}
public class Ex8 {
        public static void main(String args[]) {
                GrandParent gail = new GrandParent();
                Parent sue = new Parent();
                System.out.println(" 當子類別擁有與父類別同方法名稱時稱為 overwrite");
                System.out.println("gail.eyes():" + gail.eyes());
                System.out.println("sue.eyes():" + sue.eyes());
        }
}
```

執行結果:

> 當子類別擁有與父類別同方法名稱時稱為 overwrite gail.eyes():blue
> sue.eyes():green

GrandParent 由於沒有父類別（除了 Object 以外），所以 gail 的實例只有一層；然而，sue 的實例卻有兩層，外層對應 Parent，內層對應 GrandParent。

Media:Ex8.swfl 觀看執行動畫

以下這個範例將透過宣告在數個不同類別的實例方法的呼叫，更深入的闡釋 super 與 this 的特性:

```
class A {
        String color() {
                return "blue";
```

```
        }
        String getColor() {
                return this.color();
        }
}
class B extends A {
        String color() {
                return "green";
        }
        String getColor1() {
                return this.color();
        }
        String getColor2() {
                return super.color();
        }
}
class C extends B {
        String color() {
                return "red";
        }
        String getColor3() {
                return this.color();
        }
        String getColor4() {
                return super.color();
        }
}
public class Ex10 {
        public static void main(String args[]) {
                A a = new A();
                B b = new B();
                C c = new C();
                System.out.println("a.color():" + a.color());
                System.out.println("a.getColor():" + a.getColor());
                System.out.println("b.color():" + b.color());
                System.out.println("b.getColor1():" + b.getColor1());
                System.out.println("b.getColor2():" + b.getColor2());
                System.out.println("c.color():" + c.color());
                System.out.println("c.getColor():" + c.getColor());
                System.out.println("c.getColor1():" + c.getColor1());
                System.out.println("c.getColor2():" + c.getColor2());
                System.out.println("c.getColor3():" + c.getColor3());
                System.out.println("c.getColor4():" + c.getColor4());
        }
}
```

執行結果:

```
a.color():blue
a.getColor():blue
b.color():green
b.getColor1():green
b.getColor2():blue
c.color():red
c.getColor():red
c.getColor1():red
c.getColor2():blue
c.getColor3():red
c.getColor4():green
```

這個程式最讓人訝異的是呼叫 c.getColor() 的結果是 red。因為 c.getColor 會呼叫宣告在 A 之內的 getColor；而 getColor 內的 this 指的是 c，不是 A；而 c 的 getColor 會傳回 red。

Media:Ex10.swfl 觀看執行動畫

以下的範例將使用「寵物 (Pet)」類別及「狗 (Dog)」及「貓 (Cat)」兩個子類別。狗及貓都擁有「聲音 (sound)」這個方法，但狗跟貓的叫聲卻是不同的，此時我們可以利用抽象類別來實作這個例子。

```
abstract class Pet {
        abstract String sound();
}
class Dog extends Pet {
        String sound() {
                return " 汪汪";
        }
}
class Cat extends Pet {
        String sound() {
                return " 喵喵";
        }
}
public class Ex11 {
        public static void main(String args[]) {
                Dog d = new Dog();
                Cat c = new Cat();
                System.out.println("d.sound():" + d.sound());
                System.out.println("c.sound():" + c.sound());
        }
}
```

執行結果:

```
d.sound(): 汪汪
c.sound(): 喵喵
```

上例中 d 與 c 兩個變數若宣告成 Pet 型態，則答案仍然會一樣。

Media:Ex11.swfl 觀看執行動畫

以下這個範例是一個整合了 abstract class, super, this, array 的應用。這個範例的特色是宣告了 Area 這個 abstract class 及 getArea 這個抽象方法，讓 Square 及 Triangle 來繼承，而 whichBig 這個方法則可以比較任何兩個 Area 實例（Square 及 Triangle 的實例都是 Area 的實例），何者的面積較大。此外，在 main 中的 a 陣列，也可以存放任何的 Area 的實例，因此 a 可以存放 Square 的實例，也可以存放 Triangle 的實例：

