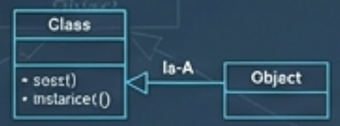
歡迎來到 ITP4510：資料結構與演算法的 OOP 核心

從零開始的物件導向大師課 (Masterclass in OOP Concepts)

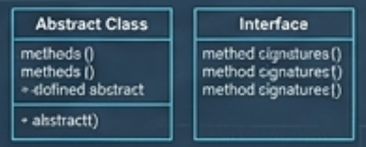
課程核心架構 (The Syllabus)

1. 基礎與關聯 (The Foundation): Class, Object, & Is-A

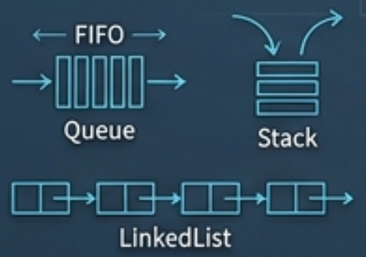


2. 變形金剛 (Polymorphism): 同一指令，多種型態

3. 藍圖與契約 (The Rulemakers): Abstract Class vs Interface



4. 實戰應用 (DSA Application): Queue, Stack, & LinkedList



教授的學習承諾 (Professor's Promise)

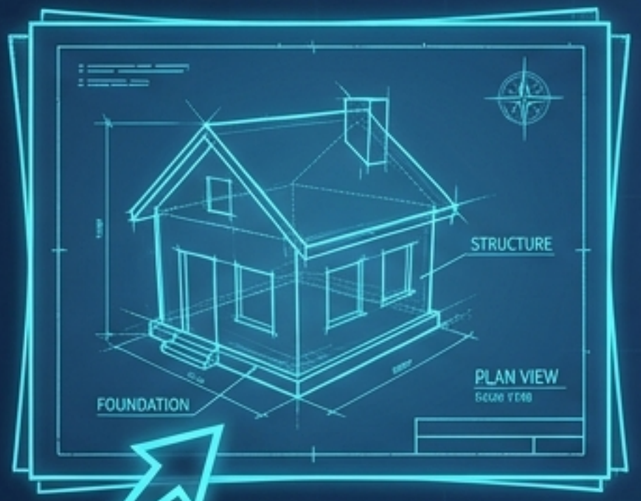
「學習程式語言就像學習如何當一個造物主。今天，我們不只要看懂投影片，我們要拆解背後的邏輯。如果你不能用最簡單的話把概念解釋給一個八歲小孩聽，你就不算真正懂了。今天，我們要把複雜的 Java 概念，變成你直覺的一部分。」

Feynman Analogy Box

課前補給：物件與類別 (Objects & Classes) PPT 未明說，但你必須懂的底層邏輯 (The Unspoken Foundation)
類別 (Class) 是一張建築藍圖 (Blueprint)；物件 (Object) 則是依照藍圖蓋出來的真實房子 (House)。
藍圖不能住人，實體化的房子才可以。

類別 (Class)

- 定義了資料的**屬性** (Attributes) 與**行為** (Behaviors)。
- 教授提點：造物主規定了「人」有名字、能走路。




```
// 1. 藍圖 (Class Blueprint)
public class Person {
    String name;          // Attribute
    public void walk() { ... } // Behavior
}
```

物件 (Object / Instance)

- 類別在記憶體中的具體化產物。必須使用 new 關鍵字來誕生。
- 教授提點：世界上有幾十億個具體的「人」，每一個都是獨立的 Object。

```
// 2. 實體化 (Instantiation)
Person p1 = new Person(); // p1 就是一個具體的 Object
```

