基礎程式設計（Java）　講義

多型與介面：向上轉型、動態繫結、abstract、interface

Deitel & Deitel《Java How to Program: Early Objects》11e, Ch.10

> **本章為自學延伸，非課程考試範圍。** 老師課程未教第 10 章；本講義依教科書整理，供想多學一層物件導向的人自修，不會出現在期中／期末考。建議先熟 CH9 繼承再讀本章。

§ 1　一、名詞速查表

中文	English	一句話定義	科目	章節
多型	polymorphism	同一個方法呼叫，依物件「實際型別」跑出不同版本的行為	Java	CH10
向上轉型	upcasting	用 superclass 型別的變數去指向 subclass 物件（自動、安全）	Java	CH10
動態繫結	dynamic binding / late binding	執行期才依物件實際型別決定呼叫哪個覆寫方法	Java	CH10
抽象類別	abstract class	用 `abstract` 宣告、不能 `new`，只能被繼承當共同基底	Java	CH10
抽象方法	abstract method	只有簽名沒有 body 的方法，強制子類別覆寫實作	Java	CH10
具象類別	concrete class	沒有抽象方法、可以 `new` 的一般類別	Java	CH10
final 方法	final method	加 `final` 的方法，子類別不能覆寫它	Java	CH10
final 類別	final class	加 `final` 的類別，不能被繼承（如 `String`）	Java	CH10
介面	interface	一組方法簽名的「契約」，用 `interface` 宣告、由類別 `implements`	Java	CH10
實作	implements	類別宣告「我履行某介面的契約」，須提供其所有抽象方法	Java	CH10

中文	English	一句話定義	科目	章節
多重介面	multiple interfaces	一個類別可同時 `implements` 多個介面（補繼承只能一個）	Java	CH10
向下轉型	downcasting	把 superclass 變數轉回 subclass 型別，須先 `instanceof` 確認	Java	CH10
instanceof	instanceof	判斷物件「實際上是不是某型別」的運算子，回傳 boolean	Java	CH10
default 方法	default method	介面裡帶 body 的方法（Java 8+），實作類別可不覆寫直接用	Java	CH10

§ 2 二、核心概念

CH9 學會用 `extends` 建立繼承階層、用 `@Override` 覆寫父類別方法。CH10 把繼承的真正威力解鎖：**多型（polymorphism）**——讓「同一段呼叫程式碼」對不同子類別物件，自動跑出各自的版本，不必寫一堆 `if (型別 == ...)` 分流。

> 全章最關鍵的一句話：**變數的型別決定「能呼叫哪些方法」，物件的實際型別決定「實際跑哪個版本」**。`Shape s = new Circle();` 裡 `s` 的型別是 `Shape`（所以只能呼叫 Shape 有的方法），但 `s.area()` 跑的是 `Circle` 的 `area()`——因為 heap 上那個物件實際是 Circle。這個「執行期才決定跑哪版」就是**動態繫結（dynamic binding）**。

第二條主線是**抽象（abstraction）**：有些基底概念（如「圖形 Shape」）只是用來「統一型別」，本身不該被 `new`（沒有「一個圖形」這種具體東西，只有圓、方、三角）。這種類別宣告為 `abstract`，可含**抽象方法**（只給簽名、強制子類別實作）。

第三條主線是**介面（interface）**：比抽象類別更純的契約，只規定「要會做什麼」，不管「是什麼」。Java 的類別只能 `extends` 一個父類別，卻能 `implements` 多個介面——這是 Java 表達「多重能力」的方式。

§ 3 三、主要內容

3.1 1. 多型與向上轉型：一個型別裝多種物件

繼承讓 subclass「is-a」superclass，所以 **superclass 型別的變數可以指向 subclass 物件**，這叫**向上轉型（upcasting）**，自動發生、永遠安全：

```
Shape s1 = new Circle(5);       // Circle is-a Shape，向上轉型，OK
Shape s2 = new Square(3);       // Square is-a Shape，OK
Shape[] shapes = { s1, s2 };    // 一個 Shape[] 可裝各種子類別物件
```

