THE COMPLETE JAVA PROGRAMMING COURSE

Java 程式設計完整學習教材

中級篇 INTERMEDIATE LEVEL

深入物件導向的封裝、繼承與多型，掌握介面、例外處理、集合框架、泛型與檔案 I/O，最後以一個完整的圖書管理系統專案整合所有知識。適合已掌握 Java 基礎語法的學習者。

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 封裝與繼承 **6**

1.1 封裝：把資料關進門裡 6

1.2 繼承：站在既有類別的肩膀上 9

1.3 super 關鍵字 13

1.4 方法覆寫與 @Override 14

1.5 建構子鏈與初始化順序 15

1.6 人人都是 Object 的後代 17

1.7 final：封印繼承與覆寫 20

1.8 本章小結 20

1.9 本章練習 21

2 多型、抽象類別與介面 **23**

2.1 多型：同一訊息，不同行為 23

2.2 向下轉型與 instanceof 26

2.3 抽象類別：留白的父類別 28

2.4 介面：能力的契約 30

2.5 綜合範例：付款系統 35

2.6 本章小結 37

2.7 本章練習 37

3 例外處理 **39**

3.1 為什麼需要例外 39

3.2 Throwable 家族 40

3.3 try-catch：接住例外 41

3.4 finally 與 try-with-resources 43

3.5 throw、throws 與自訂例外 45

3.6 例外鏈：保留因果 46

3.7 閱讀堆疊追蹤 47
3.8 例外處理最佳實務 48
3.9 綜合範例：ATM 提款機 49
3.10 本章小結 51
3.11 本章練習 51

4 集合框架 53
4.1 集合框架總覽 53
4.2 泛型語法與自動裝箱 54
4.3 List：有序、可重複 55
4.4 走訪集合的四種方式 57
4.5 Set：不重複的集合 60
4.6 Map：鍵值對 62
4.7 排序：Comparator 與 Collections 工具 64
4.8 集合選型指南 67
4.9 綜合範例：單字頻率統計 67
4.10 本章小結 69
4.11 本章練習 70

5 泛型 71
5.1 沒有泛型的年代 71
5.2 泛型類別 72
5.3 泛型方法 74
5.4 有界型別參數 75
5.5 萬用字元 76
5.6 泛型與繼承：不變性 79
5.7 型別抹除 80
5.8 綜合範例：型別安全的泛型堆疊 81
5.9 本章小結 83
5.10 本章練習 83

6 檔案與 I/O 85
6.1 檔案的中繼資料：File 與 Path 85

6.2 I/O 流的分類 86
6.3 位元組流與緩衝 87
6.4 字元流與編碼 88
6.5 NIO.2 的 Files 工具箱 90
6.6 物件序列化 92
6.7 綜合範例：CSV 成績檔處理 94
6.8 本章小結 97
6.9 本章練習 98

7 列舉、內部類別與常用 API 99

7.1 enum 基礎：告別整數常數 99
7.2 帶欄位與方法的 enum 100
7.3 巢狀類別四人組 104
7.4 匿名類別實作 Comparator 106
7.5 Math 與 Random 107
7.6 日期時間：java.time 109
7.7 綜合範例：訂單狀態機 111
7.8 本章小結 114
7.9 本章練習 114

8 綜合實戰——圖書管理系統 116

8.1 需求分析 116
8.2 系統設計 117
8.3 model 層：三個資料類別 117
8.4 exception 層：讓錯誤會說話 122
8.5 repository 層：可抽換的儲存 123
8.6 service 層：商業規則的權威 127
8.7 app 層：選單迴圈 129
8.8 完整執行示範 132
8.9 知識點回顧與延伸挑戰 134
8.10 本章小結 134
8.11 本章練習 135

第 1 章

封裝與繼承

本章學習目標

學完本章你將能夠：

- 以 private 欄位搭配 getter 與 setter 實作完整的封裝，並在 setter 中驗證資料。
- 分辨四種存取修飾詞的可見範圍，說明欄位為什麼應該宣告為 private。
- 用 extends 建立繼承關係，判斷 is-a 與 has-a，在繼承與組合之間做正確選擇。
- 掌握 super 的兩種用途：呼叫父類別建構子與呼叫父類別方法。
- 正確覆寫方法、理解 @Override 標註的保護作用，並完整說出物件的初始化順序。
- 依照契約正確覆寫 Object 類別的 toString、equals 與 hashCode。

在初級篇裡，你已經會定義類別、建立物件、撰寫建構子與方法。本章要把物件導向真正的威力釋放出來：**封裝**（encapsulation）讓每個物件為自己的資料把關，使「不合法的狀態」根本不可能出現；**繼承**（inheritance）讓新類別站在既有類別的肩膀上，重複利用並擴充其行為。這兩個概念是物件導向三大支柱的前兩根，第三根「多型」將在下一章登場。

本章的範例圍繞兩條軸線：銀行帳戶（示範封裝與資料驗證）以及員工與經理（示範繼承、覆寫與 super）。請務必親手輸入並執行每一個範例——封裝與繼承的許多細節，唯有看到編譯器報錯、看到輸出順序，才會真正內化。

1.1 封裝：把資料關進門裡

1.1.1 完整的封裝實踐

封裝的公式很簡單：**欄位一律 private，存取一律走公開方法**。真正的重點在於：setter 與其他修改狀態的方法，必須在資料進門之前完成驗證。以下是一個完整的示範：

範例：BankAccount.java

```
/**
 * 銀行帳戶：封裝的完整示範。
 * 欄位全部 private，外界只能透過公開方法存取，
 * 每一次修改都經過驗證，守住「餘額永不為負」這條不變條件。
 */
```

```java
public class BankAccount {

    private String owner;    // 戶名：private，外界無法直接竄改
    private int balance;     // 餘額（元）：不變條件是 balance >= 0

    /** 建構子：物件誕生的第一刻就必須是合法狀態 */
    public BankAccount(String owner, int initialBalance) {
        setOwner(owner);          // 重複利用 setter 裡的驗證邏輯
        deposit(initialBalance);  // 初始金額視為第一筆存款來驗證
    }

    /** getter：唯讀出口，外界只能「看」不能「改」 */
    public String getOwner() {
        return owner;
    }

    /** setter：資料進門前先驗證，不合法就拒收 */
    public void setOwner(String owner) {
        // 驗證失敗就拋出例外，讓錯誤在第一時間現形（詳見第 3 章）
        if (owner == null || owner.isBlank()) {
            throw new IllegalArgumentException("戶名不可為空");
        }
        this.owner = owner;       // this.owner 是欄位，owner 是參數
    }

    /** 注意：刻意不提供 setBalance！餘額只能透過存提款改變 */
    public int getBalance() {
        return balance;
    }

    /** 存款：金額必須為非負數 */
    public void deposit(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("存款金額不可為負");
        }
        balance += amount;        // 通過驗證才允許修改欄位
    }

    /** 提款：金額必須為正，且不可超過餘額 */
    public void withdraw(int amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("提款金額必須為正");
```

```
        }
        if (amount > balance) {   // 守住不變條件的最後防線
            throw new IllegalArgumentException("餘額不足");
        }
        balance -= amount;
    }
}
```

注意幾個設計決策。第一，`balance` 沒有對應的 setter：不是每個欄位都需要成對的 getter 與 setter，「餘額」在語意上只能透過存款與提款改變，直接開放 `setBalance` 反而是在破壞封裝。第二，建構子重複利用 `setOwner` 與 `deposit` 的驗證邏輯，確保「從物件誕生的第一刻起」所有規則就已生效。用一個測試程式觀察封裝的效果：

範例：BankAccountDemo.java

```
/** 封裝的效果：非法操作進不了門，物件狀態永遠合法 */
public class BankAccountDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("小明", 1000);

        account.deposit(500);        // 合法：存入 500 元
        account.withdraw(300);       // 合法：提出 300 元
        System.out.println(account.getOwner() + "的餘額："
                + account.getBalance() + " 元");

        // account.balance = -999;   // 編譯錯誤：balance 是 private

        try {
            account.withdraw(9999);  // 非法：超過餘額
        } catch (IllegalArgumentException e) {
            // 物件拒絕進入非法狀態，餘額毫髮無傷
            System.out.println("提款失敗：" + e.getMessage());
        }
        System.out.println("最終餘額：" + account.getBalance() + " 元");
    }
}
```

▶ 執行結果

```
小明的餘額：1200 元
提款失敗：餘額不足
最終餘額：1200 元
```

被註解掉的那一行 `account.balance = -999;` 若取消註解，編譯器會直接報錯：`balance` 在 `BankAccountDemo` 中不可見。這正是我們要的效果——**錯誤越早被抓到越好**，編譯期就擋下，勝過執行期才爆炸。

1.1.2 不變條件：封裝真正守護的東西

> **說明**　「餘額永遠大於等於零」這類**在物件整個生命週期中恆真的規則**，稱為不變條件（invariant）。封裝的本質不是「把欄位藏起來」這個動作，而是藉由收斂所有修改入口，讓不變條件「無法被破壞」。如果欄位是 public，任何一行程式都可能把它改成非法值，要找出誰改壞的，得翻遍整個專案；欄位是 private 時，嫌疑犯只剩下類別內的少數幾個方法。這就是為什麼欄位應該一律 private——不是教條，而是把除錯範圍從「全世界」縮小到「一個檔案」。

1.1.3 四種存取修飾詞

Java 提供四個層級的存取控制。沒有寫任何修飾詞時稱為「預設存取」（也叫套件私有，package-private）：

修飾詞	同類別	同套件	不同套件的子類別	任何地方
`private`	可	不可	不可	不可
（預設）	可	可	不可	不可
`protected`	可	可	可	不可
`public`	可	可	可	可

實務上的選擇原則：欄位用 `private`；提供給外界的行為用 `public`；打算讓子類別使用或覆寫、但不想對外公開的方法用 `protected`；預設存取則適合「只在同套件內部協作」的輔助類別與方法。原則是**從最嚴格的開始給**，日後放寬容易，收緊卻可能牽動大量既有程式。

1.2 繼承：站在既有類別的肩膀上

1.2.1 extends 與 is-a 關係

繼承用 `extends` 關鍵字宣告。子類別（subclass）自動獲得父類別（superclass）的非 private 成員，並可以新增自己的欄位與方法、覆寫父類別的方法。判斷該不該繼承，最好的試金石是 **is-a 句型**：「經理是一種員工」說得通，所以 `Manager extends Employee` 合理；「汽車是一種引擎」說不通，所以汽車不該繼承引擎。

範例：Employee.java

```
1 /**
```

```
 * 員工：繼承體系的父類別。
 * 把所有員工共同的狀態與行為集中在這裡，讓子類別重複利用。
 */
public class Employee {

    private String name;    // 姓名：private，連子類別都不能直接存取
    private int salary;     // 月薪（元）

    /** 建構子：初始化所有員工共同的欄位 */
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public int getSalary() {
        return salary;
    }

    /** 年薪：預設演算法是月薪乘以 12 */
    public int annualPay() {
        return salary * 12;
    }

    /** 自我介紹：設計上預期會被子類別覆寫 */
    public String describe() {
        return "員工" + name + "，月薪 " + salary + " 元";
    }
}
```

範例：Manager.java

```
/**
 * 經理：繼承 Employee，額外持有「管理加給」。
 * Manager is-a Employee：任何需要員工的地方都能接受經理。
 */
public class Manager extends Employee {

    private int bonus;      // 每月管理加給：Manager 特有的欄位
```

```

    /** 建構子：先請父類別初始化共同欄位，再處理自己的欄位 */
    public Manager(String name, int salary, int bonus) {
        super(name, salary);   // 必須是第一行：先蓋地基，再蓋樓
        this.bonus = bonus;
    }

    /** 覆寫年薪：在父類別的計算結果上追加管理加給 */
    @Override
    public int annualPay() {
        // super.annualPay()：重複利用父類別的公式，避免複製貼上
        return super.annualPay() + bonus * 12;
    }

    /** 覆寫自我介紹：沿用父類別的描述再補充自己的資訊 */
    @Override
    public String describe() {
        return super.describe() + "，加給 " + bonus + " 元";
    }
}
```

範例：InheritanceDemo.java

```
/** 子類別自動擁有父類別的公開成員 */
public class InheritanceDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("小華", 40000);
        Manager mgr = new Manager("小美", 60000, 15000);

        // getName 定義在 Employee；Manager 一行都沒寫就直接能用
        System.out.println(emp.describe());
        System.out.println(mgr.describe());
        System.out.println(mgr.getName() + "的年薪："
                + mgr.annualPay() + " 元");
    }
}
```

▶ 執行結果

```
員工小華，月薪 40000 元
員工小美，月薪 60000 元，加給 15000 元
小美的年薪：900000 元
```

追蹤一下第三行輸出：`mgr.annualPay()` 執行的是 `Manager` 覆寫的版本，它先呼叫 `super.annualPay()` 取得 $60000 \times 12 = 720000$，再加上 $15000 \times 12 = 180000$，總計 900000 元。輸出與推導完全一致。

1.2.2 子類別到底繼承了什麼

一個常見的誤解是「private 欄位不會被繼承」。精確的說法是：子類別的物件確實含有父類別的 private 欄位（`Manager` 物件的記憶體中確實存著 `name` 與 `salary`），否則 `getName()` 不可能運作；但子類別的程式碼不能直接存取這些欄位，必須透過父類別提供的非 private 方法。也就是說：private 欄位是「有繼承、沒門路」，這正是封裝跨越繼承關係依然有效的證明。另外兩件不被繼承的東西是建構子（只能用 `super` 呼叫，不能被當成自己的建構子）與 static 成員的「覆寫資格」（static 方法只有遮蔽、沒有覆寫）。

1.2.3 繼承還是組合

繼承不是重複利用程式碼的唯一手段。另一條路是組合（composition）：把既有類別的物件當成欄位持有，將工作「委派」給它，兩者的關係是 has-a（擁有）：

範例：Car.java 組合而非繼承

```
/** 引擎：一個獨立的元件類別 */
class Engine {
    /** 發動引擎 */
    public void start() {
        System.out.println("引擎發動");
    }
}

/** 汽車「擁有」引擎（has-a）：用組合，不用繼承 */
public class Car {
    // 把引擎當成欄位持有：Car 的能力建立在 Engine 之上
    private final Engine engine = new Engine();

    /** 開車：把「發動」這件事委派給引擎物件 */
    public void drive() {
        engine.start();
        System.out.println("汽車出發");
    }
}
```

> **提示**　繼承與組合的選擇準則：先問 is-a 還是 has-a。「經理是一種員工」用繼承；「汽車擁有引擎」用組合。若兩者都說得通，**優先選擇組合**：繼承會把父類別的實作細節暴露給子類別（耦合較緊），父類別一改，所有子類別都可能被波及；組合只依賴對方的公開介面，關係鬆、容易替換。業界流傳的口訣是「多用組合，少用繼承」——繼承保留給真正穩定的 is-a 階層。

1.3　super 關鍵字

`super` 有兩種用途。第一種是**呼叫父類別建構子**：`super(引數)` 必須寫在子類別建構子的第一行，因為子類別的欄位初始化必須建立在「父類別部分已經初始化完成」的地基上。若你沒有寫，編譯器會自動插入無參數的 `super()`；一旦父類別沒有無參數建構子（例如上面的 `Employee` 與 `BankAccount`），你就必須親手呼叫，否則編譯失敗。第二種用途是**呼叫父類別方法**：`super.方法名()` 可以在覆寫版本中重複利用父類別的邏輯，`Manager.describe()` 就是典型用法。

範例：SavingsAccount.java

```
/** 定存帳戶：示範 super 與「繼承欄位但不能直接存取」 */
public class SavingsAccount extends BankAccount {

    private final int ratePercent;   // 年利率（整數百分比）

    public SavingsAccount(String owner, int balance, int ratePercent) {
        // 父類別沒有無參數建構子，這裡必須明確呼叫 super(...)，
        // 而且它必須是建構子的第一行敘述
        super(owner, balance);
        this.ratePercent = ratePercent;
    }

    /** 一年後的預估餘額（本金加單利） */
    public int projectedBalance() {
        // 不能直接寫 balance：它是父類別的 private 欄位，
        // 子類別「擁有」它但「碰不到」，必須走公開的 getter
        return getBalance() + getBalance() * ratePercent / 100;
    }

    public static void main(String[] args) {
        SavingsAccount savings = new SavingsAccount("小美", 10000, 2);
        System.out.println("戶名：" + savings.getOwner());
        System.out.println("一年後預估："
```

```
                + savings.projectedBalance() + " 元");
    }
}
```

▶ 執行結果

```
戶名：小美
一年後預估：10200 元
```

1.4 方法覆寫與 @Override

覆寫（override）是子類別以**完全相同的方法簽章**（方法名稱與參數列）重新實作父類別的方法。合法覆寫的規則如下：方法名稱與參數列必須一致；回傳型別必須相同（或是父類別回傳型別的子型別，稱為協變回傳）；存取層級只能放寬不能收窄（父類別是 `public`，子類別不能改成 `protected`）；不能覆寫 `private`、`static` 與 `final` 方法。注意覆寫（override）與多載（overload）是兩回事：多載是「同名不同參數列」的多個方法並存，發生在同一個類別內也可以。

規則不少，但你不需要背——`@Override` 標註會替你檢查。看看沒有它會發生什麼事：

範例：OverrideTrap.java

```
/** 動物父類別 */
class Animal {
    /** 發出叫聲：設計上預期被子類別覆寫 */
    public void makeSound() {
        System.out.println("（某種動物的聲音）");
    }
}

/** 貓：本想覆寫 makeSound，卻把方法名拼錯了 */
class Cat extends Animal {
    /** 少了大寫 S！這不是覆寫，而是「新增」了一個方法 */
    public void makesound() {
        System.out.println("喵！");
    }
}

/** 觀察拼錯方法名的後果 */
public class OverrideTrap {
    public static void main(String[] args) {
        Cat cat = new Cat();
```

```
        cat.makeSound();    // 執行繼承來的父類別版本：覆寫根本沒發生
        cat.makesound();    // 拼錯字的那個「新方法」還健在
    }
}
```

▶ 執行結果

```
（某種動物的聲音）
喵！
```

程式照常編譯、照常執行，只是行為完全不是你要的——這種「安靜的錯誤」最難抓。加上 `@Override` 之後，編譯器會驗證「這個方法真的覆寫了父類別的某個方法」，拼錯字立刻現形：

範例：加上 Override 標註讓錯誤現形

```
class Cat extends Animal {
    @Override                    // 明確宣告：我就是要覆寫
    public void makesound() {    // 編譯器立刻抓到拼字錯誤
        System.out.println("喵！");
    }
}
```

▶ 執行結果

```
Cat.java:2: error: method does not override or implement a method from a supertype
    @Override
    ^
1 error
```

> **注意**　拼錯方法名稱（或寫錯參數型別）時，Java 不會報錯，而是默默把它當成新方法或多載，父類別版本原封不動地被繼承下來，程式的行為悄悄跑偏。防禦方法只有一個：每一個覆寫方法都加上 **`@Override`**，沒有例外。這行標註零成本，卻能把一類極難除錯的問題轉成一眼可見的編譯錯誤。

1.5　建構子鏈與初始化順序

建立一個子類別物件時，初始化像接力賽一樣層層向上又層層落下。完整順序是：(1) 父類別靜態初始化區塊（類別第一次載入時，僅一次）；(2) 子類別靜態初始化區塊（僅一次）；(3) 父類別實體初始化區塊與欄位初始器；(4) 父類別建構子本體；(5) 子類別實體初始化區塊與

欄位初始器；(6) 子類別建構子本體。用程式驗證：

範例：InitOrder.java

```
/** 父類別：每個初始化階段都印一行，讓執行順序現形 */
class Parent {
    // 靜態初始化區塊：類別第一次被載入時執行，而且只執行一次
    static {
        System.out.println("1. Parent 靜態初始化區塊");
    }

    // 實體初始化區塊：每次 new 都會在建構子本體之前執行
    {
        System.out.println("3. Parent 實體初始化區塊");
    }

    /** 父類別建構子 */
    Parent() {
        System.out.println("4. Parent 建構子");
    }
}

/** 子類別 */
class Child extends Parent {
    static {
        System.out.println("2. Child 靜態初始化區塊");
    }

    {
        System.out.println("5. Child 實體初始化區塊");
    }

    /** 子類別建構子：第一行隱含 super() */
    Child() {
        // 編譯器自動插入 super()：先完成父類別的初始化
        System.out.println("6. Child 建構子");
    }
}

/** 驗證完整的初始化順序 */
public class InitOrder {
    public static void main(String[] args) {
        System.out.println("=== 第一次 new Child() ===");
```

```
        new Child();
        System.out.println("=== 第二次 new Child() ===");
        new Child();   // 靜態區塊不會再執行
    }
}
```

▶ 執行結果

```
=== 第一次 new Child() ===
1. Parent 靜態初始化區塊
2. Child 靜態初始化區塊
3. Parent 實體初始化區塊
4. Parent 建構子
5. Child 實體初始化區塊
6. Child 建構子
=== 第二次 new Child() ===
3. Parent 實體初始化區塊
4. Parent 建構子
5. Child 實體初始化區塊
6. Child 建構子
```

觀察兩個關鍵：靜態區塊屬於「類別」，只在類別載入時執行一次，第二次 `new` 就不再出現；實體層級的初始化一定是「父類別先完成、子類別才開始」，這條建構子鏈（constructor chaining）一路追溯到所有類別的始祖——`Object`。

1.6　人人都是 Object 的後代

沒有寫 `extends` 的類別，一律隱含繼承 `java.lang.Object`。因此 `Object` 定義的方法人人都有，其中三個最值得正確覆寫：`toString`（物件的文字描述）、`equals`（邏輯相等的定義）與 `hashCode`（雜湊碼，供雜湊集合使用）。

1.6.1　toString、equals 與 hashCode 的正確覆寫

預設的 `toString` 回傳「類別名加雜湊碼」（例如 `Book@1b6d3586`），對人類毫無意義；預設的 `equals` 等同於 `==`，只認「同一個物件」。以下是三者一起正確覆寫的標準範本：

範例：Book.java

```
import java.util.Objects;

/**
 * 書籍：示範 toString、equals 與 hashCode 的正確覆寫。
```

```
 * 以 ISBN 作為「邏輯身分」：ISBN 相同就視為同一本書。
 */
public class Book {

    private final String isbn;    // 國際標準書號：身分依據
    private final String title;   // 書名：只是描述性資料

    public Book(String isbn, String title) {
        this.isbn = isbn;
        this.title = title;
    }

    /** 覆寫 toString：println 會自動呼叫，回傳有意義的描述 */
    @Override
    public String toString() {
        return "Book{isbn=" + isbn + ", title=" + title + "}";
    }

    /**
     * 覆寫 equals：定義「邏輯相等」。
     * 標準流程：同一物件快速通過；null 或型別不符直接否定；
     * 最後才比較身分欄位。
     */
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {                // 1. 自己跟自己一定相等
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;                 // 2. null 與不同類別都不相等
        }
        Book other = (Book) obj;          // 3. 檢查過了，安全向下轉型
        return isbn.equals(other.isbn);   // 4. 只以 ISBN 判斷身分
    }

    /** 覆寫 hashCode：equals 相等的物件必須有相同的雜湊碼 */
    @Override
    public int hashCode() {
        return Objects.hash(isbn);        // 與 equals 使用同一組欄位
    }
}
```

這份範本蘊含 equals 契約的幾個要求：**自反性**（x.equals(x) 恆為 true，由第 1 步保證）、**對稱性**（x.equals(y) 與 y.equals(x) 結果一致，用 getClass() 比較型別可確保這點；若改用 instanceof，則允許子類別物件參與比較，但必須小心子類別再加欄位時對稱性會破裂）、**null 安全**（x.equals(null) 必須回傳 false 而非拋例外，由第 2 步保證）。參數型別必須是 Object——寫成 equals(Book obj) 就變成多載，集合類別呼叫的仍是預設版本，這也是務必加 @Override 的原因之一。

範例：BookDemo.java

```
/** 驗證 toString 與 equals、hashCode 契約 */
public class BookDemo {
    public static void main(String[] args) {
        Book a = new Book("978-0134685991", "Effective Java");
        Book b = new Book("978-0134685991", "Effective Java");

        // println 自動呼叫 toString()
        System.out.println(a);

        // == 比較「參考」：a 與 b 是兩個不同的物件
        System.out.println("a == b：" + (a == b));

        // equals 比較「邏輯身分」：ISBN 相同即視為同一本書
        System.out.println("a.equals(b)：" + a.equals(b));

        // 契約檢查：equals 相等，hashCode 也必須相等
        System.out.println("雜湊碼相同：" + (a.hashCode() == b.hashCode()));
    }
}
```

▶ 執行結果

```
Book{isbn=978-0134685991, title=Effective Java}
a == b：false
a.equals(b)：true
雜湊碼相同：true
```

