COMP1110/1140/6710 —Structured Programming（S2 2026）

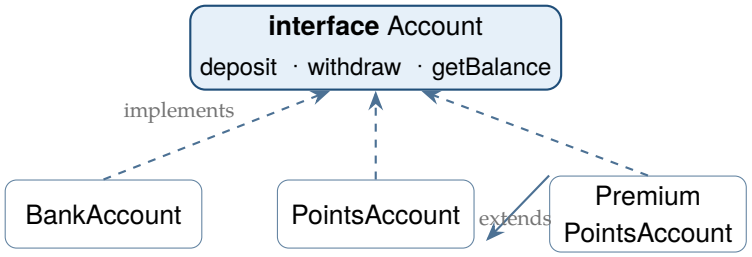
Java Week 6–7

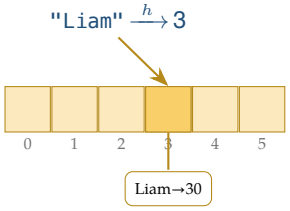
物件導向設計、測試與雜湊表

完全學習教材

封裝 • 抽象資料型別 • 介面 • 子型別與 LSP • 繼承

JUnit 與 TDD • 攤銷複雜度 • 轉型與泛型 • Collections • 雜湊函數與雜湊表





教材內容　觀念講解、圖解、逐步實作、JUnit 測試、複雜度分析、Workshop / Lab 導讀、自我測驗

隨附程式　`code/` 資料夾內 8 組可直接編譯執行的 Java 範例（含完整中文註解）

適用對象　已完成 Week 1–5（陣列、函式、遞迴、類別基礎、例外處理）的 Java 學習者

依據 ANU COMP1110 S2 2026 課程網站之 Lecture 06A、06B、07A、07B 講義，以及 2026 S1 版完整講義（Collections、Hashing）與 JUnit 5 官方文件編寫

目錄

1 這份教材要教你什麼 4

1.1 兩週的觀念地圖 5
1.2 關於 Java 版本與課堂寫法 5
1.3 隨附程式怎麼跑 5

2 Lecture 06A（上）：封裝——把狀態藏起來 8

2.1 值與實體：為什麼需要「可變的物件」 8
2.2 類別的解剖：欄位、建構子、方法與 `this` 8
2.3 存取修飾子：誰看得到？ 9
2.4 封裝的定義與兩個理由 10
2.5 用例外守住不變量 10
2.6 Getter 與 Setter 12
2.7 套件與 `import` 12
2.8 練習 13

3 Lecture 06A（下）：介面與抽象資料型別 13

3.1 介面就是契約 13
3.2 為什麼要多這一層？——解耦 14
3.3 抽象資料型別 15
3.4 案例：聊天紀錄的兩種實作 15
3.5 宣告型別決定你能呼叫什麼 18
3.6 最經典的 ADT：List 18
3.7 巢狀類別：把實作細節藏得更深 20
3.8 練習 20

4 Lecture 06B：子型別、繼承與 Liskov 替換原則 21

4.1 子型別：型別就是集合 21
4.2 繼承：重用另一個類別的實作 21
4.3 抽象類別 vs. 介面 23
4.4 動態分派：靜態型別 vs. 執行期型別 24
4.5 `final` 與 `static` 25
4.6 Liskov 替換原則：什麼時候繼承是「對的」？ 25
4.7 練習 27

5 Week 6 Workshop 導讀：設計聊天協定 28

5.1 什麼是協定？ . . . 28
5.2 一份協定草案 . . . 28
5.3 把訊息變成型別：`sealed interface` + `record` . . . 29
5.4 Java 網路程式的最小骨架 . . . 30
5.5 Workshop 討論清單 . . . 32

6 Lab 導讀：遞迴與排序 32

6.1 遞迴：兩個部分，一個信念 . . . 33
6.2 輔助參數與對外包裝 . . . 34
6.3 排序：兩條後置條件 . . . 35
6.4 實測：比較次數 . . . 36
6.5 把演算法當參數：`Sorter` 介面 . . . 37
6.6 Lab 練習 . . . 37

7 Lecture 07A（上）：JUnit 測試與測試驅動開發 40

7.1 為什麼要寫自動化測試？ . . . 40
7.2 JUnit 5 入門 . . . 40
7.3 案例：`GuestList` 的黑箱測試 . . . 41
7.4 兩種實作，一套測試 . . . 43
7.5 測試驅動開發 . . . 45
7.6 防禦性複製 . . . 46
7.7 陣列倍增：為什麼滿了要「乘二」而不是「加一」？ . . . 46
7.8 在命令列執行 JUnit . . . 46
7.9 練習 . . . 47

8 Lecture 07A（下）：轉型、泛型與 Collections 47

8.1 `Object` 與轉型 . . . 47
8.2 泛型：把型別當參數 . . . 48
8.3 型別擦除：泛型在執行期不存在 . . . 49
8.4 Java Collections 框架 . . . 50
8.5 練習 . . . 52

9 Lecture 07B：雜湊函數、雜湊表與攤銷複雜度 53

9.1 核心想法：把鍵變成陣列索引 . . . 53
9.2 雜湊函數 . . . 53

9.3 雜湊表：桶與碰撞 55
9.4 負載因子、擴容與攤銷複雜度 58
9.5 `equals` 與 `hashCode` 的契約 59
9.6 標準函式庫：`HashMap` 與 `HashSet` 60
9.7 練習 61

10 Week 7 Workshop 導讀：設計聊天客戶端的介面 62
10.1 為什麼要切這一刀？ 62
10.2 介面長什麼樣？ 63
10.3 用假實作測 UI 64
10.4 真實作的骨架與討論清單 65

11 總複習：觀念地圖、常見錯誤與自我測驗 66
11.1 一頁看完兩週 67
11.2 最常見的十二個錯誤 67
11.3 自我測驗（20 題） 68
11.4 中英術語對照 70

A 附錄：隨附程式完整原始碼 70
A.1 01_encapsulation：封裝 71
A.2 02_adt_interfaces：介面與抽象資料型別 75
A.3 03_inheritance_lsp：繼承與 LSP 82
A.4 04_lab_recursion_sorting：遞迴與排序 88
A.5 05_junit_tdd：JUnit 與 TDD 96
A.6 06_generics_collections：泛型與 Collections 103
A.7 07_hashing：雜湊 110
A.8 08_chat_protocol：聊天協定與客戶端 121

1　這份教材要教你什麼

前五週你已經學會了 Java 的「動詞」：變數、迴圈、陣列、函式、遞迴、例外處理，也初步認識了類別（class）與物件（object）。從第 6 週開始，課程的重心轉向**如何把程式組織成可以長大、可以測試、可以被別人安心使用的形狀**——這就是**物件導向設計**（object-oriented design）的核心，也是 Week 5 測驗之後、期末專案之前最關鍵的兩週。

本教材依照課程網站的週次表，完整涵蓋以下內容：

週次	活動	主題（本教材章節）
Week 6	Lecture 06A	封裝、抽象資料型別、介面（第 2、3 節）
	Lecture 06B	子型別、Liskov 替換原則、繼承（第 4 節）
	Workshop	聊天協定設計（第 5 節，COMP1140 專案）
	Lab	遞迴、排序（第 6 節）
Week 7	Lecture 07A	JUnit 測試、測試驅動開發、轉型、泛型、Collections（第 7、8 節）
	Lecture 07B	雜湊函數、雜湊表、攤銷複雜度（第 9 節）
	Workshop	聊天客戶端介面設計（第 10 節，COMP1140 專案）

讀完之後你將能夠：

- 用 `private` 欄位與公開方法**守住資料不變量**，並說出「封裝」帶來的兩個好處。
- 用 `interface` 定義**抽象資料型別**，寫出多種可互換的實作，並理解「宣告型別決定你能呼叫什麼」。
- 分辨**子型別**與**繼承**，解釋**動態分派**，並用 LSP 的四條規則判斷一個繼承關係是否「行為上正確」。
- 用 **JUnit 5** 寫黑箱單元測試、依**測試驅動開發**的節奏工作，並用**攤銷分析**解釋為什麼陣列倍增的插入是 $O(1)$。
- 讀懂並寫出**泛型**類別與方法，知道型別擦除的限制，熟練使用 `List` / `Set` / `Map`。
- 說出好的**雜湊函數**該有的性質、解釋**雜湊表**如何達到平均 $O(1)$，並避開 `equals` / `hashCode` 與可變鍵的地雷。

如何使用本教材

- 每一節開頭是**觀念**，接著是**完整可執行的範例**，最後是**練習**（附解答）。建議邊讀邊把 `code/` 資料夾裡對應的程式打開來跑。
- 「🔖 定義」框是考試會直接問的東西；「⚠ 注意」框是實作時最常踩的坑。
- 第 5、10 節的 Workshop 內容屬於 COMP1140（進階班）專案，COMP1110 / 6710 的同學不會被考，但其中的**介面設計思維**正是 Lecture 06A、07A 的最佳應用範例，很值得一讀。

- 第 11 節是總複習：觀念地圖、常見錯誤清單與 20 題自我測驗。

1.1　兩週的觀念地圖

圖 1 把這兩週的所有主題串成一張圖：左邊是「如何把資料與行為包起來」，中間是「如何讓不同的包裝可以互換」，右邊是「如何確認它們真的正確、真的夠快」。你會發現所有主題最後都指向同一個目標——**寫出別人（包括三個月後的你自己）可以安心使用與修改的程式**。

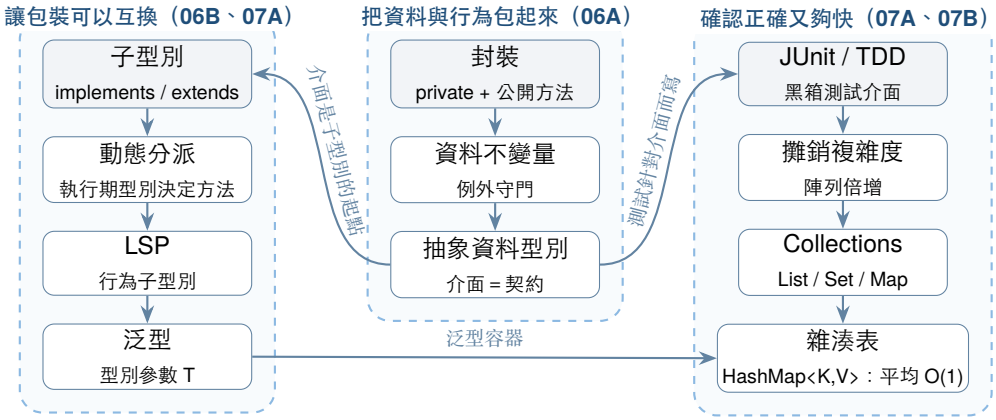


圖 1　Week 6–7 觀念地圖。介面（ADT）是整張圖的樞紐：它既是子型別關係的起點，也是黑箱測試的對象。

1.2　關於 Java 版本與課堂寫法

課程講義使用 Java 25 的兩個新寫法；為了讓範例能在任何 Java 17 以上的環境（含 IntelliJ 預設設定）編譯，本教材與隨附程式一律使用「傳統寫法」。兩者完全等價，對照如下：

課程講義（Java 25）	本教材（相容寫法）
`void main() { ... }`	`public static void main(String[] args) { ... }`（放在一個 `public class` 裡）
`IO.println(x);`	`System.out.println(x);`
`IO.readln(" 提示");`	`new Scanner(System.in).nextLine()`（先印出提示）

1.3　隨附程式怎麼跑

`code/` 底下每個資料夾都是獨立的一組範例，各自有一個含 `main` 的類別（`Main`、`Demo` 或以測試命名）。在該資料夾內：

```
$ javac *.java          # 編譯資料夾內所有檔案
$ java -ea Main         # 執行；-ea 開啟 assert（教材所有自我檢查都依賴它）
```

唯一的例外是 `05_junit_tdd`：它的測試需要 JUnit 5。在 IntelliJ 中把游標停在紅色的 `@Test` 上按 `Alt+Enter` → 「Add JUnit5 to classpath」即可；命令列的做法見第 7.8 節。

資料夾	對應章節	內容
`01_encapsulation`	§2	`Account`、`BankAccount`、`PointsAccount`、`PremiumPointsAccount`
`02_adt_interfaces`	§3	`ChatHistory` 兩種實作、`SimpleList` 與鏈結串列實作
`03_inheritance_lsp`	§4	抽象類別 `BankAccount` 家族、LSP 合規與違規範例
`04_lab_recursion_sorting`	§6	遞迴函式集、四種排序與比較次數實測
`05_junit_tdd`	§7	`GuestList` 兩種實作與 JUnit 5 黑箱測試
`06_generics_collections`	§8	泛型 `GuestList<T>`、`Pair<A,B>`、Collections 導覽
`07_hashing`	§9	字串雜湊實驗、鏈結雜湊表、線性探測、`HashMap` 陷阱
`08_chat_protocol`	§5、10	協定訊息解析器、Echo 伺服器 / 客戶端、`ChatClient` 介面

第一部分

Week 6：封裝、抽象與子型別

Lecture 06A • Lecture 06B • Week 6 Workshop • Lab：遞迴與排序

2　Lecture 06A（上）：封裝——把狀態藏起來

2.1　值與實體：為什麼需要「可變的物件」

先想想數字 5。你把它加 1，並不是「改變了 5」，而是得到一個全新的值 6。5 本身不可變（immutable），除了它的值之外沒有任何「身分」。Week 5 學過的 `record` 也是這樣：兩個欄位相同的 record 就是「同一個值」，可以放心用 `equals` 比較。

再想想你的銀行帳戶。你存入 100 元，並不會得到一個新帳戶，而是同一個帳戶的餘額被更新了。如果好幾個人共用一個帳戶，他們必須看到同一份資料——每次操作都複製一個新帳戶出來，是不可能正確的。

值（value）vs. 實體（entity）

值沒有身分，只有內容：`5`、`"hello"`、`new Point(1,2)`。相等就是內容相等。
實體有身分，且狀態會隨時間改變：銀行帳戶、聊天室、發電機。兩個實體即使此刻內容一樣，也是不同的東西。
Java 用 `record` 表達值，用**可變的 class** 表達實體。一個 `record` 只能表達實體在某一瞬間的快照，不能表達實體本身。

2.2　類別的解剖：欄位、建構子、方法與 `this`

一個類別（class）是「可變物件的藍圖」。下面這個 `BankAccount` 是本節的主角，先看全貌，再逐一拆解：

```
public class BankAccount implements Account {

    // 私有欄位：只有這個類別裡的程式碼看得到
    private int balance;                          // 資料不變量：balance >= 0

    /** 建構子：確保物件「出生」時就處於合法狀態。 */
    public BankAccount(int initialDeposit) {
        if (initialDeposit < 0) {
            throw new IllegalArgumentException("開戶金額不可為負數：" + initialDeposit);
        }
        this.balance = initialDeposit;
    }

    public void deposit(int amount) {
        if (amount < 0) {                         // 違反前置條件 → 拒絕
            throw new IllegalArgumentException("存款金額不可為負數：" + amount);
        }
        this.balance = this.balance + amount;     // 不變量仍成立
    }
```

```
20
21      public void withdraw(int amount) {
22          if (amount < 0) {
23              throw new IllegalArgumentException("提款金額不可為負數：" + amount);
24          }
25          if (amount > this.balance) {            // 物件目前的「狀態」不允許此操作
26              throw new IllegalStateException("餘額不足：餘額 " + this.balance + "，欲提領
    " + amount);
27          }
28          this.balance = this.balance - amount;   // 因為 amount <= balance，結果仍 >= 0
29      }
30
31      public int getBalance() {                   // Getter：讀得到、寫不到
32          return this.balance;
33      }
34  }
```

程式 1　BankAccount：封裝的標準範例（節錄自 01_encapsulation/BankAccount.java）

- **欄位**（field）：`private int balance;`——物件的**狀態**。每個 BankAccount 物件都有自己的一份。
- **建構子**（constructor）：名字與類別相同、**沒有回傳型別**（連 void 都沒有）的特殊方法，在 `new BankAccount(100)` 時執行。它的任務是讓物件一出生就滿足所有不變量。
- **方法**（method）：定義在類別內的函式，可以直接使用該物件的欄位。`acct.deposit(50)` 讀作「請 acct 這個物件執行 deposit」。
- `this`：在方法內代表「目前正在被操作的那個物件」。`this.balance` 就是「我的 balance」。當沒有同名的參數或區域變數時，`this.` 可以省略，但寫出來更清楚。

你其實早就在用物件了

`"hello".length()`、`guess.charAt(3)`、`arr.length`——字串與陣列都是物件。`str.length()` 是**呼叫方法**（有括號），`arr.length` 是**讀取欄位**（沒有括號，而且不能賦值）。現在你只是從「使用別人的類別」升級成「設計自己的類別」。

2.3　存取修飾子：誰看得到？

欄位、方法、甚至整個類別都可以加上**存取修飾子**（access modifier）：

修飾子	名稱	誰可以存取
`public`	公開	任何地方的任何程式碼。
`private`	私有	只有同一個類別裡的程式碼。
（不寫）	套件私有（package-private）	同一個套件內如同 `public`，套件外如同 `private`。
`protected`	受保護	同套件，加上所有子類別（第 4 節）。

⚠ `private` 是「類別」的界線，不是「物件」的界線

存取修飾子講的是程式碼寫在哪裡（語法範圍），而不是執行時是哪個物件。所以在 `BankAccount` 類別裡，一個帳戶物件的方法可以讀寫另一個帳戶物件的 `private balance`：

```
public void transferTo(BankAccount other, int amount) {
    this.withdraw(amount);
    other.balance = other.balance + amount;   // 合法：同一個類別內
}
```

這在寫 `equals`、`transfer` 這類「同型別互動」的方法時非常方便。

2.4 封裝的定義與兩個理由

封裝（encapsulation）

把欄位標為 `private` 以隱藏內部狀態，並提供公開方法作為唯一的互動管道。這樣做有兩個理由：

1. **完整性**（integrity）：我們能維持資料不變量（data invariant）——「在每一次方法呼叫之前與之後都必須成立的性質」。因為外界寫不到 `balance`，`account.balance = -500;` 根本無法編譯；想改餘額只能經過 `deposit` / `withdraw`，而那裡有我們的檢查邏輯。於是「餘額永遠 ≥ 0」成為保證，而非期望。
2. **抽象**（abstraction）：把**介面**（外界看得到的公開方法）與**實作**（私有欄位與方法內容）分開。使用者只需認識介面，認知負擔大幅降低；而我們日後想把餘額改用 `long`、改成以「分」為單位、甚至改存到資料庫，只要 `getBalance()` 的行為不變，呼叫者完全不會察覺。

2.5 用例外守住不變量

當方法被要求做不可能的事——存負數、提領超過餘額——正確的反應是**立刻失敗**，而不是默默做出錯誤結果。Java 的做法是**拋出例外**（throw an exception）。內建的兩種例外幾乎涵蓋所有情況：

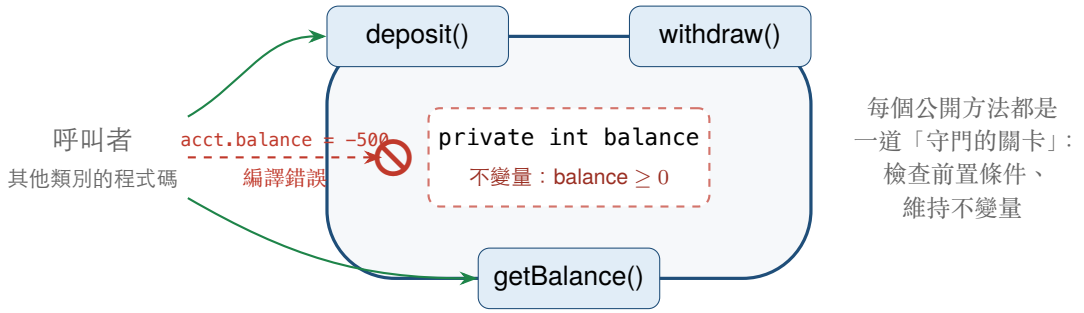


圖 2　封裝就像一顆膠囊：狀態在裡面，外界只能敲門（呼叫公開方法），不能直接伸手進去。

例外	何時使用
`IllegalArgumentException`	參數不合法：呼叫者違反了前置條件（例如 `deposit(-5)`）。
`IllegalStateException`	參數沒問題，但物件目前的狀態不允許這個操作（例如餘額 120 時 `withdraw(1000)`）。

呼叫端若能處理，就用 `try` / `catch` 接住：

```
try {
    acct.withdraw(1000);                                    // 餘額不足
} catch (IllegalStateException e) {
    System.out.println("攔到 IllegalStateException：" + e.getMessage());
}
assert acct.getBalance() == 120 : "失敗的操作不應改變狀態";
```

程式 2　接住例外（節錄自 `01_encapsulation/Main.java`）

⚠ 不要「吞掉」例外

初學者常為了讓錯誤訊息消失而寫 `catch (Exception e) { }`。這等於告訴程式「所有錯誤——包含你沒想到的邏輯 bug、`NullPointerException`、系統故障——通通忽略」。請只接住你知道怎麼處理的例外；不知道怎麼處理，就讓它往上傳。

assert 與例外的分工

Week 2 學的 `assert` 用於檢查「我自己的程式邏輯絕不該違反」的事（例如迴圈不變量、後置條件），而且預設不執行，是開發期的自我檢查。例外則用於「呼叫者或外界可能會做錯」的事，永遠生效，是公開介面的一部分。`BankAccount` 的 `deposit(-5)` 應該丟例外而不是 `assert`，因為關掉 `-ea` 之後負數存款就會悄悄成功。

2.6 Getter 與 Setter

Getter 與 Setter

Getter 讓外界「讀」一個私有欄位（`getBalance()`），但不能「寫」。
Setter 讓外界「設定」某個屬性（例如 `setName(String)`），它可以先檢查再接受，也可以拒絕（不改狀態或丟例外）；成功後對應的 getter 應回傳剛設定的值。

注意 `BankAccount` **刻意不提供** `setBalance`：「任意設定餘額」不是一個合理的帳戶操作。`deposit` 也不是 setter——它改變的是餘額，但你設定的是「增加多少」而不是「餘額變成多少」。**不要反射性地為每個欄位都寫一對 getter/setter**；先問：這個型別合理的操作是什麼？公開的方法應該對應**操作**，而不是對應欄位。

2.7 套件與 `import`

程式變大以後，「所有類別都在同一個資料夾」會撞名（世界上有多少人寫過 `Person`？）。**套件**（package）把相關的類別收成一組，並形成階層：

```
1 package au.edu.anu.comp1110.bank;      // 檔案第一行：宣告這個檔案屬於哪個套件
2
3 import java.util.Arrays;               // 引入單一類別
4 import java.util.*;                    // 引入整個套件（不建議：容易撞名）
```

- 套件名稱全部**小寫**，資料夾結構必須與套件名稱**一致**，例如：`au/edu/anu/comp1110/bank/BankAccount.java`。
- **完整名稱**（fully qualified name）是「套件. 類別」，例如 `java.util.Arrays`。`import` 只是讓你可以省略前綴。
- 對外發布的程式庫慣例用「反轉的網域名稱」開頭：Google 的是 `com.google.…`，ANU 的會是 `au.edu.anu.…`。
- 沒寫 `package` 的檔案屬於**預設套件**（default package）。它只適合小型練習：明確套件裡的程式碼**無法**存取預設套件裡的東西。本教材的範例為了方便都用預設套件。
- 執行有套件的類別要用完整名稱，且要站在套件根目錄的上一層：`java au.edu.anu.comp1110.bank.Main`。

2.8 練習

1-1：發電機

設計一個 `Generator` 類別：欄位 `fuel`（目前油量）與 `capacity`（油箱容量），資料不變量 $0 \le$ `fuel` $\le$ `capacity`。方法：`run(int hours)` 每小時耗油 5 單位，油不夠時拋出 `IllegalStateException`；`refill(int amount)` 加油，會溢出或 `amount` 為負時拋出 `IllegalArgumentException`；`fuelRemaining()` 是 getter。寫一個 `main` 用 `assert` 測試每一條規則。

1-2：找出封裝漏洞

下面的類別有一個嚴重的封裝漏洞，它在哪裡？外界可以怎麼利用它打破不變量？

```java
public class Roster {
    private String[] names = new String[10];   // 不變量：沒有 null 之後還有名字
    private int count = 0;
    public void add(String n) { names[count] = n; count = count + 1; }
    public String[] getNames() { return names; }
}
```

1-2 解答

`getNames()` 把內部陣列的參照直接交出去。呼叫者寫 `r.getNames()[0] = null;` 或 `r.getNames()[5] = "Intruder";` 就能繞過 `add` 改動內部狀態，不變量形同虛設。解法是回傳防禦性複製（defensive copy）：`return Arrays.copyOf(names, count);`。第 7.6 節會再深入討論。

3 Lecture 06A（下）：介面與抽象資料型別

3.1 介面就是契約

上一節的 `BankAccount` 第一行寫著 `implements Account`。`Account` 長這樣：

```java
public interface Account {
    /** 存入 amount。前置條件：amount >= 0 */
    void deposit(int amount);

    /** 提出 amount。前置條件：0 <= amount <= getBalance() */
    void withdraw(int amount);

    /** @return 目前餘額，後置條件：回傳值 >= 0 */
    int getBalance();
```

```
}
```

程式 3　Account 介面（01_encapsulation/Account.java）

介面（interface）

介面是一份**契約**：列出一組方法的**簽名**（名稱、參數、回傳型別）與**規格**（前置 / 後置條件，寫在註解裡），但**沒有任何實作**——沒有欄位、沒有方法本體。一個類別宣告 `implements Account`，就是承諾提供契約裡的**每一個**方法，少一個都無法編譯。

幾條語法規則：

- 介面裡的方法自動是 `public` 且抽象的（沒有本體，以分號結尾），不必也不該再寫 `public`。
- 實作類別的方法**必須**是 `public`，並建議加上 `@Override`——這個標註讓編譯器幫你檢查「我真的在實作介面的方法」，拼錯名字時會立刻報錯，而不是默默變成一個無關的新方法。
- 介面**不能被 `new`**：`new Account()` 是編譯錯誤，因為它沒有任何實作。
- 一個類別可以同時實作**多個**介面：`class Foo implements Account, Comparable<Foo>`。

3.2　為什麼要多這一層？——解耦

`Main.java` 裡有一個只認識 `Account` 的函式：

```
static void handleAccount(Account a) {      // 參數型別是介面，不是某個具體類別
    System.out.println("  餘額：" + a.getBalance());
    a.deposit(500);
    System.out.println("  存入 500 後：" + a.getBalance());
}
```

於是 `handleAccount(new BankAccount(0))` 與 `handleAccount(new PointsAccount())` 都合法，而 `PointsAccount` 的內部長得完全不一樣（它有 `points` 欄位、開戶送 10000 點、每次存入加送 1%）。執行結果：

```
BankAccount：
  餘額：0
  存入 500 後：500
PointsAccount：
  餘額：10000
  存入 500 後：10505
```

介面的三個好處

1. **解耦**（decoupling）：`handleAccount` 對實作零依賴。日後新增 `CryptoAccount`，這個函式一個字都不必改。
2. **可替換**：測試時可以塞一個假的 `Account` 進去（第 10 節的 `FakeChatClient` 就是這麼做的）。
3. **分工**：介面是團隊的「合約書」。你寫使用端、同學寫實作端，只要雙方都對著同一份介面，最後一定能接起來——這正是 Week 6 / 7 Workshop 聊天專案的做法。

3.3 抽象資料型別

抽象資料型別（Abstract Data Type，ADT）

一個 ADT 是「一組操作」加上「這些操作的行為規格」，完全不提及資料如何儲存。在 Java 裡，**介面**是表達 ADT 的自然工具；實作該介面的每一個類別則是一種**資料結構**（data structure）——ADT 的一種具體實現。

一句話：**ADT 說「做什麼」，資料結構說「怎麼做」。**

3.4 案例：聊天紀錄的兩種實作

考慮聊天室要保存的訊息紀錄。它的 ADT 非常簡單：

```
public interface ChatHistory {
    void addMessage(String msg);   // 新增一則訊息
    void display();                // 最新在前，逐行印出
    int size();                    // 目前保存的訊息數量
}
```

程式 4　`ChatHistory` ADT（`02_adt_interfaces/ChatHistory.java`）

同一份契約，我們寫出兩種截然不同的資料結構。

實作一：環狀緩衝區

固定容量的陣列，滿了以後新訊息覆蓋最舊的那一則。索引用 `% arr.length` 取餘數「繞回」開頭，所以叫**環狀緩衝區**（circular buffer）。

```
private final String[] arr;      // 固定大小
private int nextAvailable;       // 下一則要寫入的索引
private int size;                // 目前保存的數量（<= arr.length）

@Override
public void addMessage(String msg) {
```

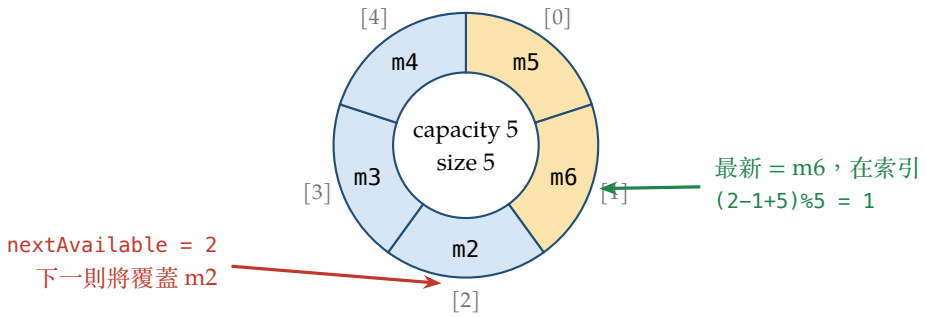


圖 3　容量 5 的環狀緩衝區已存入 m1–m6：m1 已被 m6 覆蓋。

```
    this.arr[this.nextAvailable] = msg;
    this.nextAvailable = (this.nextAvailable + 1) % this.arr.length;   // 繞回
    if (this.size < this.arr.length) {
        this.size = this.size + 1;   // 還沒滿才 +1；滿了就是覆蓋
    }
}

@Override
public void display() {
    int idx = (this.nextAvailable - 1 + this.arr.length) % this.arr.length;  // 最新
    for (int i = 0; i < this.size; i++) {
        System.out.println(this.arr[idx]);
        idx = (idx - 1 + this.arr.length) % this.arr.length;   // 往回退一格
    }
}
```

