✅ Java Basic

Last edited time	@March 19, 2025 9:40 AM
Summary	（1）遞增運算子（2）抽象介面（3）靜態非靜態（4）函式介面

前言

📍001📍編譯單元（Compilation Unit）

- 一個 **Compilation Unit（編譯單元）** 是指一個 Java 原始碼檔案（`.java` 檔案），其內容由 **單一的類別、介面、列舉或記錄（record）定義**，以及相關的 `package`、`import` 和註解等組成。
  - `package` **宣告（可選）**：若有 `package`，則必須放在檔案的最上方。
  - `import` **指令（可選）**：用於引入其他 Java 類別或套件。
  - **類別、介面、列舉、記錄定義**：可以包含多個類別，但最多只有一個 `public` 類別。

📍002📍原始檔名（Filename）

1. **公開類別（`public class`）的名稱必須與檔案名稱相同**：若 `.java` 檔案內含有一個 `public` 類別，那麼該類別的名稱必須與檔案名稱一致，否則會編譯錯誤。
2. **一個 `.java` 檔案內可以有多個類別，但最多只有一個 `public` 類別**

   ▼ 若 `.java` 檔案內包含多個類別，則這些類別不能宣告為 `public`，否則該 `public` 類別的名稱必須與檔名相同。

```
// 檔名：Main.java
public class Main {
    public static void main(String[] args) {
        Helper helper = new Helper();
        helper.sayHello();
    }
}
// 這個類別沒有宣告為 public，因此可以與 `Main` 放在同一個檔案
class Helper {
    void sayHello() {
        System.out.println("Hello from Helper!");
    }
}
```

3. **沒有 `public` 類別時，檔名可以與其中任何類別名稱相同**：如果 `.java` 檔案內的類別都 **沒有** `public` 修飾符，那麼該檔案的名稱可以是其中任意一個類別的名稱。
4. 介面（`interface`）、列舉（`enum`）、記錄（`record`）的 `public` 規則與類別相同

📍003📍命名慣例（Naming Convention）

類別名稱不要取跟標準類別的名稱一樣。你可以為自己的類別選擇更具描述性的名稱，以避免衝突。標準類別（或稱為預設類別）是由 Java 標準庫提供的類別，如 `String`、`Integer`、`System` 等。這些類別有特定的功能和用途，並且會被廣泛使用。如果你在自己的程式中創建與這些標準類別名稱相同的類別名稱，會引發以下問題：

1. **產生名稱衝突**：當你的自定義類別與標準類別名稱相同時，編譯器無法區分你創建的類別和 Java 標準庫中的類別。這會導致編譯錯誤或運行時錯誤，因為程式會嘗試引用錯誤的類別。
2. **隱藏標準類別的功能**：即便在某些情況下沒有編譯錯誤，使用與標準類別相同的名稱會使程式碼變得混淆，並可能隱藏 Java 標準庫的功能。例如，如果你定義了與 `System` 類相同名稱的類別，這將導致你無法使用 `System.out.println()` 等標準輸出功能。

3. **誤導其他開發者：**將自定義類別命名為與標準類別相同的名稱會讓其他開發者感到困惑。他們可能會認為你的類別是 Java 標準庫的一部分，或者誤以為它具有某些特殊的功能，可能會導致維護上的困難。

運算

📍004📍算術除法 （/）：Java 的整數除法 (`int / int`) 會自動捨去小數點，可以使用 `/` 運算子搭配 `int` 型別，或是使用 `Math.floorDiv()` 方法。

▼ 指如果你處理的是財務計算，可以使用 BigDecimal 來確保結果的精確性：`RoundingMode.DOWN`：表示「無條件捨去」，不管小數部分是多少，都直接去掉。

```
int result = Math.floorDiv(7, 3); // 結果是 2
System.out.println(result);

//如果你處理的是財務計算，可以使用 BigDecimal 來確保結果的精確性：

import java.math.BigDecimal;
import java.math.RoundingMode;

BigDecimal a = new BigDecimal("7");
BigDecimal b = new BigDecimal("3");
BigDecimal result = a.divide(b, 0, RoundingMode.DOWN); // 無條件捨去。0：表示保留 0 位小數（即只取整數部分）Roun
System.out.println(result); // 結果是 2
```

📍004-1📍java.lang.ArithmeticException: / by zero 。在 Java 中，當你嘗試用 int **0** 來做除數時，會發生 `java.lang.ArithmeticException: / by zero` 錯誤。

1. **整數除法 (`int / int`)：**如果你嘗試用 0 來做除數，就會觸發 `ArithmeticException`。
2. **浮點數除法 (`double / double`)：`double` 或 `float` 類型不會拋出例外**，而是會返回 **Infinity** （Double or Float / 0.0 ==正無限大或負無限大）**或 NaN**（0.0/0.0，任何數與NaN比較均為false）。
3. **如何避免 `/ by zero` 錯誤：**在進行整數除法前，（1）應該先檢查除數是否為 0。（2）如果不確定程式中是否可能出現除以 0 的情況，可以使用 `try-catch` 來捕捉例外，防止程式崩潰。

- Arithmetic Exception 是屬於 Runtime Exception 的一種。

📍005📍餘數運算（%）：(1) 先不考慮正負號直接做餘數運算。(2) 正負號由被除數決定。

1. **第一步**：先計算 `|A| % |B|`（忽略正負號的絕對值運算）
2. **第二步**：結果的正負號 **與** `A` **相同**（即 **被除數** 的正負號）

- 數學上的「模運算」（mod）與 Java 的 `%` 運算符略有不同：數學 `mod` 的結果總是非負，**Java 的** `%` **運算** 的餘數 **與被除數** `A` **同號**。

📍006📍單元運算子（Unary Operators）：`++` 運算子（遞增運算子）有 **前置遞增（`++x`）** 和 **後置遞增（`x++`）** 兩種形式，兩者的行為不同。。

- `++x` **（前置遞增）：(1) 先加 1，再進行其他運算（2）**適用於需要 **立即更新變數值** 的情境

▼ `x++` **（後置遞增）：(1) 先執行運算，再加 1（2）**適用於 **需要先使用舊值再遞增** 的情境

```
public class Main {
    public static void main(String[] args) {
        int a = 3;
        int b = ++a + 5; // a 先加 1 (a = 4)，再做 b = 4 + 5
        System.out.println("a = " + a); // 4
        System.out.println("b = " + b); // 9

        int c = 3;
```

```
        int d = c++ + 5; // 先做 d = 3 + 5，再讓 c = 4
        System.out.println("c = " + c); // 4
        System.out.println("d = " + d); // 8
    }
}
```

📍007📍邏輯運算子（Logical Operators）

運算子	名稱	規則
`&&`	**邏輯 AND（且）**	兩者都為 `true`，結果才是 `true`
`||`	**邏輯 OR（或）**	其一為 `true`，結果為 `true`
`^`	**邏輯 XOR（互斥或）**	兩者相同為 `false`，不同為 `true`
`!`	**邏輯 NOT（反轉）**	`true` 變 `false`，`false` 變 `true`

📍004-1📍**短路運算（Short-Circuit Evaluation）**

- `&&`（AND）：**若第一個條件是 `false`，後面條件不執行**
- `||`（OR）：**若第一個條件是 `true`，後面條件不執行**
- ▼ `^`（XOR）：**沒有短路運算，兩邊都一定會執行**
  - 在 **加密** 中，XOR 被用來對資料進行 **加密** 和 **解密**，其特性可讓數據在處理過程中得到反轉，並且是對稱的（加密和解密使用相同的 XOR 密鑰）。
  - **影像處理** 中，XOR 可以用來 **比較像素差異** 或進行 **圖像合成**。
- `&&` **和** `||` **有短路運算**，而 `^` 沒有

📍008📍型別轉換（Casting）：(1) 先不考慮正負號直接做餘數運算。(2) 正負號由被除數決定。

1. 主要型別：Java 會自動將 `byte`, `short`, `char` 轉換為 `int`，因為 CPU 運算以 `int` 或 `long` 為主
2. 自動類型轉換（Widening Conversion）：較小的類型可以自動轉換為較大的類型（Widening Conversion），避免溢位
3. 強制類型轉換（Narrowing Conversion）：從較大的類型轉換為較小的類型時（Narrowing Conversion），需要強制轉換（Casting），可能會導致數據丟失或溢位
4. 浮點數轉換為整數時，會捨棄小數部分
5. 在 Java 中，當你指定常數值時，如果該常數符合目標變數的範圍，通常不需要顯式轉換（`cast`）。但仍然會檢查範圍限制，超出範圍的值會導致編譯錯誤。

關鍵

📍009📍**值傳遞（Call by Value）**

在 Java 中，**所有參數傳遞都是 Call by Value（值傳遞）**，這表示：

▼ **基本型別（Primitive types）**：傳遞的是變數的**值**（副本）。

```
public class CallByValueExample {
    public static void modifyValue(int num) {
        num = 100; // 這裡改變的是 num 的副本，並不影響外部變數
    }

    public static void main(String[] args) {
        int x = 50;
        modifyValue(x);
        System.out.println(x);  // ➡️ 輸出仍然是 50，因為 x 沒有被改變
    }
}
```