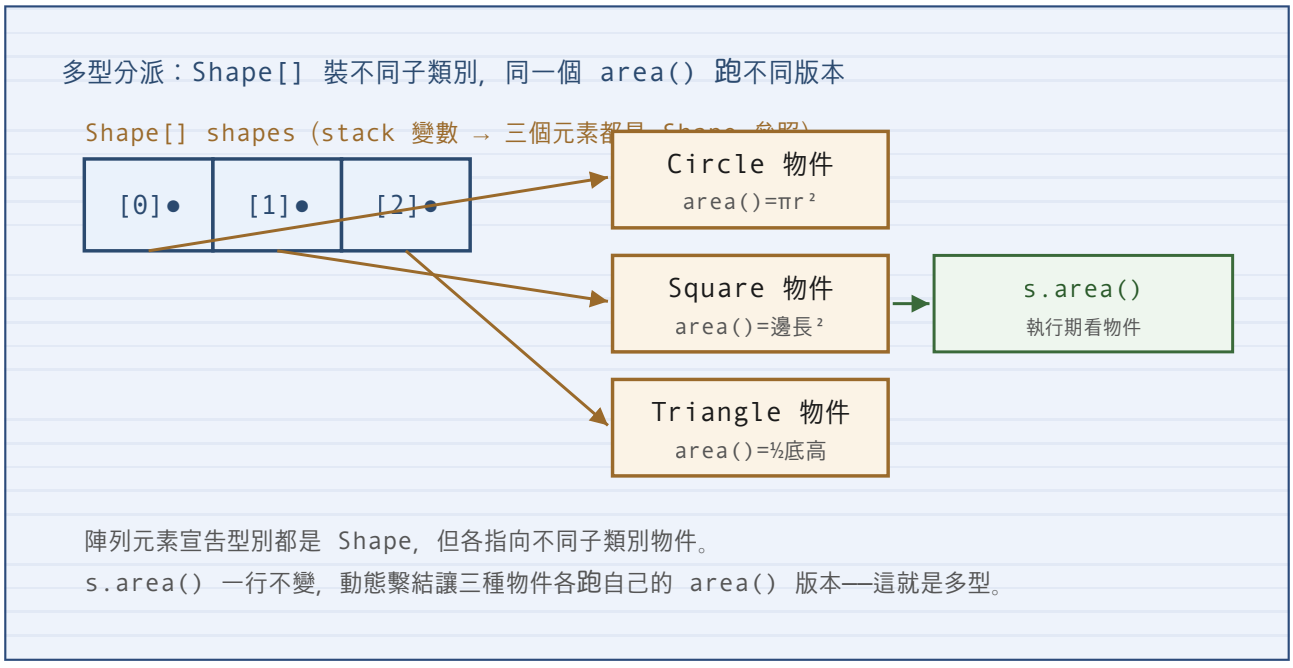
- 變數 `s1` 的**宣告型別是 Shape**：透過 `s1` 只能呼叫 Shape 定義的方法（看不到 Circle 特有的方法）。
- 但 `s1` 指向的**物件實際是 Circle**：`s1.area()` 會跑 Circle 的版本。

3.2 2. 動態繫結：執行期才決定跑哪個版本

當不同子類別都覆寫了同一個方法，透過 superclass 變數呼叫時，**JVM 在執行期看物件的實際型別**，決定呼叫哪個版本。這就是動態繫結，也是多型的引擎：

```
for (Shape s : shapes)
    System.out.printf("%s 面積 = %.2f%n", s.name(), s.area());
// 同一行 s.area()，Circle 跑 πr²、Square 跑邊長²、Triangle 跑 ½底高
```

> 對比：CH6 的方法**多載（overloading）**是「同名不同參數」，編譯期就依引數型別選定（靜態繫結）。多型的**覆寫（overriding）**是「同簽名、不同類別」，執行期才依物件選定（動態繫結）。兩者完全不同。