程式 5　環狀緩衝區的核心（節錄自 `CircularBufferChatHistory.java`）

⚠ 為什麼要 `+ arr.length`？

Java 的 `%` 對負數會得到負數：`(0 - 1) % 5` 是 `-1`，拿去當索引會直接 `ArrayIndexOutOfBoundsException`。先加一個 `arr.length` 保證被除數非負，`(0 - 1 + 5) % 5 = 4` 才是我們要的「從 0 退到 4」。（或者用 `Math.floorMod(idx - 1, arr.length)`，第 9 節算雜湊桶時會再用到。）

實作二：鏈結串列

不限數量、永不覆蓋。新訊息永遠**接在最前面**（prepend），所以串列天生就是「最新在前」的順序。

```
public class LinkedListChatHistory implements ErasableChatHistory {
```

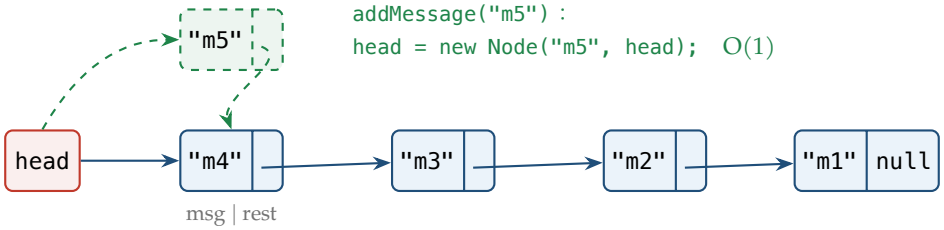


圖 4　鏈結串列版本：每則訊息是一個 Node，新訊息成為新的 head。

```
    private static class Node {          // 靜態巢狀類別：外界看不到
        final String msg;
        final Node rest;
        Node(String msg, Node rest) { this.msg = msg; this.rest = rest; }
    }

    private Node head = null;            // null 代表空串列
    private int size = 0;

    @Override
    public void addMessage(String msg) {
        this.head = new Node(msg, this.head);   // 新節點指向舊 head，再成為新 head
        this.size = this.size + 1;
    }

    @Override
    public void display() {
        Node current = this.head;
        while (current != null) {
            System.out.println(current.msg);
            current = current.rest;
        }
    }

    @Override public int size() { return this.size; }
    @Override public void erase() { this.head = null; this.size = 0; }
}
```

程式 6　鏈結串列的核心（節錄自 LinkedListChatHistory.java）

同一段程式碼，兩種實作

```
static void simulateChat(ChatHistory history) {   // 只認識 ChatHistory
    history.addMessage("Alice: 嗨！");
```

```
    history.addMessage("Bob: 你好～");
    history.addMessage("Alice: 週六要一起讀 Java 嗎？");
    history.addMessage("Bob: 好啊，下午兩點圖書館見。");
    System.out.println("  （共 " + history.size() + " 則，最新在前）");
    history.display();
}
simulateChat(new LinkedListChatHistory());        // 4 則全留
simulateChat(new CircularBufferChatHistory(3));   // 只留最近 3 則
```

實作	addMessage	display	空間	取捨
環狀緩衝區	$O(1)$	$O(\text{capacity})$	固定	舊訊息會遺失；記憶體可預測
鏈結串列	$O(1)$	$O(n)$	隨 n 成長	全部保留；每個節點多一個參照的開銷

3.5　宣告型別決定你能呼叫什麼

`LinkedListChatHistory` 實作的其實是 `ErasableChatHistory`——一個擴充了 `ChatHistory` 的介面：

```
public interface ErasableChatHistory extends ChatHistory {
    void erase();          // 在 ChatHistory 的三個方法之外，再多要求一個
}
```

於是同一個物件，用不同型別的變數接住，能呼叫的方法就不同：

```
ChatHistory h1 = new LinkedListChatHistory();
h1.addMessage("hello");
// h1.erase();   // 編譯錯誤！h1 的「宣告型別」是 ChatHistory，契約裡沒有 erase()

ErasableChatHistory h2 = new LinkedListChatHistory();
h2.erase();      // OK
```

型別驅動存取（type-driven access）

編譯器只看變數的**宣告型別**（靜態型別）來決定「哪些方法存在」，不看它實際指向什麼物件。這是刻意的：`h1` 的使用者只被承諾了 `ChatHistory` 的三個操作，所以我們可以隨時把 `h1` 換成環狀緩衝區而不會弄壞任何程式碼。第 4 節會看到：**方法存不存在**由宣告型別決定，但**執行哪個版本**由執行期型別決定。

3.6　最經典的 ADT：List

串列（list）是「有順序、允許重複、大小可變」的集合——最後一點正是它與陣列的根本差別。一個最小的 List ADT：

```java
public interface SimpleList {
    int size();
    Object get(int index);   // 前置條件：0 <= index < size()
    void add(Object o);      // 加到尾端；後置條件：size() 增加 1
    void remove(Object o);   // 移除第一個 equals(o) 的元素；不存在則不動作
}
```

程式 7　SimpleList ADT（02_adt_interfaces/SimpleList.java）

元素型別暫時用 Object（Java 裡所有物件的共同祖先，所以什麼都能放）；第 8 節會用泛型把它升級成 SimpleList<T>，讓編譯器幫我們記住裡面放的是什麼。

LinkedSimpleList 用鏈結串列實作它。這裡的重點不是演算法，而是**封裝把「使用串列」變得多簡單**：使用者不必知道 Node 的存在，不必自己區分「空串列」與「非空串列」，不必記得維護 size。

```java
@Override
public void remove(Object o) {
    if (this.head == null) return;                    // 空串列
    if (Objects.equals(this.head.value, o)) {         // 移除的是第一個節點
        this.head = this.head.next;
        this.size = this.size - 1;
        return;
    }
    Node prev = this.head;
    while (prev.next != null) {
        if (Objects.equals(prev.next.value, o)) {
            prev.next = prev.next.next;               // 跳過被移除的節點
            this.size = this.size - 1;
            return;
        }
        prev = prev.next;
    }
}
```

程式 8　remove 的實作（節錄自 LinkedSimpleList.java）

Objects.equals(a, b)

a.equals(b) 在 a 為 null 時會爆 NullPointerException。java.util.Objects.equals(a, b) 會先處理 null（兩個都 null 回 true，只有一個 null 回 false），是比較兩個可能為 null 的物件時的標準寫法。

3.7　巢狀類別：把實作細節藏得更深

Node 是「只有 LinkedSimpleList 自己需要知道」的東西，所以我們把它宣告在類別裡面並標為 private。Java 有兩種寫法：

	靜態巢狀類別（private static class Node）	內部類別（private class Node）
與外層物件的關係	獨立；只是「放在裡面」的普通類別	每個 Node 都隱含綁定一個外層物件，可直接讀外層的欄位
何時用	節點不需要碰外層狀態時（預設選擇）	需要存取外層的 private 欄位或方法時
範例	LinkedListChatHistory.Node	LinkedSimpleList.Node

兩者在本節的用途下行為相同；課程講義用的是內部類別，實務上若不需要外層狀態，加 static 更省記憶體也更清楚。

3.8　練習

2-1：陣列版 List

寫一個 ArraySimpleList implements SimpleList，內部用 Object[] data 與 int size。add 在陣列滿時把容量加倍（Arrays.copyOf）；remove 需要把後面的元素往前搬。寫完後把 Main 裡的 new LinkedSimpleList() 換成你的類別——**其他程式碼一個字都不必改**，這就是 ADT 的威力。第 9.4 節會分析為什麼「加倍」是正確的策略。

2-2：介面 vs. 實作

以下敘述哪些正確？

(a) ChatHistory h = new ChatHistory(); 可以編譯。

(b) 一個類別可以同時 implements ChatHistory, SimpleList。

(c) ErasableChatHistory 型別的變數一定可以呼叫 display()。

(d) 把 CircularBufferChatHistory 的 arr 改名成 buffer，會影響使用 ChatHistory 的程式。

2-2 解答

(a) ✗ 介面不能 new。(b) ✓ 可以實作多個介面，只要全部方法都提供。(c) ✓ ErasableChatHistory extends ChatHistory，所以包含 display()。(d) ✗ arr 是 private 實作細節，外界根本看不到——這正是封裝的意義。

4　Lecture 06B：子型別、繼承與 Liskov 替換原則

上一節我們把 `new PointsAccount()` 交給一個型別為 `Account` 的參數，編譯器欣然接受。這背後的規則叫**子型別**。本節把它講清楚，再介紹 Java 重用程式碼的機制——**繼承**，最後用 **Liskov 替換原則**回答一個關鍵問題：什麼時候「A 繼承 B」是正確的設計？

4.1　子型別：型別就是集合

子型別（subtype）

若任何需要 U 的地方都可以安全地提供一個 T，則 T 是 U 的子型別（T *is a* U）。
在 Java 中，`class T implements U`、`class T extends U`、`interface T extends U` 都會建立「T 是 U 的子型別」的關係，而且這個關係是**可傳遞**的。

把每個型別想成「所有屬於它的物件」形成的集合，子型別就是子集合。每一個 `PremiumPointsAccount` 物件同時也是一個 `PointsAccount`；每一個 `PointsAccount` 同時也是一個 `Account`；而每一個 `Account`（以及所有其他物件）都是 `Object`。



圖 5　型別即集合：子型別關係就是子集合關係。Premium ⊂ Points ⊂ Account ⊂ Object，所以「需要 `Account` 的地方，給它任何一個子集合裡的物件都行」。

`Object`：所有東西的共同祖先

Java 的每一個類別都（直接或間接）是 `Object` 的子型別，所以下面的程式合法：

```
Object o = "hello";                                  // String 是 Object
Object[] mixed = { 42, "text", new BankAccount(3) };   // 42 會自動包成 Integer
```

代價是：透過 `Object` 型別的變數，你只能呼叫 `Object` 自己的方法（`toString`、`equals`、`hashCode`……）。想拿回原本的能力，需要**轉型**（第 8.1 節）。

4.2　繼承：重用另一個類別的實作

介面只共享契約；當幾個類別連實作都大同小異時，可以讓一個類別 `extends` 另一個，把欄位與方法繼承過來。本節的範例是一個帳戶家族（`code/03_inheritance_lsp`）：

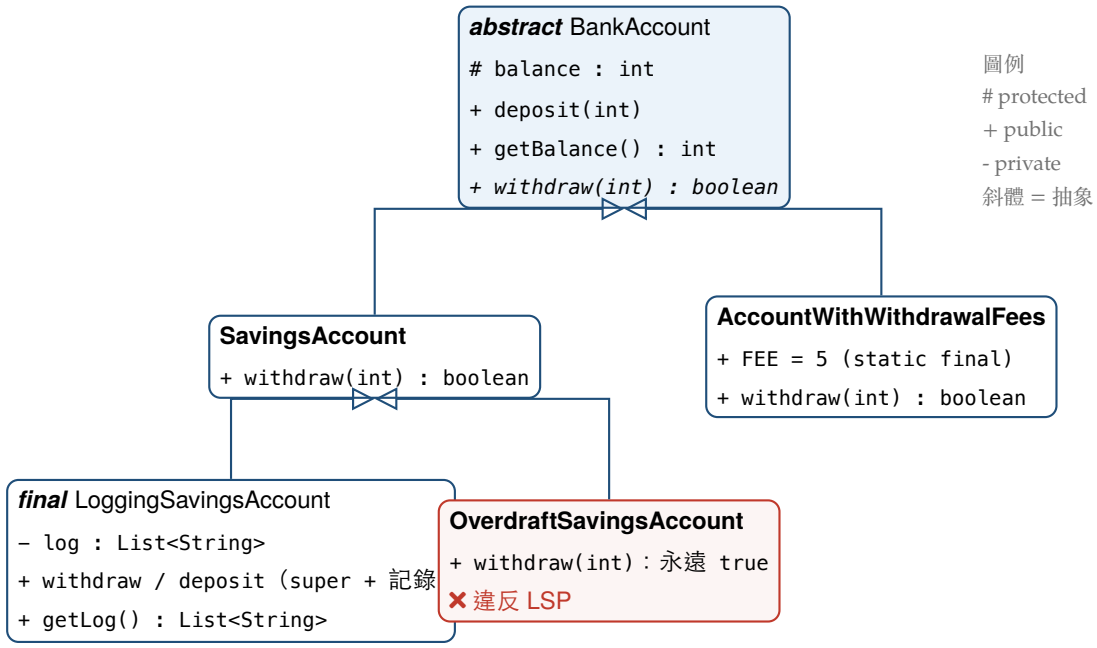


圖 6　03_inheritance_lsp 的類別階層（UML 風格；空心三角箭頭 = extends）。

```java
public abstract class BankAccount {
    protected int balance;                  // 子類別可直接存取

    public BankAccount() { this.balance = 0; }

    public void deposit(int amount) {       // 具體方法：所有子類別直接繼承
        if (amount < 0) throw new IllegalArgumentException("存款金額不可為負數：" +
    amount);
        this.balance = this.balance + amount;
    }
    public int getBalance() { return this.balance; }

    /** 抽象方法：每種帳戶的提款規則不同，只宣告不實作。 */
    public abstract boolean withdraw(int amount);
}
```

程式 9　抽象父類別（節錄自 BankAccount.java）

```java
public class SavingsAccount extends BankAccount {
    public SavingsAccount() {
        super();                                    // 呼叫父類別的建構子
    }
    @Override
    public boolean withdraw(int amount) {
        if (amount < 0) throw new IllegalArgumentException("提款金額不可為負數：" +
    amount);
```

```
        if (this.balance < amount) return false;  // 餘額不足：拒絕，狀態不變
        this.balance = this.balance - amount;     // protected 欄位：子類別可直接存取
        return true;
    }
}
```

程式 10　子類別補上抽象方法（SavingsAccount.java）

繼承的語法規則整理：

- `class B extends A`：B 是 A 的**子類別**（subclass），A 是 B 的**父類別**（superclass）。B 自動擁有 A 的所有**非私有**欄位與方法（`private` 成員存在於物件裡，但 B 的程式碼看不到）。
- 一個類別**只能 extends 一個**父類別（單一繼承），但可以 implements 多個介面。沒寫 `extends` 的類別預設 `extends Object`。
- **覆寫**（override）：子類別重新定義父類別的方法（相同簽名），一定加 `@Override` 讓編譯器檢查。
- `super.method(...)`：呼叫**父類別版本**的方法。常見寫法是「先做父類別的事，再加上自己的事」：

```
@Override
public boolean withdraw(int amount) {
    boolean ok = super.withdraw(amount);       // 父類別的真正邏輯
    this.log.add(ok ? "withdraw " + amount : "withdraw " + amount + " FAILED");
    return ok;
}
```

- **建構子不會被繼承**。子類別建構子的第一行（隱含或明寫）是 `super(...)`，先把「父類別那部分」建好。若父類別沒有無參數建構子，你**必須**明寫 `super(引數)`。
- `protected`：讓子類別能存取，但無關的類別不能——比 `private` 寬、比 `public` 窄。

4.3　抽象類別 vs. 介面

`abstract class` 是「不完整的類別」：可以有欄位、具體方法與**抽象方法**（只宣告不實作），但**不能被 new**。子類別必須實作所有抽象方法，否則自己也得標為 `abstract`。

	介面	抽象類別
能否有欄位（狀態）	只能有 `static final` 常數	可以（balance）
能否有具體方法	只有 `default` 方法（§8.4）	可以（deposit）
一個類別能繼承幾個	多個 `implements`	只能 `extends` 一個
適合表達	**能力／契約**（「可以存提款」）	**共同骨架**（「所有帳戶都有餘額」）
何時選它	預設選擇；解耦最徹底	多個實作共享大量相同程式碼時

4.4　動態分派：靜態型別 vs. 執行期型別

```
PointsAccount p = new PremiumPointsAccount();   // 靜態型別 PointsAccount，執行期 Premium
p.deposit(500);                                 // 執行的是 Premium 版！
System.out.println(p.getBalance());             // 10555，不是 10505
```

程式 11　節錄自 `01_encapsulation/Main.java`

計算過程：

- Premium 版的 `deposit(500)`：金額乘 1.1 得 550；
- 接著呼叫 `super.deposit(550)`：`PointsAccount` 加送 1%，得 $550 + 5 = 555$；
- 餘額 $10000 + 555 = 10555$。

動態分派（dynamic dispatch）

呼叫 `x.m()` 時，**編譯器**用 `x` 的**靜態型別**（static type）（宣告型別）檢查 `m` 是否存在；**執行期**則依 `x` 目前指向的物件的**執行期型別**（runtime type）（new 出來的那個類別）決定執行哪一個版本的 `m`——一路往上找到最近的定義。

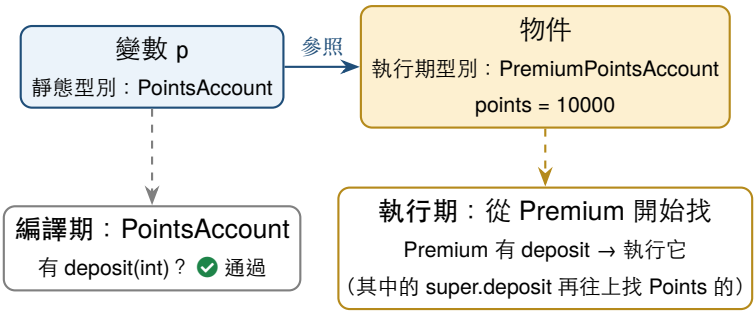


圖 7　同一個呼叫的兩個階段：靜態型別決定「合法性」，執行期型別決定「行為」。

動態分派讓「一個陣列、一個迴圈、多種行為」成為可能：

```
BankAccount[] accounts = {
    new SavingsAccount(), new LoggingSavingsAccount(), new AccountWithWithdrawalFees()
};
for (BankAccount a : accounts) {
    a.deposit(100);
    boolean ok = a.withdraw(100);   // 靜態型別 BankAccount，執行的是各子類別的 withdraw
    System.out.println("  " + a.getClass().getSimpleName() + "：存 100 領 100 → " + ok +
    "，餘額 " + a.getBalance());
}
```

程式 12　節錄自 `03_inheritance_lsp/Main.java`

```
SavingsAccount：存 100 領 100 → true，餘額 0
LoggingSavingsAccount：存 100 領 100 → true，餘額 0
AccountWithWithdrawalFees：存 100 領 100 → false，餘額 100
```

4.5 final 與 static

- final 類別（public final class LoggingSavingsAccount）：**不能再被繼承**。final 方法：不能被覆寫。final 欄位：只能在建構子中賦值一次（arr、log）。宣告 final 是在說「我沒有為被繼承而設計，請不要繼承我」——Java 的 String 就是 final。
- static 成員屬於**類別本身**而非個別物件，所有物件共用一份。用類別名稱存取：

```
public static final int FEE = 5;                // 常數：static final，全大寫命名
int total = amount + AccountWithWithdrawalFees.FEE;
```

  Week 1–5 寫的所有「純函式」（static int gcd(int a, int b)）都是 static 方法：它們不需要任何物件的狀態。

- 判斷原則：**需要讀寫某個物件的欄位 → 實例方法；只依賴參數 → static 方法**。在 static 方法裡沒有 this，所以不能直接用實例欄位——這是 main 裡常見的編譯錯誤來源。

4.6 Liskov 替換原則：什麼時候繼承是「對的」？

編譯器只檢查簽名——只要方法名稱與型別對得上，任何 extends 都能編譯。但「合法」不等於「正確」。看 Main.java 裡這個以 SavingsAccount 的契約為前提寫的函式：

```
/** 依 SavingsAccount 的契約：提領剛好等於餘額的金額一定成功，之後餘額為 0。 */
static void withdrawEverything(SavingsAccount acct) {
    int all = acct.getBalance();
    boolean ok = acct.withdraw(all);
    assert ok : "契約保證：提領等於餘額的金額必定成功";
    assert acct.getBalance() == 0 : "契約保證：領光之後餘額為 0";
}
```

再看這個「合法」的子類別：

```
public class OverdraftSavingsAccount extends SavingsAccount {
    @Override
    public boolean withdraw(int amount) {
        this.balance = this.balance - amount;   // 不檢查餘額
        return true;                            // 永遠宣稱成功
    }
}
```

程式 13　LSP 違規範例（`OverdraftSavingsAccount.java`）

```
=== 5. LSP：違規的子型別會讓正確的程式碼出錯 ===
  提領 80 → 成功? true，餘額 = 0
  再提 1000 → 餘額 = -1000（負數！不變量被打破）
  ★ 編譯器沒有抱怨，但以 SavingsAccount 契約為前提的程式碼全都可能出錯。
```

Liskov 替換原則（Liskov Substitution Principle，LSP）

Barbara Liskov（1987）：若對每個型別 S 的物件 o_1，都存在型別 T 的物件 o_2，使得任何以 T 定義的程式 P 在把 o_2 換成 o_1 後行為不變，則 S 是 T 的子型別。

白話版：**子型別的物件必須能在任何地方替換父型別的物件，而不讓依賴父型別契約的程式出錯。滿足這個條件的子型別稱為行為子型別**（behavioural subtype）——它不只是簽名相容，連行為都相容。

要成為行為子型別，覆寫的方法必須遵守四條規則：

1. 前置條件不能更強
父類別接受的輸入，子類別都要接受。
✗ 父：amount ≥ 0；子：amount ≥ 10

2. 後置條件不能更弱
父類別承諾的結果，子類別都要做到。
✗ 父：餘額不足回 false；子：永遠回 true

3. 不變量必須保持
父類別的資料不變量，子類別不能打破。
✗ 父：balance ≥ 0；子：允許負數

4. 歷史約束
父類別不允許的狀態變化，子類別不能偷偷做。
✗ 父：不可變的點；子：加了 setX()

口訣：要求不能變多，承諾不能變少。
`OverdraftSavingsAccount` 同時違反了規則 2 與 3。

圖 8　行為子型別的四條規則。

重用 vs. 正確：`AccountWithWithdrawalFees` 的抉擇

「每次提款收 5 元手續費」的帳戶，該繼承誰？繼承 `SavingsAccount` 能重用最多程式碼，但 `SavingsAccount` 的契約說「提領 $\leq$ 餘額的金額一定成功」，手續費會讓「剛好領光」失敗——違反規則 2。所以它改成繼承契約較寬鬆的 `BankAccount`（只承諾「嘗試提款並回報結果」），犧牲一點重用換取行為正確。

繼承是為了表達「is-a」的行為關係，不是為了少打幾行字。若只是想借用實作，讓新類別持有一個舊類別的物件（組合，composition）通常比繼承更安全。

⚠ 經典陷阱：正方形是矩形嗎？

數學上是；但若 `Rectangle` 有 `setWidth` / `setHeight`，`Square extends Rectangle` 就會違反 LSP：呼叫者 `r.setWidth(5); r.setHeight(3);` 預期面積 15，正方形卻會得到 9 或 25。問題出在**可變性**——若兩者都是不可變的 record，`Square` 當 `Rectangle` 的子型別就沒有任何問題。LSP 的很多違規都源自「父類別允許的狀態變化，子類別無法同時滿足」。

4.7 練習

✎ 3-1：追蹤動態分派

已知 `PointsAccount` 開戶 10000 點、`deposit(a)` 加 `a + a/100`；`PremiumPointsAccount.deposit(a)` 呼叫 `super.deposit(a * 110 / 100)`。下面程式印出什麼？

```
Account x = new PremiumPointsAccount();
x.deposit(1000);
PointsAccount y = new PointsAccount();
y.deposit(1000);
System.out.println(x.getBalance() + " " + y.getBalance());
```

🔑 3-1 解答

x：$1000\times110/100=1100$，`PointsAccount.deposit(1100)` 加 $1100+11=1111$，得 11111。y：$1000+10=1010$，得 11010。輸出 `11111 11010`。注意 x 的靜態型別是 `Account`，但 `deposit` 執行的仍是 Premium 版本。

✎ 3-2：判斷 LSP

`Stack` 介面規定 `push(x)` 後 `pop()` 回傳 `x`，且 `pop()` 對空堆疊拋出 `IllegalStateException`。以下子類別哪些違反 LSP？為什麼？

(a) `LoggingStack`：每次 push/pop 額外印出一行訊息，其他行為相同。

(b) `BoundedStack`：最多放 10 個，第 11 次 `push` 拋出例外。

(c) `ForgivingStack`：空堆疊 `pop()` 回傳 `null` 而不拋例外。

🔑 3-2 解答

(a) ✅ 合規：契約全部保留，只多了副作用。(b) ❌ 前置條件變強（父類別的 `push` 沒有「數量上限」的要求）——規則 1。(c) ❌ 後置條件變弱（父類別承諾會拋例外，子類別不拋）——規則 2；依賴「空堆疊會拋例外」寫的程式會失控。

5　Week 6 Workshop 導讀：設計聊天協定

這一節的定位

Week 6 / 7 的 Workshop 是 COMP1140 專案「多人聊天系統」的一部分：全班分成伺服器組與客戶端組，最後任何人的客戶端都要能連上任何人的伺服器。這件事能成功的唯一前提，是大家先對一份協定規格達成共識——這正是 Lecture 06A「介面即契約」在系統層級的版本。COMP1110 / 6710 不會考這一節，但它是本週觀念最好的實戰演練。

5.1　什麼是協定？

通訊協定（protocol）

兩個程式之間關於「可以送什麼訊息、每則訊息長什麼樣、在什麼順序下合法、對方該怎麼回應」的完整約定。協定跟程式語言無關：只要遵守規格，用 Java 寫的客戶端可以和用 Python 寫的伺服器對話。

網際網路本身就是靠協定運作的：HTTP（網頁）、SMTP（寄信）、IRC（最早的聊天室）都以 **RFC**（Request for Comments）文件公開規格。它們有幾個共同的設計選擇，值得我們直接借用：

- **純文字、一行一則訊息**。人可以直接用 `nc` / `telnet` 手敲指令測試伺服器，除錯時用肉眼就能看懂封包。
- **每行以「指令字」開頭，後面接以空白分隔的欄位；最後一個欄位可以含空白**（自由文字）。
- **回應帶狀態碼**（HTTP 的 200 / 404、SMTP 的 250 / 550）：程式好判斷，人也好讀。
- **明確的狀態機**：哪些指令在「尚未登入」時合法？哪些要「已加入房間」才能用？規格必須寫清楚。

5.2　一份協定草案

隨附程式 `08_chat_protocol` 實作了下面這份極簡協定（欄位以單一空白分隔；`<text...>` 可含空白；每行以 `\n` 結尾；UTF-8 編碼）：

客戶端 → 伺服器		伺服器 → 客戶端	
`LOGIN <name>`	登入；名字不可含空白	`OK [<text>]`	上一個請求成功
`JOIN <room>`	加入房間（不存在則建立）	`ERR <code> <text>`	上一個請求失敗，附錯誤碼
`SAY <room> <text...>`	在房間發言	`MSG <room> <name> <text...>`	有人（含自己）在房間發言
`QUIT`	離開	`BYE`	伺服器確認並關閉連線

錯誤碼借用 HTTP 的慣例：`400` 格式錯誤、`401` 尚未登入、`403` 不在該房間。

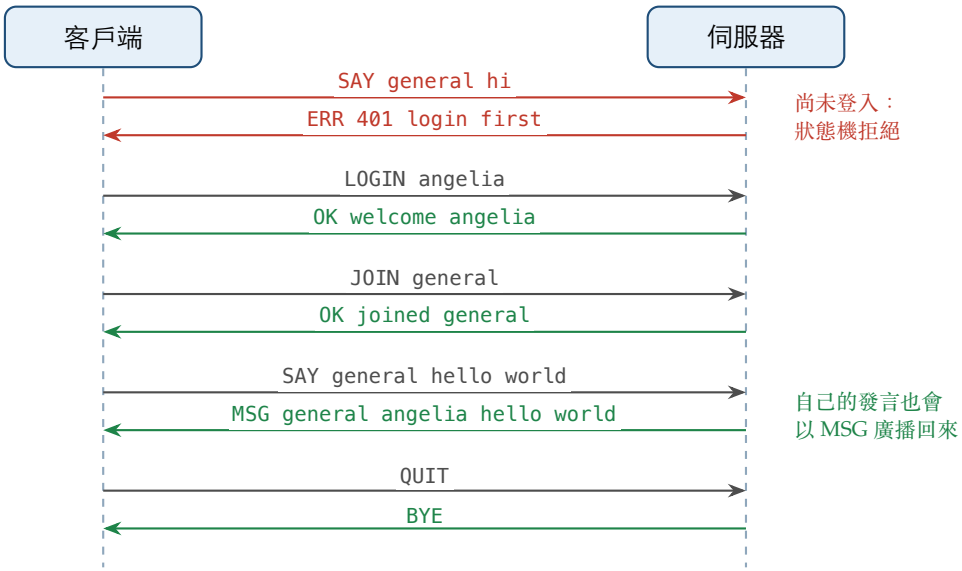


圖 9　EchoClient 與 EchoServer 的一次完整對話（訊息序列圖）。

5.3　把訊息變成型別：sealed interface + record

協定是文字，但程式內部不該到處 split(" ")。好的做法是把「文字 ↔ 結構化資料」的轉換集中在一個地方，其餘程式碼只操作型別安全的物件：

```
public sealed interface ChatMessage
        permits ChatMessage.Login, ChatMessage.Join, ChatMessage.Say, ChatMessage.Quit,
                ChatMessage.Ok, ChatMessage.Err, ChatMessage.Msg, ChatMessage.Bye {

    record Login(String name) implements ChatMessage {}
    record Join(String room) implements ChatMessage {}
    record Say(String room, String text) implements ChatMessage {}
    record Quit() implements ChatMessage {}
    record Ok(String text) implements ChatMessage {}
    record Err(int code, String text) implements ChatMessage {}
    record Msg(String room, String name, String text) implements ChatMessage {}
    record Bye() implements ChatMessage {}

    /** 一行文字 → 訊息；格式不合法就拋 ProtocolException。 */
    static ChatMessage parse(String line) { ... }

    /** 訊息 → 一行文字。性質：parse(encode(m)).equals(m) */
    static String encode(ChatMessage m) {
        return switch (m) {                        // sealed：編譯器確認八種都有處理
            case Login l -> "LOGIN " + l.name();
            case Join j  -> "JOIN " + j.room();
            case Say s   -> "SAY " + s.room() + " " + s.text();
            case Quit q  -> "QUIT";
```

```
            case Ok o    -> o.text().isEmpty() ? "OK" : "OK " + o.text();
            case Err e   -> "ERR " + e.code() + " " + e.text();
            case Msg g   -> "MSG " + g.room() + " " + g.name() + " " + g.text();
            case Bye b   -> "BYE";
        };
    }
}
```