▼ **參考型別（Reference types，例如 `Object`、`Array`）**：傳遞的是**物件的記憶體位址（引用）的副本**，但仍然是**值傳遞**。但是，如果我們試圖讓參數 `p` 指向新的物件，則不會影響原始變數。

```
class Person {
    String name;
```

```
    Person(String name) {
        this.name = name;
    }
}

public class ReferenceExample {
    public static void modifyPerson(Person p) {
        p.name = "Alice";  // ✅ 修改物件內容（有效）
    }

    public static void main(String[] args) {
        Person person = new Person("Bob");
        modifyPerson(person);
        System.out.println(person.name);  // ➡️ 輸出 "Alice"，因為物件內容被改變了
    }
}
//============================  modifyReference(p) 只是改變 p 的本地副本，不會影響 main() 裡的 person 變
public class ReferenceExample {
    public static void modifyReference(Person p) {
        p = new Person("Charlie");  // ❌ 這只是在方法內部改變 p，不影響外部變數
    }

    public static void main(String[] args) {
        Person person = new Person("Bob");
        modifyReference(person);
        System.out.println(person.name);  // ➡️ 輸出 "Bob"，沒有變成 "Charlie"
    }
}
```

型別	傳遞方式	可以修改內容？	可以改變變數本身的引用？
基本型別（`int`、`double`、`char`...）	**值傳遞**	❌ 無法修改原變數	❌
參考型別（`Object`、`Array`...）	**記憶體位址的副本（仍然是值傳遞）**	✅ 可以修改物件內容	❌ 無法改變原變數的引用

📍010📍 `static` 關鍵字（`static`）

`static` **關鍵字：** `static` 在 Java 中主要用來修飾變數、方法、類別等元素，表示它們屬於**類別本身**，而不是某個具體的物件。這意味著靜態成員不需要透過物件來存取，可以直接通過類別來存取。

1️⃣ 靜態變數（`static` 變數）

- `static`告訴Java編譯器，遇到static修飾的東西，就要優先分配記憶體空間給它。
- **描述：** 靜態變數是類別的共享變數，所有該類別的物件都共用同一份靜態變數，每個類別檔只會載入一次，所以只有一份靜態變數。
- 只會在程式啟動時（JVM在載入類別檔時）分配一次記憶體空間，就會為類別的靜態變數配置記憶體，其他類別就可以存取此類別的靜態變數，並且這塊記憶體會一直存在，直到程式結束。
- 靜態變數的值會被所有物件共享，對它的修改會影響所有物件。

2️⃣ 靜態方法（`static` 方法）

- **描述：** 靜態方法屬於類別，能夠在不創建物件的情況下調用。
- 靜態方法只能訪問靜態變數和靜態方法，無法直接訪問非靜態成員(實例變數或方法)（它不依賴於物件）。
- 靜態方法通常用於不需要物件狀態的情況，比如工具方法。
- **靜態方法（`static`）無法 Override（覆寫）：**靜態方法（`static` 方法）不能被 `override`，因為它們屬於 **類別（class）**，而非物件（instance），因此他們綁定是在編譯時執行，而不是在運行時。

▼ **靜態方法的** `hidding` ：如果在子類別中定義了一個與父類別 `static` 方法相同的方法，這不是 **覆寫（override）**，而是 **隱藏（method hiding）**。當子類別與父類別都有同名的靜態方法時，子類別實例的靜

- 可以透過類別名稱或是物件參考存取靜態變數或呼叫靜態方法。

態方法調用的是父類別的靜態方法，而非子類別的靜態方法；但若是實例方法則是調用子類別的方法。

- **靜態方法是屬於類別的**，不屬於物件。不會因為 `p2` 實際上是 `Child` 物件而執行 `Child.show()`。
- **這與多型（polymorphism）無關**，父類別變數仍然會執行父類別的 `static` 方法。
- **如果不是 `static` 方法，則 `p2.show()` 會執行 `Child.show()`**（因為多型機制適用於 `instance` 方法）。

```
class Parent {
    static void show() {
        System.out.println("Parent 的靜態方法");
    }
}

class Child extends Parent {
    static void show() {  // 這不是 override，而是 method hiding
        System.out.println("Child 的靜態方法");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent p1 = new Parent();
        Parent p2 = new Child();
        Child c = new Child();

        p1.show();  // 輸出：Parent 的靜態方法
        p2.show();  // 輸出：Parent 的靜態方法（⚠️ 不會呼叫子類別
        c.show();   // 輸出：Child 的靜態方法
    }
}
```

3️⃣ 靜態區塊 (`static` 區塊)

- **描述：** 靜態區塊在類別被載入時執行一次，通常用來進行靜態變數的初始化。
    - 靜態區塊在類別第一次被載入時執行，並且只執行一次，無論創建多少個物件。
    - 它可以用來處理靜態變數初始化或其他靜態設置。

4️⃣ 靜態類別 (嵌套靜態類別)

- 當一個類別是靜態的，它就不依賴於外部類別的實例，可以獨立存在。
- 靜態內部類別可以直接訪問外部類別的靜態成員。
- 靜態內部類別在內部類別不需要外部類別的物件時非常有用。
- 某類別如果沒有實例變數或是該類別的物件沒有任何不同的狀態，應將此類別的方法全部設計為靜態方法。優點是不用new物件就能使用該類別的方法，節省物件所佔用的記憶體和new所需要的執行時間。
- Math和system都不能產生物件，因為他們的建構子都含有修飾字private。不能new math()

5️⃣ 靜態變數的生命週期

- **記憶體分配：** 在程式的進入點 `main()` 方法執行之前，靜態變數就會分配記憶體並初始化，並且在整個程式的執行期間都不會釋放掉static資料的記憶體空間，所以才稱為靜態。

- **記憶體釋放：** 靜態變數會在程式結束時釋放，並且在此期間，靜態變數始終存在於記憶體中，這就是為什麼靜態變數是**共享的**，所有該類別的實例都會共用同一個靜態變數。

📍011📍靜態與非靜態（Static & NonStatic）

在 Java 中，靜態方法和非靜態成員（物件成員）之間有一些關鍵區別，這些區別是由於它們的生命週期和記憶體分配的方式所造成的。

🔷 靜態 (Static) → 屬於類別

- **靜態成員（方法或變數）** 是**屬於類別本身**，不需要透過物件就可以存取。**所有物件共用** 相同的靜態成員（變數或方法）。
- **呼叫方式：** 使用 **類別名稱** 來存取 (`ClassName.methodName()` 或 `ClassName.variableName`)。

▼ 📌 **注意：靜態方法內部不能直接使用非靜態成員**，因為由於靜態方法在物件實例創建之前就已經存在，物件的非靜態成員（如實例變數和實例方法）還不存在。因此，靜態方法無法直接訪問非靜態欄位或方法。

```
class MyClass {
    static int staticVar = 10;  // 靜態變數
    int nonStaticVar = 20;      // 非靜態變數

    static void staticMethod() {
        System.out.println("Static variable: " + staticVar);
        // 無法訪問非靜態變數
        // System.out.println("Non-static variable: " + nor
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass.staticMethod();  // 可以直接調用靜態方法
    }
}
```

在上面的程式中，`staticMethod` 只能訪問靜態變數 `staticVar`，無法直接訪問非靜態變數 `nonStaticVar`，因為靜態方法在類別載入時就會執行，而此時 `nonStaticVar` 尚未初始化。

🔷 非靜態 (Non-Static) → 屬於物件

- **非靜態成員（方法或變數） 必須透過物件來存取**，因為它們屬於**某個特定的物件**。
- **呼叫方式：** 先建立物件，再用 **dot notation (`.`)** 來存取。

▼ 非靜態方法會在物件被創建後執行，因此它可以直接訪問物件的非靜態欄位和靜態欄位。

```
class MyClass {
    static int staticVar = 10;  // 靜態變數
    int nonStaticVar = 20;      // 非靜態變數

    void instanceMethod() {
```

🔷 兩者的核心差異

	靜態 (Static)	**非靜態 (Non-Static)**
屬於	類別 (Class)	物件 (Object)
存取方式	**類別名稱.方法()** 或 **類別名稱.變數**	物件.方法() 或 物件.變數
是否需要建立物件？	❌ 不需要	✅ 需要先建立物件
是否可共用？	✅ 所有物件共用	❌ 每個物件都有自己的值
是否能存取非靜態？	❌ **靜態方法不能存取非靜態成員**	✅ 非靜態可以存取靜態成員

```
        System.out.println("Non-static variable: " + nonS
        System.out.println("Static variable: " + staticVar);
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.instanceMethod();  // 透過物件來調用非靜態方
    }
}
```

🔷 何時使用靜態 vs 非靜態

✅ **靜態 (Static) 適合：**

- 不需要依賴物件狀態的工具方法 (如 `Math.sqrt()`)。
- 共享的資料，例如常數 (`final static`) 或計數器。

✅ **非靜態 (Non-Static) 適合：**

- 與物件的**狀態（變數）相關**的方法。
- 每個物件都應該有自己的變數值，例如 `Person` 的 `name`。

靜態方法 (Static Method)

- **描述：** 靜態方法是屬於類別的，而不是某個物件。這意味著靜態方法在整個程式執行期間只有一份實例。
- **記憶體分配：** 靜態方法在類別載入時就會被分配記憶體，而不需要創建物件。
- **訪問限制：** 靜態方法只能存取**靜態變數**和**靜態方法**，因為它不依賴於物件實例，並且在程式剛載入時，物件的實例變數還沒有被初始化。

非靜態方法 (Instance Method)