3.3 3. 抽象類別與抽象方法：強制子類別實作

「Shape」這個概念沒有具體的面積算法（要看是圓是方），所以它該是**抽象類別**：宣告 `abstract`、**不能被** `new`，只能被繼承。它可以有**抽象方法**（只有簽名、沒有 body、用分號結尾），強制每個具象子類別都提供實作：

```
public abstract class Shape {
    public abstract double area();           // 抽象方法：沒有 body，強制子類別實作
    public String describe() {                // 抽象類別仍可有一般方法
        return "面積 = " + area();            // 可呼叫抽象方法（執行期跑子類別版本）
    }
}
public class Circle extends Shape {
    private double r;
    public Circle(double r) { this.r = r; }
    @Override public double area() { return Math.PI * r * r; }   // 必須實作
}
// Shape s = new Shape();    ← 編譯錯：抽象類別不能 new
// Shape s = new Circle(5); ← OK：Circle 是具象子類別
```

- 類別只要**有一個抽象方法**就必須宣告為 `abstract`。
- 子類別若**沒實作完所有抽象方法**，它自己也必須是 `abstract`（不能 `new`）。
- 抽象類別**可以有建構子、欄位、一般方法**，只是不能直接被實例化——它的建構子由子類別 `super(...)` 呼叫。

3.4 4. final 方法與 final 類別：禁止覆寫／繼承

`final` 用在繼承上有兩種意思：

```
public class Account {
    public final double getBalance() { return balance; }  // final 方法：子類別不能覆寫
}
public final class String { ... }                          // final 類別：不能被繼承
```

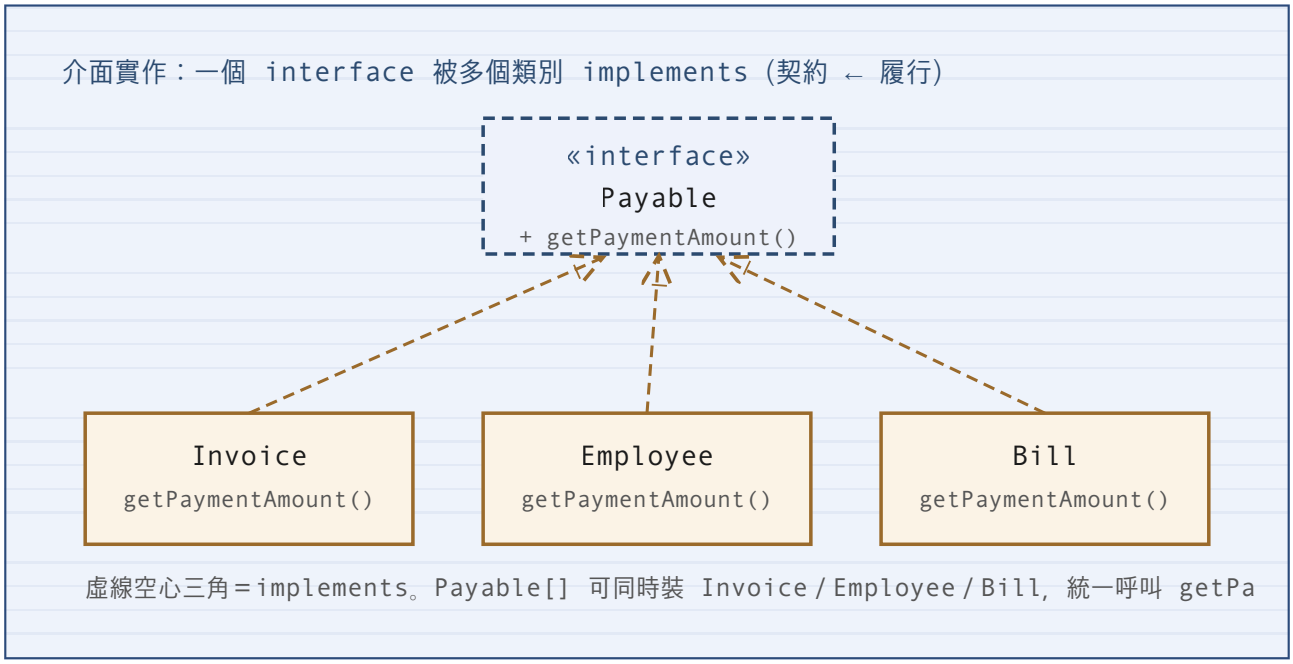
- `final` **方法**：子類別不能覆寫它，保證行為固定（常用於不想被改掉的核心邏輯）。
- `final` **類別**：不能被 `extends`（如 `java.lang.String`、`Integer`），其所有方法自動視為 final。
- 與多型的關係：`final` 方法**不參與動態繫結**（沒有別的版本可選），編譯器可直接靜態繫結、效能略好。