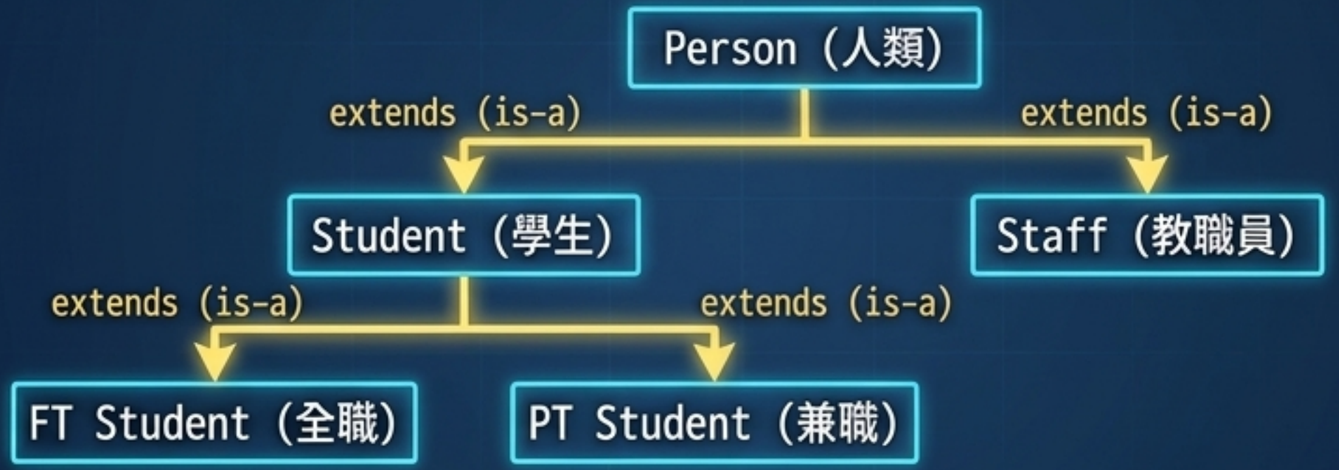
THE ARCHITECT'S BLUEPRINT GLOBAL STYLE

Feynman Analogy Box

繼承與關聯："Is-A" Relationship

站在巨人的肩膀上（Standing on the Shoulders of Giants）

核心法則（Core Rule）：
In an "is-a" relationship, an object of a subclass type may also be treated as an object of its superclass type.（子類別可以被視為父類別的一種）



教授的費曼解析（Professor's Feynman Breakdown）：

"Is-A"（繼承 Inheritance）：
Student is a Person.（學生是人 ➡ 使用 extends 繼承屬性）。

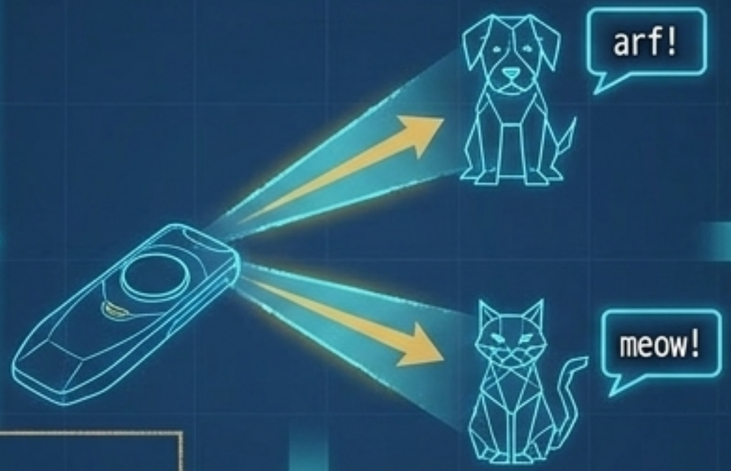
"Has-A"（組合 Composition）：
Student has a Backpack.（學生有背包 ➡ 這是變數包含，把 Backpack 放在 Student 類別裡）。

記憶點（Memory Hook）： 只要你在現實中能說出 "A 是 B 的一種"，在 Java 裡就可以大膽使用 extends！

Feynman Analogy Box

多型（Polymorphism）：千變萬化的神經網絡

同一個指令，各自表述（One Interface, Multiple Implementations）



想像你有一個「萬用遙控器」，上面只有一個按鈕叫 Speak()。你對著狗按，牠會「汪汪」；對著貓按，牠會「喵喵」遙控器不關心對象是誰，它只負責發送指令，對象會根據自己的天性（型態）做出反應。這就是多型（Polymorphism）！

```
// 1. 建立階層 (The Setup)
abstract class Animal {
  public abstract void Speak(); } // 遙控器按鈕
}
class Dog extends Animal {
  public void Speak() { System.out.println("arf!"); } }
}
class Cat extends Animal {
  public void Speak() { System.out.println("meow!"); } }
}
```

```
// 2. 展現魔術 (The Magic)
Animal a[] = new Animal[2]; // 宣告一個「動物」陣列 (父類別參考)
a[0] = new Dog(); // 裝入狗實體 (子類別 Object)
a[1] = new Cat(); // 裝入貓實體 (子類別 Object)

a[0].Speak();  // 輸出: arf! (Dynamic Binding 動態綁定)
a[1].Speak();  // 輸出: meow!
```

