THE COMPLETE JAVA PROGRAMMING COURSE

Java 程式設計
完整學習教材

初級篇 BEGINNER LEVEL

從零開始學習 Java：建置開發環境、掌握變數與資料型別、運算子、流程控制、陣列與方法，學會字串處理，最後推開物件導向程式設計的大門。適合完全沒有程式設計經驗的初學者。

以 Java 17 LTS 為基準 · 涵蓋 Java 21 重要新特性 · 所有範例皆附詳細中文註解 · 編譯日期：2026 年 8 月 14 日

如何使用本教材

《Java 程式設計完整學習教材》共分三冊，對應三個學習階段：初級篇、中級篇與進階篇。每一章都包含觀念講解、大量附詳細中文註解的範例程式、預期執行結果、常見錯誤提醒，以及章末的動手練習。三冊循序漸進，完整走完即具備從零基礎到資深工程師所需的 Java 核心知識。

三冊內容總覽

冊別	適合對象	主要內容
初級篇	沒有程式設計經驗的初學者	開發環境、變數與資料型別、運算子、流程控制、陣列、方法、字串處理、物件導向入門
中級篇	已掌握 Java 基礎語法的學習者	封裝、繼承與多型、介面、例外處理、集合框架、泛型、檔案與 I/O、列舉與內部類別、綜合實戰專案
進階篇	邁向資深工程師的開發者	Lambda 與函數式程式設計、Stream API、多執行緒與並行、反射與註解、JVM 原理、網路與 JDBC、設計模式與現代 Java

閱讀慣例

內文中的程式碼片段會以等寬字型呈現，例如 `System.out.println()` 與 `List<String>`。完整的範例程式則放在如下的程式碼框中，每個範例都盡可能加上詳細的中文註解——註解正是本教材的精華所在：

範例：Hello.java ——本教材的程式碼呈現方式

```
/**
 * 教材中的每個範例都會像這樣：
 * 用文件註解說明類別的用途，用行內註解解釋每個步驟的意圖。
 */
public class Hello {
    public static void main(String[] args) {
        // 將訊息輸出到終端機（主控台）
        System.out.println("你好，Java！");
```

```
    }
}
```

▶ 執行結果

```
你好，Java！
```

此外，教材中穿插四種資訊框，各有不同的用途：

> **提示**　提供實用的小技巧與業界最佳實務，幫助你寫出更專業的程式。

> **注意**　標示初學者常犯的錯誤與容易踩到的陷阱，閱讀時請特別留意。

> **說明**　補充延伸知識與底層原理說明，第一次閱讀時可以先略過，回頭再讀。

> **動手練習**
>
> 每章最後都有動手練習。程式設計是「做中學」的技能，請務必實際完成。

環境需求

- **JDK 17 或更新版本**：本教材以 Java 17 LTS 為基準，並在適當章節介紹 Java 21 的重要新特性（如虛擬執行緒、switch 模式匹配）。
- **文字編輯器或 IDE**：初期建議用命令列搭配任一編輯器，熟悉編譯流程後再改用 IntelliJ IDEA 或 VS Code 提升效率。
- **終端機**：用來執行 `javac`（編譯）與 `java`（執行）指令。

學習方法建議

1. **每個範例都親手輸入並執行**：不要複製貼上，打字的過程就是在建立語法的肌肉記憶。
2. **先讀註解、再讀程式**：範例中的註解解釋了「為什麼這樣寫」，而不只是「寫了什麼」。
3. **改壞它，再修好它**：試著修改範例、觀察編譯器的錯誤訊息，這是理解 Java 編譯器脾氣的最快方法。
4. **完成章末練習**：練習題附有提示，請先自行嘗試，卡住了再看提示。

目錄

如何使用本教材 1

1 認識 Java 與建置開發環境 7

1.1 Java 是什麼 . . . 7
1.2 JVM、JRE 與 JDK：三兄弟的分工 . . . 9
1.3 安裝 JDK 17 . . . 10
1.4 第一個程式：HelloWorld . . . 11
1.5 註解：寫給人看的說明 . . . 13
1.6 看懂編譯錯誤訊息 . . . 14
1.7 選擇你的開發工具 . . . 16
1.8 本章小結 . . . 16
1.9 本章練習 . . . 17

2 變數與資料型別 18

2.1 變數：帶標籤的盒子 . . . 18
2.2 命名規則與慣例 . . . 19
2.3 八種基本型別 . . . 20
2.4 型別轉換 . . . 24
2.5 final 常數 . . . 26
2.6 var：讓編譯器幫你寫型別 . . . 27
2.7 基本型別與參考型別初探 . . . 28
2.8 用 Scanner 讀取使用者輸入 . . . 29
2.9 本章小結 . . . 30
2.10 本章練習 . . . 30

3 運算子與運算式 32

3.1 算術運算子 . . . 32
3.2 遞增與遞減 . . . 34
3.3 指定與複合指定運算子 . . . 35

3.4 比較運算子 . . . 36
3.5 邏輯運算子與短路求值 . . . 37
3.6 位元運算子簡介 . . . 38
3.7 三元運算子 . . . 39
3.8 字串串接的加號陷阱 . . . 40
3.9 運算子優先順序 . . . 41
3.10 浮點數的精度問題 . . . 42
3.11 綜合範例：BMI 計算機 . . . 43
3.12 本章小結 . . . 44
3.13 本章練習 . . . 45

4 流程控制 46

4.1 if：讓程式學會判斷 . . . 46
4.2 switch：多選一的專用語法 . . . 48
4.3 while：先檢查再執行 . . . 51
4.4 do-while：先執行再檢查 . . . 52
4.5 for：計數迴圈的標準款 . . . 53
4.6 巢狀迴圈：九九乘法表 . . . 54
4.7 break 與 continue . . . 55
4.8 綜合範例：猜數字遊戲 . . . 57
4.9 本章小結 . . . 58
4.10 本章練習 . . . 59

5 陣列 60

5.1 宣告與建立陣列 . . . 60
5.2 走訪陣列：兩種 for 的選擇 . . . 62
5.3 陣列是參考型別 . . . 63
5.4 陣列基礎演算法 . . . 65
5.5 Arrays 工具類別 . . . 67
5.6 二維陣列 . . . 68
5.7 綜合範例：成績統計程式 . . . 70
5.8 本章小結 . . . 72

5.9 本章練習 72

6 方法 73

6.1 為什麼需要方法 73
6.2 參數、引數與回傳值 75
6.3 Java 是傳值呼叫 76
6.4 方法多載 77
6.5 可變參數 varargs 79
6.6 遞迴：呼叫自己的方法 80
6.7 變數作用域 82
6.8 綜合範例：重構簡易計算機 83
6.9 本章小結 85
6.10 本章練習 85

7 字串處理 86

7.1 建立字串與不可變性 86
7.2 比較字串：== 與 equals 87
7.3 常用方法巡禮 88
7.4 字串與數字互轉 92
7.5 StringBuilder：可變的字串工作檯 93
7.6 格式化輸出 95
7.7 綜合範例：文字分析器 97
7.8 本章小結 99
7.9 本章練習 99

8 物件導向入門 101

8.1 為什麼需要物件導向 101
8.2 類別與物件：設計圖與成品 102
8.3 建構子：物件的出廠設定 106
8.4 封裝：把資料鎖進保險箱 107
8.5 static：屬於類別的成員 109
8.6 套件與 import 111
8.7 綜合範例：BankAccount 112

8.8 本冊總結與下一步 115
8.9 本章小結 115
8.10 本章練習 115

第 1 章

認識 Java 與建置開發環境

本章學習目標

學完本章你將能夠：

- 說出 Java 的設計理念「一次編寫、到處執行」，並解釋它背後的原理。
- 分辨 JVM、JRE、JDK 三者的角色與關係。
- 在自己的電腦上安裝 JDK 17，並用指令驗證安裝成功。
- 親手撰寫、編譯並執行你人生的第一個 Java 程式。
- 看懂常見的編譯錯誤訊息，並自行修正。
- 使用三種註解，養成為程式寫說明的好習慣。

歡迎踏上 Java 的學習之旅！學程式和學一門外語很像：一開始要先認識這門語言的背景與「發音規則」，接著開口說出第一句話。本章就是這趟旅程的起點——我們會先認識 Java 是什麼、為什麼全世界有數百萬名工程師靠它吃飯，然後把開發環境架設起來，最後親手寫出第一個能執行的程式。

請不要小看「安裝環境」與「印出一行字」這兩件事。根據多年的教學經驗，初學者最容易在第一天就被環境問題勸退；反過來說，只要今天能讓電腦聽話地印出一行訊息，後面的學習就會順利得多。萬事起頭難，我們把「難」的部分拆成一小步一小步，慢慢來。

1.1 Java 是什麼

Java 是一種「物件導向」的程式語言，1995 年由昇陽電腦（Sun Microsystems）的 James Gosling 團隊發表，2010 年隨著昇陽被甲骨文（Oracle）收購，Java 也轉由 Oracle 主導發展。它誕生至今將近三十年，始終穩居全世界最多人使用的程式語言前幾名。你每天使用的許多服務——網路銀行的後端系統、電商平台、Android 手機上的許多應用程式、甚至遊戲《Minecraft》——背後都有 Java 的身影。

1.1.1 一次編寫、到處執行

Java 最著名的口號是「一次編寫、到處執行」（Write Once, Run Anywhere）。要理解這句話的價值，可以先看看傳統編譯式語言（如 C 語言）的困境：C 的程式會被編譯成「特定作業系統與特定硬體」才看得懂的機器碼，所以同一份程式想在 Windows、macOS 與 Linux 上執行，就得

分別編譯三次，還常常要針對各平台修改程式碼。

Java 的解法非常聰明：程式不直接編譯成機器碼，而是編譯成一種中間格式，稱為「位元組碼」（bytecode）。位元組碼不是給任何實體電腦看的，而是給一台「虛擬的電腦」看的——這台虛擬電腦就是 Java 虛擬機器（JVM，Java Virtual Machine）。只要某個平台上有對應版本的 JVM，同一份位元組碼就能原封不動地在那裡執行。換句話說：

- 開發者只需要寫一份程式、編譯一次。
- 「跨平台」這件苦差事，交給各平台的 JVM 去煩惱。
- 你在 macOS 上編譯出來的程式，拿到 Windows 的 JVM 上照樣能跑。

1.1.2 Java 的應用領域與生態系

Java 的生態系極為龐大，這也是企業持續選用它的主因——幾乎任何需求都找得到成熟的函式庫與框架。常見的應用領域包括：

- **企業後端系統**：Spring 框架家族是全球網站後端的主流選擇，銀行、保險、電信等大型系統大量採用 Java。
- **Android 應用程式**：Android 官方開發語言長期以 Java 為核心（近年加入 Kotlin，而 Kotlin 也執行在 JVM 上）。
- **大數據處理**：Hadoop、Kafka、Elasticsearch 等重量級大數據工具都以 Java（或 JVM 語言）打造。
- **金融與交易系統**：Java 的穩定性與效能使它成為證券交易、支付清算系統的常客。
- **遊戲與工具**：《Minecraft》以 Java 開發；IntelliJ、Eclipse 這些開發工具本身也是 Java 寫的。

下表列出 Java 發展史上幾個關鍵版本。本教材以 Java 17 為基準——它是長期支援版本（LTS，Long-Term Support），語法現代、業界採用率高，非常適合作為學習的起點。

年份	版本	重要意義
1995	Java 1.0	正式誕生，主打「一次編寫、到處執行」
2004	Java 5	加入泛型、列舉等重要語法
2014	Java 8	Lambda 與 Stream，劃時代的版本
2021	Java 17	長期支援版本（LTS），本教材的基準
2023	Java 21	目前最新的長期支援版本

> **說明**　自 2017 年起，Java 改為每半年發布一個新版本，其中每兩年挑一版作為 LTS（長期支援版本）提供多年的更新維護。企業與教學通常鎖定 LTS 版本，例如 11、17、21。

> 你在本書學到的內容，在更新的版本上一樣適用。

1.2 JVM、JRE 與 JDK：三兄弟的分工

初學者第一個容易混淆的地方，就是這三個長得很像的縮寫。可以用「餐廳廚房」來比喻：JVM 是爐子（真正把菜煮熟的核心設備）、JRE 是完整的外場與廚房設備（讓一道菜能被端上桌的所有東西）、JDK 則是整間附設烹飪教室的餐廳（除了能出菜，還提供讓你「開發新菜色」的全套工具）。

名稱	全名	內容與用途
JVM	Java Virtual Machine	執行位元組碼的虛擬機器，是跨平台的關鍵核心
JRE	Java Runtime Environment	JVM 加上核心類別庫；只需要「執行」Java 程式的人裝這個就夠
JDK	Java Development Kit	JRE 加上編譯器與各種開發工具；要「開發」程式的我們必須安裝它

三者是「包含」關係：JDK 包含 JRE，JRE 包含 JVM。身為開發者，你只要記得一件事：**安裝 JDK 就對了**，另外兩個會一起附上。

1.2.1 一支程式的旅程：從原始碼到執行

理解了三兄弟的分工後，我們把「寫好的程式如何被執行」的完整流程走一遍。假設你寫了一個檔案叫 `Hello.java`：

1. **撰寫原始碼**：你用文字編輯器寫下人類看得懂的 Java 程式，存成副檔名為 `.java` 的檔案。
2. **編譯**：執行 `javac Hello.java`，編譯器檢查語法，沒問題就產出位元組碼檔 `Hello.class`。
3. **載入**：執行 `java Hello` 時，JVM 啟動並把 `Hello.class` 載入記憶體。
4. **執行**：JVM 逐條解讀位元組碼並執行；經常被執行的「熱點」程式碼還會被即時編譯成機器碼，越跑越快。

> **說明**　「即時編譯」（JIT，Just-In-Time compilation）是 Java 高效能的祕密武器。JVM 一邊執行一邊觀察：哪段程式被反覆執行上千次，就把那段位元組碼翻譯成最佳化過的機器碼。這就是為什麼 Java 程式「暖機」後速度可以逼近 C 語言。第一次閱讀看不懂沒關係，先有個印象即可。

1.3 安裝 JDK 17

接下來把工具裝起來。我們推薦初學者到 Adoptium 官網（`https://adoptium.net`）下載免費的 Eclipse Temurin JDK 17 安裝檔，它是目前業界最常用的 OpenJDK 發行版之一，macOS 與 Windows 都有圖形化安裝程式，一路按「下一步」即可完成。

1.3.1 macOS 安裝

方式一（推薦）：到 Adoptium 官網下載 `.pkg` 安裝檔，雙擊安裝。

方式二：如果你已經會用 Homebrew 套件管理器，也可以用指令安裝：

▶ 執行結果

```
$ brew install openjdk@17
==> Fetching openjdk@17
==> Pouring openjdk@17--17.0.12.arm64_sequoia.bottle.tar.gz
... (安裝訊息略)
```

用 Homebrew 安裝後，還需要依照畫面提示建立一個系統連結，macOS 內建的 `java` 指令才找得到它：

▶ 執行結果

```
$ sudo ln -sfn /opt/homebrew/opt/openjdk@17/libexec/openjdk.jdk \
    /Library/Java/JavaVirtualMachines/openjdk-17.jdk
```

1.3.2 Windows 安裝

到 Adoptium 官網下載 `.msi` 安裝檔，執行後在安裝選項中把「Add to PATH」（加入環境變數）與「Set JAVA_HOME variable」勾選起來，其餘保持預設值即可。安裝完成後，開啟「命令提示字元」或 PowerShell 進行下一步的驗證。

1.3.3 驗證安裝

無論用哪種方式安裝，最後都請打開終端機（macOS 的 Terminal 或 Windows 的命令提示字元），輸入以下兩個指令驗證：

▶ 執行結果

```
$ java -version
openjdk version "17.0.12" 2024-07-16
OpenJDK Runtime Environment Temurin-17.0.12+7 (build 17.0.12+7)
OpenJDK 64-Bit Server VM Temurin-17.0.12+7 (build 17.0.12+7, mixed mode)
```

```
$ javac -version
javac 17.0.12
```

> **提示**　你看到的版本號碼末段（例如 17.0.12 的 12）和本書不同完全沒關係，只要主要版本是 17 或更新即可。如果出現「找不到指令」之類的錯誤，最常見的原因是安裝時沒有設定環境變數——重新執行安裝程式並勾選「Add to PATH」通常就能解決。

1.4　第一個程式：HelloWorld

環境就緒，正式開工！請建立一個練習用的資料夾（例如 `java-practice`），用任何文字編輯器（記事本、TextEdit、VS Code 都行）建立一個名為 `HelloWorld.java` 的檔案，輸入以下內容。請務必親手逐字輸入，不要複製貼上——打字的過程就是在建立肌肉記憶。

範例：HelloWorld.java

```java
/**
 * 我的第一個 Java 程式。
 * 這個程式只做一件事：在主控台印出一行問候訊息。
 */
public class HelloWorld {

    /**
     * main 方法：整個程式的進入點。
     * 執行程式時，JVM 會從這裡開始，由上往下逐行執行。
     */
    public static void main(String[] args) {
        // System.out.println 會把括號裡的文字印到主控台，
        // 印完後自動換行
        System.out.println("Hello, World!");
    }
}
```

存檔後回到終端機，切換到檔案所在的資料夾，先編譯、再執行：

▶ 執行結果

```
$ javac HelloWorld.java
$ ls
HelloWorld.class  HelloWorld.java
$ java HelloWorld
Hello, World!
```

注意三個細節：第一，`javac` 編譯成功時「不會顯示任何訊息」，沉默就是好消息；第二，編譯後資料夾多了一個 `HelloWorld.class`，這就是位元組碼檔案；第三，執行時用的指令是 `java HelloWorld`，給的是「類別名稱」，不是檔名，所以不要加上 `.class` 或 `.java` 副檔名。

> **提示**　Java 11 之後，簡單的單一檔案程式可以跳過 `javac`，直接用 `java HelloWorld.java` 一步執行，很適合快速做小實驗。不過初學階段請至少完整走過幾次「先編譯、再執行」的流程，理解兩個階段各自在做什麼。

1.4.1 逐行解析：每個字都有它的理由

這短短幾行程式，其實濃縮了 Java 的許多核心概念。我們一個一個拆開看：

- `public class HelloWorld`：定義一個「公開的類別」，名字叫 HelloWorld。類別是 Java 程式的基本單位，所有程式碼都必須住在某個類別裡面。`public` 表示這個類別對外公開，任何人都能使用它。類別的詳細意義會在第 8 章完整說明，目前你可以把它想成「程式的容器」。
- `public static void main(String[] args)`：這是 **main 方法**，整個程式的進入點——JVM 規定：想被執行的程式，必須提供一個長得「一模一樣」的 main 方法。拆開來看：`public` 表示公開（JVM 在外面才呼叫得到）；`static` 表示這是靜態方法（JVM 不必先建立物件就能呼叫，原因見第 8 章）；`void` 表示這個方法不回傳任何值；`String[] args` 是一個字串陣列參數，用來接收執行時附加的指令列引數。
- `System.out.println("Hello, World!");`：真正做事的一行。`System` 是 Java 內建的類別，`out` 是它裡面代表「標準輸出（主控台）」的物件，`println` 則是「印出並換行」的方法。括號中用雙引號包住的 `"Hello, World!"` 是一個字串，也就是要印出的內容。
- **大括號**：`{` 與 `}` 標出「範圍」——類別的內容是一對大括號，方法的內容又是一對。它們必須成對出現，巢狀包裹，就像俄羅斯娃娃。
- **分號**：每個「敘述」（statement）結尾都要加分號 `;`，它是敘述的句點。忘記分號是初學者最常見的編譯錯誤，沒有之一。

> **注意**　兩條鐵律，現在就刻在腦海裡：第一，**檔名必須與 public 類別同名**，大小寫也要完全一致——類別叫 `HelloWorld`，檔案就必須叫 `HelloWorld.java`，存成 `helloworld.java` 就會編譯失敗，錯誤訊息會是「class HelloWorld is public, should be declared in a file named HelloWorld.java」。第二，**Java 嚴格區分大小寫**：`System` 與 `system`、`main` 與 `Main` 都是完全不同的東西。

1.4.2 再練一次：println 與 print

輸出是初期最重要的觀察工具，多練一個範例。這次同時認識 `println` 的兄弟 `print`——它印完「不會」換行：

範例：SelfIntroduction.java

```java
/**
 * 自我介紹程式：練習 println 與 print 的差別。
 */
public class SelfIntroduction {
    public static void main(String[] args) {
        // println：印出文字後「會」換行（ln 是 line 的縮寫）
        System.out.println("大家好，我是小明。");
        System.out.println("我今年 18 歲，正在學習 Java。");

        // print：印出文字後「不會」換行，
        // 下一次的輸出會緊接在同一行後面
        System.out.print("Java ");
        System.out.print("好好玩！");

        // 單獨呼叫 println() 不給內容，效果是「只換行」
        System.out.println();
        System.out.println("學習正式開始！");
    }
}
```

▶ 執行結果

```
大家好，我是小明。
我今年 18 歲，正在學習 Java。
Java 好好玩！
學習正式開始！
```

對照程式與輸出：兩次 `print` 的內容「Java 」與「好好玩！」被接在同一行，這正是它與 `println` 的差別。

1.5 註解：寫給人看的說明

程式碼是寫給電腦執行的，但更是寫給「人」讀的——包括三個月後已經忘光細節的你自己。註解（comment）是完全被編譯器忽略的文字，專門用來向讀者說明意圖。Java 提供三種註解：

範例：CommentDemo.java

```java
/**
 * 文件註解（Javadoc）：以斜線加兩顆星號開頭。
 * 慣例上放在類別與方法的正上方，說明「這是什麼、怎麼用」。
 * 專用工具可以抽取這種註解，自動產生 API 說明文件。
 */
public class CommentDemo {
    public static void main(String[] args) {
        // 單行註解：從雙斜線開始到行尾為止，都是給人看的

        /* 多行註解：從斜線星號開始，到星號斜線結束，
           中間可以橫跨很多行，適合較長的說明文字 */

        int price = 100;    // 註解也可以跟在程式碼後面
        System.out.println("價格：" + price);
    }
}
```