3.5 5. 介面 interface：純粹的能力契約

介面（interface） 只規定「要會做什麼」，不規定「是什麼」或「怎麼存資料」。用 `interface` 宣告，方法預設都是 `public abstract`（不用寫）；類別用 `implements` 履行契約，必須提供所有方法的實作：

```
public interface Drawable {
    void draw();                    // 介面方法預設 public abstract，不必寫修飾子
}
public class Circle implements Drawable {
    @Override public void draw() { System.out.println("畫一個圓"); }  // 必須實作
}
Drawable d = new Circle();          // 介面也能當型別、向上轉型，多型一樣成立
d.draw();                           // 動態繫結：跑 Circle 的 draw()
```

- 介面**不能** `new`（它沒有 body），但**能當變數型別**——這讓多型不必靠繼承同一個父類別。
- 介面裡的欄位自動是 `public static final`（常數）。
- 抽象類別 vs 介面：抽象類別表達「是什麼」（is-a，可有欄位/建構子/一般方法）；介面表達「會什麼」（can-do，純契約）。一個類別只能 `extends` 一個抽象類別，卻能 `implements` 多個介面。



3.6 6. 多重介面、向下轉型與 instanceof

一個類別只能 `extends` **一個**父類別，但能 `implements` **多個**介面，逗號分隔——這是 Java 表達「同時具備多種能力」的方式：

```
public class Bird extends Animal implements Flyable, Comparable<Bird> {
    @Override public void fly() { ... }                    // 履行 Flyable
    @Override public int compareTo(Bird o) { ... }         // 履行 Comparable
}
```

當你拿著 superclass／介面型別的變數，想用回子類別特有的方法時，要**向下轉型（downcasting）**，且必須先用 `instanceof` 確認實際型別，否則可能 `ClassCastException`：

```
for (Shape s : shapes) {
    if (s instanceof Circle) {            // 先確認 s 實際是 Circle
        Circle c = (Circle) s;            // 安全向下轉型
        System.out.println("半徑 = " + c.getRadius());   // 呼叫 Circle 特有方法
    }
}
```

- 向上轉型自動且安全；**向下轉型要明寫（子類別）且先 `instanceof` 檢查**。
- `instanceof` 對 `null` 一律回 `false`（不會 NPE）。

3.7 7. default 方法（Java 8+，輕量補充）

Java 8 起，介面方法可以帶 body，稱 **default 方法**（用 `default` 修飾）。實作類別可以**不覆寫、直接沿用**，讓介面新增方法時不會逼所有舊類別都改：

```
public interface Greeter {
    String name();
    default String greet() {              // default 方法：有 body
        return "Hello, " + name();        // 實作類別可直接用，也可選擇覆寫
    }
}
public class En implements Greeter {
    @Override public String name() { return "Sue"; }   // 只需實作抽象的 name()
    // greet() 不寫也能用 → 回 "Hello, Sue"
}
```

