基礎程式設計（Java）　講義

遞迴：方法呼叫自己

Deitel & Deitel《Java How to Program: Early Objects》11e, Ch.18

§ 1　一、名詞速查表

中文	English	一句話定義	科目	章節
遞迴	recursion	方法在自己的方法體裡直接或間接呼叫自己	Java	CH18
遞迴方法	recursive method	會呼叫自己以解決問題的方法	Java	CH18
基底條件	base case	不需再遞迴、能直接給出答案的最簡單情況；遞迴的終止點	Java	CH18
遞迴步驟	recursive step	把問題拆成「更小的同類問題」並呼叫自己的那一步	Java	CH18
收斂	convergence	每次遞迴都讓問題更接近基底條件，最終會停	Java	CH18
呼叫堆疊	call stack	記錄方法呼叫順序的堆疊，後呼叫的先返回（LIFO）	Java	CH18
活動記錄	activation record	堆疊框；存某次方法呼叫的參數與區域變數	Java	CH18
展開	unwinding	遞迴呼叫一路 push 到基底條件的下行過程	Java	CH18
回溯（回繞）	returning / unwind	從基底條件一路 return、pop 框、把結果往回算的上行過程	Java	CH18
無限遞迴	infinite recursion	沒有基底條件或永遠到不了，遞迴呼叫停不下來	Java	CH18
堆疊溢位	StackOverflowError	呼叫層疊太深、堆疊框超過記憶體上限時丟出的錯誤	Java	CH18
迭代	iteration	用迴圈（`for`／`while`）重複，而非呼叫自己	Java	CH18
階乘	factorial	`n! = n × (n-1) × … × 1`，`0! = 1`；典型遞迴例	Java	CH18
費氏數列	Fibonacci series	`0, 1, 1, 2, 3, 5…`，每項為前兩項之和	Java	CH18
河內塔	Towers of Hanoi	三柱搬碟謎題，遞迴解法的經典示範	Java	CH18

§ 2　二、核心概念

遞迴是「方法呼叫自己」來解決問題的技巧。把一個大問題，化成一個或多個**規模更小、但形式相同**的子問題；當問題小到不能再小（**基底條件**），就直接給答案、不再呼叫自己。

> 一個遞迴方法只直接解決**最簡單的情況（基底條件）**。被要求解較複雜的問題時，它把問題拆成兩塊：**自己能立刻處理的部分**，和**一塊跟原問題同形式、但更小的部分**。因為這塊仍像原問題，方法就**呼叫自己一份新的**去處理它（**遞迴呼叫／遞迴步驟**）。每次遞迴都要讓問題更接近基底條件（**收斂**），否則永遠停不下來。

每個遞迴方法都必須具備兩件事，缺一不可：

1. **基底條件（base case）**：最簡單、能直接回答的情況，作為終止點。例如 `factorial(0)` 與 `factorial(1)` 都回 `1`，不再呼叫自己。
2. **遞迴步驟（recursive step）**：把問題縮小一級後呼叫自己，並用回來的結果組出答案。例如 `factorial(n)` 回 `n * factorial(n - 1)`。

> **與 CH6 的關係**：第 6 章學方法呼叫時，已知每次呼叫都會在**呼叫堆疊**頂端 push 一個**活動記錄**、`return` 後 pop。遞迴只是「呼叫的對象剛好是自己」，堆疊機制完全一樣——只是同一個方法的多個活動記錄會同時疊在堆疊上，每個框各有自己的參數值。

若漏了基底條件、或遞迴步驟沒有收斂到基底條件，就會無限遞迴：堆疊框越疊越高，最終超過記憶體上限，Java 丟出 `StackOverflowError`。這是遞迴最常見的錯誤。

§ 3 三、主要內容

3.1 1. 遞迴的兩個必要部位

```
public static long factorial(int n) {
    if (n <= 1) {                // ① 基底條件：n 為 0 或 1，直接回 1
        return 1;
    } else {                     // ② 遞迴步驟：縮小一級後呼叫自己
        return n * factorial(n - 1);
    }
}
```