▶ 執行結果

```
價格：100
```

> **提示**　好註解的三個原則：(1) 解釋「為什麼」而不是「做什麼」——`price = price * 2;` 後面寫「乘以二」是廢話，寫「週年慶期間全面兩倍計價」才有價值；(2) 程式改了，註解要跟著改，過期的註解比沒有註解更害人；(3) 不要用註解「封存」大段廢棄程式碼，需要歷史紀錄請交給版本控制工具（如 Git）。本教材的範例會刻意寫非常多註解，因為它們同時肩負「教學」的任務。

1.6 看懂編譯錯誤訊息

初學階段有一半的時間在跟編譯錯誤打交道，所以「讀懂錯誤訊息」是最值得提早投資的技能。編譯器不是在罵你，它是在「精準地告訴你哪裡需要修改」。我們故意寫兩個錯誤程式，看看編譯器怎麼說。

1.6.1 案例一：忘記分號

錯誤示範：MissingSemicolon.java

```java
/**
```

```
 * 錯誤示範：第 7 行的結尾故意漏掉分號。
 */
public class MissingSemicolon {
    public static void main(String[] args) {
        // 下面這行結尾少了分號，編譯器會停下來抱怨
        System.out.println("Hello")
    }
}
```

▶ 執行結果

```
$ javac MissingSemicolon.java
MissingSemicolon.java:7: error: ';' expected
        System.out.println("Hello")
                                   ^
1 error
```

錯誤訊息的解讀方式：`MissingSemicolon.java:7` 指出「哪個檔案的第幾行」；`error:` 後面是問題描述——「預期這裡該有個分號」；下一行把出錯的原始碼印出來，並用插入符號 `^` 指出確切位置。補上分號重新編譯，問題解決。

1.6.2 案例二：大小寫寫錯

錯誤示範：WrongCase.java

```
/**
 * 錯誤示範：第 6 行把 System 誤寫成小寫的 system。
 */
public class WrongCase {
    public static void main(String[] args) {
        system.out.println("Hello");   // System 寫成了 system！
    }
}
```

▶ 執行結果

```
$ javac WrongCase.java
WrongCase.java:6: error: package system does not exist
        system.out.println("Hello");   // System 寫成了 system！
              ^
1 error
```

這次的訊息比較迂迴：編譯器說「找不到名為 system 的套件」。它猜不到你想寫的是 `System`，只能按照字面意義去找小寫的 `system`，找不到就報錯。遇到「does not exist」或「cannot find symbol」這類訊息，第一件事就是檢查拼字與大小寫。

閱讀編譯錯誤的要領整理如下：

- 先看「檔名：行號」，直接跳到出錯的位置附近。
- `error:` 之後的一句話是關鍵描述，搭配 `^` 的位置判讀。
- 一次可能列出很多錯誤，**永遠從第一個開始修**——後面的錯誤常常只是第一個錯誤引發的連鎖反應，修好第一個，其餘可能全消失。
- 修正後重新編譯，重複這個循環直到編譯器「沉默」為止。

1.7 選擇你的開發工具

專業開發者通常使用「整合開發環境」（IDE，Integrated Development Environment）。IDE 集編輯器、編譯器、除錯器於一身，還提供自動完成、即時錯誤提示等功能。主流選擇如下：

工具	特色	適合對象
IntelliJ IDEA Community	功能最完整、業界市占最高，社群版免費	想長期深耕 Java 的人
VS Code 加 Java 擴充套件	輕量、啟動快、外掛生態豐富	已熟悉 VS Code 的人
Eclipse	老牌免費 IDE，企業舊專案常見	需要接觸既有專案的人

> **提示**　建議的學習路線：**前幾章先用「文字編輯器加終端機」的原始方式**，親手執行 `javac` 與 `java`，把編譯流程內化成直覺；之後再改用 IntelliJ IDEA 或 VS Code 享受效率。太早依賴 IDE 的「一鍵執行」，容易讓你對底層流程一知半解，之後遇到環境問題會不知從何查起。

1.8 本章小結

- Java 於 1995 年誕生，是穩居全球前幾名的物件導向程式語言，廣泛應用於企業後端、Android、大數據與金融系統。
- 「一次編寫、到處執行」的關鍵：原始碼編譯成位元組碼，由各平台的 JVM 負責執行。
- JDK 包含 JRE，JRE 包含 JVM；開發者安裝 JDK 即可。
- 完整流程：`.java` 原始碼，經 `javac` 編譯成 `.class` 位元組碼，再由 `java` 啟動 JVM 執行。

- main 方法是程式的進入點，寫法固定：`public static void main(String[] args)`。
- 檔名必須與 public 類別同名；Java 嚴格區分大小寫；每個敘述以分號結尾。
- `println` 印出後換行，`print` 不換行。
- 三種註解：`//` 單行、`/* */` 多行、`/** */` 文件註解；註解要解釋「為什麼」。
- 編譯錯誤訊息會標出檔名、行號與問題描述，永遠從第一個錯誤開始修。

1.9 本章練習

動手練習

1. 觀念題：請說明 JVM、JRE、JDK 三者的關係，以及為什麼開發者需要安裝的是 JDK。（提示：包含關係——開發工具、執行環境、虛擬機器。）
2. 觀念題：為什麼 Java 可以「一次編寫、到處執行」？C 語言為什麼不行？（提示：想想位元組碼與 JVM 各自扮演的角色。）
3. 實作題：在你的電腦上安裝 JDK 17，並在終端機執行 `java -version` 與 `javac -version`，確認兩者的主要版本一致。（提示：若指令找不到，檢查安裝時是否勾選了加入 PATH。）
4. 實作題：寫一個 `AboutMe` 類別，用三行輸出你的名字、興趣與學習 Java 的目標；其中至少一行用 `print` 與 `println` 組合完成。（提示：注意檔名必須是 AboutMe.java。）
5. 除錯題：把 HelloWorld 程式中的 `println` 改成 `Println` 再編譯，觀察並記下錯誤訊息；再把 `public class HelloWorld` 改成 `public class helloworld` 試一次。（提示：一個會出現 cannot find symbol，另一個與檔名規則有關。）
6. 觀念題：`//` 與 `/* */` 兩種註解各適合什麼場合？文件註解 `/** */` 又特別在哪裡？（提示：行尾短說明、跨行長說明、可產生 API 文件。）

第 2 章

變數與資料型別

本章學習目標

學完本章你將能夠：

- 宣告變數、初始化變數，並依照命名慣例取出好名字。
- 說出 Java 八種基本型別的用途、大小與範圍。
- 分辨自動型別提升與強制轉型，並避開溢位與資料遺失的陷阱。
- 使用 final 宣告常數，並理解 var 型別推斷的使用時機。
- 初步分辨基本型別與參考型別的差異。
- 使用 Scanner 讀取使用者輸入，寫出會互動的程式。

上一章我們讓程式成功「說話」了，但那個程式只會印出寫死的文字。真正的程式必須能記住東西——玩家的血量、購物車的金額、使用者輸入的名字。這些被記住的資料就存放在「變數」（variable）裡。本章先學會建立與命名變數，接著認識 Java 為不同種類的資料準備的八種基本型別，最後讓程式第一次「聽」懂你從鍵盤輸入的內容。

Java 是「靜態型別」語言：每個變數在出生時就必須說清楚自己裝的是哪一種資料，而且終生不能改變。這個規定初期會讓你覺得囉嗦，但它讓編譯器能在程式執行「之前」就抓出一大堆錯誤——這正是大型系統偏愛 Java 的原因之一。

2.1 變數：帶標籤的盒子

把記憶體想像成一整面置物櫃。宣告一個變數，就是向 Java 申請一個大小合適的格子，並在上面貼一張標籤（變數名稱）。之後你隨時可以透過標籤找到這個格子，把值放進去或取出來。宣告的語法是「型別名稱」，例如 `int age` 表示「給我一個裝整數的格子，標籤寫 age」。

範例：VariableBasics.java

```
/**
 * 變數的宣告、初始化與重新指定值。
 */
public class VariableBasics {
    public static void main(String[] args) {
        // 宣告：申請一個放整數的盒子，標籤叫 age
```

```
        int age;

        // 指定值：把 18 放進盒子。第一次指定值稱為「初始化」
        age = 18;
        System.out.println("年齡：" + age);

        // 宣告的同時就初始化，是實務上最常見的寫法
        int height = 172;
        System.out.println("身高：" + height);

        // 變數的值可以被新值覆蓋——這正是「變」數的意義
        age = 19;   // 過了一年，長大一歲
        System.out.println("新年齡：" + age);

        // 同一行可宣告多個同型別變數（建議少用，可讀性較差）
        int x = 1, y = 2;
        System.out.println("x + y = " + (x + y));
    }
}
```

▶ 執行結果

```
年齡：18
身高：172
新年齡：19
x + y = 3
```

留意輸出裡的 `"年齡："` `+ age`：加號在這裡的作用是把文字與變數的值「串接」成一個字串，這是輸出時最常用的技巧，下一章會深入它的規則。

> **注意**　區域變數（宣告在方法裡的變數）在使用前必須先指定值。只宣告不初始化就拿來用，例如 `int a;` 之後直接 `System.out.println(a);`，會得到編譯錯誤「variable a might not have been initialized」。Java 寧可擋下你，也不讓一個內容不明的盒子流入程式——這是它嚴謹性格的展現。

2.2　命名規則與慣例

變數名稱有「規則」與「慣例」兩個層次。規則是編譯器強制的法律，違反就編譯失敗；慣例是業界共同的默契，違反雖能編譯，但會被同事皺眉。

規則只有幾條：名稱由字母、數字、底線與錢字符號組成；不能以數字開頭；不能使用 Java 的保留字（如 `class`、`int`、`public`）；大小寫視為不同名稱。

慣例則以「駝峰式命名」（camelCase）為核心：變數與方法名稱第一個單字全小寫，之後每個單字的字首大寫，例如 `totalPrice`、`studentCount`。類別名稱則每個單字字首都大寫（PascalCase），例如 `HelloWorld`、`BankAccount`。

名稱	合法嗎	說明
`userName`	合法	標準 camelCase，推薦寫法
`totalPrice2`	合法	數字可以出現在開頭以外的位置
`_temp`	合法但不建議	底線開頭慣例上留給特殊用途
`2ndPlace`	不合法	不能以數字開頭
`user-name`	不合法	連字號不是名稱的合法字元
`class`	不合法	保留字不能當名稱

提示　名稱是最便宜的文件。`int s = 90;` 三個月後沒人記得 s 是什麼；`int studentScore = 90;` 則一眼就懂。取名的原則：寧可長而清楚，不要短而神祕。變數命名用名詞（`orderCount`），布林變數用 is、has 開頭（`isFinished`、`hasTicket`），讀起來就像一句話。

2.3　八種基本型別

Java 內建八種「基本型別」（primitive type），依用途分為四類：整數（`byte`、`short`、`int`、`long`）、浮點數（`float`、`double`）、字元（`char`）與布林（`boolean`）。全貌如下表：

型別	位元數	範圍／說明	預設值	字面值範例
`byte`	8	−128 到 127	`0`	`(byte) 100`
`short`	16	−32768 到 32767	`0`	`(short) 30000`
`int`	32	約 ±21 億	`0`	`123`
`long`	64	約 $\pm 9.2 \times 10^{18}$	`0L`	`123L`
`float`	32	約 7 位有效數字	`0.0f`	`3.14f`
`double`	64	約 15 位有效數字	`0.0`	`3.14`
`char`	16	單一 Unicode 字元	`'\u0000'`	`'A'`
`boolean`	—	真或假	`false`	`true`

表中「預設值」是指欄位與陣列元素未指定值時自動獲得的值（第 5 章與第 8 章會再遇到）；區域變數沒有預設值，如前一節所述必須自行初始化。實務上的選擇很簡單：整數用 `int`，數字可能超過 21 億才用 `long`；小數一律用 `double`；`byte`、`short`、`float` 多半只在節省記憶體或對接特定格式時才登場。

2.3.1 整數型別與字面值

程式碼中直接寫出的值稱為「字面值」（literal），例如 `123`。整數字面值有幾種讓程式更好讀的寫法：

範例：IntegerLiterals.java

```java
/**
 * 整數字面值的各種寫法：底線分隔、long 後綴、二進位與十六進位。
 */
public class IntegerLiterals {
    public static void main(String[] args) {
        // 整數字面值預設是 int 型別
        int population = 23_000_000;    // 底線只是分隔，方便人眼閱讀
        System.out.println("台灣人口約：" + population);

        // 超過 int 範圍的字面值必須加 L 後綴，存入 long
        long worldPop = 8_000_000_000L;
        System.out.println("世界人口約：" + worldPop);

        // 二進位字面值：0b 開頭（b 是 binary 的縮寫）
        int flags = 0b1010;             // 等於十進位的 10
        System.out.println("0b1010 = " + flags);

        // 十六進位字面值：0x 開頭，常見於顏色與底層資料
        int color = 0xFF;               // 等於十進位的 255
        System.out.println("0xFF = " + color);
    }
}
```

▶ 執行結果

```
台灣人口約：23000000
世界人口約：8000000000
0b1010 = 10
0xFF = 255
```

注意兩件事：底線只是給人看的，印出來時不會出現；寫 `8_000_000_000` 而不加 `L` 會直接編譯錯誤，因為這個字面值本身已經超過 `int` 的極限。

> **說明**　為什麼 `int` 的範圍是 -2^{31} 到 $2^{31}-1$（大約正負 21 億）？因為它固定佔 32 個位元，其中 1 個位元表示正負號，剩下 31 個位元表示數值。你可以用 `Integer.MAX_VALUE`

> 與 `Integer.MIN_VALUE` 取得精確的邊界值：2147483647 與 −2147483648。

2.3.2 浮點型別

帶小數點的字面值預設是 `double`；想存進 `float` 必須加上 `f` 後綴。兩者的差別在精度：

範例：FloatingPointDemo.java

```
/**
 * double 與 float 的精度差異。
 */
public class FloatingPointDemo {
    public static void main(String[] args) {
        // double：雙精度，約可保留 15 位有效數字
        double pi = 3.141592653589793;
        System.out.println("pi = " + pi);

        // float：單精度，只能保留約 7 位有效數字，
        // 字面值必須加 f 後綴，否則編譯器當它是 double
        float f = 1.23456789f;
        System.out.println("f = " + f);   // 尾數被捨入了！

        // 科學記號寫法：1.5e-3 表示 1.5 乘以 10 的負 3 次方
        double tiny = 1.5e-3;
        System.out.println("tiny = " + tiny);
    }
}
```

▶ 執行結果

```
pi = 3.141592653589793
f = 1.2345679
tiny = 0.0015
```

觀察輸出：我們存入 `1.23456789`，印出來卻是 `1.2345679`——`float` 的精度只有約 7 位，塞不下的部分被捨入了。這不是臭蟲，是型別天生的限制。實務上小數一律先選 `double`；至於「連 double 都存不準」的情況（例如金額計算），第 3 章會示範並給出解法。

2.3.3 char 與 Unicode

`char` 存放「恰好一個」字元，字面值用單引號。Java 的字元採用 Unicode 編碼，所以中文字、日文字都是合法的 `char`：

範例：CharDemo.java

```java
/**
 * char 的本質：一個 0 到 65535 的整數，對應一個 Unicode 字元。
 */
public class CharDemo {
    public static void main(String[] args) {
        // char 字面值用單引號，只能放一個字元
        char grade = 'A';
        System.out.println("等第：" + grade);

        char word = '中';               // 中文字也是一個 char
        System.out.println("字元：" + word);

        // 把 char 轉成 int，就能看到它背後的 Unicode 編號
        System.out.println("它的編號：" + (int) word);

        // 也可以用 Unicode 跳脫序列來表示同一個字
        char same = '\u4E2D';           // 與 '中' 完全相同
        System.out.println("same = " + same);

        // 字元可以參與運算——因為它骨子裡是整數
        char next = (char) (grade + 1);
        System.out.println("A 的下一個字母：" + next);
    }
}
```

▶ 執行結果

```
等第：A
字元：中
它的編號：20013
same = 中
A 的下一個字母：B
```

> **注意**　單引號與雙引號在 Java 中意義完全不同：`'A'` 是字元（`char`），`"A"` 是只有一個字的字串（`String`）。寫 `char c = "A";` 會編譯錯誤。這與某些語言（如 Python）兩者通用的習慣不同，從其他語言轉來的讀者請特別注意。

2.3.4 boolean

`boolean` 只有 `true` 與 `false` 兩種值，專門回答「是非題」。它是下一章比較運算與第 4 章流程控制的靈魂：

範例：BooleanDemo.java

```
/**
 * boolean：程式世界的是非題。
 */
public class BooleanDemo {
    public static void main(String[] args) {
        boolean isAdult = true;
        boolean hasTicket = false;
        System.out.println("成年了嗎？" + isAdult);
        System.out.println("有票嗎？" + hasTicket);

        // boolean 最常見的來源：比較運算的結果
        int age = 16;
        boolean canVote = age >= 18;    // 16 >= 18 不成立
        System.out.println("可以投票嗎？" + canVote);
    }
}
```

▶ 執行結果

```
成年了嗎？true
有票嗎？false
可以投票嗎？false
```

2.4 型別轉換

不同型別之間的值經常需要互相流動：整數存進浮點數、大範圍塞進小範圍。Java 把轉換分成兩種：安全的「自動提升」與有風險的「強制轉型」。

2.4.1 自動提升：小杯倒大杯

把小範圍型別的值放進大範圍型別，就像把小杯的水倒進大杯，絕不會溢出，所以 Java 自動放行，不需任何額外語法。提升的方向是：

- `byte` 到 `short` 到 `int` 到 `long` 到 `float` 到 `double`（由小到大單向流動）。
- `char` 也可以提升為 `int` 以上的型別。

範例：WideningDemo.java

```java
/**
 * 自動型別提升：小範圍流向大範圍，安全且不需要任何語法。
 */
public class WideningDemo {
    public static void main(String[] args) {
        int i = 100;
        long l = i;      // int 自動提升為 long
        double d = l;    // long 自動提升為 double

        System.out.println("i = " + i);
        System.out.println("l = " + l);
        System.out.println("d = " + d);   // 變成小數形式的 100.0
    }
}
```

▶ 執行結果

```
i = 100
l = 100
d = 100.0
```

2.4.2 強制轉型：大杯倒小杯的風險

反過來，把大範圍的值塞進小範圍型別，可能裝不下。Java 不會自動放行，你必須寫出「轉型運算子」（型別）明確表態：「我知道有風險，後果自負」。風險有兩種：小數被截斷，以及數值溢位。

範例：NarrowingDemo.java

```java
/**
 * 窄化轉型的兩種風險：小數截斷與數值溢位。
 */
public class NarrowingDemo {
    public static void main(String[] args) {
        // 風險一：小數轉整數時，小數部分被「無條件捨去」
        double price = 3.99;
        int dollars = (int) price;      // 不是四捨五入！
        System.out.println("3.99 轉 int = " + dollars);

        // 風險二：超出目標型別的範圍時，數值會「繞圈」失真
        int big = 130;
```

```
        byte small = (byte) big;        // byte 只能到 127
        System.out.println("130 轉 byte = " + small);

        // 更隱蔽的溢位：int 運算結果超過 21 億，不會報錯，
        // 只會默默給你一個錯得離譜的值
        int a = 2_000_000_000;
        int sum = a + a;
        System.out.println("20 億 + 20 億 = " + sum);

        // 解法：先把其中一個運算元提升為 long 再相加
        long safeSum = (long) a + a;
        System.out.println("正確結果 = " + safeSum);
    }
}
```

▶ 執行結果

```
3.99 轉 int = 3
130 轉 byte = -126
20 億 + 20 億 = -294967296
正確結果 = 4000000000
```

130 為什麼變成 −126？因為 `byte` 只有 8 個位元，130 的二進位超出了它能表示的範圍，多出來的位元被硬生生砍掉，剩下的位元恰好是 −126 的表示法。你不需要現在就會算，但要記住現象：**窄化轉型與整數溢位都不會報錯，只會安靜地給錯值**。

> **注意**　「安靜的錯誤」比「大聲的錯誤」危險一百倍。歷史上不少重大軟體事故源自整數溢位——系統沒有當機，只是算出了錯誤的金額或座標。防禦方式：數值可能很大時提早用 `long`；轉型前先確認值域；金額計算避免用浮點數（第 3 章詳述）。

2.5　final 常數

有些值一旦設定就不該再變：稅率、圓周率、遊戲人數上限。在宣告前加上 `final`，這個變數就只能被指定一次值，之後任何修改都會被編譯器擋下。慣例上常數名稱使用全大寫，單字之間用底線分隔：

範例：FinalDemo.java

```
/**
 * final 常數：只能指定一次值，天塌下來都不能改。
 */
```

```java
public class FinalDemo {
    public static void main(String[] args) {
        // 常數命名慣例：全大寫、單字用底線分隔
        final double TAX_RATE = 0.05;
        final int MAX_PLAYERS = 4;

        double price = 1000;
        double total = price * (1 + TAX_RATE);
        System.out.println("含稅價：" + total);
        System.out.println("人數上限：" + MAX_PLAYERS);

        // TAX_RATE = 0.08;   // 取消註解會編譯錯誤：
        // cannot assign a value to final variable TAX_RATE
    }
}
```

▶ 執行結果

```
含稅價：1050.0
人數上限：4
```

> **提示**　程式裡憑空出現的數字稱為「魔術數字」（magic number），例如 `if (count > 4)` 的 4 是什麼意思？沒人知道。把它命名成 `MAX_PLAYERS` 之後，程式自己會說話，而且將來規則改成 6 人時只需要改一個地方。看到重複出現、有業務意義的數字，就該考慮宣告成 final 常數。