- default 方法**有 body**，所以不算抽象方法、不強制實作。
- 介面也可有 `static` 方法（用 `介面名.方法()` 呼叫）。這些是 Java 8+ 讓介面更實用的補強，初學知道「介面方法可以有預設實作」即可，細節進階再深究。

§ 4 四、語法與 API 速查

```
Shape s = new Circle(5);              // 向上轉型：superclass 變數指 subclass 物件（自動、安全）
Shape[] arr = { new Circle(5), new Square(3) };  // 一個陣列裝多種子類別
for (Shape x : arr) x.area();         // 動態繫結：執行期依物件實際型別跑對應 area()

public abstract class Shape {         // 抽象類別：不能 new
    public abstract double area();    // 抽象方法：無 body、分號結尾、強制子類別實作
}
public class Circle extends Shape {
    @Override public double area() { ... }   // 具象子類別必須實作所有抽象方法
}

public final 型別 方法() { ... }      // final 方法：子類別不能覆寫
public final class 類名 { ... }       // final 類別：不能被繼承（如 String）

public interface Drawable {           // 介面：純契約
    void draw();                      // 方法預設 public abstract（不用寫）
    default void hint() { ... }       // default 方法（Java 8+）：有 body，可不覆寫
}
public class C implements Drawable, Comparable<C> {   // 多重介面：逗號分隔
    @Override public void draw() { ... }
}

if (s instanceof Circle) {            // 向下轉型前先用 instanceof 確認
    Circle c = (Circle) s;            // 明寫（子類別）才能轉
}
```

- **能呼叫什麼看「變數型別」，實際跑哪版看「物件型別」**（動態繫結）。
- **抽象類別 vs 介面**：抽象類別＝is-a（可有欄位/建構子/一般方法、只能繼承一個）；介面＝can-do（純契約、可實作多個）。

§ 5 五、常見錯誤

- `new` **抽象類別**：`new Shape()` 編譯錯（`Shape is abstract; cannot be instantiated`）；抽象類別只能被繼承、由具象子類別 `new`。
- **子類別漏實作抽象方法卻沒宣告 abstract**：留著抽象方法沒實作，子類別自己又不是 `abstract` → 編譯錯。
- **透過 superclass 變數呼叫 subclass 特有方法**：`Shape s = new Circle(); s.getRadius();` 編譯錯——`s` 的型別是 Shape，看不到 Circle 特有方法；要先向下轉型。
- **沒 `instanceof` 就強制向下轉型**：`Square sq = (Square) shapeThatIsCircle;` 執行期 `ClassCastException`；先 `if (s instanceof Square)` 再轉。
- **以為向上轉型會「丟失」覆寫**：`Shape s = new Circle(); s.area();` 仍跑 Circle 版本——動態繫結看的是物件，不是變數型別。
- **覆寫 vs 多載搞混**：覆寫是同簽名、執行期動態繫結；多載是同名不同參數、編譯期靜態決定。

- **想覆寫 `final` 方法**：`final` 方法子類別不能覆寫，硬寫編譯錯。
- **介面當成可 `new`**：`new Drawable()` 編譯錯（介面沒 body）；介面只能當型別、由 `implements` 的類別 `new`。
- `implements` **沒實作完介面所有方法**：少實作任一方法 → 編譯錯（除非該類別宣告為 `abstract`）。
- **default 方法誤以為一定要覆寫**：default 方法有 body，可不覆寫直接沿用；只有抽象方法才強制實作。

§ 6 六、練習題

例題 1：抽象類別 Shape + 多型陣列

設計抽象類別 `Shape`（抽象方法 `double area()`），子類別 `Circle`（半徑）與 `Square`（邊長）各實作 `area()`。在 `main` 用 `Shape[]` 裝一個 Circle 與一個 Square，迴圈印出各自面積。