程式 14　訊息 ADT（節錄自 `08_chat_protocol/ChatMessage.java`）

- `sealed interface ... permits ...`：**封閉**的子型別集合——除了列出的八個 record，沒有別人能實作 `ChatMessage`。這讓 `switch` 模式比對可以不用 **`default`**：漏掉任何一種訊息，編譯器就報錯。
- 每種訊息是一個 `record`：不可變、自動有 `equals` / `toString`。`Say("general", "hi")` 讀起來遠比 `parts[1]`、`parts[2]` 清楚。
- 解析時對每種格式錯誤拋 `ProtocolException`（一個自訂的 `RuntimeException`）。伺服器接住它並回 `ERR 400`，**永遠不會因為壞輸入而崩潰**——這是伺服器程式的鐵律。

往返性質（round-trip property）

對協定編碼器最重要的測試只有一條：**任何合法訊息 `m`，`parse(encode(m))` 必須 `equals(m)`**。`ProtocolTest.java` 對八種訊息各測一次（包含含空白與 emoji 的文字），再確認七種壞輸入（空行、未知指令、缺欄位、錯誤碼不是整數……）都被拒絕。第 7 節學了 JUnit 之後，可以把它改寫成 `@Test` 方法。

5.4　Java 網路程式的最小骨架

Java 用 `java.net` 套件裡的 `ServerSocket` 與 `Socket` 建立 TCP 連線，之後就像讀寫檔案一樣，用 `BufferedReader` 與 `PrintWriter` 一行一行收發：

```
try (ServerSocket server = new ServerSocket(port)) {            // 在 port 上「聽」
    while (true) {
        try (Socket client = server.accept();                    // 阻塞直到有人連進來
             BufferedReader in = new BufferedReader(
                     new InputStreamReader(client.getInputStream(), "UTF-8"));
             PrintWriter out = new PrintWriter(client.getOutputStream(), true,
    StandardCharsets.UTF_8)) {
            handle(in, out);                                     // 跟這位客戶完整對話
        }
    }
}

```

```java
static void handle(BufferedReader in, PrintWriter out) throws IOException {
    String name = null;                                          // 狀態：null = 尚未登入
    String line;
    while ((line = in.readLine()) != null) {                     // null = 對方關閉了連線
        ChatMessage reply;
        try {
            ChatMessage m = ChatMessage.parse(line);
            switch (m) {
                case ChatMessage.Login l -> { name = l.name(); reply = new ChatMessage.Ok
    ("welcome " + name); }
                case ChatMessage.Join j  -> reply = new ChatMessage.Ok("joined " + j.room
    ());
                case ChatMessage.Say s   -> reply = name == null
                        ? new ChatMessage.Err(401, "login first")
                        : new ChatMessage.Msg(s.room(), name, s.text());
                case ChatMessage.Quit q  -> { out.println(ChatMessage.encode(new
    ChatMessage.Bye())); return; }
                default -> reply = new ChatMessage.Err(400, "clients may not send " +
    line.split(" ")[0]);
            }
        } catch (ChatMessage.ProtocolException e) {
            reply = new ChatMessage.Err(400, e.getMessage());    // 壞輸入 → ERR，而不是崩
    潰
        }
        out.println(ChatMessage.encode(reply));                  // println 加換行並 flush
    }
}
```

程式 15　單執行緒伺服器（節錄自 EchoServer.java）

在兩個終端機分別執行 `java EchoServer 5555` 與 `java EchoClient localhost 5555`，客戶端會印出：

```
>>> SAY general hi
<<< ERR 401 login first
    解析結果：Err[code=401, text=login first]
>>> LOGIN angelia
<<< OK welcome angelia
    解析結果：Ok[text=welcome angelia]
>>> JOIN general
<<< OK joined general
    解析結果：Ok[text=joined general]
>>> SAY general hello world
<<< MSG general angelia hello world
```

```
    解析結果：Msg[room=general, name=angelia, text=hello world]
>>> DANCE
<<< ERR 400 未知的指令：DANCE
    解析結果：Err[code=400, text=未知的指令：DANCE]
>>> QUIT
<<< BYE
    解析結果：Bye[]
```

⚠ 這個伺服器一次只能服務一個人

`server.accept()` 與 `in.readLine()` 都會阻塞（等到有東西才回來），所以第二位客戶要等第一位 `QUIT` 之後才會被理。真正的聊天室需要多執行緒（每個連線一條 thread）與共享狀態的同步——那是 Week 8 Workshop 的主題。協定設計本身不受影響：規格寫好了，實作可以之後再升級。

5.5　Workshop 討論清單

在 Workshop 中分組討論協定時，至少要對下面這些問題給出寫進規格書的答案：

1. **識別**：名字可以重複嗎？大小寫視為相同嗎？長度上限？允許哪些字元？
2. **狀態機**：未登入時送 `JOIN` 怎麼辦？重複 `LOGIN`？對不存在的房間 `SAY`？
3. **訊息內容**：文字裡出現換行怎麼處理（禁止？跳脫？）。編碼一律 UTF-8？
4. **廣播語意**：自己的發言要不要收到 `MSG`？（本草案：要，這讓 UI 顯示邏輯只有一條路。）
5. **錯誤處理**：客戶端送出無法解析的行，伺服器回 `ERR` 還是直接斷線？伺服器主動斷線前要不要送 `BYE`？
6. **版本**：協定之後一定會改，第一則訊息要不要帶版本號（`HELLO 1.0`）？

✎ 4-1：擴充協定

為協定新增「列出所有房間」（`LIST`）與「私訊」（`DM <name> <text...>`）。寫出：(a) 規格表中的新列，含伺服器回應格式與可能的錯誤碼；(b) 在 `ChatMessage` 中新增對應的 record 與 `parse` / `encode` 分支——注意 `sealed` 的 `permits` 清單與 `encode` 的 `switch` 都要更新，否則無法編譯（這正是 sealed 的價值）。

6　Lab 導讀：遞迴與排序

Week 6 的 Lab 回頭鞏固兩個 Week 4–5 的核心技巧：**遞迴與排序**。它們在本週的位置並不突兀——第 3 節的鏈結串列天生適合遞迴處理，第 8 節的 `Collections.sort` 與 `TreeMap` 都建立在排序之上，而「把演算法當參數傳」正是介面的又一個應用。隨附程式在 `code/` 底下的 `04_lab_recursion_sorting` 資料夾。

6.1　遞迴：兩個部分，一個信念

遞迴（recursion）

一個函式在定義中**呼叫自己**。每個正確的遞迴函式都有：

1. **基底情況**（base case）：問題小到可以**直接**回答，不再遞迴。
2. **遞迴情況**（recursive case）：把問題化成**更小的同類問題**，假設小問題已解，再把答案組合起來。

而且必須能說明「每次遞迴呼叫都讓某個量**嚴格變小**並終將抵達基底」，否則永不終止。

```java
/** n! = n × (n-1)!，0! = 1。前置條件：n >= 0 */
static long factorial(int n) {
    if (n == 0) return 1;                // 基底情況
    return n * factorial(n - 1);         // 遞迴情況：n 嚴格變小
}

/** 陣列 arr[from..] 的總和：第一個元素 + 其餘元素的總和。 */
static int sum(int[] arr, int from) {
    if (from >= arr.length) return 0;    // 基底：沒有元素了
    return arr[from] + sum(arr, from + 1);
}

/** 迴文判斷：頭尾相同，且中間也是迴文。 */
static boolean isPalindrome(String s) {
    if (s.length() <= 1) return true;
    if (s.charAt(0) != s.charAt(s.length() - 1)) return false;
    return isPalindrome(s.substring(1, s.length() - 1));
}
```

程式 16　節錄自 `Recursion.java`

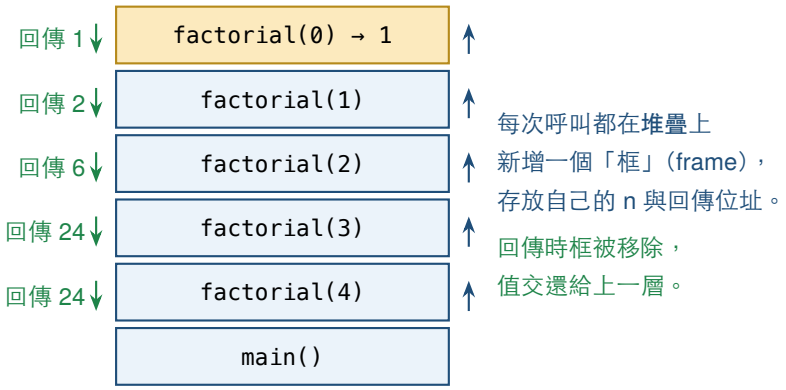


圖 10　`factorial(4)` 的呼叫堆疊。往上是呼叫（問題變小），往下是回傳（答案組合）。

相信遞迴（the recursive leap of faith）

寫遞迴情況時，不要在腦中展開所有層次。只做三件事：(1) 確認呼叫的是「更小」的問題；(2) 假設那個呼叫會正確回傳；(3) 用它的結果組出這一層的答案。`isPalindrome` 只需說：「如果頭尾相同，且假設 `isPalindrome(中間)` 是對的，那整體就是對的。」這其實就是數學歸納法。

6.2　輔助參數與對外包裝

很多遞迴需要「多出來的參數」記錄進度（`sum` 的 `from`、二分搜尋的 `lo` / `hi`）。呼叫者不該知道這些細節，所以慣例是寫一個**對外的簡潔版本**包住它——這和封裝的精神一致：

```
static int binarySearch(int[] haystack, int needle, int lo, int hi) {
    if (lo > hi) return -1;                          // 基底：範圍變空，確定不存在
    int mid = (lo + hi) / 2;
    if (haystack[mid] == needle)      return mid;
    else if (haystack[mid] < needle)  return binarySearch(haystack, needle, mid + 1, hi);
      // 右半
    else                              return binarySearch(haystack, needle, lo, mid - 1);
      // 左半
}

/** 對外版本：呼叫者不需要知道 lo/hi。 */
static int binarySearch(int[] haystack, int needle) {
    return binarySearch(haystack, needle, 0, haystack.length - 1);
}
```

程式 17　遞迴二分搜尋（分治法）

二分搜尋每一層把範圍**減半**，所以只需 $O(\log n)$ 層——這是**分治法**（divide and conquer）的典型：**分**（切成更小的子問題）、**治**（遞迴解決）、**合**（組合答案）。快速冪 `power(base, exp)`

也是：算一次 power(base, exp/2) 再平方，2^{1000} 只需約 10 層。

StackOverflowError

每層遞迴都占用一個堆疊框，而堆疊有限（預設約幾 MB）。countDown(1_000_000) 會在數萬層時拋出 StackOverflowError——即使邏輯完全正確。兩個常見原因：(1) 忘了基底情況或永遠碰不到它（factorial(-1) 會一路往負數跑）；(2) 問題規模本身太大（用遞迴走訪一百萬個節點的鏈結串列）。後者改用迴圈即可。**遞迴適合「結構本身是遞迴的」問題**：樹、分治、巢狀結構；一維線性走訪用迴圈就好。

6.3　排序：兩條後置條件

排序的規格

sort(int[] arr) 執行後必須同時滿足：

1. arr 遞增排序：對所有 i，arr[i] <= arr[i+1]；
2. arr 是原本元素的一個排列：沒有多、沒有少、沒有變。

很多人只測第 1 條。一個把所有元素都設成 0 的「排序」也滿足第 1 條！Sorting.checkSorter 用 Arrays.equals(toSort, expected) 一次驗兩條（expected 由 Arrays.sort 產生）。

四種演算法的核心想法與**迴圈不變量**（每回合開始時都成立的事）：

演算法	每一回合做什麼	迴圈不變量
選擇排序	從未排序區找最小值，換到未排序區最前面	arr[0..i-1] 已排序，且都 ≤ 未排序區的每個元素
氣泡排序	相鄰兩兩比較，把最大值一路「冒」到尾端；整趟沒交換就提前結束	arr[end+1..] 已排序，且都 ≥ 前面的每個元素
插入排序	取下一個元素，向左挪出空位後「插入」正確位置	arr[0..i-1] 已排序（但不一定是最小的那些）
合併排序	切半 → 遞迴排序兩半 → 合併兩個已排序陣列	（遞迴）子呼叫回傳的陣列已排序

```
static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        // INV: arr[0..i-1] 已排序
        int x = arr[i];
        int j = i - 1;
        while (j >= 0 && less(x, arr[j])) {
            arr[j + 1] = arr[j];            // 比 x 大的元素往右挪一格
            j = j - 1;
        }
```

```
        arr[j + 1] = x;                     // 放進空出來的位置
    }
}

static int[] mergeSort(int[] input, int start, int end) {
    if (end < start)  return new int[]{};              // 基底：空範圍
    if (end == start) return new int[]{input[start]};  // 基底：單一元素本身就是排好的
    int mid = (start + end) / 2;                       // 分
    int[] left  = mergeSort(input, start, mid);        // 治
    int[] right = mergeSort(input, mid + 1, end);
    return merge(left, right);                         // 合：O(left + right)
}
```

程式 18　插入排序與合併排序（節錄自 `Sorting.java`）

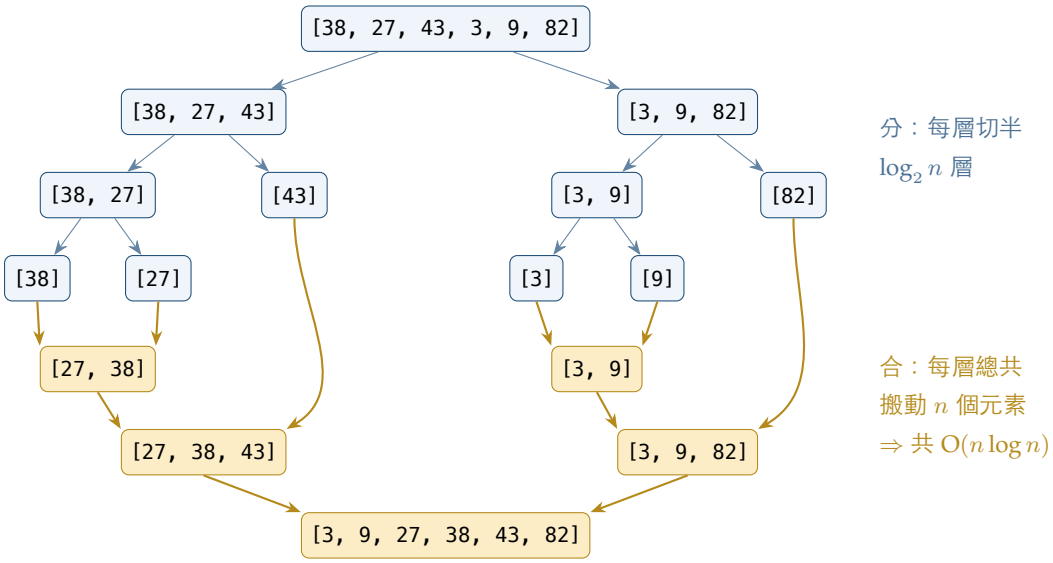


圖 11　合併排序的遞迴樹：藍色向下是「分」，金色向上是「合」。

6.4　實測：比較次數

`Sorting.java` 把每次 `<` 比較都記錄下來，對 $n = 2000$ 的三種資料實測（`java -ea Sorting`）：

演算法	隨機	已排序（最佳）	反向（最差）	Big-O（最差）
選擇排序	1,999,000	1,999,000	1,999,000	$O(n^2)$，永遠 $n(n-1)/2$
氣泡排序	1,994,905	1,999	1,999,000	$O(n^2)$；已排序時 $O(n)$
插入排序	999,913	1,999	1,999,000	$O(n^2)$；已排序時 $O(n)$
合併排序	19,363	11,088	10,878	$O(n \log n)$，與輸入順序幾乎無關

- **選擇排序完全不看資料**：不管輸入如何都是 $\binom{n}{2} = 1{,}999{,}000$ 次。
- **氣泡與插入排序對「幾乎排好」的資料很快**（$n - 1 = 1999$ 次），這是它們在小型或近乎有序資料上仍被使用的原因。

- 合併排序約 $n\log_2 n \approx 2000 \times 11 = 22{,}000$ 的量級，比 n^2 少了一百倍；n 越大差距越誇張。代價是需要 $O(n)$ 的額外空間。Java 的 `Arrays.sort` 對物件用的是合併排序的改良版（TimSort）。
- **穩定性**（stability）：相等的元素排序後保持原本的相對順序。`merge` 中「相等取左邊」就是為了穩定。當你先依名字排序、再依年級排序，穩定排序能保證同年級的人仍按名字排列。

6.5　把演算法當參數：`Sorter` 介面

測試程式如何對四種排序跑同一組測試？定義一個只有一個方法的介面，再把**方法參照**（method reference）當成它的實作傳進去：

```
interface Sorter { void sort(int[] arr); }          // 函式介面：只有一個抽象方法

static void checkSorter(String name, Sorter sorter) {
    for (int[] original : cases) {
        int[] toSort = original.clone(), expected = original.clone();
        Arrays.sort(expected);
        sorter.sort(toSort);                         // 呼叫「被傳進來的演算法」
        assert Arrays.equals(toSort, expected) : name + " 失敗：" + Arrays.toString(
    toSort);
    }
}
checkSorter("insertionSort", Sorting::insertionSort);   // Sorting::insertionSort 自動變
    成一個 Sorter
checkSorter("mergeSort", Sorting::mergeSort);
```

這是第 3 節「介面 = 契約」的另一種用法：契約只有一條「給你一個陣列，把它排好」，任何滿足契約的函式都能插進來。第 8 節的 `Comparator` 與第 10 節的 `Listener` 都是同一個模式。

6.6　Lab 練習

5-1：遞迴基本功

用遞迴（不可用迴圈）實作並用 `assert` 測試：(a) `static int countOccurrences(int[] arr, int x, int from)`——`x` 在 `arr[from..]` 出現的次數；(b) `static boolean isSortedRec(int[] arr, int from)`——`arr[from..]` 是否遞增；(c) `static int gcd(int a, int b)`——歐幾里得算法：$\gcd(a,0) = a$，$\gcd(a,b) = \gcd(b, a \bmod b)$。對每一題寫出：基底情況是什麼？哪個量嚴格變小？

5-1 提示

(a) 基底 `from >= arr.length` 回 0；遞迴 `(arr[from] == x ? 1 : 0) + countOccurrences(arr, x, from + 1)`；變小的量是 `arr.length - from`。(b) 基底 `from >= arr.length - 1` 回 `true`；遞迴 `arr[from] <= arr[from+1] && isSortedRec(arr, from + 1)`。(c) 基底 `b == 0` 回 `a`；遞迴 `gcd(b, a % b)`；變小的量是 `b`（因為 $a \bmod b < b$）。

5-2：排序的第二條後置條件

寫一個故意錯誤的 `fakeSort`：把陣列每個元素都設成 0。把它交給 `checkSorter`，觀察哪一個 `assert` 抓到它。接著寫 `isPermutation(int[] a, int[] b)`（不使用 `Arrays.sort`；提示：對每個元素計數），使測試不必依賴標準函式庫的排序。

5-3：改成遞迴的鏈結串列

回到第 3 節的 `LinkedSimpleList`，把 `get(int index)` 改寫成遞迴的私有輔助方法 `get(Node current, int index)`：基底 `index == 0` 回傳 `current.value`；否則 `get(current.next, index - 1)`。比較它與迴圈版本的可讀性，並說明為什麼串列很長時仍應保留迴圈版本。

第二部分

Week 7：測試、泛型與雜湊表

Lecture 07A • Lecture 07B • Week 7 Workshop

7　Lecture 07A（上）：JUnit 測試與測試驅動開發

到目前為止我們用 `assert` 加一個 `main` 來檢查程式。它能用，但有三個問題：所有檢查擠在一個方法裡，第一個失敗就會讓後面的全部跳過；沒有統一的報告；而且很難「對同一份契約測多種實作」。**JUnit** 是 Java 世界的標準測試框架，正好解決這三件事。

7.1　為什麼要寫自動化測試？

- **回歸保護**（regression）：每次修改後一鍵重跑，立刻知道有沒有弄壞以前能動的東西。
- **可執行的規格**：測試把介面註解裡的前置 / 後置條件變成**會跑的程式**，比文字更精確、永不過時。
- **設計回饋**：難以測試的程式通常也難以使用；先寫測試會逼你把介面設計好。

黑箱測試與白箱測試

黑箱測試（black-box）只透過**公開介面**操作物件，只依據**契約**判斷對錯，完全不看實作。同一份黑箱測試可以套用到該介面的**每一個**實作。
白箱測試（white-box）知道實作細節（例如「陣列滿了會倍增」），可以針對特定分支設計輸入，但換一種實作就失效。本節以黑箱測試為主，並會標出唯一一個白箱例外。

7.2　JUnit 5 入門

一個 JUnit 測試就是一個標了 `@Test` 的方法。JUnit 會自動找到所有這樣的方法、逐一執行、統計結果：

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;   // assertEquals、assertTrue…

class CalculatorTest {
    @Test
    void additionWorks() {
        assertEquals(4, 2 + 2);                      // (預期值, 實際值)
    }
}
```

程式 19　最小的 JUnit 5 測試

常用的斷言（都在 `org.junit.jupiter.api.Assertions`）：

斷言	意義
`assertEquals(expected, actual)`	兩者 equals；**預期值在前**，失敗訊息才會正確。
`assertTrue(cond)` / `assertFalse(cond)`	條件為真 / 假。
`assertNull(x)` / `assertNotNull(x)`	x 是 / 不是 null。
`assertArrayEquals(expected, actual)`	兩個陣列長度相同且逐元素相等（== 比陣列只比參照，永遠不要用）。
`assertThrows(E.class, () -> ...)`	執行 lambda 時必須拋出 E（或其子類別）；回傳該例外供進一步檢查。
`fail("msg")`	無條件失敗，用於「不該執行到這裡」。

每個斷言都可以多接一個字串參數當失敗訊息：`assertEquals(1, gl.count(), "重複加入不應被重複計算")`。

JUnit 的生命週期：每個測試都是全新的世界

JUnit 對每一個 `@Test` 方法都會 new 一個新的測試類別物件來執行。所以測試之間不會共享欄位狀態，順序也不重要——這是刻意的：一個測試留下的垃圾不該影響另一個。標了 `@BeforeEach` 的方法會在每個測試前執行，適合放「準備一份乾淨的受測物件」的程式碼。

7.3　案例：GuestList 的黑箱測試

Lecture 07A 的主角是「賓客名單」——一個集合語意的 ADT：一個人要嘛會來、要嘛不會來，同一個人 add 兩次等於 add 一次。

```
public interface GuestList {
    void add(String name);          // 記錄某位賓客會來
    void remove(String name);       // 記錄某位賓客不來了（原本不在名單上則什麼都不做）
    boolean isComing(String name);  // 該賓客目前是否會來
    int count();                    // 目前會來的人數（不重複計算）
    String[] coming();              // 會來的名字（防禦性複製）
}
```

程式 20　`05_junit_tdd/GuestList.java`

測試只認識這個介面。為了讓同一套測試能套用到多種實作，我們把測試類別寫成抽象的，把「要測哪一種實作」交給一個抽象的工廠方法：

```
abstract class GuestListTest {

    abstract GuestList create();            // 由子類別決定要測哪一種實作

    private GuestList gl;

    @BeforeEach
    void setUp() {
```

```
        gl = create();                          // 每個測試開始前都拿到一份全新的空名單
    }

    @Test
    @DisplayName("新名單是空的")
    void emptyListHasNobody() {
        assertFalse(gl.isComing("Fabian"));
        assertEquals(0, gl.count());
        assertEquals(0, gl.coming().length);
    }

    @Test
    @DisplayName("集合語意：同一個人 add 兩次等於 add 一次")
    void addingTwiceIsIdempotent() {
        gl.add("Fabian");
        gl.add("Fabian");
        assertTrue(gl.isComing("Fabian"));
        assertEquals(1, gl.count(), "重複加入不應被重複計算");
        gl.remove("Fabian");
        assertFalse(gl.isComing("Fabian"), "移除一次之後必須完全消失");
        assertEquals(0, gl.count());
    }

    @Test
    @DisplayName("coming() 回傳的是防禦性複製，改它不影響名單")
    void comingReturnsDefensiveCopy() {
        gl.add("Fabian");
        gl.add("Nicholas");
        String[] snapshot = gl.coming();
        snapshot[0] = "Intruder";                       // 惡意（或不小心）改動回傳的陣列
        assertFalse(gl.isComing("Intruder"), "封裝：外部改動不得影響內部狀態");
        assertEquals(2, gl.count());
    }

    @Test
    @DisplayName("assertThrows 示範：null 名字不合法")
    void nullNameIsRejected() {
        IllegalArgumentException e =
                assertThrows(IllegalArgumentException.class, () -> gl.add(null));
        assertNotNull(e.getMessage());
    }
    // ……共 9 個測試：remove 只影響該人、從空名單 remove、移除後再加、1000 人觸發倍增……
}
```

```
class LazyGuestListTest extends GuestListTest {          // 對「懶惰」實作跑整套測試
    @Override GuestList create() { return new GuestArrayList(); }
}
class StrictGuestListTest extends GuestListTest {        // 對「嚴格」實作跑同一套測試
    @Override GuestList create() { return new GuestArrayUniqueList(); }
}
```

程式 21　節錄自 GuestListTest.java

測試案例從哪裡來？

每一個測試都對應契約裡的一句話，或一個邊界：

- 空：新名單沒有人、count() == 0、coming() 長度 0。
- 單一與多個：add 一人、兩人；remove 只影響被移除的那一位。
- 重複：同一人 add 兩次仍算一人；remove 一次後完全消失（這條最容易讓實作出錯）。
- 不存在：對空名單或不在名單上的人 remove，不能出錯。
- 來回：移除後再加回來。
- 大量：1000 人——確保內部陣列放大時不丟資料（黑箱看不到倍增，但「很多」總是值得測）。
- 封裝：改動 coming() 的回傳值不得影響名單。
- 例外：assertThrows 檢查 add(null)。注意：介面沒規定 null 行為，所以這一條其實是白箱測試，只對我們的實作成立。

7.4　兩種實作，一套測試

「懶惰」實作 GuestArrayList 的 add 不檢查重複，直接塞進陣列；重複要等 remove 時一次清光，count 則只在名字「第一次出現」時計數：

```
public class GuestArrayList implements GuestList {
    String[] data = new String[2];       // package-private：子類別可直接存取
    int used = 0;                        // data[0..used-1] 是「用過」的格子（可能含 null 洞與重複）

    @Override
    public void add(String name) {
        if (name == null) throw new IllegalArgumentException("賓客名字不可為 null");
        if (this.used == this.data.length) {          // 滿了：換一個兩倍大的陣列
            String[] old = this.data;
            this.data = new String[this.used * 2];
            for (int i = 0; i < this.used; i++) this.data[i] = old[i];   // O(n) 複製
```

```
        }
        this.data[this.used] = name;
        this.used = this.used + 1;
    }

    @Override
    public int count() {
        int n = 0;
        for (int i = 0; i < this.used; i++) {
            if (this.data[i] != null && indexOf(this.data[i]) == i) n = n + 1;   // 只數第一次出現
        }
        return n;
    }
    // remove：掃完整個陣列，把所有同名都設成 null；isComing：線性搜尋；coming：複製到新陣列
}
```

程式 22　節錄自 GuestArrayList.java

「嚴格」實作 GuestArrayUniqueList 繼承它，只改 add（先查 isComing），因而多了一條不變量「非 null 的名字互不相同」，於是 remove 找到第一個就能停、count 不必查重。兩者對外行為完全一致——是 LSP 合規的子型別——所以 18 個測試（9 × 2 種實作）全部通過：

```
Test run finished after 185 ms
[         5 containers found      ]
[         5 containers successful ]
[        18 tests found           ]
[        18 tests started         ]
[        18 tests successful      ]
[         0 tests failed          ]
```

測試是「對介面」寫的，所以能重用

兩種實作的內部差很多(一個有洞和重複、一個沒有)，但測試一行都不用改。這件事只有在測試完全不碰實作細節時才成立——變數宣告成 GuestList，不要宣告成 GuestArrayList；永遠不要在測試裡讀 data 或 used。

7.5 測試驅動開發

測試驅動開發（Test-Driven Development，TDD）

先寫測試，再寫程式，以極短的循環反覆：

1. 紅：為下一個小功能寫一個測試，執行——它必須失敗（否則測試沒測到東西）。
2. 綠：寫最少的程式讓測試通過。允許醜、允許寫死。
3. 重構：整理程式（去重複、改名、抽方法），過程中測試必須一直保持綠色。

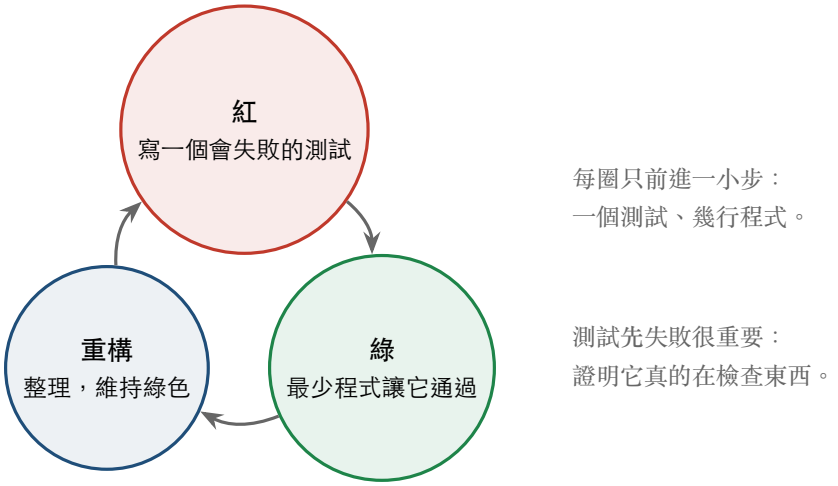


圖 12　TDD 的紅- 綠- 重構循環。

用 `GuestList` 走一遍 TDD 的前三圈：

1. 紅：寫 `emptyListHasNobody`。此時 `GuestArrayList` 連類別都沒有，編譯失敗也算「紅」。綠：建立類別，`isComing` 直接 `return false;`、`count` 直接 `return 0;`。通過。
2. 紅：寫 `addThenIsComing`（add 之後 isComing 為 true）。寫死的 `false` 失敗。綠：加上陣列與 `used`，`add` 存進去，`isComing` 線性搜尋。
3. 紅：寫 `addingTwiceIsIdempotent`。`count` 回傳 2，失敗——**這個測試逼我們決定**：是在 `add` 時擋重複（嚴格），還是在 `count` / `remove` 時處理（懶惰）？兩條路都通，於是有了兩種實作。重構：兩種實作共用的程式碼抽到父類別。

TDD 的實際好處

- 你永遠只需要專注「讓這一個測試通過」，問題被切成小塊。
- 寫完時測試覆蓋率自然接近 100%，而且每個測試都曾經失敗過——你知道它們真的有用。
- 「最少程式」原則防止過度設計；「重構」步驟確保程式不會越寫越亂。
- 在作業與專案中，TDD 讓你隨時都有一個可以交出去的版本——只是功能還不全。

7.6 防禦性複製

`coming()` 若直接 `return this.data;`，測試 `comingReturnsDefensiveCopy` 會失敗：呼叫者改陣列就等於改名單內部。規則很簡單：

防禦性複製（defensive copy）

當一個方法要回傳內部的可變物件（陣列、`List`、`StringBuilder`……）時，回傳一份副本；當一個方法接收外界的可變物件並要長期保存時，也先複製一份再存。不可變的東西（`String`、`int`、`record`）不需要複製。