2.6　var：讓編譯器幫你寫型別

Java 10 引入了 `var`：宣告區域變數時，讓編譯器從右邊的初始值「推斷」型別，你就不必重複寫一次。務必理解：`var` 不是「動態型別」，型別在編譯的當下就已經確定，之後同樣不能改變。

範例：VarDemo.java

```java
/**
 * var 區域變數型別推斷：型別由右邊的初始值決定。
 */
public class VarDemo {
    public static void main(String[] args) {
        var count = 10;          // 推斷為 int
        var name = "Java";       // 推斷為 String
```

```
        var ratio = 0.5;        // 推斷為 double

        System.out.println(count + ", " + name + ", " + ratio);

        // 型別推斷完就固定了：var 不是「什麼都能裝」的魔法盒
        // count = "十";   // 編譯錯誤：不能把字串塞進 int 變數
    }
}
```

▶ 執行結果

```
10, Java, 0.5
```

var 的限制與使用建議：

- 只能用於區域變數，欄位、方法參數、回傳型別都不能用。
- 宣告時必須同時給初始值，否則編譯器無從推斷。
- 初始值不能是 `null`——推斷不出型別。
- 使用時機：右邊已經把型別寫得清清楚楚時（例如 `var sc = new Scanner(System.in);` ↪ ），用 var 可減少重複；右邊型別不明顯時，老實寫出型別對讀者更友善。

2.7 基本型別與參考型別初探

Java 的型別世界分成兩大陣營。基本型別就是本章的八種，變數的盒子裡直接放「值本身」；除此之外的一切型別——字串 `String`、陣列、以及你未來自己定義的類別——都是參考型別（reference type），變數的盒子裡放的不是資料本身，而是資料所在位置的「參考」，就像盒子裡放的不是電視機，而是電視的遙控器。

現在先記住兩個立即有感的差別：

- 基本型別的變數互相指定時，複製的是「值」；參考型別複製的是「遙控器」，兩個變數會遙控同一台電視。這件事在第 5 章的陣列與第 6 章的方法傳參會產生重要影響。
- `String` 開頭是大寫、不在八種基本型別之列——它是參考型別，所以字串比較不能用 `==`，細節留到第 7 章。

> **說明**　記憶體的角度：基本型別的值輕巧固定，直接存放在變數所在的位置；參考型別的實際資料住在一塊叫「堆積」（heap）的記憶體區域，變數只保存它的地址。這個模型現在只需要有個畫面，等學到陣列與物件時，我們會反覆回來使用它。

2.8 用 Scanner 讀取使用者輸入

到目前為止，程式的資料都是我們寫死的。Scanner 是 Java 內建的輸入工具，能讀取使用者從鍵盤輸入的內容，讓程式真正動起來。使用它需要三步：第一，在檔案開頭 `import java.util.Scanner;` 引入；第二，用 `new Scanner(System.in)` 建立掃描器；第三，依照想讀的型別呼叫對應方法——`nextInt()` 讀整數、`nextDouble()` 讀小數、`nextLine()` 讀一整行文字。

範例：ScannerDemo.java

```java
import java.util.Scanner;

/**
 * 讀取使用者輸入：詢問姓名、年齡與身高，再組合成回覆。
 */
public class ScannerDemo {
    public static void main(String[] args) {
        // System.in 代表鍵盤輸入，把它交給 Scanner 處理
        Scanner scanner = new Scanner(System.in);

        System.out.print("請輸入你的名字：");
        String name = scanner.nextLine();       // 讀一整行文字

        System.out.print("請輸入你的年齡：");
        int age = scanner.nextInt();            // 讀一個整數

        System.out.print("請輸入你的身高（公尺）：");
        double height = scanner.nextDouble();   // 讀一個小數

        System.out.println("你好，" + name + "！");
        System.out.println("明年你就 " + (age + 1) + " 歲了。");
        System.out.println("你的身高是 " + height + " 公尺。");

        scanner.close();    // 用完關閉，釋放系統資源
    }
}
```

▶ 執行結果

```
請輸入你的名字：小明
請輸入你的年齡：18
請輸入你的身高（公尺）：1.72
你好，小明！
```

```
明年你就 19 歲了。
你的身高是 1.72 公尺。
```

範例中的提示文字用 `print` 而不是 `println`，讓游標停在同一行等待輸入，這是比較好的使用體驗。另外注意 `(age + 1)` 的括號：先做數學加法，再參與字串串接（原因見第 3 章）。

> **注意**　經典陷阱：`nextInt()` 只讀走「數字」，使用者按下的 Enter（換行字元）還留在輸入緩衝區裡。如果下一步呼叫 `nextLine()`，它會立刻讀到那個殘留的換行，回傳空字串，看起來就像「程式跳過了輸入」。解法：在 `nextInt()` 之後、真正要讀文字之前，先補一次 `scanner.nextLine()` 把殘留的換行吃掉。

2.9　本章小結

- 變數是帶標籤的記憶體盒子；宣告語法為「型別名稱」，區域變數使用前必須初始化。
- 命名規則：不能數字開頭、不能用保留字、大小寫敏感；慣例上變數用 camelCase，類別用 PascalCase，常數全大寫。
- 八種基本型別：整數選 `int`（超過 21 億用 `long`），小數選 `double`，字元 `char` 用單引號，是非題用 `boolean`。
- 整數字面值可用底線分隔、`0b` 二進位與 `0x` 十六進位；long 字面值要加 `L`，float 要加 `f`。
- 小範圍到大範圍自動提升；反向必須強制轉型，可能發生小數截斷與溢位，而且不會報錯。
- `final` 讓變數變成只能指定一次的常數，消滅魔術數字。
- `var` 只能用在有初始值的區域變數，型別在編譯期就固定。
- 八種基本型別存「值」，其他型別（含 `String`）存「參考」。
- `Scanner` 搭配 `nextInt`、`nextDouble`、`nextLine` 讀取輸入；小心 nextInt 之後殘留的換行字元。

2.10　本章練習

動手練習

1. 觀念題：下列名稱哪些是合法的變數名稱？`myScore`、`3dModel`、`user_name`、`int`、`priceInNTD`。不合法的請說明原因。（提示：數字開頭與保留字是兩大殺手。）
2. 觀念題：`int` 的最大值是 2147483647，請問再加 1 會發生什麼事？程式會當機嗎？（提示：溢位會「繞圈」回到最小值，而且不會報錯。）

3. 實作題：宣告八種基本型別的變數各一個，指定合理的值並全部印出；long 與 float 記得加後綴。（提示：字面值後綴 L 與 f 忘記加會編譯錯誤。）
4. 實作題：寫一個攝氏轉華氏的程式：用 Scanner 讀入攝氏溫度（整數），印出華氏溫度。公式為攝氏乘以 9 除以 5 再加 32。測試輸入 37 應得到 98.6。（提示：小心整數除法！9 / 5 會變成 1，把其中一個數字寫成 9.0 試試。）
5. 實作題：用 Scanner 連續讀入你的姓名（一行文字）與幸運數字（整數），先讀數字再讀姓名，觀察是否踩到換行殘留的陷阱，並修正它。（提示：nextInt 之後補一次 nextLine。）
6. 觀念題：`var text = "hello";` 之後，能不能執行 `text = 42;`？為什麼？（提示：var 的型別在編譯期就固定了。）

第 3 章

運算子與運算式

本章學習目標

學完本章你將能夠：

- 使用算術運算子計算，並解釋整數除法與取餘數的行為。
- 分辨 i++ 與 ++i 的差異，正確使用遞增遞減與複合指定運算子。
- 使用比較與邏輯運算子組合條件，並解釋短路求值。
- 避開字串串接與加號混用的經典陷阱。
- 說明浮點數為什麼會有精度誤差，以及正確的比較方式。
- 綜合運用本章工具，寫出會互動的 BMI 計算機。

上一章我們學會把資料放進變數，但只會「存放」的程式什麼也做不了——資料必須被計算、比較、組合，才會產生價值。讓資料動起來的工具就是「運算子」（operator），而運算子把變數與字面值組合起來的式子稱為「運算式」（expression），每個運算式最終都會算出一個值。

本章的內容像是一盒工具：每種運算子單獨看都很簡單，但它們彼此搭配時藏著幾個非常經典的陷阱——整數除法的截斷、字串串接的順序、浮點數的精度誤差。這些陷阱每年坑掉無數初學者，我們會逐一拆解，並在章末用一個完整的 BMI 計算機把所有工具串起來。

3.1 算術運算子

Java 提供五個基本算術運算子：

運算子	意義	範例	結果
`+`	加法	`7 + 2`	`9`
`-`	減法	`7 - 2`	`5`
`*`	乘法	`7 * 2`	`14`
`/`	除法	`7 / 2`	`3`（不是 3.5 !）
`%`	取餘數	`7 % 2`	`1`

其中除法是第一個重點：**整數除以整數，結果仍是整數**，小數部分被直接丟棄（向零截斷），不是四捨五入。想得到小數結果，至少要讓其中一邊是浮點數。

範例：ArithmeticDemo.java

```java
/**
 * 算術運算子總覽：特別注意整數除法與取餘數的行為。
 */
public class ArithmeticDemo {
    public static void main(String[] args) {
        int a = 7;
        int b = 2;

        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));

        // 整數除法：小數部分直接捨去
        System.out.println("a / b = " + (a / b));

        // 想要小數結果，至少一邊要是浮點數
        System.out.println("a / 2.0 = " + (a / 2.0));

        // 取餘數：7 除以 2，商 3「餘 1」
        System.out.println("a % b = " + (a % b));

        // 負數的除法與餘數：商向零截斷，餘數跟著被除數的正負號
        System.out.println("-7 / 2 = " + (-7 / 2));
        System.out.println("-7 % 2 = " + (-7 % 2));
    }
}
```

▶ 執行結果

```
a + b = 9
a - b = 5
a * b = 14
a / b = 3
a / 2.0 = 3.5
a % b = 1
-7 / 2 = -3
-7 % 2 = -1
```

取餘數 % 是低調的實用高手，常見用途：

- 判斷奇偶：`n % 2 == 0` 成立就是偶數。
- 判斷整除：`n % 3 == 0` 表示 n 是 3 的倍數。

- 取出位數：`n % 10` 是 n 的個位數（第 4 章會用它反轉數字）。
- 循環計數：不斷加一的索引對長度取餘數，就能繞圈圈（例如星期幾、輪播圖）。

3.1.1 除以零會怎樣

數學老師說不能除以零，Java 的態度則分成兩種。浮點數除以零不會讓程式當掉，而是回傳特殊值；整數除以零則會直接拋出例外、中斷程式：

錯誤示範：DivideByZero.java

```
/**
 * 整數與浮點數除以零的不同下場。
 */
public class DivideByZero {
    public static void main(String[] args) {
        // 浮點數除以零：得到特殊值，程式繼續走
        System.out.println(1.0 / 0);    // 正無限大
        System.out.println(-1.0 / 0);   // 負無限大
        System.out.println(0.0 / 0);    // NaN：Not a Number

        // 整數除以零：拋出例外，程式立刻中斷！
        System.out.println(1 / 0);
    }
}
```

▶ 執行結果

```
Infinity
-Infinity
NaN
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at DivideByZero.main(DivideByZero.java:12)
```

這是你第一次看到「例外」（exception）：執行期發生的錯誤事件，未處理時會終止程式並印出發生位置。例外處理是中級篇的重頭戲，目前先學會看懂這種訊息即可——它告訴你錯誤種類（ArithmeticException）、原因（除以零）與行號（第 12 行）。

3.2 遞增與遞減

「把變數加一」是程式裡出現頻率最高的動作，因此有專屬的縮寫：`i++`（後置）與 `++i`（前置），同理有 `i--` 與 `--i`。兩者都讓變數加一，差別在「運算式本身的值」：後置先交出舊值再加一，前置先加一再交出新值。

範例：IncrementDemo.java

```java
/**
 * 後置 i++ 與前置 ++i 的差異。
 */
public class IncrementDemo {
    public static void main(String[] args) {
        int a = 5;
        int b = a++;    // 後置：先把舊值 5 交給 b，a 才變 6
        System.out.println("a = " + a + ", b = " + b);

        int c = 5;
        int d = ++c;    // 前置：c 先變 6，再把新值 6 交給 d
        System.out.println("c = " + c + ", d = " + d);

        // 單獨成一行使用時，前置與後置效果完全相同
        int count = 0;
        count++;
        ++count;
        System.out.println("count = " + count);
    }
}
```

▶ 執行結果

```
a = 6, b = 5
c = 6, d = 6
count = 2
```

> **提示**　最佳實務：讓 `++` 與 `--` 單獨成一行，不要塞進複雜運算式裡。像 `int x = i++ + ++i;` 這種程式雖然合法，但沒有人能一眼看懂，程式碼審查時一定會被退件。清楚永遠比炫技重要。

3.3 指定與複合指定運算子

`=` 是「指定」運算子：把右邊的值放進左邊的變數，不是數學的「等於」。搭配算術運算子有一系列縮寫：`a += 5` 等同 `a = a + 5`，其餘 `-=`、`*=`、`/=`、`%=` 依此類推。

範例：CompoundAssignDemo.java

```java
/**
```

```
 * 複合指定運算子：對變數「就地更新」的縮寫。
 */
public class CompoundAssignDemo {
    public static void main(String[] args) {
        int score = 100;

        score += 20;    // score = score + 20
        System.out.println("加 20 後：" + score);

        score -= 50;    // score = score - 50
        System.out.println("減 50 後：" + score);

        score *= 2;     // score = score * 2
        System.out.println("乘 2 後：" + score);

        score /= 3;     // 整數除法：140 / 3 = 46
        System.out.println("除 3 後：" + score);

        score %= 7;     // 46 % 7 = 4
        System.out.println("對 7 取餘：" + score);
    }
}
```

▶ 執行結果

```
加 20 後：120
減 50 後：70
乘 2 後：140
除 3 後：46
對 7 取餘：4
```

> **說明**　複合指定運算子還偷偷附贈了自動轉型：`byte b = 10; b += 5;` 可以編譯，但拆開寫成 `b = b + 5;` 反而錯誤——因為 `b + 5` 的運算結果被提升為 `int`，塞回 byte 需要強制轉型。複合版本相當於幫你補了 `b = (byte) (b + 5);`。知道這個細節，未來看到相關的編譯錯誤就不會一頭霧水。

3.4 比較運算子

比較運算子把兩個值做比較，產出 `boolean` 結果，是流程控制的基石：`==`（等於）、`!=`（不等於）、`>`、`<`、`>=`、`<=`。

範例：ComparisonDemo.java

```java
/**
 * 比較運算子：結果一律是 boolean。
 */
public class ComparisonDemo {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;

        System.out.println("x == y：" + (x == y));
        System.out.println("x != y：" + (x != y));
        System.out.println("x < y：" + (x < y));
        System.out.println("x >= 10：" + (x >= 10));
    }
}
```

▶ 執行結果

```
x == y：false
x != y：true
x < y：true
x >= 10：true
```

> **注意**　兩個必踩地雷提前排除：第一，= 是指定、== 才是比較，寫錯位置輕則編譯錯誤，重則邏輯默默出錯。第二，== 只適合比較基本型別；比較字串內容必須用 `equals()` 方法，原因與示範見第 7 章——先把這條規則背起來：**字串比較不用兩個等號**。

3.5　邏輯運算子與短路求值

邏輯運算子把多個 boolean 條件組合成複合條件：`&&`（且）、`||`（或）、`!`（非）。真值表如下：

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

重點特性是「短路求值」（short-circuit evaluation）：`&&` 的左邊若是 `false`，整體必為 false，右邊根本不會執行；`||` 的左邊若是 `true`，右邊同樣被跳過。下面的程式用 `++x` 當「探測器」

證明這件事——如果右邊有執行，x 就會留下痕跡：

範例：ShortCircuitDemo.java

```java
/**
 * 短路求值的證明：用 ++x 偵測右邊運算式到底有沒有被執行。
 */
public class ShortCircuitDemo {
    public static void main(String[] args) {
        int x = 0;

        // && 左邊是 false：右邊被跳過，x 不會加一
        boolean r1 = (x > 0) && (++x > 0);
        System.out.println("r1 = " + r1 + ", x = " + x);

        // || 左邊是 true：右邊同樣被跳過
        boolean r2 = (x == 0) || (++x > 0);
        System.out.println("r2 = " + r2 + ", x = " + x);

        // 左邊無法決定結果時，右邊才會執行：這次 x 真的加一了
        boolean r3 = (x == 0) && (++x > 0);
        System.out.println("r3 = " + r3 + ", x = " + x);
    }
}
```

▶ 執行結果

```
r1 = false, x = 0
r2 = true, x = 0
r3 = true, x = 1
```

短路不只是效能優化，更是實用的防禦技巧。經典寫法：`s != null && s.length() > 0`——若 s 是 null，左邊直接判定 false，右邊的 `s.length()` 不會執行，也就不會因為對 null 呼叫方法而爆炸。條件的順序因此有了意義：把「保鏢」放左邊。

3.6　位元運算子簡介

位元運算子直接對整數的二進位位元動手，常見於底層開發、旗標處理與效能敏感的場合。初學階段認識即可：`&`（位元且）、`|`（位元或）、`^`（互斥或）、`~`（取反）、`<<`（左移）、`>>`（右移）。

範例：BitwiseDemo.java

```java
/**
 * 位元運算子速覽：以 12（1100）與 10（1010）為例。
 */
public class BitwiseDemo {
    public static void main(String[] args) {
        int a = 12;     // 二進位 1100
        int b = 10;     // 二進位 1010

        System.out.println("a & b = " + (a & b));   // 1000
        System.out.println("a | b = " + (a | b));   // 1110
        System.out.println("a ^ b = " + (a ^ b));   // 0110
        System.out.println("~a = " + (~a));         // 位元全反

        // 左移一位等於乘 2，右移一位等於除 2（正整數時）
        System.out.println("3 << 2 = " + (3 << 2));
        System.out.println("16 >> 2 = " + (16 >> 2));
    }
}
```

▶ 執行結果

```
a & b = 8
a | b = 14
a ^ b = 6
~a = -13
3 << 2 = 12
16 >> 2 = 4
```

注意區分：&& 與 || 是「邏輯」運算子，操作 boolean 且會短路；單一個 & 與 | 用在整數時是「位元」運算子。日常業務邏輯幾乎只用得到邏輯運算子。

3.7　三元運算子

三元運算子 條件 ? 值A : 值B 是迷你版的二選一：條件成立取值 A，否則取值 B。它是「運算式」，會算出一個值，適合簡短的判斷：

範例：TernaryDemo.java

```java
/**
 * 三元運算子：一行完成「二選一」。
 */
public class TernaryDemo {
```

```
    public static void main(String[] args) {
        int score = 75;

        // 條件成立取問號後的值，否則取冒號後的值
        String result = (score >= 60) ? "及格" : "不及格";
        System.out.println(result);

        int a = 10;
        int b = 20;
        int max = (a > b) ? a : b;      // 取較大值
        System.out.println("較大值：" + max);

        String parity = (a % 2 == 0) ? "偶數" : "奇數";
        System.out.println(a + " 是" + parity);
    }
}
```

▶ 執行結果

```
及格
較大值：20
10 是偶數
```

三元運算子可以巢狀，但兩層以上可讀性就會急速惡化，此時應改用下一章的 `if-else` 敘述。原則：三元運算子只留給「一眼能看懂」的場合。

3.8 字串串接的加號陷阱

`+` 有雙重身分：兩邊都是數字時做加法；**任一邊是字串時，做字串串接**，並把另一邊轉成文字。配上「由左到右」的計算順序，就產生了每年騙倒無數初學者的經典陷阱：

範例：ConcatTrap.java

```
/**
 * 加號的雙重身分：加法與字串串接，順序決定一切。
 */
public class ConcatTrap {
    public static void main(String[] args) {
        // 由左到右："1" + 2 先變成字串 "12"，再接上 3
        System.out.println("1" + 2 + 3);

        // 由左到右：1 + 2 是數字加法得 3，再與 "3" 串接
```

```
        System.out.println(1 + 2 + "3");

        // 陷阱重現：本想印 95 分，卻印出 905！
        System.out.println("總分：" + 90 + 5);

        // 解法：用括號先完成數學加法
        System.out.println("總分：" + (90 + 5));
    }
}
```

▶ 執行結果

```
123
33
總分：905
總分：95
```

> **注意**　規則整理：+ 一旦碰到字串，之後整條式子都會被「串接」感染。`"總分：" + 90 + 5` 的計算過程是：先算 `"總分：" + 90` 得到字串「總分：90」，再接上 5 變成「總分：905」。凡是要在輸出裡做數學運算，一律加括號，例如 `"總分：" + (90 + 5)`。

3.9　運算子優先順序

一條式子有多個運算子時，誰先誰後？Java 的優先順序與數學直覺大致相同（先乘除後加減），完整順位如下表（同列由左到右結合，指定運算子除外）：

優先度	運算子	說明
1（最高）	`()`	括號
2	`++ -- ! ~`	一元運算子
3	`* / %`	乘、除、取餘
4	`+ -`	加、減
5	`<< >>`	位元移位
6	`< <= > >=`	大小比較
7	`== !=`	相等比較
8	`& ^ |`	位元邏輯
9	`&&`	邏輯且
10	`||`	邏輯或
11	`? :`	三元運算子
12（最低）	`= += -= 等`	指定

> **提示**　你不需要背這張表。業界的共識是：**不確定就加括號**。括號不但免除記憶負擔，更把你的意圖白紙黑字寫給讀者看。`(a && b) || (c && d)` 遠比省掉括號的版本清楚，效能上也毫無差別。