為什麼這很重要？(Why it matters?) 我們不需要寫 dogSpeak() 或 catSpeak()。只要呼叫 Speak()，Java 會在執行期間（Runtime）自動決定執行哪個版本！

Architect's Blueprint

抽象類別 (Abstract Classes)：半成品的藍圖

定義基礎，保留彈性 (Defining the Base, Leaving Blanks)

Feynman Analogy Box:

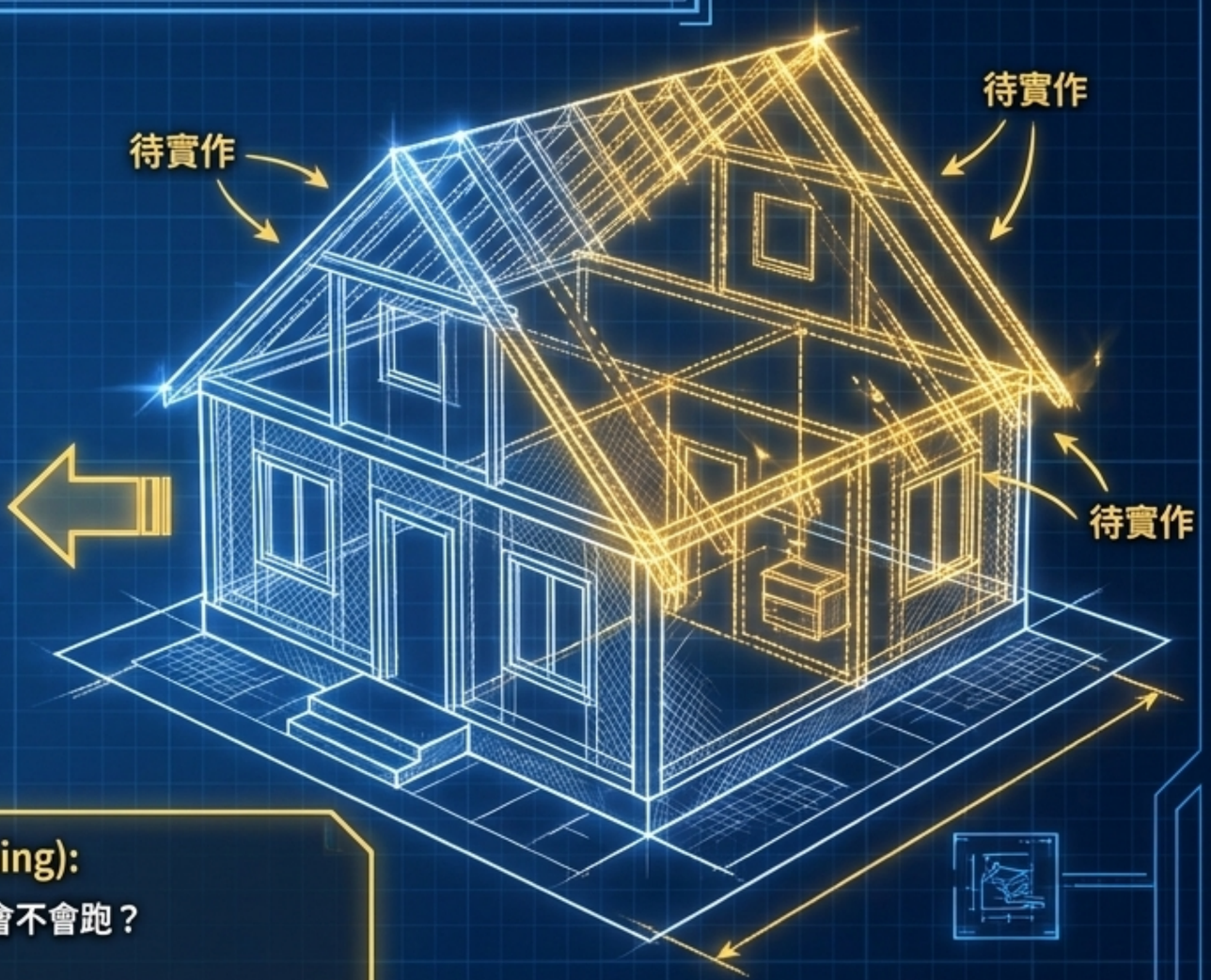
Abstract Class (抽象類別) 就像是建商的「毛胚屋」。它幫你建好了牆壁和水管 (具體變數/方法)，但「裝潢設計」它留空了 (抽象方法)，強迫買家 (子類別) 必須自己決定怎麼蓋。