> **注意**　hashCode 與 equals 之間有一條鐵的契約：**equals 判定相等的兩個物件，hashCode 必須回傳相同的值**。只覆寫 equals 而不覆寫 hashCode，兩個「邏輯上相等」的物件會落在雜湊集合的不同格子裡——HashSet 去不了重、HashMap 用等值的鍵查不到資料，而且這種錯誤時隱時現，極難除錯（第 4 章會親眼見到後果）。口訣：**兩者永遠成對**

> 覆寫，且使用同一組欄位。

1.7 final：封印繼承與覆寫

`final` 用在類別上表示「禁止繼承」，用在方法上表示「允許繼承類別，但這個方法不准覆寫」。標準函式庫中 `String` 與 `Integer` 都是 final 類別，確保它們的不可變性質不會被子類別破壞：

範例：final 類別與 final 方法

```
/** final 類別：禁止被繼承，確保行為不會被子類別竄改 */
public final class AuthToken {

    private final String value;   // final 欄位：一旦設定即不可變

    public AuthToken(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}

/** 允許繼承，但封印關鍵方法 */
class PaymentBase {
    /** final 方法：稽核邏輯攸關安全，不准子類別改寫 */
    public final void audit() {
        System.out.println("寫入稽核紀錄");
    }
}
```

設計繼承體系是一種承諾：父類別的每個可覆寫方法，都是留給未來子類別的擴充點。若一個類別當初不是為了被繼承而設計，宣告成 `final` 反而是負責任的做法——這也是「多用組合、少用繼承」精神的延伸。

1.8 本章小結

- 封裝的公式：欄位一律 `private`，修改一律經過會驗證的公開方法；其目的在守護不變條件，把除錯範圍從全專案縮小到單一類別。
- 不是每個欄位都需要 setter；像餘額這類狀態，應該只透過語意明確的操作（存款、提

款）改變。

- 四種存取修飾詞由嚴到寬：`private`、預設（同套件）、`protected`、`public`；原則是從最嚴格的開始給。
- 繼承表達 is-a 關係；組合表達 has-a 關係；兩者都說得通時優先用組合，降低耦合。
- 子類別物件含有父類別的 private 欄位，但不能直接存取，必須透過父類別的公開方法。
- `super(...)` 呼叫父類別建構子且必須是第一行；`super.方法()` 可在覆寫版本中重複利用父類別邏輯。
- 每個覆寫方法都要加 `@Override`，把拼錯名稱、寫錯參數的隱形錯誤轉成編譯錯誤。
- 初始化順序：父靜態、子靜態（各僅一次）、父實體區塊、父建構子、子實體區塊、子建構子。
- `toString` 讓輸出可讀；`equals` 定義邏輯相等且參數必須是 `Object`；`equals` 與 `hashCode` 永遠成對覆寫並使用同一組欄位。
- `final` 類別禁止繼承、`final` 方法禁止覆寫；不為繼承而設計的類別，宣告 final 是負責任的做法。

1.9 本章練習

動手練習

1. 觀念題：為什麼「把欄位宣告成 public，但大家約定好不要亂改」不是可行的封裝策略？請從不變條件與除錯成本兩個角度說明。（提示：回想 `balance` 若被任意一行程式改成負數，要如何找出兇手。）
2. 實作題：設計 `Temperature` 類別，內含攝氏溫度欄位，setter 拒絕低於絕對零度（攝氏負 273.15 度）的值；另提供 `toFahrenheit()` 方法。撰寫測試程式驗證合法與非法輸入。（提示：非法輸入時拋出 `IllegalArgumentException`。）
3. 實作題：建立 `Vehicle` 父類別（欄位：廠牌、時速上限）與 `ElectricCar` 子類別（新增欄位：電池容量），子類別建構子必須用 `super` 初始化共同欄位，並覆寫 `describe()` 方法沿用父類別描述再補充電池資訊。（提示：模仿本章 `Employee` 與 `Manager` 的寫法。）
4. 觀念題：下列類別的 `main` 會印出什麼？先在紙上推導再執行驗證：父類別與子類別各有一個靜態區塊、一個實體區塊與一個建構子，且 `main` 中連續 `new` 了兩次子類別。（提示：靜態區塊只執行一次。）
5. 實作題：為 `Student` 類別（欄位：學號、姓名）以學號為身分依據覆寫 `equals` 與 `hashCode`，並寫程式驗證：兩個學號相同、姓名不同的物件 `equals` 為 `true` 且雜湊碼相同。（提示：使用 `Objects.hash` 與 `getClass()` 檢查。）

6. 挑戰題：把 `BankAccount` 改用組合的方式重新設計一個 `JointAccount`（聯名帳戶，內含兩個 `BankAccount`），體會「不透過繼承也能重複利用」的感覺。（提示：提款時先檢查兩個帳戶的餘額總和。）

第 2 章

多型、抽象類別與介面

本章學習目標

學完本章你將能夠：

- 解釋多型的核心概念：同一訊息、不同行為，並說明動態繫結的運作方式。
- 分辨「參考型別決定可呼叫的方法、實際型別決定執行哪個版本」，並知道欄位不具多型。
- 安全地向下轉型：使用 instanceof 檢查與 Java 16 的模式匹配寫法。
- 設計抽象類別與抽象方法，用多型陣列統一處理一族物件。
- 掌握介面的完整能力：常數、抽象方法、default、static 與 private 方法，並解決鑽石衝突。
- 依照準則在抽象類別與介面之間做出正確選擇。

上一章的繼承讓 `Manager` 重複利用了 `Employee` 的程式碼，但繼承真正的殺手級應用不是省程式碼，而是本章的主角——**多型**（polymorphism）：讓同一個呼叫式，在不同物件身上展現不同行為。有了多型，新增一種員工、一種形狀、一種付款方式時，既有的處理邏輯一行都不用改。這是物件導向「對修改封閉、對擴充開放」精神的技術基礎。

本章由多型出發，依序引入兩種「刻意不完整」的型別：抽象類別（把共同骨架留下、把差異交給子類別）與介面（純粹的能力契約），最後以一個付款系統收尾，為進階篇的策略模式鋪路。

2.1 多型：同一訊息，不同行為

2.1.1 向上轉型與動態繫結

把子類別物件指派給父類別參考，稱為**向上轉型**（upcasting）。因為子類別物件「是一種」父類別物件，這個動作永遠安全、不需要任何轉型語法。魔法發生在呼叫方法的那一刻：

範例：PolymorphismDemo.java

```
/** 動物父類別：定義共同的訊息介面 */
class Animal {
    /** 發出叫聲：每種動物有自己的版本 */
```

```
    public void speak() {
        System.out.println("（動物的聲音）");
    }
}

/** 狗 */
class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("汪汪！");
    }

    /** Dog 特有的方法：用 Animal 參考時「看不到」它 */
    public void fetch() {
        System.out.println("狗去撿球");
    }
}

/** 貓 */
class Cat extends Animal {
    @Override
    public void speak() {
        System.out.println("喵～");
    }
}

/** 多型：同一訊息，不同行為 */
public class PolymorphismDemo {
    public static void main(String[] args) {
        // 向上轉型：把子類別物件交給父類別參考，永遠安全
        Animal a1 = new Dog();
        Animal a2 = new Cat();

        a1.speak();   // 動態繫結：執行 Dog 的版本
        a2.speak();   // 動態繫結：執行 Cat 的版本

        // a1.fetch();   // 編譯錯誤：Animal 型別沒有宣告 fetch()

        // 多型陣列：不同子類別的物件放進同一個陣列統一處理
        Animal[] zoo = { new Dog(), new Cat(), new Dog() };
        for (Animal animal : zoo) {
            animal.speak();   // 每個元素各自執行自己的覆寫版本
```

```
        }
    }
}
```

▶ 執行結果

```
汪汪！
喵～
汪汪！
喵～
汪汪！
```

這裡有一組必須分清楚的概念。`a1` 的**參考型別**（宣告型別）是 `Animal`，**實際型別**（物件真正的類別）是 `Dog`。兩者分工如下：

- **參考型別決定「能呼叫什麼」**：編譯器只看宣告型別，`Animal` 沒有宣告 `fetch()`，所以 `a1.fetch()` 直接編譯失敗——即使物件其實是 `Dog`。
- **實際型別決定「執行哪個版本」**：執行期 JVM 依照物件真正的類別找出覆寫版本，這個機制稱為**動態繫結**（dynamic binding）或晚期繫結。

多型陣列的迴圈是這個機制的經典應用：迴圈本體只寫一次 `animal.speak()`，每一輪執行的版本卻各不相同。日後新增 `Bird` 類別，這段迴圈一個字都不用改。

2.1.2 欄位不具多型

動態繫結只發生在**實體方法**上。欄位（以及 static 方法）採用**靜態繫結**：編譯期就依照參考型別決定。若子類別宣告了與父類別同名的欄位，兩個欄位會同時存在（稱為欄位遮蔽），存取哪一個由參考型別決定：

範例：FieldBinding.java

```
/** 父類別：一個欄位、一個方法 */
class Base {
    String tag = "Base 的欄位";

    String describe() {
        return "Base 的方法";
    }
}

/** 子類別：宣告同名欄位（遮蔽），並覆寫方法 */
class Derived extends Base {
```

```
    String tag = "Derived 的欄位";   // 欄位遮蔽：兩個欄位同時存在

    @Override
    String describe() {
        return "Derived 的方法";
    }
}

/** 方法看「實際型別」，欄位看「參考型別」 */
public class FieldBinding {
    public static void main(String[] args) {
        Base obj = new Derived();          // 向上轉型

        // 方法：動態繫結，執行 Derived 覆寫的版本
        System.out.println(obj.describe());

        // 欄位：靜態繫結，取到 Base 宣告的那一份
        System.out.println(obj.tag);
    }
}
```

▶ 執行結果

```
Derived 的方法
Base 的欄位
```

> **說明**　為什麼欄位不做動態繫結？欄位存取只是讀一塊記憶體，沒有「行為」可言，JVM 若對每次欄位存取都做執行期查找，效能損失巨大且沒有語意上的收益。實務結論有二：第一，**永遠不要在子類別宣告與父類別同名的欄位**，遮蔽只會製造混亂；第二，欄位維持 private 並透過方法存取（第 1 章的封裝），自然就把「欄位無多型」的問題隔絕在外——因為方法是有多型的。

2.2　向下轉型與 instanceof

向上轉型之後，若需要呼叫子類別特有的方法，就得**向下轉型**（downcasting）回子類別型別。向下轉型必須寫明強制轉型語法，而且有風險：物件的實際型別若不符，執行期會拋出 `ClassCastException`。正確做法是先用 `instanceof` 檢查。Java 16 起的**模式匹配 instanceof** 更把「檢查、轉型、命名變數」三步併成一步：

範例：CastingDemo.java

```java
/** 向下轉型與 instanceof 檢查（沿用 Animal、Dog、Cat） */
public class CastingDemo {
    public static void main(String[] args) {
        Animal pet = new Dog();      // 實際型別是 Dog

        // 錯誤示範（已註解）：轉成不相干的型別，執行期直接爆炸
        // Cat cat = (Cat) pet;      // 拋出 ClassCastException

        // 傳統寫法：先用 instanceof 檢查，確認後再轉型
        if (pet instanceof Dog) {
            Dog dog = (Dog) pet;     // 檢查過了，安全
            dog.fetch();
        }

        // 模式匹配 instanceof（Java 16+）：檢查、轉型、命名一氣呵成
        if (pet instanceof Dog dog) {
            dog.fetch();             // dog 已經是 Dog 型別，直接可用
        }

        // 條件不成立時，模式變數不會生效，區塊不執行
        if (pet instanceof Cat cat) {
            cat.speak();
        } else {
            System.out.println("pet 不是貓，轉型分支未執行");
        }
    }
}
```

▶ 執行結果

```
狗去撿球
狗去撿球
pet 不是貓，轉型分支未執行
```

一個設計上的提醒：如果你的程式充滿「`instanceof` 加向下轉型」的分支，通常代表多型沒有用好——與其在外面問「你是什麼型別」再分別處理，不如把差異化的行為定義成可覆寫的方法，讓每個類別自己處理。向下轉型應該是偶一為之的工具，不是日常結構。

2.3 抽象類別：留白的父類別

有時父類別的某個方法「一定存在，但無法有合理的預設實作」。例如每種形狀都能算面積，但「形狀」本身算不出面積——公式只有具體的圓形、矩形才知道。這時把方法宣告成 `abstract`（只有簽章、沒有本體），並把類別也標成 `abstract`：

範例：Shape.java 與其子類別

```
/**
 * 抽象類別 Shape：所有形狀的共同輪廓。
 * area() 是抽象方法：宣告「每種形狀都會算面積」，但把公式留給子類別。
 */
abstract class Shape {

    private final String name;   // 共同狀態放在父類別

    protected Shape(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    /** 抽象方法：沒有本體，以分號結束 */
    public abstract double area();

    /** 一般方法可以呼叫抽象方法：執行期動態繫結到子類別實作 */
    public String report() {
        return name + "面積 = " + area();
    }
}

/** 圓形：面積公式 pi 乘以半徑平方 */
class Circle extends Shape {
    private final double radius;

    public Circle(double radius) {
        super("圓形");           // 把名稱交給父類別保管
        this.radius = radius;
    }

```

```
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

/** 矩形：面積公式為寬乘以高 */
class Rectangle extends Shape {
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        super("矩形");
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
```

注意 `report()` 這個一般方法：它呼叫了自己類別裡「還不存在實作」的 `area()`。這完全合法——執行期 `this` 一定是某個具體子類別的物件，動態繫結會找到正確的公式。「父類別定義流程、子類別填入細節」，這正是進階篇樣板方法模式的雛形。

範例：ShapeDemo.java

```
/** 用多型陣列統一處理所有形狀 */
public class ShapeDemo {
    public static void main(String[] args) {
        // Shape s = new Shape("形狀");   // 編譯錯誤：抽象類別不能實體化

        Shape[] shapes = {
            new Circle(1.0),
            new Rectangle(3.0, 4.0),
            new Circle(2.0)
        };

        double total = 0;
        for (Shape s : shapes) {
```

```
            System.out.println(s.report());   // 各自執行正確的 area()
            total += s.area();                // 累加總面積
        }
        System.out.printf("總面積 = %.2f%n", total);
    }
}
```

▶ 執行結果

```
圓形面積 = 3.141592653589793
矩形面積 = 12.0
圓形面積 = 12.566370614359172
總面積 = 27.71
```

> **注意**　抽象類別不能實體化：`new Shape(...)` 是編譯錯誤，因為抽象方法沒有本體，這種物件根本不完整。同時注意連鎖規則：子類別若沒有實作全部的抽象方法，它自己也必須宣告成 `abstract`，否則編譯失敗。抽象類別仍然可以有建構子（如上例的 `protected Shape(String)`）——它不能被 `new`，但子類別建構子需要透過 `super(...)` 呼叫它來初始化共同欄位。

2.4 介面：能力的契約

抽象類別仍然是「類別」，受限於單一繼承：一個類別只能有一個直接父類別。但現實中的能力常常是橫切的——會飛的東西不全是鳥，可排序的東西不全是數字。Java 用**介面**（interface）表達這種「能力契約」：介面只規定「你會做什麼」（方法簽章），不限制「你是什麼」（繼承體系），而且一個類別可以**實作**（implement）任意多個介面。

2.4.1 初識標準介面：Comparable

標準函式庫大量使用介面表達能力。`Comparable` 介面代表「我能與同類比較大小」——只要實作它，`Arrays.sort` 就知道怎麼排序你的物件：

範例：ComparableDemo.java

```
import java.util.Arrays;

/**
 * 產品：實作 Comparable，宣告「產品之間可以比較大小」。
 * compareTo 回傳負數、零、正數，分別代表小於、等於、大於。
 */
```

```java
class Product implements Comparable<Product> {
    private final String name;
    private final int price;

    Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public int compareTo(Product other) {
        // 委託 Integer.compare：以價格由小到大
        return Integer.compare(this.price, other.price);
    }

    @Override
    public String toString() {
        return name + "(" + price + "元)";
    }
}

/** 實作 Comparable 之後，標準函式庫就知道怎麼排序 */
public class ComparableDemo {
    public static void main(String[] args) {
        Product[] items = {
            new Product("鍵盤", 1200),
            new Product("滑鼠", 450),
            new Product("螢幕", 5500)
        };
        Arrays.sort(items);    // sort 內部會呼叫 compareTo
        System.out.println(Arrays.toString(items));
    }
}
```

▶ 執行結果

```
[滑鼠(450元), 鍵盤(1200元), 螢幕(5500元)]
```

Arrays.sort 的作者從未見過 Product，卻能正確排序它——因為排序演算法只依賴 Comparable 這份契約。這就是「面向介面設計」的威力。與之配套的 Comparator 介面則把比較邏輯抽成獨立物件，讓同一種資料能依不同規則排序，第 4 章將深入使用。

2.4.2 介面成員的演進：常數、default、static 與 private 方法

早期的介面只能放抽象方法與常數。Java 8 為了讓標準函式庫能演進（在既有介面加方法，又不弄壞全世界已經寫好的實作類別），引入了 `default` 方法（帶預設實作，實作類別可沿用可覆寫）與 `static` 方法；Java 9 再補上 `private` 方法，供介面內部共用邏輯：

範例：AudioDemo.java

```
/** 音訊播放器介面：抽象方法、常數、default、static、private 全到齊 */
interface AudioPlayer {

    /** 介面中的欄位隱含 public static final：這是一個常數 */
    int MAX_VOLUME = 100;

    /** 抽象方法：實作類別必須提供本體 */
    void play(String track);

    /** default 方法：附帶預設實作，實作類別可以直接沿用或覆寫 */
    default void playAll(String[] tracks) {
        for (String t : tracks) {
            if (isValid(t)) {      // 呼叫介面自己的 private 方法
                play(t);           // 呼叫抽象方法：繫結到實作類別
            }
        }
    }

    /** static 方法：屬於介面本身，用介面名稱呼叫 */
    static int defaultVolume() {
        return MAX_VOLUME / 2;
    }

    /** private 方法（Java 9+）：介面內部共用的輔助邏輯 */
    private boolean isValid(String track) {
        return track != null && !track.isBlank();
    }
}

/** 老式音響：只實作必要的抽象方法，playAll 直接用預設版本 */
class Stereo implements AudioPlayer {
    @Override
    public void play(String track) {
        System.out.println("播放中：" + track);
    }
```

```
}

/** 驗證 default、static 方法與介面常數 */
public class AudioDemo {
    public static void main(String[] args) {
        Stereo stereo = new Stereo();
        // Stereo 沒有寫 playAll：直接繼承介面的預設實作
        stereo.playAll(new String[] { "夜曲", "青花瓷" });

        // static 方法屬於介面本身
        System.out.println("預設音量：" + AudioPlayer.defaultVolume());
        // 介面常數：隱含 public static final
        System.out.println("音量上限：" + AudioPlayer.MAX_VOLUME);
    }
}
```

▶ 執行結果

```
播放中：夜曲
播放中：青花瓷
預設音量：50
音量上限：100
```

介面的方法隱含 `public`，欄位隱含 `public static final`——這也是為什麼介面「沒有實體欄位、沒有建構子」：它不描述狀態，只描述能力。

2.4.3 實作多個介面與鑽石問題

一個類別可以同時實作多個介面：`class A implements X, Y`。當兩個介面提供了**同名的 default 方法**，就出現所謂的鑽石問題——編譯器無法替你決定繼承哪一份實作，因此強制你自己覆寫，並可用 `介面名.super.方法名()` 指定沿用的版本：

範例：DiamondDemo.java

```
/** 相機能力 */
interface Camera {
    default String describe() {
        return "能拍照";
    }
}

/** 電話能力 */
interface Phone {
```

```
    default String describe() {
        return "能通話";
    }
}

/**
 * 智慧型手機同時實作兩個介面：
 * 兩邊都有 default 的 describe()，編譯器無法決定要繼承哪一個，
 * 因此「強制」我們自己覆寫來排解衝突。
 */
class SmartPhone implements Camera, Phone {
    @Override
    public String describe() {
        // 「介面名.super.方法名」：明確指定沿用哪個介面的版本
        return Camera.super.describe() + "，而且" + Phone.super.describe();
    }
}

/** 驗證鑽石問題的解法 */
public class DiamondDemo {
    public static void main(String[] args) {
        System.out.println(new SmartPhone().describe());
    }
}
```

▶ 執行結果

```
能拍照，而且能通話
```

2.4.4 介面與抽象類別的選擇

兩者都能定義「不完整的型別」，選擇準則整理如下：

比較面向	抽象類別	介面
關係語意	is-a（是一種）	can-do（具備某能力）
繼承數量	只能繼承一個	可以實作多個
欄位	任意實體欄位	只能有常數
建構子	有（供子類別呼叫）	無
方法種類	一般、抽象皆可	抽象、default、static、private
存取修飾詞	不限	方法隱含 public
典型用途	一族類別共享狀態與部分實作	跨類別族系的能力契約

> **提示**　實務上的判斷順序：**預設先考慮介面**，因為它不佔用寶貴的單一繼承名額，耦合也最鬆；只有當一族類別需要**共享欄位（狀態）**或**大量共同實作**時，才動用抽象類別。兩者也常常合作：抽象類別實作介面、先消化掉一部分方法，讓具體子類別更輕鬆——集合框架中 `AbstractList` 之於 `List` 就是這個模式。

2.5　綜合範例：付款系統

最後把本章觀念整合成一個付款系統：`Payment` 介面定義付款能力，兩種付款方式各自實作，結帳櫃台只面對介面。請特別觀察 `checkout` 方法——它是**多型參數**的典型：任何實作 `Payment` 的物件都能傳入，日後新增付款方式時這個方法完全不必修改。這種「把演算法抽成可替換物件」的手法，正是進階篇策略模式的原型。

範例：CheckoutDemo.java

```
/**
 * 付款方式介面：所有付款方式的共同契約。
 * 只規定「能做什麼」，不限制「怎麼做」。
 */
interface Payment {

    /** 執行扣款，回傳是否成功 */
    boolean pay(int amount);

    /** default 方法：多數付款方式共用的手續費（1%，不足 1 元捨去） */
    default int fee(int amount) {
        return amount / 100;
    }
}

/** 信用卡付款 */
class CreditCardPayment implements Payment {

    private final String cardNumber;   // 卡號：實務上必須加密保存

    CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    @Override
    public boolean pay(int amount) {
        // 只顯示末四碼：資訊安全的基本禮貌
```

```
        String tail = cardNumber.substring(cardNumber.length() - 4);
        System.out.println("信用卡（末四碼 " + tail + "）扣款 "
                + amount + " 元，手續費 " + fee(amount) + " 元");
        return true;
    }
}

/** LINE Pay 付款：覆寫 fee 提供免手續費優惠 */
class LinePayPayment implements Payment {

    @Override
    public boolean pay(int amount) {
        System.out.println("LINE Pay 扣款 " + amount
                + " 元，手續費 " + fee(amount) + " 元");
        return true;
    }

    /** 行銷活動：LINE Pay 免手續費 */
    @Override
    public int fee(int amount) {
        return 0;
    }
}

/** 結帳櫃台：只認識 Payment 介面，不關心實際的付款方式 */
public class CheckoutDemo {

    /** 多型參數：任何實作 Payment 的物件都能傳進來 */
    static void checkout(Payment method, int amount) {
        if (method.pay(amount)) {
            System.out.println("結帳完成");
        }
    }

    public static void main(String[] args) {
        // 同一個 checkout 方法，搭配不同的付款策略
        checkout(new CreditCardPayment("4311-1234-5678-9012"), 1500);
        checkout(new LinePayPayment(), 800);
    }
}
```

▶ 執行結果

```
信用卡（末四碼 9012）扣款 1500 元，手續費 15 元
結帳完成
LINE Pay 扣款 800 元，手續費 0 元
結帳完成
```

追蹤一下輸出：信用卡扣款 1500 元時，`fee` 使用介面的 default 實作，$1500 \div 100 = 15$ 元；LINE Pay 覆寫了 `fee` 回傳 0。同一個 `method.pay(amount)` 呼叫式，兩種物件走出兩條路——本章的一切，濃縮在這兩行輸出裡。

2.6 本章小結

- 多型：同一訊息（方法呼叫），不同物件展現不同行為；基礎是向上轉型與動態繫結。
- 參考型別決定「能呼叫哪些方法」（編譯期檢查）；實際型別決定「執行哪個覆寫版本」（執行期繫結）。
- 欄位與 static 方法採靜態繫結、不具多型；避免宣告與父類別同名的欄位。
- 向下轉型前先檢查：傳統寫法用 `instanceof` 加轉型，Java 16 起可用模式匹配 `if (obj instanceof Dog dog)` 一步到位。
- 抽象類別不能實體化；抽象方法只有簽章，子類別必須全部實作，否則自己也得是抽象類別。
- 抽象類別的一般方法可以呼叫抽象方法：父類別定骨架、子類別填細節。
- 介面是能力契約：欄位隱含 `public static final`，方法隱含 `public`；Java 8 起有 default 與 static 方法，Java 9 起有 private 方法。
- default 方法的存在理由是介面演進：替既有介面加方法而不破壞既有實作。
- 兩個介面的 default 方法衝突（鑽石問題）時，實作類別必須覆寫，並可用「介面名.super.方法名」指定沿用版本。
- 選擇準則：預設先想介面；需要共享狀態與大量共同實作時才用抽象類別。