> 1. `public abstract class Shape { public abstract double area(); }` 2. 兩個子類別 `extends Shape`、`@Override public double area()` 3. `Shape[] shapes = { new Circle(2), new Square(3) };` 迴圈 `s.area()`

```
public abstract class Shape {
    public abstract double area();
}
class Circle extends Shape {
    private double r;
    public Circle(double r) { this.r = r; }
    @Override public double area() { return Math.PI * r * r; }
}
class Square extends Shape {
    private double side;
    public Square(double side) { this.side = side; }
    @Override public double area() { return side * side; }
}
// Shape[] shapes = { new Circle(2), new Square(3) };
// for (Shape s : shapes) System.out.printf("%.2f%n", s.area());  // 12.57、9.00
```

易錯：`new Shape()`（抽象類別不能 new）；子類別漏 `@Override` 或漏實作 area()。

例題 2：介面 Payable + 多重實作

定義介面 `Payable { double getPaymentAmount(); }`，由 `Invoice`（單價×數量）與 `SalariedEmployee`（月薪）各實作。在 `main` 用 `Payable[]` 裝兩者，迴圈印出每個的應付金額。

> 1. `interface Payable { double getPaymentAmount(); }` 2. 兩類別 `implements Payable`、各 `@Override` 實作 3. `Payable[] items = { new Invoice(3, 19.99), new SalariedEmployee(50000) };`

```
interface Payable { double getPaymentAmount(); }
class Invoice implements Payable {
    private int qty; private double price;
    public Invoice(int qty, double price) { this.qty = qty; this.price =
price; }
    @Override public double getPaymentAmount() { return qty * price; }
}
class SalariedEmployee implements Payable {
    private double salary;
    public SalariedEmployee(double salary) { this.salary = salary; }
    @Override public double getPaymentAmount() { return salary; }
}
// Payable[] items = { new Invoice(3, 19.99), new SalariedEmployee(50000) };
// for (Payable p : items) System.out.printf("%.2f%n",
p.getPaymentAmount());  // 59.97、50000.00
```

易錯：`implements` 後漏實作 `getPaymentAmount()`（編譯錯）；以為介面能 `new`。

例題 3：向上轉型 + 動態繫結

設計 `Animal`（方法 `String speak()` 回 `"..."`），子類別 `Dog`（回 `"汪"`）與 `Cat`（回 `"喵"`）覆寫之。用 `Animal a = new Dog();` 與 `a = new Cat();` 各呼叫一次 `a.speak()`，觀察同一變數依物件跑不同版本。

> 1. `Animal` 含 `public String speak() { return "..."; }` 2. `Dog`、`Cat` 各 `@Override public String speak()` 3. `Animal a = new Dog(); ... a = new Cat();` 各印一次

```
public class Animal {
    public String speak() { return "..."; }
    public static void main(String[] args) {
        Animal a = new Dog();
        System.out.println(a.speak());   // 汪（動態繫結 → Dog 版）
        a = new Cat();
        System.out.println(a.speak());   // 喵（同一變數，現在指 Cat 物件）
    }
}
class Dog extends Animal { @Override public String speak() { return "汪"; } }
class Cat extends Animal { @Override public String speak() { return "喵"; } }
```

易錯：以為 `a` 宣告成 Animal 就跑 Animal 版（其實看物件、跑子類別版）。

例題 4：instanceof 向下轉型

接例題 1 的 `Shape`／`Circle`／`Square`。給 `Circle` 加 `getRadius()`。在 `main` 走訪 `Shape[]`，用 `instanceof` 判斷是 `Circle` 才印半徑、是 `Square` 才印邊長。

> 1. `Circle` 加 `public double getRadius()`、`Square` 加 `getSide()` 2. `for (Shape s : shapes)` 內 `if (s instanceof Circle) { Circle c = (Circle) s; … }` 3. `else if (s instanceof Square) { … }`