```
abstract class Area {
        int high, weight;
        Area(int a, int b) {
                high = a;
                weight = b;
        }
        abstract int getArea();
        boolean whichBig(Area p) {
                return (this.getArea() > p.getArea());
        }
}
class Square extends Area {
        Square(int a, int b) {
                super(a, b);
        }
        int getArea() {
                return high * weight;
        }
}
class Triangle extends Area {
        Triangle(int a, int b) {
                super(a, b);
        }
        int getArea() {
                return (high * weight) / 2;
        }
}
public class Ex12 {
        public static void main(String args[]) {
                int total = 0;
                Area[] a = { new Square(6, 6), new Triangle(10, 10) };
                System.out.println("Is a[0] bigger than a[1]?" + a[0].whichBig(a[1]));
                for (int i = 0; i &lt; a.length; i++) {
                        total = total + a[i].getArea();
                }
                System.out.println("total:" + total);
        }
```

```
38 }
```

執行結果:

```
1 Is a[0] bigger than a[1]?false
2 total:86
```

Media:Ex13new.swfl 觀看執行動畫及詳細解說

CHAPTER

TWELVE

介面

前一個單元，介紹了以抽象類別內的抽象方法，來將需求強制加在抽象類別的直接子類別上。這種功能提供了程式設計師必需先想、先設計，然後才開始寫程式的好習慣。這個功能還提供了程式設計師以下的可能：

1. 將額外的註解放入抽象方法中，讓程式碼有更容易瞭解；
2. 可以將一個類別交給其他人來寫；
3. 如果某個程式設計師忘了按照需求定義抽象方法所強制的名稱與類別，編譯器可以自動產生錯誤信息。

除了抽象類別能宣告抽象方法之外，Java 還提供了 **介面（interface）** 來宣告抽象方法。不同於抽象類別：

1. 介面只提供抽象方法或常數宣告，而不能定義變數或一般的方法。
2. 一個子類別只能繼承一個父類別，但卻可以 **實作（implements）** 許多個介面。因此，介面模擬了 **多重繼承（multiple inheritance）** 的功能，但是卻可以避免一個類別繼承了多個父類別，但卻可能同時繼承了多個同名的變數或方法而造成混淆。

假設在設計 Exam 這個程式時，你已經想過要提供一個 score 方法給 ChineseExam 與 EnglishExam。這時你可以先將 ScoreInterface 設計如下：

```
1 public interface ScoreInterface {
2   public abstract int score();
3 }
```

而 ChineseExam 與 EnglishExam 則 implements 這個 interface:

```
1 class ChineseExam extends Exam implements ScoreInterface {
2   //...
3 }
4 class EnglishExam extends Exam implements ScoreInterface {
5   //...
6 }
```

這時 ChineseExam 與 EnglishExam 便必須提供 score 方法，而且要有同樣的輸出入參數的型態，要不然 Java 的編譯器便會產生錯誤信息。因此使用 interface 的一大好處是讓 Java 的編譯器，也可以幫助維持程式的一致性；而不用透過人工，這個容易出錯的方式來維持。

Interface 也可以成為一個變數的型態，而所宣告的變數可以用來呼叫這個 interface 內宣告的方法。例如：

```
// ...

public class Demo {
  public static void main(String argv[]) {
        ScoreInterface ex;
        ex = new ChineseExam(35, 35, 18, "Chinese exam");
        ex.score();
        ex = new EnglishExam(30, 20, 26, "English exam");
        ex.score();
  }
}
```

然而，如果上例的 ChineseExam 與 EnglishExam 類別中定義了 report 方法，而 ScoreInterface 中卻沒有宣告。這時下列的程式碼便會在編譯時產生型態錯誤：

```
// ...

public class Demo {
  public static void main(String argv[]) {
        ScoreInterface ex;
        ex = new ChineseExam(35, 35, 18, "Chinese exam");
        ex.report();
        ex = new EnglishExam(30, 20, 26, "English exam");
        ex.report();
  }
}
```

改正的方式是透過轉型：

```
// ...

public class Demo {
  public static void main(String argv[]) {
        ScoreInterface ex;
        ex = new ChineseExam(35, 35, 18, "Chinese exam");
        ((ChineseExam)ex).report();
        ex = new EnglishExam(30, 20, 26, "English exam");
        ((EnglishExam)ex).report();
  }
}
```