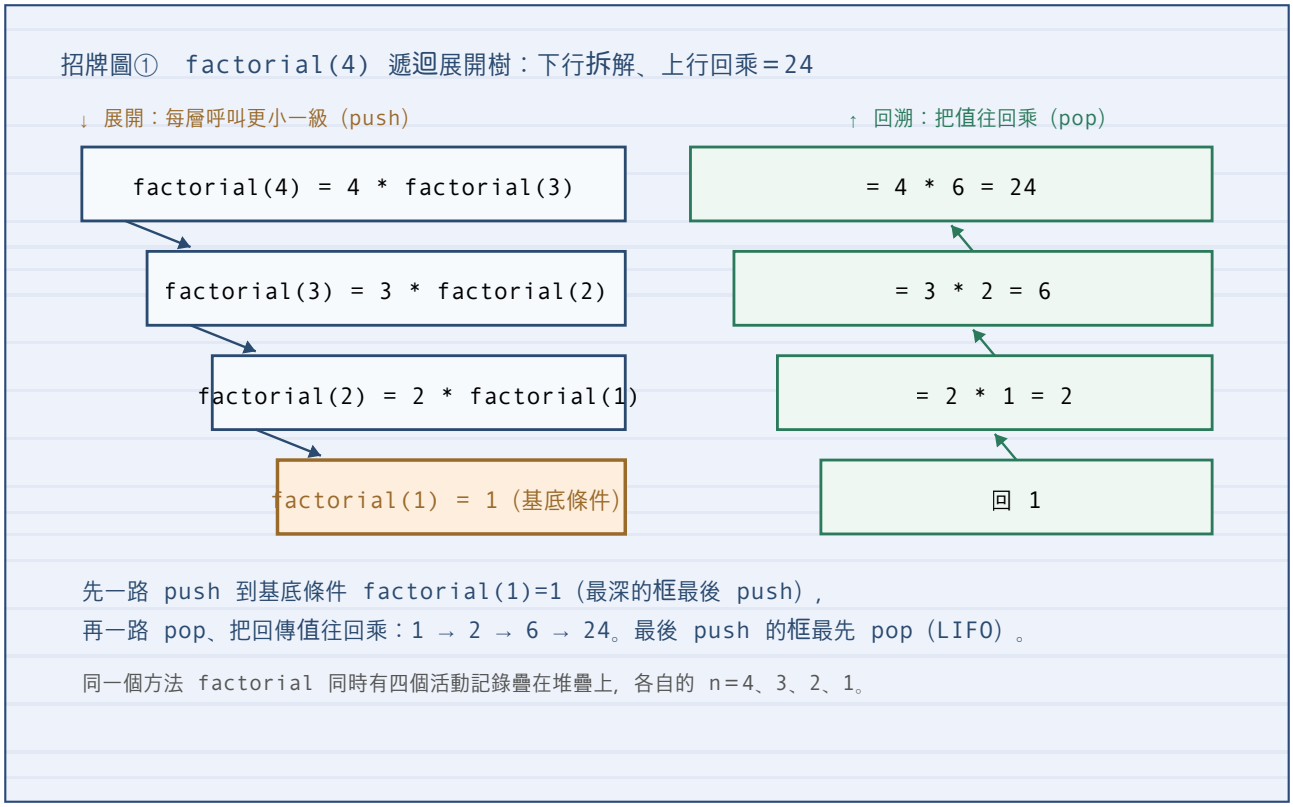
- **基底條件先寫**：先擋掉最簡單的情況再遞迴，邏輯清楚也避免漏寫。
- **遞迴步驟要收斂**：`factorial(n - 1)` 的引數比 `n` 小 1，每次呼叫都更接近基底條件 `n <= 1`，所以一定會停。
- 把 `n - 1` 寫成 `n`、或基底條件寫錯（如 `n < 0`），都會讓它到不了終止點而無限遞迴。