- **描述：** 非靜態方法是與物件實例相關的，每個物件都會有自己的方法副本，並且這些方法可以訪問該物件的實例變數和類別變數。
- **訪問限制：** 非靜態方法既可以訪問**非靜態變數**，也可以訪問**靜態變數**。

📍012📍**成員變數、區域變數、類別變數（Instance Variable、Local Variable、Class Variable）**

1. 成員變數（Instance Variable）

- 定義在 **類別內、方法外** 的變數，屬於 **每個物件的實例**，有效範圍為整個物件。
- **不同物件** 會有 **不同的成員變數副本**（Instance），使用時需用物件dot notation。
- **預設值**：會根據型別自動初始化（例如 `int` 預設為 `0`，`String` 預設為 `null`）
- **實例方法（Instance Method）** 是物件的方法⇒**必須先建立物件，才能呼叫方法**。

2. 區域變數（Local Variable）

- 定義在**方法或區塊內** 的變數，只能在該區塊內使用。
- **不能有未初始化的值**，必須手動賦值後才能使用。

3. 類別變數（Class Variable/ 靜態變數 `static`）

- 使用 `static` **修飾**，表示**所有物件共用**。
- 存在**類別本身**，而非物件內，所以**可透過類別名稱存取**。
- **只會存在一份**，不論有多少物件，都共用同一個變數值，好處是節省記憶體。
- 如果方法加上 `static`，⇒**類別方法（Static Method）**，可以直接用類別名稱呼叫，而不需要 `new` 物件。

變數類型	定義位置	是否需要初始化	生命週期	是否獨立於物件	存
成員變數（Instance Variable）	類別內、方法外	會有預設值	物件存在時	否，每個物件都有自己的副本	ob
區域變數（Local Variable）	方法內或區塊內	**必須手動初始化**	方法執行時	是，每次呼叫方法時產生新的變數	只

- **方法執行完後變數就會被釋放**（生命週期短）。

類別變數（Class Variable / `static`）	`static` 修飾的變數	會有預設值	程式執行期間	是，所有物件共用同一份	Cl 或

📍013📍方法過載（Method Overloading）

- **方法過載（Method Overloading）指的是在同一個類別中，定義多個名稱相同但參數不同的方法**。這些方法**具有相似的功能**，但可以處理不同類型或數量的參數。

1. 方法名稱必須相同，參數列**（參數數量、類型或順序）必須不同**。
2. **回傳值型態可以相同或不同，但不能僅靠回傳型態來區分過載方法**。
3. 可以減少我們記憶體上的負擔。指string class裡面有9個value of的方法。

- **方法覆寫（Method Overriding）是指在子類別**中定義一個與**父類別**中已經存在的方法名稱、參數列、回傳型態完全相同的方法，以達到改寫父類別方法的效果。覆寫的目的是讓子類別能夠提供一個**新的實現**，來取代父類別的方法。

1. **方法名稱必須相同**：子類別的方法名稱要與父類別的方法名稱一致。
2. **參數列必須相同**：子類別覆寫的方法參數類型、數量和順序必須與父類別方法完全一致。
3. **回傳型態必須相同**：回傳型態應與父類別的方法相同，但可以是**協變回傳型態（covariant return type）**，即子類別方法可以回傳父類別的子類型。
4. **存取修飾詞**：子類別覆寫的方法的存取修飾詞不能比父類別方法的存取修飾詞更嚴格。若父類別的方法是 `public`，則子類別覆寫的方法也必須是 `public`，不能是 `protected` 或 `private`。
5. **拋出異常**：子類別覆寫的方法所拋出的異常類型不能比父類別方法拋出的異常類型更多。子類別可以選擇不拋出異常，或者拋出與父類別相同或更具體的異常類型。

方法過載（Overloading） vs 方法覆寫（Overriding）

比較項目	方法過載（Overloading）	方法覆寫（Overriding）
發生範圍	同一個類別內	父類別與子類別之間
方法名稱	必須相同	必須相同
參數列	必須不同（數量、類型、順序不同）	必須相同
回傳值	可以相同或不同	必須相同（但 `covariant return type` 允許回傳子類別型別）
存取修飾詞	不受影響	限制不能縮小存取範圍（例如 `public` 方法不能改成 `private`）
目的	讓方法處理不同類型的輸入	改寫父類別的方法以實現不同行為

▼ 6. `super` **關鍵字：**當子類別方法覆寫了父類別的方法，但仍然需要呼叫父類別的版本時，可以使用 `super` 關鍵字來調用父類別的方法。

```
class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        super.sound(); // 呼叫父類別的 sound 方法
        System.out.println("Dog barks");
    }
}

public class OverridingExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
```

```
        dog.sound(); // ➡️ 輸出：Animal makes a sound\nDog barks
    }
}
```

📍014📍 abstract 關鍵字（abstract）

abstract **是 Java 的關鍵字，用於建立抽象類別（abstract class）和抽象方法（abstract method）。它的主要目的是 讓子類別繼承並實作抽象方法，以實現多型（polymorphism）。**

- 定義模板：強制子類實現特定方法，從而確保一致性。抽象類別通常是一組類別的共同組先，其主要功能在於定義同一組規格。
- 共享代碼：可以在抽象類別裡面提供一部分實作，讓子類可以繼承部分實作，減少代碼重複。
- 邏輯分層：有助於將複雜的邏輯分層，提供清晰的設計結構。子類別實作特定細節，可以增加多態性

1️抽象類別（abstract class）

- **不能直接建立物件**（new）。抽象類別不能產生物件，抽象類別通常是要被繼承，由子類別override所有抽象方法，提供方法本體，再由子類別產生物件。
- **可以包含普通方法。抽象類別不一定要有 abstract 方法，**但仍然**不能被實例化。**
- 若子類別繼承後沒overriding所有父代類別裡的抽象方法時，要繼續宣告為 abstract 類別。
- （1）**建立「規格」**，讓子類別來實作指抽象類別通常為一組類別的祖先，主要功能在定義一組行為的規格。（2）**共享部分行為**（普通方法），但強制子類別**實作關鍵功能**（抽象方法）。
- 即使某類別並沒有內含抽象方法，該類別一樣可以宣告為抽象類別。這樣的class不能用於產生物件。限制直接實例化、它仍然可能是一個概念上的父類，**作為框架或 API 的基礎類別**

2️抽象方法（abstract method）

- **不能有大括號** {}（即沒有方法實作，不可以有方法本體）。
- **只能在抽象類別內**（abstract class）。：含有抽象方法的類別，一定要宣告為抽象類別
- 必須被子類別覆寫（override），否則子類別本身也必須宣告為 abstract。
- 使用場景：定義一個「規範」，但不提供實作，讓所有子類別都能遵循這個規範。
- 抽象方法只能有abstract public 和protected三個修飾字，或是default存取權限。

abstract 的特點

項目	抽象類別（abstract class）	抽象方法（abstract method）
能否有實作？	✅ **可以** 包含已實作的方法	❌ **不能** 有方法本體
能否建立物件？	❌ **不能**（new 會錯誤）	❌ **不能獨立存在**，只能放在抽象類別
子類別	✅ **可以繼承，但必須** 實作所有抽象方法	✅ **子類別必須覆寫**
修飾符	abstract class	abstract public/protected

🔥 抽象（abstract） VS 介面（interface）

特性	抽象類別（abstract class）	介面（interface）
方法類型	有 abstract 方法，也可以有普通方法	**只能有抽象方法**（Java 8+ 可有 default 方法）
變數	可以有 final 或非 final 變數	**所有變數預設 public static final（常數）**
繼承方式	**單一繼承**（只能繼承一個抽象類別）	**多重實作**（可以實作多個介面）
	定義一組相似的類別，部分功能共享	**定義規範**，讓不同類別實作相同行為

📍015📍 介面（interface）

interface 是用來**定義一組規範，而不包含任何具體實作的方法。**與 abstract class（抽象類別）不同，介面**通常用來讓不相關的類別實作相同的行為，** Java**不支援多重繼承，但支援多重介面實作。**

▼ 1️ 介面（interface）的特性

📌 **使用建議：**

- **如果是「行為規範」，應該用** interface。

```
interface Animal {  // 定義介面
    void makeSound();  // ✅ 預設是 public abstract
}

// ✅ Dog 類別實作 Animal 介面
class Dog implements Animal {
    public void makeSound() {  // ✅ 必須覆寫介面方法
        System.out.println("Woof! Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();  // Woof! Woof!
    }
}
```

- `interface Animal` 內的方法 **預設為** `public abstract`，所以 `Dog` 必須實作 `makeSound()`。
- `Dog` 用 `implements` 來實作 `Animal`（**與** `extends` **繼承不同**）。
- **不能** `new Animal()`**，因為介面沒有方法實作**。

1. **不能建立物件**（`new`）。
2. **所有方法**都是 `public abstract`（即使不寫 `public`）。
3. **所有變數**都是 `public static final`**(常數)**，宣告後需要立即設定初值。
4. **允許一個類別實作多個介面（多重繼承）**。

▼ **5.Java 8+**可有 `default` **方法**(有預設實作)和靜態方法。，且方法的回傳型態前加入`default`

- 實作介面的類別可以override預設方法，也可以不要 override 。
- 靜態方法也要編寫方法本體，作為該方法的行為。

```
<modifier> interface interfacename1 extends interface
[public] datatype var1 = constant1;
[public] datatype method1 (argument list);

//=======================================
interface Animal {
    void makeSound();

    default void sleep() {  // ✅ 介面也能有實作（Java 8+
        System.out.println("Sleeping...");
    }
}

class Dog implements Animal {
```