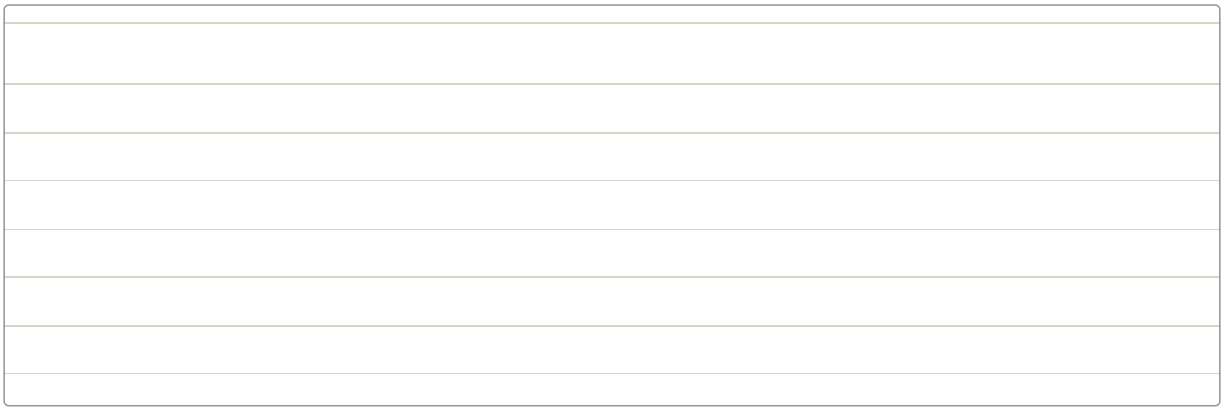
```
// 走訪 Shape[]，依實際型別向下轉型取特有資訊
for (Shape s : shapes) {
    if (s instanceof Circle) {
        Circle c = (Circle) s;
        System.out.println("圓，半徑 = " + c.getRadius());
    } else if (s instanceof Square) {
        Square sq = (Square) s;
        System.out.println("方，邊長 = " + sq.getSide());
    }
}
```

易錯：沒 `instanceof` 就 `(Circle) s` → `ClassCastException`；轉型寫成 `(Shape)` 沒意義。

例題 5：多重介面

定義 `Flyable { void fly(); }` 與 `Swimmable { void swim(); }`，讓 `Duck` 同時 `implements` 兩者。在 `main` 建一隻 Duck 並各呼叫 `fly()`、`swim()`。

> 1. 兩個介面各一個方法 2. `class Duck implements Flyable, Swimmable`（逗號分隔） 3. 兩個方法都要 `@Override` 實作



```
interface Flyable { void fly(); }
interface Swimmable { void swim(); }
public class Duck implements Flyable, Swimmable {
    @Override public void fly() { System.out.println("鴨子拍翅飛"); }
    @Override public void swim() { System.out.println("鴨子划水游"); }
    public static void main(String[] args) {
        Duck d = new Duck();
        d.fly();
        d.swim();
    }
}
```

易錯：多重介面用 `extends` 接（類別實作介面要用 `implements`）；只實作其中一個介面的方法（兩個都要）。

§ 7 七、自我檢核

- [ ] 能解釋多型：同一個方法呼叫對不同子類別物件跑不同版本。
- [ ] 知道向上轉型自動且安全，且「能呼叫什麼看變數型別、跑哪版看物件型別」。
- [ ] 講得出動態繫結是「執行期依物件實際型別決定呼叫哪個覆寫版本」。
- [ ] 會寫抽象類別與抽象方法，知道抽象類別不能 `new`、子類別必須實作所有抽象方法。
- [ ] 知道 `final` 方法不能覆寫、`final` 類別不能繼承。

- [ ] 會定義 `interface` 並用 `implements` 履行，知道介面能當型別、不能 `new`。
- [ ] 會用多重介面（一個類別 `implements` 多個），知道補繼承只能一個的限制。
- [ ] 會用 `instanceof` 確認後再向下轉型，避免 `ClassCastException`。
- [ ] 知道 default 方法有 body、可不覆寫沿用（Java 8+）。