```
String[] copy = new String[count()];      // 全新陣列
// ……把非 null 且第一次出現的名字複製進去……
return copy;                              // 呼叫者改 copy 不影響 data

// 第 3 節的另一個例子：
public List<String> getLog() { return new ArrayList<>(this.log); }
```

7.7 陣列倍增：為什麼滿了要「乘二」而不是「加一」？

`GuestArrayList.add` 在陣列滿時把容量加倍。每次放大都要 $O(n)$ 複製，聽起來很貴，但因為容量呈 $2, 4, 8, 16, \ldots$ 成長，放大的次數只有 $\log_2 n$ 次，把總複製成本平均到每一次 `add` 上，每次只多付常數——這叫攤銷 $O(1)$。若改成每次只加一格，每次 `add` 都要複製整個陣列，n 次 `add` 就變成 $O(n^2)$。完整的分析（含圖）在第 9.4 節，因為雜湊表的擴容用的是一樣的道理。

7.8 在命令列執行 JUnit

IntelliJ 會自動處理 JUnit；若想在終端機執行，下載 JUnit 的獨立執行檔（一個 jar）即可：

```
$ cd code/05_junit_tdd
$ curl -LO https://repo1.maven.org/maven2/org/junit/platform/junit-platform-console-
    standalone/1.12.2/junit-platform-console-standalone-1.12.2.jar
$ javac -cp junit-platform-console-standalone-1.12.2.jar *.java
$ java -jar junit-platform-console-standalone-1.12.2.jar execute --class-path . --scan-
    class-path
```

7.9 練習

6-1：為 ChatHistory 寫黑箱測試

為第 3 節的 ChatHistory 寫一個抽象的 ChatHistoryTest，至少包含：新紀錄 size() == 0；add 三則後 size() == 3；ErasableChatHistory.erase() 後 size() == 0。用兩個子類別分別測 LinkedListChatHistory 與 CircularBufferChatHistory(3)。思考：「add 四則後 size() 是多少？」對兩種實作答案不同——這代表什麼？介面契約該怎麼改？

6-1 提示

環狀緩衝區會回 3，鏈結串列會回 4。這表示 ChatHistory 的契約沒有說清楚「size 是否可能小於 add 的次數」。兩種修法：(a) 把契約寫成「size() 回傳目前保存的則數，實作可以丟棄舊訊息」，測試改成 assertTrue(h.size() <= 4 && h.size() >= 1)；或 (b) 把「有容量上限」抽成另一個介面 BoundedChatHistory。黑箱測試常常就是這樣暴露契約的漏洞——這是它最有價值的副作用。

6-2：先紅再綠

用 TDD 實作一個 Counter 類別：increment()、reset()、value()，以及「value() 永遠不會超過建構子給的上限，超過時 increment() 拋出 IllegalStateException」。嚴格遵守：每次只寫一個測試、先看到它失敗、再寫最少的程式。記錄你總共跑了幾圈。

8 Lecture 07A（下）：轉型、泛型與 Collections

第 3 節的 SimpleList 用 Object 當元素型別，「什麼都能放」——但代價是取出來時只剩 Object，得自己轉型回去，而且編譯器無法阻止你把 Integer 塞進一個「應該只放 String」的串列。本節先把轉型講清楚，再介紹 Java 的解法：泛型，最後導覽建立在泛型之上的 **Collections** 框架。隨附程式在 code/06_generics_collections。

8.1 Object 與轉型

```
Object x = "hello";         // 向上轉型（upcast）：自動，永遠安全
Object y = 42;              // 自動裝箱：int → Integer，再向上轉型
// x.length();              // 編譯錯誤：編譯器只知道 x 是 Object

String s = (String) x;      // 向下轉型（downcast）：「相信我，它其實是 String」
System.out.println(s.length());                 // 5

Integer bad = (Integer) x; // 編譯器放行，執行期拋出 ClassCastException！
```

程式 23　節錄自 Demo.java

轉型（cast）

向上轉型（子型別 → 父型別）是自動且永遠安全的，因為每個 String 都是 Object。
向下轉型（父型別 → 子型別）要明寫 (T) expr，編譯器不檢查正確性，而是在執行期確認物件的實際型別；不符就拋 ClassCastException。
轉型不會改變物件，只是換一副眼鏡看同一個東西。

安全的向下轉型要先問過 instanceof。Java 16 起的模式比對（pattern matching）讓檢查與宣告變數一步完成：

```
if (y instanceof Integer n) {          // 若 y 真的是 Integer，就把它綁定到變數 n
    System.out.println(n + 1);         // 43，不需要再寫 (Integer) y
}
```

轉型是一種「程式碼氣味」

程式裡到處出現 (String)、instanceof，通常表示型別資訊在某處被丟掉了（例如把東西放進 Object[]）。與其事後轉回來，不如一開始就讓編譯器記住型別——這正是泛型的用途。合理的轉型只剩少數情況：實作 equals(Object o) 時、以及處理真正異質的資料。

8.2 泛型：把型別當參數

泛型（generics）與型別參數（type parameter）

在類別或介面名稱後加上 <T>，T 就成為一個型別的佔位符，可以在該類別內當成型別使用。使用時以型別引數填入：GuestList<String> 是名字的名單，GuestList<Integer> 是學號的名單。同一份程式碼，對每一種型別都型別安全，且不需要轉型。

```
public interface GuestList<T> {
    void add(T guest);
    void remove(T guest);
    boolean isComing(T guest);
    int count();
    T[] coming();              // ★ 這一行藏著一個陷阱，見 7.3
}
```

程式 24　泛型版的 GuestList（06_generics_collections/GuestList.java）

```
GuestList<String> names = new ArrayGuestList<>();   // <>（鑽石）：讓編譯器推論型別引數
names.add("Fabian");
// names.add(42);          // 編譯錯誤！42 不是 String——錯誤在編譯期就被抓到
String first = names.comingList().get(0);           // 不需要 (String) 轉型

```

```
GuestList<Integer> ids = new ArrayGuestList<>();
ids.add(1001);                                    // int 自動裝箱成 Integer
```

幾種常見的泛型寫法：

```
/** 泛型方法：<T> 寫在回傳型別前面；T 由編譯器從引數推論。 */
static <T> T firstOrDefault(List<T> list, T fallback) {
    return list.isEmpty() ? fallback : list.get(0);
}

/** 有界型別參數：T 必須是 Comparable<T>，才保證能呼叫 compareTo。 */
static <T extends Comparable<T>> T max(List<T> list) {
    T best = list.get(0);
    for (T x : list) if (x.compareTo(best) > 0) best = x;
    return best;
}

/** 兩個型別參數的泛型 record。 */
public record Pair<A, B>(A first, B second) {
    public static <A, B> Pair<A, B> of(A first, B second) { return new Pair<>(first,
    second); }
    public Pair<B, A> swap() { return new Pair<>(this.second, this.first); }
}

Pair<String, Integer> p = Pair.of("Liam", 30);      // p.swap() 是 Pair<Integer, String>
```

程式 25　泛型方法、有界型別參數、多個型別參數（節錄自 Demo.java 與 Pair.java）

命名慣例

型別參數用單一大寫字母：T（type）、E（element）、K / V（key / value）、A / B。看到 Map<K, V> 就知道是「鍵型別 K、值型別 V」。

8.3　型別擦除：泛型在執行期不存在

Java 的泛型只存在於編譯期。編譯器用 T 檢查完型別後，就把它擦除（erase）成 Object，產生的 bytecode 跟第 3 節的 Object 版本幾乎一樣。這帶來幾個真實的限制：

1. 不能 **new T[n]**：執行期不知道 T 是誰。標準做法是內部用 Object[]，取出時轉型：

```
private Object[] data = new Object[2];   // 不能是 T[]！

@Override
@SuppressWarnings("unchecked")           // 我知道這個轉型沒被檢查，我負責它的安全
public T[] coming() {
```

```
    return (T[]) Arrays.copyOf(this.data, this.used);   // 未受檢查的轉型（unchecked
     cast）
}
```

程式 26　節錄自 ArrayGuestList.java

只要封裝保證「陣列裡只放過 T」，這個轉型在實務上是安全的。

2. **T[] 回傳值的陷阱**：上面那個陣列在執行期真的是 Object[]，所以：

```
Object[] ok   = names.coming();    // OK
String[] boom = names.coming();    // 編譯通過，執行期 ClassCastException！
```

```
=== 4. 型別擦除的陷阱 ===
  Object[] ok = [Fabian, Nicholas]
  String[] boom = names.coming() → ClassCastException！
   （T[] 在執行期被擦除成 Object[]，這是 Java 泛型的真實限制）
  正解 comingList() = [Fabian, Nicholas]（用 List<T> 就沒有這個問題）
```

編譯器在 String[] boom = ... 這行偷偷插入了一個「檢查它是 String[]」的指令，而它不是。**正解：泛型類別的方法回傳 List<T>，不要回傳 T[]**——標準函式庫也是這麼做的（Collection.toArray() 回傳 Object[]）。

3. **型別引數不能是基本型別**：沒有 List<int>，只有 List<Integer>。Java **會自動裝箱**（autoboxing）（int → Integer）與拆箱，大多時候無感，但有一個著名陷阱：

```
Integer a = 127, b = 127, c = 1000, d = 1000;
a == b          // true ——小整數（-128..127）被快取，碰巧是同一個物件
c == d          // false！不同的 Integer 物件
c.equals(d)     // true ——比較物件永遠用 equals
```

8.4　Java Collections 框架

自己寫 LinkedSimpleList 是為了理解原理；實務上一律使用 java.util 提供的、經過千錘百鍊的泛型集合。整個框架就是一組**介面**（ADT）加上多種**實作**（資料結構）——本週所有觀念的集大成：

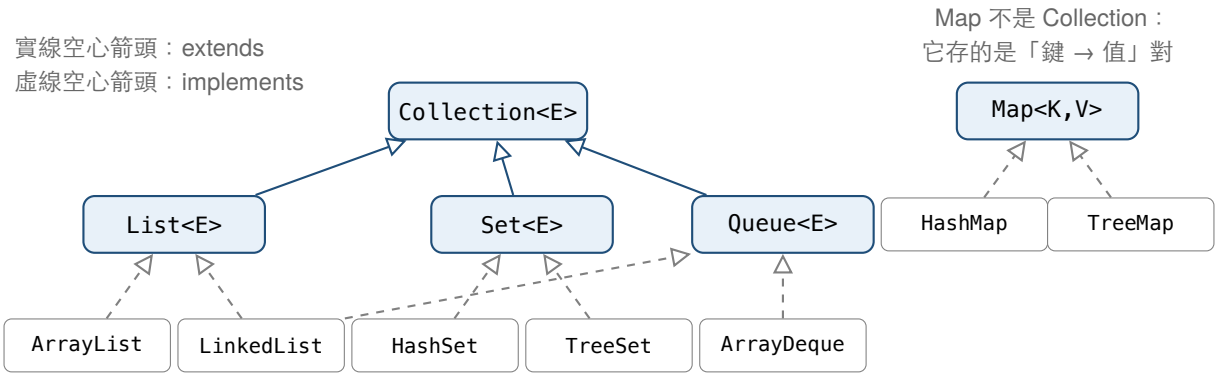


圖 13　Collections 框架的核心介面與常用實作（藍色是介面 / ADT，白色是資料結構）。

介面	語意	實作與效能
`List<E>`	有順序、可重複、用索引存取	`ArrayList`：get $O(1)$，尾端 add 攤銷 $O(1)$，中間插刪 $O(n)$（預設選擇）。`LinkedList`：兩端插刪 $O(1)$，`get(i)` $O(n)$。
`Set<E>`	不重複、沒有索引	`HashSet`：`add` / `contains` 平均 $O(1)$，順序不定。`TreeSet`：$O(\log n)$，走訪時由小到大。
`Map<K,V>`	鍵 → 值；鍵不重複	`HashMap`：`put` / `get` 平均 $O(1)$。`TreeMap`：$O(\log n)$，依鍵排序。
`Queue<E>` / `Deque<E>`	先進先出 / 雙端	`ArrayDeque`：兩端 $O(1)$；也可當堆疊用。

```java
List<String> fruits = new ArrayList<>();      // 宣告用介面 List，實作用 ArrayList
fruits.add("banana"); fruits.add("apple"); fruits.add("cherry"); fruits.add("apple");
System.out.println(fruits.get(1) + " " + fruits.contains("cherry"));   // apple true
Collections.sort(fruits);                     // [apple, apple, banana, cherry]
fruits.remove("apple");                       // 只移除第一個
for (String f : fruits) System.out.println(f);   // for-each：走訪任何 Collection

Set<String> guests = new TreeSet<>();         // 自動排序、不重複
guests.add("Liam"); guests.add("Thomas"); guests.add("Alice"); guests.add("Liam");
System.out.println(guests);                   // [Alice, Liam, Thomas]

Map<String, Integer> ages = new HashMap<>();
ages.put("Liam", 30);
ages.put("Liam", 31);                         // 同一個鍵：覆蓋
ages.get("Nobody");                           // null（不存在的鍵）
ages.getOrDefault("Nobody", -1);              // -1

Map<Character, Integer> freq = new TreeMap<>();          // 經典應用：計數
for (char c : "mississippi".toCharArray()) {
    freq.put(c, freq.getOrDefault(c, 0) + 1);
}
```

```
System.out.println(freq);                    // {i=4, m=1, p=2, s=4}

List<Integer> nums = List.of(3, 1, 4, 1, 5);  // of()：不可變的集合
nums.add(9);                                  // UnsupportedOperationException！
List<Integer> mutable = new ArrayList<>(nums);          // 要可變就複製一份
```

程式 27　Collections 導覽（節錄自 Demo.java）

三個使用習慣

1. 變數宣告用介面，new 用實作：List<String> xs = new ArrayList<>();。日後改成 LinkedList 只改一個字；方法參數也宣告成 List / Collection，接受度最大。
2. 不確定就用 **ArrayList** ／ **HashMap** ／ **HashSet**：它們是各類別的預設選擇；需要排序走訪才換 Tree*，需要頻繁在前端插刪才換 ArrayDeque。
3. **List.of** ／ **Set.of** ／ **Map.of** 建立的集合不可變：適合常數表；要改就 new ArrayList<>(…) 複製。

介面裡的 default 方法

Map.getOrDefault、List.sort、Collection.forEach 都是介面裡帶有實作的方法（default 關鍵字），Java 8 引入它是為了在不弄壞既有實作的前提下擴充介面。它的意義是：實作類別可以覆寫它，但不覆寫也能用。介面仍然不能有實例欄位，所以 default 方法只能透過其他抽象方法完成工作。

8.5　練習

7-1：把 SimpleList 泛型化

把第 3 節的 SimpleList 與 LinkedSimpleList 改成 SimpleList<T> / LinkedSimpleList<T>。內部類別 Node 的 value 欄位型別要改成什麼？get 的回傳型別呢？改完後 Main 裡的 list.get(1).equals("banana") 可以改寫成什麼更簡潔的形式？

7-1 提示

Node 是內部（非 static）類別，可以直接使用外層的 T：T value;；T get(int index)。因為 get 回傳 String，可以寫 list.get(1).equals("banana") 不變，但若想用 String 專屬方法（如 length()）就不再需要轉型。若你把 Node 改成 static，它就看不到外層的 T，必須自己宣告 static class Node<T>。

7-2：選對容器

對每個需求選擇最合適的介面與實作，並說明理由：(a) 記錄一篇文章中每個單字出現的次數，最後依字母順序印出；(b) 判斷使用者輸入的 ID 是否曾出現過（百萬筆）；(c) 依時間順序保存聊天訊息並能隨機讀取第 i 則；(d) 待處理的工作佇列：先進先出。

7-2 解答

(a) `Map<String,Integer>`，用 `TreeMap`（需要依鍵排序）；或先用 `HashMap` 計數再排序一次。(b) `Set<String>`，用 `HashSet`（只要 contains，不需要順序，平均 $O(1)$）。(c) `List<String>`，用 `ArrayList`（尾端 add 攤銷 $O(1)$，隨機 get $O(1)$）。(d) `Queue<Job>`，用 `ArrayDeque`（兩端 $O(1)$）。

9　Lecture 07B：雜湊函數、雜湊表與攤銷複雜度

上一節說 `HashSet.contains` 與 `HashMap.get` 平均只要 $O(1)$——不管裡面有一百個還是一億個元素。這聽起來像魔法：在 `ArrayList` 裡找東西要 $O(n)$，在排序好的 `TreeSet` 裡要 $O(\log n)$，怎麼可能不看幾乎所有元素就知道某個鍵在不在？答案是雜湊。隨附程式在 `code/07_hashing`。

9.1　核心想法：把鍵變成陣列索引

陣列有一個其他結構都沒有的超能力：`arr[i]` 是 $O(1)$，不管陣列多大。如果我們能把任何鍵（字串、座標、學號……）**算成**一個 $0 \ldots m-1$ 的整數，就能直接把它放進大小為 m 的陣列。這個「算成整數」的函式就是**雜湊函數**，那個陣列就是**雜湊表**。

9.2　雜湊函數

雜湊函數（hash function）

一個把鍵映射到整數（雜湊值、hash code）的函式 h。要能用在雜湊表上，它必須：

1. **尊重等價**：`a.equals(b)` $\Rightarrow$ `h(a) == h(b)`。這是**正確性**要求，違反就會「放進去卻找不到」。
2. **確定性**：同一個鍵每次都得到同一個值（所以不能用隨機數、時間、記憶體位址）。
3. **快**：通常 $O(\text{鍵的大小})$，否則抵消了 $O(1)$ 的好處。
4. **分布均勻**：不同的鍵應「盡量」得到不同的值。這是**效能**要求：撞得越多，表就越慢。

注意反方向不成立：`h(a) == h(b)` 不代表 `a.equals(b)`——不同的鍵撞在一起叫**碰撞**（collision）。

字串最常用的雜湊是**多項式滾動雜湊**，也正是 `java.lang.String.hashCode()` 的演算法：

```
/** h = s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]；每步「乘 31 再加下一個字元」。 */
```

```java
static int goodHash(String s) {
    int h = 0;
    for (int i = 0; i < s.length(); i++) {
        h = 31 * h + s.charAt(i);          // int 會溢位——沒關係，溢位後仍是確定性的
    }
    return h;
}

static int mediocreHash(String s) {        // 只把字元加總：位置資訊丟了
    int h = 0;
    for (char c : s.toCharArray()) h = h + c;
    return h;
}

static int badHash(String s) { return 0; } //  「正確」但沒用：所有鍵撞在同一桶

/** 雜湊值 → 桶索引。floorMod 永遠回傳 0..capacity-1（hashCode 可能是負數！） */
static int bucketOf(int hash, int capacity) {
    return Math.floorMod(hash, capacity);
}
```

程式 28　節錄自 `StringHashing.java`

```
  goodHash("hello") = 99162322，String.hashCode() = 99162322
  goodHash(ab) = 3105，goodHash(ba) = 3135
  mediocreHash(ab) = 195，mediocreHash(ba) = 195（碰撞！）
  goodHash(長句) = -609428141（可能是負數）
=== 4. 分布品質：1000 個鍵放進 128 個桶 ===
  goodHash       最擠的桶 =  21 個鍵，空桶 =  28 / 128
  mediocreHash   最擠的桶 =  70 個鍵，空桶 =  73 / 128
  badHash        最擠的桶 = 1000 個鍵，空桶 = 127 / 128
   （理想狀況：1000/128 ≈ 8 個鍵一桶）
```

- 乘 31 讓位置影響結果（`"ab"` ≠ `"ba"`）；31 是奇質數，且 $31h = 32h - h$ 可以用位移快速計算。
- 三個函式都滿足「尊重等價」與「確定性」，所以都**正確**；差別全在**分布**。`badHash` 讓雜湊表退化成一條鏈結串列——每個操作 $\mathrm{O}(n)$。
- **加密雜湊**（SHA-256 等）為了「無法從雜湊值反推、無法刻意製造碰撞」而設計，慢上幾百倍。雜湊表不需要這些性質，所以**不用**加密雜湊。

⚠ `Math.abs(h) % m` 是錯的

`h` 為負時 `h % m` 也是負的，會讓陣列索引越界；很多人改寫成 `Math.abs(h) % m`，但 `Math.abs(Integer.MIN_VALUE)` 仍然是負數（2^{31} 放不進 `int`）。永遠用 `Math.floorMod(h, m)`。

9.3　雜湊表：桶與碰撞

雜湊表（hash table）

一個有 m 個**桶**（bucket）的陣列。鍵 k 放在桶 `floorMod(h(k), m)`。因為鍵的數量通常遠大於 m（甚至無限，例如所有字串），**碰撞是必然的**（鴿籠原理），所以雜湊表的設計核心就是**如何處理碰撞**。

方法一：鏈結（chaining）

每個桶是一條鏈結串列，串著所有雜湊到該桶的 (key, value)。這是 Java `HashMap` 的基本策略。

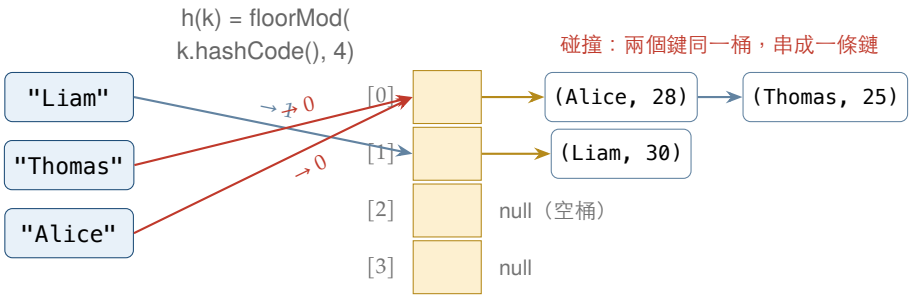


圖 14　容量 4 的鏈結雜湊表放入三個鍵（HashDemo 第 1 段的實際狀態）。

```java
public class ChainingHashMap<K, V> {
    private static class Node<K, V> {          // (key, value) + 同桶的下一個節點
        final K key;
        V value;
        Node<K, V> next;
        Node(K key, V value, Node<K, V> next) {
            this.key = key; this.value = value; this.next = next;
        }
    }
    private static final double MAX_LOAD_FACTOR = 0.75;
    private Node<K, V>[] buckets;
    private int size = 0;

    private int indexFor(K key, int length) {
        return Math.floorMod(key.hashCode(), length);
    }
```

```
17
18      private Node<K, V> findNode(K key) {
19          int idx = indexFor(key, this.buckets.length);      // 1. 用 hashCode 找桶
20          Node<K, V> n = this.buckets[idx];
21          while (n != null) {
22              if (n.key.equals(key)) return n;               // 2. 用 equals 確認
23              n = n.next;
24          }
25          return null;
26      }
27
28      public V put(K key, V value) {
29          Node<K, V> existing = findNode(key);
30          if (existing != null) {                            // 已存在：覆蓋，回傳舊值
31              V old = existing.value;
32              existing.value = value;
33              return old;
34          }
35          int idx = indexFor(key, this.buckets.length);
36          this.buckets[idx] = new Node<>(key, value, this.buckets[idx]);   // 插在鏈頭：O
    (1)
37          this.size = this.size + 1;
38          if ((double) this.size / this.buckets.length > MAX_LOAD_FACTOR) {
39              resize();                                      // 負載因子超標：倍增＋重新雜
    湊
40          }
41          return null;
42      }
43
44      public V get(K key) {
45          Node<K, V> n = findNode(key);
46          return n == null ? null : n.value;
47      }
48  }
```

程式 29　鏈結雜湊表的核心（節錄自 ChainingHashMap.java）

```
=== 1. 手刻的鏈結雜湊表 ===
  容量 4，size 3：
    [0] → (Alice, 28) → (Thomas, 25)
    [1] → (Liam, 30)
    [2]
    [3]
  put(Bob) 後容量 8（倍增＋重新雜湊）：
```

```
    [0] → (Alice, 28)
    [1] → (Liam, 30)
    [4] → (Thomas, 25)
    [5] → (Bob, 41)
=== 2. 大量資料：鏈保持很短 ===
  100000 個鍵，容量 262144，最長的鏈 = 3
```

注意擴容後 Thomas 從桶 0 搬到桶 4：桶索引依賴 `m`，`m` 變了每個鍵都要重新雜湊（rehash）。十萬個鍵放進 262144 個桶，最長的鏈只有 3——這就是平均 $O(1)$ 的來源：**每個桶平均只有 n/m 個元素，只要把 n/m 壓在常數以下，找一個鍵只需看常數個節點。**

方法二：開放定址與線性探測

不用鏈結串列，所有元素直接放在桶陣列裡；碰撞時往下一格找空位（到尾端繞回開頭）。



圖 15　線性探測：碰撞時往右找第一個空格。查詢時也用同樣路徑，遇到空格就能確定「不存在」。

```java
public boolean add(int x) {
    if (contains(x)) return false;
    if ((this.size + 1) * 2 > this.table.length) resize();   // 負載因子超過 0.5 就倍增
    int i = indexFor(x, this.table.length);
    while (this.table[i] != EMPTY) {                         // 線性探測：往下一格找
        i = (i + 1) % this.table.length;                     // 到尾端就繞回開頭
    }
    this.table[i] = x;
    this.size = this.size + 1;
    return true;
}
```

程式 30　節錄自 `LinearProbingIntSet.java`

	鏈結（chaining）	線性探測（linear probing）
碰撞處理	同桶串成鏈結串列	往下一格找空位
優點	簡單；負載因子可超過 1 仍能運作；刪除容易	記憶體連續、快取友善；不需要節點物件
缺點	每個節點多一個參照與物件開銷	容易群聚（連續占用區越長越慢）；負載因子須壓低（0.5）；刪除需要「墓碑」標記，否則會切斷探測路徑
誰在用	Java `HashMap`（鏈太長時改成紅黑樹）	Python `dict`、Rust `HashMap` 的變體

9.4 負載因子、擴容與攤銷複雜度

負載因子（load factor）

$\alpha = n/m$：元素數除以桶數，即「每桶平均幾個元素」。鏈結法的平均查找成本約為 $1 + \alpha$；只要 α 是常數，操作就是 $O(1)$。Java `HashMap` 在 $\alpha > 0.75$ 時把桶數加倍並重新雜湊。

擴容一次要 $O(n)$（每個元素重新雜湊），看起來會拖垮「$O(1)$」的宣稱。關鍵在於擴容很少發生。這和第 7 節陣列倍增的情況一模一樣，我們用那個更簡單的例子分析：

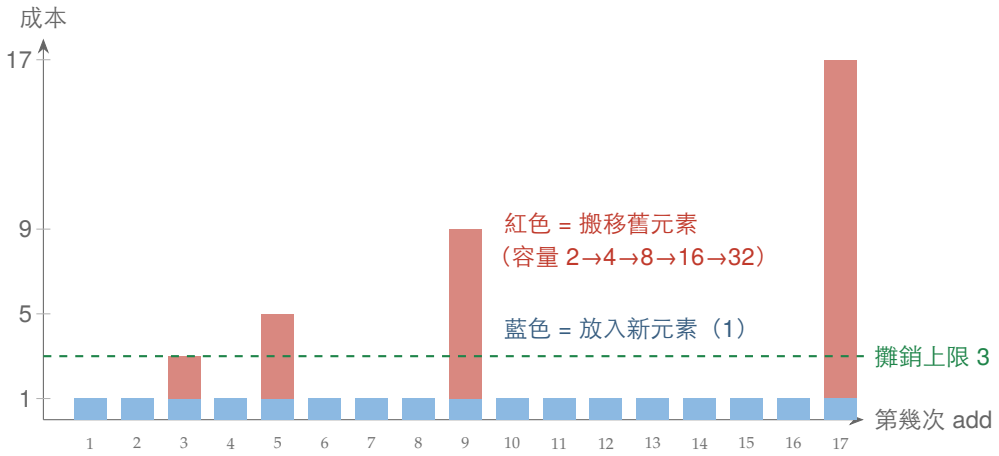


圖 16　陣列倍增下每次 add 的實際成本。尖峰很高但越來越稀疏；把紅色面積「攤」到所有 add 上，每次不超過 2。

攤銷複雜度（amortised complexity）

對一串 n 個操作，**總成本**除以 n。單一操作可能很貴，但若貴的操作**足夠稀少**，平均下來仍是常數。

陣列倍增的分析：從容量 1 開始連續 add n 個元素，搬移發生在容量 $1, 2, 4, \ldots$ 滿的時候，總搬移量 $= 1 + 2 + 4 + \cdots + 2^{\lfloor \log_2 n \rfloor} < 2n$。加上放入本身的 n，總成本 $< 3n$，所以**每次 add 攤銷** $O(1)$。

對照：若每次只加一格，搬移量是 $1 + 2 + \cdots + n = O(n^2)$，攤銷後每次 $O(n)$。

雜湊表的 rehash 完全同理：桶數倍增、負載因子固定，所以 `put` 的攤銷成本是 $O(1)$。這就

是為什麼我們說 HashMap 是「平均 $O(1)$」而不是「最差 $O(1)$」：單一次 put 可能觸發整表重建，但攤平後每次只多付常數。

三種「$O(1)$」的差別

- 最差情況 $O(1)$：每一次都保證常數（arr[i]）。
- 攤銷 $O(1)$：任何一連串操作的總量是線性的（ArrayList.add 尾端）。這是確定的保證，不依賴機率。
- 平均（期望） $O(1)$：假設雜湊函數分布均勻，期望值是常數（HashMap.get）。遇到刻意製造碰撞的敵手，仍可能退化成 $O(n)$。

9.5 equals 與 hashCode 的契約

雜湊表先用 **hashCode** 找桶，再用 **equals** 確認。若你的類別覆寫了 equals 卻沒有覆寫 hashCode，兩個「相等」的物件會落在不同的桶——放進去，卻找不到：