- **如果是「類別繼承關係」，應該用** `abstract class`。
- **若要多重繼承(多個來源)**或找不到共同的父類別來進行多型時，**用** `interface`。
- 當某類別同時繼承父類別與實作某介面時，關鍵字extends必須在關鍵字implements之前。
- Java只允許單一繼承。

```
    public void makeSound() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();  // Woof!
        dog.sleep();      // Sleeping...
    }
}
```

▼ **6.介面的應用場景**：介面通常用於**定義規範**👉Java 的 `Comparable` 介面

```
class Student implements Comparable<Student> {
    String name;
    int score;

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

    @Override
    public int compareTo(Student other) {
        return this.score - other.score;  // 按分數排序
    }
}
```

類別

📍016📍類別轉換&包裝類別（Type Conversion＆Wrapper Class）

`Integer.parseInt()`：將合法的數字字串轉成基本型別 `int`，但如果字串包含非數字的「奇怪字元」，例如 `"123a"`，則會拋出 `NumberFormatException`，表示字串格式不符合數字轉換的要求。

方法	回傳型別	主要用途	特色
`ClassName.parseXXX()`	基本型別 (`int`, `double` 等)	解析字串成對應的**基本數值型別**	只回傳基本型別，不會回傳物件，效能較好。
`ClassName.valueOf(參數)`	包裝類別 (`Integer`, `Double` 等)	將參數轉換為本方法所屬的類別物件	回傳的是物件，可使用額外方法

- `parseInt()` 只回傳基本型別 `int`，所以不存在「位址」的概念，因為基本型別不是物件，而是直接存放在記憶體中的值。`int` 是值類型，比較時只會比較數值，不涉及記憶體位址。

▼ `valueOf()` 會回傳 `Integer` 物件，且 範圍在 `-128 ~ 127` 內的數字會共用相同的物件，因為 Java 內部有 **整數緩存池（Integer Cache）**。

```
Integer x = Integer.valueOf(100);
Integer y = Integer.valueOf(100);
System.out.println(x == y);  // true，因為 100 在 -128 ~ 127 內，使用相同物件
Integer m = Integer.valueOf(200);
```

```
Integer n = Integer.valueOf(200);
System.out.println(m == n);  // false，因為 200 超過緩存範圍，創建了新的物件
```

▼ 在 **Java 的包裝類別（Wrapper Class）** 中，**每個物件只能包裝一個基本型態的數值，且該數值是不可變（Immutable）的**。包裝類別的設計是 **不可變物件（Immutable Object）**，被 `final` 修飾。

- 乍看之下，好像 `x` 的值變了？**其實並沒有！**Java並沒有修改原本的 `Integer` 物件，**而是建立了一個新的 `Integer` 物件，並讓 `x` 指向新的物件**。`Integer` 類別內部的 `value` 是 `final` 修飾的。

```
Integer x = 10;
System.out.println(x);  // 輸出 10

x = 20;
System.out.println(x);  // 輸出 20

//================================
public final class Integer extends Number implements Comparable<Integer> {
    private final int value;
}
```

- 自動裝箱（**Auto-boxing**）和自動拆箱（**Auto-unboxing**）是從 **Java 5.0** 開始引入的功能，使得 **基本型態（primitive type）** 和 **包裝類別（wrapper class）** 之間的轉換變得更加簡單和自動化。
  - **自動裝箱（Auto-boxing）**：自動裝箱指的是 **基本型態** 自動轉換為對應的 **包裝類別物件**。這是在賦值或操作時，由 Java 編譯器自動完成的。

  ▼ **自動拆箱（Auto-unboxing）**：自動拆箱指的是 **包裝類別物件** 自動轉換為對應的 **基本型態**。這同樣是由 Java 編譯器在需要時自動完成的。

  ```
  Integer a = 5;        // 自動裝箱
  Integer b = 10;       // 自動裝箱

  int sum = a + b;      // 自動拆箱：a 和 b 先被轉換為基本型態，再進行加法運算
  System.out.println(sum); // 輸出 15
  ```

  - **1. 對於 `null` 的處理**：當包裝類別為 `null` 時，會發生 `NullPointerException`，因為 `null` 無法自動拆箱為基本型態。
  - **2. 性能問題**：自動裝箱和拆箱過程會涉及 **額外的物件創建**，這可能會導致性能上的開銷，尤其是在頻繁進行數據處理的情況下。

📍017📍物件導向（OOP）

📌 封裝（Encapsulation）

封裝是將**資料（變數）與操作資料的方法（函式）**放在同一個類別中，並透過 **存取修飾符（Access Modifiers）** 來保護資料，使外部無法直接存取，而必須透過方法來存取和修改。

🔷 主要概念

1. **變數應設為 `private`**，不讓外部類別直接存取。
2. **提供 `public` 的 `getter` 和 `setter` 方法** 來讀取和修改變數。
3. **提升安全性**，避免外部類別隨意修改物件內部的狀態。

✅ **封裝的好處**：

- 防止外部直接修改變數，確保資料的安全性。

📌 繼承（Inheritance）

繼承允許子類別（`Subclass`）**擁有父類別（`Superclass`）的屬性與方法**，但**不會繼承建構子**。Java 中的繼承透過 `extends` 關鍵字來實現。

🔷 主要概念

1. **子類別擁有父類別的屬性與方法**，但**不會繼承建構子**。is_a releationship
2. **Java 是單一繼承語言**，一個類別只能有一個直接父類別。X是Y的一種。
3. **所有類別的最上層父類別是 `Object`**，即 `class A extends Object` 是隱含的。

- 讓類別內部的實作細節對外部隱藏，提高模組化與維護性。

✅ **繼承的好處**：

- **減少重複程式碼**，子類別可直接使用父類別的方法。
- **提高可擴展性**，可以在子類別中新增或覆寫方法。

📌 多型（Polymorphism）

多型允許**相同的方法在不同類別中表現出不同的行為**。Java 的多型主要透過 **方法覆寫（Override）** 和 **方法重載（Overload）** 來實現。

🔷 1️⃣ **多型參數**：當方法的參數型別是父類別時，任何**父類別的子類別**都可以傳入該方法。

> 👉 **即使 `animalSpeak` 方法的參數型別是 `Animal`，但傳入 `Dog` 和 `Cat` 物件時，會執行各自覆寫的 `makeSound` 方法，這就是多型的應用。**

🔷 2️⃣ **異質集合（Heterogeneous Collection）**：當一個陣列的型別是父類別時，它可以存放任何該類別的子類別物件。

✅ **多型的好處**：

- **提高程式的靈活性**，可以用統一的方式處理不同型別的子類別。
- **減少程式碼耦合**，讓程式更容易擴展。

⬅️ END 總結

特性	內容
封裝（Encapsulation）	將資料與方法封裝在類別內，透過 `private` 限制存取，並提供 `getter` 和 `setter` 方法來存取資料。
繼承（Inheritance）	允許子類別繼承父類別的屬性與方法，但不繼承建構子。Java 只支援單一繼承，所有類別的最上層父類別是 `Object`。
多型（Polymorphism）	允許相同的方法在不同類別中表現不同的行為，包括方法覆寫、多型參數、異質集合等應用。

這三大特性是 Java **物件導向程式設計（OOP）** 的核心概念，透過適當的運用可以讓程式更靈活、可維護性更高！🚀

📍018📍應用多型思路（U）

- 1️⃣ **先找共同父類別（繼承）**：多個類別有相似的行為（方法），將這些行為定義在 **共同的父類別** 中，讓子類別繼承。`cry()` 是 **家族共同方法**，所有繼承 `Animal` 的子類別都會擁有這個方法。
- 2️⃣ **子類別繼承並擴展特有方法**：子類別可以繼承父類別的 `cry()` 方法，並添加自己的特有方法。

▼ 3️⃣ **多型參數與 `instanceof` 檢查**：當我們使用 **父類別作為參數**，就可以傳入**不同的子類別物件**，這是 **多型參數（Polymorphic Parameters）** 的應用。但如果我們想呼叫子類別的**特有方法**，則需要先進行 **型別檢查與轉換（Downcasting）**。

```
class Animal {
    public void cry() { // 家族共同方法
        System.out.println("Animal makes a sound.");
    }
}
class Dog extends Animal {
    public void bark() { // 特有方法
        System.out.println("Dog barks.");
    }
}

class Cat extends Animal {
    public void meow() { // 特有方法
        System.out.println("Cat meows.");
    }
}
public class Main {
    public static void makeAnimalCry(Animal animal) {
        animal.cry(); // 呼叫家族共同方法

        // 檢查型別，並轉換成子類別存值
        if (animal instanceof Dog) {
```

```
        Dog dog = (Dog) animal; // 向下轉型
        dog.bark();  // 呼叫 Dog 的特有方法
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal; // 向下轉型
        cat.meow();  // 呼叫 Cat 的特有方法
    }
  }

  public static void main(String[] args) {
    Animal myDog = new Dog();
    Animal myCat = new Cat();

    makeAnimalCry(myDog); // Dog
    makeAnimalCry(myCat); // Cat
  }
}
```

📍019📍向下轉型（Downcasting）

當我們使用**父類別的變數來引用子類別的物件時**，只能存取**父類別中定義的方法與屬性**，無法直接存取**子類別特有的方法**。如果要使用子類別的特有方法，必須進行**向下轉型（Downcasting）**。