Exam Case Breakdown: abstract class Eatable

```
abstract class Eatable { // ✅ 具體屬性 (已經蓋好的牆壁)
    String name;
    double price;

    public Eatable(String name, double price) { … } // ✅ 具體建構子

    // ❌ 抽象方法 (毛胚屋留白，強迫子類別實作)
    public abstract double getPrice();
}
```



教授的考試陷阱提醒 (Professor's Exam Trap Warning):

Question: `Eatable e1 = new Eatable("Burger", 50);` 會不會跑？

Answer: 絕對不會 (Compilation Error)！

Why? 因為 Eatable 是抽象的 (abstract)，你不能直接使用 new 創造毛胚屋的實體。你必須創建它的子類別 (如 class Drinks extends Eatable) 才能實體化！

介面 (Interface)：鐵一般的合約

沒有實作，只有純粹的規範 (A Pure Contract of Behavior)

Feynman Analogy Box

如果抽象類別是毛胚屋，那 介面 (Interface) 是一張「政府法規許可證」。它不提供任何牆壁，它只寫著：「只要你想成為一輛車，你必須要有 move() 的功能」。你怎麼做我不管，但你一定要有！

1. 宣告規則 (The Rules)：

- 所有方法預設為 public abstract (沒有 {} 內容)。
- 一個類別可以 實作多個 (implements multiple) 介面，突破 Java 的單一繼承限制！

2. 契約簽署 (Exam Case: Moveable)

```
// 1. 制定契約 (The Contract)
interface Moveable {
    void moveTo(int x, int y, double speed); // 隱含 public abstract
}
```

```
// 2. 簽署並履行契約 (Implementing the Contract)
abstract class Human implements Moveable {
    // 必須實作 moveTo()，否則 Human 必須宣告為 abstract
    public void moveTo(int x, int y, double speed) {
        this.locationX = x; // 具體實作 (Implementation)
    }
}
```

終極對決：Abstract Class vs. Interface

教授的獨家診斷表 (The Ultimate Diagnostic Matrix)

「何時該用 Abstract Class，何時用 Interface？記住這句話：Abstract Class 定義了你是什麼 (What you are)，Interface 定義了你能做什麼 (What you can do)。」

比較維度 (Dimension)	Abstract Class (抽象類別)	Interface (介面)
1. 設計意義 (Meaning)	Is-a (是一個...) 例：狗是一隻動物	Can-do (有能力做...) 例：狗可以移動 (Moveable)
2. 繼承限制 (Inheritance)	只能 extends 單一 (Single) 類別	可以 implements 多個 (Multiple) 介面
3. 變數狀態 (Variables)	可以有各種變數與狀態	只能是常數 (public static final)
4. 建構子 (Constructors)	有 (供子類別呼叫 super())	沒有 (它純粹是規範)

完美結合：實戰範例 (Real Exam Synthesis):

```
// 狗「是」動物，且「有能力」生活
class Dog extends Animal implements Live { ... }
```

「當我們說需要一個 Queue，其實是在要求一份行為契約。我們不在乎底層用 Array 還是 Linked List，只要遵守『先入先出 (FIFO)』即可。這就是 OOP 的 封裝 (Encapsulation)。」

OOP 在資料結構的體現：介面合約

為什麼資料結構需要 OOP？(Why does DSA need OOP?)