3.2 2. 階乘 factorial：展開與回溯

以 `factorial(4)` 為例。遞迴先一路**展開**（下行，push 框）到基底條件，再一路**回溯**（上行，pop 框、把結果往回乘）：

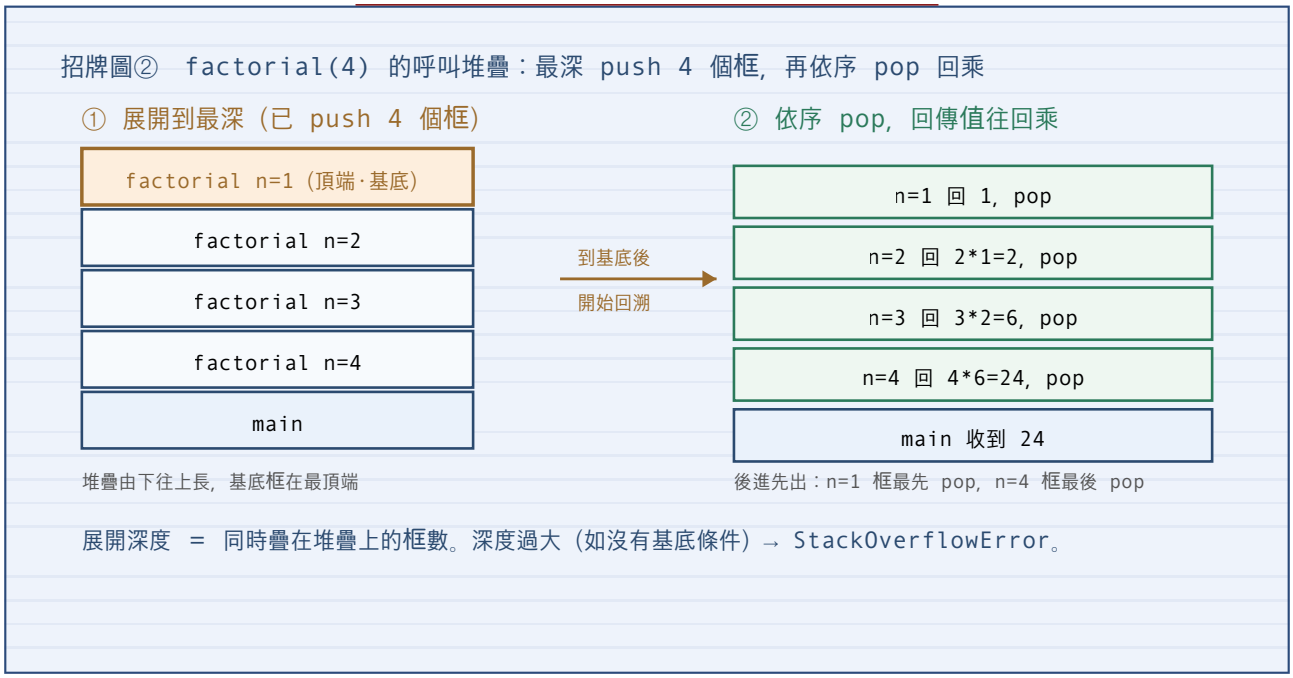
```
展開（往下呼叫，push）：
  factorial(4) = 4 * factorial(3)
  factorial(3) = 3 * factorial(2)
  factorial(2) = 2 * factorial(1)
  factorial(1) = 1                ← 基底條件，停止遞迴

回溯（往上回傳，pop，把值往回乘）：
  factorial(1) 回 1
  factorial(2) = 2 * 1  = 2
  factorial(3) = 3 * 2  = 6
  factorial(4) = 4 * 6  = 24
```



3.3 3. 遞迴在呼叫堆疊上展開與回溯

遞迴沒有任何「特殊魔法」，靠的就是 CH6 學過的**呼叫堆疊**。每次 `factorial(n)` 呼叫自己，Java 就 push 一個新的活動記錄（有自己的 `n`）；到基底條件後不再 push，開始一路 `return`、pop，把值交回上一層。



3.4 4. 費氏數列 Fibonacci

費氏數列 `0, 1, 1, 2, 3, 5, 8…`，每項是前兩項之和。它有**兩個基底條件**、且遞迴步驟呼叫自己**兩次**：

```
public static long fibonacci(int n) {
    if (n == 0 || n == 1) {      // 兩個基底條件：fib(0)=0、fib(1)=1
        return n;
    } else {                     // 遞迴步驟：呼叫自己兩次
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

- `fibonacci(n)` 同時要算 `fibonacci(n-1)` 與 `fibonacci(n-2)`，形成一棵分叉的呼叫樹。
- **效率警訊**：這種寫法會把同一個子問題重算很多次（如 `fibonacci(3)` 被算多次），呼叫次數隨 `n` **指數成長**，`n` 稍大就很慢。第 6 點會比較遞迴與迭代。

3.5 5. 河內塔 Towers of Hanoi

把 `n` 個由小到大的碟子，從來源柱搬到目標柱，借助輔助柱，規則：一次搬一片、大碟不可疊在小碟上。遞迴拆法很漂亮：