3.10　浮點數的精度問題

準備好見證一件顛覆直覺的事了嗎？在 Java 裡，`0.1 + 0.2` 不等於 `0.3`。這不是 Java 的臭蟲，而是所有使用 IEEE 754 標準的語言（包括 Python、JavaScript）的共同宿命：十進位的 0.1 換算成二進位是無限循環小數，就像十進位無法精確寫出三分之一，儲存時必然有微小誤差。

範例：PrecisionDemo.java

```
/**
 * 浮點數精度誤差：0.1 + 0.2 為什麼不等於 0.3。
 */
public class PrecisionDemo {
    public static void main(String[] args) {
        double sum = 0.1 + 0.2;
        System.out.println("0.1 + 0.2 = " + sum);
        System.out.println("等於 0.3 嗎？" + (sum == 0.3));

        // 正確的比較方式：兩數的差距小於容忍值，就視為相等
        double epsilon = 1e-9;      // 容忍十億分之一的誤差
        boolean closeEnough = Math.abs(sum - 0.3) < epsilon;
        System.out.println("夠接近 0.3 嗎？" + closeEnough);
    }
}
```

▶ 執行結果

```
0.1 + 0.2 = 0.30000000000000004
等於 0.3 嗎？false
夠接近 0.3 嗎？true
```

因此鐵律有二：**永遠不要用 `==` 直接比較浮點數；金額運算不要用 `double`**。需要十進位精確計算時，Java 提供 `BigDecimal` 類別：

範例：BigDecimalDemo.java

```
import java.math.BigDecimal;

```

```java
/**
 * BigDecimal：十進位的精確計算，金融系統的標準配備。
 */
public class BigDecimalDemo {
    public static void main(String[] args) {
        // 務必用「字串」建構！寫 new BigDecimal(0.1) 會把
        // double 已經帶有的誤差原封不動搬進來
        BigDecimal a = new BigDecimal("0.1");
        BigDecimal b = new BigDecimal("0.2");

        // BigDecimal 的加法用 add 方法，不是 + 號
        System.out.println("精確結果：" + a.add(b));
    }
}
```

▶ 執行結果

```
精確結果：0.3
```

> **說明**　何時該用什麼：科學計算、遊戲座標、統計分析等「容許微小誤差」的場合，double 又快又省，放心用；金額、費率、發票這類「一分錢都不能錯」的場合，用 BigDecimal，或改用整數以「分」為單位計算。判斷基準只有一個問題：這個數字算錯一點點，會不會有人賠錢？

3.11 綜合範例：BMI 計算機

把本章工具全部組合起來：用 Scanner 讀取身高體重（第 2 章）、浮點數除法計算 BMI、`Math.round` 處理四捨五入，最後用三元運算子給出評語。BMI 的公式是體重（公斤）除以身高（公尺）的平方。

範例：BmiCalculator.java

```java
import java.util.Scanner;

/**
 * BMI 計算機：讀取身高體重，計算 BMI 並給出評語。
 */
public class BmiCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

```

```
        System.out.print("請輸入身高（公尺）：");
        double height = scanner.nextDouble();

        System.out.print("請輸入體重（公斤）：");
        double weight = scanner.nextDouble();

        // 全程使用 double 運算，避免整數除法的截斷
        double bmi = weight / (height * height);

        // 四捨五入到小數點後一位：先放大十倍再縮回來
        double rounded = Math.round(bmi * 10) / 10.0;
        System.out.println("你的 BMI 是：" + rounded);

        // 三元運算子串出三段式評語（兩層還算可讀的極限）
        String comment = (bmi < 18.5) ? "體重偏輕"
                       : (bmi < 24) ? "體重正常" : "體重偏重";
        System.out.println("評估結果：" + comment);

        scanner.close();
    }
}
```

▶ 執行結果

```
請輸入身高（公尺）：1.7
請輸入體重（公斤）：65
你的 BMI 是：22.5
評估結果：體重正常
```

幾個值得回味的細節：`Math.round(bmi * 10) / 10.0` 的分母寫 `10.0` 而不是 `10`，因為 `Math.round` 回傳整數型別，除以整數 10 又會發生截斷；評語的三元運算子刻意排版成兩行，讓「條件、結果」對齊——程式的排版也是溝通的一部分。

3.12 本章小結

- 整數除以整數結果仍是整數（向零截斷）；想要小數，至少一邊要是浮點數。
- `%` 取餘數：判斷奇偶、整除、取位數、循環計數的利器。
- 整數除以零拋出 ArithmeticException；浮點數除以零得到 Infinity 或 NaN。
- `i++` 先用舊值再加一，`++i` 先加一再用新值；單獨成行時兩者相同，也建議只單獨成行使用。

- 複合指定 += 等運算子是就地更新的縮寫，且自帶隱含轉型。
- && 與 || 有短路求值：左邊能決定結果時，右邊不執行；可用來把 null 檢查等「保鏢」放在左邊。
- + 碰到字串就變串接，由左到右感染整條式子；輸出中要做數學運算請加括號。
- 運算子優先順序記不住沒關係：不確定就加括號。
- 浮點數有精度誤差：不要用 == 比較，改用誤差容忍值；金額計算用 BigDecimal（以字串建構）。

3.13 本章練習

動手練習

1. 觀念題：`9 / 2`、`9 % 2`、`9 / 2.0` 的結果分別是什麼？型別又是什麼？（提示：整數除法截斷；有一邊是 double 結果就是 double。）
2. 觀念題：執行 `int i = 5; int j = i++ + ++i;` 之後，i 與 j 分別是多少？請逐步展開。（提示：i++ 交出 5 後 i 變 6；++i 先變 7 再交出 7。）
3. 實作題：寫一個程式判斷某年是否為閏年。規則：能被 4 整除且不能被 100 整除，或能被 400 整除。用一條含 &&、|| 的運算式完成，測試 2024（閏年）、1900（平年）、2000（閏年）。（提示：注意優先順序，善用括號。）
4. 實作題：讀入一個總秒數（例如 3725），輸出「幾小時幾分幾秒」（1 小時 2 分 5 秒）。（提示：除法取商、% 取餘，兩者搭配。）
5. 觀念題：為什麼 `0.1 + 0.2 == 0.3` 是 false？如果是計算購物車金額，你會改用什麼做法？（提示：二進位無法精確表示 0.1；想想 BigDecimal 或整數計價。）
6. 挑戰題：擴充 BMI 計算機——加入「過輕、正常、過重、肥胖」四段評語，並把 BMI 顯示到小數點後兩位。（提示：四捨五入改成乘除 100；評語可用巢狀三元或改用下一章的 if。）

第 4 章

流程控制

本章學習目標

學完本章你將能夠：

- 使用 if、else if、else 依條件讓程式走不同的路。
- 使用傳統 switch 與箭頭語法的 switch，並避開落穿陷阱。
- 依場景選用 while、do-while 與 for 三種迴圈。
- 使用巢狀迴圈處理二維問題，例如九九乘法表。
- 使用 break 與 continue 精準控制迴圈的進行。
- 綜合運用條件與迴圈，寫出完整的猜數字遊戲。

到目前為止，我們寫的程式都是「一本道」：從 main 的第一行走到最後一行，每次執行的路線一模一樣。真實世界的程式顯然不是這樣——密碼對了才能登入、餘額不足要擋下提款、名單有幾個人就寄幾封信。讓程式「看狀況做事」與「重複做事」的語法，統稱流程控制（control flow），它是程式從「計算機」進化成「有邏輯的系統」的關鍵一步。

所有程式的流程都由三種結構組成：**循序**（由上而下依序執行，我們一直在用）、**選擇**（依條件走不同分支，即 if 與 switch）、**重複**（滿足條件時反覆執行，即迴圈）。理論上，任何複雜的程式邏輯都能用這三種結構拼裝出來。本章依序攻克後兩種。

4.1 if：讓程式學會判斷

`if` 的語法直白如英文：如果（if）條件成立，就執行大括號裡的程式；搭配 `else if` 與 `else` 可以串出多路分支。條件必須是 boolean 運算式——上一章的比較與邏輯運算子在此全面上場。

範例：GradeDemo.java

```
/**
 * 成績等第判定：多路分支的標準示範。
 */
public class GradeDemo {
    public static void main(String[] args) {
        int score = 87;

```

```
        // 由高到低逐一檢查：第一個成立的分支執行，其餘全部跳過
        if (score >= 90) {
            System.out.println("甲等：表現優異！");
        } else if (score >= 80) {
            System.out.println("乙等：表現良好。");
        } else if (score >= 70) {
            System.out.println("丙等：還不錯，繼續努力。");
        } else if (score >= 60) {
            System.out.println("丁等：低空飛過。");
        } else {
            // 前面全部不成立才會走到這裡
            System.out.println("不及格：需要加強。");
        }
    }
}
```

▶ 執行結果

```
乙等：表現良好。
```

87 分同時滿足 `score >= 80`、`score >= 70` 與 `score >= 60`，但只印出「乙等」——因為多路分支是**由上而下檢查，命中一個就結束**。這也代表條件的順序至關重要：如果把 `score >= 60` 放在最上面，任何及格分數都會被它攔截，後面的分支永遠沒有出場機會。設計多路分支時，把「最嚴格的條件」放最前面。

> **提示**　即使分支裡只有一行程式，大括號也不要省略。省略大括號的 `if` 只管轄「緊接著的一個敘述」，日後有人在下面加一行程式，就會意外脫離 if 的管轄範圍——這是史上多次重大資安漏洞的成因。花兩個字元買保險，穩賺不賠。

4.1.1 巢狀 if 與衛述句

if 裡面還可以再放 if，稱為巢狀（nested）。巢狀層數一多，程式會向右「漂移」成難以閱讀的階梯。資深工程師偏好「衛述句」（guard clause）風格：先把不合格的情況擋掉並提早離開，主要邏輯就能保持在最外層。

範例：GuardClauseDemo.java

```
/**
 * 巢狀 if 與衛述句的對比：同樣的邏輯，兩種寫法。
 */
public class GuardClauseDemo {
    public static void main(String[] args) {
```

```
        int age = 20;
        boolean hasTicket = true;

        // 寫法一：巢狀 if——層層縮排，條件一多就像迷宮
        if (age >= 18) {
            if (hasTicket) {
                System.out.println("巢狀寫法：歡迎入場！");
            } else {
                System.out.println("巢狀寫法：請先買票。");
            }
        } else {
            System.out.println("巢狀寫法：未滿 18 歲不能入場。");
        }

        // 寫法二：衛述句——先排除異常，主流程不縮排
        if (age < 18) {
            System.out.println("衛述句：未滿 18 歲不能入場。");
            return;     // 提早結束 main 方法
        }
        if (!hasTicket) {
            System.out.println("衛述句：請先買票。");
            return;
        }
        System.out.println("衛述句：歡迎入場！");
    }
}
```

▶ 執行結果

```
巢狀寫法：歡迎入場！
衛述句：歡迎入場！
```

`return` 在這裡的意思是「立刻結束目前的方法」，第 6 章會正式介紹。衛述句的哲學是：**把說「不」的理由講完，剩下的就都是「好」**——條件越多，這種風格的優勢越明顯。

4.2 switch：多選一的專用語法

當分支條件是「同一個變數等於不同的值」時，`switch` 比一長串 else if 更清楚。先看傳統語法——以及它惡名昭彰的陷阱。

4.2.1 傳統語法與落穿陷阱

錯誤示範：SwitchFallThrough.java

```java
/**
 * 傳統 switch 的落穿陷阱：故意漏掉 break，觀察災情。
 */
public class SwitchFallThrough {
    public static void main(String[] args) {
        int day = 3;

        switch (day) {
            case 1:
                System.out.println("星期一");
            case 2:
                System.out.println("星期二");
            case 3:
                System.out.println("星期三");   // 從這裡命中開始
            case 4:
                System.out.println("星期四");   // 沒 break，繼續掉
            default:
                System.out.println("其他日子");
        }
    }
}
```

▶ 執行結果

```
星期三
星期四
其他日子
```

day 是 3，卻連星期四和 default 都印出來了！這就是「落穿」（fall-through）：傳統 switch 命中一個 case 後，會一路往下執行到底，除非遇到 `break` 才停。正確的傳統寫法是每個 case 結尾都加 `break;`。

> **注意**　忘記 `break` 是傳統 switch 最著名的陷阱：程式能編譯、能執行，只是默默多做了一堆事。如果你使用傳統語法，請把「每個 case 都要 break」當成反射動作；更好的選擇是直接使用下面的箭頭語法，從語法層面根絕這個問題。

4.2.2 箭頭語法：Java 14 之後的建議寫法

Java 14 起，switch 支援 `->` 箭頭語法：不會落穿、不需要 break、多個值可用逗號並列，甚至能把整個 switch 當成「會算出值」的運算式使用。

範例：SwitchArrowDemo.java

```java
/**
 * 箭頭語法三連發：敘述形式、運算式形式、yield 區塊。
 */
public class SwitchArrowDemo {
    public static void main(String[] args) {
        int month = 4;

        // 用法一：敘述形式。命中哪個箭頭就只執行那一行
        switch (month) {
            case 12, 1, 2 -> System.out.println("冬天");
            case 3, 4, 5 -> System.out.println("春天");
            case 6, 7, 8 -> System.out.println("夏天");
            case 9, 10, 11 -> System.out.println("秋天");
            default -> System.out.println("月份不正確");
        }

        // 用法二：運算式形式。整個 switch 算出一個值
        int quarter = switch (month) {
            case 1, 2, 3 -> 1;
            case 4, 5, 6 -> 2;
            case 7, 8, 9 -> 3;
            case 10, 11, 12 -> 4;
            default -> 0;
        };
        System.out.println(month + " 月是第 " + quarter + " 季");

        // 用法三：分支需要多行邏輯時，用大括號加 yield 交出值
        String desc = switch (month) {
            case 4 -> {
                String base = "春暖花開";
                yield base + "的四月";   // yield：這個分支的值
            }
            default -> "其他月份";
        };
        System.out.println(desc);
    }
}
```

▶ 執行結果

```
春天
4 月是第 2 季
```

```
春暖花開的四月
```

注意運算式形式的兩個規則：結尾的分號不能省（整個 switch 是指定敘述的一部分）；分支必須涵蓋所有可能值（所以通常需要 default）。

4.3　while：先檢查再執行

`while` 是最單純的迴圈：條件成立就執行主體，執行完回頭再檢查，直到條件不成立。它適合「事先不知道要跑幾圈」的場合。經典範例：把一個整數的位數反轉。

範例：ReverseNumber.java

```java
/**
 * 數字反轉：1234 變 4321。
 * 核心技巧：% 10 取個位數、/ 10 丟掉個位數。
 */
public class ReverseNumber {
    public static void main(String[] args) {
        int number = 1234;
        int reversed = 0;

        System.out.println("原始數字：" + number);

        // 只要還有位數沒拆完，就繼續
        while (number > 0) {
            int digit = number % 10;            // 取出個位數
            reversed = reversed * 10 + digit;   // 接到結果尾端
            number = number / 10;               // 丟掉個位數
            System.out.println("取出 " + digit +
                "，目前結果：" + reversed);
        }

        System.out.println("反轉完成：" + reversed);
    }
}
```

▶ 執行結果

```
原始數字：1234
取出 4，目前結果：4
取出 3，目前結果：43
取出 2，目前結果：432
```

```
取出 1，目前結果：4321
反轉完成：4321
```

請拿紙筆跟著跑一遍：number 依序是 1234、123、12、1、0，reversed 依序是 4、43、432、4321。這種「人腦模擬執行」叫做 trace，是培養程式邏輯最有效的練習，沒有之一。

4.4 do-while：先執行再檢查

`do-while` 把檢查移到迴圈的尾端：主體至少執行一次，之後才開始檢查條件。「至少一次」的天性使它成為輸入驗證的完美選擇——總得先問過一次，才知道輸入合不合法：

範例：InputValidation.java

```java
import java.util.Scanner;

/**
 * 輸入驗證：不斷詢問，直到使用者輸入 1 到 5 之間的數字。
 */
public class InputValidation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int choice;

        // 先執行一次（至少要問一次），再檢查要不要重來
        do {
            System.out.print("請輸入 1 到 5 之間的數字：");
            choice = scanner.nextInt();
        } while (choice < 1 || choice > 5);   // 不合法就再問一次

        System.out.println("你選擇了：" + choice);
        scanner.close();
    }
}
```

▶ 執行結果

```
請輸入 1 到 5 之間的數字：9
請輸入 1 到 5 之間的數字：0
請輸入 1 到 5 之間的數字：3
你選擇了：3
```

與 while 的差異就一句話：while 可能一次都不執行，do-while 保證至少執行一次。注意

do-while 結尾的 `while (...)` 後面有分號，這是初學者常漏掉的小地方。

4.5 for：計數迴圈的標準款

已知「要跑幾圈」時，`for` 把迴圈三要素收納進一行括號：`for (初始化; 條件; 更新)`。三個部分的執行時機是理解 for 的關鍵：

1. **初始化**：進入迴圈前執行一次，通常宣告計數器 `int i = 1`。
2. **條件**：每圈開始前檢查，成立才執行主體，不成立就離開迴圈。
3. **更新**：每圈主體執行完才執行，通常是 `i++`，然後回到步驟 2。

範例：ForSum.java

```
/**
 * 用 for 迴圈計算 1 加到 100。
 * 執行順序：初始化、條件、主體、更新，之後不斷重複
 * 「條件、主體、更新」直到條件不成立。
 */
public class ForSum {
    public static void main(String[] args) {
        int sum = 0;

        for (int i = 1; i <= 100; i++) {
            sum += i;   // 每圈把 i 累加進 sum
        }

        System.out.println("1 加到 100 = " + sum);
    }
}
```

▶ 執行結果

```
1 加到 100 = 5050
```

計數器 `i` 宣告在 for 的括號裡，作用域僅限這個迴圈——迴圈結束後 i 就消失了，這是良好的封閉性。三種迴圈的選擇準則整理如下：

迴圈	檢查時機	適用場景
`for`	每圈開始前	已知次數的計數式重複
`while`	每圈開始前	次數未知，可能一次都不執行
`do-while`	每圈結束後	次數未知，但至少執行一次

4.6 巢狀迴圈：九九乘法表

迴圈裡再放迴圈，就能處理「表格狀」的問題：外層每走一格，內層完整跑完一輪。最經典的示範就是九九乘法表——外層控制「第幾列」，內層控制「第幾行」：

範例：MultiplicationTable.java

```java
/**
 * 九九乘法表：巢狀迴圈的經典應用。
 * 外層 i 是被乘數（列），內層 j 是乘數（行）。
 */
public class MultiplicationTable {
    public static void main(String[] args) {
        for (int i = 1; i <= 9; i++) {
            for (int j = 1; j <= 9; j++) {
                // printf 的 %2d 讓數字至少佔兩格，欄位才會對齊
                //  （printf 的完整用法見第 7 章，先會用即可）
                System.out.printf("%d*%d=%2d ", i, j, i * j);
            }
            System.out.println();   // 一列印完，換行
        }
    }
}
```

▶ 執行結果

```
1*1= 1 1*2= 2 1*3= 3 1*4= 4 1*5= 5 1*6= 6 1*7= 7 1*8= 8 1*9= 9
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
```

讀懂巢狀迴圈的訣竅：盯著外層的一圈看。當 `i` 是 3 時，內層的 `j` 從 1 完整跑到 9，印出整列「3 的乘法」，然後換行；外層才前進到 `i = 4`。總執行次數是外層乘以內層，$9 \times 9 = 81$ 次——巢狀迴圈的成本成長得很快，這也是效能分析中 $O(n^2)$ 的由來。

4.7 break 與 continue

迴圈裡有兩個流程控制的好幫手：`break` 立刻「離開整個迴圈」；`continue` 跳過本圈剩餘的程式，直接進入下一圈。

範例：BreakContinueDemo.java

```java
/**
 * break：找到就走人；continue：這筆跳過，繼續下一筆。
 */
public class BreakContinueDemo {
    public static void main(String[] args) {
        // break 示範：找出 100 以上第一個 7 的倍數
        for (int i = 100; i <= 200; i++) {
            if (i % 7 == 0) {
                System.out.println("找到了：" + i);
                break;      // 任務完成，剩下的不用看了
            }
        }

        // continue 示範：計算 1 到 10 的奇數和
        int sum = 0;
        for (int i = 1; i <= 10; i++) {
            if (i % 2 == 0) {
                continue;   // 偶數：跳過累加，直接下一圈
            }
            sum += i;       // 只有奇數會走到這行
        }
        System.out.println("1 到 10 的奇數和：" + sum);
    }
}
```

▶ 執行結果

```
找到了：105
1 到 10 的奇數和：25
```

4.7.1 標籤 break：一次跳出多層

`break` 預設只跳出「最內層」迴圈。想從巢狀迴圈一次全身而退，可以幫外層迴圈取名字（標籤），再 `break 標籤`：

範例：LabelBreakDemo.java

```java
/**
 * 標籤 break：在巢狀迴圈中找到目標後，一次跳出兩層。
 */
public class LabelBreakDemo {
    public static void main(String[] args) {
        // 找出乘積恰好是 6 的第一組 (i, j)
        outer:                              // 幫外層迴圈取名
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                System.out.println("檢查 " + i + " x " + j);
                if (i * j == 6) {
                    System.out.println("找到：" + i + " x " + j);
                    break outer;            // 直接跳出外層迴圈
                }
            }
        }
        System.out.println("搜尋結束");
    }
}
```

▶ 執行結果

```
檢查 1 x 1
檢查 1 x 2
檢查 1 x 3
檢查 2 x 1
檢查 2 x 2
檢查 2 x 3
找到：2 x 3
搜尋結束
```

若這裡用普通 break，只會跳出內層，外層會繼續跑 `i = 3` 的一輪。標籤 break 使用機會不多，但在多層搜尋時非常乾淨俐落。

> **注意**　迴圈的兩大經典災難：**無窮迴圈**與**差一錯誤**（off-by-one）。無窮迴圈通常是忘了更新條件變數——`while (n > 0)` 的主體裡忘記改 n，程式就永遠繞不出來（此時可按 Ctrl+C 強制終止）。差一錯誤則是邊界少一圈或多一圈：想跑 5 次卻寫 `i < 4`，或想取索引 0 到 4 卻寫 `i <= 5`。防禦方法：邊界值親手代入頭尾各驗一次——「i 第一圈是多少？最後一圈是多少？總共幾圈？」三問答對再往下寫。