```
static class PointNoHash {                          // 錯誤：只覆寫 equals，忘了 hashCode
    final int x, y;
    @Override public boolean equals(Object o) { return o instanceof PointNoHash p && p.x
    == x && p.y == y; }
}

static class Point {                                // 正確：兩個一起覆寫，且用同一組欄位
    final int x, y;
    @Override public boolean equals(Object o) { return o instanceof Point p && p.x == x
    && p.y == y; }
    @Override public int hashCode() { return Objects.hash(x, y); }
}

record Coord(int x, int y) {}                       // 正確：record 自動產生一致的 equals 與
    hashCode
```

程式 31　節錄自 HashDemo.java

```
=== 5. equals/hashCode 契約 ===
  只覆寫 equals：contains(new PointNoHash(1,2)) = false  ← 找不到！
  equals+hashCode：contains(new Point(1,2)) = true
  record：contains(new Coord(1,2)) = true
```

hashCode 契約（java.lang.Object 的規定）

1. 同一個物件在其 equals 用到的欄位不變時，hashCode() 必須回傳同一個值。
2. a.equals(b) 為真 ⇒ a.hashCode() == b.hashCode()。
3. 不相等的物件可以有相同的 hashCode（但越少越好）。

實務規則：覆寫 **equals** 就一定要覆寫 **hashCode**，且用同一組欄位；最簡單的寫法是 Objects.hash(f1, f2, ...)；最安全的做法是用 **record** 當鍵。

可變的鍵是災難

鍵放進表後若被修改，它的 hashCode 變了，但它還躺在舊的桶裡：

```
List<String> key = new ArrayList<>(List.of("a"));
Map<List<String>, Integer> map = new HashMap<>();
map.put(key, 42);
key.add("b");                  // 改變了鍵 → hashCode 變了
map.get(key);                  // null！值還在表裡，卻永遠找不到；size 仍是 1
```

規則：雜湊表的鍵必須是不可變的——String、Integer、record 都安全；ArrayList、可變的自訂類別都不行。

9.6 標準函式庫：HashMap 與 HashSet

```
Map<String, Integer> wordCount = new HashMap<>();
for (String w : words) {
    wordCount.merge(w, 1, Integer::sum);          // 有就 +1，沒有就放 1（一行完成計數）
}
// {the=3, and=2, bird=1, cat=1, dog=1}（順序不保證）

Set<Integer> seen = new HashSet<>();
for (int d : new int[]{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5}) {
    if (!seen.add(d)) System.out.println("重複：" + d);   // add 回傳 false = 已存在
}
```

程式 32　節錄自 HashDemo.java

三種常用容器的複雜度總表（n 個元素）：

	查找	插入	刪除	有序走訪	備註
`ArrayList`（未排序）	$O(n)$	尾端攤銷 $O(1)$	$O(n)$	需先排序 $O(n \log n)$	保持插入順序、可重複
`TreeMap` / `TreeSet`	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$，天然有序	鍵需可比較（`Comparable`）
`HashMap` / `HashSet`	平均 $O(1)$	攤銷平均 $O(1)$	平均 $O(1)$	不支援（順序不定）	鍵需正確的 `equals` / `hashCode`、不可變

9.7 練習

8-1：手算雜湊

容量 $m = 8$ 的鏈結雜湊表，使用 `floorMod(h, 8)`，依序放入以下 (鍵, 雜湊值)：(A, 17)、(B, 25)、(C, 6)、(D, -3)、(E, 33)。畫出每個桶的鏈（新節點插在鏈頭）。接著放入 (F, 9)，負載因子變成多少？依 0.75 的門檻要不要擴容？

8-1 解答

索引：A→1、B→1、C→6、D→5（`floorMod(-3,8)=5`）、E→1。桶 1：E → B → A（E 最後插入所以在鏈頭）；桶 5：D；桶 6：C；其他空。放入 F（9→1）後 $n = 6$，$\alpha = 6/8 = 0.75$，不超過 0.75，不擴容（本教材的實作用「$>$」判斷）；再放一個才會。

8-2：修好這個鍵

下面的類別當作 `HashMap` 的鍵會出什麼問題？寫出修正版（兩種：手寫 `hashCode`、改成 `record`）。

```
class StudentId {
    String faculty; int number;
    @Override public boolean equals(Object o) {
        return o instanceof StudentId s && s.faculty.equals(faculty) && s.number ==
    number;
    }
}
```

8-2 解答

沒有覆寫 `hashCode`，兩個相等的 `StudentId` 幾乎必然落在不同桶，`get` 找不到。且欄位可變（沒有 `final`），放進表後改 `number` 也會讓它失蹤。修法一：加上 `@Override public int hashCode() { return Objects.hash(faculty, number); }` 並把欄位改 `final`。修法二：`record StudentId(String faculty, int number) {}`——一行搞定，且天生不可變。

8-3：攤銷推理

某個 `ArrayList` 的變體在滿時把容量**乘 1.5**（Java 實際上就是這麼做的）而不是乘 2。攤銷成本仍是 $O(1)$ 嗎？若改成「加 1000 格」呢？用等比 / 等差級數的總和說明。

8-3 解答

乘 1.5：搬移總量是公比 1.5 的等比級數 $\sum 1.5^k \approx 3n$，仍是 $O(n)$ 總量、攤銷 $O(1)$（任何大於 1 的倍率都成立，只是常數不同）。加 1000 格：搬移量 $1000+2000+\cdots \approx n^2/2000 = O(n^2)$，攤銷 $O(n)$——常數再大也救不了等差成長。

10　Week 7 Workshop 導讀：設計聊天客戶端的介面

Week 6 Workshop 決定了網路上跑什麼（協定）；Week 7 Workshop 決定程式內部怎麼切：把「跟伺服器講話」的程式碼（客戶端程式庫）與「跟使用者互動」的程式碼（UI）分成兩塊，中間用一個介面隔開。目標是——任何人寫的 UI 都能接上任何人寫的客戶端程式庫。這是 Lecture 06A「介面即契約」與 Lecture 07A「對介面測試」的綜合演練。

10.1　為什麼要切這一刀？

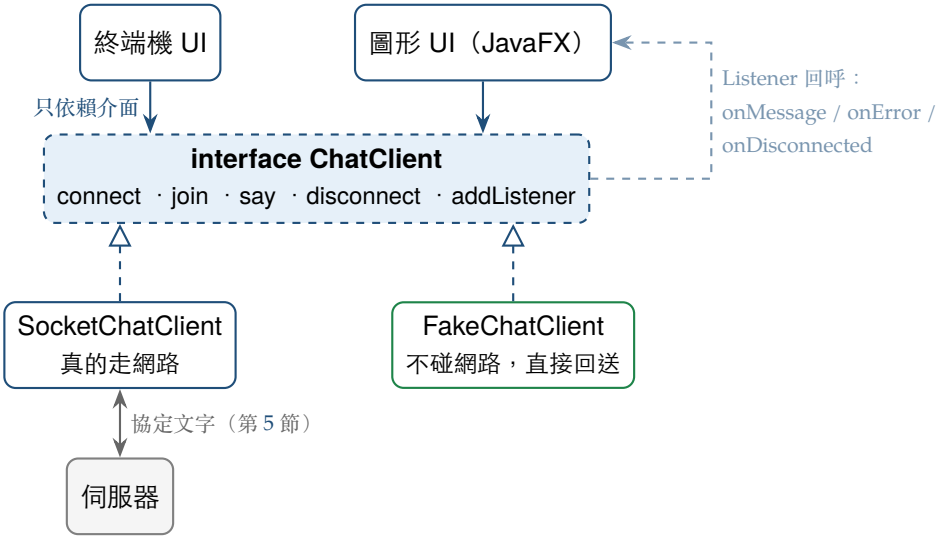


圖 17　UI 只認識 `ChatClient` 介面，所以可以接真的網路實作，也可以接假的實作來測試。

- **可測試**：UI 邏輯（訊息怎麼排版、錯誤怎麼顯示）可以用 `FakeChatClient` 在**沒有伺服器**時測試——甚至在伺服器組還沒寫完之前。
- **可替換**：終端機版與圖形版 UI 共用同一個程式庫；程式庫從單執行緒升級成多執行緒，UI 一行都不用改。
- **可分工**：介面就是合約。UI 組與程式庫組各自對著介面開發，最後一定接得起來。

10.2 介面長什麼樣？

```java
public interface ChatClient {

    /** 事件監聽者：UI 實作這個介面，程式庫在事件發生時呼叫它。 */
    interface Listener {
        void onMessage(String room, String from, String text);
        void onError(int code, String text);
        void onDisconnected();
    }

    void addListener(Listener listener);              // 可註冊多個

    /** 連線並登入。@throws IOException 網路失敗；@throws IllegalStateException 若已連線 */
    void connect(String host, int port, String name) throws java.io.IOException;

    void join(String room);

    /** 前置條件：已連線且已 join 該房間。 */
    void say(String room, String text);

    /** 告知伺服器離開並關閉連線；之後任何操作都是 IllegalStateException。 */
    void disconnect();

    boolean isConnected();
}
```

程式 33　08_chat_protocol/ChatClient.java

四個設計決定，每一個都對應本週學過的原則：

1. **方法對應「使用者能做的事」，而不是協定。**介面裡沒有 `sendLine(String)`、沒有 `LOGIN` / `SAY` 這些字。協定是實作細節——**封裝**的原則說：UI 不該知道、也不該依賴它。日後協定改版，UI 不受影響。

2. **伺服器推來的事件用回呼（callback）。**別人的訊息何時到達，UI 無法預知，所以不能用「呼叫方法拿回傳值」的方式取得。改成 UI 註冊一個 `Listener`，程式庫在事件發生時反過來呼叫 UI。這是第 6 節 `Sorter`「把行為當參數傳」的同一招。

3. **前置條件與例外寫進契約。**未連線就 `say` → `IllegalStateException`（呼叫者的錯，第 2 節）；網路斷掉 → `IOException`（環境的錯，呼叫者必須處理，所以是受檢例外）。UI 因此知道該接哪些例外。

4. **行為子型別。**任何實作——真的或假的——都必須遵守同一份契約（例如「`say` 之後，所

有 listener 會收到一次 onMessage」）。否則用假實作測過的 UI，接上真實作就壞了；這正是第 4 節 LSP 要防的事。

10.3　用假實作測 UI

FakeChatClient 完全不碰網路：say 直接把訊息回送給所有 listener（模擬伺服器的 MSG 廣播），不在房間裡就回送 onError(403, ...)（模擬 ERR）。UI 端用匿名類別實作 Listener：

```
StringBuilder screen = new StringBuilder();                    // 模擬「螢幕」
ChatClient client = new FakeChatClient();                      // 宣告型別是介面
client.addListener(new ChatClient.Listener() {                 // 匿名類別：現場實作介面
    public void onMessage(String room, String from, String text) {
        screen.append("[").append(room).append("] ").append(from).append(": ").append(
    text).append("\n");
    }
    public void onError(int code, String text) { screen.append("ERROR ").append(code).
    append(": ").append(text).append("\n"); }
    public void onDisconnected()               { screen.append("(disconnected)\n"); }
});
client.connect("localhost", 5555, "angelia");
client.say("general", "hi");                                   // 還沒 join → onError
client.join("general");
client.say("general", "hi again");                             // → onMessage
client.disconnect();                                           // → onDisconnected
assert screen.toString().equals("ERROR 403: not in room general\n[general] angelia: hi
    again\n(disconnected)\n");
```

程式 34　節錄自 ProtocolTest.java

```
=== 用 FakeChatClient 測 UI 邏輯（不需要網路）===
ERROR 403: not in room general
[general] angelia: hi again
(disconnected)
  斷線後操作 → IllegalStateException：尚未連線
全部通過。
```

匿名類別與 lambda

new ChatClient.Listener() { ... } 是匿名類別：一個沒有名字、只用一次的實作。若介面只有一個抽象方法（如 Sorter、Runnable、Comparator），可以進一步簡寫成 lambda：Sorter s = arr -> Arrays.sort(arr);。Listener 有三個方法所以不能用 lambda；另一種設計是拆成三個單方法介面（onMessage(MessageHandler) 等），各自都能用 lambda——這是 Workshop 中值得討論的取捨。

10.4　真實作的骨架與討論清單

真的 SocketChatClient 會用第 5 節的 Socket / BufferedReader / PrintWriter，並且需要一條背景執行緒不停 readLine()，把每行 parse 成 ChatMessage 後分派給 listener：

```
private void readLoop() {                       // 在背景執行緒執行
    String line;
    while ((line = in.readLine()) != null) {
        ChatMessage m = ChatMessage.parse(line);
        switch (m) {
            case ChatMessage.Msg g -> listeners.forEach(l -> l.onMessage(g.room(), g.name
    (), g.text()));
            case ChatMessage.Err e -> listeners.forEach(l -> l.onError(e.code(), e.text()
    ));
            case ChatMessage.Bye b -> { listeners.forEach(ChatClient.Listener::
    onDisconnected); return; }
            default -> { /* OK：忽略，或用來解除等待中的 connect/join */ }
        }
    }
    listeners.forEach(ChatClient.Listener::onDisconnected);   // 連線被對方關閉
}
```

程式 35　SocketChatClient 的讀取迴圈（示意）

Workshop 中應該寫進介面文件的問題：

1. connect 是阻塞到伺服器回 OK 才返回，還是立刻返回、之後用 onConnected 回呼通知？（前者簡單，後者不會卡住 UI。）
2. 回呼在哪一條執行緒被呼叫？圖形 UI 通常只能在 UI 執行緒更新畫面——這是 Week 8 的主題，但介面文件現在就要說清楚。
3. 重複 connect、未 join 就 say、disconnect 之後再操作：各自是例外還是靜默忽略？（本介面：全部 IllegalStateException。）
4. 伺服器主動斷線（沒有 BYE）時，程式庫如何得知並通知 UI？（readLine() 回傳 null。）
5. say 失敗（ERR）時，UI 如何知道是哪一則發言失敗？需要在協定或介面加上請求編號嗎？

9-1：第二個 UI

寫一個終端機 UI：用 Scanner 讀使用者輸入，/join room 呼叫 join，其他文字呼叫 say，/quit 呼叫 disconnect；用 Listener 把收到的訊息印到螢幕。先用 FakeChatClient 跑起來。思考：你的 UI 有沒有任何一行程式碼提到 FakeChatClient 以外的實作類別？除了 new 那一行之外，不應該有。

9-2：對介面寫黑箱測試

用 JUnit 5 為 `ChatClient` 寫一個抽象測試類別（仿照第 7 節的 `GuestListTest`），至少測：未連線就 `say` 拋出 `IllegalStateException`；`join` 後 `say` 會讓 listener 收到恰好一次 `onMessage`；`disconnect` 後 `isConnected()` 為 false 且 listener 收到 `onDisconnected`。先用 `FakeChatClient` 子類別跑過；日後 `SocketChatClient` 寫好時，同一套測試應該也要能過。

11　總複習：觀念地圖、常見錯誤與自我測驗

11.1 一頁看完兩週

觀念	一句話	關鍵字／範例
封裝	欄位 `private`，只透過公開方法改狀態，藉此守住資料不變量並分離介面與實作	`BankAccount.balance`；`IllegalArgumentException` vs `IllegalStateException`
介面	只有簽名與規格、沒有實作的契約；實作類別必須提供每個方法	`implements`、`@Override`、不能 `new`
ADT	一組操作加上規格，不談儲存方式；介面表達 ADT，類別是資料結構	`ChatHistory`：環狀緩衝區 vs 鏈結串列
子型別	需要 U 的地方都能給 T；型別即集合，子型別即子集合	`implements` / `extends`；`Object` 是根
繼承	重用父類別的欄位與方法；覆寫改行為；`super` 呼叫父版本	`extends`、抽象類別、`protected`、建構子不繼承
動態分派	能不能呼叫看靜態型別；呼叫哪個版本看執行期型別	`PointsAccount p = new PremiumPointsAccount()`
LSP	子型別必須能替換父型別而不弄壞依賴契約的程式：要求不能變多、承諾不能變少	`OverdraftSavingsAccount` 違規；`AccountWithWithdrawalFees` 的抉擇
JUnit	`@Test` 方法 + 斷言；每個測試一個新物件；`@BeforeEach` 準備	`assertEquals(expected, actual)`、`assertThrows`
黑箱測試	只透過介面、只依契約判斷；一套測試可測多種實作	抽象測試類別 + 工廠方法 `create()`
TDD	紅（先寫會失敗的測試）→ 綠（最少程式）→ 重構	每圈一小步；測試先失敗才有意義
轉型	向上自動安全；向下要明寫且執行期檢查	`ClassCastException`、`instanceof T t`
泛型	型別當參數，編譯期型別安全、免轉型；執行期被擦除	`GuestList<T>`、`<T extends Comparable<T>>`、不能 `new T[]`
Collections	介面（List / Set / Map）+ 實作（ArrayList / HashSet / HashMap…）	宣告用介面；`of()` 不可變；`getOrDefault`
雜湊函數	尊重等價、確定、快、分布均勻	`31*h + c`；`Math.floorMod`
雜湊表	鍵 → 桶索引；碰撞用鏈結或探測；負載因子超標就倍增重雜湊	平均 $O(1)$；`equals` / `hashCode` 契約；鍵不可變
攤銷複雜度	總成本除以操作數；倍增讓貴的操作足夠稀少	陣列倍增總成本 $< 3n$

11.2 最常見的十二個錯誤

1. ✗ 為每個欄位反射性地寫 getter / setter。✓ 公開方法對應操作（`deposit`），不對應欄位。
2. ✗ 回傳內部陣列或 `List` 的參照。✓ 防禦性複製（`Arrays.copyOf`、`new ArrayList<>(x)`）。
3. ✗ `catch (Exception e) {}` 吞掉所有錯誤。✓ 只接自己知道怎麼處理的例外。
4. ✗ 忘記 `@Override`，拼錯方法名變成新方法而沒有覆寫。✓ 永遠加 `@Override`。

5. ❌ 用 == 比較 `String`、`Integer` 或任何物件。✅ 用 `equals`（或 `Objects.equals` 防 null）。
6. ❌ 為了少打字而繼承一個契約更強的類別（手續費帳戶繼承 `SavingsAccount`）。✅ 繼承表達行為上的 is-a；否則用組合。
7. ❌ 在 static 方法裡直接用實例欄位。✅ 需要物件狀態就寫實例方法，或把物件當參數傳入。
8. ❌ 測試裡宣告 `GuestArrayList gl` 並讀 `gl.data`。✅ 宣告成介面型別，只用公開方法。
9. ❌ `assertEquals(actual, expected)` 顛倒。✅ 預期值在前，失敗訊息才正確。
10. ❌ 泛型類別回傳 `T[]`。✅ 回傳 `List<T>`；內部用 `Object[]`。
11. ❌ 覆寫 `equals` 不覆寫 `hashCode`；用可變物件當鍵。✅ 兩個一起覆寫（`Objects.hash`）或直接用 record。
12. ❌ `Math.abs(h) % m` 算桶索引。✅ `Math.floorMod(h, m)`。

11.3 自我測驗（20 題）

先不看解答，每題用一兩句話回答。

Q1. 封裝帶來的兩個好處是什麼？各舉 `BankAccount` 的一個例子。

Q2. `deposit(-5)` 應該拋出 `IllegalArgumentException` 還是 `IllegalStateException`？餘額 10 時 `withdraw(50)` 呢？

Q3.「`private` 欄位只有同一個物件能存取」——對還是錯？

Q4. 介面與抽象類別各在什麼情況下是較好的選擇？

Q5. `ChatHistory h = new LinkedListChatHistory(); h.erase();` 為什麼無法編譯？

Q6. 用一句話定義子型別。`PremiumPointsAccount` 是 `Account` 的子型別嗎？為什麼？

Q7. `Account x = new PremiumPointsAccount();` 之後呼叫 `x.deposit(100);`，執行的是哪個類別的 `deposit`？這叫什麼？

Q8. 建構子會被繼承嗎？子類別建構子的第一行隱含在做什麼？

Q9. 列出行為子型別的四條規則。`OverdraftSavingsAccount` 違反哪幾條？

Q10. `Square extends Rectangle` 什麼時候違反 LSP、什麼時候不違反？

Q11. JUnit 為什麼對每個 `@Test` 方法建立一個新的測試物件？

Q12. 黑箱測試與白箱測試的差別？`GuestListTest` 裡哪一個測試其實是白箱？

Q13. TDD 中「測試必須先失敗」的目的是什麼？

Q14. `Object o = 42; String s = (String) o;` 會在編譯期還是執行期出錯？出什麼錯？

Q15. 為什麼泛型類別裡不能寫 `new T[10]`？標準的替代做法是什麼？

Q16. `Integer a = 1000, b = 1000; a == b` 的結果是什麼？為什麼？

Q17. 要「依字母順序印出每個單字的出現次數」，該用哪種 `Map` 實作？若只需要次數不需順序呢？

Q18. 一個好的雜湊函數需要哪四個性質？哪一個是正確性要求、哪些是效能要求？

Q19. 說明鏈結雜湊表 `get(key)` 的步驟，以及為什麼 `equals` 與 `hashCode` 必須一致。

Q20. 為什麼陣列倍增的 `add` 是攤銷 $O(1)$，而每次加一格是攤銷 $O(n)$？

自我測驗解答

A1. 完整性（`balance` 永遠 ≥ 0，因為只能經 `deposit` / `withdraw` 改）與抽象（使用者只需知道三個方法；內部改成 `long` 不影響呼叫者）。

A2. `IllegalArgumentException`（參數本身不合法）；`IllegalStateException`（參數合法但目前狀態不允許）。

A3. 錯。`private` 是類別層級的界線：同類別的程式碼可以存取另一個同類別物件的私有欄位。

A4. 介面：表達能力 / 契約、要多重實作、想最大解耦（預設選擇）。抽象類別：多個實作共享大量欄位與具體方法。

A5. `h` 的靜態型別是 `ChatHistory`，契約裡沒有 `erase()`；編譯器只看靜態型別決定方法是否存在。

A6. 任何需要 U 的地方都可以安全提供 T，則 T 是 U 的子型別。是：`Premium extends PointsAccount implements Account`，子型別可傳遞。

A7. `PremiumPointsAccount` 的版本（執行期型別）；動態分派。

A8. 不會。隱含呼叫 `super()`，先建構父類別那部分；父類別沒有無參數建構子時必須明寫 `super(...)`。

A9. 前置條件不能更強、後置條件不能更弱、不變量保持、歷史約束。違反 2（永遠回 true）與 3（餘額可負）。

A10. 有 `setWidth` / `setHeight` 的可變矩形會違反（設寬 5 高 3 期望面積 15）；不可變的 record 不違反。

A11. 讓測試彼此獨立、順序無關，一個測試留下的狀態不會影響另一個。

A12. 黑箱只透過公開介面依契約判斷，可套用所有實作；白箱依賴實作細節。`nullNameIsRejected`（介面沒規定 null 行為）。

A13. 證明測試真的在檢查目標行為（否則可能是永遠通過的無效測試），並確認新功能確實尚未存在。

A14. 執行期；`ClassCastException`。編譯器只知道 `o` 是 `Object`，轉型合法與否要到執行期才知道。

A15. 型別擦除：執行期不知道 `T` 是什麼。用 `Object[]` 儲存，取出時 `(T)` 轉型並加 `@SuppressWarnings("unchecked")`，或直接用 `ArrayList<T>`。

A16. `false`（大多數 JVM）：1000 超出 `Integer` 快取範圍（-128..127），兩個是不同物件；比較物件要用 `equals`。

A17. `TreeMap`（依鍵排序走訪）；`HashMap`（平均 $O(1)$，不需順序）。

A18. 尊重等價（正確性）、確定性（正確性）、快、分布均勻（後兩者效能）。

A19. 算 `floorMod(key.hashCode(), m)` 找桶 → 沿鏈用 `equals` 比對。若相等的鍵雜湊值不同，會找錯桶而找不到。

A20. 倍增：搬移總量 $1+2+4+\cdots<2n$，總成本 $<3n$，平均常數。加一格：搬移 $1+2+\cdots+n=O(n^2)$，平均 $O(n)$。

11.4 中英術語對照

封裝	encapsulation	抽象資料型別	abstract data type (ADT)
資料不變量	data invariant	存取修飾子	access modifier
介面	interface	實作	implementation
子型別	subtype	父類別 / 子類別	superclass / subclass
繼承	inheritance	覆寫	override
動態分派	dynamic dispatch	靜態 / 執行期型別	static / runtime type
抽象類別	abstract class	行為子型別	behavioural subtype
Liskov 替換原則	Liskov Substitution Principle	前置 / 後置條件	pre- / postcondition
單元測試	unit test	黑箱 / 白箱	black-box / white-box
測試驅動開發	test-driven development	斷言	assertion
防禦性複製	defensive copy	轉型	cast
泛型	generics	型別參數 / 引數	type parameter / argument
型別擦除	type erasure	自動裝箱	autoboxing
雜湊函數	hash function	雜湊表	hash table
桶	bucket	碰撞	collision
鏈結	chaining	線性探測	linear probing
負載因子	load factor	重新雜湊	rehash
攤銷複雜度	amortised complexity	通訊協定	protocol

A　附錄：隨附程式完整原始碼

以下是 `code/` 資料夾內所有 Java 檔案的完整內容，與本文節錄的版本一致。每個資料夾都可以獨立編譯（`javac *.java`）並以 `java -ea <Main 類別>` 執行；`05_junit_tdd` 需要 JUnit 5（見

第 7.8 節）。

A.1 01_encapsulation：封裝

01_encapsulation/Account.java

```java
/**
 * 帳戶介面（interface）：只描述「能做什麼」（契約），完全不描述「怎麼做」。
 *
 * 任何 implements Account 的類別都必須提供這三個方法，
 * 而使用 Account 的程式碼只需要認識這份契約即可。
 */
public interface Account {

    /**
     * 存入 amount。
     * @param amount 前置條件：amount >= 0
     */
    void deposit(int amount);

    /**
     * 提出 amount。
     * @param amount 前置條件：0 <= amount <= getBalance()
     */
    void withdraw(int amount);

    /**
     * @return 目前餘額，後置條件：回傳值 >= 0
     */
    int getBalance();
}
```

01_encapsulation/BankAccount.java

```java
/**
 * 銀行帳戶：示範「封裝」（encapsulation）。
 *
 * 資料不變量（data invariant）：balance >= 0
 * 因為 balance 是 private，外界不可能寫 account.balance = -500;
 * 唯一能改變餘額的路徑是 deposit / withdraw，而這兩個方法都會守住不變量。
 */
public class BankAccount implements Account {

    // 私有欄位：只有這個類別裡的程式碼看得到
    private int balance;

```

```
    /**
     * 建構子（constructor）：確保物件「出生」時就處於合法狀態。
     * @param initialDeposit 開戶金額，前置條件：>= 0
     */
    public BankAccount(int initialDeposit) {
        if (initialDeposit < 0) {
            throw new IllegalArgumentException(
                    "開戶金額不可為負數：" + initialDeposit);
        }
        this.balance = initialDeposit;
    }

    @Override
    public void deposit(int amount) {
        if (amount < 0) {                       // 違反前置條件 → 拒絕
            throw new IllegalArgumentException("存款金額不可為負數：" + amount);
        }
        this.balance = this.balance + amount;   // 不變量仍成立
    }

    @Override
    public void withdraw(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("提款金額不可為負數：" + amount);
        }
        if (amount > this.balance) {            // 物件目前的「狀態」不允許此操作
            throw new IllegalStateException(
                    "餘額不足：餘額 " + this.balance + "，欲提領 " + amount);
        }
        this.balance = this.balance - amount;   // 因為 amount <= balance，結果仍 >= 0
    }

    /** Getter：讓外界「讀」得到餘額，但「寫」不到。 */
    @Override
    public int getBalance() {
        return this.balance;
    }
}
```

01_encapsulation/PointsAccount.java

```
/**
 * 飛行常客點數帳戶：另一種完全不同的 Account 實作。
 *
 * 行為與 BankAccount 不同（開戶送 10000 點、每次存入加送 1%），
 * 但同樣遵守 Account 介面規定的前置／後置條件，
```

```
 * 因此任何「只認識 Account」的程式碼都能直接使用它。
 */
public class PointsAccount implements Account {

    private int points = 10000;   // 新會員禮：開戶就有 10000 點

    @Override
    public void deposit(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("點數不可為負數：" + amount);
        }
        // 每次存入加送 1%（整數除法，不足 100 點就沒有紅利）
        this.points = this.points + amount + amount / 100;
    }

    @Override
    public void withdraw(int amount) {
        if (amount < 0 || amount > this.points) {
            throw new IllegalStateException("點數不足或金額不合法：" + amount);
        }
        this.points = this.points - amount;
    }

    @Override
    public int getBalance() {
        return this.points;
    }
}
```

01_encapsulation/PremiumPointsAccount.java

```
/**
 * 高級點數帳戶：繼承（extends）PointsAccount，只改寫 deposit。
 *
 * 用來示範「動態分派」（dynamic dispatch）：
 *   PointsAccount a = new PremiumPointsAccount();
 *   a.deposit(500);   // 執行的是 PremiumPointsAccount 的 deposit，而不是 PointsAccount 的
 */
public class PremiumPointsAccount extends PointsAccount {

    /** 高級會員：存入金額先乘 1.1，再交給父類別處理（父類別再送 1%）。 */
    @Override
    public void deposit(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("點數不可為負數：" + amount);
        }
```

```
        super.deposit(amount * 110 / 100);   // super.deposit：呼叫父類別的版本
    }
}
```

01_encapsulation/Main.java

```
/**
 * 01_encapsulation 示範程式。
 *
 * 編譯與執行：
 *   javac *.java
 *   java -ea Main
 */
public class Main {