2.7 本章練習

動手練習

1. 觀念題：`Animal a = new Dog();` 之後，`a.fetch()` 無法編譯，但 `a.speak()` 執行的卻是 `Dog` 的版本。請分別用「參考型別」與「實際型別」解釋這兩個現象。（提示：一個發生在編譯期，一個發生在執行期。）
2. 實作題：擴充本章的 `Shape` 體系，新增 `Triangle` 類別（底與高），並確認 `ShapeDemo`

的迴圈不需修改就能處理它。（提示：面積公式為底乘高除以二。）

3. 實作題：定義 `Describable` 介面（含 `String describe()` 抽象方法與一個 default 方法 `void printDescription()` 負責印出描述），讓第 1 章的 `Employee` 實作它。（提示：default 方法內可以呼叫 `describe()`。）
4. 觀念題：為什麼 Java 允許實作多個介面，卻只允許繼承一個類別？若允許多重繼承類別，會出現什麼歧義？（提示：想想兩個父類別各有一份同名「欄位」與有狀態的方法實作時，子類別該用哪一份。）
5. 實作題：為付款系統新增 `BankTransferPayment`（轉帳手續費固定 15 元，覆寫 `fee`），並在 `main` 中用同一個 `checkout` 完成三種付款。（提示：只需新增類別，`checkout` 不准修改。）
6. 挑戰題：寫一個 `Payment[]` 多型陣列裝入三種付款方式，走訪陣列對每筆訂單試算手續費，並用模式匹配 `instanceof` 找出其中的信用卡付款、印出其末四碼。（提示：`if (p instanceof CreditCardPayment c)`。）

第 3 章

例外處理

本章學習目標

學完本章你將能夠：

- 說明例外機制相較於「回傳錯誤碼」的優勢，並描繪 Throwable 的類別階層。
- 分辨受檢（checked）與未受檢（unchecked）例外，理解兩者背後的設計哲學。
- 熟練使用 try-catch、多重 catch、multi-catch、finally 與 try-with-resources。
- 分辨 throw 與 throws，設計帶欄位的自訂例外並正確選擇父類別。
- 用例外鏈保留錯誤因果，並能逐行讀懂堆疊追蹤。
- 遵循業界最佳實務：不吞例外、精準捕獲、訊息帶上下文。

程式不會永遠走在快樂路徑上：檔案可能不存在、輸入可能是亂碼、餘額可能不足。一個系統的品質，往往不是看它正常運作時多優雅，而是看它出錯時多可靠。Java 的例外（exception）機制把「錯誤的表達、傳遞與處理」從一般的回傳值中獨立出來，讓錯誤既不會被忽略，也不會淹沒主要邏輯。

前兩章打造的類別已經開始拋出 `IllegalArgumentException`，本章正式解釋這一切如何運作：從機制的動機、類別階層、語法全家桶，一路到自訂例外與業界實務，最後用一台 ATM 提款機整合所有概念。

3.1 為什麼需要例外

在例外機制普及之前，函式用「特殊回傳值」回報失敗：找不到就回傳 `-1`、失敗就回傳 `null` 或錯誤碼。這種做法有三個致命缺陷，看一個典型案例：

範例：ErrorCodeStyle.java

```
/** 用「錯誤碼」回報失敗的傳統做法：問題重重 */
public class ErrorCodeStyle {

    /** 找不到就回傳 -1：呼叫端很容易忘記檢查 */
    static int indexOf(int[] data, int target) {
        for (int i = 0; i < data.length; i++) {
            if (data[i] == target) {
```

```
                return i;
            }
        }
        return -1;    // 魔術數字：錯誤訊號混在正常回傳值裡
    }

    public static void main(String[] args) {
        int[] scores = { 70, 85, 90 };
        int index = indexOf(scores, 100);
        // 忘了檢查 -1 就直接使用：錯誤悄悄往下游擴散
        System.out.println("找到成績，位於索引 " + index);
    }
}
```

▶ 執行結果

```
找到成績，位於索引 -1
```

程式「正常結束」，卻印出一個荒謬的結果。錯誤碼的缺陷在於：(1) **可以被忽略**——編譯器不會強迫呼叫端檢查回傳值，錯誤默默往下游擴散，等到真正炸開時已離案發現場很遠；(2) **訊號與資料混在同一條通道**——`-1` 既是錯誤也是合法的 `int`，語意全靠註解與默契；(3) **錯誤資訊貧乏**——一個整數說不清楚「什麼壞了、為什麼、在哪裡」。例外機制針對性地解決這三點：例外不處理就會讓程式終止（無法被無聲忽略）、例外走獨立的傳遞通道（不佔用回傳值）、例外是完整的物件（能攜帶訊息、欄位與堆疊追蹤）。

3.2　Throwable 家族

所有能被 `throw` 拋出、被 `catch` 接住的東西，都是 `Throwable` 的後代。整個家族的版圖如下：

- `Throwable`：一切可拋出物的始祖
  - `Error`：JVM 層級的嚴重問題，程式不應捕捉。代表：`OutOfMemoryError`（記憶體耗盡）、`StackOverflowError`（呼叫堆疊溢位）。
  - `Exception`：程式可以合理應對的狀況
    - 受檢例外（checked）：`Exception` 的直接後代。代表：`IOException`、`FileNotFoundException`、`SQLException`。
    - `RuntimeException` 及其後代（unchecked，未受檢）：`NullPointerException`、`ArrayIndexOutOfBoundsException`、`IllegalArgumentException`、`ArithmeticException`、`ClassCastException`。

受檢與未受檢的差別，是 Java 例外設計的核心：

比較面向	受檢例外（checked）	未受檢例外（unchecked）
繼承位置	`Exception`（非 Runtime 分支）	`RuntimeException` 分支
代表例外	`IOException`、`SQLException`	`NullPointerException` 等
編譯器強制？	是：必須 catch 或宣告 throws	否：可處理也可放行
語意	可預期的外部狀況	程式本身的邏輯錯誤
典型場景	檔案、網路、商業規則	參數不合法、索引越界
正確反應	就地處理或明確上報	修程式，而不是包 try

設計哲學是這樣的：**受檢例外**代表「就算程式寫得完美也可能發生」的外部狀況（磁碟滿了、網路斷了、餘額不足），編譯器強迫你正視它；**未受檢例外**代表「程式寫錯了」（對 `null` 呼叫方法、陣列越界），正確的對策是修正程式，而不是到處包 try-catch 遮掩。這也是第 1 章的 setter 拋 `IllegalArgumentException` 而不是自訂受檢例外的原因——傳錯參數是呼叫端的臭蟲。

3.3　try-catch：接住例外

基本語法：把可能出事的程式放進 `try` 區塊，用一個以上的 `catch` 區塊分類接住。例外發生的當下，`try` 中剩餘的敍述全部跳過，控制權轉移到第一個「型別相符」的 `catch`：

範例：TryCatchDemo.java

```
/** try-catch 基本流程與多重 catch 的比對順序 */
public class TryCatchDemo {

    /** 取出陣列元素、解析成整數、再拿來當除數 */
    static int compute(String[] data, int index) {
        int value = Integer.parseInt(data[index]);   // 可能拋兩種例外
        return 100 / value;                          // 也可能除以零
    }

    public static void main(String[] args) {
        String[] data = { "4", "abc", "0" };
        int[] tests = { 0, 1, 2, 9 };   // 用不同索引觸發不同例外

        for (int i : tests) {
            try {
                System.out.println("結果：" + compute(data, i));
            } catch (NumberFormatException e) {
                // 較特定的例外放前面（它是 RuntimeException 的後代）
                System.out.println("格式錯誤：" + e.getMessage());
            } catch (ArithmeticException e) {
```

```
                System.out.println("算術錯誤：" + e.getMessage());
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("索引錯誤：" + e.getMessage());
            }
        }
        System.out.println("程式正常結束");
    }
}
```

▶ 執行結果

```
結果：25
格式錯誤：For input string: "abc"
算術錯誤：/ by zero
索引錯誤：Index 9 out of bounds for length 3
程式正常結束
```

四輪迭代各自展示一條路徑：索引 0 一路順利；索引 1 在 `parseInt` 拋出 `NumberFormatException`；索引 2 解析成功但除以零；索引 9 連取值都失敗。注意例外發生在 `compute` 內部、卻在 `main` 被接住——例外會沿著呼叫鏈向上傳遞，直到遇見合適的 catch，這讓「出錯的地方」與「決定怎麼辦的地方」可以分離。

> **注意**　多個 catch 的比對順序是由上而下、先中先贏，因此子類別例外必須放在父類別之前。如果把 `catch (Exception e)` 放在第一個，後面的 catch 永遠輪不到——Java 編譯器會直接報錯「exception has already been caught」，但若父子關係比較隱晦（例如 `IllegalArgumentException` 與它的子類別 `NumberFormatException`），順序寫反就是一個編譯錯誤等著你。口訣：**先子後父，由特殊到一般**。

當多種例外的處理方式一模一樣時，用 multi-catch 把它們併在同一個 catch 中（以直線符號分隔），避免複製貼上：

範例：multi-catch 語法片段

```
try {
    int value = Integer.parseInt(args[0]);   // 可能拋 NumberFormat
    System.out.println(100 / value);         // 可能拋 Arithmetic
} catch (NumberFormatException | ArithmeticException e) {
    // 兩種例外共用同一段處理；e 的型別是兩者最近的共同父類別
    System.out.println("輸入無法處理：" + e.getMessage());
}
```

3.4　finally 與 try-with-resources

3.4.1　finally：保證執行的收尾區塊

`finally` 區塊無論 try 成功、失敗、甚至中途 `return`，都保證執行，傳統上用來釋放資源。最反直覺的一點是它與 `return` 的互動——先看再解釋：

範例：FinallyDemo.java

```
/** finally 的執行時機：即使 try 中有 return，finally 仍會先執行 */
public class FinallyDemo {

    static String load(boolean fail) {
        try {
            if (fail) {
                throw new IllegalStateException("讀取失敗");
            }
            System.out.println("  try：成功讀取資料");
            return "資料";      // 準備回傳前，finally 仍會插隊執行
        } catch (IllegalStateException e) {
            System.out.println("  catch：" + e.getMessage());
            return "預設值";
        } finally {
            // 無論成功、失敗、甚至已經 return，這裡都保證執行
            System.out.println("  finally：關閉資源");
        }
    }

    public static void main(String[] args) {
        System.out.println("回傳：" + load(false));
        System.out.println("回傳：" + load(true));
    }
}
```

▶ 執行結果

```
  try：成功讀取資料
  finally：關閉資源
回傳：資料
  catch：讀取失敗
  finally：關閉資源
回傳：預設值
```

執行到 `return "資料";` 時，回傳值先被計算並保存，接著執行 `finally`，最後才真正把值交回呼叫端——所以「finally：關閉資源」印在「回傳：資料」之前。也因此**絕對不要在 finally 裡寫 return**：它會覆蓋 try 與 catch 的回傳值、甚至吃掉尚未傳遞的例外。

3.4.2 try-with-resources：自動關閉資源

手動在 finally 關資源既囉嗦又容易寫錯（關閉動作本身也可能拋例外）。Java 7 起，任何實作 `AutoCloseable` 介面的物件，都能宣告在 try 的小括號中，離開區塊時自動呼叫 `close()`。自訂一個資源類別來觀察時機與順序：

範例：TryWithResourcesDemo.java

```
/** 自訂資源：實作 AutoCloseable 就能交給 try-with-resources 管理 */
class Resource implements AutoCloseable {

    private final String name;

    Resource(String name) {
        this.name = name;
        System.out.println("開啟 " + name);
    }

    /** 使用資源 */
    void use() {
        System.out.println("使用 " + name);
    }

    /** try 區塊結束時被自動呼叫，不必手動釋放 */
    @Override
    public void close() {
        System.out.println("關閉 " + name);
    }
}

/** 驗證多個資源的關閉順序：後開的先關（後進先出） */
public class TryWithResourcesDemo {
    public static void main(String[] args) {
        // 小括號中可以宣告多個資源，以分號分隔
        try (Resource a = new Resource("資源A");
             Resource b = new Resource("資源B")) {
            a.use();
            b.use();
```

```
        }   // 這裡自動先呼叫 b.close()，再呼叫 a.close()
        System.out.println("try 區塊之後");
    }
}
```

▶ 執行結果

```
開啟 資源A
開啟 資源B
使用 資源A
使用 資源B
關閉 資源B
關閉 資源A
try 區塊之後
```

關閉順序是**後進先出**：後宣告的資源可能依賴先宣告的（例如包裝流依賴底層流），所以必須先關。即使 try 中途拋出例外，所有已開啟的資源仍會依序關閉。第 6 章的檔案操作將全面使用這個語法。

3.5　throw、throws 與自訂例外

兩個關鍵字一字之差，角色完全不同：`throw`（動詞，無 s）用在方法**內部**，真正拋出一個例外物件，例如 `throw new IllegalArgumentException("...")`；`throws`（宣告，有 s）用在方法簽章上，聲明「本方法可能拋出這些受檢例外，呼叫我的人請自行處理」。未受檢例外不需要列在 throws（列了也無妨，可作為文件）。

當標準例外的語意不夠精確時，就該自訂例外。選擇父類別的準則承襲前面的哲學：**可預期的商業狀況繼承 Exception（受檢），呼叫端的程式錯誤繼承 RuntimeException（未受檢）**。以下兩個自訂例外將用於本章壓軸的 ATM 範例：

範例：InvalidAmountException.java

```
/**
 * 提款金額不合法：呼叫端的程式錯誤（傳了不該傳的值），
 * 因此繼承 RuntimeException（unchecked），不強迫捕捉。
 */
public class InvalidAmountException extends RuntimeException {

    public InvalidAmountException(String message) {
        super(message);   // 把錯誤訊息交給父類別保管
    }
}
```

範例：InsufficientFundsException.java

```
/**
 * 餘額不足：正常營運中「可預期」的商業狀況，
 * 繼承 Exception（checked），強迫呼叫端正視並處理。
 * 例外可以帶欄位：把「差多少錢」這種上下文一併帶給呼叫端。
 */
public class InsufficientFundsException extends Exception {

    private final int shortage;   // 不足的金額（元）

    public InsufficientFundsException(int shortage) {
        // 先組出對人類友善的訊息，交給父類別保管
        super("餘額不足，還差 " + shortage + " 元");
        this.shortage = shortage;
    }

    /** 呼叫端可以取回差額，做進一步的補救處理 */
    public int getShortage() {
        return shortage;
    }
}
```

注意 InsufficientFundsException 帶著一個 shortage 欄位：例外是完整的類別，除了訊息字串，還能攜帶結構化的資料，讓呼叫端不必去「解析錯誤訊息」就能取得差額——這是自訂例外相較於標準例外最大的加值空間。

3.6 例外鏈：保留因果

分層的系統中，低階例外常需要「翻譯」成高階例外：資料層的 NumberFormatException 對使用者毫無意義，包成「設定檔錯誤」才有溝通價值。但翻譯時絕不能丟掉原始例外——把它作為 cause（肇因）傳給新例外的建構子，形成例外鏈：

範例：ChainDemo.java

```
/** 高階例外：對呼叫端有意義的「設定檔錯誤」 */
class ConfigException extends RuntimeException {
    public ConfigException(String message, Throwable cause) {
        super(message, cause);   // cause 會被記住，形成因果鏈
    }
}

```

```java
/** 例外鏈：把底層例外包成高階例外，因果完整保留 */
public class ChainDemo {

    /** 讀取連接埠設定：把低階的解析錯誤翻譯成高階錯誤 */
    static int readPort(String raw) {
        try {
            return Integer.parseInt(raw);
        } catch (NumberFormatException e) {
            // 用第二個參數把原始例外接上因果鏈，證據不丟失
            throw new ConfigException("連接埠設定值無效：" + raw, e);
        }
    }

    public static void main(String[] args) {
        try {
            readPort("eighty");
        } catch (ConfigException e) {
            System.out.println("捕獲：" + e.getMessage());
            // getCause() 可以取回最初的元凶
            System.out.println("肇因：" + e.getCause());
        }
    }
}
```

▶ 執行結果

```
捕獲：連接埠設定值無效：eighty
肇因：java.lang.NumberFormatException: For input string: "eighty"
```

有例外鏈時，`printStackTrace` 會連同所有 cause 一起印出（以「Caused by:」開頭的段落），除錯時可以一路追到最初的案發現場。

3.7 閱讀堆疊追蹤

例外一路沒人接住時，執行緒終止，JVM 把堆疊追蹤（stack trace）印到標準錯誤。讀懂它是每位 Java 工程師的基本功：

範例：StackTraceDemo.java

```java
/** 產生一個未捕獲的例外，觀察堆疊追蹤的結構 */
public class StackTraceDemo {

```

```
    /** 第三層：例外真正的發源地 */
    static int level3(String s) {
        return Integer.parseInt(s);   // 這裡拋出 NumberFormatException
    }

    /** 第二層：只是路過，沒有處理例外 */
    static int level2(String s) {
        return level3(s);
    }

    public static void main(String[] args) {
        level2("oops");               // 呼叫鏈的起點
    }
}
```

▶ 執行結果

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "oops"
    at java.base/java.lang.NumberFormatException.forInputString(
    ↪ NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:662)
    at java.base/java.lang.Integer.parseInt(Integer.java:778)
    at StackTraceDemo.level3(StackTraceDemo.java:6)
    at StackTraceDemo.level2(StackTraceDemo.java:11)
    at StackTraceDemo.main(StackTraceDemo.java:15)
```

逐行解讀：第一行是「哪條執行緒、哪種例外、什麼訊息」；接下來每一行 `at` 是呼叫堆疊的一格，**由上而下代表由深到淺**——最上面是例外誕生的位置，最下面是 `main`。括號內是「檔名加行號」：`StackTraceDemo.java:6` 正是 `parseInt` 那一行。實戰讀法：**由上往下掃，找到第一行屬於你自己程式碼的 `at`**（跳過 `java.base` 開頭的函式庫內部），那裡通常就是要修的地方。函式庫內部的行號（如 `Integer.java:662`）會因 JDK 版本而略有不同，不必在意。

3.8 例外處理最佳實務

> **注意**　**最嚴重的反模式是吞掉例外**：`catch (Exception e) { }` 空區塊讓錯誤憑空消失，系統帶著壞狀態繼續跑，數週後在毫不相干的地方炸開，而所有證據早已湮滅。最低限度也要記錄下來（開發階段可用 `e.printStackTrace()`，正式系統用日誌框架）；若當下真的無法處理，就讓它繼續往上拋。同樣危險的是**一把抓**：`catch (Exception e)` 會把你根本沒預期到的錯誤（包括程式臭蟲）也攔下來，掩蓋真正的問題。

> **提示**　三條正向原則：(1) **捕獲要精準**——只 catch 你知道怎麼處理的型別，處理不了的讓它上拋；(2) **訊息要有上下文**——「餘額不足，還差 1000 元」遠勝於「錯誤」，建立例外時把關鍵變數放進訊息與欄位；(3) **及早失敗（fail fast）**——像第 1 章的 setter 一樣在入口就驗證並拋出例外，錯誤越早現形，離病灶越近，修起來越便宜。

3.9　綜合範例：ATM 提款機

把本章所有概念組裝起來：`Atm` 類別的 `withdraw` 以防衛子句依序檢查規則，違反時分別拋出未受檢的 `InvalidAmountException`（金額格式錯誤，屬於呼叫端臭蟲）與受檢的 `InsufficientFundsException`（餘額不足，屬於可預期的商業狀況）；呼叫端用多重 catch 分類處理，流程不會因為單筆失敗而中斷。

範例：Atm.java

```
/**
 * ATM 提款機：整合自訂例外的完整範例。
 * 規則：金額必須是 1000 的正倍數，且不可超過餘額。
 */
public class Atm {

    private int balance;   // 帳戶餘額（元）

    public Atm(int balance) {
        this.balance = balance;
    }

    /**
     * 提款。
     * InvalidAmountException 是 unchecked：不必列在 throws（列出也可），
     * InsufficientFundsException 是 checked：必須宣告，呼叫端必須處理。
     */
    public void withdraw(int amount) throws InsufficientFundsException {
        // 防衛子句：先把不合法的輸入擋在門外
        if (amount <= 0 || amount % 1000 != 0) {
            throw new InvalidAmountException(
                    "金額必須是 1000 的正倍數，收到：" + amount);
        }
        if (amount > balance) {
            // 建立例外時帶入上下文（差額），對呼叫端更有用
            throw new InsufficientFundsException(amount - balance);
        }
```

```
        balance -= amount;   // 所有檢查都通過，才真正扣款
        System.out.println("吐鈔 " + amount + " 元，餘額 "
                + balance + " 元");
    }

    public int getBalance() {
        return balance;
    }
}
```

範例：AtmDemo.java

```
/** 模擬一連串提款操作，示範兩類例外各自的處理方式 */
public class AtmDemo {
    public static void main(String[] args) {
        Atm atm = new Atm(5000);
        int[] requests = { 3000, 500, 3000, 2000 };

        for (int amount : requests) {
            System.out.println("== 申請提款 " + amount + " 元 ==");
            try {
                atm.withdraw(amount);
            } catch (InvalidAmountException e) {
                // unchecked：不接也能編譯，但這裡選擇就地處理
                System.out.println("輸入有誤：" + e.getMessage());
            } catch (InsufficientFundsException e) {
                // checked：編譯器強迫我們面對這個商業狀況
                System.out.println(e.getMessage()
                        + "（本帳戶可提上限 " + atm.getBalance() + " 元）");
            }
        }
        System.out.println("最終餘額：" + atm.getBalance() + " 元");
    }
}
```

▶ 執行結果

```
== 申請提款 3000 元 ==
吐鈔 3000 元，餘額 2000 元
== 申請提款 500 元 ==
輸入有誤：金額必須是 1000 的正倍數，收到：500
== 申請提款 3000 元 ==
```