▼ **向上轉型（Upcasting）：**向上轉型指的是 **「子類別物件轉換為父類別型別」**，這是**自動發生**的，無需額外轉換。

- 向上轉型後，`animal` 變數雖然實際上是一個 `Dog` 物件，但只能存取 `Animal` 類別的方法，無法使用 `bark()`。

```
class Animal {
  public void makeSound() {
    System.out.println("Animal makes a sound.");
  }
}

class Dog extends Animal {
  public void bark() {
    System.out.println("Dog barks.");
  }
}

public class Main {
  public static void main(String[] args) {
    Animal animal = new Dog();  // ✅ 向上轉型，自動發生
    animal.makeSound(); // 可以呼叫父類別的方法
    // animal.bark(); ❌ 錯誤：父類別型別的變數無法直接使用子類別的方法
  }
}
```

▼ **向下轉型（Downcasting）：**如果想存取子類別特有的方法，就必須將 **父類別的變數轉回子類別型別**，這稱為**向下轉型**。

```
class Animal {
  public void makeSound() {
    System.out.println("Animal makes a sound.");
  }
}

class Dog extends Animal {
```

```
    public void bark() {
        System.out.println("Dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // 向上轉型
        Dog dog = (Dog) animal;  // ✅ 向下轉型
        dog.bark();  // ✅ 成功呼叫 Dog 的方法
    }
}
```

▼ 檢查型別上下關係：**使用** `instanceof` **避免** `ClassCastException`。如果向下轉型時，變數實際上不是該類型，則會拋出 `ClassCastException` 錯誤。因此，轉型前可以使用 `instanceof` 來檢查變數。

```
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // 向上轉型

        if (animal instanceof Dog) { // 檢查 animal 是否為 Dog 的實例
            Dog dog = (Dog) animal; // ✅ 安全向下轉型
            dog.bark();
        } else {
            System.out.println("轉型失敗，animal 不是 Dog 的實例");
        }
    }
}
```

轉型方式	描述	是否需要強制轉型
向上轉型（Upcasting）	**子類別 → 父類別**，可以使用父類別的方法，但不能使用子類別特有的方法。	❌ 自動發生
向下轉型（Downcasting）	**父類別 → 子類別**，可以使用子類別特有的方法，但要確保變數實際是該子類型，否則會拋錯。	✅ 需要 `(子類別型別)`

▼ **用** `new` **創建出來的物件，型態永遠不會改變**。物件在記憶體中的真實類型（即 **實例的類型**）是固定的。指 `new Dog()` 創建出來的物件，永遠都是 `Dog`，不會變成 `Animal` 或 `Cat`。

```
class Animal {}
class Dog extends Animal {}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();  // ✅ dog 變數指向一個 Dog 物件
        Animal animal = dog;  // ✅ 向上轉型：變數 animal 指向 Dog 物件
        Object obj = animal;  // ✅ 向上轉型：變數 obj 指向 Dog 物件

        // dog 這個物件仍然是 Dog，雖然它被不同類型的變數存放
    }
}
```

▼ 存放物件的變數（參考變數）可以改變型態，但只能在 父類別與子類別之間轉換。**變數可以指向不同類型的物件**，但只能在繼承關係內轉換（向上轉型 & 向下轉型）。

```
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound.");
    }
}

class Dog extends Animal {
    public void makeSound() {  // 覆寫父類別方法
        System.out.println("Dog barks.");
    }

    public void fetch() {
        System.out.println("Dog fetches a ball.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // ✅ 向上轉型：變數 myAnimal 指向 Dog 物件
        myAnimal.makeSound();  // ⚠️ 輸出 "Dog barks."（因為多型）

        // myAnimal.fetch(); ❌ 錯誤：父類別 Animal 沒有 fetch() 方法

        if (myAnimal instanceof Dog) {
            Dog myDog = (Dog) myAnimal;  // ✅ 向下轉型
            myDog.fetch();  // ✅ 輸出 "Dog fetches a ball."
        }
    }
}
```

📍020📍泛型（Generics）

泛型（Generics）是 **一種類型參數化機制**，允許編寫程式時，使用 **未來才確定的類型**（允許類別、介面、方法在使用時指定操作的資料型態），並在編譯時進行型態檢查，並在**實例化時** 指定該類型。

- 可在編譯時期就檢查出型別錯誤。
- 取出類別內的資料時不必進行型態轉換，可以直接指定給適當型別的變數
- 允許多個用逗點隔開的參數

▼ 1. **泛型類別：**使用尖括號 `<>` 包住型態參數。型態參數通常是單一的大寫字母（指 `T`, `E`, `K`, `V`），也可以使用其他的名稱。**泛型的參數只能代表類別，不能代表個別物件**。

```
// 定義泛型類別
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>();
        intBox.setValue(100);
        System.out.println(intBox.getValue());  // 輸出 100

        Box<String> strBox = new Box<>();
        strBox.setValue("Hello");
        System.out.println(strBox.getValue());  // 輸出 Hello
    }
}
//在這個範例中，Box<T> 類別被定義為泛型類別，T 可以在使用類別時指定具體的型態，像是 Integer 或 String。
```

▼ 2. **泛型方法：**泛型也可以用在方法上，可以讓方法處理不同型態的參數並返回相應型態的結果。不論方法所在的類別或介面是否定義為泛型，方法都可定義為泛型方法=一般類別也可以定義泛型方法。

```
// 定義泛型方法
public class Main {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"apple", "banana", "cherry"};

        // 呼叫泛型方法
        printArray(intArray);
        printArray(strArray);
    }
}
//這個方法 printArray 可以接受任何型態的陣列，並打印出陣列中的元素。
```

▼ 3. **型態上限（`extends`）：**允許你設定型態參數的上限，泛型類別或方法可以接受某一型態或其子類型，限定泛型的範圍。`T extends Number` 表示型態 `T` 必須是 `Number` 類別或其子類別。

```
// 設定型態上限為 Number
public class NumberBox<T extends Number> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public double getValue() {
        return value.doubleValue();  // 因為 T extends Number，所以可以安全調用 doubleValue()
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        NumberBox<Integer> intBox = new NumberBox<>();
        intBox.setValue(10);
        System.out.println(intBox.getValue());  // 輸出 10.0

        // NumberBox<String> strBox = new NumberBox<>();  // 編譯錯誤：String 不是 Number 的子類別
    }
}
//在這個例子中，NumberBox 類別限制了型態參數 T 必須是 Number 類別或其子類別。這樣可以確保方法中對數字資料的處理
```

▼ 4. **物件型態在 `new` 時決定：**在 Java 中，泛型的型態是 **在編譯時確定的**，而不會在執行時動態決定。然而，使用泛型的物件（例如陣列）在 `new` 時型態才被確定。

```
// 定義泛型方法來創建陣列
public class Main {
    public static <T> T[] createArray(int size, Class<T> clazz) {
        @SuppressWarnings("unchecked")
        T[] array = (T[]) java.lang.reflect.Array.newInstance(clazz, size);
        return array;
    }

    public static void main(String[] args) {
        // 創建一個 Integer 陣列
        Integer[] intArray = createArray(10, Integer.class);
        System.out.println(intArray.length);  // 輸出 10

        // 創建一個 String 陣列
        String[] strArray = createArray(5, String.class);
        System.out.println(strArray.length);  // 輸出 5
    }
}
//在這個範例中，泛型方法 createArray 根據所提供的型態參數 T 和 size 來創建指定型態的陣列。注意，這種情況下 new 操作
```

▼ 5.泛型的通配符（Wildcard，`?`）

通配符 `?` 代表「**未知類型**」，可用於方法參數，允許方法接受不同類型的泛型物件。

上界通配符（`? extends T`）允許傳入 `T` 及其子類別的物件：

下界通配符（`? super T`）允許傳入 `T` 及其父類別的物件：

```
public static void printNumbers(List<? extends Number> list) {
    for (Number n : list) {
        System.out.println(n);
    }
}
public static void addNumbers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
```

- **提升類型安全（Type Safety）**：透過泛型，程式在 **編譯時** 就能確保物件類型的正確性，避免在執行時期才發生類型錯誤。
- **減少類別轉換（Type Casting）**：泛型使我們可以編寫更 **靈活且可讀性更高** 的程式，避免重複撰寫類別轉換的程式碼，減少類型轉換的次數。

📍021📍匿名類別（Anonymous Class）

匿名類別（Anonymous Class） 是 **沒有名稱的內部類別**，在 **產生物件時才定義類別**，通常用於 **簡化程式碼，當場實作介面或繼承類別**。

匿名類別的特點

1. **沒有類別名稱**，只能 **用一次**。
2. **必須繼承某個類別或實作某個介面**。
3. **只能在建立物件時使用** `new` **定義**。
4. **無法有建構子（但可以用初始化區塊** `instance initializer` **來處理）**。