4.8 綜合範例：猜數字遊戲

本章的畢業專題：電腦從 1 到 100 隨機挑一個數字，玩家反覆猜測，程式提示「太大」或「太小」，直到猜中。這個小遊戲用到了隨機數、while 迴圈、多路 if 與計數器——正好是本章與前兩章的總複習。

範例：GuessNumber.java

```
import java.util.Random;
import java.util.Scanner;

/**
 * 猜數字遊戲：電腦出題（1 到 100），玩家猜到對為止。
 */
public class GuessNumber {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();

        // nextInt(100) 回傳 0 到 99，加 1 平移成 1 到 100
        int answer = random.nextInt(100) + 1;
        int guess = 0;      // 初始化成不可能的值，確保進入迴圈
        int count = 0;      // 記錄猜測次數

        System.out.println("我想好了一個 1 到 100 的數字！");

        // 還沒猜中就一直問
        while (guess != answer) {
            System.out.print("請輸入你的猜測：");
            guess = scanner.nextInt();
            count++;

            if (guess > answer) {
                System.out.println("太大了，再小一點。");
            } else if (guess < answer) {
                System.out.println("太小了，再大一點。");
            } else {
                System.out.println("恭喜答對！你總共猜了 " +
                    count + " 次。");
            }
        }
        scanner.close();
```

```
    }
}
```

以下是答案恰好是 58 的一次實際遊玩過程（每次執行答案都不同）：

▶ 執行結果

```
我想好了一個 1 到 100 的數字！
請輸入你的猜測：50
太小了，再大一點。
請輸入你的猜測：75
太大了，再小一點。
請輸入你的猜測：60
太大了，再小一點。
請輸入你的猜測：55
太小了，再大一點。
請輸入你的猜測：58
恭喜答對！你總共猜了 5 次。
```

提示　觀察上面的遊玩策略：每次都猜「剩餘範圍的中間值」，範圍就會對半縮小——這正是資訊科學中大名鼎鼎的「二分搜尋」。用這個策略，1 到 100 的數字最多 7 次一定猜中（$2^7 = 128 > 100$）。另外，產生隨機數也可以用 `Math.random()`，它回傳 0 到 1 之間的 double；`(int) (Math.random() * 100) + 1` 的效果與本例相同，兩種寫法都值得認識。

4.9 本章小結

- 程式流程由循序、選擇、重複三種結構組成。
- if 與 else if 由上而下檢查，命中即止；條件順序至關重要，最嚴格的放前面；大括號永遠不省略。
- 衛述句風格：先擋掉異常情況並提早 return，主邏輯不縮排。
- 傳統 switch 命中後會一路落穿，每個 case 都要 break；Java 14 的箭頭語法不落穿、可並列多值、可當運算式（多行分支用 yield 交值）。
- while 先檢查再執行；do-while 先執行再檢查（至少一次），適合輸入驗證。
- for 三要素：初始化執行一次；條件每圈開始前檢查；更新在主體之後執行。
- 巢狀迴圈處理表格狀問題，總次數是內外層相乘。
- break 跳出整個迴圈，continue 跳過本圈；標籤 break 可一次跳出多層。
- 小心無窮迴圈（忘記更新條件變數）與差一錯誤（邊界多一圈或少一圈），邊界值要親手

驗證。

4.10 本章練習

動手練習

1. 觀念題：while 與 do-while 的差別是什麼？「請使用者輸入密碼，直到正確為止」適合哪一種？為什麼？（提示：至少要問一次。）
2. 觀念題：傳統 switch 忘記寫 break 會發生什麼事？箭頭語法為什麼沒有這個問題？（提示：落穿。）
3. 實作題：經典 FizzBuzz——印出 1 到 30，遇到 3 的倍數印 Fizz、5 的倍數印 Buzz、同時是 3 和 5 的倍數印 FizzBuzz。（提示：先檢查 15 的倍數，順序錯了就全毀。）
4. 實作題：用迴圈印出 1000 以內的費氏數列：1、1、2、3、5、8、13……（提示：用兩個變數存「前兩項」，每圈輪流更新。）
5. 實作題：用巢狀迴圈印出 5 層的星號直角三角形——第 1 層 1 顆星，第 5 層 5 顆星。（提示：內層迴圈的次數取決於外層的計數器。）
6. 挑戰題：改良猜數字遊戲——限制最多 7 次機會，次數用完印出「挑戰失敗，答案是 X」；並在玩家輸入不在 1 到 100 範圍時提示「範圍錯誤」且不消耗次數。（提示：while 條件加上次數判斷，範圍檢查用 continue。）

第 5 章

陣列

本章學習目標

學完本章你將能夠：

- 宣告、建立與初始化陣列，並用索引存取元素。
- 解釋索引從 0 開始的規則，避開越界例外。
- 依場景選用傳統 for 與增強式 for 走訪陣列。
- 說明陣列是參考型別的意義，並正確複製陣列。
- 實作找最大值、加總、反轉、搜尋等基礎演算法。
- 使用二維陣列處理表格資料，完成成績統計程式。

想像你是老師，要記錄全班 100 位學生的成績。用上一章的知識，你得宣告 `score1`、`score2` 一路寫到 `score100`——光宣告就一百行，想算平均還得把一百個變數名字全部加起來。這顯然不對勁。程式世界的解法是「陣列」（array）：一排連續的格子，共用一個名字，用編號區分彼此。一個變數，就能管理一百筆、一萬筆資料。

陣列也是我們遇到的第一個「參考型別」實例。上一章埋下的伏筆——盒子裡放的是遙控器，不是電視——會在本章第一次真正發揮威力（也埋下第一批地雷）。請特別留意「複製陣列」那一節。

5.1 宣告與建立陣列

建立陣列有兩種方式：知道長度但還沒有資料時，用 `new 型別[長度]`；資料已經確定時，直接用大括號字面值。無論哪種方式，陣列的長度一旦決定就不能改變。

範例：ArrayBasics.java

```
/**
 * 陣列的兩種建立方式與索引存取。
 */
public class ArrayBasics {
    public static void main(String[] args) {
        // 方式一：指定長度建立，每格自動填入預設值 0
        int[] scores = new int[5];
```

```
        scores[0] = 90;     // 索引從 0 開始！
        scores[1] = 85;
        scores[4] = 70;     // 最後一格的索引是「長度減 1」

        System.out.println("第一格：" + scores[0]);
        System.out.println("最後一格：" + scores[4]);
        System.out.println("沒指定的格子：" + scores[2]);

        // 方式二：用字面值直接給資料，長度自動算好
        int[] primes = {2, 3, 5, 7, 11};
        System.out.println("陣列長度：" + primes.length);
        System.out.println("第三個質數：" + primes[2]);
    }
}
```

▶ 執行結果

```
第一格：90
最後一格：70
沒指定的格子：0
陣列長度：5
第三個質數：5
```

三個立刻要內化的重點：

- **索引從 0 開始**：第一個元素是 `scores[0]`，長度為 5 的陣列，合法索引是 0 到 4。
- **length 是屬性不是方法**：陣列用 `scores.length`（沒有括號）；之後學的字串卻是 `s.length()` 方法，這個不一致是 Java 的歷史包袱，只能記住。
- **元素有預設值**：數值型別是 0、`boolean` 是 false、參考型別是 `null`——所以上例沒指定的 `scores[2]` 印出 0。

5.1.1 越界：最常見的執行期錯誤

拿不存在的索引存取陣列，編譯器不會發現（它不追蹤執行期的值），但執行時 JVM 會立刻拋出例外：

錯誤示範：IndexError.java

```
public class IndexError {
    public static void main(String[] args) {
        int[] data = {10, 20, 30};

```

```
        System.out.println("長度：" + data.length);
        System.out.println("最後一個元素：" + data[2]);

        // 長度是 3，合法索引只有 0、1、2——data[3] 不存在！
        System.out.println(data[3]);
    }
}
```

▶ 執行結果

```
長度：3
最後一個元素：30
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
    ↪ bounds for length 3
    at IndexError.main(IndexError.java:9)
```

> **注意**　ArrayIndexOutOfBoundsException（陣列索引越界）是初學者最頻繁遇到的執行期例外，九成的成因是「差一錯誤」：忘記索引從 0 開始，或迴圈條件寫成 `i <= data.length`（多跑一圈）。黃金守則：最後一個元素永遠是 `data[data.length - 1]`；迴圈條件永遠是 `i < data.length`（小於，不是小於等於）。

5.2 走訪陣列：兩種 for 的選擇

逐一處理陣列中的每個元素稱為「走訪」（traverse）。傳統 for 迴圈透過索引存取；Java 5 之後的「增強式 for」（for-each）語法更簡潔，直接把每個元素依序交到你手上：

範例：TraverseDemo.java

```
/**
 * 傳統 for 與增強式 for 的對照。
 */
public class TraverseDemo {
    public static void main(String[] args) {
        int[] scores = {88, 92, 79, 65, 70};

        // 傳統 for：有索引 i 可用，適合需要「位置」的場合
        for (int i = 0; i < scores.length; i++) {
            System.out.println("第 " + (i + 1) + " 位：" + scores[i]);
        }

        // 增強式 for：讀作「對 scores 裡的每個 score」
```

```
        // 不需要索引時，這種寫法更簡潔、更不會越界
        int sum = 0;
        for (int score : scores) {
            sum += score;
        }
        System.out.println("總分：" + sum);
    }
}
```

▶ 執行結果

```
第 1 位：88
第 2 位：92
第 3 位：79
第 4 位：65
第 5 位：70
總分：394
```

兩種寫法的取捨：

面向	傳統 for	增強式 for
取得索引位置	可以	不行
修改元素內容	可以（`scores[i] = ...`）	不行（拿到的是複本）
倒著走、跳著走	可以	不行（固定從頭到尾）
語法簡潔安全	普通	佳（不可能越界）

原則：**單純讀取全部元素用增強式 for；需要索引或要改值時用傳統 for**。注意增強式 for 中對迴圈變數賦值（如 `score = 0`）改的只是複本，原陣列毫髮無傷——這是初學者常見的誤解。

5.3 陣列是參考型別

第 2 章說過：參考型別的變數裡放的是「遙控器」。現在來看這件事的實際後果。把陣列變數指定給另一個變數，複製的只是遙控器——兩個變數控制的是同一排格子：

範例：ArrayReference.java

```
import java.util.Arrays;

/**
 * 陣列是參考型別：等號只複製「遙控器」，不複製陣列本身。
 */
```

```java
public class ArrayReference {
    public static void main(String[] args) {
        int[] original = {1, 2, 3};

        // 這行「沒有」複製陣列！兩個變數指向同一個陣列
        int[] alias = original;
        alias[0] = 99;      // 透過 alias 改.....
        System.out.println("original[0] = " + original[0]);

        // 正確複製方式一：Arrays.copyOf（最常用）
        int[] copy = Arrays.copyOf(original, original.length);
        copy[0] = 1;        // 改複製品，本尊不受影響
        System.out.println("original[0] = " + original[0]);
        System.out.println("copy[0] = " + copy[0]);

        // 正確複製方式二：System.arraycopy（較底層）
        // 參數：來源, 來源起點, 目的地, 目的起點, 複製幾個
        int[] copy2 = new int[3];
        System.arraycopy(original, 0, copy2, 0, 3);
        System.out.println("copy2 = " + Arrays.toString(copy2));
    }
}
```

▶ 執行結果

```
original[0] = 99
original[0] = 99
copy[0] = 1
copy2 = [99, 2, 3]
```

第一行輸出是本節的靈魂：明明改的是 `alias[0]`，`original[0]` 卻跟著變成 99——因為它們是同一個陣列的兩個名字。想要真正的獨立複本，必須用 `Arrays.copyOf` 或 `System.arraycopy` 把每個格子的值搬過去。

> **說明**　比較陣列也有同樣的陷阱：`a == b` 比的是「兩支遙控器是否控制同一台電視」，不是內容！比較兩個陣列的內容是否相同，要用 `Arrays.equals(a, b)`。同理，直接 `System.out.println(a)` 印出的是型別加上雜湊碼（類似 `[I@7ad041f3` 的神祕文字），想看內容請用 `Arrays.toString(a)`。

5.4 陣列基礎演算法

接下來是陣列的實戰價值：把「走訪」與「條件判斷」組合成常用演算法。這些模式未來會反覆出現，值得練到閉著眼睛都寫得出來。

5.4.1 找最大值、最小值與平均

範例：MaxMinAverage.java

```
/**
 * 一趟走訪同時找出最大值、最小值與總和。
 */
public class MaxMinAverage {
    public static void main(String[] args) {
        int[] scores = {72, 88, 95, 60, 79};

        // 起手式：先假設第一個元素就是最大值兼最小值
        int max = scores[0];
        int min = scores[0];
        int sum = 0;

        for (int score : scores) {
            if (score > max) {
                max = score;    // 看到更大的，更新擂台主
            }
            if (score < min) {
                min = score;
            }
            sum += score;
        }

        // 平均要轉型成 double，否則整數除法會截斷小數
        double average = (double) sum / scores.length;
        System.out.println("最高分：" + max);
        System.out.println("最低分：" + min);
        System.out.println("總分：" + sum);
        System.out.println("平均：" + average);
    }
}
```

▶ 執行結果

```
最高分：95
最低分：60
總分：394
平均：78.8
```

5.4.2 反轉與線性搜尋

範例：ReverseAndSearch.java

```java
import java.util.Arrays;

/**
 * 就地反轉陣列，以及最樸素的搜尋法——線性搜尋。
 */
public class ReverseAndSearch {
    public static void main(String[] args) {
        int[] data = {10, 20, 30, 40, 50};

        // 反轉：頭尾指標互換內容，一路向中間逼近
        int left = 0;
        int right = data.length - 1;
        while (left < right) {
            int temp = data[left];      // 暫存，否則值會被蓋掉
            data[left] = data[right];
            data[right] = temp;
            left++;
            right--;
        }
        System.out.println("反轉後：" + Arrays.toString(data));

        // 線性搜尋：從頭到尾逐一比對，找到就停
        int target = 30;
        int foundIndex = -1;    // -1 是「沒找到」的慣用暗號
        for (int i = 0; i < data.length; i++) {
            if (data[i] == target) {
                foundIndex = i;
                break;          // 已找到，不必再看後面
            }
        }
        System.out.println(target + " 位於索引：" + foundIndex);
    }
```

```
}
```

▶ 執行結果

```
反轉後：[50, 40, 30, 20, 10]
30 位於索引：2
```

交換兩個值必須借助第三個變數 temp——直接互相指定會讓其中一個值先被蓋掉。這個「三杯水交換」手法是基本功，第 6 章討論方法傳參時它還會再登場。

5.5 Arrays 工具類別

自己寫演算法是為了理解原理；日常開發則直接用 Java 內建的 `java.util.Arrays` 工具類別，它把常用操作全包了：

範例：ArraysUtil.java

```java
import java.util.Arrays;

/**
 * Arrays 工具類別常用方法巡禮。
 */
public class ArraysUtil {
    public static void main(String[] args) {
        int[] data = {5, 2, 8, 1, 9};

        // toString：轉成好讀的文字
        System.out.println("排序前：" + Arrays.toString(data));

        // sort：由小到大排序（直接修改原陣列）
        Arrays.sort(data);
        System.out.println("排序後：" + Arrays.toString(data));

        // binarySearch：二分搜尋。前提：陣列必須已排序！
        int index = Arrays.binarySearch(data, 8);
        System.out.println("8 的索引：" + index);

        // fill：整個陣列填滿同一個值
        int[] filled = new int[5];
        Arrays.fill(filled, 7);
        System.out.println("填滿 7：" + Arrays.toString(filled));

```

```
        // equals：逐格比較內容（回顧：== 比的是遙控器）
        int[] a = {1, 2, 3};
        int[] b = {1, 2, 3};
        System.out.println("a == b：" + (a == b));
        System.out.println("內容相同：" + Arrays.equals(a, b));
    }
}
```

▶ 執行結果

```
排序前：[5, 2, 8, 1, 9]
排序後：[1, 2, 5, 8, 9]
8 的索引：3
填滿 7：[7, 7, 7, 7, 7]
a == b：false
內容相同：true
```

> **提示**　binarySearch 之所以要求「先排序」，是因為二分搜尋的原理是不斷把搜尋範圍砍半：跟中間值比大小，就能丟掉一半的資料。對已排序的百萬筆資料，線性搜尋最多要看一百萬次，二分搜尋只要約 20 次（$O(n)$ 對 $O(\log n)$ 的差距）。對未排序的陣列呼叫它，會得到毫無意義的結果。

5.6　二維陣列

陣列的每個格子也可以再放一個陣列，形成「二維陣列」——本質是「陣列的陣列」，用來表示表格：第一個索引選「列」，第二個索引選「行」。典型應用：多位學生、多個科目的成績表。

範例：TwoDimensional.java

```
/**
 * 二維陣列：3 位學生乘 3 個科目的成績表。
 * scores[i][j] 代表第 i 位學生第 j 科的分數。
 */
public class TwoDimensional {
    public static void main(String[] args) {
        int[][] scores = {
            {85, 92, 78},   // 學生 0 的國文、英文、數學
            {60, 71, 66},   // 學生 1
            {90, 88, 95}    // 學生 2
        };

```

```
        // 外層走「每一位學生」，內層走「每一個科目」
        for (int i = 0; i < scores.length; i++) {
            int total = 0;
            for (int j = 0; j < scores[i].length; j++) {
                total += scores[i][j];
            }
            // 平均四捨五入到小數點後一位
            double avg = Math.round(total / 3.0 * 10) / 10.0;
            System.out.println("學生 " + (i + 1) + "：總分 " +
                total + "，平均 " + avg);
        }
    }
}
```

▶ 執行結果

```
學生 1：總分 255，平均 85.0
學生 2：總分 197，平均 65.7
學生 3：總分 273，平均 91.0
```

讀法解密：`scores.length` 是列數（學生數 3），`scores[i].length` 是第 i 列的長度（科目數 3），`scores[1][2]` 是學生 1 的第 2 科成績（66）。

5.6.1 不規則陣列

既然二維陣列是「陣列的陣列」，每一列的長度自然可以不同——只要先固定列數，再逐列建立不同長度的內層陣列，這稱為不規則陣列（jagged array）：

範例：JaggedArray.java

```
/**
 * 不規則陣列：每一列的長度可以不同。
 */
public class JaggedArray {
    public static void main(String[] args) {
        // 先只決定「有 3 列」，每列的長度稍後各自決定
        int[][] triangle = new int[3][];
        triangle[0] = new int[]{1};
        triangle[1] = new int[]{1, 2};
        triangle[2] = new int[]{1, 2, 3};

        // 增強式 for 也能走訪二維陣列：外層拿到的是「一列」
        for (int[] row : triangle) {
```

```
            for (int value : row) {
                System.out.print(value + " ");
            }
            System.out.println();
        }
    }
}
```

▶ 執行結果

```
1
1 2
1 2 3
```

5.7 綜合範例：成績統計程式

期末總驗收：給定全班成績，計算總分、平均、最高最低分、高於平均的人數，以及各分數區間的人數分布。這支程式綜合了本章所有技巧——多趟走訪、複製排序、計數器陣列：

範例：GradeStats.java

```
import java.util.Arrays;

/**
 * 成績統計：總分、平均、極值、高於平均人數與區間分布。
 */
public class GradeStats {
    public static void main(String[] args) {
        int[] scores = {72, 88, 95, 60, 79, 83, 51, 90};

        // 第一步：總分與平均
        int sum = 0;
        for (int s : scores) {
            sum += s;
        }
        double average = (double) sum / scores.length;

        // 第二步：最高最低分——先複製再排序，保護原始資料
        int[] sorted = Arrays.copyOf(scores, scores.length);
        Arrays.sort(sorted);
        int min = sorted[0];
        int max = sorted[sorted.length - 1];
```

```
22
23          // 第三步：高於平均的人數
24          int aboveCount = 0;
25          for (int s : scores) {
26              if (s > average) {
27                  aboveCount++;
28              }
29          }
30
31          // 第四步：區間分布——用小陣列當「計數器」
32          // buckets[0]：60 以下，buckets[1]：60 到 79，buckets[2]：80 以上
33          int[] buckets = new int[3];
34          for (int s : scores) {
35              if (s < 60) {
36                  buckets[0]++;
37              } else if (s < 80) {
38                  buckets[1]++;
39              } else {
40                  buckets[2]++;
41              }
42          }
43
44          System.out.println("人數：" + scores.length);
45          System.out.println("總分：" + sum);
46          System.out.println("平均：" + average);
47          System.out.println("最高分：" + max + "，最低分：" + min);
48          System.out.println("高於平均：" + aboveCount + " 人");
49          System.out.println("不及格 " + buckets[0] +
50              " 人，及格 " + buckets[1] +
51              " 人，優良 " + buckets[2] + " 人");
52      }
53  }
```

▶ 執行結果

```
人數：8
總分：618
平均：77.25
最高分：95，最低分：51
高於平均：5 人
不及格 1 人，及格 3 人，優良 4 人
```