```
餘額不足，還差 1000 元（本帳戶可提上限 2000 元）
== 申請提款 2000 元 ==
吐鈔 2000 元，餘額 0 元
最終餘額：0 元
```

追蹤流程：初始餘額 5000，第一筆 3000 成功（餘額 2000）；第二筆 500 不是 1000 的倍數，被防衛子句擋下；第三筆 3000 超過餘額 2000，差額恰為 1000；第四筆 2000 成功，餘額歸零。特別觀察：`withdraw` 把所有檢查放在任何狀態修改之前——這保證例外發生時物件狀態毫髮無傷，不會出現「錢扣了一半」的慘劇，與第 1 章的不變條件一脈相承。

3.10 本章小結

- 錯誤碼的三宗罪：可被忽略、訊號與資料混雜、資訊貧乏；例外機制對症下藥。
- 階層：`Throwable` 分為不應捕捉的 `Error` 與可以處理的 `Exception`；後者再分受檢與未受檢（`RuntimeException` 分支）。
- 受檢例外表達「可預期的外部與商業狀況」，編譯器強迫處理；未受檢例外表達「程式邏輯錯誤」，正解是修程式。
- 多重 catch 由上而下比對：先子後父；處理邏輯相同時用 multi-catch 合併。
- `finally` 保證執行，連 try 中的 `return` 都攔得住；不要在 finally 中寫 return。
- try-with-resources 自動關閉 `AutoCloseable` 資源，多個資源後進先出。
- `throw` 拋出例外物件；`throws` 在簽章上聲明可能拋出的受檢例外。
- 自訂例外：商業狀況繼承 `Exception`，程式錯誤繼承 `RuntimeException`；善用欄位攜帶結構化上下文。
- 翻譯例外時用 cause 建構子保留因果鏈；`getCause()` 可取回元凶。
- 讀堆疊追蹤：由上往下找到第一行自己的程式碼；絕不吞例外、捕獲要精準、訊息要有上下文。

3.11 本章練習

動手練習

1. 觀念題：`FileNotFoundException` 設計成受檢、`NullPointerException` 設計成未受檢，請分別說明理由。（提示：哪一個是「程式再完美也可能發生」？哪一個是「程式寫錯了」？）
2. 觀念題：以下程式印出什麼？`try` 中 `return 1;`，`finally` 中 `System.out.println`

↪ ("F"); ——若把 finally 改成 return 2; 又會如何？為什麼後者是危險寫法？（提示：回傳值先保存、finally 插隊執行。）

3. 實作題：寫一個 safeDivide(int a, int b) 方法：b 為零時拋出帶訊息的 ArithmeticException，並在 main 中示範捕獲；再改寫成回傳錯誤碼的版本，比較兩者的呼叫端程式碼。（提示：體會「無法忽略」的價值。）
4. 實作題：自訂 AgeOutOfRangeException（帶 age 欄位），讓 Person 的 setter 在年齡不在 0 到 150 之間時拋出。思考它該是受檢還是未受檢，說明理由。（提示：傳入非法年齡是誰的錯？）
5. 實作題：自訂 Connection 類別實作 AutoCloseable，在建構子與 close() 各印一行，用 try-with-resources 同時開啟三個，驗證關閉順序為後進先出。（提示：模仿本章 Resource 範例。）
6. 挑戰題：擴充 ATM：新增每日提款上限 60000 元的規則與對應自訂例外 DailyLimitExceededException，並讓它與 InsufficientFundsException 共用一個自訂父類別 AtmException，呼叫端只用一個 catch 接住兩者。（提示：例外也能形成繼承階層。）

第 4 章

集合框架

本章學習目標

學完本章你將能夠：

- 描繪集合框架的整體階層：List、Set、Queue 與自成體系的 Map。
- 熟練 ArrayList 與 LinkedList 的操作，並依效能特性選擇。
- 用四種方式走訪集合，並正確地在走訪中刪除元素。
- 解釋 HashSet 去重的原理，知道自訂類別必須覆寫 equals 與 hashCode。
- 熟練 HashMap 的操作與三種走訪，分辨 Hash、LinkedHash、Tree 三種實作的順序特性。
- 用 Comparator 與 lambda 自訂排序規則，並依需求選出正確的集合型別。

初級篇的陣列有兩個天生的限制：**長度固定**（建立後不能增減，塞滿了得手動搬家）與**功能陽春**（搜尋、去重、鍵值對應都要自己寫迴圈）。Java 的**集合框架**（Collections Framework）用一組精心設計的介面與實作類別解決這些問題：容量自動成長、常用操作一應俱全，而且以介面為中心，換一種實作不必改動使用端的程式。

集合框架也是前三章知識的第一個大型應用現場：它以介面定義契約（第 2 章）、以泛型保證型別安全（下一章深入）、以 `equals` 與 `hashCode` 契約運作（第 1 章）。本章聚焦最常用的成員，學會之後，你日常八成的資料處理需求都有現成的工具。

4.1 集合框架總覽

框架的骨幹是兩棵介面樹。`Collection` 是「一群元素」的容器；`Map` 存的是「鍵值對」，自成體系、不繼承 `Collection`：

- `Collection`（介面）：單一元素的容器
  - `List`：有序、可重複、可用索引存取——`ArrayList`、`LinkedList`
  - `Set`：不可重複——`HashSet`、`LinkedHashSet`、`TreeSet`
  - `Queue`：佇列（先進先出）——`ArrayDeque`、`PriorityQueue`；其子介面 `Deque` 是雙端佇列，也常拿來當堆疊
- `Map`（介面）：鍵值對，鍵不可重複——`HashMap`、`LinkedHashMap`、`TreeMap`

命名規律值得記住：**前綴代表資料結構**（Hash 雜湊表、Linked 鏈結、Tree 紅黑樹、Array 陣列），**後綴代表介面**。看到 `TreeMap` 就該反射出「用樹實作的 Map，所以依鍵排序」。

4.2 泛型語法與自動裝箱

集合宣告時用角括號指定元素型別，例如 `List<String>`。Java 7 起右邊可以用菱形運算子省略型別參數，由編譯器推斷：`List<String> list = new ArrayList<>();`。宣告時建議**左邊用介面型別**（`List` 而非 `ArrayList`），日後要換實作只需改 `new` 的那一端。

角括號裡只能放參考型別，不能放 `int` 這類基本型別，因此每個基本型別都有對應的**包裝類別**（`Integer`、`Double`、`Boolean` 等）。Java 會自動在兩者之間轉換——放進集合時**自動裝箱**（autoboxing），取出運算時**自動拆箱**（unboxing）。方便歸方便，有一個著名陷阱：

範例：BoxingTrap.java

```java
/** 包裝類別的快取陷阱：== 比較的是參考，不是數值 */
public class BoxingTrap {
    public static void main(String[] args) {
        // 自動裝箱：int 值自動包成 Integer 物件
        Integer a = 100;
        Integer b = 100;
        Integer c = 1000;
        Integer d = 1000;

        // -128～127 落在快取區間：a 與 b 指向同一個快取物件
        System.out.println("100 == 100：" + (a == b));

        // 1000 超出快取區間：c 與 d 是兩個不同的物件
        System.out.println("1000 == 1000：" + (c == d));

        // 正解：包裝物件一律用 equals 比較數值
        System.out.println("equals 比較：" + c.equals(d));
    }
}
```

▶ 執行結果

```
100 == 100：true
1000 == 1000：false
equals 比較：true
```

> **注意**　`Integer` 會快取 −128 到 127 的物件，裝箱時這個範圍回傳同一顆快取物件，範圍外則每次建立新物件——於是 `==`（比較參考）對小數字碰巧為 `true`、對大數字為 `false`，測試時全過、上線後炸裂。鐵律：**包裝物件比較數值一律用 `equals`**（或先拆箱成基本型別再比）。這正是第 1 章「`==` 比參考、`equals` 比內容」的又一次體現。

4.3　List：有序、可重複

4.3.1　ArrayList 常用操作

`ArrayList` 是使用率最高的集合，底層是一塊會自動擴容的陣列：容量不足時配置一塊約 1.5 倍大的新陣列並搬移既有元素。搬家偶爾發生、平攤下來尾端新增仍是常數時間，但這也解釋了它的效能特性：依索引存取極快，頭部或中間插入刪除則需要搬動後續元素。

範例：ArrayListBasics.java

```java
import java.util.ArrayList;
import java.util.List;

/** ArrayList 常用操作總覽 */
public class ArrayListBasics {
    public static void main(String[] args) {
        // 用介面型別宣告 + 菱形運算子：右邊的型別參數可以省略
        List<String> fruits = new ArrayList<>();

        fruits.add("蘋果");        // 加到尾端
        fruits.add("香蕉");
        fruits.add("櫻桃");
        fruits.add(1, "藍莓");     // 插到索引 1，後面的元素往後挪
        System.out.println("目前內容：" + fruits);

        System.out.println("取索引 2：" + fruits.get(2));
        System.out.println("元素個數：" + fruits.size());
        System.out.println("包含香蕉？" + fruits.contains("香蕉"));
        System.out.println("香蕉的索引：" + fruits.indexOf("香蕉"));

        fruits.set(0, "鳳梨");     // 覆蓋索引 0 的元素
        fruits.remove("藍莓");     // 依值刪除（第一個相符者）
        fruits.remove(1);          // 依索引刪除
        System.out.println("調整之後：" + fruits);
        System.out.println("是否為空：" + fruits.isEmpty());
    }
```

```
}
```

▶ 執行結果

```
目前內容：[蘋果, 藍莓, 香蕉, 櫻桃]
取索引 2：香蕉
元素個數：4
包含香蕉？true
香蕉的索引：2
調整之後：[鳳梨, 櫻桃]
是否為空：false
```

逐步追蹤最後兩步：`remove("藍莓")` 依值刪除（內部用 `equals` 比對），清單變成鳳梨、香蕉、櫻桃；`remove(1)` 依索引刪除香蕉，剩下鳳梨與櫻桃。整數清單要特別小心這組多載：`list.remove(1)` 刪的是索引 1，`list.remove(Integer.valueOf(1))` 才是刪除值為 1 的元素。

4.3.2 LinkedList 與效能對照

`LinkedList` 底層是**雙向鏈結串列**：每個節點持有元素以及指向前後節點的參考。它同時實作了 `List` 與 `Deque` 介面，因此還提供 `addFirst`、`removeFirst` 等頭尾操作：

範例：LinkedList 當雙端佇列的片段

```
LinkedList<String> queue = new LinkedList<>();
queue.addLast("甲");     // 尾端加入：一般排隊
queue.addLast("乙");
queue.addFirst("急件");  // 頭端插隊：O(1)，不必搬動任何元素
String next = queue.removeFirst();   // 取出隊頭：「急件」
```

兩種 List 的效能對照（n 為元素個數）：

操作	ArrayList	LinkedList
依索引存取 `get(i)`	$O(1)$：陣列直接定址	$O(n)$：從頭或尾逐節點走
尾端新增 `add(e)`	平攤 $O(1)$	$O(1)$
頭端插入與刪除	$O(n)$：整批往後搬	$O(1)$：改幾個參考
中間插入與刪除	$O(n)$：搬移後續元素	定位 $O(n)$，接鏈 $O(1)$
記憶體	連續緊湊	每節點多兩個參考的負擔

實務建議：**預設用 ArrayList**。現代硬體上連續記憶體對快取極度友善，ArrayList 幾乎全面勝出；只有「頻繁在頭端進出」的佇列型場景值得考慮 `LinkedList` 或效能更好的 `ArrayDeque`。

4.4 走訪集合的四種方式

範例：TraversalDemo.java

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/** 四種走訪方式的比較 */
public class TraversalDemo {
    public static void main(String[] args) {
        List<String> langs =
                new ArrayList<>(List.of("Java", "Kotlin", "Go"));

        // 方式一：傳統索引 for——需要索引值的時候用
        for (int i = 0; i < langs.size(); i++) {
            System.out.println("索引 " + i + "：" + langs.get(i));
        }

        // 方式二：for-each——最常用，簡潔且不會寫錯邊界
        for (String lang : langs) {
            System.out.print(lang + " ");
        }
        System.out.println();

        // 方式三：Iterator——走訪中需要刪除元素時的正規做法
        Iterator<String> it = langs.iterator();
        while (it.hasNext()) {
            System.out.print(it.next().length() + " ");
        }
        System.out.println();

        // 方式四：forEach 搭配 lambda（進階篇詳述，先感受寫法）
        langs.forEach(lang -> System.out.print(lang + " "));
        System.out.println();
    }
}
```

▶ 執行結果

```
索引 0：Java
索引 1：Kotlin
```

```
索引 2：Go
Java Kotlin Go
4 6 2
Java Kotlin Go
```

迭代器（Iterator）是集合框架的統一走訪協定：`hasNext()` 問「還有嗎」，`next()` 取下一個。for-each 只是它的語法糖——編譯器會把方式二改寫成方式三。順帶一提，`List.of(...)` 建立的是不可變清單，任何修改都會拋 `UnsupportedOperationException`，所以上面用 `new ArrayList<>(...)` 包一層取得可變複本。

4.4.1 走訪中刪除元素

「把清單裡的偶數都刪掉」是新手墳場。直覺寫法——在 for-each 裡呼叫 `remove`——會當場翻車：

範例：BrokenRemoval.java

```java
import java.util.ArrayList;
import java.util.List;

/** 錯誤示範：在 for-each 走訪中直接刪除元素 */
public class BrokenRemoval {
    public static void main(String[] args) {
        List<Integer> nums = new ArrayList<>(List.of(1, 2, 3, 4));
        for (Integer n : nums) {      // for-each 底層其實是 Iterator
            if (n % 2 == 0) {
                nums.remove(n);       // 動到結構，迭代器隨即失效
            }
        }
        System.out.println(nums);     // 執行不到這一行
    }
}
```

▶ 執行結果

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1095)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:1049)
    at BrokenRemoval.main(BrokenRemoval.java:8)
```

> **注意**　集合內部有一個「結構修改次數」計數器，迭代器建立時會記下當時的值；走訪途中若集合被迭代器以外的管道修改（如直接呼叫 `list.remove`），下一次 `next()` 發現計數對不上，立刻拋出 `ConcurrentModificationException`。這是**快速失敗**（fail-fast）設計：與其讓走訪在錯亂的結構上繼續、產生無法解釋的結果，不如立刻報錯。堆疊追蹤指向 for 那一行（第 8 行），正是隱藏的 `next()` 呼叫點。

兩種正解——透過迭代器自己的 `remove`，或交給 `removeIf`：

範例：RemoveWhileIterating.java

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3  import java.util.List;
4
5  /** 走訪中刪除元素的兩種正解 */
6  public class RemoveWhileIterating {
7      public static void main(String[] args) {
8          // 目標：刪除清單中所有偶數
9          List<Integer> nums = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));
10
11         // 正解一：透過 Iterator 自己的 remove 方法刪除
12         Iterator<Integer> it = nums.iterator();
13         while (it.hasNext()) {
14             if (it.next() % 2 == 0) {
15                 it.remove();   // 由迭代器代勞，內部狀態保持一致
16             }
17         }
18         System.out.println("正解一：" + nums);
19
20         // 正解二：removeIf 搭配條件（Java 8+，最簡潔）
21         List<Integer> nums2 = new ArrayList<>(List.of(1, 2, 3, 4, 5, 6));
22         nums2.removeIf(n -> n % 2 == 0);   // 把「該刪的條件」交給集合
23         System.out.println("正解二：" + nums2);
24     }
25 }
```

▶ 執行結果

```
正解一：[1, 3, 5]
正解二：[1, 3, 5]
```

4.5 Set：不重複的集合

Set 的唯一使命是不重複。主力實作 HashSet 的去重原理直接建立在第 1 章的契約上：加入元素時先用 hashCode() 定位到雜湊桶，再用 equals() 與桶內元素比對，兩關都判定「相同」才視為重複。因此——**自訂類別要放進 HashSet，必須同時正確覆寫 equals 與 hashCode**，缺一不可：

範例：HashSetDemo.java

```
import java.util.LinkedHashSet;
import java.util.Objects;
import java.util.Set;

/** 會員：正確覆寫 equals 與 hashCode，HashSet 才能正確去重 */
class Member {
    private final String id;      // 會員編號：身分依據
    private final String name;

    Member(String id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        return id.equals(((Member) obj).id);   // 以編號判斷同一人
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);   // 與 equals 使用同一組欄位
    }

    @Override
    public String toString() {
        return name + "(" + id + ")";
```

```
    }
}

/** 驗證去重是否生效 */
public class HashSetDemo {
    public static void main(String[] args) {
        // 用 LinkedHashSet 讓輸出順序可預測（依插入順序）
        Set<Member> members = new LinkedHashSet<>();

        members.add(new Member("M001", "小明"));
        members.add(new Member("M002", "小華"));
        // 編號重複：equals 與 hashCode 判定為同一人，不會加入
        boolean added = members.add(new Member("M001", "小明"));

        System.out.println("重複加入成功嗎？" + added);
        System.out.println("集合內容：" + members);
        // contains 也是靠 hashCode 與 equals 運作
        System.out.println("包含 M002 嗎？"
                + members.contains(new Member("M002", "任意名字")));
    }
}
```

▶ 執行結果

```
重複加入成功嗎？false
集合內容：[小明(M001), 小華(M002)]
包含 M002 嗎？true
```

若忘了覆寫 hashCode，兩個「編號相同」的物件會落在不同的雜湊桶，equals 連被呼叫的機會都沒有——重複元素照收不誤、contains 找不到東西，正是第 1 章警告過的契約災難。三種 Set 的順序特性比較：

範例：SetOrderDemo.java

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

/** 三種 Set 的順序行為比較 */
public class SetOrderDemo {
    public static void main(String[] args) {
        String[] data = { "banana", "apple", "cherry", "apple" };
```

```

        Set<String> hash = new HashSet<>();       // 不保證任何順序
        Set<String> linked = new LinkedHashSet<>(); // 記住插入順序
        Set<String> tree = new TreeSet<>();       // 隨時保持排序

        for (String s : data) {   // 同一批資料放進三種 Set
            hash.add(s);
            linked.add(s);
            tree.add(s);
        }

        // HashSet 的走訪順序不可依賴，因此只印元素個數
        System.out.println("元素個數（重複已剔除）：" + hash.size());
        System.out.println("LinkedHashSet：" + linked);
        System.out.println("TreeSet：" + tree);
    }
}
```

▶ 執行結果

```
元素個數（重複已剔除）：3
LinkedHashSet：[banana, apple, cherry]
TreeSet：[apple, banana, cherry]
```

HashSet 最快但不保證走訪順序（本書凡需要穩定輸出之處一律改用 LinkedHashSet）；LinkedHashSet 額外用一條鏈結記住插入順序；TreeSet 用紅黑樹隨時維持排序（操作為 $O(\log n)$），但元素必須實作 Comparable，或在建構時提供 Comparator。

4.6 Map：鍵值對

Map 是「查表」的抽象：以鍵找值，鍵不可重複。HashMap 的原理與 HashSet 同源（事實上 HashSet 內部就是一顆 HashMap），自訂類別當鍵時同樣必須覆寫 equals 與 hashCode：

範例：MapBasics.java

```
import java.util.LinkedHashMap;
import java.util.Map;

/** Map 常用操作：以「商品庫存」為例 */
public class MapBasics {
    public static void main(String[] args) {
        // 用 LinkedHashMap 讓走訪順序可預測（HashMap 不保證順序）
```

```
        Map<String, Integer> stock = new LinkedHashMap<>();

        stock.put("鍵盤", 10);    // 新增鍵值對
        stock.put("滑鼠", 25);
        stock.put("螢幕", 3);
        stock.put("鍵盤", 12);    // 鍵已存在：新值覆蓋舊值

        System.out.println("鍵盤庫存：" + stock.get("鍵盤"));
        // get 查無此鍵回傳 null；getOrDefault 可以給預設值
        System.out.println("耳機庫存：" + stock.getOrDefault("耳機", 0));
        System.out.println("有賣螢幕嗎？" + stock.containsKey("螢幕"));

        stock.remove("螢幕");     // 依鍵刪除

        // merge：鍵存在就用函數合併新舊值；不存在就直接放入
        stock.merge("滑鼠", 5, Integer::sum);   // 25 + 5 = 30
        stock.merge("耳機", 8, Integer::sum);   // 不存在，直接放 8

        System.out.println("目前庫存：" + stock);
        System.out.println("商品種類數：" + stock.size());
    }
}
```

▶ 執行結果

```
鍵盤庫存：12
耳機庫存：0
有賣螢幕嗎？true
目前庫存：{鍵盤=12, 滑鼠=30, 耳機=8}
商品種類數：3
```

`merge` 是「計數、累加」場景的利器：第三個參數 `Integer::sum` 是方法參考，意思是「把兩個數相加」（進階篇詳述，這裡先當成慣用片語）。三種走訪方式如下——只要鍵、只要值、或鍵值都要：

範例：MapTraversal.java

```
import java.util.LinkedHashMap;
import java.util.Map;

/** Map 的三種走訪方式 */
public class MapTraversal {
    public static void main(String[] args) {
```

```
        Map<String, Integer> scores = new LinkedHashMap<>();
        scores.put("國文", 88);
        scores.put("英文", 95);
        scores.put("數學", 72);

        // 走訪一：keySet()——只需要鍵
        for (String subject : scores.keySet()) {
            System.out.print(subject + " ");
        }
        System.out.println();

        // 走訪二：values()——只需要值
        int total = 0;
        for (int score : scores.values()) {
            total += score;   // 自動拆箱：Integer 轉回 int
        }
        System.out.println("總分：" + total);

        // 走訪三：entrySet()——鍵值都要（完整走訪的首選）
        for (Map.Entry<String, Integer> entry : scores.entrySet()) {
            System.out.println(entry.getKey() + "：" + entry.getValue());
        }
    }
}
```

▶ 執行結果

```
國文 英文 數學
總分：255
國文：88
英文：95
數學：72
```

鍵值都需要時務必用 `entrySet()`：走訪 `keySet()` 再對每個鍵呼叫 `get` 等於把查找做了兩次。順序特性與 Set 完全平行：`HashMap` 不保證順序、`LinkedHashMap` 依插入順序、`TreeMap` 依鍵排序。

4.7　排序：Comparator 與 Collections 工具

第 2 章的 `Comparable` 定義的是型別「與生俱來」的單一排序；`Comparator` 則把比較邏輯抽成獨立物件，同一種資料可以隨場合換規則。搭配 lambda 與工廠方法，排序程式碼精簡得驚人：

範例：SortingDemo.java

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

/** 學生：示範排序用的簡單資料類別 */
class Student {
    final String name;   // 聚焦排序示範，欄位用套件層級存取
    final int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public String toString() {
        return name + "=" + score;
    }
}

/** Comparator 搭配 lambda：排序規則想怎麼換就怎麼換 */
public class SortingDemo {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>(List.of(
                new Student("Bob", 92),
                new Student("Cindy", 85),
                new Student("Amy", 85),
                new Student("David", 78)));

        // 依分數遞增：comparingInt 接收「怎麼取出比較鍵」的函數
        list.sort(Comparator.comparingInt(s -> s.score));
        System.out.println("分數遞增：" + list);

        // 依分數遞減：reversed() 反轉既有規則
        list.sort(Comparator.comparingInt((Student s) -> s.score)
                            .reversed());
        System.out.println("分數遞減：" + list);

        // 多鍵排序：先比分數（遞減），同分再依姓名字母序
        list.sort(Comparator.comparingInt((Student s) -> s.score)
                            .reversed()
```

```
                                .thenComparing(s -> s.name));
        System.out.println("同分比名：" + list);
    }
}
```

▶ 執行結果

```
分數遞增：[David=78, Cindy=85, Amy=85, Bob=92]
分數遞減：[Bob=92, Cindy=85, Amy=85, David=78]
同分比名：[Bob=92, Amy=85, Cindy=85, David=78]
```

觀察前兩行輸出中 Cindy 與 Amy 的相對位置：兩人同為 85 分，順序維持原樣——Java 的排序是**穩定排序**，同值元素不會被打亂；第三行加上 `thenComparing` 之後，同分改依姓名排序，Amy 才排到 Cindy 前面。另外，`Collections` 工具類別提供一批操作集合的靜態方法：

範例：CollectionsDemo.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/** Collections 工具類別的常用靜態方法 */
public class CollectionsDemo {
    public static void main(String[] args) {
        List<Integer> nums = new ArrayList<>(List.of(5, 1, 4, 2, 3));

        Collections.sort(nums);       // 遞增排序（就地修改）
        System.out.println("排序：" + nums);

        Collections.reverse(nums);    // 反轉
        System.out.println("反轉：" + nums);

        System.out.println("最大值：" + Collections.max(nums));
        System.out.println("最小值：" + Collections.min(nums));

        // Collections.shuffle(nums);   // 隨機洗牌：每次結果不同

        // 不可修改視圖：任何寫入操作都會拋出例外
        List<Integer> readOnly = Collections.unmodifiableList(nums);
        try {
            readOnly.add(99);
        } catch (UnsupportedOperationException e) {
            System.out.println("唯讀清單拒絕修改");
```

```
        }
    }
}
```

▶ 執行結果

```
排序：[1, 2, 3, 4, 5]
反轉：[5, 4, 3, 2, 1]
最大值：5
最小值：1
唯讀清單拒絕修改
```

`unmodifiableList` 回傳的是視圖而非複本：把它交給外部程式，對方就只能讀不能寫，是保護內部集合的常用手法（注意原清單若變動，視圖看到的內容也會跟著變）。

4.8 集合選型指南

需求	建議集合
有序清單、依索引隨機存取	`ArrayList`
頻繁在頭尾兩端進出（佇列、堆疊）	`ArrayDeque`
去除重複、只問「有沒有」	`HashSet`
去重且保留插入順序	`LinkedHashSet`
去重且隨時排好序	`TreeSet`
以鍵查值、計數、分組	`HashMap`
以鍵查值且保留插入順序	`LinkedHashMap`
以鍵查值且依鍵排序	`TreeMap`

> **提示**　選型的思考順序：先問**資料的形狀**（一串元素選 `Collection` 家族；鍵值對選 `Map` 家族），再問**要不要去重**（要就 Set），最後問**順序需求**（不在乎選 Hash 系最快；要插入順序選 Linked 系；要排序選 Tree 系）。三個問題答完，答案幾乎唯一。

4.9 綜合範例：單字頻率統計

收尾範例是文字處理的經典任務：統計一段文字中每個單字出現的次數，並列出前三名。解法動員了本章多位主角——`HashMap` 計數、`entrySet` 搬運、`Comparator` 排序：

範例：WordFrequency.java

```
import java.util.ArrayList;
import java.util.Comparator;
```

```java
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 單字頻率統計：集合框架的綜合應用。
 * 流程：切割文字、用 HashMap 計數、搬進 List 排序、輸出前三名。
 */
public class WordFrequency {
    public static void main(String[] args) {
        String text = "the quick brown fox jumps over the lazy dog "
                + "the fox is quick and the dog is lazy";

        // 步驟一：以空白切割成單字陣列
        String[] words = text.split(" ");

        // 步驟二：HashMap 累計次數（鍵：單字；值：出現次數）
        Map<String, Integer> freq = new HashMap<>();
        for (String w : words) {
            freq.merge(w, 1, Integer::sum);   // 首次放 1，之後累加
        }

        // 步驟三：Map 本身沒有順序概念，搬進 List 才能排序
        List<Map.Entry<String, Integer>> entries =
                new ArrayList<>(freq.entrySet());

        // 次數遞減；次數相同再依字母序，讓結果完全可預測
        entries.sort(Comparator
                .comparingInt((Map.Entry<String, Integer> e)
                        -> e.getValue())
                .reversed()
                .thenComparing(Map.Entry::getKey));