為什麼要用匿名類別？

✔ **簡化程式碼**（不需要額外定義新的類別）。

✔ **適合一次性使用的情境**（如 GUI 事件監聽器）。

✔ **避免程式碼膨脹**，讓程式更精簡。

```
new 父類別或介面() {
    // 類別的定義（覆寫方法）
};
```

▼ 指不用建立新的 class Dog implements Animal，節省程式碼&不需要額外建立 Car extends Vehicle，直接使用匿名類別。

```
interface Animal {
    void makeSound();
}

public class Main {
    public static void main(String[] args) {
        // 使用匿名類別
        Animal dog = new Animal() {
            @Override
            public void makeSound() {
                System.out.println("汪汪汪！");
            }
        };

        dog.makeSound();  // 輸出：汪汪汪！
    }
✅ 優點：不用建立新的 class Dog implements Animal，節省程式碼。
❌ 缺點：如果要重複使用這個類別，就不適合用匿名類別。

abstract class Vehicle {
    abstract void move();
}
public class Main {
    public static void main(String[] args) {
        Vehicle car = new Vehicle() {
            @Override
            void move() {
                System.out.println("車子正在行駛！");
            }
        };

        car.move();  // 輸出：車子正在行駛！
    }
}
✅ 優點：不需要額外建立 Car extends Vehicle，直接使用匿名類別。
```

▼ 指使用匿名類別來處理按鈕點擊事件：這裡使用 **AWT（Abstract Window Toolkit）** 來建立一個簡單的 GUI，當按鈕被點擊時，會在控制台輸出 **「按鈕被點擊了！」**。

```
import java.awt.*;
import java.awt.event.*;

public class AnonymousClassExample {
    public static void main(String[] args) {
        // 建立 Frame（視窗）
        Frame frame = new Frame("匿名類別範例");
        Button button = new Button("點擊我");

        // 使用匿名類別來註冊按鈕的監聽器
        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("按鈕被點擊了！");
            }
        });

        // 設定視窗的基本屬性
        frame.add(button);
        frame.setSize(300, 200);
        frame.setLayout(new FlowLayout());
        frame.setVisible(true);

        // 增加視窗關閉事件，避免程式無法關閉
        frame.addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                frame.dispose();
            }
        });
    }
}
```

- `new ActionListener()` **是一個匿名類別，它實作了** `ActionListener` **介面**。
- `actionPerformed()` 方法會在按鈕點擊時執行。
- `new WindowAdapter()` **是一個匿名類別，它繼承了** `WindowAdapter` **類別**。
- `windowClosing()` 方法在視窗關閉時執行 `frame.dispose();`，避免程式無法結束。

📍022📍匿名類別（Anonymous Class）

Lambda運算式：`（參數）→expression` 。Expression 的值是回傳值。這樣的 Lambda 運算式其實是一種**匿名函式（沒有名稱的方法）**。

多行程式碼（需要 `{}` 和 `return`）：如果 Lambda 需要 **多行程式碼**，就必須加上 `{}` 並且手動 `return` ：

多個參數：如果 Lambda 需要**多個參數**，就要用 `()` 括起來：

參數型別：Lambda **通常可以省略參數型別**，因為 Java 會自動推斷：`BiFunction<Integer, Integer, Integer> add = (Integer a, Integer b) → a + b;`

無參數 Lambda：如果 Lambda 沒有參數，必須寫 `()`：

方式	優點	缺點
匿名類別	簡潔、適合一次性使用	不能重複使用，程式可讀性較低
具名類別	可重複使用、可讀性較高	程式碼變長，可能造成程式碼膨脹

📍023📍函式介面與函數式程式設計（Functional Interface&Functional Programming）

在 **Java 8** 引入 **Lambda 表達式** 之後，函式介面（Functional Interface）成為 Java **函數式程式設計**（Functional Programming）的基礎。

- 函式介面（Functional Interface） 是 **只能有一個抽象方法** 的 介面（Interface），但可以有 `default` 或 `static` 方法。可以使用 `Lambda 表達式` 來簡化程式碼。
- 可以使用 `@FunctionalInterface` 註解，確保介面符合規範。
- **內建函式介面** (`Function`、`Predicate`、`Consumer` 等) 提供常見操作，提升開發效率。
- **函數式程式設計**（Functional Programming）讓 Java **更簡潔、更易讀**，適合處理集合 (`Stream API`)、並行運算等需求。

```
@FunctionalInterface
interface MyFunctionalInterface {
    void doSomething();  // 只有一個抽象方法
}

public class Main {
    public static void main(String[] args) {
        MyFunctionalInterface obj = () -> System.out.println("Hello, Functional Interface!");
        obj.doSomething();
    }
}
```

Java 內建的函式介面

Java **標準函式介面** 位於 `java.util.function`，常見的有：

函式介面	參數	回傳值	說明
`Supplier<T>`	無	`T`	產生並回傳值
`Consumer<T>`	`T`	無	消費輸入，不回傳
`Function<T, R>`	`T`	`R`	轉換 `T` 為
`Predicate<T>`	`T`	`boolean`	判斷條件，回傳 `true/false`

```
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<String, Integer> stringToLength = str -> str.length();
        System.out.println(stringToLength.apply("Hello"));  // 輸出 5
    }
}
```

在 Java 中，stringToLength不是傳統的函式（方法），而是一個 函式
這裡 stringToLength 是 Function<Integer, Integer> 的一個物件，不
所以我們不能用 stringToLength("Hello") 的方式呼叫它，而是要用 物

▼ **為什麼要用 `Function`？**

雖然 `apply()` 這樣的呼叫方式看起來比較麻煩，但它的**彈性**更大！

`Function<T, R>` 讓我們可以：

1. **把函數當作變數來傳遞**（Functional Programming 的核心概念）
2. **方便組合多個函數**（像 `.andThen()`、`.compose()`）
3. **更靈活地處理泛型（Generic）**

舉個例子，我們可以輕鬆 **把兩個函數組合起來**：

```
Function<Integer, Integer> square = num -> nu
Function<Integer, String> toString = num -> "結

// 先平方，再轉字串
Function<Integer, String> squareThenString =

System.out.println(squareThenString.apply(3))
```

📍024📍字串（`String`）

在 Java 中，字串（`String`）被視為不可變（immutable）物件，也就是說，一旦建立字串物件後，該字串的內容就無法被修改。這是 Java 中字串處理的一個重要特性。

1. 字串不可變（Immutable）

- 一旦字串物件被創建，其內容（字元序列）是固定的，不能被修改。
- 當進行字串的修改操作時，實際上是創建了新的字串物件，而原來的字串物件保持不變。
- Ⓜ**性能考量：** 由於字串常常被重複使用，且字串池策略的存在，將字串設為不可變可以有效提高效能和記憶體利用率。
- Ⓜ**安全性：** 不可變的字串可以保證在多線程環境下的安全性，避免了對字串內容的意外修改。
- Ⓜ**一致性：** 字串不可變保證了其內容不會被改變，這對於某些應用場景（例如哈希計算、加密等）是至關重要的。

2. 字串池策略（String Pool）

Java 使用 **字串池**（也稱為字串常數池）來優化字串的儲存，從而節省記憶體。字串池是一個特殊的內存區域，其中存儲了所有使用字串常數（如 `"Hello"`）創建的字串。當遇到相同的字串常數時，Java 會直接使用池中已經存在的字串物件，而不是創建新的字串物件。

字串池的工作原理：

- **字串常數：** 當字串字面量（例如 `"Hello"`）被創建時，它會被存儲在字串池中。
- **重用字串：** 如果程序中後來再次使用相同的字串常數，Java 會直接返回字串池中的那個字串物件，而不是創建一個新的物件。

3. new 操作符和字串池的區別

▼ 如果使用 `new` 關鍵字創建字串，則不會直接引用字串池中的字串，而是會創建一個新的 `String` 物件，即使字串內容相同。

```
String str1 = "Hello";
String str2 = "Hello";
System.out.println(str1 == str2);  // 輸出 true。在這個例子中，雖然 str1 和 str2 是兩個不同的變數，但它們指向字串池中的同
String str3 = new String("Hello");
System.out.println(str1 == str3);  // 輸出 false
```

語法

📍025📍註釋（Annotation）

在 Java 中，**註解 (Annotation)** 是一種提供**額外資訊**的機制，通常用於**編譯器指示、程式碼分析或運行時處理**。

- **註解名稱** 必須是一個**已定義的註解 (Annotation) 類別**。 `@註解名稱(參數1 = 初值1, 參數2 = 初值2)`
- 註解中的**參數 (Element)** 必須有初值，除非該成員已提供**預設值**。
- **註解不需要分號 (;)**，它與一般的 Java 語法不同，在要說明的Java資訊面前加入注釋等說明性資訊。兩者之間除了comment以外不可以寫其他內容。
- 指 `@Override`、`@Deprecated`、`@SuppressWarnings`、`@Target(ElementType.METHOD)`、`@Retention(RetentionPolicy.RUNTIME)`、

▼ 指**自訂註解**：除了 Java 內建的註解，開發者也可以定義自己的註解。

```
@interface MyAnnotation {
    String value(); // 註解的成員
}

@MyAnnotation(value = "Hello")
class Example {}

@interface MyAnnotation {
    String name() default "Default Name";
}
@MyAnnotation
```

```
class Example {}

Java 提供 @Target 來限制註解可用的位置：
@Target(ElementType.METHOD) // 只能用在方法上
註解的保留範圍 (Retention) 指的是註解在哪個階段還存在：
@Retention(RetentionPolicy.RUNTIME)
```

📍026📍Switch case 的常數值（constant values of Switch case）

▼ 在 Java 中，`switch` 的 `case` 後面**不能使用運算式(expression)**、**變數(variables)**或**方法調用（method call）**，只能使用 **常數值（constant values、enum）** 或 **常量變數（`final` 修飾的變數）**。

```
int x = 2;
switch (x) {
   case x:  // ❌ 錯誤！不能用變數
      System.out.println("x matched");
      break;
}
final int VALUE = 2;  // `final` 讓它成為常量
switch (2) {
   case VALUE:  // ✅ 允許
      System.out.println("Matched VALUE");
      break;
}
```

