

CHƯƠNG 3

KẾ THỪA & ĐA HÌNH – TRÁI TIM CỦA OOP

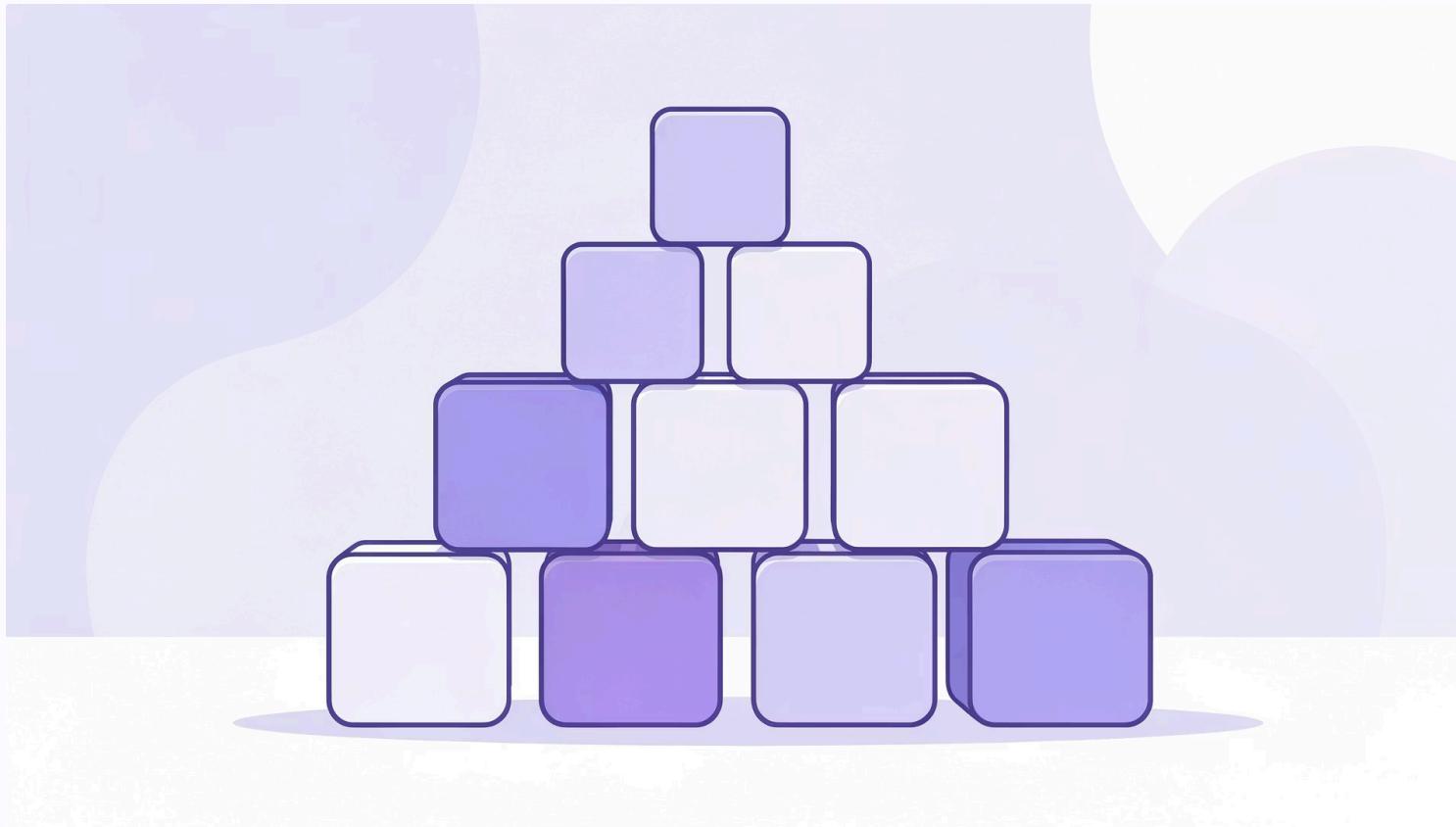
Nắm vững Inheritance và Polymorphism – nền tảng xây dựng hệ thống phần mềm linh hoạt và dễ bảo trì.