The Data Structure Contract (Exam Case: Queue Interface)

使用者 (User)：只看合約，知道可以用 enqueue() 把資料排隊。

```
interface Queue {
   public abstract boolean isEmpty();
   public abstract int size();
   public abstract void enqueue(Object item)
                                        throws QueueFullException;
   public abstract Object dequeue() throws QueueEmptyException;
}
```

The User's World

Abstraction Barrier

The Implementer's World

Hidden Implementation Details (LinkedQueue, ArrayQueue)

實作者 (Implementer)：必須寫出類別去 implements Queue，隱藏底層複雜度。

鏈結串列 (Linked List) 的 OOP 解構

Composition 的完美示範 (The Anatomy of a Node)

「用 OOP 的 X 光機，透視最常考的 LinkedList。它不是連續的記憶體，而是透過 Object Reference (物件參考) 互相牽手串接起來的。」

1. 節點的自我牽手 (The "Has-A" Relationship):

```
class ListNode {
    int roomID;       // Node "has-a" data
    ListNode next;    // Node "has-a" reference to another Node!

    public ListNode(int roomID, ListNode next) { ... }
}
```

Feynman Analogy:

就像一群人排隊，每個人用一隻手 (next) 抓住下一個人的衣服。

2. 記憶體視角 (Memory View):

LinkedList 類別本身不是 Node，它透過 head 和 tail 變數來「管理」整條鏈。

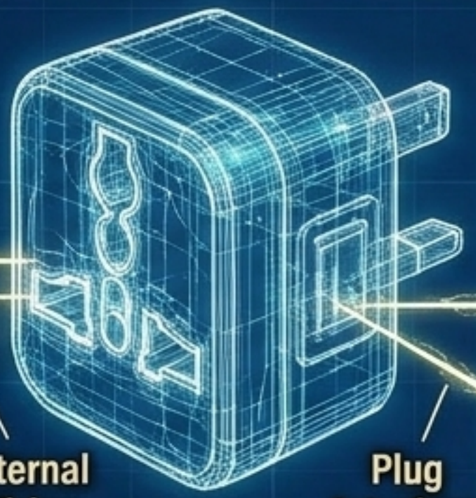
高階應用：用繼承 (Inheritance) 打造資料結構

一秒變身 Stack (Building a Stack from a Linked List)

「魔王題：寫好了一個完美的 LinkedList，該如何最快做成一個 Stack (堆疊)？答案是：利用 extends 借用力量！」

父類別引擎 (The Engine):

```
class LinkedList {
    int roomIID;
    public void addToHead(Object item) { ... }
    public Object removeFromHead()
            throws EmptyListException { ... }
}
```



子類別轉接器 (The Adapter - Exam B1 Case):

```
class ListStack extends LinkedList {
    public ListStack() { super(); } // 呼叫父類別建構子

    // Stack 的 LIFO 行為，直接轉接給 List 的頭部操作！
    public void push(Object item) {
        addToHead(item);   // 隱藏複雜度，重用程式碼！
    }
    public Object pop() {
        return removeFromHead();
    }
}
```

費曼的頓悟 (The Aha Moment):

這叫做 **Adapter Pattern (轉接器模式)**。我們不需要重寫指標 (pointers) 邏輯，OOP 讓程式碼具備 **極度重用性 (Highly Reusable)**！

教授的總結與複習 (The Grand Synthesis)

將藍圖化為現實 (Bringing It All Together)

The OOP to DSA Roadmap

1. 物件 (Objects) ➜
創造了 ListNode 實體來儲存資料。

2. 組合 (Composition) ➜
Node 內部包含另一個 Node 的參考，形成鏈結。

3. 介面 (Interfaces) ➜
制定 Queue 與 Stack 的標準操作合約。

4. 多型 (Polymorphism) ➜
允許 Queue q = new LinkedQueue();，彈性替換底層實作。

5. 繼承 (Inheritance) ➜
讓 ListStack 瞬間擁有 LinkedList 的能力，免去重工。

🔥 考試高分秘訣 (Exam Pro-Tips):

- 遇到 "Write the interface..." ➜ 只寫 public abstract 方法，結尾加分號 ;，絕對不要寫大括號 {}！
- 遇到 "Write the abstract class..." ➜ 可以寫建構子 (Constructor)，可以宣告實體變數！
- 遇到 "Extends vs Implements..." ➜ 先問自己：這是 "Is-a" (是一種) 還是 "Can-do" (能做什麼)？