值得學走的兩個設計決策：排序前先 `copyOf` 複製，避免破壞原始順序（呼應本章的參考

型別觀念）；區間統計用「計數器陣列」而不是三個獨立變數——如果將來要分十個區間，陣列版本幾乎不用改。

5.8 本章小結

- 陣列用一個名字管理一整排同型別資料；長度固定，用 `length` 屬性取得。
- 索引從 0 開始，最後一格是 `length - 1`；越界會拋出 ArrayIndexOutOfBoundsException。
- 元素有預設值：數值 0、boolean 為 false、參考型別為 null。
- 單純讀取用增強式 for；需要索引或修改元素用傳統 for。
- 陣列是參考型別：= 只複製參考，兩個變數會指向同一個陣列；真複製用 `Arrays.copyOf` ↪ 。
- 比內容用 `Arrays.equals`、印內容用 `Arrays.toString`，不要用 == 與直接列印。
- 基礎演算法四件套：找極值（擂台法）、加總平均（記得轉型）、反轉（頭尾指標）、線性搜尋（找到就 break）。
- `Arrays.sort` 就地排序；`binarySearch` 只對已排序陣列有效。
- 二維陣列是「陣列的陣列」，外層列內層行；每列長度可以不同（不規則陣列）。

5.9 本章練習

動手練習

1. 觀念題：長度為 10 的陣列，合法索引範圍是多少？`for (int i = 0; i <= 10; i` ↪ `++)` 走訪它會發生什麼事？（提示：差一錯誤與越界例外。）
2. 觀念題：`int[] b = a;` 之後修改 `b[0]`，`a[0]` 會不會變？如果希望不會，該怎麼寫？（提示：參考複製與 Arrays.copyOf 的差別。）
3. 實作題：建立一個 int 陣列存放 1 到 20，用一個迴圈把偶數元素全部改成它的兩倍，最後用 `Arrays.toString` 印出。（提示：需要修改元素，選哪種 for？）
4. 實作題：寫程式找出陣列 `{4, 9, 2, 9, 1}` 中最大值「第一次出現」的索引。（提示：擂台法，同時記錄值與位置。）
5. 實作題：宣告二維陣列存放 4 位學生的 2 科成績，印出每位學生的總分，以及「每一科」的全班平均。（提示：科目平均要固定行、走訪列——內外層迴圈對調。）
6. 挑戰題：不使用 Arrays.sort，自己實作選擇排序：每一輪從未排序區找最小值，換到未排序區的最前面。（提示：雙層迴圈加上「三杯水」交換。）

第 6 章

方法

本章學習目標

學完本章你將能夠：

- 定義方法並呼叫它，正確區分參數與引數。
- 使用回傳值與 void，並用 return 提早結束方法。
- 解釋 Java 的傳值呼叫，預測基本型別與陣列參數的行為差異。
- 使用方法多載與可變參數設計彈性的介面。
- 寫出正確終止的遞迴方法，並追蹤它的呼叫過程。
- 說明變數作用域，並依單一職責原則重構程式。

寫到現在，你的 main 方法應該越來越長了——計算、驗證、輸出全部擠在一起，想重複某段邏輯只能複製貼上。複製貼上是程式設計的萬惡之源：同一段邏輯散落多處，改需求時漏改一處就是臭蟲。業界把解藥稱為 DRY 原則（Don't Repeat Yourself，別重複你自己），而 Java 實踐 DRY 的基本工具就是「方法」（method）：把一段有名字的邏輯封裝起來，需要時呼叫名字即可。

方法帶來三個立即的好處：**重用**（寫一次，用一百次）、**可讀**（`calculateBmi(w, h)` 比二十行計算式更能表達意圖）、**好測試**（小積木可以個別驗證）。本章從動機出發，一路學到方法的進階玩法——多載、可變參數與遞迴。

6.1 為什麼需要方法

先看「沒有方法」的痛。下面的程式想在報表中畫三條分隔線，同一行程式被複製了三次——今天想把等號換成減號，就得改三個地方：

範例：RepeatPain.java

```
/**
 * 反面教材：畫分隔線的邏輯被複製了三次。
 */
public class RepeatPain {
    public static void main(String[] args) {
        System.out.println("==============================");
```

```
        System.out.println("成績單");
        System.out.println("==============================");
        System.out.println("國文：90 分");
        System.out.println("==============================");
    }
}
```

▶ 執行結果

```
==============================
成績單
==============================
國文：90 分
==============================
```

用方法重構：把「畫分隔線」封裝起來，定義一次、呼叫三次：

範例：LinePrinter.java

```
/**
 * 用方法消除重複：分隔線的畫法只定義一次。
 */
public class LinePrinter {

    /** 印出一條 30 個等號組成的分隔線 */
    static void printLine() {
        System.out.println("==============================");
    }

    public static void main(String[] args) {
        printLine();                    // 呼叫：跳去執行方法
        System.out.println("成績單");    // 執行完回到這裡繼續
        printLine();
        System.out.println("國文：90 分");
        printLine();
    }
}
```

輸出與前一版完全相同，但質變發生了：想改分隔線樣式，現在只需要改 **一個地方**。注意執行流程——程式遇到 `printLine();` 時「跳」進方法執行，執行完再「跳」回原處繼續往下走。

方法定義的完整語法解剖（以 `static int add(int a, int b)` 為例）：

- `static`：修飾詞。本章的方法都加 static，讓 main 能直接呼叫（背後原因在第 8 章揭曉）。
- `int`：回傳型別——方法算完後交回什麼型別的值；不交回任何值就寫 `void`。
- `add`：方法名稱，慣例用「動詞開頭的 camelCase」。
- `(int a, int b)`：參數列——這個方法需要外界提供的材料清單，每個參數都要寫型別。

6.2 參數、引數與回傳值

兩個容易混用的術語，其實分工明確：參數（parameter）是方法定義處的「材料清單」，引數（argument）是呼叫時實際遞過去的「材料」。以下面的程式為例，`width` 與 `height` 是參數，呼叫處的 5 與 3 是引數。

範例：AreaCalculator.java

```
/**
 * 參數、引數、回傳值與 return 提早結束的完整示範。
 */
public class AreaCalculator {

    /**
     * 計算矩形面積。
     * 參數 width、height 是方法內部的區域變數，
     * 呼叫時由引數的值來初始化。
     */
    static int rectangleArea(int width, int height) {
        int area = width * height;
        return area;    // 把結果交回呼叫端，方法就此結束
    }

    /** 印出報告。不需要交回任何值，回傳型別寫 void */
    static void printReport(String name, int area) {
        if (area <= 0) {
            System.out.println(name + "：尺寸不合法！");
            return;     // void 方法也能用 return 提早離開
        }
        System.out.println(name + " 的面積是 " + area);
    }

    public static void main(String[] args) {
        // 回傳值可以存進變數……
        int a1 = rectangleArea(5, 3);
```

```
        printReport("客廳", a1);

        // ……也可以直接嵌在更大的運算式裡
        System.out.println("兩房總面積：" +
            (rectangleArea(4, 3) + rectangleArea(2, 3)));

        // 引數也可以是另一個方法的回傳值
        printReport("儲藏室", rectangleArea(0, 5));
    }
}
```

▶ 執行結果

```
客廳 的面積是 15
兩房總面積：18
儲藏室：尺寸不合法！
```

關於 `return` 的三個規則：有回傳型別的方法，每條執行路徑都必須 return 一個對應型別的值；`void` 方法可以完全不寫 return，也可以用「不帶值的 return」提早離開（衛述句的好搭檔）；return 之後的程式碼永遠執行不到，編譯器甚至會直接報錯（unreachable statement）。

6.3 Java 是傳值呼叫

重頭戲來了。呼叫方法時，引數是怎麼交給參數的？Java 的答案永遠只有一種：**傳值呼叫**（pass by value）——把引數的值複製一份給參數。這句話對基本型別與參考型別會產生截然不同的表面行為，是面試的常客，也是初學者的大魔王：

範例：PassByValue.java

```
/**
 * 傳值呼叫的兩張面孔：交換失敗與修改陣列成功。
 */
public class PassByValue {

    /** 企圖交換兩個整數——注定失敗 */
    static void swap(int x, int y) {
        int temp = x;   // x、y 只是 a、b 的「複本」
        x = y;
        y = temp;
        System.out.println("方法內：x = " + x + ", y = " + y);
    }

```

```
    /** 把陣列的第一個元素改成 99——會成功！ */
    static void modifyFirst(int[] arr) {
        // arr 是參考的複本，但它指向的是「同一個」陣列
        arr[0] = 99;
    }

    public static void main(String[] args) {
        int a = 1;
        int b = 2;
        swap(a, b);
        System.out.println("方法外：a = " + a + ", b = " + b);

        int[] numbers = {1, 2, 3};
        modifyFirst(numbers);
        System.out.println("陣列內容：" + numbers[0] + ", " +
            numbers[1] + ", " + numbers[2]);
    }
}
```

▶ 執行結果

```
方法內：x = 2, y = 1
方法外：a = 1, b = 2
陣列內容：99, 2, 3
```

> **說明**　用遙控器比喻一次講透。基本型別：盒子裡放的是「值本身」，複製給參數的就是值的複本——方法內怎麼折騰複本，原變數都不動，所以交換失敗。參考型別（陣列）：盒子裡放的是「遙控器」，複製給參數的是遙控器的複本——遙控器有兩支，電視只有一台。透過複本遙控器按下「把第一格改成 99」，唯一那台電視當然變了。但請注意：如果方法內寫 `arr = new int[3];`，只是讓複本遙控器改配對到新電視，呼叫端的遙控器毫無感覺。一句話總結：**Java 複製的永遠是盒子裡的東西——值或參考——而不是它們指向的實體。**

6.4 方法多載

同一個類別裡可以有多個**同名**方法，只要它們的參數列（型別、數量或順序）不同，這稱為多載（overloading）。編譯器會根據你傳入的引數自動挑選版本。其實你早就在用多載了——`println` 能印字串、整數、小數，正是因為它有一大票多載版本。

範例：OverloadDemo.java

```java
/**
 * 方法多載：同名不同參數列，編譯器依引數挑選版本。
 */
public class OverloadDemo {

    static int add(int a, int b) {
        System.out.print("[int 版本] ");
        return a + b;
    }

    static double add(double a, double b) {
        System.out.print("[double 版本] ");
        return a + b;
    }

    static int add(int a, int b, int c) {
        System.out.print("[三參數版本] ");
        return a + b + c;
    }

    public static void main(String[] args) {
        System.out.println(add(1, 2));
        System.out.println(add(1.5, 2.5));
        System.out.println(add(1, 2, 3));
    }
}
```

▶ 執行結果

```
[int 版本] 3
[double 版本] 4.0
[三參數版本] 6
```

注意　多載只看參數列，與回傳型別無關。兩個方法若參數列完全相同、只有回傳型別不同，編譯器直接報錯——因為呼叫時它無從分辨你要哪一個。另外，多載版本之間的行為應該「語意一致」：都叫 add 就都該做加法，若 int 版做加法、double 版做乘法，使用者遲早翻車。

6.5 可變參數 varargs

如果連「參數個數」都想開放呢？在型別後面加三個點（`int... nums`），方法就能接受任意數量的引數，方法內部把它當成陣列使用：

範例：VarargsDemo.java

```
/**
 * 可變參數：呼叫端想傳幾個就傳幾個。
 */
public class VarargsDemo {

    /** nums 在方法內就是一個 int 陣列 */
    static int sum(int... nums) {
        int total = 0;
        for (int n : nums) {
            total += n;
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(sum());              // 傳 0 個也合法
        System.out.println(sum(10));
        System.out.println(sum(1, 2, 3));
        System.out.println(sum(1, 2, 3, 4, 5));
    }
}
```

▶ 執行結果

```
0
10
6
15
```

規則有二：一個方法只能有一個可變參數，而且它必須排在參數列的最後——否則編譯器無法判斷哪些引數屬於誰。例如 `static void log(String tag, int... values)` 合法；把兩者順序對調則不合法。

6.6 遞迴：呼叫自己的方法

方法可以呼叫別的方法，那能不能呼叫自己？可以，這叫遞迴（recursion）。遞迴的思考方式是「把大問題丟給小一號的自己」：5 的階乘是「5 乘上 4 的階乘」，4 的階乘是「4 乘上 3 的階乘」……一路縮小到「1 的階乘就是 1」為止。每個遞迴都必須有兩個要素：

- **終止條件**（base case）：小到不能再小、可以直接回答的情況。沒有它，遞迴就是無底洞。
- **遞迴關係**：如何把問題縮小一號，並用小問題的答案組出大問題的答案。

範例：Factorial.java

```java
/**
 * 用遞迴計算階乘：n! = n * (n-1) * ... * 2 * 1。
 */
public class Factorial {

    static long factorial(int n) {
        // 終止條件：1 以下直接回答，不再往下呼叫
        if (n <= 1) {
            return 1;
        }
        // 遞迴關係：把 n-1 的問題交給「小一號的自己」
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(i + "! = " + factorial(i));
        }
    }
}
```

▶ 執行結果

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

呼叫 `factorial(3)` 時，記憶體裡的「呼叫堆疊」是這樣運作的（每一層呼叫都會疊上一個新的工作紀錄，算完再一層層拆掉）：

- `factorial(3)` 需要 `3 * factorial(2)`，先暫停，往下呼叫。
- `factorial(2)` 需要 `2 * factorial(1)`，再暫停，再往下。
- `factorial(1)` 命中終止條件，直接回傳 1。
- 回到 `factorial(2)`：`2 * 1` 得 2，回傳。
- 回到 `factorial(3)`：`3 * 2` 得 6，回傳。完工。

> **注意**　忘記寫終止條件（或條件永遠碰不到），方法就會無限呼叫自己，呼叫堆疊被塞爆，JVM 拋出 `StackOverflowError`——史上最有名的程式錯誤之一，知名問答網站 Stack Overflow 就是以它命名的。寫遞迴的第一步永遠是：**先寫終止條件，再寫遞迴關係，並確認每次遞迴都朝終止條件前進。**

再看一個經典：費氏數列（每一項是前兩項之和）。它有兩個終止條件與兩路遞迴呼叫：

範例：Fibonacci.java

```java
/**
 * 費氏數列：1, 1, 2, 3, 5, 8, ...（前兩項之和）。
 */
public class Fibonacci {

    static long fib(int n) {
        if (n <= 2) {
            return 1;   // 終止條件：第 1、2 項定義為 1
        }
        return fib(n - 1) + fib(n - 2);   // 兩路遞迴
    }

    public static void main(String[] args) {
        // 印出數列的前 10 項
        for (int i = 1; i <= 10; i++) {
            System.out.print(fib(i) + " ");
        }
        System.out.println();
    }
}
```

▶ 執行結果

```
1 1 2 3 5 8 13 21 34 55
```

> **說明**　這個寫法直觀，但效能很差：`fib(50)` 會觸發指數等級的重複計算（`fib(48)` 被算了兩次、`fib(47)` 三次……）。用第 4 章的迴圈改寫只需線性時間。這是重要的一課：**遞迴是思考工具，不一定是最佳實作**——能自然轉成迴圈的遞迴，通常轉掉更好；真正適合遞迴的是樹狀結構等「天生巢狀」的問題（中級篇再會）。

6.7　變數作用域

作用域（scope）指「變數在哪個範圍內存在且可用」。Java 的規則很一致：**變數活在宣告它的那對大括號裡，出了大括號就消失**。參數的作用域是整個方法；迴圈變數的作用域只在迴圈裡：

範例：ScopeDemo.java

```java
/**
 * 作用域示範：變數只活在宣告它的大括號內。
 */
public class ScopeDemo {

    static void printSquare(int n) {    // 參數 n：整個方法可用
        int square = n * n;             // 區域變數：只活在此方法
        System.out.println(n + " 的平方是 " + square);
    }

    public static void main(String[] args) {
        printSquare(4);

        for (int i = 1; i <= 2; i++) {  // i 只活在這個 for 裡
            int doubled = i * 2;        // 每圈都是全新的 doubled
            System.out.println("i = " + i + "，兩倍 = " + doubled);
        }

        // 下面兩行若取消註解都會編譯錯誤：
        // System.out.println(i);       // i 已隨迴圈結束而消失
        // System.out.println(square);  // square 屬於別的方法
    }
}
```

▶ 執行結果

```
4 的平方是 16
i = 1，兩倍 = 2
```

```
i = 2，兩倍 = 4
```

作用域不是刁難，而是保護：範圍越小，變數能被誤用的機會就越少。好習慣是**在最接近使用處、最小的範圍內宣告變數**。

最後認識一個名詞：**遮蔽**（shadowing）——內層範圍出現與外層同名的變數時，內層名稱會暫時「遮住」外層。Java 不允許方法裡的區域變數彼此同名，所以初級階段少有機會撞上；但「參數與類別欄位同名」是合法且常見的遮蔽，第 8 章介紹 `this` 關鍵字時，你就會親手解決這個歧義。

> **提示**　方法設計的四條軍規：(1) **單一職責**——一個方法只做一件事，「計算並列印並存檔」就是三件事，拆開；(2) **命名用動詞**，名字要能讀出行為，如 `calculateTotal`、`isValidEmail`；(3) **參數不要超過三四個**，太多代表這個方法管太寬；(4) **長度以一眼看完為度**，超過一個螢幕就該考慮拆分。評斷標準只有一個：半年後的你能不能十秒內看懂它。

6.8　綜合範例：重構簡易計算機

驗收時間：把「讀輸入、算結果、印報告」全部拆成方法，讓 main 縮成一份流程總表。對照看看：同樣的功能，這一版的每個片段都可以獨立理解、獨立重用：

範例：CalculatorV2.java

```
import java.util.Scanner;

/**
 * 簡易計算機 2.0：main 只負責指揮，實際工作全部交給方法。
 */
public class CalculatorV2 {

    /** 顯示提示文字並讀取一個整數 */
    static int readNumber(Scanner scanner, String prompt) {
        System.out.print(prompt);
        return scanner.nextInt();
    }

    /** 依運算符號計算；除以零時提示並回傳 0 */
    static double calculate(int a, int b, char op) {
        switch (op) {
            case '+': return a + b;
            case '-': return a - b;
```

```
            case '*': return a * b;
            case '/':
                if (b == 0) {
                    System.out.println("除數不能為零！");
                    return 0;
                }
                return (double) a / b;   // 轉型避免整數除法
            default:
                System.out.println("不支援的運算：" + op);
                return 0;
        }
    }

    /** 統一的結果輸出格式 */
    static void printResult(int a, int b, char op, double value) {
        System.out.println(a + " " + op + " " + b + " = " + value);
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int a = readNumber(scanner, "請輸入第一個數字：");
        int b = readNumber(scanner, "請輸入第二個數字：");

        // 同一組數字，四種運算各展示一次
        printResult(a, b, '+', calculate(a, b, '+'));
        printResult(a, b, '-', calculate(a, b, '-'));
        printResult(a, b, '*', calculate(a, b, '*'));
        printResult(a, b, '/', calculate(a, b, '/'));

        scanner.close();
    }
}
```

▶ 執行結果

```
請輸入第一個數字：12
請輸入第二個數字：5
12 + 5 = 17.0
12 - 5 = 7.0
12 * 5 = 60.0
12 / 5 = 2.4
```

順帶一提：`calculate` 裡的 `case '+': return a + b;` 沒有寫 break——因為 `return` 已經直接離開整個方法，落穿根本無從發生。這是傳統 switch 少數可以安心省略 break 的場合。

6.9 本章小結

- 方法實踐 DRY 原則：邏輯只定義一次，到處呼叫；同時提升可讀性與可測試性。
- 定義語法四件套：修飾詞、回傳型別（無則 void）、動詞命名、參數列；參數是清單，引數是實料。
- 有回傳型別的方法每條路徑都要 return；void 方法可用「不帶值的 return」提早離開。
- Java 永遠是傳值呼叫：基本型別複製值（方法內改不動外面），參考型別複製參考（可透過它修改同一個物件的內容）。
- 多載：同名方法靠參數列區分，與回傳型別無關；語意要一致。
- 可變參數 `int... nums` 在方法內是陣列；只能有一個且必須放參數列最後。
- 遞迴必備終止條件與遞迴關係；忘記終止條件會導致 StackOverflowError；能自然改寫成迴圈的遞迴通常該改寫。
- 變數活在宣告它的大括號內；在最小必要範圍內宣告變數。
- 方法設計軍規：單一職責、動詞命名、參數精簡、長度一眼可讀。

6.10 本章練習

動手練習

1. 觀念題：參數與引數差在哪裡？在定義 `static int max(int a, int b)` 與呼叫式 `max(3, 7)` 中，誰是參數、誰是引數？（提示：定義處與呼叫處。）
2. 觀念題：為什麼 `swap(a, b)` 交換不了 main 裡的兩個 int，但方法卻能修改傳入陣列的內容？（提示：複製的是值還是遙控器。）
3. 實作題：寫 `static boolean isPrime(int n)` 判斷質數，並用它印出 2 到 50 之間的所有質數。（提示：從 2 檢查到 n 的平方根即可；先處理 n 小於 2 的情況。）
4. 實作題：多載三個 `max` 方法：兩個 int、三個 int、兩個 double，並在 main 各呼叫一次驗證。（提示：三參數版可以重用兩參數版：max(max(a, b), c)。）
5. 實作題：用遞迴寫 `static int sumTo(int n)` 計算 1 加到 n；先寫終止條件。再思考：n 給 100000 會發生什麼事？（提示：堆疊深度是有限的。）
6. 重構題：把第 5 章的成績統計程式重構成多個方法：`sum`、`average`、`maxOf` 各自獨立，main 只負責呼叫與輸出。（提示：每個方法都接收 int 陣列參數。）

第 7 章

字串處理

本章學習目標

學完本章你將能夠：

- 解釋字串的不可變性與字串常數池的基本概念。
- 正確使用 equals 比較字串內容，遠離 == 陷阱。
- 熟練運用查詢、擷取、轉換、分割等常用字串方法。
- 完成字串與數字的互相轉換，並認識轉換失敗的例外。
- 說明何時該用 StringBuilder 取代字串串接。
- 使用 String.format 與 printf 做格式化輸出。

回顧一下：我們其實從第 1 章的 `"Hello, World!"` 開始就天天在用字串（`String`）了。字串是程式與人類交會的地方——使用者輸入的是文字，網頁與報表輸出的是文字，檔案與網路傳輸的內容大多也是文字。可以說，實務程式有一半的工作是在「搬運與加工文字」，字串因此是所有型別中方法最豐富、使用頻率最高的一個。