        // 步驟四：輸出前三名
        System.out.println("出現次數前三名：");
        for (int i = 0; i < 3; i++) {
            Map.Entry<String, Integer> e = entries.get(i);
            System.out.println((i + 1) + ". " + e.getKey()
                    + "：" + e.getValue() + " 次");
        }
    }
}
```

▶ 執行結果

```
出現次數前三名：
1. the：4 次
2. dog：2 次
3. fox：2 次
```

驗算一下：the 出現 4 次；dog、fox、is、lazy、quick 各出現 2 次，依字母序 dog 與 fox 排進前三名。這裡計數用 `HashMap`（不需要順序、速度優先），輸出前才排序——「儲存用最快的結構、呈現前再整理順序」是集合實務的常見節奏。

4.10 本章小結

- 集合框架兩大體系：`Collection`（List 有序可重複、Set 不重複、Queue 佇列）與 `Map`（鍵值對）；命名規律是「資料結構前綴加介面後綴」。
- 宣告用介面型別加菱形運算子：`List<String> list = new ArrayList<>();`。
- 自動裝箱讓基本型別進出集合；比較包裝物件一律用 `equals`，`==` 會踩中 -128 到 127 的快取陷阱。
- `ArrayList` 底層是自動擴容陣列：隨機存取 $O(1)$、頭端插入 $O(n)$；預設選它，頭尾頻繁進出才考慮 `ArrayDeque`。
- 走訪有四式：索引 for、for-each、`Iterator`、`forEach` 加 lambda。
- 走訪中刪除元素必用 `Iterator.remove` 或 `removeIf`，直接呼叫集合的 remove 會拋 `ConcurrentModificationException`。
- `HashSet` 去重靠 `hashCode` 定位加 `equals` 比對：自訂類別必須兩者一起覆寫。
- 順序特性：Hash 系不保證、Linked 系記插入順序、Tree 系自動排序（需 `Comparable` 或 `Comparator`，操作 $O(\log n)$）。
- `Map` 常用：`put`、`get`、`getOrDefault`、`containsKey`、`remove`、`merge`；鍵值都要時走 `entrySet()`。
- `Comparator` 搭配 `comparing` 系列、`reversed`、`thenComparing` 組合排序規則；Java 排序是穩定排序。

4.11 本章練習

動手練習

1. 觀念題：`Integer a = 127, b = 127;` 時 `a == b` 為 `true`，改成 128 之後變 `false`，為什麼？正確的比較方式是什麼？（提示：整數快取區間與參考比較。）
2. 實作題：建立一個 `ArrayList` 存放 1 到 20，先用 `removeIf` 刪除所有 3 的倍數，再用 `Iterator` 手動刪除所有大於 15 的數，印出結果。（提示：兩種正解各練一次。）
3. 實作題：自訂 `Point` 類別（欄位 x、y），先「不覆寫」`equals` 與 `hashCode`，把兩個座標相同的點加入 `HashSet` 觀察大小；再正確覆寫後重測。（提示：親眼見證契約的效果。）
4. 實作題：用 `HashMap` 統計一句英文中每個「字母」出現的次數，輸出時依字母順序排列。（提示：`merge` 計數；輸出前把 keySet 搬進 `TreeSet` 或改用 `TreeMap`。）
5. 實作題：對 `Student` 清單完成三種排序：依分數遞減、依姓名字母序、先依分數遞減同分再依姓名。（提示：`comparingInt`、`reversed`、`thenComparing`。）
6. 挑戰題：用 `Map<String, List<String>>` 把一批單字依「字首字母」分組，例如 a 對應 apple 與 ant。走訪輸出時依字首排序。（提示：`computeIfAbsent(key, k -> new ArrayList<>())` 可以優雅地建立分組清單。）

第 5 章

泛型

本章學習目標

學完本章你將能夠：

- 說明泛型要解決的問題：把型別錯誤從執行期提前到編譯期。
- 自行設計泛型類別（單一與多個型別參數）與泛型方法，並遵循命名慣例。
- 用有界型別參數（extends）限制型別，寫出既通用又安全的演算法。
- 分辨三種萬用字元的適用時機，並以 PECS 原則設計方法簽章。
- 解釋泛型的不變性、陣列的協變性，以及型別抹除帶來的各種限制。
- 實作一個型別安全的泛型資料結構。

上一章你已經在使用泛型：`List<String>` 保證清單裡只有字串。本章換到另一邊，學會設計泛型——寫出像 `ArrayList<E>` 那樣「一份程式碼、服務所有型別」的類別與方法。泛型是 Java 5 引入的編譯期機制，核心理念只有一句話：**讓編譯器在寫程式的當下就抓到型別錯誤，而不是讓使用者在執行期看到程式爆炸。**

理解泛型的最好起點，是回到沒有泛型的年代，看看工程師們踩過什麼坑。

5.1 沒有泛型的年代

Java 5 之前，集合的元素型別一律是 `Object`：什麼都能放，取出來卻只知道是 `Object`，必須自己強制轉型。編譯器對「放錯東西」完全沉默，錯誤直到執行期才引爆：

範例：RawTypeDanger.java

```
import java.util.ArrayList;
import java.util.List;

/** raw type 的執行期炸彈：編譯器閉嘴，執行期爆炸 */
public class RawTypeDanger {
    public static void main(String[] args) {
        // raw type：不帶型別參數的集合，什麼都能放
        List names = new ArrayList();
        names.add("小明");
```

```
        names.add(42);                // 整數也放得進去，編譯器只給警告

        for (Object obj : names) {
            // 取出時必須自己轉型：輪到 42 的時候當場爆炸
            String name = (String) obj;
            System.out.println("名字長度：" + name.length());
        }
    }
}
```

▶ 執行結果

```
名字長度：2
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer
    ↪ cannot be cast to class java.lang.String (java.lang.Integer and java.lang.
    ↪ String are in module java.base of loader 'bootstrap')
    at RawTypeDanger.main(RawTypeDanger.java:14)
```

問題的根源：「這個清單只放字串」的約定只存在於工程師的腦中，編譯器無從得知。泛型就是把這個約定寫進型別系統——宣告 `List<String>` 之後，`add(42)` 直接編譯失敗，取出時也不再需要轉型。錯誤發生的時間點從「使用者操作到那一行的深夜」提前到「你寫下那一行的當下」，這就是泛型的全部價值。

不帶型別參數的寫法（`List`、`ArrayList`）稱為 **raw type**，僅為了與老程式碼相容而存在。新程式碼永遠不要使用 raw type——編譯器發出的「unchecked」警告，就是在提醒你型別安全已經破功。

5.2　泛型類別

在類別名稱後面加上角括號與型別參數，這個類別就成了泛型類別。型別參數是「型別的佔位符」，等到使用端宣告時才填入具體型別。命名有業界慣例：`T`（Type，一般型別）、`E`（Element，集合元素）、`K` 與 `V`（Key 與 Value，鍵與值）、`R`（Result，回傳值）：

範例：Box 與 Pair 泛型類別

```
/**
 * 泛型類別 Box：型別參數 T 是「型別的佔位符」，
 * 等到使用的那一刻才決定 T 是什麼。
 */
class Box<T> {

    private T content;   // 欄位的型別用型別參數表示
```

```

    /** 放入內容 */
    public void put(T content) {
        this.content = content;
    }

    /** 取出內容：回傳型別就是 T，呼叫端完全不必轉型 */
    public T get() {
        return content;
    }
}

/** Pair：兩個型別參數（K 與 V）的泛型類別 */
class Pair<K, V> {

    private final K key;
    private final V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    @Override
    public String toString() {
        return "(" + key + ", " + value + ")";
    }
}
```

範例：BoxDemo.java

```
/** 使用泛型類別 */
public class BoxDemo {
    public static void main(String[] args) {
```

```
        // T 指定為 String：put 只收 String，get 直接回傳 String
        Box<String> gift = new Box<>();   // 菱形運算子：右側型別可省略
        gift.put("音樂盒");
        String content = gift.get();      // 不需要 (String) 轉型
        System.out.println("盒子裡是：" + content);

        // gift.put(42);   // 編譯錯誤：42 不是 String，錯誤提前現形

        // K 是 String、V 是 Integer
        Pair<String, Integer> entry = new Pair<>("身高", 175);
        System.out.println(entry.getKey() + "：" + entry.getValue());
        System.out.println("完整輸出：" + entry);
    }
}
```

▶ 執行結果

```
盒子裡是：音樂盒
身高：175
完整輸出：(身高, 175)
```

對照 RawTypeDanger：同樣是「放進去、拿出來」，Box<String> 讓錯誤的 put(42) 在編譯期就被攔下，正確的 get() 則不需要任何轉型。型別檢查與轉型的苦工全部由編譯器代勞。

5.3 泛型方法

型別參數也可以只屬於單一方法：宣告在回傳型別之前的角括號中。泛型方法不需要類別本身是泛型的，static 方法也適用。多數情況下編譯器能從引數推斷出型別參數，呼叫端寫起來與一般方法無異：

範例：GenericMethodDemo.java

```
import java.util.List;

/** 泛型方法：型別參數宣告在回傳型別「之前」 */
public class GenericMethodDemo {

    /** 交換陣列中兩個位置的元素：對任何參考型別都適用 */
    static <T> void swap(T[] array, int i, int j) {
        T tmp = array[i];      // 型別參數讓暫存變數的型別正確
        array[i] = array[j];
        array[j] = tmp;
```

```
    }

    /** 回傳清單的第一個元素：回傳型別跟著傳入的清單走 */
    static <T> T first(List<T> list) {
        return list.get(0);
    }

    public static void main(String[] args) {
        String[] words = { "甲", "乙", "丙" };
        swap(words, 0, 2);     // 型別推斷：T 自動推為 String
        System.out.println(String.join("", words));

        Integer[] nums = { 1, 2, 3 };
        // 也可以明確指定型別參數（通常不必，編譯器推得出來）
        GenericMethodDemo.<Integer>swap(nums, 0, 1);
        System.out.println(nums[0] + ", " + nums[1] + ", " + nums[2]);

        List<Double> prices = List.of(19.99, 5.5);
        double p = first(prices);   // T 推斷為 Double，再自動拆箱
        System.out.println("第一個價格：" + p);
    }
}
```

▶ 執行結果

```
丙乙甲
2, 1, 3
第一個價格：19.99
```

5.4 有界型別參數

「任何型別都可以」有時太寬鬆。想寫一個找最大值的方法，方法內必須呼叫 `compareTo`——那 T 就不能是隨便的型別，必須「會比較」。用 `extends` 設定上界（介面也用 extends，不用 implements）：

範例：BoundedTypeDemo.java

```
/** 有界型別參數：T 必須「會比較大小」才能找最大值 */
public class BoundedTypeDemo {

    /**
     * 回傳三者中的最大值。
```

```
     * 界限「extends Comparable」保證 T 一定實作了 Comparable，
     * 否則方法本體根本無法呼叫 compareTo。
     */
    static <T extends Comparable<T>> T max(T a, T b, T c) {
        T result = a;                    // 先假設 a 最大
        if (b.compareTo(result) > 0) {   // 界限保證 compareTo 存在
            result = b;
        }
        if (c.compareTo(result) > 0) {
            result = c;
        }
        return result;
    }

    public static void main(String[] args) {
        // Integer 與 String 都實作了 Comparable，直接可用
        System.out.println("最大整數：" + max(3, 9, 5));
        System.out.println("最大字串：" + max("pear", "apple", "kiwi"));
        // max(new Object(), ...)：編譯錯誤，Object 沒實作 Comparable
    }
}
```

▶ 執行結果

```
最大整數：9
最大字串：pear
```

界限是雙向的承諾：對方法本體，它保證 compareTo 一定存在，程式才編得過；對呼叫端，它擋掉不具備能力的型別。界限可以不只一個——`<T extends A & B>` 表示 T 必須同時滿足 A 與 B（最多一個類別且必須寫在最前面，其餘為介面），例如 `<T extends Animal & Comparable<T>>`。

5.5 萬用字元

型別參數解決「定義泛型」的問題，**萬用字元**（wildcard）`?` 則解決「接收各種泛型」的問題，共三種形式：無界 `<?>`（任何型別的清單都收）、上界 `<? extends T>`（T 或其子型別的清單）、下界 `<? super T>`（T 或其父型別的清單）：

範例：WildcardDemo.java

```
import java.util.ArrayList;
import java.util.List;
```

```
3
4  /** 萬用字元三兄弟：無界 ?、上界 ? extends、下界 ? super */
5  public class WildcardDemo {
6
7      /** 無界萬用字元：任何 List 都能收，但元素只能當 Object 讀 */
8      static void printAll(List<?> list) {
9          for (Object item : list) {   // 只知道它「是個物件」
10             System.out.print(item + " ");
11         }
12         System.out.println();
13     }
14
15     /** 上界：收 Number 或其子型別的清單——只能讀（生產者） */
16     static double sum(List<? extends Number> list) {
17         double total = 0;
18         for (Number n : list) {      // 讀出來保證是 Number
19             total += n.doubleValue();
20         }
21         // list.add(1);   // 編譯錯誤：實際型別未知，寫入不安全
22         return total;
23     }
24
25     /** 下界：收 Integer 或其父型別的清單——可以安全寫入（消費者） */
26     static void fillCount(List<? super Integer> list, int n) {
27         for (int i = 1; i <= n; i++) {
28             list.add(i);             // 寫入 Integer 一定相容
29         }
30     }
31
32     public static void main(String[] args) {
33         List<Integer> ints = List.of(1, 2, 3);
34         List<Double> doubles = List.of(0.5, 1.5);
35
36         printAll(ints);       // List<?> 什麼清單都能印
37         printAll(doubles);
38
39         // 上界：Integer、Double 的清單都可以傳進 sum
40         System.out.println("整數和：" + sum(ints));
41         System.out.println("小數和：" + sum(doubles));
42
43         // 下界：List<Number>（Integer 的父型別清單）可以傳入
44         List<Number> numbers = new ArrayList<>();
```

```
        fillCount(numbers, 3);
        System.out.println("寫入結果：" + numbers);
    }
}
```

▶ 執行結果

```
1 2 3
0.5 1.5
整數和：6.0
小數和：2.0
寫入結果：[1, 2, 3]
```

為什麼「上界只能讀、不能寫」？`List<? extends Number>` 的實際型別可能是 `List<Integer>` 也可能是 `List<Double>`——編譯器無法確定。讀是安全的：不管哪一種，取出來一定「是一種 Number」。寫則不安全：往可能是 `List<Double>` 的清單裡放 `Integer` 顯然錯誤，既然無法排除這個可能，編譯器乾脆禁止一切寫入（`null` 除外）。反過來，`List<? super Integer>` 的實際型別是 `List<Integer>`、`List<Number>` 或 `List<Object>`——不管哪一種，「放進一個 Integer」都合法，所以寫入 **Integer** 安全；但讀出來只能當 `Object`，因為元素實際型別無法保證。

5.5.1 PECS 原則

把上面的推理濃縮成一條設計口訣，就是著名的 **PECS**：Producer Extends, Consumer Super——資料的生產者（你從它讀）用 `extends`，資料的消費者（你往它寫）用 `super`。標準函式庫的複製方法正是教科書級示範：

範例：PECS 的標準示範片段

```
/** 把來源清單的元素全部複製到目的清單 */
static <T> void copy(List<? super T> dst, List<? extends T> src) {
    for (T item : src) {    // src 是生產者：從它讀出 T（extends）
        dst.add(item);      // dst 是消費者：把 T 寫進它（super）
    }
}

// 呼叫範例：來源是 List<Integer>，目的地可以是 List<Number>
// List<Number> dst = new ArrayList<>();
// copy(dst, List.of(1, 2, 3));   // T 推斷為 Integer
```

> **提示**　設計方法簽章時的操作步驟：先問「這個參數對方法而言是讀進來還是寫出去」。只讀用 `? extends`、只寫用 `? super`、又讀又寫就只能用精確型別 `List<T>`。遵循 PECS 的簽章能接受的引數範圍最大，這正是 `Collections.copy` 等標準方法的實際寫法。

5.6　泛型與繼承：不變性

`Integer` 是 `Number` 的子類別，但 `List<Integer>` 不是 `List<Number>` 的子型別——泛型是不變的（invariant）。對照組是陣列：`Integer[]` 可以指派給 `Number[]`（協變，covariant）。陣列的寬容其實是歷史包袱，它把型別錯誤延後到執行期：

範例：CovarianceDemo.java

```
/** 陣列是協變的：編譯器放行，執行期才發現放錯東西 */
public class CovarianceDemo {
    public static void main(String[] args) {
        // 陣列：Integer[] 可以指派給 Number[]（協變）
        Number[] numbers = new Integer[2];
        numbers[0] = 1;          // OK：放 Integer
        try {
            numbers[1] = 3.14;   // 編譯過關，執行期才被抓包
        } catch (ArrayStoreException e) {
            System.out.println("執行期例外：" + e);
        }

        // 泛型：List<Integer> 不能指派給 List<Number>（不變性）
        // List<Number> list = new ArrayList<Integer>();   // 編譯錯誤
        // 這正是泛型的初衷：把錯誤攔在編譯期，而不是執行期
        System.out.println("陣列內容：" + numbers[0] + ", " + numbers[1]);
    }
}
```

▶ 執行結果

```
執行期例外：java.lang.ArrayStoreException: java.lang.Double
陣列內容：1, null
```

> **注意**　`numbers` 的宣告型別是 `Number[]`，放 `3.14` 在編譯器眼中完全合法；但它實際指向 `Integer[]`，JVM 只能在執行期用 `ArrayStoreException` 補救——注意輸出第二行，該格子還停留在 `null`。泛型拒絕協變，正是拒絕把這種錯誤留到執行期；需要「一個方

> 法收多種元素型別的清單」時，正解不是協變，而是上一節的萬用字元。

5.7　型別抹除

Java 的泛型靠**型別抹除**（type erasure）實作：型別參數只活在編譯期，編譯完成後被抹除——`Box<T>` 的 `T` 換成 `Object`（有界限則換成界限型別），需要之處自動補上轉型。好處是與 Java 5 之前的位元組碼完全相容；代價是執行期「看不到」型別參數：

範例：ErasureDemo.java

```
import java.util.ArrayList;
import java.util.List;

/** 型別抹除的可觀察證據 */
public class ErasureDemo {
    public static void main(String[] args) {
        List<String> strings = new ArrayList<>();
        List<Integer> integers = new ArrayList<>();

        // 執行期兩者是同一個類別物件：型別參數已被抹除
        System.out.println(strings.getClass() == integers.getClass());
        System.out.println(strings.getClass().getName());
    }
}
```

▶ 執行結果

```
true
java.util.ArrayList
```

> **說明**　型別抹除解釋了泛型的所有「不能」：**不能 `new T()`**（執行期不知道 `T` 是誰，生不出物件）；**不能建立泛型陣列** `new T[10]`（陣列必須在執行期知道元素型別以便檢查協變寫入，抹除後無從檢查）；**不能 `obj instanceof List<String>`**（執行期只剩 `List`，角括號內容早已消失，只能寫 `obj instanceof List<?>`）；**`static` 成員不能使用類別的型別參數**（static 成員屬於類別，而抹除後所有參數化版本共用同一個類別）。遇到這些限制時，繞法通常是改用 `List<T>` 取代陣列，或把 `Class<T>` 物件當參數傳入。

5.8 綜合範例：型別安全的泛型堆疊

最後把本章知識落地成一個實用的資料結構：堆疊（stack），後進先出（LIFO）。內部用 `ArrayList` 儲存（組合優於繼承，第 1 章），對外只暴露堆疊該有的四個操作：

範例：GenericStack 泛型堆疊

```
import java.util.ArrayList;
import java.util.List;

/**
 * 型別安全的泛型堆疊（LIFO：後進先出）。
 * 內部用 ArrayList 保存元素，尾端視為堆疊頂端。
 */
class GenericStack<E> {

    // 組合而非繼承：內部細節不外洩，公開介面完全由我們決定
    private final List<E> elements = new ArrayList<>();

    /** 推入：把元素放上堆疊頂端 */
    public void push(E element) {
        elements.add(element);           // 加到尾端就是頂端
    }

    /** 彈出：移除並回傳頂端元素 */
    public E pop() {
        if (isEmpty()) {
            // 空堆疊還要彈出是呼叫端的錯：拋 unchecked 例外
            throw new IllegalStateException("堆疊是空的");
        }
        return elements.remove(elements.size() - 1);
    }

    /** 偷看：回傳頂端元素但不移除 */
    public E peek() {
        if (isEmpty()) {
            throw new IllegalStateException("堆疊是空的");
        }
        return elements.get(elements.size() - 1);
    }

    /** 是否為空 */
```

```
    public boolean isEmpty() {
        return elements.isEmpty();
    }

    /** 目前元素個數 */
    public int size() {
        return elements.size();
    }
}
```

設計筆記：選擇「包一個 List」而不是「繼承 ArrayList」，是因為繼承會把 get、add(index, e) 等六十多個方法全部曝露出去，任何人都能從中間戳破堆疊的紀律；組合讓公開介面完全由我們決定——這正是第 1 章「繼承與組合」討論的實戰版。測試一下：

範例：GenericStackDemo.java

```
/** 測試泛型堆疊：同一份程式碼服務所有元素型別 */
public class GenericStackDemo {
    public static void main(String[] args) {
        // 模擬瀏覽器的「上一頁」歷史堆疊
        GenericStack<String> history = new GenericStack<>();
        history.push("首頁");
        history.push("商品列表");
        history.push("商品明細");

        System.out.println("目前頁面：" + history.peek());
        System.out.println("頁面數量：" + history.size());

        // 連續按兩次上一頁
        System.out.println("返回：" + history.pop());
        System.out.println("返回：" + history.pop());
        System.out.println("目前頁面：" + history.peek());

        // 換一種元素型別：不需要改動 GenericStack 的任何一行
        GenericStack<Integer> numbers = new GenericStack<>();
        numbers.push(10);
        numbers.push(20);
        System.out.println("整數堆疊頂端：" + numbers.pop());
        System.out.println("清空了嗎：" + numbers.isEmpty());
    }
}
```

▶ 執行結果

```
目前頁面：商品明細
頁面數量：3
返回：商品明細
返回：商品列表
目前頁面：首頁
整數堆疊頂端：20
清空了嗎：false
```

同一份 GenericStack 原始碼，前半段裝字串、後半段裝整數，兩邊都享有完整的編譯期型別檢查——這就是泛型的承諾：一次撰寫，處處型別安全。

5.9 本章小結

- 泛型的使命：把「放錯型別」的錯誤從執行期的 ClassCastException 提前為編譯期錯誤；raw type 僅為相容而存在，新程式碼禁用。
- 泛型類別在類別名後宣告型別參數；命名慣例 T、E、K、V、R。
- 泛型方法的型別參數宣告在回傳型別之前；編譯器通常能自動推斷，也可用 類別.<型別>方法() 明確指定。
- 有界型別參數 <T extends 界限> 對內保證能力可用、對外擋掉不合格型別；多重界限用 & 串接。
- 萬用字元：<?> 只讀不寫；<? extends T> 讀出保證是 T（生產者）；<? super T> 寫入 T 安全（消費者）。
- PECS 口訣：Producer Extends, Consumer Super；又讀又寫用精確型別。
- 泛型不變、陣列協變：陣列把錯誤留給執行期的 ArrayStoreException，泛型在編譯期就拒絕。
- 型別抹除：型別參數編譯後消失，因此不能 new T()、不能建泛型陣列、instanceof 不能帶型別參數、static 成員用不到型別參數。
- 設計泛型容器時優先用組合包住既有集合，只暴露該有的操作。

5.10 本章練習

動手練習

1. 觀念題：為什麼 List<String> 不能指派給 List<Object>，但 String[] 可以指派給 Object[]？兩種設計各把錯誤攔在什麼時機？（提示：回想

ArrayStoreException 範例。）

2. 實作題：寫一個泛型方法 `<T> int countMatches(List<T> list, T target)`，回傳清單中與目標 `equals` 相等的元素個數，並分別以字串清單與整數清單測試。（提示：泛型方法加 for-each。）
3. 實作題：寫一個泛型方法 `<T extends Comparable<T>> T min(List<T> list)`，回傳清單最小值；清單為空時拋出 `IllegalArgumentException`。（提示：模仿本章的 `max` 三值版本改成走訪版本。）
4. 觀念題：方法簽章 `void addAll(List<? extends Number> dst)` 哪裡設計錯了？依 PECS 應該怎麼改？（提示：dst 是被寫入的一方，它是生產者還是消費者？）
5. 實作題：為 `GenericStack` 新增方法 `void pushAll(List<? extends E> items)`（把清單全部推入）與 `void popAll(List<? super E> dst)`（全部彈出並放入目的清單），體會 PECS 在真實 API 的樣貌。（提示：兩個萬用字元各用一次。）
6. 挑戰題：實作泛型類別 `Pair<K, V>` 的靜態工廠方法 `static <K, V> Pair<K, V> of(K key, V value)`，並解釋為什麼 static 方法必須自己宣告型別參數、不能借用類別的 K 與 V。（提示：型別抹除後 static 成員屬於誰？）

第 6 章

檔案與 I/O

本章學習目標

學完本章你將能夠：

- 分辨傳統 java.io 與現代 java.nio.file 的分工，用 Path 與 Files 操作檔案。
- 依「方向」與「單位」為 I/O 流分類，說出四個基底抽象類別。
- 用位元組流搭配緩衝區複製檔案，解釋緩衝流的效能意義。
- 用字元流讀寫文字檔，並明確指定 UTF-8 編碼避免亂碼。
- 以 try-with-resources 管理所有 I/O 資源。
- 序列化與反序列化物件，說明 serialVersionUID 與 transient 的作用。

到目前為止，程式的所有資料都活在記憶體裡，程式一結束就煙消雲散。要讓資料活得比程式久，就得寫進檔案；要處理外部世界的資料，就得從檔案讀進來。本章的主角是 Java 的 I/O（Input/Output）體系：從檔案的中繼資料操作、位元組與字元兩套流、緩衝的效能原理，到物件序列化，最後用一個 CSV 成績單處理程式串起全部。

Java 的檔案 API 有兩個世代並存：`java.io` 是 1.0 時代的元老（`File` 類別與各種流），`java.nio.file` 是 Java 7 引入的現代化改版（`Path`、`Files`）。實務準則很簡單：**路徑與檔案管理用新版，資料的流式讀寫兩代並用**——新版的工具方法內部包的正是舊版的流。

6.1 檔案的中繼資料：File 與 Path

`java.io.File` 物件代表的是一個路徑，不是檔案內容——建立 `File` 物件不會建立檔案。現代寫法把「路徑」與「操作」拆開：`Path` 是路徑的抽象（用工廠方法 `Path.of` 取得），`Files` 是操作的靜態工具箱。兩代對照：

範例：FileInfoDemo.java