Chào mừng bạn đến với chương học quan trọng nhất trong OOP! Inheritance và Polymorphism không chỉ là hai khái niệm riêng lẻ, mà chính là nền tảng giúp bạn xây dựng các hệ thống phần mềm linh hoạt, dễ mở rộng và bảo trì.

Mục tiêu học tập

Sau khi hoàn thành chương này, bạn sẽ nắm vững những kỹ năng thiết yếu:

- Hiểu sâu về Inheritance (Kế thừa) và biết khi nào nên áp dụng
- Phân biệt rõ ràng Overriding và Overloading
- Sử dụng Abstract Class và Interface một cách chính xác
- Nắm vững Polymorphism và cơ chế Dynamic Binding
- Áp dụng nguyên tắc Composition over Inheritance trong thiết kế
- Loại bỏ các câu lệnh if/else phức tạp bằng Polymorphism

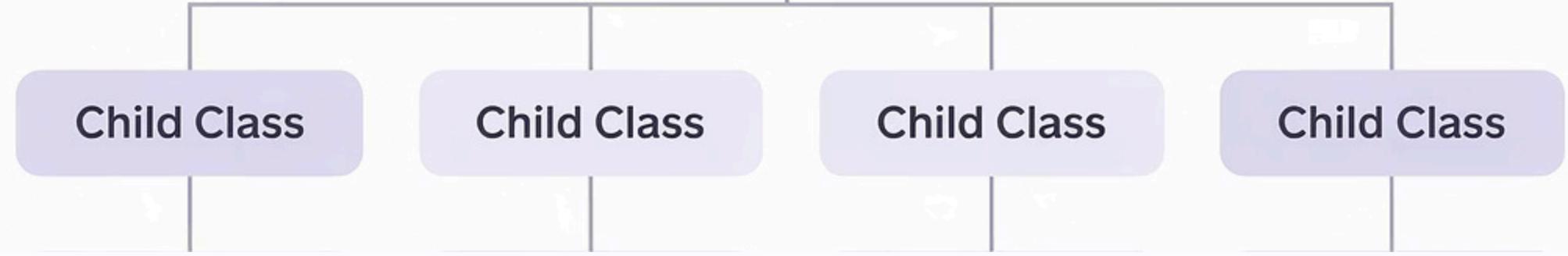


Kiến Thức Cần Có Trước Khi Bắt Đầu

Chương này xây dựng trên nền tảng vững chắc từ Chương 2. Hãy đảm bảo bạn đã nắm vững các khái niệm sau đây:

 Class và Object Chương 2, phần 2.1 <ul style="list-style-type: none">Hiểu rõ class là blueprint, object là instanceBiết cách tạo object với từ khóa <code>new</code>Nắm vững fields và methods	 Access Modifiers Chương 2, phần 2.2 <ul style="list-style-type: none">Hiểu rõ <code>private</code>, <code>protected</code>, <code>public</code>Biết khi nào nên sử dụng modifier nàoNắm được phạm vi truy cập của từng loại
 Constructors Chương 2, phần 2.3 <ul style="list-style-type: none">Cách định nghĩa constructorConstructor overloadingSử dụng từ khóa <code>this</code>	 Methods Cơ Bản Chương 2, phần 2.3 <ul style="list-style-type: none">Cách định nghĩa và gọi methodHiểu về tham số và return typeMethod signature và scope

 **Lưu ý quan trọng:** Chương 3 là **trái tim của OOP**. Nếu bạn chưa tự tin với các kiến thức trong Chương 2, hãy dành thời gian ôn tập trước khi tiếp tục. Nền tảng vững chắc sẽ giúp bạn tiếp thu kiến thức mới hiệu quả hơn rất nhiều!