    /**
     * 這個方法只認識 Account 介面：
     * 不管傳進來的是 BankAccount、PointsAccount 還是 PremiumPointsAccount，都能正常運作。
     */
    static void handleAccount(Account a) {
        System.out.println("  餘額：" + a.getBalance());
        a.deposit(500);
        System.out.println("  存入 500 後：" + a.getBalance());
    }

    public static void main(String[] args) {
        System.out.println("=== 1. 封裝：只能透過方法改變狀態 ===");
        BankAccount acct = new BankAccount(100);
        acct.deposit(50);
        acct.withdraw(30);
        System.out.println("  餘額 = " + acct.getBalance());        // 120
        assert acct.getBalance() == 120;
        // acct.balance = -500;   // 編譯錯誤！balance 是 private，這正是封裝的目的

        System.out.println("=== 2. 例外：違反前置條件時立刻失敗 ===");
        try {
            acct.withdraw(1000);                                   // 餘額不足
        } catch (IllegalStateException e) {
            System.out.println("  攔到 IllegalStateException：" + e.getMessage());
        }
        try {
            acct.deposit(-5);                                      // 負數存款
        } catch (IllegalArgumentException e) {
            System.out.println("  攔到 IllegalArgumentException：" + e.getMessage());
        }
        assert acct.getBalance() == 120 : "失敗的操作不應改變狀態";
```

```
41
42          System.out.println("=== 3. 介面：同一段程式碼處理不同實作 ===");
43          System.out.println(" BankAccount：");
44          handleAccount(new BankAccount(0));                      // 0 → 500
45          System.out.println(" PointsAccount：");
46          handleAccount(new PointsAccount());                     // 10000 → 10505
47
48          System.out.println("=== 4. 動態分派：看的是「執行期」的型別 ===");
49          PointsAccount p = new PremiumPointsAccount();           // 靜態型別 PointsAccount
50          p.deposit(500);                                         // 執行 Premium 版：550 + 5 = 555
51          System.out.println("  PremiumPointsAccount 存 500 → " + p.getBalance());
52          assert p.getBalance() == 10555;
53
54          Account[] accounts = {
55              new BankAccount(10), new PointsAccount(), new PremiumPointsAccount()
56          };
57          for (Account a : accounts) {
58              a.deposit(100);          // 每個元素各自跑自己類別的 deposit
59              System.out.println("  " + a.getClass().getSimpleName()
60                      + " → " + a.getBalance());
61          }
62          System.out.println("全部完成。");
63      }
64  }
```

A.2 02_adt_interfaces：介面與抽象資料型別

02_adt_interfaces/ChatHistory.java

```
1   /**
2    * 聊天紀錄的抽象資料型別（ADT）。
3    *
4    * 這個介面只說「能做什麼」：新增訊息、顯示訊息、查詢數量。
5    * 至於底層是用陣列、鏈結串列還是檔案來存——介面完全不管。
6    */
7   public interface ChatHistory {
8
9       /** 新增一則訊息到紀錄中。 */
10      void addMessage(String msg);
11
12      /** 依「最新在前」的順序，把訊息一行一行印出來。 */
13      void display();
14
15      /** @return 目前保存的訊息數量 */
16      int size();
17  }
```

02_adt_interfaces/ErasableChatHistory.java

```java
/**
 * 擴充介面（interface extends interface）：
 * 在 ChatHistory 的所有要求之上，再多要求一個「清空」的能力。
 *
 * 任何 ErasableChatHistory 同時也是一個 ChatHistory（子型別關係）。
 */
public interface ErasableChatHistory extends ChatHistory {

    /** 刪除所有訊息。後置條件：size() == 0 */
    void erase();
}
```

02_adt_interfaces/CircularBufferChatHistory.java

```java
/**
 * 實作一：固定容量的「環狀緩衝區」（circular buffer）。
 *
 * 只保留最近 capacity 則訊息；滿了之後，新訊息會覆蓋最舊的那一則。
 * 優點：空間固定 O(capacity)。缺點：舊訊息會遺失。
 *
 * 資料不變量：
 *   0 <= size <= arr.length
 *   0 <= nextAvailable < arr.length
 *   最新的訊息位於 (nextAvailable - 1 + arr.length) % arr.length
 */
public class CircularBufferChatHistory implements ChatHistory {

    private final String[] arr;      // 固定大小的儲存空間
    private int nextAvailable;       // 下一則訊息要寫入的索引
    private int size;                // 目前保存的訊息數量

    public CircularBufferChatHistory(int capacity) {
        if (capacity <= 0) {
            throw new IllegalArgumentException("容量必須為正數：" + capacity);
        }
        this.arr = new String[capacity];
        this.nextAvailable = 0;
        this.size = 0;
    }

    @Override
    public void addMessage(String msg) {
        this.arr[this.nextAvailable] = msg;
        // 取餘數讓索引「繞回」開頭：這就是「環狀」的由來
```

```
        this.nextAvailable = (this.nextAvailable + 1) % this.arr.length;
        if (this.size < this.arr.length) {
            this.size = this.size + 1;      // 還沒滿：數量 +1；滿了：數量維持不變（覆蓋）
        }
    }

    @Override
    public void display() {
        int i = 0;
        // 從最新的一則往回走
        int idx = (this.nextAvailable - 1 + this.arr.length) % this.arr.length;
        while (i < this.size) {
            System.out.println(this.arr[idx]);
            idx = (idx - 1 + this.arr.length) % this.arr.length;   // 往回退一格（可能繞回尾端）
            i = i + 1;
        }
    }

    @Override
    public int size() {
        return this.size;
    }
}
```

02_adt_interfaces/LinkedListChatHistory.java

```
/**
 * 實作二：用「鏈結串列」（linked list）保存訊息，數量不受限。
 *
 * 新訊息永遠接在串列最前面（prepend），所以 addMessage 是 O(1)，
 * 而且串列本身就是「最新在前」的順序，display 直接走訪即可。
 *
 * 這個類別實作的是 ErasableChatHistory，所以同時也是一個 ChatHistory。
 */
public class LinkedListChatHistory implements ErasableChatHistory {

    /**
     * 靜態巢狀類別（static nested class）：一個節點 = 一則訊息 + 指向下一節點的參照。
     * 標成 private：這是實作細節，外界不該知道我們用鏈結串列。
     */
    private static class Node {
        final String msg;
        final Node rest;

        Node(String msg, Node rest) {
            this.msg = msg;
```

```
            this.rest = rest;
        }
    }

    private Node head = null;   // null 代表空串列
    private int size = 0;

    @Override
    public void addMessage(String msg) {
        this.head = new Node(msg, this.head);   // 新節點指向舊的 head，再成為新的 head
        this.size = this.size + 1;
    }

    @Override
    public void display() {
        Node current = this.head;
        while (current != null) {
            System.out.println(current.msg);
            current = current.rest;
        }
    }

    @Override
    public int size() {
        return this.size;
    }

    @Override
    public void erase() {
        this.head = null;   // 讓 head 指向 null，所有節點就會被垃圾回收
        this.size = 0;
    }
}
```

02_adt_interfaces/SimpleList.java

```
/**
 * 最經典的 ADT：List（串列）。
 *
 * 串列是「有順序」的集合（有第一個、第二個……），允許重複，且大小可以改變
 * ——「大小可變」正是串列與陣列最大的差別。
 *
 * 注意：這裡的元素型別是 Object（任何物件都能放）；
 * 第 7 章會用「泛型」把它升級成 SimpleList<T>。
 */
public interface SimpleList {
```

```
    /** @return 串列中元素的個數 */
    int size();

    /**
     * @param index 前置條件：0 <= index < size()
     * @return 第 index 個元素（從 0 起算）
     */
    Object get(int index);

    /** 把 o 加到串列尾端。後置條件：size() 增加 1 */
    void add(Object o);

    /** 移除第一個與 o 相等（equals）的元素；若不存在則什麼都不做。 */
    void remove(Object o);
}
```

02_adt_interfaces/LinkedSimpleList.java

```
import java.util.Objects;

/**
 * SimpleList 的鏈結串列實作。
 *
 * 使用「內部類別」（inner class）Node 把節點藏在類別裡：
 * 使用者只看到 SimpleList 的四個操作，完全不需要知道 Node 的存在，
 * 也不必自己區分「空串列」與「非空串列」——這些細節全部被封裝起來了。
 *
 * 資料不變量：size 等於從 head 出發能走到的節點數。
 */
public class LinkedSimpleList implements SimpleList {

    /** 私有內部類別：外界看不到，也不需要看到。 */
    private class Node {
        Object value;
        Node next;

        Node(Object value, Node next) {
            this.value = value;
            this.next = next;
        }
    }

    private Node head = null;
    private int size = 0;

```

```
    @Override
    public int size() {
        return this.size;
    }

    @Override
    public Object get(int index) {
        if (index < 0 || index >= this.size) {          // 檢查前置條件
            throw new IndexOutOfBoundsException(
                    "index " + index + " 超出範圍 [0, " + this.size + ")");
        }
        Node current = this.head;
        int i = 0;
        while (i < index) {                             // 往前走 index 步
            current = current.next;
            i = i + 1;
        }
        return current.value;
    }

    @Override
    public void add(Object o) {
        Node node = new Node(o, null);
        if (this.head == null) {                        // 空串列：新節點就是 head
            this.head = node;
        } else {                                        // 否則走到最後一個節點，接上去
            Node current = this.head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = node;
        }
        this.size = this.size + 1;
    }

    @Override
    public void remove(Object o) {
        if (this.head == null) {
            return;                                     // 空串列：沒東西可移除
        }
        if (Objects.equals(this.head.value, o)) {       // Objects.equals 能安全處理 null
            this.head = this.head.next;                 // 移除的是第一個節點
            this.size = this.size - 1;
            return;
        }
        Node prev = this.head;
        while (prev.next != null) {
            if (Objects.equals(prev.next.value, o)) {
```

```
                prev.next = prev.next.next;              // 跳過被移除的節點
                this.size = this.size - 1;
                return;
            }
            prev = prev.next;
        }
    }

    /** 方便除錯用：印成 [a, b, c] 的形式。 */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("[");
        Node current = this.head;
        while (current != null) {
            sb.append(current.value);
            if (current.next != null) {
                sb.append(", ");
            }
            current = current.next;
        }
        return sb.append("]").toString();
    }
}
```

02_adt_interfaces/Main.java

```
/**
 * 02_adt_interfaces 示範程式。
 *
 * 編譯與執行：
 *   javac *.java
 *   java -ea Main
 */
public class Main {

    /**
     * 這個函式只認識 ChatHistory 介面。
     * 傳進來的是環狀緩衝區或鏈結串列，它完全不在乎——這就是「解耦」（decoupling）。
     */
    static void simulateChat(ChatHistory history) {
        history.addMessage("Alice: 嗨！");
        history.addMessage("Bob: 你好～");
        history.addMessage("Alice: 週六要一起讀 Java 嗎？");
        history.addMessage("Bob: 好啊，下午兩點圖書館見。");
        System.out.println("   （共 " + history.size() + " 則，最新在前）");
        history.display();
```

```
    }

    public static void main(String[] args) {
        System.out.println("=== 1. 同一段程式碼，兩種實作 ===");
        System.out.println("[鏈結串列：全部保留]");
        simulateChat(new LinkedListChatHistory());
        System.out.println("[環狀緩衝區：只留最近 3 則]");
        simulateChat(new CircularBufferChatHistory(3));

        System.out.println("=== 2. 宣告型別決定你能呼叫什麼 ===");
        ChatHistory h1 = new LinkedListChatHistory();
        h1.addMessage("hello");
        // h1.erase();   // 編譯錯誤！h1 的宣告型別是 ChatHistory，裡面沒有 erase()
        ErasableChatHistory h2 = new LinkedListChatHistory();
        h2.addMessage("hello");
        h2.erase();       // OK：ErasableChatHistory 有 erase()
        assert h2.size() == 0;
        System.out.println("  erase 之後 size = " + h2.size());

        System.out.println("=== 3. List ADT：LinkedSimpleList ===");
        SimpleList list = new LinkedSimpleList();
        list.add("apple");
        list.add("banana");
        list.add("cherry");
        System.out.println("  " + list + "，size = " + list.size());
        assert list.size() == 3;
        assert list.get(1).equals("banana");
        list.remove("banana");
        System.out.println("  移除 banana → " + list);
        assert list.size() == 2 && list.get(1).equals("cherry");
        list.remove("not-there");                // 不存在：什麼都不做
        assert list.size() == 2;
        list.remove("apple");                    // 移除第一個節點
        assert list.get(0).equals("cherry");
        try {
            list.get(5);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("  攔到 IndexOutOfBoundsException：" + e.getMessage());
        }
        System.out.println("全部完成。");
    }
}
```

A.3 03_inheritance_lsp：繼承與 LSP

03_inheritance_lsp/BankAccount.java

```java
/**
 * 抽象類別（abstract class）：所有帳戶的共同部分。
 *
 * - 有欄位與具體方法（deposit、getBalance）可以讓子類別「繼承」重用；
 * - 不能直接 new BankAccount()：它只是用來被 extends 的骨架。
 *
 * 契約（子類別都必須遵守）：
 *   資料不變量：balance >= 0
 *   deposit(amount)：前置條件 amount >= 0；後置條件 餘額增加 amount
 */
public abstract class BankAccount {

    /** protected：子類別可以直接讀寫，但無關的外部程式碼不行。 */
    protected int balance;

    public BankAccount() {
        this.balance = 0;
    }

    public void deposit(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("存款金額不可為負數：" + amount);
        }
        this.balance = this.balance + amount;
    }

    public int getBalance() {
        return this.balance;
    }

    /**
     * 抽象方法：每種帳戶的提款規則不同，所以這裡只宣告、不實作。
     * 契約：嘗試提款並回報是否成功；成功時餘額減少 amount。
     * @param amount 前置條件：amount >= 0
     */
    public abstract boolean withdraw(int amount);
}
```

03_inheritance_lsp/SavingsAccount.java

```java
/**
 * 儲蓄帳戶：繼承 BankAccount，補上 withdraw 的實作。
 *
 * SavingsAccount 的契約（比 BankAccount 更明確）：
 *   withdraw(amount)：若 amount <= balance 則扣款並回傳 true；
```

```
 *                      否則不改變狀態並回傳 false。
 *   資料不變量：balance >= 0（繼承自父類別，且永遠成立）
 */
public class SavingsAccount extends BankAccount {

    public SavingsAccount() {
        super();            // 呼叫父類別的建構子（建構子「不會」被繼承，要自己寫）
    }

    @Override
    public boolean withdraw(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("提款金額不可為負數：" + amount);
        }
        if (this.balance < amount) {
            return false;                           // 餘額不足：拒絕，狀態不變
        }
        this.balance = this.balance - amount;       // protected 欄位：子類別可直接存取
        return true;
    }
}
```

03_inheritance_lsp/LoggingSavingsAccount.java

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

/**
 * 會記錄操作的儲蓄帳戶：覆寫 withdraw，但把「真正的提款邏輯」交給 super.withdraw。
 *
 * 這是「擴充而非取代」父類別行為的典型寫法。
 * 它完全遵守 SavingsAccount 的契約（結果一模一樣，只是多記一筆 log），
 * 所以是 LSP 合規的子型別。
 *
 * final：宣告這個類別不能再被繼承。
 */
public final class LoggingSavingsAccount extends SavingsAccount {

    private final List<String> log = new ArrayList<>();

    @Override
    public boolean withdraw(int amount) {
        boolean ok = super.withdraw(amount);          // 先做父類別的事
        if (ok) {
            this.log.add("withdraw " + amount);       // 再加上自己的事
        } else {
```

```
            this.log.add("withdraw " + amount + " FAILED");
        }
        return ok;
    }

    @Override
    public void deposit(int amount) {
        super.deposit(amount);
        this.log.add("deposit " + amount);
    }

    /** 回傳「防禦性複製」：外界拿到的是副本，改不到我們內部的 log。 */
    public List<String> getLog() {
        return new ArrayList<>(this.log);
    }
}
```

03_inheritance_lsp/OverdraftSavingsAccount.java

```
/**
 * 可透支的帳戶：withdraw 永遠成功，餘額可能變成負數。
 *
 * ★ 這個類別「違反 LSP」：
 *   SavingsAccount 的契約說「餘額 >= 0」且「餘額不足時回傳 false」，
 *   但這裡打破了不變量（餘額可負）也違反了後置條件（永遠 true）。
 *   任何以 SavingsAccount 為前提寫的程式，拿到它就可能出錯——即使編譯器完全不會抱怨。
 *
 * 教材保留它，是為了讓你親眼看到 LSP 被違反的後果（見 Main.java）。
 */
public class OverdraftSavingsAccount extends SavingsAccount {

    @Override
    public boolean withdraw(int amount) {
        this.balance = this.balance - amount;   // 不檢查餘額
        return true;                            // 永遠宣稱成功
    }
}
```

03_inheritance_lsp/AccountWithWithdrawalFees.java

```
/**
 * 每次提款收 5 元手續費的帳戶。
 *
 * ★ 設計決定：它 extends BankAccount，而「不是」extends SavingsAccount。
 *
```

```
 * 若繼承 SavingsAccount，程式碼重用會更多，但會違反行為子型別：
 * 對 SavingsAccount 而言「提領剛好等於餘額的金額」一定成功，
 * 但在這裡會因為手續費而失敗——等著 SavingsAccount 行為的呼叫者就被騙了。
 *
 * 改成繼承 BankAccount（契約較寬鬆：只說「嘗試提款並回報結果」），就沒有違約問題。
 * 這就是「重用最方便的父類別」與「行為上正確的父類別」之間的張力。
 */
public class AccountWithWithdrawalFees extends BankAccount {

    public static final int FEE = 5;   // static final：常數，命名全大寫

    public AccountWithWithdrawalFees() {
        super();
    }

    @Override
    public boolean withdraw(int amount) {
        if (amount < 0) {
            throw new IllegalArgumentException("提款金額不可為負數：" + amount);
        }
        int total = amount + FEE;
        if (this.balance < total) {
            return false;
        }
        this.balance = this.balance - total;
        return true;
    }
}
```

03_inheritance_lsp/Main.java

```
/**
 * 03_inheritance_lsp 示範程式。
 *
 * 編譯與執行：
 *   javac *.java
 *   java -ea Main
 */
public class Main {

    /**
     * 這個函式是「以 SavingsAccount 的契約為前提」寫的：
     * 把餘額全部領出來——依契約，提領剛好等於餘額的金額一定成功，之後餘額為 0。
     */
    static void withdrawEverything(SavingsAccount acct) {
        int all = acct.getBalance();
```

```
        boolean ok = acct.withdraw(all);
        System.out.println("  提領 " + all + " → 成功? " + ok
                + "，餘額 = " + acct.getBalance());
        assert ok : "契約保證：提領等於餘額的金額必定成功";
        assert acct.getBalance() == 0 : "契約保證：領光之後餘額為 0";
    }

    public static void main(String[] args) {
        System.out.println("=== 1. 繼承與 super ===");
        SavingsAccount s = new SavingsAccount();
        s.deposit(100);                    // deposit 繼承自 BankAccount，沒有重寫
        System.out.println("  withdraw(30) → " + s.withdraw(30) + "，餘額 " + s.getBalance());
        System.out.println("  withdraw(500) → " + s.withdraw(500) + "，餘額 " + s.getBalance());
        assert s.getBalance() == 70;
        // BankAccount b = new BankAccount();   // 編譯錯誤：抽象類別不能實例化

        System.out.println("=== 2. 動態分派：陣列裡每個物件跑自己的 withdraw ===");
        BankAccount[] accounts = {
            new SavingsAccount(), new LoggingSavingsAccount(), new AccountWithWithdrawalFees()
        };
        for (BankAccount a : accounts) {
            a.deposit(100);
            boolean ok = a.withdraw(100);   // 靜態型別是 BankAccount，執行的卻是各子類別的版本
            System.out.println("  " + a.getClass().getSimpleName()
                    + "：存 100 領 100 → " + ok + "，餘額 " + a.getBalance());
        }

        System.out.println("=== 3. LoggingSavingsAccount：擴充而非取代 ===");
        LoggingSavingsAccount l = new LoggingSavingsAccount();
        l.deposit(50);
        l.withdraw(20);
        l.withdraw(999);
        System.out.println("  log = " + l.getLog());
        assert l.getLog().size() == 3;
        l.getLog().clear();                // 改的是副本（防禦性複製）
        assert l.getLog().size() == 3 : "外界不該能改到內部 log";

        System.out.println("=== 4. LSP：合規的子型別可以放心替換 ===");
        SavingsAccount good = new LoggingSavingsAccount();
        good.deposit(80);
        withdrawEverything(good);          // OK

        System.out.println("=== 5. LSP：違規的子型別會讓正確的程式碼出錯 ===");
        SavingsAccount bad = new OverdraftSavingsAccount();
        bad.deposit(80);
        withdrawEverything(bad);           // 這一步沒問題……
        bad.withdraw(1000);                // 但契約說不可能發生的事發生了：
        System.out.println("  再提 1000 → 餘額 = " + bad.getBalance() + "（負數！不變量被打破）");
```

```
        if (bad.getBalance() < 0) {
            System.out.println("  ★ 編譯器沒有抱怨，但以 SavingsAccount 契約為前提的程式碼全都可能出錯。");
        }
        System.out.println("全部完成。");
    }
}
```

A.4 04_lab_recursion_sorting：遞迴與排序

04_lab_recursion_sorting/Recursion.java

```
import java.util.Arrays;

/**
 * Lab 主題一：遞迴（recursion）。
 *
 * 每個遞迴函式都由兩部分組成：
 *   - 基底情況（base case）：問題小到可以直接回答。
 *   - 遞迴情況（recursive case）：把問題化成「更小的同類問題」，再把答案組合起來。
 *
 * 編譯與執行：javac Recursion.java && java -ea Recursion
 */
public class Recursion {

    /** n! = n × (n-1)!，0! = 1。前置條件：n >= 0 */
    static long factorial(int n) {
        if (n == 0) {                       // 基底情況
            return 1;
        }
        return n * factorial(n - 1);        // 遞迴情況：問題變小（n-1）
    }

    /** 陣列 arr[from..] 的總和：第一個元素 + 其餘元素的總和。 */
    static int sum(int[] arr, int from) {
        if (from >= arr.length) {           // 基底：沒有元素了
            return 0;
        }
        return arr[from] + sum(arr, from + 1);
    }

    /** 反轉字串：最後一個字元 + 反轉「去掉最後一個字元的字串」。 */
    static String reverse(String s) {
        if (s.length() <= 1) {              // 基底：空字串或單一字元
            return s;
        }
```

```
        return s.charAt(s.length() - 1) + reverse(s.substring(0, s.length() - 1));
    }

    /** 迴文判斷：頭尾相同，且中間也是迴文。 */
    static boolean isPalindrome(String s) {
        if (s.length() <= 1) {
            return true;
        }
        if (s.charAt(0) != s.charAt(s.length() - 1)) {
            return false;
        }
        return isPalindrome(s.substring(1, s.length() - 1));
    }

    /** 快速冪：base^exp，利用「平方」讓問題每次減半（分治法）。前置條件：exp >= 0 */
    static long power(long base, int exp) {
        if (exp == 0) {
            return 1;
        }
        long half = power(base, exp / 2);    // 只算一次，再重複使用
        if (exp % 2 == 0) {
            return half * half;
        }
        return half * half * base;
    }

    /**
     * 遞迴二分搜尋（分治法）。
     * @param haystack 已由小到大排序的陣列
     * @return needle 的索引；找不到回傳 -1
     */
    static int binarySearch(int[] haystack, int needle, int lo, int hi) {
        if (lo > hi) {                       // 基底：範圍變空，確定不存在
            return -1;
        }
        int mid = (lo + hi) / 2;
        if (haystack[mid] == needle) {
            return mid;
        } else if (haystack[mid] < needle) {
            return binarySearch(haystack, needle, mid + 1, hi);   // 往右半邊找
        } else {
            return binarySearch(haystack, needle, lo, mid - 1);   // 往左半邊找
        }
    }

    /** 對外的簡潔版本：呼叫者不需要知道 lo/hi 這兩個「多出來的參數」。 */
    static int binarySearch(int[] haystack, int needle) {
        return binarySearch(haystack, needle, 0, haystack.length - 1);
```

```
    }

    /** 河內塔：把 n 個盤子從 from 移到 to，回傳移動次數（= 2^n − 1）。 */
    static int hanoi(int n, char from, char to, char via, boolean print) {
        if (n == 0) {
            return 0;
        }
        int moves = hanoi(n − 1, from, via, to, print);   // 先把上面 n−1 個移到暫存柱
        if (print) {
            System.out.println("    盤子 " + n + "：" + from + " → " + to);
        }
        moves = moves + 1;
        moves = moves + hanoi(n − 1, via, to, from, print); // 再把 n−1 個移到目標柱
        return moves;
    }

    /** 沒有基底情況、或永遠碰不到基底情況，就會 StackOverflowError。 */
    static int countDown(int n) {
        if (n == 0) {
            return 0;
        }
        return 1 + countDown(n − 1);
    }

    public static void main(String[] args) {
        System.out.println("=== 遞迴測試 ===");
        assert factorial(0) == 1;
        assert factorial(5) == 120;
        assert factorial(10) == 3628800L;
        System.out.println("  5! = " + factorial(5) + "，10! = " + factorial(10));

        int[] a = {3, 1, 4, 1, 5, 9, 2, 6};
        assert sum(a, 0) == 31;
        assert sum(new int[]{}, 0) == 0;
        System.out.println("  sum" + Arrays.toString(a) + " = " + sum(a, 0));

        assert reverse("").equals("");
        assert reverse("a").equals("a");
        assert reverse("Java").equals("avaJ");
        System.out.println("  reverse(\"Java\") = " + reverse("Java"));

        assert isPalindrome("");
        assert isPalindrome("racecar");
        assert !isPalindrome("Java");
        System.out.println("  isPalindrome(\"racecar\") = " + isPalindrome("racecar"));

        assert power(2, 10) == 1024;
        assert power(3, 0) == 1;
```

```
        assert power(7, 3) == 343;
        System.out.println("  2^10 = " + power(2, 10));

        int[] sorted = {1, 3, 5, 7, 9, 11, 13};
        assert binarySearch(sorted, 7) == 3;
        assert binarySearch(sorted, 1) == 0;
        assert binarySearch(sorted, 13) == 6;
        assert binarySearch(sorted, 4) == -1;
        assert binarySearch(new int[]{}, 4) == -1;
        System.out.println("  binarySearch(sorted, 7) = " + binarySearch(sorted, 7));

        System.out.println("  河內塔 3 個盤子：");
        int moves = hanoi(3, 'A', 'C', 'B', true);
        assert moves == 7;
        assert hanoi(10, 'A', 'C', 'B', false) == 1023;
        System.out.println("  共 " + moves + " 步");

        System.out.println("  countDown(1000) = " + countDown(1000));
        try {
            // 遞迴太深：每一層呼叫都要占一個堆疊框（stack frame），堆疊是有限的。
            // 實際能撐幾層取決於 JVM 設定（-Xss）與方法大小，通常是數千到數萬層。
            countDown(1_000_000);
        } catch (StackOverflowError e) {
            System.out.println("  countDown(1000000) → StackOverflowError（堆疊溢位）");
        }
        System.out.println("全部通過。");
    }
}
```

04_lab_recursion_sorting/Sorting.java

```
import java.util.Arrays;
import java.util.Random;

/**
 * Lab 主題二：排序（sorting）。
 *
 * 任何「就地」（in-place）排序都必須滿足兩個後置條件：
 *   1. 執行後陣列是遞增排序的；
 *   2. 執行後的陣列是原本元素的一個「排列」（沒有多、沒有少、沒有變）。
 * 很多人只記得第 1 條，忘了第 2 條！
 *
 * 編譯與執行：javac Sorting.java && java -ea Sorting
 */
public class Sorting {