```
import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

```

```java
/** File（傳統）與 Path、Files（現代）的對照 */
public class FileInfoDemo {
    public static void main(String[] args) throws IOException {
        // 先用現代 API 建立測試檔（內容是 16 個 ASCII 字元）
        Path path = Path.of("demo.txt");
        Files.writeString(path, "Hello, File I/O!",
                StandardCharsets.UTF_8);

        // ---- 傳統 java.io.File：物件代表「路徑」而非內容 ----
        File file = new File("demo.txt");
        System.out.println("存在嗎？" + file.exists());
        System.out.println("是目錄嗎？" + file.isDirectory());
        System.out.println("大小：" + file.length() + " 位元組");

        // ---- 現代 java.nio.file：Path 表路徑，Files 提供操作 ----
        System.out.println("存在嗎？" + Files.exists(path));
        System.out.println("大小：" + Files.size(path) + " 位元組");
        System.out.println("檔名：" + path.getFileName());

        Files.delete(path);   // 收尾：刪除測試檔
        System.out.println("刪除後還存在嗎？" + Files.exists(path));
    }
}
```

▶ 執行結果

```
存在嗎？true
是目錄嗎？false
大小：16 位元組
存在嗎？true
大小：16 位元組
檔名：demo.txt
刪除後還存在嗎？false
```

留意 `main` 簽章上的 `throws IOException`：檔案操作幾乎都會拋這個受檢例外（第 3 章），示範程式直接上拋，正式程式則應該視情境處理。傳統 `File` 仍常見於舊程式與部分 API（例如 `listFiles()` 列目錄），但新程式建議以 `Path` 為中心。

6.2　I/O 流的分類

Java 把所有輸入輸出抽象成流（stream）：資料像水流一樣，從來源流進程式（輸入），或從程式流向目的地（輸出）。所有流類別依兩個維度分類——**方向**（輸入或輸出）與**單位**（位元組

或字元），交叉出四個基底抽象類別：

	位元組流（處理任何檔案）	字元流（只處理文字）
輸入	`InputStream`	`Reader`
輸出	`OutputStream`	`Writer`

所有具體流類別都是這四位始祖的後代，名字直接透露血統：`FileInputStream` 是「檔案的位元組輸入流」，`BufferedReader` 是「帶緩衝的字元輸入流」。選邊的準則：圖片、壓縮檔、執行檔等二進位資料用位元組流；文字內容用字元流，讓編碼轉換有人代勞。

6.3 位元組流與緩衝

位元組流的代表任務是檔案複製。核心模式值得背下來：準備一塊緩衝區陣列，`read` 回傳「實際讀到的位元組數」，讀到檔案結尾（EOF）時回傳 `-1`：

範例：FileCopyDemo.java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

/** 用位元組流複製檔案：一次搬一桶（緩衝區），不是一滴一滴搬 */
public class FileCopyDemo {

    /** 核心邏輯：來源讀進緩衝區，再寫到目的地 */
    static void copy(String from, String to) throws IOException {
        // try-with-resources：兩個流都保證會被關閉
        try (FileInputStream in = new FileInputStream(from);
             FileOutputStream out = new FileOutputStream(to)) {

            byte[] buffer = new byte[4096];   // 4 KB 緩衝區
            int count;                        // 本輪實際讀到的位元組數

            // read 回傳 -1 代表已到檔案結尾（EOF）
            while ((count = in.read(buffer)) != -1) {
                // 只寫出「實際讀到」的部分：最後一輪常常裝不滿
                out.write(buffer, 0, count);
            }
        }
    }
```

```
26
27      public static void main(String[] args) throws IOException {
28          // 準備一個 10000 位元組的來源檔
29          Path source = Path.of("source.bin");
30          Files.write(source, new byte[10000]);
31
32          copy("source.bin", "target.bin");      // 執行複製
33
34          System.out.println("來源大小："
35                  + Files.size(Path.of("source.bin")) + " 位元組");
36          System.out.println("目標大小："
37                  + Files.size(Path.of("target.bin")) + " 位元組");
38
39          Files.delete(Path.of("source.bin"));  // 清理測試檔
40          Files.delete(Path.of("target.bin"));
41      }
42  }
```

▶ 執行結果

```
來源大小：10000 位元組
目標大小：10000 位元組
```

兩個必懂細節：第一，`write(buffer, 0, count)` 只寫出實際讀到的部分——最後一輪緩衝區通常裝不滿，寫整塊會混入上一輪的舊資料，這是檔案複製最常見的臭蟲。第二，資源宣告在 try 的小括號中，無論成功失敗都會自動關閉（第 3 章的 try-with-resources），本章所有範例一體適用。

> **提示**　每一次 `read` 與 `write` 呼叫底層都可能是一次系統呼叫，成本高昂：一個位元組一個位元組地讀寫，速度可能差上百倍。兩個對策擇一即可：像本例**自備緩衝區陣列**；或把流包進**緩衝流**——`BufferedInputStream` 與 `BufferedOutputStream` 內建一塊緩衝區，攔下大部分的實體讀寫，例如 `new BufferedInputStream(new FileInputStream("a.bin"))`。這種「用一個流包住另一個流、加上新能力」的手法是 I/O 體系的招牌設計（裝飾者模式，進階篇詳述）。習慣上：**只要是檔案流，先包一層緩衝再說**。

6.4　字元流與編碼

文字檔的本質仍是位元組，「字元如何變成位元組」由**編碼**（encoding）決定：UTF-8 之下一個中文字佔 3 個位元組，Big5 之下佔 2 個。字元流（`Reader` 與 `Writer`）替你做這層轉換，但**用哪種編碼必須說清楚**。

> **注意**　歷史上 `FileReader` 與 `FileWriter` 的無參數建構子使用「平台預設編碼」——同一個檔案在 Windows（傳統中文版預設 MS950）寫入、在 macOS 或 Linux（預設 UTF-8）讀取，中文就成了亂碼。這類問題往往在跨機器部署時才爆發，極難回溯。鐵律：**讀寫文字檔一律明確指定編碼**，首選 `StandardCharsets.UTF_8`。Java 11 起 `FileReader` 有了帶 `Charset` 參數的建構子，Java 18 起平台預設編碼統一為 UTF-8，但「明講」永遠是最保險的作法。

實務上更常用 `Files` 的工廠方法直接取得「帶緩衝、指定編碼」的字元流。`BufferedReader` 的招牌方法是 `readLine()`：一次讀一行（不含換行符號），讀到結尾回傳 `null`：

範例：TextFileDemo.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;

/** 字元流讀寫文字檔：務必明確指定 UTF-8 編碼 */
public class TextFileDemo {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("notes.txt");

        // 寫入：newBufferedWriter 建立「緩衝 + UTF-8」的字元流
        try (BufferedWriter writer =
                Files.newBufferedWriter(path, StandardCharsets.UTF_8)) {
            writer.write("第一行：Java 檔案處理");
            writer.newLine();                  // 跨平台的換行寫法
            writer.write("第二行：中文需要正確編碼");
            writer.newLine();
            writer.write("第三行：完");
            writer.newLine();
        }   // 離開 try 時自動 flush 並關閉

        // 讀取：readLine 一次讀一行，讀到結尾回傳 null
        try (BufferedReader reader =
                Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
            String line;
            int lineNo = 1;
```

```
            while ((line = reader.readLine()) != null) {
                System.out.println(lineNo + " | " + line);
                lineNo++;                       // 行號遞增
            }
        }

        Files.delete(path);                     // 清理測試檔
    }
}
```

▶ 執行結果

```
1 | 第一行：Java 檔案處理
2 | 第二行：中文需要正確編碼
3 | 第三行：完
```

`while ((line = reader.readLine()) != null)` 這個「邊讀邊判斷」的慣用句型，是 Java 文字處理的標準開場白。輸出端除了 `BufferedWriter`，還有一位老朋友 `PrintWriter`——它提供你熟悉的 `println` 與 `printf`，適合產生格式化報表，本章結尾的綜合範例會用到。

6.5　NIO.2 的 Files 工具箱

小檔案的讀寫，`Files` 的便利方法一行就能解決，內部自動開流、自動關閉、預設 UTF-8：

範例：FilesToolboxDemo.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardCopyOption;
import java.util.List;

/** NIO.2 的 Files 工具方法：一行搞定常見檔案操作 */
public class FilesToolboxDemo {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("todo.txt");

        // 一行寫入多行文字（自動加換行，預設 UTF-8）
        Files.write(path, List.of("買牛奶", "繳電費", "回覆信件"));

        // readString：整個檔案讀成一個字串（小檔案最方便）
        String all = Files.readString(path);
```

```
        System.out.println("字元總數：" + all.length());

        // readAllLines：整個檔案讀成 List<String>
        List<String> lines = Files.readAllLines(path);
        System.out.println("共 " + lines.size() + " 行，第二行是："
                + lines.get(1));

        // copy：複製檔案（REPLACE_EXISTING：目標存在就覆蓋）
        Path backup = Path.of("todo-backup.txt");
        Files.copy(path, backup, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("備份存在：" + Files.exists(backup));

        Files.delete(path);      // delete：刪除檔案
        Files.delete(backup);
        System.out.println("清理完成：" + !Files.exists(path));
    }
}
```

▶ 執行結果

```
字元總數：13
共 3 行，第二行是：繳電費
備份存在：true
清理完成：true
```

驗算第一行輸出：三行各為 3、3、4 個中文字，加上三個換行符號，共 3+1+3+1+4+1 = 13 個字元。要注意 `readString` 與 `readAllLines` 會把整個檔案載入記憶體，只適合小檔案；大檔案請回到 `BufferedReader` 逐行處理，或用 `Files.lines`（回傳逐行的 `Stream<String>`，進階篇詳述）。搬移與刪除則有 `Files.move` 與 `Files.delete`。走訪目錄的現代寫法如下：

範例：ListDirDemo.java

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.stream.Stream;

/** 走訪目錄內容 */
public class ListDirDemo {
    public static void main(String[] args) throws IOException {
        // 建立測試目錄與三個檔案
        Path dir = Files.createDirectories(Path.of("data-dir"));
        Files.writeString(dir.resolve("a.txt"), "A");
```

```
        Files.writeString(dir.resolve("b.csv"), "B");
        Files.writeString(dir.resolve("c.log"), "C");

        // Files.list 回傳 Stream<Path>（Stream 於進階篇詳述）
        // 作業系統列出的順序不保證，先排序讓輸出可預測
        try (Stream<Path> entries = Files.list(dir)) {
            entries.sorted().forEach(p ->
                    System.out.println("發現檔案：" + p.getFileName()));
        }

        // 清理：先刪檔案再刪目錄（目錄必須是空的才刪得掉）
        Files.delete(dir.resolve("a.txt"));
        Files.delete(dir.resolve("b.csv"));
        Files.delete(dir.resolve("c.log"));
        Files.delete(dir);
        System.out.println("清理完成");
    }
}
```

▶ 執行結果

```
發現檔案：a.txt
發現檔案：b.csv
發現檔案：c.log
清理完成
```

`dir.resolve("a.txt")` 把子路徑接到目錄後面，是拼接路徑的正規方式（比手動串字串可靠，不必煩惱斜線）。傳統寫法 `file.listFiles()` 回傳 `File[]`，在舊程式碼中仍隨處可見。注意 `Files.list` 回傳的 Stream 也要用 try-with-resources 關閉，它底層握著作業系統的目錄資源。

6.6 物件序列化

前面存的都是文字與位元組，能不能把**整個物件**直接存檔？可以——**序列化**（serialization）把物件轉成位元組串流，**反序列化**把它還原。只要類別實作 `Serializable` 這個**標記介面**（沒有任何方法，純粹表態「我可以被序列化」），就能交給 `ObjectOutputStream` 與 `ObjectInputStream` 處理：

範例：SerializationDemo.java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.nio.file.Files;
import java.nio.file.Path;

/** 可序列化的會員檔案：實作 Serializable 這個「標記介面」 */
class Profile implements Serializable {

    // 序列化版本號：類別演進時用來判斷新舊版本是否相容
    private static final long serialVersionUID = 1L;

    private final String name;    // 會被寫入檔案
    private final int level;      // 會被寫入檔案

    // transient：敏感或可重建的欄位，序列化時直接跳過
    private transient String sessionToken;

    Profile(String name, int level, String sessionToken) {
        this.name = name;
        this.level = level;
        this.sessionToken = sessionToken;
    }

    @Override
    public String toString() {
        return "Profile{name=" + name + ", level=" + level
                + ", sessionToken=" + sessionToken + "}";
    }
}

/** 序列化與反序列化的完整流程 */
public class SerializationDemo {
    public static void main(String[] args)
            throws IOException, ClassNotFoundException {

        Profile original = new Profile("小明", 5, "TOKEN-9527");
        System.out.println("存檔前：" + original);

        // 序列化：把物件轉成位元組寫入檔案
        try (ObjectOutputStream out = new ObjectOutputStream(
```

```
                new FileOutputStream("profile.ser"))) {
            out.writeObject(original);
        }

        // 反序列化：從位元組還原物件（readObject 回傳 Object，需轉型）
        Profile restored;
        try (ObjectInputStream in = new ObjectInputStream(
                new FileInputStream("profile.ser"))) {
            restored = (Profile) in.readObject();
        }

        // transient 欄位不會被還原：參考型別會是 null
        System.out.println("還原後：" + restored);

        Files.delete(Path.of("profile.ser"));   // 清理
    }
}
```

▶ 執行結果

```
存檔前：Profile{name=小明, level=5, sessionToken=TOKEN-9527}
還原後：Profile{name=小明, level=5, sessionToken=null}
```

`transient` 修飾的欄位不參與序列化：適合放密碼、憑證等不該落地的敏感資料，或快取這類可重建的資料——還原後參考型別欄位是 `null`、數值型別是 0，由程式自行補建。序列化會連同整張物件圖（欄位參考到的物件）一起處理，因此那些物件也必須可序列化。

> **說明**　`serialVersionUID` 是類別的版本指紋。反序列化時 JVM 會比對「檔案裡記錄的版本號」與「目前類別的版本號」，不一致就拋出 `InvalidClassException`。若不明確宣告，編譯器會依類別結構自動產生一個——只要類別改了一個欄位，自動值就變，舊檔案立刻全部讀不回來。因此凡是實作 `Serializable` 的類別，都應該**手動宣告** `serialVersionUID`，並在確認新舊版本相容時保持不變。另外提醒：Java 原生序列化格式綁定 JVM、跨語言不通，對外交換資料實務上多改用 JSON 或 CSV 等文字格式——下一節就是示範。

6.7　綜合範例：CSV 成績檔處理

收尾範例模擬真實工作流程：讀取 CSV（逗號分隔）成績檔、解析成物件、計算統計，再把報表寫成另一個檔案。讀取用 `BufferedReader`、寫入用 `PrintWriter`、解析用 `String.split`、統計用集合，一次動員三章的知識：

範例：GradeReport.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

/**
 * CSV 成績檔處理：讀取、解析、統計、輸出報表。
 * 整合本章的檔案讀寫與第 4 章的集合知識。
 */
public class GradeReport {

    /** 成績紀錄：一列 CSV 對應一個物件 */
    static class Record {
        final String name;
        final int score;

        Record(String name, int score) {
            this.name = name;
            this.score = score;
        }
    }

    public static void main(String[] args) throws IOException {
        Path csv = Path.of("scores.csv");
        Path report = Path.of("report.txt");

        // 步驟零：準備測試用的 CSV 檔（實務上檔案已經存在）
        Files.write(csv, List.of(
                "姓名,分數",
                "小明,88",
                "小華,95",
                "小強,61",
                "小美,73"));

        // 步驟一：逐行讀取並解析成物件
        List<Record> records = new ArrayList<>();
        try (BufferedReader reader =
```

```
                Files.newBufferedReader(csv, StandardCharsets.UTF_8)) {
            String line = reader.readLine();      // 先讀掉標題列
            while ((line = reader.readLine()) != null) {
                String[] cols = line.split(",");  // 以逗號切出欄位
                records.add(new Record(cols[0],
                        Integer.parseInt(cols[1])));
            }
        }

        // 步驟二：計算統計數字
        int total = 0;
        int highest = Integer.MIN_VALUE;
        String topStudent = "";
        for (Record r : records) {
            total += r.score;
            if (r.score > highest) {   // 追蹤最高分
                highest = r.score;
                topStudent = r.name;
            }
        }
        double average = (double) total / records.size();

        // 步驟三：把報表寫入檔案（PrintWriter 提供 printf 等便利方法）
        try (PrintWriter writer = new PrintWriter(
                Files.newBufferedWriter(report, StandardCharsets.UTF_8))) {
            writer.println("成績統計報表");
            writer.println("人數：" + records.size());
            writer.printf("平均：%.2f%n", average);
            writer.println("最高分：" + topStudent
                    + "（" + highest + " 分）");
        }

        // 步驟四：把報表讀回來印到螢幕，確認結果
        for (String line : Files.readAllLines(report)) {
            System.out.println(line);
        }

        Files.delete(csv);       // 清理測試檔
        Files.delete(report);
    }
}
```

▶ 執行結果

```
成績統計報表
人數：4
平均：79.25
最高分：小華（95 分）
```

驗算：總分 $88+95+61+73=317$，平均 $317\div 4=79.25$，最高分小華 95。幾個實戰要點：讀掉標題列的技巧是「先呼叫一次 `readLine` 不使用其值」；`(double) total` 的強制轉型避免整數除法把小數吃掉；`PrintWriter` 包住 `Files.newBufferedWriter` 的組合同時獲得緩衝、UTF-8 與 `printf` 三種好處。真實世界的 CSV 還有引號包裹、欄位內含逗號等變化，屆時建議使用成熟的解析函式庫，但「逐行讀、切欄位、轉型別」的骨架不變。

6.8 本章小結

- 兩代 API 分工：`java.io` 提供流；`java.nio.file` 的 `Path` 表路徑、`Files` 提供現代化操作；新程式以 `Path` 為中心。
- 四個基底抽象類別：位元組流 `InputStream` 與 `OutputStream`；字元流 `Reader` 與 `Writer` ↪ 。二進位用位元組流，文字用字元流。
- 檔案複製的標準模式：緩衝區陣列加 `while ((count = in.read(buffer)) != -1)`，寫出時務必只寫 `count` 個位元組。
- `read` 回傳 -1 代表檔案結尾；`readLine` 回傳 `null` 代表讀完。
- 檔案流務必包緩衝（`BufferedInputStream` 等）或自備緩衝區，避免逐位元組的系統呼叫。
- 文字檔一律明確指定編碼（首選 `StandardCharsets.UTF_8`），依賴平台預設編碼是亂碼的頭號來源。
- 所有 I/O 資源都交給 try-with-resources 自動關閉。
- `Files` 工具箱：`readString`、`readAllLines`、`write`、`copy`、`move`、`delete`、`exists`、`list`；整檔載入的方法只適合小檔案。
- 序列化：實作 `Serializable`、用 `ObjectOutputStream` 存檔；`transient` 欄位不落地；務必手動宣告 `serialVersionUID`。

6.9 本章練習

動手練習

1. 觀念題：複製圖片檔時誤用字元流（`FileReader` 加 `FileWriter`），檔案會發生什麼事？為什麼？（提示：字元流會對位元組做編碼解讀，二進位資料不是合法文字。）
2. 觀念題：`out.write(buffer)` 與 `out.write(buffer, 0, count)` 差在哪裡？什麼情況下前者會寫出錯誤的資料？（提示：最後一輪讀取通常裝不滿緩衝區。）
3. 實作題：寫一個程式統計指定文字檔的行數、非空白行數與總字元數，檔案不存在時給出友善的錯誤訊息而非堆疊追蹤。（提示：`Files.exists` 先檢查，或捕獲 `IOException`。）
4. 實作題：寫一個「日記程式」：每次執行從鍵盤讀一段文字，連同當天日期附加到 diary.txt 末尾（不覆蓋既有內容）。（提示：`Files.newBufferedWriter` 可加參數 `StandardOpenOption.APPEND` 與 `CREATE`。）
5. 實作題：讓第 4 章的 `Member` 類別實作 `Serializable`，把一個 `ArrayList<Member>` 整批序列化到檔案再讀回來，驗證內容一致。（提示：集合本身就是可序列化的，直接 `writeObject` 整個清單。）
6. 挑戰題：擴充本章的 `GradeReport`：改成從命令列引數接收輸入檔名，加入「不及格名單」段落，並處理「分數欄位不是數字」的髒資料列（跳過並記錄警告，不讓程式中斷）。（提示：對 `Integer.parseInt` 包 try-catch，第 3 章。）

第 7 章

列舉、內部類別與常用 API

本章學習目標

學完本章你將能夠：

- 用 enum 取代整數常數，寫出帶欄位、方法與各常數專屬行為的列舉。
- 分辨四種巢狀類別（靜態巢狀、內部、區域、匿名）與各自的使用時機。
- 用匿名類別現場實作介面，為進階篇的 lambda 鋪路。
- 熟用 `Math` 與 `Random`，寫出正確的範圍亂數。
- 用 `java.time` 處理日期時間：計算、格式化、解析。
- 綜合以上工具實作一個訂單狀態機。

前六章打通了物件導向與核心容器，本章補齊三塊日常開發天天用到的拼圖：列舉（enum）讓「固定的一組選項」型別安全；巢狀類別讓輔助類別待在該待的地方；再加上 `Math`、`Random` 與 `java.time` 這些標準 API——第 8 章的專案會同時用上它們（訂單狀態、借閱日期都在等著）。

7.1 enum 基礎：告別整數常數

需要表達「星期幾」時，老派做法是 `public static final int MON = 0;` 一路排下去。問題在型別不安全：參數宣告成 `int`，傳 99 或「星期一加星期二」這種無意義運算，編譯器一律放行。`enum` 把選項升格為獨立型別：值只能是列舉出的那幾個，錯誤在編譯期全數攔截。

範例：EnumBasicsDemo.java ——enum 宣告與內建方法

```
/** 星期列舉：一組有名字的、型別安全的常數 */
enum Weekday {
    MON, TUE, WED, THU, FRI, SAT, SUN
}

public class EnumBasicsDemo {
    public static void main(String[] args) {
        Weekday today = Weekday.SAT;

        System.out.println("today = " + today);   // 自動印常數名
```

```
        System.out.println("名稱：" + today.name());
        System.out.println("序號：" + today.ordinal());  // 從 0 起算
        System.out.println("解析：" + Weekday.valueOf("MON"));

        // values()：取得全部常數，順序同宣告順序
        for (Weekday d : Weekday.values()) {
            System.out.print(d + " ");
        }
        System.out.println();

        // enum 進 switch：case 直接寫常數名，不必加型別前綴
        switch (today) {
            case SAT, SUN -> System.out.println("週末：睡到自然醒");
            case MON -> System.out.println("星期一：收心上工");
            default -> System.out.println("平日：好好上課");
        }
    }
}
```

▶ 執行結果