```java
// 把 n 個碟子從 from 柱搬到 to 柱，借助 aux 柱
public static void hanoi(int n, char from, char to, char aux) {
    if (n == 1) {                                  // 基底條件：只剩一片，直接搬
        System.out.println("搬 1：" + from + " → " + to);
    } else {
        hanoi(n - 1, from, aux, to);               // ① 先把上面 n-1 片搬到輔助柱
        System.out.println("搬 " + n + "：" + from + " → " + to);  // ② 搬最大片到目標
        hanoi(n - 1, aux, to, from);               // ③ 再把那 n-1 片從輔助柱搬到目標
    }
}
```

- 把「搬 n 片」化成「搬 n-1 片 + 搬 1 片 + 再搬 n-1 片」三步，每次規模減 1，最終到基底條件 `n == 1`。
- `hanoi(n)` 需要 `2^n - 1` 次搬移：`hanoi(3)` 要 7 步、`hanoi(4)` 要 15 步。

3.6 6. 遞迴 vs 迭代

兩者都靠「重複」與「終止條件」，但機制與成本不同。**任何遞迴都能改寫成迭代，反之亦然**，選哪個看可讀性與效能。

面向	遞迴（recursion）	迭代（iteration）
重複方式	方法呼叫自己	迴圈（`for`／`while`）
終止	基底條件成立	迴圈條件變 `false`
額外成本	每次呼叫 push 一個堆疊框（記憶體＋時間開銷）	不需額外堆疊框
風險	太深會 `StackOverflowError`	通常無此風險
適用	問題天然遞迴（樹、河內塔、分治）	單純累加／計數

```java
// 階乘的迭代版：同樣的事，用迴圈、不呼叫自己
public static long factorialIter(int n) {
    long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;              // 累乘
    }
    return result;
}
```

- 階乘、求和這類**線性**問題，迭代版通常更省（無呼叫開銷、不會堆疊溢位）。
- 河內塔、樹走訪這類**天然遞迴**問題，遞迴版可讀性遠勝迭代。
- 課本提醒：遞迴常被選用是因為它**更貼近問題、程式更易懂、易除錯**，代價是效能與記憶體開銷。

3.7 7. 無限遞迴與 StackOverflowError

漏掉基底條件、或遞迴步驟不收斂，遞迴永遠停不下來，堆疊框無限疊高直到溢位：

```
public static long bad(int n) {
    return n * bad(n - 1);      // 缺基底條件！n 一路變負也不停 → StackOverflowError
}
```

- 三個必檢點：**有沒有基底條件**、**遞迴步驟有沒有讓引數朝基底條件收斂**、**所有路徑都會抵達基底條件嗎**。
- `StackOverflowError` 是 `Error`（非一般 `Exception`），通常不該 catch，而是修正遞迴邏輯。
- 與 CH6 提過的呼叫堆疊上限同源：每個框佔記憶體，疊太深就爆。

§ 4 四、語法與 API 速查

```
回傳型別 方法名(參數) {
    if (基底條件) { return 直接答案; }        // base case：不再呼叫自己
    else { return 用 方法名(更小的引數) 組出答案; }   // recursive step：收斂
}
```

- **遞迴方法**：方法體裡（直接或間接）呼叫自己。
- **基底條件 base case**：最簡單、可直接回答的情況，遞迴的終止點；可有多個（費氏有兩個）。
- **遞迴步驟 recursive step**：把問題縮小一級後呼叫自己，並用回來的結果組答案；必須**收斂**到基底條件。
- **階乘**：`factorial(n) = (n<=1) ? 1 : n * factorial(n-1)`。
- **費氏**：`fib(n) = (n<=1) ? n : fib(n-1) + fib(n-2)`（兩個基底、呼叫兩次、指數成長）。
- **河內塔**：`hanoi(n-1, from, aux, to)` → 搬最大片 → `hanoi(n-1, aux, to, from)`；步數 `2^n - 1`。
- **呼叫堆疊**：每次遞迴 push 一個活動記錄，到基底後依序 pop、回傳值往回算（LIFO；同 CH6）。
- **遞迴 vs 迭代**：兩者可互換；迭代省堆疊框、遞迴貼近問題；線性問題偏迭代、天然遞迴問題偏遞迴。
- **StackOverflowError**：無限遞迴（漏基底／不收斂）導致堆疊溢位。

§ 5 五、常見錯誤

- **漏掉基底條件**：方法永遠呼叫自己，停不下來，丟 `StackOverflowError`。每個遞迴方法都要先寫終止點。

- **遞迴步驟不收斂**：呼叫自己時引數沒有更接近基底條件（如該 `n-1` 卻寫 `n`），到不了終止點。
- **基底條件寫錯範圍**：如階乘寫 `if (n == 0)` 卻可能傳入負數、或從不等於 0 的值，跳過終止點而溢位。改用 `n <= 1` 較安全。
- **以為遞迴有特殊機制**：遞迴就是普通方法呼叫自己，靠的是同一套呼叫堆疊；同一方法的多個活動記錄各有自己的參數值。
- **費氏遞迴效能誤判**：`fib(n-1) + fib(n-2)` 會重算大量子問題、呼叫次數指數成長，`n` 稍大就慢，不適合大 `n`。
- **遞迴回傳值忘了組合**：寫成 `factorial(n - 1);`（沒乘 `n`、沒 `return`）等於白算，要 `return n * factorial(n - 1);`。
- **無謂用遞迴**：單純累加／計數用迴圈即可，硬改遞迴只增加堆疊開銷與溢位風險。
- **catch 掉 StackOverflowError**：它是 `Error`、代表邏輯壞了，應修正遞迴而非吞掉。

§ 6 六、練習題

例題 1：遞迴階乘

寫 `static long factorial(int n)`，用遞迴算 `n!`（`0! = 1`）。在 `main` 印 `factorial(5)`（應為 120）。

> 1. 基底條件：`if (n <= 1) return 1;` 2. 遞迴步驟：`else return n * factorial(n - 1);` 3. `main` 印 `factorial(5)`