```

```
    // 用來計算「比較次數」，方便比較演算法的效率
    static long comparisons = 0;

    static boolean less(int x, int y) {
        comparisons = comparisons + 1;
        return x < y;
    }

    static void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }

    /** 選擇排序：每回合從「未排序區」挑出最小值，換到未排序區的最前面。O(n^2) */
    static void selectionSort(int[] arr) {
        int i = 0;
        while (i < arr.length) {
            // INV: arr[0..i-1] 已排序，且都 <= arr[i..] 的每個元素
            int minPos = i;
            int j = i + 1;
            while (j < arr.length) {
                if (less(arr[j], arr[minPos])) {
                    minPos = j;
                }
                j = j + 1;
            }
            swap(arr, i, minPos);
            i = i + 1;
        }
    }

    /** 氣泡排序（含提前結束）：相鄰兩兩比較，把最大值一路「冒」到尾端。O(n^2)，已排序時 O(n) */
    static void bubbleSort(int[] arr) {
        int end = arr.length - 1;
        while (end > 0) {
            // INV: arr[end+1..] 已排序，且都 >= arr[0..end] 的每個元素
            boolean swapped = false;
            int j = 0;
            while (j < end) {
                if (less(arr[j + 1], arr[j])) {
                    swap(arr, j, j + 1);
                    swapped = true;
                }
                j = j + 1;
            }
            if (!swapped) {
                return;              // 整趟都沒交換：已經排好了，提前結束
```

```
            }
            end = end - 1;
        }
    }

    /** 插入排序：把下一個元素「插入」已排序區的正確位置。O(n^2)，已排序時 O(n) */
    static void insertionSort(int[] arr) {
        int i = 1;
        while (i < arr.length) {
            // INV: arr[0..i-1] 已排序
            int x = arr[i];
            int j = i - 1;
            while (j >= 0 && less(x, arr[j])) {
                arr[j + 1] = arr[j];     // 比 x 大的元素往右挪一格
                j = j - 1;
            }
            arr[j + 1] = x;              // 放進空出來的位置
            i = i + 1;
        }
    }

    /**
     * 合併兩個「已排序」的陣列成一個已排序的新陣列。
     * 只需各走訪一次：O(left.length + right.length)。
     */
    static int[] merge(int[] left, int[] right) {
        int[] out = new int[left.length + right.length];
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (!less(right[j], left[i])) {      // left[i] <= right[j]：取左邊（相等取左邊 → 穩定）
                out[k] = left[i];
                i = i + 1;
            } else {
                out[k] = right[j];
                j = j + 1;
            }
            k = k + 1;
        }
        while (i < left.length) {                // 左邊剩下的直接搬過來
            out[k] = left[i];
            i = i + 1;
            k = k + 1;
        }
        while (j < right.length) {               // 右邊剩下的直接搬過來
            out[k] = right[j];
            j = j + 1;
            k = k + 1;
```

```
        }
        return out;
    }

    /** 合併排序（分治法）：切半 → 遞迴排序兩半 → 合併。O(n log n)，但需要 O(n) 額外空間 */
    static int[] mergeSort(int[] input, int start, int end) {
        if (end < start) {                      // 基底：空範圍
            return new int[]{};
        }
        if (end == start) {                     // 基底：單一元素本身就是排好的
            return new int[]{input[start]};
        }
        int mid = (start + end) / 2;            // Divide
        int[] left = mergeSort(input, start, mid);      // Conquer
        int[] right = mergeSort(input, mid + 1, end);
        return merge(left, right);              // Combine
    }

    /** 對外版本：讓合併排序也符合 sort(int[]) 的「就地」契約（把結果複製回原陣列）。 */
    static void mergeSort(int[] arr) {
        int[] sorted = mergeSort(arr, 0, arr.length - 1);
        System.arraycopy(sorted, 0, arr, 0, arr.length);
    }

    static boolean isSorted(int[] arr) {
        int i = 0;
        while (i < arr.length - 1) {
            if (arr[i] > arr[i + 1]) {
                return false;
            }
            i = i + 1;
        }
        return true;
    }

    // ---------- 測試 ----------

    /** 排序演算法的介面（函式介面）：讓測試程式可以「把演算法當參數傳」。 */
    interface Sorter {
        void sort(int[] arr);
    }

    static void checkSorter(String name, Sorter sorter) {
        int[][] cases = {
            {},                             // 空陣列
            {42},                           // 單一元素
            {7, 7},                         // 兩個相同
            {1, 2, 3, 4, 5, 6, 7, 8, 9},    // 已排序
```

```
            {9, 8, 7, 6, 5, 4, 3, 2, 1},     // 反向排序
            {6, -5, 212, 42, 19, 0, 0, 5, 9718},   // 隨機（含負數與重複）
            {-1, -2, -3, -4, -5},
        };
        for (int[] original : cases) {
            int[] toSort = original.clone();
            int[] expected = original.clone();
            Arrays.sort(expected);                          // 用標準函式庫的答案當「標準答案」
            sorter.sort(toSort);
            assert isSorted(toSort) : name + " 後置條件 1 失敗：" + Arrays.toString(toSort);
            assert Arrays.equals(toSort, expected)          // 同時檢查後置條件 2（是原元素的排列）
                    : name + " 後置條件 2 失敗：" + Arrays.toString(toSort);
        }
        System.out.println("  " + name + "：" + cases.length + " 組測試通過");
    }

    static void countComparisons(String name, Sorter sorter, int[] data) {
        int[] copy = data.clone();
        comparisons = 0;
        sorter.sort(copy);
        System.out.printf("  %-14s 比較次數 = %,d%n", name, comparisons);
    }

    public static void main(String[] args) {
        System.out.println("=== 正確性測試 ===");
        checkSorter("selectionSort", Sorting::selectionSort);
        checkSorter("bubbleSort", Sorting::bubbleSort);
        checkSorter("insertionSort", Sorting::insertionSort);
        checkSorter("mergeSort", Sorting::mergeSort);

        int n = 2000;
        int[] random = new Random(42).ints(n, 0, 100_000).toArray();
        int[] sorted = random.clone();
        Arrays.sort(sorted);
        int[] reversed = new int[n];
        for (int i = 0; i < n; i++) {
            reversed[i] = sorted[n - 1 - i];
        }

        System.out.println("=== 比較次數（n = " + n + "，隨機資料）===");
        countComparisons("selectionSort", Sorting::selectionSort, random);
        countComparisons("bubbleSort", Sorting::bubbleSort, random);
        countComparisons("insertionSort", Sorting::insertionSort, random);
        countComparisons("mergeSort", Sorting::mergeSort, random);

        System.out.println("=== 比較次數（n = " + n + "，已排序資料：最佳情況）===");
        countComparisons("selectionSort", Sorting::selectionSort, sorted);
        countComparisons("bubbleSort", Sorting::bubbleSort, sorted);
```

```
        countComparisons("insertionSort", Sorting::insertionSort, sorted);
        countComparisons("mergeSort", Sorting::mergeSort, sorted);

        System.out.println("=== 比較次數（n = " + n + "，反向排序：最差情況）===");
        countComparisons("selectionSort", Sorting::selectionSort, reversed);
        countComparisons("bubbleSort", Sorting::bubbleSort, reversed);
        countComparisons("insertionSort", Sorting::insertionSort, reversed);
        countComparisons("mergeSort", Sorting::mergeSort, reversed);
        System.out.println("全部通過。");
    }
}
```

A.5 05_junit_tdd：JUnit 與 TDD

05_junit_tdd/GuestList.java

```
/**
 * 賓客名單介面：一個「集合」（set）語意的 ADT。
 *
 * 語意：一位賓客要嘛「會來」要嘛「不會來」；
 *       同一個人 add 兩次的效果，和 add 一次完全一樣。
 * 介面完全沒有規定要怎麼實作——這正是黑箱測試能「一份測試、多種實作」的原因。
 */
public interface GuestList {

    /** 記錄某位賓客會來。 */
    void add(String name);

    /** 記錄某位賓客不來了（若原本就不在名單上，什麼都不做）。 */
    void remove(String name);

    /** @return 該賓客目前是否會來 */
    boolean isComing(String name);

    /** @return 目前會來的賓客人數（不重複計算） */
    int count();

    /**
     * @return 目前會來的賓客名字（防禦性複製：呼叫者改這個陣列不會影響名單）
     */
    String[] coming();
}
```

05_junit_tdd/GuestArrayList.java

```java
import java.util.Objects;

/**
 * 實作一：「懶惰」（lazy）的陣列版。
 *
 * add 不檢查重複，直接塞進陣列；重複要等到 remove 時才「一次把所有同名的都清成 null」。
 * 陣列滿了就「倍增」（array doubling）——這是攤銷 O(1) 插入的關鍵技巧。
 *
 * 資料不變量：
 *   0 <= used <= data.length
 *   data[used..] 全部是 null（尚未使用）
 * 注意：data[0..used-1] 可能含 null（被移除的洞）也可能含重複的名字。
 */
public class GuestArrayList implements GuestList {

    // package-private（沒有修飾子）：同一個套件裡的類別（例如子類別）可以直接存取
    String[] data = new String[2];
    int used = 0;                          // data[0..used-1] 是「用過」的格子

    @Override
    public void add(String name) {
        if (name == null) {                          // null 會跟「被移除的洞」混在一起，拒絕
            throw new IllegalArgumentException("賓客名字不可為 null");
        }
        if (this.used == this.data.length) {         // 滿了：換一個兩倍大的陣列
            String[] old = this.data;
            this.data = new String[this.used * 2];
            int i = 0;
            while (i < this.used) {
                this.data[i] = old[i];               // 把舊資料逐一複製過去：O(n)
                i = i + 1;
            }
        }
        this.data[this.used] = name;
        this.used = this.used + 1;
    }

    @Override
    public void remove(String name) {
        int i = 0;
        while (i < this.used) {                      // 懶惰法：必須掃完整個陣列，清掉所有同名
            if (this.data[i] != null && this.data[i].equals(name)) {
                this.data[i] = null;
            }
            i = i + 1;
        }
    }
```

```
48
49      @Override
50      public boolean isComing(String name) {
51          int i = 0;
52          while (i < this.used) {
53              if (Objects.equals(this.data[i], name)) { // Objects.equals：兩邊都可能是 null 也安全
54                  return true;
55              }
56              i = i + 1;
57          }
58          return false;
59      }
60
61      @Override
62      public int count() {
63          int n = 0;
64          int i = 0;
65          while (i < this.used) {
66              // 只在「第一次出現」時計數，這樣重複的名字不會被重複計算
67              if (this.data[i] != null && indexOf(this.data[i]) == i) {
68                  n = n + 1;
69              }
70              i = i + 1;
71          }
72          return n;
73      }
74
75      /** @return name 第一次出現的索引；不存在則回傳 -1 */
76      int indexOf(String name) {
77          int i = 0;
78          while (i < this.used) {
79              if (this.data[i] != null && this.data[i].equals(name)) {
80                  return i;
81              }
82              i = i + 1;
83          }
84          return -1;
85      }
86
87      @Override
88      public String[] coming() {
89          String[] copy = new String[count()];        // 全新的陣列：防禦性複製
90          int k = 0;
91          int i = 0;
92          while (i < this.used) {
93              if (this.data[i] != null && indexOf(this.data[i]) == i) {
94                  copy[k] = this.data[i];
95                  k = k + 1;
```

```
            }
            i = i + 1;
        }
        return copy;
    }
}
```

05_junit_tdd/GuestArrayUniqueList.java

```
/**
 * 實作二：「嚴格」（strict）的陣列版——用繼承重用 GuestArrayList 的程式碼。
 *
 * add 先檢查 isComing，已經在名單上就直接 return，因此多了一條更強的資料不變量：
 *   data[0..used-1] 中所有非 null 的名字互不相同。
 *
 * 有了這條不變量，remove 找到第一個就可以停（不必掃完整個陣列），
 * count 也不必再檢查重複。
 *
 * 對外行為（介面契約）與 GuestArrayList 完全一致 → 這是 LSP 合規的子型別。
 */
public class GuestArrayUniqueList extends GuestArrayList {

    @Override
    public void add(String name) {
        if (isComing(name)) {
            return;                    // 已經在名單上：什麼都不做
        }
        super.add(name);               // 其餘邏輯（含陣列倍增）跟父類別一樣，直接重用
    }

    @Override
    public void remove(String name) {
        int i = 0;
        while (i < this.used) {
            if (this.data[i] != null && this.data[i].equals(name)) {
                this.data[i] = null;
                return;                // 不變量保證最多一個：找到就可以停
            }
            i = i + 1;
        }
    }

    @Override
    public int count() {
        int n = 0;
        int i = 0;
```

```
        while (i < this.used) {
            if (this.data[i] != null) {   // 不必再檢查重複
                n = n + 1;
            }
            i = i + 1;
        }
        return n;
    }
}
```

05_junit_tdd/GuestListTest.java

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.util.Arrays;

import static org.junit.jupiter.api.Assertions.*;

/**
 * GuestList 的 JUnit 5 黑箱測試。
 *
 * 重點：
 *   - 變數宣告成 GuestList（介面），而不是某個實作類別：測的是「契約」，不是實作細節。
 *   - 每個 @Test 方法都是獨立的：JUnit 會為每個測試建立一個「新的」測試物件，
 *     所以 @BeforeEach 裡建立的名單不會在測試之間互相干擾。
 *   - 同一組測試對兩種實作都跑一次（見 LazyTest / StrictTest 兩個子類別）。
 *
 * 在 IntelliJ 中：把游標放在紅色的 @Test 上按 Alt+Enter → Add JUnit5 to classpath，
 * 然後按類別名旁邊的綠色播放鍵即可執行。
 */
abstract class GuestListTest {

    /** 由子類別決定要測哪一種實作（工廠方法）。 */
    abstract GuestList create();

    private GuestList gl;

    @BeforeEach
    void setUp() {
        gl = create();                          // 每個測試開始前都拿到一份全新的空名單
    }

    @Test
    @DisplayName("新名單是空的")
```

```
    void emptyListHasNobody() {
        assertFalse(gl.isComing("Fabian"));
        assertEquals(0, gl.count());
        assertEquals(0, gl.coming().length);
    }

    @Test
    @DisplayName("add 之後 isComing 為 true")
    void addThenIsComing() {
        gl.add("Fabian");
        assertTrue(gl.isComing("Fabian"));
        assertFalse(gl.isComing("Nicholas"));
        assertEquals(1, gl.count());
    }

    @Test
    @DisplayName("集合語意：同一個人 add 兩次等於 add 一次")
    void addingTwiceIsIdempotent() {
        gl.add("Fabian");
        gl.add("Fabian");
        assertTrue(gl.isComing("Fabian"));
        assertEquals(1, gl.count(), "重複加入不應被重複計算");
        gl.remove("Fabian");
        assertFalse(gl.isComing("Fabian"), "移除一次之後必須完全消失");
        assertEquals(0, gl.count());
    }

    @Test
    @DisplayName("remove 只影響被移除的人")
    void removeAffectsOnlyThatGuest() {
        gl.add("Fabian");
        gl.add("Nicholas");
        gl.remove("Fabian");
        assertFalse(gl.isComing("Fabian"));
        assertTrue(gl.isComing("Nicholas"));
        assertEquals(1, gl.count());
    }

    @Test
    @DisplayName("從空名單或不存在的人 remove 不會出錯")
    void removeFromEmptyOrMissing() {
        gl.remove("Fabian");                    // 空名單
        assertFalse(gl.isComing("Fabian"));
        gl.add("Ann");
        gl.remove("Bob");                       // 不存在的人
        assertTrue(gl.isComing("Ann"));
        assertEquals(1, gl.count());
    }
```

```
83
84      @Test
85      @DisplayName("移除後可以再加回來")
86      void reAddAfterRemove() {
87          gl.add("Fabian");
88          gl.remove("Fabian");
89          gl.add("Fabian");
90          assertTrue(gl.isComing("Fabian"));
91          assertEquals(1, gl.count());
92      }
93
94      @Test
95      @DisplayName("加入很多人：觸發陣列倍增也要正確")
96      void manyGuestsTriggerResize() {
97          int i = 0;
98          while (i < 1000) {
99              gl.add("guest" + i);
100             i = i + 1;
101         }
102         assertEquals(1000, gl.count());
103         assertTrue(gl.isComing("guest0"));
104         assertTrue(gl.isComing("guest999"));
105         assertFalse(gl.isComing("guest1000"));
106     }
107
108     @Test
109     @DisplayName("coming() 回傳的是防禦性複製，改它不影響名單")
110     void comingReturnsDefensiveCopy() {
111         gl.add("Fabian");
112         gl.add("Nicholas");
113         String[] snapshot = gl.coming();
114         assertEquals(2, snapshot.length);
115         String[] sortedSnapshot = snapshot.clone();
116         Arrays.sort(sortedSnapshot);
117         assertArrayEquals(new String[]{"Fabian", "Nicholas"}, sortedSnapshot);
118
119         snapshot[0] = "Intruder";              // 惡意（或不小心）改動回傳的陣列
120         assertFalse(gl.isComing("Intruder"), "封裝：外部改動不得影響內部狀態");
121         assertEquals(2, gl.count());
122     }
123
124     @Test
125     @DisplayName("assertThrows 示範：null 名字不合法（本實作選擇拋出例外）")
126     void nullNameIsRejected() {
127         // 這個測試示範 assertThrows 的用法；GuestList 介面本身沒有規定 null 的行為，
128         // 所以嚴格來說它是「白箱」測試——只對我們自己的實作成立。
129         IllegalArgumentException e =
130                 assertThrows(IllegalArgumentException.class, () -> gl.add(null));
```

```
        assertNotNull(e.getMessage());
    }
}

/** 對「懶惰」實作跑整套測試。 */
class LazyGuestListTest extends GuestListTest {
    @Override
    GuestList create() {
        return new GuestArrayList();
    }
}

/** 對「嚴格」實作跑同一套測試。 */
class StrictGuestListTest extends GuestListTest {
    @Override
    GuestList create() {
        return new GuestArrayUniqueList();
    }
}
```

A.6 06_generics_collections：泛型與 Collections

06_generics_collections/GuestList.java

```
/**
 * 泛型版的 GuestList：型別參數 T 是「元素型別的佔位符」。
 *
 * GuestList<String> 就是名字的集合、GuestList<Integer> 就是學號的集合……
 * 寫一次介面，就能安全地用在任何型別上，而且不需要轉型（cast）。
 */
public interface GuestList<T> {

    void add(T guest);

    void remove(T guest);

    boolean isComing(T guest);

    int count();

    /** @return 目前會來的賓客（防禦性複製） */
    T[] coming();
}
```

06_generics_collections/ArrayGuestList.java

```java
import java.util.Arrays;

/**
 * 泛型版的「嚴格」陣列實作。
 *
 * 型別擦除（type erasure）的限制：執行期不知道 T 是什麼，所以「不能」寫 new T[n]。
 * 標準做法：內部用 Object[] 存，取出時再轉型成 T。
 * 只要我們靠封裝保證「陣列裡只放過 T」，這個「未受檢查的轉型」在實務上就是安全的。
 *
 * 資料不變量：data[0..used-1] 非 null 且互不相同（用 equals 比較）。
 */
public class ArrayGuestList<T> implements GuestList<T> {

    private Object[] data = new Object[2];   // 不能是 T[]！
    private int used = 0;

    @Override
    public void add(T guest) {
        if (guest == null) {
            throw new IllegalArgumentException("賓客不可為 null");
        }
        if (isComing(guest)) {
            return;
        }
        if (this.used == this.data.length) {
            this.data = Arrays.copyOf(this.data, this.used * 2);   // 標準函式庫幫我們倍增＋複製
        }
        this.data[this.used] = guest;
        this.used = this.used + 1;
    }

    @Override
    public void remove(T guest) {
        int i = 0;
        while (i < this.used) {
            if (this.data[i].equals(guest)) {
                // 用最後一個元素填補洞，保持 data[0..used-1] 沒有 null
                this.data[i] = this.data[this.used - 1];
                this.data[this.used - 1] = null;
                this.used = this.used - 1;
                return;
            }
            i = i + 1;
        }
    }
```

```
    @Override
    public boolean isComing(T guest) {
        int i = 0;
        while (i < this.used) {
            if (this.data[i].equals(guest)) {
                return true;
            }
            i = i + 1;
        }
        return false;
    }

    @Override
    public int count() {
        return this.used;
    }

    /**
     * ★ 型別擦除的陷阱：這個陣列在執行期「真的」是 Object[]。
     * 呼叫端若寫 String[] s = list.coming(); 會在執行期拋出 ClassCastException！
     * （寫 Object[] s = list.coming(); 則沒問題。）見 Demo.java 的示範與 comingList() 的正解。
     */
    @Override
    @SuppressWarnings("unchecked")   // 告訴編譯器：我知道這個轉型沒被檢查，我負責它的安全
    public T[] coming() {
        // Arrays.copyOf 回傳的是新陣列（防禦性複製）；(T[]) 是「未受檢查的轉型」
        return (T[]) Arrays.copyOf(this.data, this.used);
    }

    /** 正解：回傳 List<T>。泛型集合沒有陣列的執行期型別問題，這也是標準函式庫的做法。 */
    @SuppressWarnings("unchecked")
    public java.util.List<T> comingList() {
        java.util.List<T> result = new java.util.ArrayList<>();
        int i = 0;
        while (i < this.used) {
            result.add((T) this.data[i]);   // 單一元素的轉型：執行期什麼都不檢查（T 已被擦除）
            i = i + 1;
        }
        return result;
    }
}
```

06_generics_collections/Pair.java

```
/**
```

```
 * 有兩個型別參數的泛型 record：Pair<A, B>。
 *
 * record 會自動產生建構子、存取器（first()、second()）、equals、hashCode 與 toString，
 * 是不可變（immutable）資料的最佳寫法——這也讓它成為安全的雜湊表鍵（見第 8 章）。
 */
public record Pair<A, B>(A first, B second) {

    /** 泛型「靜態方法」：型別參數寫在回傳型別前面。 */
    public static <A, B> Pair<A, B> of(A first, B second) {
        return new Pair<>(first, second);
    }

    /** 交換前後：回傳一個新的 Pair<B, A>（record 是不可變的，所以不是「修改自己」）。 */
    public Pair<B, A> swap() {
        return new Pair<>(this.second, this.first);
    }
}
```

06_generics_collections/Demo.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * 06_generics_collections 示範程式：Object、轉型、泛型、型別擦除、Collections API。
 *
 * 編譯與執行：
 *   javac *.java
 *   java -ea Demo
 */
public class Demo {

    /** 泛型方法：對「任何型別」的 List 都能用；T 由編譯器從參數推論。 */
    static <T> T firstOrDefault(List<T> list, T fallback) {
        if (list.isEmpty()) {
            return fallback;
        }
        return list.get(0);
```

```
    }

    /** 有界型別參數：T 必須是 Comparable<T>，才能呼叫 compareTo。 */
    static <T extends Comparable<T>> T max(List<T> list) {
        if (list.isEmpty()) {
            throw new IllegalArgumentException("空串列沒有最大值");
        }
        T best = list.get(0);
        for (T x : list) {
            if (x.compareTo(best) > 0) {
                best = x;
            }
        }
        return best;
    }

    public static void main(String[] args) {
        System.out.println("=== 1. Object：所有類別的共同祖先 ===");
        Object x = "hello";        // String 是 Object 的子型別
        Object y = 42;             // 自動裝箱：int → Integer，Integer 是 Object 的子型別
        System.out.println("  x = " + x + "，y = " + y);
        // x.length();             // 編譯錯誤：編譯器只知道 x 是 Object，Object 沒有 length()

        System.out.println("=== 2. 轉型（cast）與 ClassCastException ===");
        String s = (String) x;     // 向下轉型：告訴編譯器「相信我，它其實是 String」
        System.out.println("  s.length() = " + s.length());
        try {
            Integer bad = (Integer) x;   // 編譯器放行，執行期才發現不對
            System.out.println(bad);
        } catch (ClassCastException e) {
            System.out.println("  攔到 ClassCastException：" + e.getMessage());
        }
        // 安全的做法：先用 instanceof 檢查（Java 16+ 的「模式比對」還能順便宣告變數）
        if (y instanceof Integer n) {
            System.out.println("  y 是 Integer，加 1 = " + (n + 1));
        }

        System.out.println("=== 3. 泛型：型別安全，不必轉型 ===");
        GuestList<String> names = new ArrayGuestList<>();   // <> 讓編譯器推論型別參數
        names.add("Fabian");
        names.add("Nicholas");
        names.add("Fabian");                                 // 集合語意：不重複
        // names.add(42);          // 編譯錯誤！42 不是 String：錯誤在編譯期就被抓到
        assert names.count() == 2;
        System.out.println("  names.count() = " + names.count());

        GuestList<Integer> ids = new ArrayGuestList<>();
        ids.add(1001);
```

```
        ids.add(1002);
        assert ids.isComing(1001) && !ids.isComing(9999);
        System.out.println("  ids.isComing(1001) = " + ids.isComing(1001));

        System.out.println("=== 4. 型別擦除的陷阱 ===");
        Object[] ok = names.coming();          // OK：執行期的陣列真的是 Object[]
        System.out.println("  Object[] ok = " + Arrays.toString(ok));
        try {
            String[] boom = names.coming();    // 編譯通過，但執行期插入的 checkcast String[] 會失敗
            System.out.println(boom.length);
        } catch (ClassCastException e) {
            System.out.println("  String[] boom = names.coming() → ClassCastException！");
            System.out.println("  （T[] 在執行期被擦除成 Object[]，這是 Java 泛型的真實限制）");
        }
        List<String> safe = ((ArrayGuestList<String>) names).comingList();
        System.out.println("  正解 comingList() = " + safe + "（用 List<T> 就沒有這個問題）");

        System.out.println("=== 5. Collections API：List ===");
        List<String> fruits = new ArrayList<>();              // 宣告用介面 List，實作用 ArrayList
        fruits.add("banana");
        fruits.add("apple");
        fruits.add("cherry");
        fruits.add("apple");                                  // List 允許重複、保持順序
        System.out.println("  fruits = " + fruits + "，size = " + fruits.size());
        System.out.println("  get(1) = " + fruits.get(1) + "，contains(cherry) = " + fruits.contains("cherry"));
        Collections.sort(fruits);
        System.out.println("  排序後 = " + fruits);
        fruits.remove("apple");                               // 只移除第一個
        System.out.println("  remove(apple) → " + fruits);
        for (String f : fruits) {                             // for-each：走訪任何 Collection
            System.out.println("    - " + f);
        }

        System.out.println("=== 6. Collections API：Set ===");
        Set<String> guests = new TreeSet<>();                 // TreeSet：自動排序、不重複
        guests.add("Liam");
        guests.add("Thomas");
        guests.add("Alice");
        guests.add("Liam");                                   // 重複：無效
        System.out.println("  TreeSet = " + guests + "（排序）");
        Set<String> hashGuests = new HashSet<>(guests);       // HashSet：更快，但順序不保證
        System.out.println("  HashSet contains(Alice) = " + hashGuests.contains("Alice"));
        assert guests.size() == 3 && hashGuests.size() == 3;

        System.out.println("=== 7. Collections API：Map ===");
        Map<String, Integer> ages = new HashMap<>();          // 鍵 → 值
```

```
        ages.put("Liam", 30);
        ages.put("Thomas", 25);
        ages.put("Liam", 31);                                   // 同一個鍵：覆蓋舊值
        System.out.println("  ages.get(Liam) = " + ages.get("Liam") + "，get(Nobody) = " + ages.get("Nobody"));
        System.out.println("  getOrDefault(Nobody, -1) = " + ages.getOrDefault("Nobody", -1));
        for (Map.Entry<String, Integer> e : new TreeMap<>(ages).entrySet()) {  // TreeMap：依鍵排序走訪
            System.out.println("    " + e.getKey() + " → " + e.getValue());
        }
        // 經典應用：計數
        Map<Character, Integer> freq = new TreeMap<>();
        for (char c : "mississippi".toCharArray()) {
            freq.put(c, freq.getOrDefault(c, 0) + 1);
        }
        System.out.println("  字元頻率 = " + freq);
        assert freq.get('s') == 4;

        System.out.println("=== 8. of() 工廠方法：不可變集合 ===");
        List<Integer> nums = List.of(3, 1, 4, 1, 5);
        try {
            nums.add(9);
        } catch (UnsupportedOperationException e) {
            System.out.println("  List.of(...) 是不可變的：add 會拋出 UnsupportedOperationException");
        }
        List<Integer> mutable = new ArrayList<>(nums);          // 要可變就複製一份
        mutable.add(9);
        System.out.println("  new ArrayList<>(nums) 之後 add(9) → " + mutable);

        System.out.println("=== 9. 泛型方法與 record ===");
        System.out.println("  firstOrDefault([], \"none\") = " + firstOrDefault(new ArrayList<String>(), "none"));
        System.out.println("  max([3,1,4,1,5]) = " + max(nums));
        System.out.println("  max([\"pear\",\"apple\"]) = " + max(List.of("pear", "apple")));
        Pair<String, Integer> p = Pair.of("Liam", 30);
        System.out.println("  p = " + p + "，p.swap() = " + p.swap());
        assert p.equals(new Pair<>("Liam", 30));                // record 自動有正確的 equals

        System.out.println("=== 10. 自動裝箱的陷阱：Integer 請用 equals 比較 ===");
        Integer a = 127, b = 127, c = 1000, d = 1000;
        System.out.println("  127 == 127 → " + (a == b) + "（小整數被快取，碰巧相同）");
        System.out.println("  1000 == 1000 → " + (c == d) + "（不同物件！）");
        System.out.println("  1000.equals(1000) → " + c.equals(d));
        System.out.println("全部完成。");
    }
}
```

A.7 07_hashing：雜湊

07_hashing/StringHashing.java

```java
/**
 * 雜湊函數（hash function）實驗：字串的多項式滾動雜湊 vs 壞的雜湊。
 *
 * 編譯與執行：javac StringHashing.java && java -ea StringHashing
 */
public class StringHashing {

    /**
     * 多項式滾動雜湊（polynomial rolling hash）：
     *   h = s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
     * 每一步把累積值乘 31 再加下一個字元，所以「位置」也會影響結果（"ab" != "ba"）。
     * 這基本上就是 java.lang.String.hashCode() 的演算法。
     * 注意：int 會溢位（overflow），這沒關係——溢位後的值一樣是確定性的。
     */
    static int goodHash(String s) {
        int h = 0;
        int i = 0;
        while (i < s.length()) {
            h = 31 * h + s.charAt(i);
            i = i + 1;
        }
        return h;
    }

    /** 只把字元加總：位置資訊丟了，"ab" 與 "ba" 必撞（分布差）。 */
    static int mediocreHash(String s) {
        int h = 0;
        for (char c : s.toCharArray()) {
            h = h + c;
        }
        return h;
    }

    /** 「正確」但完全沒用的雜湊：滿足等價一致性與確定性，卻讓所有鍵撞在同一桶。 */
    static int badHash(String s) {
        return 0;
    }

    /** 把雜湊值變成合法的桶索引：先取絕對值再取餘數（注意 Integer.MIN_VALUE 的陷阱）。 */
    static int bucketOf(int hash, int capacity) {
        return Math.floorMod(hash, capacity);   // floorMod 永遠回傳 0..capacity-1，比 Math.abs(h) % c 安全
    }
```

```
43
44      interface Hasher {
45          int hash(String s);
46      }
47
48      /** 統計 words 被 hasher 分到 capacity 個桶時，最擠的桶有幾個鍵、有幾個空桶。 */
49      static void report(String name, Hasher hasher, String[] words, int capacity) {
50          int[] counts = new int[capacity];
51          for (String w : words) {
52              counts[bucketOf(hasher.hash(w), capacity)]++;
53          }
54          int maxBucket = 0, empty = 0;
55          for (int c : counts) {
56              if (c > maxBucket) {
57                  maxBucket = c;
58              }
59              if (c == 0) {
60                  empty++;
61              }
62          }
63          System.out.printf("  %-13s 最擠的桶 = %3d 個鍵，空桶 = %3d / %d%n", name, maxBucket, empty,
     capacity);
64      }
65
66      public static void main(String[] args) {
67          System.out.println("=== 1. 雜湊函數必須「尊重等價」：equals 的兩個字串雜湊值相同 ===");
68          String a = "hello";
69          String b = new String("hello");             // 不同物件、相同內容
70          assert a != b && a.equals(b);
71          assert goodHash(a) == goodHash(b);
72          assert a.hashCode() == b.hashCode();
73          System.out.println("  goodHash(\"hello\") = " + goodHash(a) + "，String.hashCode() = " + a
    .hashCode());
74          assert goodHash(a) == a.hashCode() : "我們的實作應與 String.hashCode 一致";
75
76          System.out.println("=== 2. 位置很重要 ===");
77          System.out.println("  goodHash(ab) = " + goodHash("ab") + "，goodHash(ba) = " + goodHash("
    ba"));
78          System.out.println("  mediocreHash(ab) = " + mediocreHash("ab") + "，mediocreHash(ba) = "
    + mediocreHash("ba") + "（碰撞！）");
79
80          System.out.println("=== 3. 溢位是正常現象：值仍然是確定性的 ===");
81          String longStr = "The quick brown fox jumps over the lazy dog";
82          System.out.println("  goodHash(長句) = " + goodHash(longStr) + "（可能是負數）");
83          System.out.println("  桶索引（容量 16）= " + bucketOf(goodHash(longStr), 16));
84
85          System.out.println("=== 4. 分布品質：1000 個鍵放進 128 個桶 ===");
86          String[] words = new String[1000];
```

```
        for (int i = 0; i < words.length; i++) {
            words[i] = "user" + i;                    // 很規律的鍵：對雜湊函數是嚴苛的考驗
        }
        report("goodHash", StringHashing::goodHash, words, 128);
        report("mediocreHash", StringHashing::mediocreHash, words, 128);
        report("badHash", StringHashing::badHash, words, 128);
        System.out.println("   （理想狀況：1000/128 ≈ 8 個鍵一桶）");
        System.out.println("全部通過。");
    }
}
```