- Java的switch**是基於編譯期的** `lookup table`（查找表）來執行的，而不是在運行時評估表達式。如果允許運算式，每次 `switch` 執行時都要重新計算，影響效能，所以 Java要求 `case` 只能使用編譯時**可確定的值**（constant values）。

📍027📍三種迴圈與continue（Three Loops & continue）

- 在 Java 中，`continue` 用來 **立刻跳過當前迴圈的剩餘程式碼**，並繼續執行下一次的迴圈。這樣可以跳過某些條件下不需要執行的程式碼，繼續下一次的迴圈。

  ▼ 在 `for` 迴圈中，`continue` 會跳過該次迴圈的 **剩餘程式碼**，然後會 **執行遞增條件**（例如 `i++`），接著重新進入迴圈條件判斷部分，決定是否繼續下一次迴圈。

```
for (int i = 0; i < 5; i++) {
if (i == 2) {
continue;  // 當 i 等於 2 時跳過本次迴圈
}
System.out.println(i);  // 當 i == 2 時不會印出
}
0
1
3
4
package ch04.for_stmt;

public class ContinueDemo00 {
   public static void main(String[] args) {
      int x = 0, sum = 0;
      for (x = 1; x <= 10; x++) {
         if (x % 5 == 0) {
            continue;
         }
         sum += x;
```

```
            System.out.println(" x=" + x + ", sum=" + sum);
        }
        System.out.println("Sum=" + sum);
    }
}
```

▼ 在 `while` 迴圈中，`continue` 會跳過當前循環，然後會立刻返回到 **條件判斷** 部分，繼續進行下一次的迴圈。

```
int i = 0;
while (i < 5) {
    i++;  // 遞增條件
    if (i == 3) {
        continue;  // 當 i 等於 3 時跳過本次迴圈
    }
    System.out.println(i);
}
1
2
4
5
package ch04.while_stmt;

public class ContinueDemo01 {
    public static void main(String[] args) {
        int x = 0, sum = 0;
        x = 1;
        while (x <= 10) {
            if (x % 5 == 0) {
                x++; //沒有此行會有無限迴圈。
                continue;
            }
            sum += x;
            System.out.println(" x=" + x + ", sum=" + sum);
            x++;
        }
        System.out.println("Sum=" + sum);
    }
}
```

▼ 在 `do-while` 迴圈中，`continue` 的行為也相似，但要注意 `do-while` **會至少執行一次迴圈體內的程式碼**，即使條件不成立。

```
int i = 0;
do {
    i++;  // 遞增條件
    if (i == 4) {
        continue;  // 當 i 等於 4 時跳過本次迴圈
    }
    System.out.println(i);
} while (i < 5);
1
2
```

```
3
5
```

- `while` & `do-while` 迴圈：要加上遞增條件，不然會進入無限迴圈。

📍028📍 `this` 關鍵字（this）

`this` 關鍵字： `this` 是一個指向當前物件的引用，通常用來區分**實例變數**和局部變數（包括方法或建構子的參數）之間的名稱衝突。`this` 也可以用來呼叫當前物件的其他方法或建構子。

▼ 自我說明的能力：當方法或建構子的參數名稱與類別的實例變數名稱相同時，使用 this 來區分兩者，這樣程式碼更具可讀性。

```
class Car {
    private int speed;  // 實例變數

    // 建構子
    public Car(int speed) {
        this.speed = speed;  // 使用 this 來區分實例變數和參數變數
    }

    public void displaySpeed() {
        System.out.println("Car speed: " + this.speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car(100);  // 傳遞參數到建構子
        car.displaySpeed();      // 輸出：Car speed: 100
    }
}
```

▼ `this` **只能用於建構子，且必須是建構子內的第一個敘述：**在建構子中，`this` 可以用來調用同一類別的其他建構子（即建構子鏈接）。這種語法必須是建構子內的第一個敘述。

```
class Car {
    private int speed;
    private String model;

    // 第一個建構子
    public Car() {
        this(0, "Unknown");  // 呼叫第二個建構子
    }

    // 第二個建構子
    public Car(int speed, String model) {
        this.speed = speed;
        this.model = model;
    }

    public void displayInfo() {
        System.out.println("Car model: " + model + ", Speed: " + speed);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();  // 呼叫第一個建構子，會自動鏈接到第二個建構子
        car1.displayInfo();    // 輸出：Car model: Unknown, Speed: 0

        Car car2 = new Car(120, "Toyota");  // 直接使用第二個建構子
        car2.displayInfo();    // 輸出：Car model: Toyota, Speed: 120
    }
}
```

▼ `this` **不可以在靜態方法內使用**：靜態方法屬於類別本身，不依賴於物件實例，因此無法使用 `this` 來指向當前物件。靜態方法不需要物件實例即可執行。

📍029📍 `final` 關鍵字（`final`）

1️⃣ `final` 類別：**不能被繼承**。如果一個類別被標記為 `final`，則這個類別 不能被繼承，也就是 不能有子類別。

- 防止繼承，確保類別的設計不會被更改（例如 `String` 類別本身就是 `final`）。
- 避免 不必要的多型（polymorphism），提高安全性與可讀性。

2️⃣ `final` 方法：**子類別不能覆寫**。如果某個方法標記為 `final`，則 子類別不能覆寫（override） 這個方法。

- 確保方法行為不變，避免子類別修改關鍵方法（例如 `Object` 類別中的 `getClass()` 方法就是 `final`）。
- 提高安全性，避免惡意或無意的覆寫導致行為異常。

3️⃣ `final` 變數：不可更改數值

- `final` 變數 一旦設定初值後，就 不能再被修改。👉常數（constants），例如 `Math.PI`。確保變數的值不會意外被修改，提高程式安全性。
- `final` 變數 可以是實例變數、類別變數（`static final`）、區域變數（方法內變數）。

🔷 `final` 與 `static` 一起使用。如果 `final` 和 `static` 一起使用，則該變數會成為 常數（constant），所有物件共享同一個值。

🔷 `final` **是唯一能用在區域變數的修飾字**，可以用來保護方法內的變數不被修改。

`final` 用法	作用	限制	例子
類別（class）	**不能被繼承**	不能有子類別	`final class Animal {}`
方法（method）	**不能被覆寫**	子類別不能 `override`	`final void makeSound() {}`
變數（variable）	**不可修改數值**	只能設定一次初值	`final int x = 10;`

📍030📍 可變長度參數（Varargs）

- **可變長度參數(Varargs)**：在 Java 中，**可變長度參數**允許方法接收**可變數量**的**相同類型參數**，語法為：`returnType methodName(type... parameterName)`
  - `type... name` 表示**可變參數**，可以傳入 0 個或多個相同類型的值。
  - 可變參數**必須是方法的最後一個參數**。
  - **一個方法只能有一個可變參數**。

▼ 👉：`int... numbers` 會將所有傳入的數字視為整數陣列 `int[]`。🔷 printNumbers(1, 2, 3); 等價於 printNumbers(new int[]{1, 2, 3});。

```
public class VarargsExample {
    public static void printNumbers(int... numbers) {
        for (int num : numbers) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
```

```
    public static void main(String[] args) {
        printNumbers(1, 2, 3);     // ➡️ 輸出：1 2 3
        printNumbers(10, 20, 30, 40); // ➡️ 輸出：10 20 30 40
        printNumbers();            // ✅ 允許沒有參數，輸出空行
    }
}
```

📍031📍例外處理（Exception Handling）

`Throwable`：(所有例外的祖先)：例外表示程式執行時可能遇到的任何非正常情況而導致程式無法繼續執行。

- `Error`：(錯誤，通常代表 JVM 層級的問題)。開發者通常無法或不應該處理，**不會使用 `try-catch` 處理**，而是**讓 JVM 自行處理**！
  - 指 `OutOfMemoryError`（記憶體不足）、`StackOverflowError`（遞迴太深導致堆疊溢位）
- `Exception`：(異常，代表應用程式層級的錯誤)。開發者處理，**使用 `try-catch` 或 `throws`**
  - `CheckedException`（已檢查例外，需強制處理)：子代不能丟出父代沒有的`CheckedException`，通常都是執行環境異常或是使用者操作不當有關。
    - 指 `IOException`、`SQLException`、`InterruptedException`（執行緒被中斷）
  - `UnCheckedException`(未檢查例外，編譯器不強制處理)
    - `RuntimeException`：(執行時例外，不強制處理）指 `NullPointerException`、`IndexOutOfBoundsException`、`ArithmeticException`、`IllegalArgumentException`

🔷 `RuntimeException`（執行時例外）

`RuntimeException` 及其子類別屬於**未檢查例外（Unchecked Exception)**，不強制要求 `try-catch` 處理，例如：

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `ArithmeticException`（除以 0）

👉 這類例外通常是程式錯誤導致，**應該在開發時避免，而不是使用 `try-catch` 來掩蓋問題！**

🔷 處理法則

✅ 處理法則：`try-catch-finally`**（處理例外)**。我是最終使用者。用try catch finally處理掉。

▼ 當我是**最終使用者**時，應該使用 `try-catch-finally` 來處理：`finally` 一定會執行，適合用來關閉資源（例如資料庫連線、檔案）！

```
try {
    int result = 10 / 0;  // 會拋出 ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("除數不能為 0：" + e.getMessage());
} finally {
    System.out.println("這段程式一定會執行！");
}
```