字串也是你正式深入研究的第一個參考型別。它有兩個與直覺相悖的特性——「不可變」與「== 比較的陷阱」——每年榨乾無數初學者的除錯時間。本章先把這兩顆地雷拆掉，再帶你巡禮常用方法，最後用一個文字分析器收尾。

7.1 建立字串與不可變性

建立字串最自然的方式就是字面值：`String s = "Java";`。關於它的第一個重要事實是：**字串物件一旦建立，內容就永遠不能改變**（immutable，不可變）。所有看起來像「修改」的方法，其實都是造一個新字串還給你，原字串分毫未動：

範例：ImmutableDemo.java

```
/**
 * 字串的不可變性：所謂「修改」其實是產生新字串。
 */
public class ImmutableDemo {
    public static void main(String[] args) {
        String original = "Hello";
```

```
7
8           // concat 不會改動 original，而是回傳一個新字串
9           String result = original.concat(", Java!");
10          System.out.println("original = " + original);
11          System.out.println("result = " + result);
12
13          // toUpperCase 也一樣：原字串不動如山
14          String upper = original.toUpperCase();
15          System.out.println("upper = " + upper);
16          System.out.println("original 還是：" + original);
17      }
18  }
```

▶ 執行結果

```
original = Hello
result = Hello, Java!
upper = HELLO
original 還是：Hello
```

所以「想改字串」的正確姿勢是**把回傳值接回來**：`s = s.toUpperCase();`。忘記接回傳值、以為呼叫方法就會就地修改，是這個章節第一常見的錯誤。

> **說明**　為什麼要設計成不可變？主因有三：(1) **安全**——字串被大量用於檔名、帳號、網址，若可變，傳出去的字串可能被別人偷改；(2) **共享**——正因為不可變，JVM 才敢把相同內容的字面值放進「字串常數池」(string pool) 共用一份，省下大量記憶體：寫兩次 `"java"` 字面值，拿到的其實是同一個物件；(3) **執行緒安全**——多執行緒同時讀它也不會出事（進階篇再談）。

7.2 比較字串：== 與 equals

初學者陷阱排行榜的不動冠軍來了。比較兩個字串的內容是否相同，必須用 `equals()` 方法；`==` 比的是「兩個變數是否指向同一個物件」（還記得遙控器嗎？`==` 問的是「同一台電視嗎」，不是「畫面一樣嗎」）：

範例：EqualsDemo.java

```
1   /**
2    * == 與 equals 的差異：比「同一個物件」還是比「內容」。
3    */
4   public class EqualsDemo {
```

```
    public static void main(String[] args) {
        String s1 = "java";
        String s2 = "java";              // 與 s1 共用常數池的同一份
        String s3 = new String("java");  // 強迫建立全新的物件

        System.out.println("s1 == s2：" + (s1 == s2));
        System.out.println("s1 == s3：" + (s1 == s3));
        System.out.println("s1.equals(s3)：" + s1.equals(s3));

        // 忽略大小寫的內容比較
        String s4 = "JAVA";
        System.out.println("忽略大小寫：" + s1.equalsIgnoreCase(s4));

        // compareTo：依字典順序比較，回傳負數、零或正數
        System.out.println("apple 對 banana：" +
            "apple".compareTo("banana"));
        System.out.println("apple 對 apple：" +
            "apple".compareTo("apple"));
    }
}
```

▶ 執行結果

```
s1 == s2：true
s1 == s3：false
s1.equals(s3)：true
忽略大小寫：true
apple 對 banana：-1
apple 對 apple：0
```

> **注意**　最陰險的地方在於 `s1 == s2` 竟然是 true——因為兩個字面值共用常數池的同一個物件，讓你誤以為 `==` 可以用。等到字串來自使用者輸入、檔案或 `new String()` 時，`==` 就會突然回報 false，臭蟲難找得要命。鐵律：**字串內容比較，永遠用 equals，沒有例外**。順帶的防禦技巧：把已知常數放前面寫成 `"yes".equals(input)`，即使 input 是 null 也不會爆炸。

7.3　常用方法巡禮

String 的方法非常多，依用途分成「查詢」與「加工」兩批來認識。先是查詢類——問長度、問位置、問內容：

範例：InspectDemo.java

```java
/**
 * 查詢類方法：length、charAt、substring、indexOf 等。
 */
public class InspectDemo {
    public static void main(String[] args) {
        String s = "Hello, Java!";
        // 索引尺標：H0 e1 l2 l3 o4 逗5 空6 J7 a8 v9 a10 驚11

        System.out.println("長度：" + s.length());
        System.out.println("索引 7 的字元：" + s.charAt(7));

        // substring(起點)：從起點取到底
        System.out.println("取 7 到底：" + s.substring(7));
        // substring(起點, 終點)：含起點、不含終點！
        System.out.println("取 7 到 10：" + s.substring(7, 11));

        System.out.println("第一個 a 在：" + s.indexOf("a"));
        System.out.println("最後一個 a 在：" + s.lastIndexOf("a"));
        System.out.println("包含 Java？" + s.contains("Java"));
        System.out.println("以 Hello 開頭？" + s.startsWith("Hello"));
        System.out.println("以驚嘆號結尾？" + s.endsWith("!"));
    }
}
```

▶ 執行結果

```
長度：12
索引 7 的字元：J
取 7 到底：Java!
取 7 到 10：Java
第一個 a 在：8
最後一個 a 在：10
包含 Java？true
以 Hello 開頭？true
以驚嘆號結尾？true
```

特別注意 `substring(7, 11)` 的規則：**含起點、不含終點**（數學上的半開區間）。好處是「終點減起點」剛好等於取出的長度：$11 - 7 = 4$ 個字元，正是 Java 四個字母。找不到時，`indexOf` 回傳 -1，這是慣用的「沒找到」訊號，寫條件判斷時記得處理。

接著是加工類——去空白、取代、切割、組合：

範例：TransformDemo.java

```java
/**
 * 加工類方法：trim、replace、split、join 與大小寫轉換。
 */
public class TransformDemo {
    public static void main(String[] args) {
        String messy = "  Java 好好玩  ";

        // trim 去掉頭尾半形空白；strip（Java 11+）更聰明，
        // 連 Unicode 的各種空白字元（如全形空白）都能處理
        System.out.println("[" + messy.trim() + "]");
        System.out.println("[" + messy.strip() + "]");

        // replace：把「所有」出現的目標全部換掉
        System.out.println("banana".replace('a', 'o'));

        // 大小寫轉換
        System.out.println("Java".toUpperCase());
        System.out.println("Java".toLowerCase());

        // split：依分隔符號切成字串陣列
        String csv = "蘋果,香蕉,芭樂";
        String[] fruits = csv.split(",");
        for (String fruit : fruits) {
            System.out.println("水果：" + fruit);
        }

        // String.join：split 的相反——把多段文字接起來
        String joined = String.join(" > ", "青銅", "白銀", "黃金");
        System.out.println(joined);

        // isEmpty：長度為 0 才 true；isBlank：全是空白也 true
        System.out.println("".isEmpty() + " / " + "  ".isEmpty());
        System.out.println("".isBlank() + " / " + "  ".isBlank());
    }
}
```

▶ 執行結果

```
[Java 好好玩]
[Java 好好玩]
bonono
JAVA
```

```
java
水果：蘋果
水果：香蕉
水果：芭樂
青銅 > 白銀 > 黃金
true / false
true / true
```

7.3.1 逐字元走訪

需要逐字檢查內容時（統計、驗證、加密都用得到），有兩種慣用姿勢：用索引搭配 `charAt`，或先 `toCharArray()` 轉成字元陣列再用增強式 for：

範例：VowelCounter.java

```java
/**
 * 逐字元走訪的兩種方式：統計母音數量。
 */
public class VowelCounter {
    public static void main(String[] args) {
        String word = "programming";
        int count = 0;

        // 方式一：索引搭配 charAt
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            if (c == 'a' || c == 'e' || c == 'i' ||
                c == 'o' || c == 'u') {
                count++;    // 是母音，計數器加一
            }
        }
        System.out.println("母音數量：" + count);

        // 方式二：轉成字元陣列，用增強式 for
        for (char c : word.toCharArray()) {
            System.out.print(c + "-");
        }
        System.out.println();
    }
}
```

▶ 執行結果

```
母音數量：3
p-r-o-g-r-a-m-m-i-n-g-
```

7.4 字串與數字互轉

使用者輸入、檔案內容讀進來都是字串，運算前得先轉成數字；反向的轉換也一樣常見。轉換失敗（例如把 `"abc"` 當數字）會拋出 NumberFormatException：

範例：ParseDemo.java

```
1  public class ParseDemo {
2      public static void main(String[] args) {
3          // 字串轉數字：Integer.parseInt 與 Double.parseDouble
4          String input = "123";
5          int number = Integer.parseInt(input);
6          System.out.println(number + 1);     // 數字加法
7
8          double d = Double.parseDouble("3.14");
9          System.out.println(d * 2);
10
11         // 數字轉字串：String.valueOf（或直接串接空字串）
12         String text = String.valueOf(42);
13         System.out.println(text + 1);       // 這是字串串接！
14
15         // 轉換失敗會拋出例外，程式中斷
16         int bad = Integer.parseInt("abc");
17         System.out.println(bad);            // 永遠執行不到
18     }
19 }
```

▶ 執行結果

```
124
6.28
421
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
    at java.base/java.lang.Integer.parseInt(Integer.java:668)
    at ParseDemo.main(ParseDemo.java:16)
```

（例外訊息的中段行號依 JDK 版本而略有差異。）對照第三行輸出與第一行輸出，體會 124

與 421 的差別：一個是數字加法，一個是字串串接——第 3 章的加號陷阱在轉換場景再次現身。如何優雅地「接住」轉換失敗的例外而不讓程式中斷，是中級篇例外處理章節的主題。

7.5 StringBuilder：可變的字串工作檯

字串不可變帶來一個效能隱憂：**在迴圈裡用 + 反覆串接**時，每一圈都要建立新字串、把舊內容全部複製過去。串接 n 次的總複製量約是 $O(n^2)$——一萬次串接就要搬動約五千萬個字元。解法是 `StringBuilder`：一個可變的字串工作檯，在同一塊緩衝區上修改，接近 $O(n)$：

範例：BuilderDemo.java

```java
/**
 * StringBuilder 基本操作：append、insert、delete、reverse。
 */
public class BuilderDemo {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Java");

        sb.append(" 好好玩");       // 在尾端追加（就地修改！）
        System.out.println(sb);

        sb.insert(0, "學 ");        // 在索引 0 處插入
        System.out.println(sb);

        sb.delete(0, 2);            // 刪除索引 0 到 1 的字元
        System.out.println(sb);

        sb.reverse();               // 就地反轉
        System.out.println(sb);

        // 需要真正的 String 時，呼叫 toString 轉換
        String result = sb.reverse().toString();
        System.out.println("最終結果：" + result);
    }
}
```

▶ 執行結果

```
Java 好好玩
學 Java 好好玩
Java 好好玩
玩好好 avaJ
```

```
最終結果：Java 好好玩
```

效能差距不是理論，跑給你看——同樣串接五萬次，兩種寫法的耗時：

範例：ConcatPerformance.java

```java
/**
 * 效能對決：迴圈中 String 串接 vs StringBuilder。
 */
public class ConcatPerformance {
    public static void main(String[] args) {
        int times = 50_000;

        // 方式一：String 的 +=（每圈都複製整個舊字串）
        long start1 = System.currentTimeMillis();
        String s = "";
        for (int i = 0; i < times; i++) {
            s += "x";
        }
        long time1 = System.currentTimeMillis() - start1;

        // 方式二：StringBuilder 的 append（就地追加）
        long start2 = System.currentTimeMillis();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < times; i++) {
            sb.append("x");
        }
        long time2 = System.currentTimeMillis() - start2;

        System.out.println("String 串接耗時：" + time1 + " 毫秒");
        System.out.println("StringBuilder 耗時：" + time2 + " 毫秒");
    }
}
```

▶ 執行結果

```
String 串接耗時：1268 毫秒
StringBuilder 耗時：3 毫秒
```

（實際數字依電腦而異，但數量級的差距一目瞭然。）

> **提示**　選擇準則很簡單：少量、一次性的串接用 + 就好（編譯器會自動優化單一運算式裡的串接）；迴圈中的累積式串接，一律改用 **StringBuilder**。另外你可能會查到另一個類似的類別 StringBuffer：它是執行緒安全但較慢的舊版本，單執行緒程式（目前你寫的都是）用 StringBuilder 即可。

7.6 格式化輸出

把變數塞進固定格式的文字時，用 + 串接會又長又醜。`String.format` 回傳格式化後的字串，`System.out.printf` 則直接輸出，兩者共用同一套「格式符號」：以百分號開頭的佔位符：

符號	意義	範例輸出
`%d`	十進位整數	`42`
`%f`	浮點數（預設 6 位小數）	`3.140000`
`%.2f`	保留兩位小數（四捨五入）	`3.14`
`%s`	字串	`Java`
`%c`	單一字元	`A`
`%b`	布林值	`true`
`%,d`	加千分位逗號的整數	`1,234,567`
`%8.1f`	寬度 8、一位小數、靠右	`     3.1`
`%n`	換行	—
`%%`	百分比符號本身	`%`

範例：FormatDemo.java

```java
/**
 * String.format 與 printf：同一套格式符號的兩種用法。
 */
public class FormatDemo {
    public static void main(String[] args) {
        String name = "小華";
        int age = 20;
        double height = 1.687;

        // String.format：組出字串，先存起來再使用
        String intro = String.format("%s 今年 %d 歲", name, age);
        System.out.println(intro);

        // printf：直接輸出。%.2f 會四捨五入到兩位小數
        System.out.printf("身高：%.2f 公尺%n", height);
        System.out.printf("存款：%,d 元%n", 12345678);
```

```
        System.out.printf("靠右對齊：[%8.1f]%n", 3.14);
        System.out.printf("達成率：%d%%%n", 95);
    }
}
```

▶ 執行結果

```
小華 今年 20 歲
身高：1.69 公尺
存款：12,345,678 元
靠右對齊：[     3.1]
達成率：95%
```

7.6.1 文字區塊：多行字串的救星

Java 15 正式加入「文字區塊」（text block）：用三個雙引號包住多行文字，不必再用 \n 與跳脫字元拼裝，寫 JSON、SQL、HTML 範本時特別舒服：

範例：TextBlockDemo.java

```
/**
 * 文字區塊（Java 15+）：所見即所得的多行字串。
 */
public class TextBlockDemo {
    public static void main(String[] args) {
        // 傳統寫法：跳脫字元讓人眼花
        String oldJson = "{\n  \"name\": \"Java\",\n  \"age\": 30\n}";
        System.out.println(oldJson);

        // 文字區塊：內容怎麼排，輸出就怎麼排
        //  （共同的前導縮排會被自動去除）
        String newJson = """
            {
              "name": "Java",
              "age": 30
            }""";
        System.out.println(newJson);
    }
}
```

▶ 執行結果

```
{
```

```
  "name": "Java",
  "age": 30
}
{
  "name": "Java",
  "age": 30
}
```

7.7 綜合範例：文字分析器

期末專題：給定一段文字，統計字元數與單字數、找出出現最多次的字母，並提供迴文判斷（正著讀與倒著讀相同，如 level）。本例綜合了走訪、計數器陣列（第 5 章）、方法拆分（第 6 章）與本章的各路字串技巧：

範例：TextAnalyzer.java

```java
/**
 * 文字分析器：字元數、單字數、最高頻字母與迴文判斷。
 */
public class TextAnalyzer {
    public static void main(String[] args) {
        String text = "banana boat";

        // 統計不含空白的字元數
        int charCount = 0;
        for (char c : text.toCharArray()) {
            if (c != ' ') {
                charCount++;
            }
        }
        System.out.println("字元數（不含空白）：" + charCount);

        // 單字數：用空白切開後數段數
        String[] words = text.split(" ");
        System.out.println("單字數：" + words.length);

        // 找最高頻字母：長度 26 的計數器陣列
        int[] freq = new int[26];
        for (char c : text.toCharArray()) {
            if (c >= 'a' && c <= 'z') {
                freq[c - 'a']++;    // 'a' 對應 0，'b' 對應 1……
```

```
            }
        }
        int maxIndex = 0;
        for (int i = 1; i < 26; i++) {
            if (freq[i] > freq[maxIndex]) {
                maxIndex = i;       // 出現更高頻的字母，更新
            }
        }
        char mostFrequent = (char) ('a' + maxIndex);
        System.out.println("最高頻字母：" + mostFrequent +
            " (" + freq[maxIndex] + " 次) ");

        // 迴文判斷交給獨立方法，main 保持清爽
        System.out.println("level 是迴文？" + isPalindrome("level"));
        System.out.println("java 是迴文？" + isPalindrome("java"));
    }

    /** 頭尾雙指標逐一比對，全部相同即為迴文 */
    static boolean isPalindrome(String s) {
        int left = 0;
        int right = s.length() - 1;
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                return false;   // 一發現不同，立刻下結論
            }
            left++;
            right--;
        }
        return true;
    }
}
```

▶ 執行結果

```
字元數（不含空白）：10
單字數：2
最高頻字母：a（4 次）
level 是迴文？true
java 是迴文？false
```

程式裡的 `freq[c - 'a']++` 是精巧的一手：字元本質是整數（第 2 章），`'b' - 'a'` 等於 1，於是 26 個小寫字母恰好對應到陣列索引 0 到 25——不需要 26 個 if，一行搞定所有字母的

計數。

7.8 本章小結

- 字串不可變：所有「修改」方法都回傳新字串，記得把回傳值接回來。
- 相同的字面值共用字串常數池的同一個物件；`new String` 則強制建立新物件。
- 比較內容永遠用 `equals`（或 equalsIgnoreCase）；== 比的是「是否同一個物件」。
- 查詢類方法：length、charAt、substring（含頭不含尾）、indexOf（找不到回傳 -1）、contains、startsWith 與 endsWith。
- 加工類方法：replace、trim 與 strip、大小寫轉換、split 切割與 String.join 組合、isEmpty 與 isBlank。
- 字串轉數字用 `Integer.parseInt` 等；失敗會拋出 NumberFormatException；數字轉字串用 `String.valueOf`。
- 迴圈中累積串接請用 StringBuilder（append、insert、delete、reverse），效能與 + 串接有數量級差距。
- 格式化輸出：`String.format` 回傳字串、`printf` 直接輸出；常用符號 `%d`、`%s`、`%.2f`、`%n`。
- 文字區塊（三個雙引號）讓多行字串所見即所得。

7.9 本章練習

動手練習

1. 觀念題：`String a = "hi"; String b = new String("hi");` 之後，`a == b` 與 `a.equals(b)` 各是什麼？為什麼？（提示：常數池與 new 的差別。）
2. 觀念題：`"HelloWorld".substring(5, 8)` 的結果是什麼？（提示：含起點、不含終點，先畫索引尺標。）
3. 實作題：讀入一個 email 字串，檢查是否恰好包含一個小老鼠符號且不以它開頭結尾，並取出帳號與網域分別印出。（提示：indexOf、lastIndexOf 與 substring 的組合。）
4. 實作題：把句子 `"the quick brown fox"` 的每個單字字首轉成大寫，輸出 The Quick Brown Fox。（提示：split 切開，charAt(0) 配 toUpperCase，用 StringBuilder 組回去。）
5. 實作題：寫方法統計一段文字中每個「數字字元」出現的次數，例如 `"a1b22c333"` 輸出 1 出現 1 次、2 出現 2 次、3 出現 3 次。（提示：計數器陣列配 `c - '0'`。）

6. 挑戰題：強化迴文判斷——忽略大小寫與空白，讓「A man a plan a canal Panama」也判定為迴文。（提示：先 toLowerCase 再 replace 掉空白，或雙指標跳過空白。）

第 8 章

物件導向入門

本章學習目標

學完本章你將能夠：

- 解釋類別與物件的關係，定義自己的類別。
- 使用 new 建立物件，存取欄位並呼叫方法。
- 使用建構子初始化物件，並運用 this 與建構子多載。
- 用 private 欄位加上 getter 與 setter 實現封裝。
- 分辨 static 成員與實體成員的差異與適用場景。
- 完成一個具備驗證邏輯的完整 BankAccount 類別。

本冊的壓軸登場。到目前為止，我們的程式由「零散的變數」與「處理它們的方法」組成——這種風格稱為程序式程式設計。它應付小程式綽綽有餘，但當資料一多、關係一複雜，變數與方法就會像沒有收納的房間一樣失控。物件導向（Object-Oriented Programming，OOP）是 Java 給出的收納哲學：**把資料與操作資料的行為，打包成一個個「物件」**。

Java 從骨子裡就是物件導向語言——你寫的第一行程式就住在 class 裡，用了七章的 `String`、`Scanner`、`Arrays` 全是別人寫好的類別。本章終於輪到你自己設計類別。這是初級篇最重要的一章，也是通往中級篇（繼承、多型、介面）的大門。

8.1 為什麼需要物件導向

先感受一次「沒有物件」的混亂。想管理三位學生的姓名與成績，程序式的寫法只能攤出一堆平行變數：

反面教材：ProceduralPain.java

```
/**
 * 沒有物件的世界：三位學生的資料散落在六個變數裡。
 */
public class ProceduralPain {
    public static void main(String[] args) {
        String name1 = "小明";
        int score1 = 90;
```

```
        String name2 = "小美";
        int score2 = 85;
        String name3 = "小華";
        int score3 = 77;

        // 印每位學生都得小心對準編號，張冠李戴不會有任何警告
        System.out.println(name1 + "：" + score1 + " 分");
        System.out.println(name2 + "：" + score2 + " 分");
        System.out.println(name3 + "：" + score3 + " 分");
    }
}
```