07_hashing/ChainingHashMap.java

```
/**
 * 手刻的雜湊表（hash table）：用「鏈結」（chaining，也叫 separate chaining）解決碰撞。
 *
 * 結構：一個桶陣列（buckets），每個桶是一條「鏈結串列」，串著所有雜湊到該桶的 (key, value)。
 *
 * 操作平均 O(1) 的前提：
 *   1. 雜湊函數分布均勻；
 *   2. 負載因子 size / buckets.length 不要太大——超過 0.75 就把桶陣列倍增並「重新雜湊」（rehash）。
 *
 * 資料不變量：
 *   - 每個鍵最多出現在一條鏈上、且出現一次；
 *   - 鍵 k 一定位於 buckets[indexFor(k)] 這條鏈上；
 *   - size 等於所有鍵的節點總數。
 */
public class ChainingHashMap<K, V> {

    /** 鏈結串列的節點：一個 (key, value) 對，加上指向同桶下一個節點的參照。 */
    private static class Node<K, V> {
        final K key;
        V value;
        Node<K, V> next;

        Node(K key, V value, Node<K, V> next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    private static final double MAX_LOAD_FACTOR = 0.75;

    private Node<K, V>[] buckets;
```

```
    private int size = 0;

    @SuppressWarnings({"unchecked", "rawtypes"})   // 泛型陣列不能直接 new，只能 new Node[..] 再轉型
    public ChainingHashMap(int initialCapacity) {
        this.buckets = (Node<K, V>[]) new Node[Math.max(1, initialCapacity)];
    }

    public ChainingHashMap() {
        this(8);
    }

    /** 鍵 → 桶索引：用 floorMod 確保結果在 0..length-1（hashCode 可能是負數）。 */
    private int indexFor(K key, int length) {
        return Math.floorMod(key.hashCode(), length);
    }

    /** 在鍵所屬的鏈上找節點；找不到回傳 null。 */
    private Node<K, V> findNode(K key) {
        Node<K, V> n = this.buckets[indexFor(key, this.buckets.length)];
        while (n != null) {
            if (n.key.equals(key)) {      // 先用 hashCode 找桶，再用 equals 確認：兩者必須一致！
                return n;
            }
            n = n.next;
        }
        return null;
    }

    /** 新增或覆蓋。@return 舊值，若原本沒有則回傳 null */
    public V put(K key, V value) {
        if (key == null) {
            throw new IllegalArgumentException("鍵不可為 null");
        }
        Node<K, V> existing = findNode(key);
        if (existing != null) {
            V old = existing.value;
            existing.value = value;
            return old;
        }
        int idx = indexFor(key, this.buckets.length);
        this.buckets[idx] = new Node<>(key, value, this.buckets[idx]);   // 插在鏈的最前面：O(1)
        this.size = this.size + 1;
        if ((double) this.size / this.buckets.length > MAX_LOAD_FACTOR) {
            resize();
        }
        return null;
    }
```

```
    public V get(K key) {
        Node<K, V> n = findNode(key);
        return n == null ? null : n.value;
    }

    public boolean containsKey(K key) {
        return findNode(key) != null;
    }

    /** 移除鍵。@return 被移除的值，若不存在則回傳 null */
    public V remove(K key) {
        int idx = indexFor(key, this.buckets.length);
        Node<K, V> prev = null;
        Node<K, V> n = this.buckets[idx];
        while (n != null) {
            if (n.key.equals(key)) {
                if (prev == null) {
                    this.buckets[idx] = n.next;   // 移除的是鏈的第一個節點
                } else {
                    prev.next = n.next;           // 跳過 n
                }
                this.size = this.size - 1;
                return n.value;
            }
            prev = n;
            n = n.next;
        }
        return null;
    }

    public int size() {
        return this.size;
    }

    public int capacity() {
        return this.buckets.length;
    }

    /** 桶陣列倍增，並把每個節點「重新雜湊」到新位置（索引會變，因為 length 變了）。 */
    @SuppressWarnings({"unchecked", "rawtypes"})
    private void resize() {
        Node<K, V>[] old = this.buckets;
        this.buckets = (Node<K, V>[]) new Node[old.length * 2];
        for (Node<K, V> n : old) {
            while (n != null) {
                Node<K, V> next = n.next;
                int idx = indexFor(n.key, this.buckets.length);
```

```
                n.next = this.buckets[idx];
                this.buckets[idx] = n;
                n = next;
            }
        }
    }

    /** 最長的鏈有幾個節點（用來觀察碰撞情況）。 */
    public int longestChain() {
        int best = 0;
        for (Node<K, V> n : this.buckets) {
            int len = 0;
            while (n != null) {
                len = len + 1;
                n = n.next;
            }
            best = Math.max(best, len);
        }
        return best;
    }

    /** 每個桶依序印出：方便看見「桶」與「鏈」長什麼樣子。 */
    public void dump() {
        for (int i = 0; i < this.buckets.length; i++) {
            StringBuilder sb = new StringBuilder("    [" + i + "]");
            Node<K, V> n = this.buckets[i];
            while (n != null) {
                sb.append(" → (").append(n.key).append(", ").append(n.value).append(")");
                n = n.next;
            }
            System.out.println(sb);
        }
    }
}
```

07_hashing/LinearProbingIntSet.java

```
/**
 * 用「線性探測」（linear probing，一種開放定址法 open addressing）實作的整數集合。
 *
 * 不用鏈結串列：所有元素直接放在桶陣列裡。
 * 碰撞時就往下一格找（到尾端繞回開頭），直到找到空格為止。
 *
 * 優點：記憶體連續、快取友善。缺點：容易形成「群聚」（clustering）——
 * 一長串被占用的格子會讓插入與查詢都變慢；因此負載因子要壓得更低（這裡 0.5）。
 *
```

```
 * 為了簡化，本類別只支援 add / contains（刪除需要「墓碑」標記，見教材說明）。
 *
 * 資料不變量：若 x 在集合中，則從 indexFor(x) 開始往後（繞回）掃到 x 之前，不會遇到空格。
 */
public class LinearProbingIntSet {

    private static final int EMPTY = Integer.MIN_VALUE;   // 哨兵值：代表「這格是空的」
    private int[] table;
    private int size = 0;
    public int probes = 0;                                // 統計探測次數（教學用）

    public LinearProbingIntSet(int capacity) {
        this.table = new int[Math.max(2, capacity)];
        java.util.Arrays.fill(this.table, EMPTY);
    }

    private int indexFor(int x, int length) {
        return Math.floorMod(Integer.hashCode(x) * 0x9E3779B1, length);   // 乘一個大奇數「攪亂」一下
    }

    public boolean add(int x) {
        if (x == EMPTY) {
            throw new IllegalArgumentException("此實作不能存放 Integer.MIN_VALUE（被當成哨兵值）");
        }
        if (contains(x)) {
            return false;
        }
        if ((this.size + 1) * 2 > this.table.length) {   // 負載因子超過 0.5：倍增並重新插入
            resize();
        }
        int i = indexFor(x, this.table.length);
        while (this.table[i] != EMPTY) {                 // 線性探測：往下一格找
            this.probes = this.probes + 1;
            i = (i + 1) % this.table.length;             // 到尾端就繞回開頭
        }
        this.table[i] = x;
        this.size = this.size + 1;
        return true;
    }

    public boolean contains(int x) {
        int i = indexFor(x, this.table.length);
        while (this.table[i] != EMPTY) {                 // 遇到空格就可以確定不存在
            this.probes = this.probes + 1;
            if (this.table[i] == x) {
                return true;
            }
```

```
            i = (i + 1) % this.table.length;
        }
        return false;
    }

    public int size() {
        return this.size;
    }

    private void resize() {
        int[] old = this.table;
        this.table = new int[old.length * 2];
        java.util.Arrays.fill(this.table, EMPTY);
        this.size = 0;
        for (int x : old) {
            if (x != EMPTY) {
                add(x);                                  // 重新插入（索引會變）
            }
        }
    }

    /** 印出桶陣列：'.' 代表空格。 */
    public void dump() {
        StringBuilder sb = new StringBuilder("    ");
        for (int x : this.table) {
            sb.append(x == EMPTY ? "." : String.valueOf(x)).append(' ');
        }
        System.out.println(sb);
    }
}
```

07_hashing/HashDemo.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.Set;

/**
 * 07_hashing 示範程式：手刻雜湊表、線性探測、HashMap/HashSet、equals/hashCode 契約、可變鍵的災難。
 *
 * 編譯與執行：
 *   javac *.java
```

```
 *   java -ea HashDemo
 */
public class HashDemo {

    /** 一個「值型」的類別：兩個座標相同就該視為同一個鍵。 */
    @SuppressWarnings("overrides")   // 故意示範錯誤：javac 其實會警告「覆寫了 equals 卻沒覆寫 hashCode」
    static class PointNoHash {
        final int x, y;
        PointNoHash(int x, int y) { this.x = x; this.y = y; }

        @Override
        public boolean equals(Object o) {          // 只覆寫 equals、忘了 hashCode → 契約被破壞
            return o instanceof PointNoHash p && p.x == x && p.y == y;
        }
    }

    /** 正確版：equals 與 hashCode 一起覆寫，且用同一組欄位。 */
    static class Point {
        final int x, y;
        Point(int x, int y) { this.x = x; this.y = y; }

        @Override
        public boolean equals(Object o) {
            return o instanceof Point p && p.x == x && p.y == y;
        }

        @Override
        public int hashCode() {
            return Objects.hash(x, y);             // 標準寫法：把所有參與 equals 的欄位丟進去
        }

        @Override
        public String toString() { return "(" + x + "," + y + ")"; }
    }

    /** record：自動產生「一致」的 equals 與 hashCode，是最安全的鍵。 */
    record Coord(int x, int y) {}

    /** 用手刻雜湊表找出出現最多次的單字。 */
    static String mostCommonWord(String[] words) {
        ChainingHashMap<String, Integer> counts = new ChainingHashMap<>();
        for (String w : words) {
            Integer c = counts.get(w);
            counts.put(w, c == null ? 1 : c + 1);
        }
        String best = null;
        int bestCount = -1;
```

```
        for (String w : words) {
            int c = counts.get(w);
            if (c > bestCount) {
                best = w;
                bestCount = c;
            }
        }
        return best;
    }

    public static void main(String[] args) {
        System.out.println("=== 1. 手刻的鏈結雜湊表 ===");
        ChainingHashMap<String, Integer> ages = new ChainingHashMap<>(4);
        ages.put("Liam", 30);
        ages.put("Thomas", 25);
        ages.put("Alice", 28);
        System.out.println("  容量 " + ages.capacity() + "，size " + ages.size() + "：");
        ages.dump();
        ages.put("Bob", 41);                                    // 負載 4/4 > 0.75 → 觸發倍增
        System.out.println("  put(Bob) 後容量 " + ages.capacity() + "（倍增＋重新雜湊）：");
        ages.dump();
        assert ages.get("Liam") == 30 && ages.get("Nobody") == null;
        assert ages.put("Liam", 31) == 30 && ages.get("Liam") == 31;   // 覆蓋回傳舊值
        assert ages.remove("Thomas") == 25 && !ages.containsKey("Thomas");
        assert ages.size() == 3;

        System.out.println("=== 2. 大量資料：鏈保持很短 ===");
        ChainingHashMap<String, Integer> big = new ChainingHashMap<>();
        for (int i = 0; i < 100_000; i++) {
            big.put("key" + i, i);                              // 十萬個字串鍵
        }
        for (int i = 0; i < 100_000; i++) {
            assert big.get("key" + i) == i;
        }
        System.out.println("  100000 個鍵，容量 " + big.capacity() + "，最長的鏈 = " + big.
    longestChain());

        String[] words = "the cat and the dog and the bird".split(" ");
        System.out.println("  最常見的單字 = " + mostCommonWord(words));
        assert mostCommonWord(words).equals("the");

        System.out.println("=== 3. 線性探測 ===");
        LinearProbingIntSet set = new LinearProbingIntSet(8);
        for (int x : new int[]{5, 13, 21, 7, 29}) {
            set.add(x);
        }
        set.dump();
        assert set.contains(13) && !set.contains(99) && set.size() == 5;
```

```
        set.probes = 0;
        for (int i = 0; i < 1000; i++) {
            set.add(i * 37);
        }
        System.out.println("  加入 1000 個元素共探測 " + set.probes + " 次（平均每次 "
                + String.format("%.2f", set.probes / 1000.0) + " 次額外探測）");

        System.out.println("=== 4. 標準函式庫：HashMap 與 HashSet ===");
        Map<String, Integer> wordCount = new HashMap<>();
        for (String w : words) {
            wordCount.merge(w, 1, Integer::sum);              // 有就 +1，沒有就放 1
        }
        System.out.println("  wordCount = " + wordCount);
        Set<Integer> seen = new HashSet<>();
        int[] data = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
        List<Integer> dups = new ArrayList<>();
        for (int d : data) {
            if (!seen.add(d)) {                               // add 回傳 false 表示已存在
                dups.add(d);
            }
        }
        System.out.println("  重複出現的數字 = " + dups);

        System.out.println("=== 5. equals/hashCode 契約 ===");
        Set<PointNoHash> broken = new HashSet<>();
        broken.add(new PointNoHash(1, 2));
        System.out.println("  只覆寫 equals：contains(new PointNoHash(1,2)) = "
                + broken.contains(new PointNoHash(1, 2)) + "  ← 找不到！");
        Set<Point> good = new HashSet<>();
        good.add(new Point(1, 2));
        System.out.println("  equals+hashCode：contains(new Point(1,2)) = " + good.contains(new
    Point(1, 2)));
        Set<Coord> rec = new HashSet<>();
        rec.add(new Coord(1, 2));
        System.out.println("  record：contains(new Coord(1,2)) = " + rec.contains(new Coord(1, 2))
    );
        assert !broken.contains(new PointNoHash(1, 2)) && good.contains(new Point(1, 2)) && rec.
    contains(new Coord(1, 2));

        System.out.println("=== 6. 可變的鍵是災難 ===");
        List<String> key = new ArrayList<>(List.of("a"));
        Map<List<String>, Integer> map = new HashMap<>();
        map.put(key, 42);
        System.out.println("  put 之後 get(key) = " + map.get(key));
        key.add("b");                                         // 改變了鍵 → hashCode 變了
        System.out.println("  key.add(\"b\") 之後 get(key) = " + map.get(key)
                + "，size = " + map.size() + "  ← 值還在表裡，卻永遠找不到");
        assert map.get(key) == null && map.size() == 1;
```

```
        System.out.println("全部通過。");
    }
}
```

A.8 08_chat_protocol：聊天協定與客戶端

08_chat_protocol/ChatMessage.java

```
/**
 * 聊天協定的「訊息」ADT：用 sealed interface + record 把每一種合法訊息列舉出來。
 *
 * 協定草案（一行一則訊息，欄位以單一空白分隔，最後一個欄位可含空白）：
 *
 *   客戶端 → 伺服器                    伺服器 → 客戶端
 *   ---------------------------------  ---------------------------------------
 *   LOGIN <name>                        OK [<text>]
 *   JOIN <room>                         ERR <code> <text>
 *   SAY <room> <text...>                MSG <room> <name> <text...>
 *   QUIT                                BYE
 *
 * 這個類別把「文字 ↔ 結構化資料」的轉換集中在一個地方（parse / encode），
 * 伺服器與客戶端的其他程式碼就完全不需要碰字串切割。
 */
public sealed interface ChatMessage
        permits ChatMessage.Login, ChatMessage.Join, ChatMessage.Say, ChatMessage.Quit,
                ChatMessage.Ok, ChatMessage.Err, ChatMessage.Msg, ChatMessage.Bye {

    // ---- 客戶端 → 伺服器 ----
    record Login(String name) implements ChatMessage {}
    record Join(String room) implements ChatMessage {}
    record Say(String room, String text) implements ChatMessage {}
    record Quit() implements ChatMessage {}

    // ---- 伺服器 → 客戶端 ----
    record Ok(String text) implements ChatMessage {}
    record Err(int code, String text) implements ChatMessage {}
    record Msg(String room, String name, String text) implements ChatMessage {}
    record Bye() implements ChatMessage {}

    /** 協定錯誤：收到不符合規格的一行。 */
    class ProtocolException extends RuntimeException {
        private static final long serialVersionUID = 1L;

        public ProtocolException(String message) {
            super(message);
        }
```

```
    }

    /**
     * 把一行文字解析成 ChatMessage。
     * @throws ProtocolException 若格式不合法
     */
    static ChatMessage parse(String line) {
        if (line == null) {
            throw new ProtocolException("連線已關閉（讀到 null）");
        }
        String[] parts = line.split(" ", 2);            // 先切出指令字
        String command = parts[0].toUpperCase();
        String rest = parts.length > 1 ? parts[1] : "";
        switch (command) {
            case "LOGIN": {
                requireNonEmpty(rest, "LOGIN 需要名字");
                requireNoSpace(rest, "名字不可含空白");
                return new Login(rest);
            }
            case "JOIN": {
                requireNonEmpty(rest, "JOIN 需要房間名");
                requireNoSpace(rest, "房間名不可含空白");
                return new Join(rest);
            }
            case "SAY": {
                String[] p = rest.split(" ", 2);         // <room> <text...>
                if (p.length < 2 || p[0].isEmpty()) {
                    throw new ProtocolException("SAY 需要房間名與內容");
                }
                return new Say(p[0], p[1]);
            }
            case "QUIT": return new Quit();
            case "OK":   return new Ok(rest);
            case "ERR": {
                String[] p = rest.split(" ", 2);
                if (p.length < 2) {
                    throw new ProtocolException("ERR 需要錯誤碼與說明");
                }
                try {
                    return new Err(Integer.parseInt(p[0]), p[1]);
                } catch (NumberFormatException e) {
                    throw new ProtocolException("ERR 的錯誤碼必須是整數：" + p[0]);
                }
            }
            case "MSG": {
                String[] p = rest.split(" ", 3);         // <room> <name> <text...>
                if (p.length < 3) {
                    throw new ProtocolException("MSG 需要房間、名字與內容");
```

```
                }
                return new Msg(p[0], p[1], p[2]);
            }
            case "BYE":  return new Bye();
            default:
                throw new ProtocolException("未知的指令：" + parts[0]);
        }
    }

    /** 把 ChatMessage 編碼回協定文字（不含換行）。parse(encode(m)) 必須等於 m。 */
    static String encode(ChatMessage m) {
        // switch 模式比對（Java 21）：sealed 讓編譯器確認我們沒有漏掉任何一種訊息
        return switch (m) {
            case Login l -> "LOGIN " + l.name();
            case Join j  -> "JOIN " + j.room();
            case Say s   -> "SAY " + s.room() + " " + s.text();
            case Quit q  -> "QUIT";
            case Ok o    -> o.text().isEmpty() ? "OK" : "OK " + o.text();
            case Err e   -> "ERR " + e.code() + " " + e.text();
            case Msg g   -> "MSG " + g.room() + " " + g.name() + " " + g.text();
            case Bye b   -> "BYE";
        };
    }

    private static void requireNonEmpty(String s, String why) {
        if (s.isEmpty()) {
            throw new ProtocolException(why);
        }
    }

    private static void requireNoSpace(String s, String why) {
        if (s.contains(" ")) {
            throw new ProtocolException(why);
        }
    }
}
```

08_chat_protocol/ChatClient.java

```
/**
 * Week 7 Workshop 主題：聊天「客戶端程式庫」的介面設計。
 *
 * 專案目標是「任何人的 UI 都能接上任何人的客戶端程式庫」，
 * 所以 UI 只能依賴這個介面，絕對不能依賴某個特定實作類別。
 *
 * 設計重點：
```

```
 *   – 方法對應「使用者能做的事」（登入、加入房間、發言、離開），而不是協定細節。
 *   – 伺服器主動推來的事件（別人的訊息、錯誤、斷線）用「監聽者」（listener）回呼，
 *     因為 UI 不可能知道訊息何時會到。
 */
public interface ChatClient {

    /** 事件監聯者：UI 實作這個介面，客戶端程式庫在事件發生時呼叫它。 */
    interface Listener {
        void onMessage(String room, String from, String text);
        void onError(int code, String text);
        void onDisconnected();
    }

    /** 註冊監聽者（可註冊多個）。 */
    void addListener(Listener listener);

    /**
     * 連線並登入。
     * @throws java.io.IOException 網路失敗時
     * @throws IllegalStateException 若已經連線
     */
    void connect(String host, int port, String name) throws java.io.IOException;

    void join(String room);

    /** 前置條件：已連線且已 join 該房間。 */
    void say(String room, String text);

    /** 告知伺服器離開並關閉連線；之後任何操作都是 IllegalStateException。 */
    void disconnect();

    boolean isConnected();
}
```

08_chat_protocol/FakeChatClient.java

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

/**
 * 「假的」客戶端：完全不碰網路，把所有訊息直接回送給監聽者。
 *
 * 用途：在伺服器還沒寫好之前，就能開發並「測試」UI——
 * 這正是「對介面寫程式」帶來的好處：任何 ChatClient 的實作都可以互換。
 */
public class FakeChatClient implements ChatClient {
```

```
    private final List<Listener> listeners = new ArrayList<>();
    private boolean connected = false;
    private String name;
    private final List<String> rooms = new ArrayList<>();

    @Override
    public void addListener(Listener listener) {
        this.listeners.add(listener);
    }

    @Override
    public void connect(String host, int port, String name) {
        if (this.connected) {
            throw new IllegalStateException("已經連線");
        }
        this.connected = true;
        this.name = name;
    }

    @Override
    public void join(String room) {
        requireConnected();
        this.rooms.add(room);
    }

    @Override
    public void say(String room, String text) {
        requireConnected();
        if (!this.rooms.contains(room)) {
            for (Listener l : this.listeners) {
                l.onError(403, "not in room " + room);   // 模擬伺服器回 ERR
            }
            return;
        }
        for (Listener l : this.listeners) {              // 模擬伺服器把訊息廣播回來
            l.onMessage(room, this.name, text);
        }
    }

    @Override
    public void disconnect() {
        requireConnected();
        this.connected = false;
        for (Listener l : this.listeners) {
            l.onDisconnected();
        }
    }
```

```

    @Override
    public boolean isConnected() {
        return this.connected;
    }

    private void requireConnected() {
        if (!this.connected) {
            throw new IllegalStateException("尚未連線");
        }
    }
}
```

08_chat_protocol/EchoServer.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 最小的「一行一則訊息」伺服器：示範 Java 網路程式的基本骨架。
 *
 * 每收到一行，就用 ChatMessage.parse 解析；合法就回 OK（或 MSG 回送），不合法就回 ERR。
 * 收到 QUIT 就回 BYE 並關閉該連線。
 *
 * 執行：java EchoServer 5555
 * 另開一個終端機：java EchoClient localhost 5555    （或 nc localhost 5555 手動輸入）
 *
 * 注意：這個版本一次只服務一位客戶（單執行緒）。
 * 要同時服務多位客戶需要「多執行緒」，那是 Week 8 Workshop 的主題。
 */
public class EchoServer {

    public static void main(String[] args) throws IOException {
        int port = args.length > 0 ? Integer.parseInt(args[0]) : 5555;
        try (ServerSocket server = new ServerSocket(port)) {          // 在 port 上「聽」
            System.out.println("伺服器啟動，監聽 port " + port);
            while (true) {
                try (Socket client = server.accept();                 // 阻塞直到有人連進來
                     BufferedReader in = new BufferedReader(
                             new InputStreamReader(client.getInputStream(), "UTF-8"));
                     PrintWriter out = new PrintWriter(client.getOutputStream(), true, java.nio.
    charset.StandardCharsets.UTF_8)) {
```

```
                    System.out.println("客戶端已連線：" + client.getRemoteSocketAddress());
                    handle(in, out);
                    System.out.println("客戶端已離線");
                }
            }
        }
    }

    /** 對「一個」客戶端的完整對話：一直讀到 QUIT 或連線中斷。 */
    static void handle(BufferedReader in, PrintWriter out) throws IOException {
        String name = null;
        String line;
        while ((line = in.readLine()) != null) {                    // readLine 回傳 null = 對方關閉了連線
            ChatMessage reply;
            try {
                ChatMessage m = ChatMessage.parse(line);
                switch (m) {
                    case ChatMessage.Login l -> {
                        name = l.name();
                        reply = new ChatMessage.Ok("welcome " + name);
                    }
                    case ChatMessage.Join j -> reply = new ChatMessage.Ok("joined " + j.room());
                    case ChatMessage.Say s -> reply = name == null
                            ? new ChatMessage.Err(401, "login first")
                            : new ChatMessage.Msg(s.room(), name, s.text());
                    case ChatMessage.Quit q -> {
                        out.println(ChatMessage.encode(new ChatMessage.Bye()));
                        return;
                    }
                    default -> reply = new ChatMessage.Err(400, "clients may not send " + line.split(" ")[0]);
                }
            } catch (ChatMessage.ProtocolException e) {
                reply = new ChatMessage.Err(400, e.getMessage());
            }
            out.println(ChatMessage.encode(reply));                 // println 會加上換行並 flush
        }
    }
}
```

08_chat_protocol/EchoClient.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

/**
 * 最小的客戶端：連上伺服器，送出一段固定的對話，印出伺服器的每一則回應。
 *
 * 執行（先啟動 EchoServer）：java EchoClient localhost 5555
 */
public class EchoClient {

    public static void main(String[] args) throws IOException {
        String host = args.length > 0 ? args[0] : "localhost";
        int port = args.length > 1 ? Integer.parseInt(args[1]) : 5555;

        try (Socket socket = new Socket(host, port);                // 主動連到伺服器
             BufferedReader in = new BufferedReader(
                     new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8));
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true, StandardCharsets.
    UTF_8)) {

            String[] script = {
                "SAY general hi",           // 還沒登入 → ERR 401
                "LOGIN angelia",            // → OK welcome angelia
                "JOIN general",             // → OK joined general
                "SAY general hello world",  // → MSG general angelia hello world
                "DANCE",                    // 未知指令 → ERR 400
                "QUIT"                      // → BYE
            };
            for (String line : script) {
                System.out.println(">>> " + line);
                out.println(line);                                  // 送出一行
                String reply = in.readLine();                       // 等一行回應
                System.out.println("<<< " + reply);
                ChatMessage parsed = ChatMessage.parse(reply);      // 客戶端也用同一份 parser
                System.out.println("    解析結果：" + parsed);
            }
        }
    }
}
```

08_chat_protocol/ProtocolTest.java

```
/**
 * 協定解析器的測試（用 assert；也可以改寫成 JUnit）。
 *
```

```
 * 執行：javac *.java && java -ea ProtocolTest
 */
public class ProtocolTest {

    /** 「往返」性質：任何合法訊息 encode 之後再 parse，必須得到相等的物件。 */
    static void roundTrip(ChatMessage m) {
        String line = ChatMessage.encode(m);
        ChatMessage back = ChatMessage.parse(line);
        assert back.equals(m) : "往返失敗：" + m + " → \"" + line + "\" → " + back;
        System.out.println("  " + String.format("%-40s", line) + " ⇄ " + back);
    }

    static void expectError(String line) {
        try {
            ChatMessage m = ChatMessage.parse(line);
            assert false : "應該要失敗卻解析成功：" + line + " → " + m;
        } catch (ChatMessage.ProtocolException e) {
            System.out.println("  \"" + line + "\" → 拒絕：" + e.getMessage());
        }
    }

    public static void main(String[] args) throws java.io.IOException {
        System.out.println("=== 往返測試（record 自動有 equals，才能這樣比較）===");
        roundTrip(new ChatMessage.Login("angelia"));
        roundTrip(new ChatMessage.Join("general"));
        roundTrip(new ChatMessage.Say("general", "hello world, with spaces!"));
        roundTrip(new ChatMessage.Quit());
        roundTrip(new ChatMessage.Ok(""));
        roundTrip(new ChatMessage.Ok("welcome angelia"));
        roundTrip(new ChatMessage.Err(401, "login first"));
        roundTrip(new ChatMessage.Msg("general", "bob", "多位元組字元也沒問題：café 日本語"));
        roundTrip(new ChatMessage.Bye());

        System.out.println("=== 不合法的輸入必須被拒絕，而不是默默接受 ===");
        expectError("");
        expectError("DANCE");
        expectError("LOGIN");
        expectError("LOGIN two words");
        expectError("SAY general");
        expectError("ERR notanumber oops");
        expectError("MSG general bob");

        System.out.println("=== 指令不分大小寫 ===");
        assert ChatMessage.parse("login angelia").equals(new ChatMessage.Login("angelia"));

        System.out.println("=== 用 FakeChatClient 測 UI 邏輯（不需要網路）===");
        StringBuilder screen = new StringBuilder();
        ChatClient client = new FakeChatClient();
```

```
        client.addListener(new ChatClient.Listener() {          // 匿名類別實作監聽者介面
            public void onMessage(String room, String from, String text) {
                screen.append("[").append(room).append("] ").append(from).append(": ").append(text
    ).append("\n");
            }
            public void onError(int code, String text) {
                screen.append("ERROR ").append(code).append(": ").append(text).append("\n");
            }
            public void onDisconnected() {
                screen.append("(disconnected)\n");
            }
        });
        client.connect("localhost", 5555, "angelia");
        client.say("general", "hi");                            // 還沒 join → 錯誤
        client.join("general");
        client.say("general", "hi again");
        client.disconnect();
        System.out.print(screen);
        assert screen.toString().equals(
                "ERROR 403: not in room general\n[general] angelia: hi again\n(disconnected)\n");
        try {
            client.say("general", "after disconnect");
        } catch (IllegalStateException e) {
            System.out.println("  斷線後操作 → IllegalStateException：" + e.getMessage());
        }
        System.out.println("全部通過。");
    }
}
```