- 當例外類別有繼承關係時，父代例外類別要寫在子代例外類別之後。

✅ 宣告法則：`throws`**（宣告例外)**。於方法的小括號後使用throw。我不是最終使用者的時候使用。

▼ 當我是**中間的處理者**，但不打算處理例外時，可以在方法後面 `throws`：📌 **這樣的方法呼叫者就要負責處理例外！**

```
void readFile() throws IOException {
    FileReader file = new FileReader("test.txt"); // 可能拋出 IOException
}
```

- 在 Java 中，當子類別覆寫（override）父類別的方法時，**例外處理的規則如下**：
  - **可以拋出與父類別相同或更「窄」的 Checked Exception**（但不能拋出額外的 Checked Exception)。

- **可以拋出 RuntimeException（未檢查例外）**，即使父類別沒有拋出任何例外。

▼ 指**自訂例外：**我們可以自訂例外類別：可以自訂例外類別，讓程式更有彈性。

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
void checkAge(int age) throws MyException {
    if (age < 18) {
        throw new MyException("未滿 18 歲，不可註冊！");
    }
}
```

- 例外處理的重要方法

  `string getMassage()`:傳回例外的簡訊

  `void printstacktrace()`：由發生例外的那一行開始，印出呼叫的堆疊內容，呼叫堆疊紀錄的方法的順序（倒著顯示的），該方法也被稱為堆疊追蹤stack trace。

  `System.exit(int n)`參數n為return code。是java傳給作業系統的一個整數值，

📍032📍例外處理（`try-catch-finally`）

`try-catch-finally` 是 Java 例外處理機制中用來處理和控制異常的關鍵組件。這三個區塊必須按照順序排列，中間不能有其他敘述，並且可以確保程式在遇到錯誤時仍然能夠優雅地結束執行。

1 try 區塊

- **目的：** `try` 區塊中放置可能會引發例外的程式碼。
- **語法：** 當程式碼運行至這裡時，會檢查是否有例外拋出。如果有例外，會跳轉到對應的 `catch` 區塊進行處理。

2 catch 區塊

- **目的：** 用來捕獲 `try` 區塊中拋出的例外，並對其進行處理。可以有零個或多個 `catch` 區塊。
- **語法：** 根據 `try` 中的例外型別來處理。每個 `catch` 區塊必須指定捕獲的例外類型。

⚠️ **多個 `catch` 區塊：**當有多種可能的例外型別時，可以使用多個 `catch` 區塊來捕獲：

3 finally 區塊

- **目的：** 用來執行無論是否發生例外都必須執行的程式碼，通常用於釋放資源，如關閉檔案、釋放資料庫連線等。
- **語法：** `finally` 區塊可以有零個或多個。即使 `catch` 中有處理例外，或是根本沒有catch，`finally` 仍然會被執行。當沒有catch 區塊，finally區塊必須出現。

⚠️ **特別情況**：(1)當 `catch` **沒有捕獲到例外時，** `finally` **仍會執行。**(2)即使**在** `catch` **或** `finally` **內部有** `return`**，** `finally` **區塊仍然會被執行。**

▼ **使用** `finally` **的重要性：** `finally` 區塊確保即使發生了異常，程式也能安全地清理資源或執行其他必要的操作。這是資源管理（如關閉檔案、釋放資料庫連線等）中非常重要的一環。

```
FileReader fr = null;
try {
    fr = new FileReader("example.txt");
    // 执行一些文件操作
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (fr != null) {
```

```
        try {
            fr.close(); // 保證檔案一定會被關閉
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

📍033📍例外處理（Try-With-Resources,TWR ）

Try-With-Resources（TWR） 是 Java 7 引入的一種 **自動管理資源** 的機制，它能讓 **實作** `AutoCloseable` **介面的資源** 在 `try` 區塊結束後 **自動關閉**，不用寫 `finally` 來手動釋放資源。

```
try (資源定義與初始化) {
    // 使用資源
} catch (例外處理) {
    // 例外處理區塊（可選）
}
// 無需 finally 手動關閉資源
```

💡 **核心概念**

1. `try()` **小括號內宣告資源**，這些資源 **必須** 實作 `AutoCloseable` 或 `Closeable` 介面。
2. **程式區塊結束後，資源會自動關閉**（不論是否發生例外）。
3. **可以省略** `finally` **手動關閉資源**，讓程式更簡潔。

▼ 指**自動關閉檔案資源**

- `BufferedReader` 和 `FileReader` **都有實作** `AutoCloseable` **介面**，所以可以在 Try-With-Resources (`try() {}`) 中使用，並且 **不需要手動呼叫** `close()`，因為 `try` 結束時會自動關閉資源。
- `BufferedReader` **實作** `AutoCloseable`，放進 `try()` 小括號裡時，當 `try` 區塊執行完畢，Java **會自動呼叫** `br.close()`，所以 **不用手動寫** `br.close()` ！

```
import java.io.*;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        // br 會在 try 區塊結束後自動關閉
    }
}
```

其他

📍034📍存取修飾符（Access Modifiers）

- **Java 的修飾字順序沒有固定要求**，例如 `public static` 和 `static public` 作用相同。但實務上，**通常會先寫存取修飾詞（如** `public`、`private`**），然後再寫** `static`、`final` **等**，這樣可讀性較高。
- 在 Java 中，存取修飾符（Access Modifiers）用來**控制類別、方法與變數的可見性**，也就是哪些類別可以存取這些成員。Java 提供四種存取等級：
- Class前面只能加 public 或都不寫（default)

存取修飾符	同一類別	同一 package	不同 package (子類別)	不同 package (非子類別)	
`public`	✅ 可存取	✅ 可存取	✅ 可存取	✅ 可存取	工具類（Utility Classes）、`main()` 方法
`protected`（姪子可見）	✅ 可存取	✅ 可存取	✅ 可存取	❌ 無法存取	允許子類別存取，但不希望非子類別的類別存取它。
`default`(包內可見)	✅ 可存取	✅ 可存取	❌ 無法存取	❌ 無法存取	不希望被其他 package 存取。適合不需要被外部存取的 helper class 或 method。
`private`	✅ 可存取	❌ 無法存取	❌ 無法存取	❌ 無法存取	變數應該封裝（Encapsulation），只能透過 public 方法存取（getter、setter）。

📍035📍建構子（Constructor）

- 建構子（Constructor）：在 Java 中，建構子是一種特殊的方法，用來建立物件並初始化屬性，使用建構子可以以較簡潔的方式設定物件的初值。

1. **名稱必須與類別名稱相同**：讓程式碼更簡潔，避免手動初始化。
2. **沒有回傳值，也不能寫** `void`。
3. **用來初始化物件的屬性**：確保物件建立時，所有必要的屬性都已經設定好。
4. **建構子多載Constructor Overloading：**可以在同一個類別中定義**多個建構子**，但**參數的型態、數量或順序必須不同**。
5. **預設建構子（Default Constructor）：**如果沒有明確定義建構子，Java 會自動提供一個沒有參數的預設建構子。這個隱含建構子**不會初始化任何屬性**，只是讓 `new Animal()` 不會出錯。如果類別**有手動定義建構子**，Java 就不會再自動生成預設建構子：

指 `Person(String name, int age)` (初始化人物資訊)、 `Car(String brand, int year)` (不同方式建立汽車物件)、 `DatabaseConnection(String url,String user,String password)` (建立資料庫連線)

📍036📍不會執行的程式碼（Unreachable code）

- 當 Java 編譯器 **確定某段程式碼永遠無法執行**，會產生 `Unreachable code` 錯誤。指(1) `return`、 `throw` 之後的程式碼(2) `while(true)` 但沒有 `break`(3) `if (false)` 代表 **這個條件永遠不成立**

📍037📍冗餘程式碼（Redundant Code）

- **冗餘程式碼，Redundant Code：**撰寫 **多餘的程式碼**不僅會影響程式的執行效能，還會 **消耗額外的計算資源**，導致不必要的 **電池電量消耗**，這在 **行動裝置**（如手機、平板）上特別重要。

1. **佔用 CPU 資源（Consumes More CPU Cycles）**
   - **多餘的迴圈、計算、條件判斷**會讓處理器（CPU）執行額外的運算，增加電量消耗。
   - 手機的 CPU 會根據運算量動態調整時脈（clock speed），運算越多，功耗越大。
2. **影響記憶體與垃圾回收（Consumes More RAM & Triggers Garbage Collection）**
   - **建立過多物件** 會增加 記憶體（RAM） 使用量，當記憶體不足時，**Java 垃圾回收機制（Garbage Collection, GC）** 會啟動，進一步消耗 CPU 和電量。
   - **解決方案：使用字串池（String Pool）**或避免多餘物件. `String s = "Hello"; // 這樣會直接從字串池取得，不會重複創建新物件`
3. **不必要的迴圈 & 過度刷新 UI**（Unnecessary Loops & Excessive UI Updates）
   - 在手機 App（Android / iOS）中，頻繁刷新 UI 會耗費 GPU / CPU 能力，導致電量快速消耗。
   - 不必要的迴圈、背景執行緒 會讓 App 持續消耗資源，即使使用者沒在操作手機。**解決方案：加上條件，避免無限更新**