```
public class FactorialRec {
    public static long factorial(int n) {
        if (n <= 1) return 1;                  // 基底條件
        else return n * factorial(n - 1);      // 遞迴步驟：收斂到 n<=1
    }
    public static void main(String[] args) {
        System.out.println("factorial(5) = " + factorial(5));   // 120
    }
}
```

> 易錯：漏基底條件 → StackOverflowError；遞迴步驟忘了乘 `n` 或忘了 `return`。

例題 2：遞迴費氏數列

寫 `static long fib(int n)`，用遞迴算第 `n` 項費氏數（`fib(0)=0`、`fib(1)=1`）。在 `main` 印前 10 項（`fib(0)` 到 `fib(9)`）。

> 1. 兩個基底條件：`if (n == 0 || n == 1) return n;` 2. 遞迴步驟：`else return fib(n - 1) + fib(n - 2);` 3. `main` 用迴圈印 `fib(0)`～`fib(9)`

```
public class FibonacciRec {
    public static long fib(int n) {
        if (n == 0 || n == 1) return n;           // 兩個基底條件
        else return fib(n - 1) + fib(n - 2);      // 遞迴步驟：呼叫自己兩次
    }
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print(fib(i) + " ");       // 0 1 1 2 3 5 8 13 21 34
        }
        System.out.println();
    }
}
```

易錯：只寫一個基底條件；遞迴呼叫次數指數成長，大 `n` 會很慢（屬效能問題，非邏輯錯）。

例題 3：河內塔

寫 `static void hanoi(int n, char from, char to, char aux)`，印出把 `n` 個碟子從 `from` 搬到 `to`（借 `aux`）的每一步。在 `main` 呼叫 `hanoi(3, 'A', 'C', 'B')`。

> 1. 基底條件：`if (n == 1)` 直接印「from → to」 2. 遞迴步驟：先 `hanoi(n-1, from, aux, to)`，印搬最大片，再 `hanoi(n-1, aux, to, from)` 3. `main` 呼叫 `hanoi(3, 'A', 'C', 'B')`（共 7 步）

```
public class HanoiRec {
    public static void hanoi(int n, char from, char to, char aux) {
        if (n == 1) {
            System.out.println("搬 1：" + from + " → " + to);   // 基底條件
        } else {
            hanoi(n - 1, from, aux, to);                          // ① n-1 片移到輔助柱
            System.out.println("搬 " + n + "：" + from + " → " + to);  // ② 最大片移到目標
            hanoi(n - 1, aux, to, from);                          // ③ n-1 片移到目標
        }
    }
    public static void main(String[] args) {
        hanoi(3, 'A', 'C', 'B');     // 共 2^3 - 1 = 7 步
    }
}
```

易錯：三步的柱子順序（from／to／aux）擺錯；忘了中間印「搬最大片」那行；漏基底條件。

§ 7 七、自我檢核

- [ ] 能說清遞迴＝方法呼叫自己，並指出每個遞迴方法必備的兩個部位：基底條件與遞迴步驟。
- [ ] 知道基底條件是終止點（可有多個，如費氏的兩個），遞迴步驟必須收斂到基底條件。
- [ ] 能畫出 `factorial(4)` 的展開樹：下行拆解到 `factorial(1)=1`、上行回乘得 24。
- [ ] 能用呼叫堆疊解釋遞迴：每次呼叫 push 一個活動記錄、到基底後依序 pop（LIFO），同一方法多個框各有自己的參數。
- [ ] 會寫遞迴階乘、遞迴費氏（懂它有兩個基底、呼叫兩次、效能指數成長）。
- [ ] 能說出河內塔的三步遞迴拆法，知道 `n` 片需 `2^n - 1` 次搬移。
- [ ] 能比較遞迴與迭代的差異（重複方式、終止、堆疊開銷、風險、適用情境），知道兩者可互換。
- [ ] 知道無限遞迴（漏基底／不收斂）會丟 `StackOverflowError`，並能列出三個必檢點。