```
today = SAT
名稱：SAT
序號：5
解析：MON
MON TUE WED THU FRI SAT SUN
週末：睡到自然醒
```

三個內建成員最常用：`values()` 回傳全部常數的陣列、`name()` 回傳常數名字串、`valueOf("MON")` 從字串解析回常數（找不到會拋 `IllegalArgumentException`）。`ordinal()` 回傳宣告順序，**但不要拿它當商業邏輯的數值**——哪天有人在中間插一個常數，所有序號全部位移。範例中的 switch 使用 Java 14 的箭頭語法：不會貫穿（fall-through）、一個 case 可列多個常數，建議一律採用。

7.2　帶欄位與方法的 enum

Java 的 enum 遠不只常數清單——它是**功能完整的類別**：可以有欄位、建構子與方法；每個常數都是該類別的一個實體。經典示範是行星：每顆行星帶著自己的質量與半徑，還能算表面重力：

範例：PlanetDemo.java ——帶資料與行為的列舉

```java
/** 行星列舉：每個常數自帶質量（公斤）與半徑（公尺） */
enum Planet {
    // 宣告常數時以括號呼叫建構子
    MERCURY(3.303e23, 2.4397e6),
    EARTH(5.976e24, 6.37814e6),
    MARS(6.421e23, 3.3972e6);

    private final double mass;      // 質量
    private final double radius;    // 半徑

    /** 萬有引力常數 */
    private static final double G = 6.67300E-11;

    /** enum 的建構子：只在載入常數時被呼叫 */
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    /** 表面重力加速度：G 乘質量除以半徑平方 */
    double surfaceGravity() {
        return G * mass / (radius * radius);
    }

    /** 質量 otherMass 的物體在該行星上的重量（牛頓） */
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}

public class PlanetDemo {
    public static void main(String[] args) {
        double myMass = 70.0;    // 一位 70 公斤的太空旅人
        for (Planet p : Planet.values()) {
            System.out.printf("在 %s 上的重量：%.1f 牛頓%n",
                p, p.surfaceWeight(myMass));
        }
    }
}
```

▶ 執行結果

```
在 MERCURY 上的重量：259.2 牛頓
在 EARTH 上的重量：686.2 牛頓
在 MARS 上的重量：259.9 牛頓
```

> **說明**　enum 的建構子隱含 **private**（寫 public 會編譯錯誤）：實體只能是宣告的那幾個常數，外界永遠不能 new Planet(...)。這保證了「全系統只有這幾個實體」，也因此 enum 可以放心用 == 比較，天生就是完美的單例。

更進一步，enum 可以宣告**抽象方法**，強迫每個常數提供自己的實作（語法：常數名後面接大括號的「常數專屬類別本體」）——第 2 章的多型在列舉身上重演：

範例：OperationDemo.java ——每個常數各自實作

```
/** 四則運算列舉：每個常數自帶計算邏輯 */
enum Operation {
    PLUS("+") {
        @Override
        public double apply(double a, double b) {
            return a + b;
        }
    },
    MINUS("-") {
        @Override
        public double apply(double a, double b) {
            return a - b;
        }
    },
    TIMES("*") {
        @Override
        public double apply(double a, double b) {
            return a * b;
        }
    },
    DIVIDE("/") {
        @Override
        public double apply(double a, double b) {
            return a / b;
        }
    };

    private final String symbol;    // 運算符號
```

```
29
30      Operation(String symbol) {
31          this.symbol = symbol;
32      }
33
34      /** 抽象方法：每個常數都「必須」提供自己的版本 */
35      public abstract double apply(double a, double b);
36
37      public String getSymbol() {
38          return symbol;
39      }
40  }
41
42  public class OperationDemo {
43      public static void main(String[] args) {
44          double a = 12, b = 4;
45          for (Operation op : Operation.values()) {
46              // op.apply(...) 依實際常數執行不同邏輯：多型！
47              System.out.printf("%.0f %s %.0f = %.1f%n",
48                  a, op.getSymbol(), b, op.apply(a, b));
49          }
50      }
51  }
```

▶ 執行結果

```
12 + 4 = 16.0
12 - 4 = 8.0
12 * 4 = 48.0
12 / 4 = 3.0
```

比起在一個方法裡 switch 判斷運算子，這種寫法把「每種運算的邏輯」關進各自的常數：新增 `POWER` 時編譯器會強迫你實作 `apply`，永遠不會漏掉分支。

7.2.1　EnumSet 與 EnumMap

當集合的元素或鍵是列舉時，別用泛用的 `HashSet` 與 `HashMap`——標準庫有專屬的高效版本：`EnumSet` 底層是位元向量（一個 `long` 就裝得下 64 個常數的有無），`EnumMap` 底層是依 `ordinal` 索引的陣列，兩者都比雜湊結構更快、更省記憶體，且遍歷順序固定為宣告順序：

範例：EnumCollectionsDemo.java ——列舉專屬集合

```
1  import java.util.EnumMap;
```

```
import java.util.EnumSet;
import java.util.Map;

/** EnumSet 與 EnumMap：搭配 EnumBasicsDemo 的 Weekday */
public class EnumCollectionsDemo {
    public static void main(String[] args) {
        // range：取宣告順序中的一段
        EnumSet<Weekday> workdays =
            EnumSet.range(Weekday.MON, Weekday.FRI);
        // complementOf：取補集（其餘常數）
        EnumSet<Weekday> weekend = EnumSet.complementOf(workdays);

        System.out.println("工作日：" + workdays);
        System.out.println("週末：" + weekend);

        // EnumMap：鍵是列舉的 Map，建構時要提供類別資訊
        Map<Weekday, String> plan = new EnumMap<>(Weekday.class);
        plan.put(Weekday.SAT, "爬山");
        plan.put(Weekday.SUN, "寫程式");
        System.out.println("週末計畫：" + plan);
    }
}
```

▶ 執行結果

```
工作日：[MON, TUE, WED, THU, FRI]
週末：[SAT, SUN]
週末計畫：{SAT=爬山, SUN=寫程式}
```

7.3 巢狀類別四人組

類別可以宣告在另一個類別、甚至方法裡面，統稱**巢狀類別**。四種形態一次看完：

範例：NestedDemo.java ——四種巢狀類別

```
/** 給匿名類別實作用的小介面 */
interface Greeter {
    void greet();
}

/** 四種巢狀類別的最小示範 */
public class NestedDemo {
```

```
8
9       private String outerField = "外部類別的欄位";
10
11      /** 一、靜態巢狀類別：不持有外部實體，最常用 */
12      static class StaticNested {
13          String hello() {
14              return "靜態巢狀類別：與外部實體無關";
15          }
16      }
17
18      /** 二、內部類別：隱含持有一份外部類別的 this */
19      class Inner {
20          String hello() {
21              // 可以直接讀外部類別的 private 欄位
22              return "內部類別讀到：" + outerField;
23          }
24      }
25
26      void localAndAnonymous() {
27          // 三、區域類別：宣告在方法裡，只有本方法看得到
28          class Local {
29              String hello() {
30                  return "區域類別：方法內的小幫手";
31              }
32          }
33          System.out.println(new Local().hello());
34
35          // 四、匿名類別：宣告與 new 一次完成，連名字都省了
36          Greeter greeter = new Greeter() {
37              @Override
38              public void greet() {
39                  System.out.println("匿名類別：一次性的實作");
40              }
41          };
42          greeter.greet();
43      }
44
45      public static void main(String[] args) {
46          // 靜態巢狀類別：直接 new，不需要外部實體
47          System.out.println(new StaticNested().hello());
48
49          // 內部類別：必須先有外部實體，語法是 outer.new Inner()
```

```
        NestedDemo outer = new NestedDemo();
        System.out.println(outer.new Inner().hello());

        outer.localAndAnonymous();
    }
}
```

▶ 執行結果

```
靜態巢狀類別：與外部實體無關
內部類別讀到：外部類別的欄位
區域類別：方法內的小幫手
匿名類別：一次性的實作
```

種類	宣告位置	使用時機
靜態巢狀類別	類別內，加 `static`	只服務外部類別、不需外部狀態
內部類別	類別內，不加 `static`	需要直接存取外部實體的欄位
區域類別	方法內	只在單一方法內使用的小類別
匿名類別	運算式中	一次性實作介面或抽象類別

> **說明**　內部類別（非靜態）的每個實體都**偷偷抓著一份外部物件的參考**。這有兩個後果：一、沒有外部實體就造不出內部實體；二、只要內部實體活著，外部物件就無法被垃圾回收——在長生命週期場景（快取、監聽器）這是知名的記憶體洩漏來源。守則：**用不到外部實體狀態時，一律宣告成靜態巢狀類別**。第 4 章 `Map.Entry`、第 8 章專案的輔助類別都遵循這條守則。

7.4　匿名類別實作 Comparator

匿名類別最常見的用途，是替「只用一次」的介面實作省去獨立檔案。第 4 章排序用的 `Comparator` 是絕佳示範：

範例：AnonymousComparatorDemo.java ——現場實作比較器

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

/** 用匿名類別提供「依字串長度排序」的規則 */
public class AnonymousComparatorDemo {
```

```
    public static void main(String[] args) {
        List<String> cities = new ArrayList<>(List.of(
            "Kaohsiung", "Taipei", "Tainan", "Hsinchu"));

        // new 介面名() { ... }：宣告兼實體化，一步到位
        Collections.sort(cities, new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                // 短的排前面；同長度維持原順序（穩定排序）
                return Integer.compare(a.length(), b.length());
            }
        });

        System.out.println(cities);
    }
}
```

▶ 執行結果

```
[Taipei, Tainan, Hsinchu, Kaohsiung]
```

你可能覺得這語法囉嗦——沒錯。第 4 章已偷跑過的 lambda 寫法 `(a, b) -> Integer.compare(a.length(), b.length())` 就是它的精簡形：對「只有一個抽象方法的介面」，編譯器允許只寫方法本體。進階篇會完整拆解 lambda 與方法參考；現在先記住：**看到匿名類別實作單方法介面，心裡自動翻譯成 lambda**，讀懂新舊程式碼都不成問題。

7.5　Math 與 Random

`Math` 是純 `static` 工具類別，常用方法整理如下：

方法	功能	備註
`abs(x)`	絕對值	各數值型別皆有多載
`max(a, b)`、`min(a, b)`	較大、較小值	
`pow(a, b)`	a 的 b 次方	回傳 `double`
`sqrt(x)`	平方根	回傳 `double`
`round(x)`	四捨五入	`double` 版回傳 `long`
`ceil(x)`、`floor(x)`	無條件進位、捨去	回傳 `double`
`random()`	0.0 到 1.0（不含）亂數	簡便但不建議當主力

產生亂數的正式工具是 `java.util.Random`。範圍亂數的正確公式：`rand.nextInt(max - min + 1) + min` 產生 min 到 max（含兩端）的整數：

範例：MathRandomDemo.java ——Math 方法與範圍亂數

```java
import java.util.Random;

/** Math 常用方法與 Random 的正確用法 */
public class MathRandomDemo {
    public static void main(String[] args) {
        // --- Math：static 方法直接呼叫，不必建立物件 ---
        System.out.println("絕對值：" + Math.abs(-7));
        System.out.println("四捨五入：" + Math.round(3.56));
        System.out.println("無條件進位：" + Math.ceil(3.14));
        System.out.println("無條件捨去：" + Math.floor(3.99));
        System.out.println("次方：" + Math.pow(2, 10));
        System.out.println("平方根：" + Math.sqrt(144));
        System.out.println("較大值：" + Math.max(17, 23));

        // --- Random：擲骰子一萬次，驗證範圍公式 ---
        Random rand = new Random();
        int min = 7, max = 0;
        for (int i = 0; i < 10000; i++) {
            // nextInt(6) 給 0~5，加 1 平移成骰子的 1~6
            int dice = rand.nextInt(6) + 1;
            min = Math.min(min, dice);
            max = Math.max(max, dice);
        }
        System.out.println("擲一萬次：最小 " + min + "、最大 " + max);
    }
}
```

▶ 執行結果

```
絕對值：7
四捨五入：4
無條件進位：4.0
無條件捨去：3.0
次方：1024.0
平方根：12.0
較大值：23
擲一萬次：最小 1、最大 6
```

單次骰子點數當然每次執行都不同，所以範例改印一萬次的極值來驗證「範圍公式」正確。兩個補充：測試想要可重現的亂數序列，建構時給固定種子 `new Random(42)`；多執行緒環境改用 `ThreadLocalRandom.current()`（進階篇說明原因）。

7.6 日期時間：java.time

> **說明**　Java 8 之前的 `Date` 與 `Calendar` 是公認的設計失誤：可變（到處被偷改）、月份從 0 起算（十月是 9）、又非執行緒安全。Java 8 引入的 `java.time` 套件全面翻新：所有型別不可變、API 語意清楚，舊 API 只在維護老專案時才需要面對。

三個核心類別各司其職：`LocalDate`（只有日期）、`LocalTime`（只有時間）、`LocalDateTime`（兩者皆有）。搭配 `Period`（日期差）、`Duration`（時間差）與 `DateTimeFormatter`（格式化與解析）：

範例：TimeDemo.java ——java.time 操作總覽

```
import java.time.Duration;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Period;
import java.time.format.DateTimeFormatter;

/** java.time 的建立、計算、格式化與解析 */
public class TimeDemo {
    public static void main(String[] args) {
        // 教學用固定日期讓輸出可重現；實務常用 LocalDate.now()
        LocalDate today = LocalDate.of(2026, 8, 14);
        LocalTime lunch = LocalTime.of(12, 30);
        LocalDateTime meeting =
            LocalDateTime.of(2026, 8, 14, 14, 0);

        System.out.println("今天：" + today);
        System.out.println("星期：" + today.getDayOfWeek());

        // 不可變：plus 與 minus 都回傳「新物件」，原值不動
        LocalDate deadline = today.plusDays(30);
        System.out.println("30 天後截止：" + deadline);
        System.out.println("today 沒被改動：" + today);

        // with 系列：換掉某個欄位，同樣回傳新物件
        System.out.println("本月一號：" + today.withDayOfMonth(1));

        // Period：以年月日計的日期差，算年齡最順手
        LocalDate birthday = LocalDate.of(2004, 3, 21);
```

```
        Period age = Period.between(birthday, today);
        System.out.println("年齡：" + age.getYears() + " 歲又 "
            + age.getMonths() + " 個月");

        // Duration：以時分秒計的時間差
        Duration gap = Duration.between(lunch,
            meeting.toLocalTime());
        System.out.println("午餐到會議相隔 "
            + gap.toMinutes() + " 分鐘");

        // DateTimeFormatter：自訂顯示格式
        DateTimeFormatter fmt =
            DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH:mm");
        System.out.println("格式化：" + meeting.format(fmt));

        // 解析：字串轉回日期物件（ISO 格式可直接 parse）
        LocalDate parsed = LocalDate.parse("2026-10-10");
        System.out.println("解析結果：" + parsed.getMonthValue()
            + " 月 " + parsed.getDayOfMonth() + " 日");
    }
}
```

▶ 執行結果

```
今天：2026-08-14
星期：FRIDAY
30 天後截止：2026-09-13
today 沒被改動：2026-08-14
本月一號：2026-08-01
年齡：22 歲又 4 個月
午餐到會議相隔 90 分鐘
格式化：2026年08月14日 14:00
解析結果：10 月 10 日
```

格式樣板（pattern）常用代號：

代號	意義	代號	意義
`yyyy`	西元年（四位）	`HH`	時（24 小時制，補零）
`MM`	月（補零）	`mm`	分（補零）
`dd`	日（補零）	`ss`	秒（補零）
`E`	星期幾（依語系）	`a`	上午、下午標記

> **注意**　大小寫是有意義的：`MM` 是月、`mm` 是分；`HH` 是 24 小時制、`hh` 是 12 小時制。把 `yyyy-MM-dd` 誤寫成 `yyyy-mm-dd` 不會編譯錯誤，只會默默把「分鐘」印在月份的位置——用測試資料驗證一次你的樣板再上線。

以上的 `Local` 家族刻意「不帶時區」。跨時區需求（國際航班、全球服務）使用 `ZonedDateTime`，例如 `ZonedDateTime.now(ZoneId.of("Asia/Taipei"))`；系統內部與資料庫儲存則慣用 UTC 時間戳 `Instant`。時區規則複雜（夏令時間、歷史變更），原則是：**內部一律 UTC，只在顯示給人看的最後一刻轉當地時區**。

7.7 綜合範例：訂單狀態機

把本章工具組裝起來：`OrderStatus` 列舉內建狀態轉移規則（狀態機），`OrderItem` 用內部類別實作，時間計算交給 `java.time`：

範例：Order.java ——enum 狀態機加內部類別加時間

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;

/** 訂單：整合列舉狀態機、內部類別與日期時間 */
public class Order {

    /** 訂單狀態：合法轉移路徑寫死在列舉裡（巢狀 enum 隱含 static） */
    enum OrderStatus {
        CREATED, PAID, SHIPPED, COMPLETED, CANCELLED;

        /** 狀態機核心：只回答「能不能走到 target」 */
        boolean canTransitionTo(OrderStatus target) {
            // switch 運算式（Java 14）：涵蓋所有常數，不必 default
            return switch (this) {
                case CREATED -> target == PAID
                             || target == CANCELLED;
                case PAID -> target == SHIPPED
                          || target == CANCELLED;
                case SHIPPED -> target == COMPLETED;
                case COMPLETED, CANCELLED -> false;   // 終態
            };
        }
    }
```

```
26
27      /** 內部類別：訂單項目，離開訂單就沒有意義的小物件 */
28      class OrderItem {
29          private final String product;
30          private final int quantity;
31          private final int unitPrice;
32
33          OrderItem(String product, int quantity, int unitPrice) {
34              this.product = product;
35              this.quantity = quantity;
36              this.unitPrice = unitPrice;
37          }
38
39          int subtotal() {
40              return quantity * unitPrice;
41          }
42
43          @Override
44          public String toString() {
45              // 內部類別可直接使用外部訂單的 id 欄位
46              return "[" + id + "] " + product + " x" + quantity
47                  + " = " + subtotal() + " 元";
48          }
49      }
50
51      private final String id;                 // 訂單編號
52      private final LocalDateTime createdAt;   // 下單時間
53      private final List<OrderItem> items = new ArrayList<>();
54      private OrderStatus status = OrderStatus.CREATED;
55
56      Order(String id, LocalDateTime createdAt) {
57          this.id = id;
58          this.createdAt = createdAt;
59      }
60
61      void addItem(String product, int quantity, int unitPrice) {
62          items.add(new OrderItem(product, quantity, unitPrice));
63      }
64
65      /** 狀態轉移：先問狀態機，合法才放行 */
66      void changeStatus(OrderStatus target) {
67          if (!status.canTransitionTo(target)) {
```

```
            System.out.println("不允許：" + status + " -> " + target);
            return;
        }
        System.out.println("狀態轉移：" + status + " -> " + target);
        status = target;
    }

    void printSummary() {
        DateTimeFormatter fmt =
            DateTimeFormatter.ofPattern("MM/dd HH:mm");
        int total = 0;
        for (OrderItem item : items) {
            System.out.println(item);
            total += item.subtotal();
        }
        System.out.println("總金額：" + total + " 元");
        System.out.println("下單時間：" + createdAt.format(fmt));
        // 簡化規則：下單後 3 天送達
        System.out.println("預計送達："
            + createdAt.plusDays(3).format(fmt));
    }

    public static void main(String[] args) {
        Order order = new Order("A001",
            LocalDateTime.of(2026, 8, 14, 10, 30));
        order.addItem("機械鍵盤", 1, 2990);
        order.addItem("滑鼠墊", 2, 250);

        order.printSummary();
        order.changeStatus(OrderStatus.PAID);        // 合法
        order.changeStatus(OrderStatus.COMPLETED);   // 不合法：還沒出貨
        order.changeStatus(OrderStatus.SHIPPED);     // 合法
        order.changeStatus(OrderStatus.COMPLETED);   // 合法
    }
}
```

▶ 執行結果

```
[A001] 機械鍵盤 x1 = 2990 元
[A001] 滑鼠墊 x2 = 500 元
總金額：3490 元
下單時間：08/14 10:30
預計送達：08/17 10:30
```

```
狀態轉移：CREATED -> PAID
不允許：PAID -> COMPLETED
狀態轉移：PAID -> SHIPPED
狀態轉移：SHIPPED -> COMPLETED
```

把「哪些轉移合法」收進 enum，而不是散落在各處的 `if`，是狀態機的正統寫法：規則有單一權威來源，新增狀態時 switch 運算式的窮盡檢查會強迫你補齊所有分支。這個模式將在第 8 章的借閱流程中再次登場。

7.8 本章小結

- enum 讓固定選項型別安全：非法值進不來，switch 搭配箭頭語法乾淨俐落；`values`、`name`、`valueOf` 是三大內建工具，別把 `ordinal` 當商業數值。
- enum 是完整類別：可帶欄位、方法與 `private` 建構子（隱含）；抽象方法加常數專屬本體，讓每個常數自帶行為。
- 元素或鍵是列舉時，優先用 `EnumSet` 與 `EnumMap`：更快、更省、順序固定。
- 巢狀類別四種：靜態巢狀（首選）、內部（要外部狀態才用，小心持有外部參考造成洩漏）、區域（單一方法內）、匿名（一次性實作）。
- 匿名類別實作單方法介面，是 lambda 的前身；兩種寫法要能互相翻譯。
- 範圍亂數公式：`nextInt(max - min + 1) + min`；測試用固定種子讓序列可重現。
- `java.time` 全面取代 `Date` 與 `Calendar`：型別不可變，`plus`、`minus`、`with` 都回傳新物件。
- `Period` 算日期差、`Duration` 算時間差；格式化與解析用 `DateTimeFormatter`，注意樣板代號大小寫。
- 跨時區用 `ZonedDateTime`；原則是內部 UTC、顯示時才轉當地時區。

7.9 本章練習

動手練習

1. 觀念題：與 `int` 常數相比，enum 在型別安全上提供了哪些保證？`ordinal()` 為什麼不該當成商業邏輯的數值？（提示：想想「插入一個新常數」會發生什麼事。）
2. 實作題：設計 `Season` 列舉（四季），每個常數帶「平均溫度」欄位與「建議活動」方法，用 `values()` 迴圈印出完整介紹。（提示：仿照 Planet 的建構子寫法。）
3. 實作題：用 enum 實作紅綠燈 `TrafficLight`（RED、GREEN、YELLOW），提供

`next()` 方法回傳下一個狀態（紅轉綠、綠轉黃、黃轉紅），模擬十次變換。（提示：可用 switch 運算式，或每個常數覆寫抽象方法。）

4. 觀念題：四種巢狀類別各適合什麼場景？為什麼「用不到外部狀態就宣告成靜態巢狀類別」？（提示：非靜態內部類別多抓了一份什麼東西？）
5. 實作題：寫程式計算你出生至今活了幾天、幾歲又幾個月，分別用 `ChronoUnit.DAYS.between` 與 `Period.between`。（提示：`ChronoUnit` 在 `java.time.temporal` 套件。）
6. 實作題：用 `DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm")` 解析字串 `"2026/10/10 08:30"` 成 `LocalDateTime`，再以不同樣板格式化輸出。（提示：解析用 `LocalDateTime.parse(text, fmt)`。）

第 8 章

綜合實戰——圖書管理系統

本章學習目標

學完本章你將能夠：

- 從需求清單出發，規劃套件結構與類別職責，完成一個分層的主控台專案。
- 把中級篇八章的知識（封裝、介面、例外、集合、泛型、I/O、enum 與日期）整合進同一個系統。
- 用自訂例外表達商業規則，讓錯誤處理集中且友善。
- 以「對介面寫程式」設計儲存層，讓記憶體版與檔案版可以互換。
- 循「照打就能跑」的完整程式碼，獨立完成建置、執行與驗證。

前七章的知識點都是單兵訓練，本章進行合成演練：打造一個可以實際操作的**圖書管理系統**——新增書籍、註冊會員、借書還書、資料存檔，全部在主控台完成。麻雀雖小，五臟俱全：這個專案的分層方式（model、exception、repository、service、app）正是業界後端系統的標準縮影，進階篇的資料庫與網路章節會在同一副骨架上繼續長大。

本章請**開著編輯器逐檔輸入**：每個檔案一個程式碼框，照著套件路徑建立目錄、照打就能執行；打完一層，再回頭讀該層的設計說明。

8.1 需求分析

功能需求（系統要做什麼）：

- **書籍管理**：新增書籍（ISBN、書名、作者）、列出全部書籍與借閱狀態。
- **會員管理**：註冊會員（編號、姓名）。
- **借書**：書必須存在且未被借出；會員必須存在且未達借閱上限（2 本）；借期 14 天。
- **還書**：找到對應的借閱記錄結案，回報是否逾期與逾期天數。
- **持久化**：把書目存進文字檔，下次啟動可載回。

非功能需求（系統要做得多好）：

- **分層**：畫面、商業規則、資料存取徹底分離，各層只依賴下一層的介面。
- **可測試**：商業規則不依賴 `Scanner` 與檔案，可以在測試中用記憶體儲存庫直接驗證。

- **錯誤友善**：任何規則違反都以自訂例外回報，使用者看到的是清楚的中文訊息，程式不崩潰。

8.2 系統設計

套件結構（目錄即套件，一層一責）：

- `library.model`：資料模型——`Book`、`Member`、`Loan`。
- `library.exception`：領域例外——`LibraryException` 家族。
- `library.repository`：資料存取——`LibraryRepository` 介面與兩個實作。
- `library.service`：商業規則——`LibraryService`。
- `library.app`：使用者介面——`LibraryApp` 主程式。

各類別職責一覽：

類別	職責	主要知識點
`Book`	書籍資料與借閱狀態	封裝、equals 與 hashCode
`Member`	會員資料與借閱額度	封裝、類別常數
`Loan`	借閱記錄與逾期計算	不可變欄位、java.time
例外家族	表達違反的商業規則	繼承、自訂例外
`LibraryRepository`	儲存層合約	介面、多型
`InMemoryLibraryRepository`	記憶體儲存	Map、集合操作
`FileLibraryRepository`	文字檔持久化	繼承、檔案 I/O
`LibraryService`	借還書規則的權威來源	例外、防禦式檢查
`LibraryApp`	選單迴圈與訊息呈現	Scanner、switch、try-catch

依賴方向嚴格單向：`app` 依賴 `service`，`service` 依賴 `repository` 介面與 `model`，誰都不回頭依賴上層。

> **提示**　為什麼要分層？因為**變化的理由不同**：畫面想從主控台換成網頁、儲存想從檔案換成資料庫、規則想把上限改成 5 本——三種變化互不相干。分層讓每種變化只衝擊一層；`service` 依賴 `repository` 介面而非實作（第 2 章的開閉原則），更讓「換儲存方式」變成換一行 `new`。單機小專案就養成這個習慣，將來接手大系統時才不會水土不服。

8.3 model 層：三個資料類別

`Book` 以 ISBN 作為唯一識別：`equals` 與 `hashCode` 都只看 ISBN（第 1 章的標準範本），將來放進任何集合都行為正確。

檔案：Book.java ——套件 library.model