▶ 執行結果

```
小明：90 分
小美：85 分
小華：77 分
```

問題一目瞭然：name2 與 score2 之間的「屬於同一位學生」關係，只存在你的腦中，語言完全沒有幫你把關——寫成 name2 + "：" + score3 也照樣編譯執行。學生增加到三十位、每位再加上座號與電話，這種寫法就徹底崩潰了。第 5 章的陣列能解決「量」的問題，卻還是把姓名陣列與成績陣列拆成兩個世界。我們真正需要的是一種「把相關資料綁在一起」的型別——這就是類別。

8.2 類別與物件：設計圖與成品

類別（class）是設計圖，**物件**（object）是照著設計圖蓋出來的成品（也稱實體，instance）。「學生」類別描述每位學生都有姓名與成績、都能自我介紹；「小明」「小美」則是照這張圖各自蓋出來的兩個物件，資料各自獨立。定義類別的語法：

範例：Student.java

```
/**
 * 學生類別：把「一位學生的資料與行為」定義在一起。
 * 這是一張設計圖，本身還不是任何一位具體的學生。
 */
public class Student {
    // 欄位（field）：每個物件都擁有自己的一份
    String name;    // 姓名
    int score;      // 成績

    /** 方法：印出這位學生自己的資訊 */
```

```
    void printInfo() {
        // 這裡的 name 與 score，指的是「目前這個物件」的欄位
        System.out.println(name + "：" + score + " 分");
    }
}
```

這個檔案沒有 main 方法——設計圖自己不會動。用另一個類別來「蓋房子」，兩個檔案放在同一個資料夾即可：

範例：StudentDemo.java

```
/**
 * 用 Student 設計圖建立物件（實體）並操作它們。
 */
public class StudentDemo {
    public static void main(String[] args) {
        // new：照設計圖在記憶體中蓋出一個全新的 Student 物件
        Student s1 = new Student();
        s1.name = "小明";       // 用「變數.欄位」讀寫資料
        s1.score = 90;

        Student s2 = new Student();
        s2.name = "小美";
        s2.score = 85;

        // 用「變數.方法()」呼叫物件的行為
        s1.printInfo();
        s2.printInfo();

        // 兩個物件完全獨立：改 s1 不影響 s2
        s1.score = 95;          // 小明補考加分
        s1.printInfo();
        s2.printInfo();         // 小美不受影響
    }
}
```

▶ 執行結果

```
小明：90 分
小美：85 分
小明：95 分
小美：85 分
```

`new Student()` 那一刻發生的事，用文字描繪記憶體的樣貌：

- JVM 在「堆積」（heap）記憶體配置一塊空間，放進一個新的 Student 物件，其中有 name 與 score 兩個欄位。
- 變數 `s1` 本身只是一支遙控器（參考），存放這個物件的位置——第 5 章陣列的規則在此完全適用。
- 再 new 一次，堆積裡出現第二個獨立的物件，`s2` 指向它。兩台電視、兩支遙控器，互不干擾。
- 如果寫 `Student s3 = s1;`，那只是複製遙控器——`s3.score = 0` 會讓 s1 的成績一起歸零，小心。

8.2.1 欄位的預設值

第 2 章說過「區域變數用前必須初始化」，欄位卻不同：物件誕生時，每個欄位自動獲得型別的預設值：

範例：DefaultValueDemo.java

```
/**
 * 欄位有預設值，區域變數沒有——用前者驗證給你看。
 */
public class DefaultValueDemo {
    int count;          // 數值型別預設 0
    double ratio;       // 預設 0.0
    boolean active;     // 預設 false
    String label;       // 參考型別預設 null

    public static void main(String[] args) {
        DefaultValueDemo demo = new DefaultValueDemo();
        System.out.println("count = " + demo.count);
        System.out.println("ratio = " + demo.ratio);
        System.out.println("active = " + demo.active);
        System.out.println("label = " + demo.label);

        // 區域變數就沒這種待遇，下面取消註解會編譯錯誤：
        // int local;
        // System.out.println(local);   // might not have been
        //                              // initialized
    }
}
```

▶ 執行結果

```
count = 0
ratio = 0.0
active = false
label = null
```

> **注意**　「欄位有預設值、區域變數沒有」是考試與面試的常客，更是實務地雷：參考型別欄位的預設值是 `null`，忘記初始化就呼叫它的方法（如 `label.length()`），執行期會拋出 NullPointerException ——Java 世界最惡名昭彰的例外。好習慣：物件建立時就用「建構子」把每個欄位初始化成有意義的值——先看看這顆地雷爆炸的樣子，建構子隨後登場。

8.2.2　null 與 NullPointerException

參考型別欄位沒有初始化時，裡面裝的是 `null`——代表「沒有指向任何物件」的特殊值。對 `null` 呼叫方法或存取欄位，程式會在執行期當場中止：

範例：NullPointerDemo.java

```
/**
 * 對 null 參考呼叫方法：Java 最著名的執行期例外。
 */
public class NullPointerDemo {
    static String message;   // 參考型別欄位，預設值是 null

    public static void main(String[] args) {
        System.out.println("message 的值：" + message);

        // 對 null 呼叫方法，下一行會拋出例外並中止程式
        int len = message.length();
        System.out.println("長度：" + len);    // 永遠執行不到
    }
}
```

▶ 執行結果

```
message 的值：null
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.
    ↪ length()" because "NullPointerDemo.message" is null
    at NullPointerDemo.main(NullPointerDemo.java:11)
```

學會解讀這段訊息，除錯效率立刻翻倍：`Cannot invoke` 後面是「想對 null 做的事」，`because` 後面直接點名「誰是 null」，最後一行給出出事的檔名與行號。修正方向只有兩個：在使用前確實把它初始化，或先用 `if (message != null)` 檢查再使用。

8.3 建構子：物件的出廠設定

剛才的流程「先 new、再一項項指定欄位」既囉嗦又危險——忘了設定某個欄位也沒人提醒。**建構子**（constructor）解決這個問題：它是與類別**同名、沒有回傳型別**的特殊方法，在 `new` 的當下自動執行，負責把物件初始化成可用狀態。搭配 `this` 關鍵字（代表「目前這個物件」），欄位與參數同名也不會混淆：

範例：Point.java

```
/**
 * 平面座標點：示範建構子多載與 this 的兩種用法。
 */
public class Point {
    int x;
    int y;

    /** 無參數建構子：把點放在原點 */
    Point() {
        // this(...)：呼叫本類別的另一個建構子，避免重複邏輯
        this(0, 0);
        System.out.println("（用無參數建構子建立）");
    }

    /** 雙參數建構子：直接指定座標 */
    Point(int x, int y) {
        // this.x 是欄位，x 是參數——this 讓兩者涇渭分明
        this.x = x;
        this.y = y;
        System.out.println("建立點 (" + x + ", " + y + ")");
    }

    public static void main(String[] args) {
        Point origin = new Point();     // 觸發無參數版本
        Point p = new Point(3, 4);      // 觸發雙參數版本

        System.out.println("origin：(" + origin.x + ", " +
            origin.y + ")");
        System.out.println("p：(" + p.x + ", " + p.y + ")");
```

```
    }
}
```

▶ 執行結果

```
建立點 (0, 0)
（用無參數建構子建立）
建立點 (3, 4)
origin：(0, 0)
p：(3, 4)
```

觀察輸出順序：`new Point()` 先跳去執行 `this(0, 0)`（所以先印出「建立點 (0, 0)」），回來才印無參數版本自己的訊息。規則：`this(...)` 只能放在建構子的第一行。

> **注意**　你從沒寫過建構子，之前卻能 `new Student()`？因為類別完全沒有定義建構子時，編譯器會偷偷送你一個空的「預設建構子」。但 **只要你定義了任何一個建構子，這份贈品就會被收回**——此時再呼叫 `new Student()` 會編譯錯誤（constructor Student in class Student cannot be applied to given types）。若兩種用法都要支援，請像上例一樣把無參數建構子明確寫出來。

8.4　封裝：把資料鎖進保險箱

目前 Student 的欄位是裸奔狀態：任何人都能寫 `s1.score = -999;`，資料的正確性毫無防線。物件導向的第一大支柱「封裝」（encapsulation）就是為此而生：**欄位宣告成 `private`（僅限類別內部存取），對外只開放必要的 `public` 方法**，所有讀寫都得走方法這道門——門後就能設檢查哨。慣例上，讀取的方法叫 getter（`getXxx`），寫入的叫 setter（`setXxx`）：

範例：EncapsulationDemo.java

```
/**
 * 封裝示範：private 欄位加上有驗證的 setter。
 * 一個檔案可以放多個類別，但 public 類別只能有一個，
 * 且檔名必須跟 public 類別同名。
 */
class Player {
    private String name;
    private int hp;     // 血量：規定必須在 0 到 100 之間

    Player(String name, int hp) {
        this.name = name;
        setHp(hp);      // 建構子也走 setter，統一由一處把關
```

```
    }

    /** getter：對外提供唯讀窗口 */
    public int getHp() {
        return hp;
    }

    /** setter：寫入前先驗證，不合法的值一律擋下 */
    public void setHp(int hp) {
        if (hp < 0 || hp > 100) {
            System.out.println("血量 " + hp + " 不合法，維持原值");
            return;     // 衛述句：直接離開，不動欄位
        }
        this.hp = hp;
    }
}

public class EncapsulationDemo {
    public static void main(String[] args) {
        Player hero = new Player("勇者", 80);
        System.out.println("目前血量：" + hero.getHp());

        hero.setHp(150);    // 嘗試塞入不合法的值
        System.out.println("目前血量：" + hero.getHp());

        hero.setHp(65);     // 合法值正常寫入
        System.out.println("目前血量：" + hero.getHp());

        // hero.hp = 999;   // 編譯錯誤：hp 是 private，外界碰不到
    }
}
```

▶ 執行結果

```
目前血量：80
血量 150 不合法，維持原值
目前血量：80
目前血量：65
```

封裝的價值在最後一行註解：想直接繞過檢查改血量？編譯器直接說不。資料的每一次變動都必須經過你設下的關卡，「不合法的狀態根本無法存在」——這是大型系統可靠性的基石。

> **提示**　不是每個欄位都要反射性地配上 getter 與 setter。先問：外界真的需要讀它嗎？真的可以改它嗎？唯讀欄位只給 getter；不該被外界碰的連 getter 都不給。「開最小的門」是封裝的精髓。

8.5　static：屬於類別的成員

到目前為止的欄位都是「每個物件一份」。有些資料卻是全類別共用一份才合理——例如「總共建立了幾個物件」。在成員前加上 `static`，它就屬於類別本身，而不屬於任何個別物件：

範例：StaticDemo.java

```java
/**
 * static 欄位：全類別共用的計數器。
 */
class Visitor {
    static int count = 0;   // 靜態欄位：所有 Visitor 共用一份
    String name;            // 實體欄位：每個物件自己一份

    Visitor(String name) {
        this.name = name;
        count++;            // 每建立一個物件，共用計數器加一
    }
}

public class StaticDemo {
    public static void main(String[] args) {
        System.out.println("目前人數：" + Visitor.count);

        Visitor v1 = new Visitor("小明");
        Visitor v2 = new Visitor("小美");
        Visitor v3 = new Visitor("小華");

        // 靜態成員用「類別名稱.成員」存取，不需要任何物件
        System.out.println("目前人數：" + Visitor.count);
        System.out.println("v1 的名字：" + v1.name);
    }
}
```

▶ 執行結果

```
目前人數：0
目前人數：3
```

```
v1 的名字：小明
```

方法也可以是 static：不依賴任何物件狀態、純粹「輸入進、答案出」的工具方法最適合。你早就用過一整個類別的例子——Math：

範例：MathDemo.java

```java
/**
 * Math 類別：全部都是靜態方法，不必 new 直接用。
 */
public class MathDemo {
    public static void main(String[] args) {
        System.out.println(Math.max(3, 7));     // 較大值
        System.out.println(Math.abs(-5));       // 絕對值
        System.out.println(Math.sqrt(16));      // 平方根
        System.out.println(Math.pow(2, 10));    // 2 的 10 次方
    }
}
```

▶ 執行結果

```
7
5
4.0
1024.0
```

謎底揭曉時間：main 方法為什麼是 static？因為 JVM 啟動時，你的程式還沒有任何物件存在——JVM 必須能「不建立物件就呼叫 main」，static 正是這個意思。這也解釋了第 6 章所有方法都加 static 的原因：static 方法（如 main）只能直接呼叫其他 static 方法。

範例：StaticError.java

```java
/**
 * 靜態方法不能直接存取實體成員——示範正確的繞法。
 */
public class StaticError {
    String name = "小明";   // 實體欄位：屬於「某個物件」

    public static void main(String[] args) {
        // 直接寫 System.out.println(name); 會編譯錯誤：
        // non-static variable name cannot be referenced
        // from a static context
        // 因為 main 執行時根本沒有物件，哪來的 name？
```

```
12
13          // 正確做法：先建立物件，再透過物件存取
14          StaticError obj = new StaticError();
15          System.out.println(obj.name);
16      }
17  }
```

▶ 執行結果

```
小明
```

> **注意**　「non-static variable cannot be referenced from a static context」是初學者在 main 裡最常撞見的編譯錯誤。口訣：**靜態的世界裡沒有 this**——static 方法不知道「目前是哪個物件」，自然拿不到實體欄位。解法二選一：把該成員也改成 static（若它確實是全類別共用），或先 new 出物件再存取（若它屬於個別物件）。

8.6　套件與 import

類別一多就需要分類收納。「套件」（package）是 Java 的資料夾系統，慣例用公司網域倒過來命名（如 `com.example.school`），對應到實際的目錄結構。檔案第一行（註解之外）宣告所屬套件；使用其他套件的類別前，用 `import` 引入：

範例：HelloPackage.java

```
1  // 檔案第一行宣告套件：這個類別的全名是
2  // com.example.school.HelloPackage
3  package com.example.school;
4
5  // 引入其他套件的類別，之後就能用短名稱稱呼它
6  import java.util.Scanner;
7
8  public class HelloPackage {
9      public static void main(String[] args) {
10         // java.lang 套件（System、String、Math 等）自動引入，
11         // 所以我們從第 1 章用到現在都不需要 import 它們
12         System.out.println("我住在 com.example.school 套件");
13     }
14 }
```

套件宣告後，檔案必須放在對應的資料夾（`com/example/school/`）且執行時要用「全

名」。編譯時用 -d 讓 javac 自動建立目錄：

▶ 執行結果

```
$ javac -d out HelloPackage.java
$ java -cp out com.example.school.HelloPackage
我住在 com.example.school 套件
```

現在你也看懂了用了七章的咒語：`import java.util.Scanner;` 的意思是「引入 java.util 套件裡的 Scanner 類別」。初學階段的練習程式省略 package 宣告（落在「預設套件」）沒有問題，正式專案則一定會分套件。

8.7 綜合範例：BankAccount

初級篇的畢業考：設計一個銀行帳戶類別，整合本章全部觀念——private 欄位（封裝）、建構子驗證、this、static 計數器，以及具備完整驗證的存提款方法。金額用整數（元）而不是 double，呼應第 3 章的浮點數教訓：

範例：BankDemo.java

```
/**
 * 銀行帳戶：初級篇所學的總集合。
 */
class BankAccount {
    // ---- 靜態成員：所有帳戶共用 ----
    static int totalAccounts = 0;   // 總開戶數

    // ---- 實體欄位：每個帳戶各自一份，private 保護 ----
    private String owner;   // 戶名
    private int balance;    // 餘額（以元為單位，避開浮點誤差）

    /** 開戶：指定戶名與開戶金額（負數視為 0 處理） */
    BankAccount(String owner, int initialDeposit) {
        this.owner = owner;
        if (initialDeposit > 0) {
            this.balance = initialDeposit;
        }
        totalAccounts++;    // 每開一戶，共用計數器加一
    }

    /** 存款：金額必須是正數 */
    void deposit(int amount) {
        if (amount <= 0) {
```

```
            System.out.println("存款金額必須大於零！");
            return;          // 衛述句：不合法就提早離開
        }
        balance += amount;
        System.out.println(owner + " 存入 " + amount +
            "，餘額 " + balance);
    }

    /** 提款：金額須為正數，且不能超過餘額 */
    void withdraw(int amount) {
        if (amount <= 0) {
            System.out.println("提款金額必須大於零！");
            return;
        }
        if (amount > balance) {
            System.out.println(owner + " 餘額不足，無法提領 " +
                amount);
            return;
        }
        balance -= amount;
        System.out.println(owner + " 提領 " + amount +
            "，餘額 " + balance);
    }

    /** 只提供 getter：餘額不允許外界直接竄改 */
    int getBalance() {
        return balance;
    }

    /** 覆寫 toString：定義物件被 println 時顯示的文字 */
    @Override
    public String toString() {
        return "帳戶[" + owner + "，餘額 " + balance + " 元]";
    }
}

/**
 * 測試程式：開兩個帳戶，執行一連串存提款操作。
 */
public class BankDemo {
    public static void main(String[] args) {
        BankAccount a = new BankAccount("小明", 1000);
```

```
        BankAccount b = new BankAccount("小美", 500);

        a.deposit(300);     // 1000 + 300 = 1300
        a.withdraw(2000);   // 超過餘額，被擋下
        a.withdraw(800);    // 1300 - 800 = 500
        b.deposit(-100);    // 不合法金額，被擋下
        b.deposit(250);     // 500 + 250 = 750

        System.out.println("小明餘額：" + a.getBalance());
        System.out.println("小美餘額：" + b.getBalance());
        System.out.println("總開戶數：" + BankAccount.totalAccounts);

        // 把物件直接交給 println，會自動呼叫它的 toString
        System.out.println(a);
        System.out.println(b);
    }
}
```

▶ 執行結果

```
小明 存入 300，餘額 1300
小明 餘額不足，無法提領 2000
小明 提領 800，餘額 500
存款金額必須大於零！
小美 存入 250，餘額 750
小明餘額：500
小美餘額：750
總開戶數：2
帳戶[小明，餘額 500 元]
帳戶[小美，餘額 750 元]
```

請放慢速度把這支程式讀三遍——它是整冊教材的縮影：變數與型別（第 2 章）、條件判斷（第 4 章）、方法與參數（第 6 章）、字串輸出（第 7 章）、類別與封裝（本章）。兩個帳戶物件各自持有獨立的餘額，卻共用同一個 static 計數器；所有金額變動都必須通過驗證關卡。如果你能不看解說就理解每一行的用意，恭喜，初級篇的目標已經達成。

最後一個新面孔是 `toString` 方法：把物件交給 `println` 時，Java 會自動呼叫它取得物件的文字描述，不必再逐一印出欄位。方法上方的 `@Override` 標註表示「我正在覆寫繼承來的方法」——覆寫與繼承的完整故事，正是中級篇的開場主題。

8.8 本冊總結與下一步

回顧這趟旅程：你從安裝 JDK、印出第一行 Hello 開始，一路掌握了變數與型別、運算子、流程控制、陣列、方法、字串，最後親手設計了自己的類別。你已經具備用 Java 解決小型問題的完整能力。中級篇將站在「類別」這個地基上蓋出高樓：繼承與多型讓類別之間建立親屬關係、介面定義行為契約、例外處理讓程式優雅面對錯誤、集合框架與泛型取代長度固定的陣列，最後以檔案 I/O 與綜合專案收官。休息一下，我們中級篇見。

8.9 本章小結

- 物件導向把「資料」與「操作資料的行為」打包成物件，解決程序式風格資料零散的痛點。
- 類別是設計圖，物件（實體）是成品；`new` 在堆積配置物件，變數保存的是參考。
- 欄位有預設值（數值 0、boolean 為 false、參考為 null）；區域變數沒有，用前必須初始化。
- 建構子與類別同名、無回傳型別，在 new 時自動執行；可以多載；`this(...)` 呼叫其他建構子且必須在第一行。
- 自訂了任何建構子後，編譯器就不再贈送預設建構子。
- `this` 代表目前物件，用來區分同名的欄位與參數。
- 封裝：private 欄位加 public getter 與 setter，在 setter 設驗證關卡，讓不合法資料無法進入物件。
- static 成員屬於類別、全體共用，用「類別名. 成員」存取；靜態方法沒有 this，不能直接存取實體成員——main 是 static 的原因也在此。
- 套件是類別的資料夾，import 引入其他套件的類別；java.lang 自動引入。
- 對 null 呼叫方法會拋出 NullPointerException，訊息會點名誰是 null；覆寫 `toString` 可自訂物件被 println 時的文字描述。

8.10 本章練習

> **動手練習**
>
> 1. 觀念題：類別與物件的關係是什麼？在 `Student s = new Student();` 這行裡，哪個是類別？物件在哪裡？s 又是什麼？（提示：設計圖、成品、遙控器。）
> 2. 觀念題：為什麼欄位要宣告成 private？直接讓外界讀寫欄位會失去什麼？（提示：想想 setter 裡的驗證關卡。）

3. 實作題：設計 `Rectangle` 類別：private 的寬與高、帶參數的建構子、`area()` 與 `perimeter()` 方法，並在測試類別建立兩個矩形驗證。（提示：面積是寬乘高，周長是兩者和的兩倍。）
4. 實作題：替本章的 Student 類別加上建構子與 getter、setter，並在 setter 中把成績限制在 0 到 100 之間，超出範圍時印出警告且不修改。（提示：仿照 Player 的 setHp。）
5. 實作題：寫一個 `Counter` 類別，用 static 欄位統計「程式從頭到尾總共呼叫了幾次 `tick()` 方法」，在 main 建立多個 Counter 物件各呼叫幾次，驗證計數是全體共用的。（提示：static 欄位加 static getter。）
6. 挑戰題：擴充 BankAccount——加入轉帳方法 `transfer(BankAccount target, int amount)`，轉出前檢查餘額，轉帳成功與失敗都要印出明細。（提示：轉帳等於自己 withdraw 加上對方 deposit，但要先驗證再動手。）