```java
package library.model;

import java.util.Objects;

/** 書籍：ISBN 為唯一識別；基本資料不可變，只有借閱狀態會變動 */
public class Book {

    private final String isbn;      // 唯一識別，永不改變
    private final String title;     // 書名
    private final String author;    // 作者
    private boolean available = true;   // 新書進館預設可借

    public Book(String isbn, String title, String author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public boolean isAvailable() {
        return available;
    }

    /** 出借與歸還都經由這裡改變狀態（唯一的寫入口） */
    public void setAvailable(boolean available) {
        this.available = available;
    }

    /** 邏輯相等：ISBN 相同即同一本書（第 1 章範本） */
    @Override
```

```
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Book)) {
            return false;
        }
        return isbn.equals(((Book) obj).isbn);
    }

    @Override
    public int hashCode() {
        return Objects.hash(isbn);
    }

    @Override
    public String toString() {
        return "《" + title + "》 (" + author + ") ";
    }
}
```

Member 管理借閱額度：上限做成類別常數，額度的加減提供具名方法而不是裸露的 setter——呼叫端讀起來是「借出一本」而不是「把某個數字加一」：

檔案：Member.java ——套件 library.model

```
package library.model;

/** 會員：個人資料與借閱額度 */
public class Member {

    /** 借閱上限：每人同時最多 2 本（規則改這裡就好） */
    public static final int MAX_LOANS = 2;

    private final String id;      // 會員編號
    private final String name;    // 姓名
    private int activeLoans;      // 目前借閱中的本數

    public Member(String id, String name) {
        this.id = id;
        this.name = name;
    }
```

```
17
18      public String getId() {
19          return id;
20      }
21
22      public String getName() {
23          return name;
24      }
25
26      /** 是否還有借閱額度 */
27      public boolean canBorrow() {
28          return activeLoans < MAX_LOANS;
29      }
30
31      /** 借出一本：額度佔用加一 */
32      public void incrementLoans() {
33          activeLoans++;
34      }
35
36      /** 歸還一本：額度釋放，防呆不讓它變負數 */
37      public void decrementLoans() {
38          if (activeLoans > 0) {
39              activeLoans--;
40          }
41      }
42  }
```

Loan 是借閱記錄：借出當下就推算出應還日（LocalDate 不可變，plusDays 回傳新物件——第 7 章），歸還時計算逾期天數：

檔案：Loan.java ——套件 library.model

```
1   package library.model;
2
3   import java.time.LocalDate;
4   import java.time.temporal.ChronoUnit;
5
6   /** 借閱記錄：一次借書行為的完整生命週期 */
7   public class Loan {
8
9       /** 借期規則：14 天 */
10      public static final int LOAN_DAYS = 14;
```

```
11
12      private final String isbn;              // 借了哪本書
13      private final String memberId;          // 誰借的
14      private final LocalDate borrowedOn;     // 借出日
15      private final LocalDate dueOn;          // 應還日
16      private LocalDate returnedOn;           // 實際還書日（未還為 null）
17
18      public Loan(String isbn, String memberId,
19                  LocalDate borrowedOn) {
20          this.isbn = isbn;
21          this.memberId = memberId;
22          this.borrowedOn = borrowedOn;
23          // 應還日由借出日推算：plusDays 回傳新物件
24          this.dueOn = borrowedOn.plusDays(LOAN_DAYS);
25      }
26
27      /** 尚未歸還的記錄才是「進行中」 */
28      public boolean isActive() {
29          return returnedOn == null;
30      }
31
32      /** 結案：記下還書日並回傳逾期天數（準時或提早為 0） */
33      public long close(LocalDate returnDate) {
34          this.returnedOn = returnDate;
35          long overdue = ChronoUnit.DAYS.between(dueOn, returnDate);
36          return Math.max(0, overdue);    // 提早還書不是負逾期
37      }
38
39      public String getIsbn() {
40          return isbn;
41      }
42
43      public String getMemberId() {
44          return memberId;
45      }
46
47      public LocalDate getDueOn() {
48          return dueOn;
49      }
50  }
```

8.4 exception 層：讓錯誤會說話

所有領域例外繼承共同基底 `LibraryException`（checked——這些都是營運中必然發生的狀況，呼叫端必須面對，第 3 章的選擇準則）。基底提供例外鏈建構子，供包裝底層 I/O 例外使用：

檔案：LibraryException.java ——套件 library.exception

```
package library.exception;

/** 領域例外基底：checked，強迫呼叫端面對可預期的營運狀況 */
public class LibraryException extends Exception {

    public LibraryException(String message) {
        super(message);
    }

    /** 例外鏈版本：包裝底層例外（如 IOException）時使用 */
    public LibraryException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

三個具體例外各對應一條核心商業規則，訊息在建構子裡組好，呼叫端拿到就能直接顯示。注意不必為每種錯誤都開一個類別：像「查無此書」這種呼叫端不需要特別區分的情況，直接用基底例外帶清楚訊息即可：

檔案：BookNotAvailableException.java ——套件 library.exception

```
package library.exception;

/** 規則：已被借走的書不能再借 */
public class BookNotAvailableException extends LibraryException {

    public BookNotAvailableException(String title) {
        super("《" + title + "》目前出借中，請改借其他書");
    }
}
```

檔案：MemberNotFoundException.java ——套件 library.exception

```
package library.exception;

```

```
/** 規則：借書者必須是已註冊的會員 */
public class MemberNotFoundException extends LibraryException {

    public MemberNotFoundException(String memberId) {
        super("查無會員，編號：" + memberId);
    }
}
```

檔案：LoanLimitExceededException.java ——套件 library.exception

```
package library.exception;

/** 規則：會員同時借閱的數量有上限 */
public class LoanLimitExceededException extends LibraryException {

    public LoanLimitExceededException(String name, int max) {
        super("會員 " + name + " 已達借閱上限（" + max
            + " 本），請先還書");
    }
}
```

8.5　repository 層：可抽換的儲存

先立合約。service 只認識這個介面，永遠不知道資料實際放在哪裡：

檔案：LibraryRepository.java ——套件 library.repository

```
package library.repository;

import java.util.Collection;

import library.model.Book;
import library.model.Member;

/** 儲存層合約：底層用記憶體、檔案還是資料庫，實作自己決定 */
public interface LibraryRepository {

    void addBook(Book book);

    /** 依 ISBN 查書；找不到回傳 null（由服務層轉成例外） */
    Book findBook(String isbn);
```

```
15
16      Collection<Book> allBooks();
17
18      void addMember(Member member);
19
20      /** 依編號查會員；找不到回傳 null */
21      Member findMember(String memberId);
22  }
```

記憶體版用兩個 Map 實作「以鍵查值」，查詢時間接近常數（第 4 章）。選 LinkedHashMap 是刻意的：列表輸出維持加入順序，畫面穩定好對照：

檔案：InMemoryLibraryRepository.java ——套件 library.repository

```
1   package library.repository;
2
3   import java.util.Collection;
4   import java.util.LinkedHashMap;
5   import java.util.Map;
6
7   import library.model.Book;
8   import library.model.Member;
9
10  /** 記憶體版儲存庫：LinkedHashMap 讓遍歷順序等於加入順序 */
11  public class InMemoryLibraryRepository implements LibraryRepository {
12
13      private final Map<String, Book> books = new LinkedHashMap<>();
14      private final Map<String, Member> members =
15          new LinkedHashMap<>();
16
17      @Override
18      public void addBook(Book book) {
19          // 同 ISBN 再加入視為覆蓋（Map 的天然行為）
20          books.put(book.getIsbn(), book);
21      }
22
23      @Override
24      public Book findBook(String isbn) {
25          return books.get(isbn);    // 查無此鍵回傳 null
26      }
27
28      @Override
```

```
    public Collection<Book> allBooks() {
        return books.values();
    }

    @Override
    public void addMember(Member member) {
        members.put(member.getId(), member);
    }

    @Override
    public Member findMember(String memberId) {
        return members.get(memberId);
    }
}
```

檔案版繼承記憶體版：日常操作全部沿用（is-a 成立：檔案版儲存庫「是一種」記憶體儲存庫，只是多了持久化能力），額外提供存檔與載入。存檔格式刻意簡單——一本書一行、欄位用直線符號分隔：

檔案：FileLibraryRepository.java ——套件 library.repository

```
package library.repository;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;

import library.exception.LibraryException;
import library.model.Book;

/** 檔案版儲存庫：沿用記憶體版行為，追加文字檔存讀能力 */
public class FileLibraryRepository extends InMemoryLibraryRepository {

    private final Path file;    // 存檔位置

    public FileLibraryRepository(Path file) {
        this.file = file;
    }

```

```
    /** 存檔：一本書一行，欄位以 | 分隔，回傳筆數 */
    public int save() throws LibraryException {
        List<String> lines = new ArrayList<>();
        for (Book b : allBooks()) {
            lines.add(b.getIsbn() + "|" + b.getTitle() + "|"
                + b.getAuthor() + "|" + b.isAvailable());
        }
        try {
            Files.write(file, lines, StandardCharsets.UTF_8);
        } catch (IOException e) {
            // 例外鏈（第 3 章）：把底層 IO 例外包成領域例外
            throw new LibraryException("存檔失敗：" + file, e);
        }
        return lines.size();
    }

    /** 載入：檔案不存在視為第一次啟動，不算錯誤 */
    public int load() throws LibraryException {
        if (!Files.exists(file)) {
            return 0;
        }
        try {
            List<String> lines =
                Files.readAllLines(file, StandardCharsets.UTF_8);
            for (String line : lines) {
                // split 的參數是正規表示式，| 須跳脫成 \\|
                String[] cols = line.split("\\|");
                Book book = new Book(cols[0], cols[1], cols[2]);
                book.setAvailable(Boolean.parseBoolean(cols[3]));
                addBook(book);    // 沿用父類別的加入邏輯
            }
            return lines.size();
        } catch (IOException e) {
            throw new LibraryException("載入失敗：" + file, e);
        }
    }
}
```

8.6　service 層：商業規則的權威

所有借還書規則只住在這裡：畫面層不重複檢查，儲存層也不管規則。留意 borrowBook 的結構——**先完成全部檢查，才開始改動狀態**，永遠不會出現「額度扣了、書卻沒借到」的半套結果：

檔案：LibraryService.java ——套件 library.service

```
package library.service;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

import library.exception.BookNotAvailableException;
import library.exception.LibraryException;
import library.exception.LoanLimitExceededException;
import library.exception.MemberNotFoundException;
import library.model.Book;
import library.model.Loan;
import library.model.Member;
import library.repository.LibraryRepository;

/** 服務層：借還書規則集中地；規則違反一律拋自訂例外 */
public class LibraryService {

    private final LibraryRepository repo;      // 只認識介面
    private final List<Loan> loans = new ArrayList<>();

    /** 建構子注入：測試時可以塞進記憶體版儲存庫 */
    public LibraryService(LibraryRepository repo) {
        this.repo = repo;
    }

    /** 借書：四道檢查全過，才建立借閱記錄 */
    public Loan borrowBook(String isbn, String memberId,
            LocalDate date) throws LibraryException {
        Book book = repo.findBook(isbn);
        if (book == null) {
            throw new LibraryException("查無此書，ISBN：" + isbn);
        }
        Member member = repo.findMember(memberId);
```

```
        if (member == null) {
            throw new MemberNotFoundException(memberId);
        }
        if (!book.isAvailable()) {
            throw new BookNotAvailableException(book.getTitle());
        }
        if (!member.canBorrow()) {
            throw new LoanLimitExceededException(
                member.getName(), Member.MAX_LOANS);
        }

        // 檢查全數通過，才真正改動狀態
        book.setAvailable(false);
        member.incrementLoans();
        Loan loan = new Loan(isbn, memberId, date);
        loans.add(loan);
        return loan;
    }

    /** 還書：結案借閱記錄，回傳逾期天數（0 表示準時） */
    public long returnBook(String isbn, String memberId,
            LocalDate date) throws LibraryException {
        // 找「這位會員、這本書、進行中」的那筆記錄
        Loan target = null;
        for (Loan loan : loans) {
            if (loan.isActive()
                    && loan.getIsbn().equals(isbn)
                    && loan.getMemberId().equals(memberId)) {
                target = loan;
                break;
            }
        }
        if (target == null) {
            throw new LibraryException(
                "查無進行中的借閱記錄：" + memberId + " / " + isbn);
        }

        long overdueDays = target.close(date);
        repo.findBook(isbn).setAvailable(true);      // 書變回可借
        repo.findMember(memberId).decrementLoans();  // 額度釋放
        return overdueDays;
    }
```

```
77 }
```

> **注意**　檢查的順序是有意設計的：先確認「東西存在」（書、會員），再檢查「狀態允許」（可借、有額度）。反過來寫，可能對不存在的物件呼叫方法而引爆 `NullPointerException`——那是程式臭蟲，不是商業例外。另外切記「先檢查、後改動」：只要有任何一道檢查可能失敗，就不要先動手改狀態，否則失敗時得寫復原邏輯，複雜度立刻翻倍。

8.7　app 層：選單迴圈

最外層負責人機互動：印選單、讀輸入、呼叫服務、把例外翻譯成友善訊息。每一輪操作都包在 try-catch 裡——任何商業規則失敗都只影響那一輪，系統繼續服務（第 3 章 ATM 範例的放大版）：

檔案：LibraryApp.java ——套件 library.app

```java
package library.app;

import java.nio.file.Path;
import java.time.LocalDate;
import java.util.Scanner;

import library.exception.LibraryException;
import library.model.Book;
import library.model.Loan;
import library.model.Member;
import library.repository.FileLibraryRepository;
import library.service.LibraryService;

/** 主控台進入點：只做三件事——讀輸入、呼叫 service、顯示結果 */
public class LibraryApp {

    // 教學用固定營業日，讓輸出可重現；實務改用 LocalDate.now()
    private static final LocalDate TODAY =
        LocalDate.of(2026, 8, 14);

    private final Scanner scanner = new Scanner(System.in);
    private final FileLibraryRepository repo =
        new FileLibraryRepository(Path.of("books.txt"));
    private final LibraryService service =
        new LibraryService(repo);
```

```
26
27      public static void main(String[] args) {
28          LibraryApp app = new LibraryApp();
29          app.seedData();
30          app.run();
31      }
32
33      /** 種子資料：讓系統一啟動就有東西可操作 */
34      private void seedData() {
35          repo.addBook(new Book("978-1", "Java 核心技術", "王小凡"));
36          repo.addBook(new Book("978-2", "演算法圖解", "林大同"));
37          repo.addBook(new Book("978-3", "資料庫實戰", "陳美心"));
38          repo.addMember(new Member("M001", "小明"));
39      }
40
41      /** 主迴圈：直到使用者選擇離開 */
42      private void run() {
43          System.out.println("=== 圖書管理系統（營業日 "
44              + TODAY + "） ===");
45          boolean running = true;
46          while (running) {
47              printMenu();
48              String choice = scanner.nextLine().trim();
49              try {
50                  switch (choice) {
51                      case "1" -> addBook();
52                      case "2" -> listBooks();
53                      case "3" -> registerMember();
54                      case "4" -> borrowBook();
55                      case "5" -> returnBook();
56                      case "6" -> saveData();
57                      case "0" -> running = false;
58                      default -> System.out.println("無效選項，請重新輸入");
59                  }
60              } catch (LibraryException e) {
61                  // 統一的錯誤出口：訊息友善，程式不中斷
62                  System.out.println("[操作失敗] " + e.getMessage());
63              }
64          }
65          System.out.println("感謝使用，再見！");
66      }
67
```

```
68      private void printMenu() {
69          System.out.println();
70          System.out.println("[1]新增書籍 [2]書籍列表 [3]註冊會員 "
71              + "[4]借書 [5]還書 [6]存檔 [0]離開");
72          System.out.print("請選擇：");
73      }
74
75      private void addBook() {
76          System.out.print("ISBN：");
77          String isbn = scanner.nextLine().trim();
78          System.out.print("書名：");
79          String title = scanner.nextLine().trim();
80          System.out.print("作者：");
81          String author = scanner.nextLine().trim();
82          repo.addBook(new Book(isbn, title, author));
83          System.out.println("已新增《" + title + "》");
84      }
85
86      private void listBooks() {
87          System.out.println("--- 書籍列表（共 "
88              + repo.allBooks().size() + " 本）---");
89          for (Book b : repo.allBooks()) {
90              String state = b.isAvailable() ? "可借" : "出借中";
91              System.out.println("[" + state + "] " + b.getIsbn()
92                  + "  " + b.getTitle() + " / " + b.getAuthor());
93          }
94      }
95
96      private void registerMember() {
97          System.out.print("會員編號：");
98          String id = scanner.nextLine().trim();
99          System.out.print("姓名：");
100         String name = scanner.nextLine().trim();
101         repo.addMember(new Member(id, name));
102         System.out.println("已註冊會員 " + name + "（" + id + "）");
103     }
104
105     private void borrowBook() throws LibraryException {
106         System.out.print("會員編號：");
107         String memberId = scanner.nextLine().trim();
108         System.out.print("書籍 ISBN：");
109         String isbn = scanner.nextLine().trim();
```

```
        Loan loan = service.borrowBook(isbn, memberId, TODAY);
        System.out.println("借閱成功，應還日期：" + loan.getDueOn());
    }

    private void returnBook() throws LibraryException {
        System.out.print("會員編號：");
        String memberId = scanner.nextLine().trim();
        System.out.print("書籍 ISBN：");
        String isbn = scanner.nextLine().trim();
        long overdue = service.returnBook(isbn, memberId, TODAY);
        if (overdue > 0) {
            System.out.println("還書成功，逾期 " + overdue
                + " 天，請補繳逾期費");
        } else {
            System.out.println("還書成功，未逾期");
        }
    }

    private void saveData() throws LibraryException {
        int count = repo.save();
        System.out.println("書目已寫入 books.txt（共 "
            + count + " 筆）");
    }
}
```

> **提示**　輸入一律用 `nextLine()` 再自行轉換，是 Scanner 的保命守則：混用 `nextInt()` 與 `nextLine()` 時，`nextInt` 會把換行符留在緩衝區，下一個 `nextLine` 直接吃到空字串——這是初學者論壇的月經題。統一 `nextLine` 加 `trim`，世界清靜。

建置與執行：在 `library` 目錄的上層執行 `javac -d out library/app/LibraryApp.java`——編譯器會沿著 `import` 自動把其餘檔案一併編譯進 `out`；接著用 `java -cp out library.app.LibraryApp` 啟動。

8.8　完整執行示範

以下是一次完整操作：新增第 4 本書、註冊會員小美、連借兩本、第三本被上限擋下、還書、存檔、離開。提示文字（如「請選擇：」「ISBN：」）後面緊接的內容，是使用者輸入的回顯：

▶ 執行結果

```
=== 圖書管理系統（營業日 2026-08-14）===
```

```
[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：1
ISBN：978-4
書名：設計模式入門
作者：張以芯
已新增《設計模式入門》

[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：3
會員編號：M002
姓名：小美
已註冊會員 小美（M002）

[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：4
會員編號：M002
書籍 ISBN：978-1
借閱成功，應還日期：2026-08-28

[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：4
會員編號：M002
書籍 ISBN：978-2
借閱成功，應還日期：2026-08-28

[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：4
會員編號：M002
書籍 ISBN：978-3
[操作失敗] 會員 小美 已達借閱上限（2 本），請先還書

[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：5
會員編號：M002
書籍 ISBN：978-2
還書成功，未逾期

[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：6
書目已寫入 books.txt（共 4 筆）
```

```
[1]新增書籍 [2]書籍列表 [3]註冊會員 [4]借書 [5]還書 [6]存檔 [0]離開
請選擇：0
感謝使用，再見！
```

驗證存檔成果：工作目錄出現 `books.txt`，每行一本書，例如第二行是 `978-2|演算法圖解|林大同|true`（歸還後回到可借），而 `978-1` 的最後一欄是 `false`（仍在小美手上）。想體驗載入，可在 `seedData` 後加一行呼叫 `repo.load()` 再觀察書籍列表（選項 2）的變化。

8.9 知識點回顧與延伸挑戰

這個專案把中級篇各章的知識用在哪裡？對照如下：

章	知識點	在本專案的落點
第 1 章	封裝、equals 與 hashCode	`Book` 以 ISBN 定義相等
第 1 章	繼承與 super	`FileLibraryRepository` 繼承記憶體版
第 2 章	介面與多型	`service` 依賴 `LibraryRepository` 介面
第 3 章	自訂例外、例外鏈	`LibraryException` 家族、包裝 IOException
第 4 章	Map 與集合選型	`LinkedHashMap` 存書與會員
第 5 章	泛型	`Map<String, Book>`、`List<Loan>`
第 6 章	檔案 I/O	`Files.write` 與 `readAllLines` 持久化
第 7 章	java.time、switch 箭頭	借期與逾期計算、選單分派

延伸挑戰（依難度遞增）：

- **搜尋**：新增「依書名關鍵字搜尋」選項（`title.contains(keyword)` 掃一遍 `allBooks`）。
- **序列化存檔**：仿照第 6 章，把整個館藏（書加會員加借閱記錄）改用 `ObjectOutputStream` 一次存讀，比較與文字檔方案的優缺點。
- **統計報表**：用 `Map` 統計每本書的歷史借閱次數，排序印出排行榜——進階篇學完 Stream API 之後，回來把這段改寫成一行管線，體會演進。

8.10 本章小結

- 分層架構：model 放資料、exception 放錯誤語意、repository 放存取、service 放規則、app 放互動；依賴方向嚴格由外而內單向。
- 商業規則集中在 service 層，且「先檢查、後改動」，避免半套狀態。
- 儲存層以介面立約，記憶體版與檔案版可互換；檔案版繼承記憶體版，只追加持久化能力。
- 自訂例外家族讓每條規則違反都有明確型別與友善訊息；底層 `IOException` 以例外鏈包裝上拋。

- app 層把每輪操作包進 try-catch：單次失敗不中斷服務；`Scanner` 統一用 `nextLine` 讀取。
- `LinkedHashMap` 讓輸出順序穩定；`Book` 覆寫 `equals` 與 `hashCode` 後可安心放進任何集合。
- 借期與逾期全靠 `LocalDate` 與 `ChronoUnit` 計算，不可變物件讓日期邏輯無副作用。
- 這副骨架就是實務系統的縮影：進階篇會在同樣的分層上換上資料庫、多執行緒與網路介面。

8.11 本章練習

動手練習

1. 實作題：實作「列出某會員借閱中書單」：輸入會員編號，顯示每本書的書名與應還日期。（提示：需要讓 `LibraryService` 提供查詢 `loans` 的方法，回傳該會員的進行中記錄。）
2. 實作題：加入逾期罰金規則：每逾期一天 5 元，還書時一併顯示罰金。（提示：規則放 service 層或 `Loan` 內，app 層只負責顯示。）
3. 實作題：讓 `FileLibraryRepository` 連會員也一起持久化（另存 `members.txt`），啟動時自動載入。（提示：儲存庫要先補上「列出全部會員」與讀取 `activeLoans` 的方法，再仿照 `save` 與 `load` 寫一對方法。）
4. 觀念題：如果要把主控台介面換成網頁介面，本專案哪些類別完全不用改？這說明了分層的什麼價值？（提示：想想依賴方向——誰依賴誰，誰就會被改動波及。）