PHẦN 3.1

Kế Thừa (Inheritance) – Quan Hệ "IS-A"

Inheritance là một trong những trụ cột quan trọng nhất của Lập trình hướng đối tượng (OOP). Cơ chế này cho phép một lớp (subclass) kế thừa các đặc điểm và hành vi từ một lớp khác (superclass).

💡 Inheritance là gì?

Inheritance (Kế thừa) là một trong những trụ cột quan trọng nhất của OOP. Đây là cơ chế cho phép một class (gọi là subclass hay child class) kế thừa toàn bộ các đặc điểm và hành vi từ một class khác (gọi là superclass hay parent class).

Quan hệ "IS-A"

Inheritance thể hiện quan hệ "LÀ MỘT" giữa các class:

- Dog **LÀ MỘT** Animal
- Car **LÀ MỘT** Vehicle
- Student **LÀ MỘT** Person

Tưởng tượng: Di truyền DNA

Superclass (Cha): Truyền lại bộ gen (fields/methods) cho con.

Subclass (Con): Nhận bộ gen, có thể giữ nguyên hoặc biến đổi (override) để phù hợp với môi trường sống. Con cũng có thể phát triển thêm những đặc điểm mới mà cha không có.

✓ Lợi ích của Inheritance

Tái sử dụng Code

Không cần viết lại code đã có ở superclass. Giảm thiểu duplicate code và tăng tốc độ phát triển.

Dễ Bảo Trì

Khi sửa đổi superclass, tất cả subclass tự động được cập nhật. Thay đổi một lần, áp dụng nhiều nơi.

Mô Hình Hóa Thực Tế

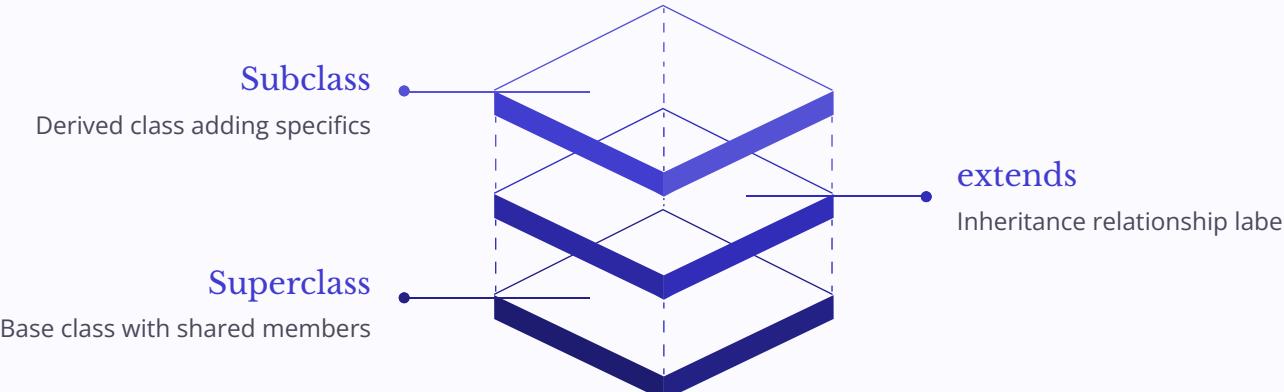
Phản ánh chính xác các quan hệ trong thế giới thực. Code trở nên trực quan và dễ hiểu hơn.

Cú Pháp extends trong Java

Cú pháp cơ bản

Để tạo quan hệ kế thừa trong Java, chúng ta sử dụng từ khóa extends:

```
class Superclass {  
    // Fields và methods của superclass  
}  
  
class Subclass extends Superclass {  
    // Fields và methods riêng của subclass  
    // + Kế thừa tất cả từ Superclass  
}
```



- 💡 **Khái niệm kế thừa:** Khi một Subclass extends một Superclass, nó sẽ tự động kế thừa tất cả các trường (fields) và phương thức (methods) không phải là private của Superclass. Điều này giúp tái sử dụng mã và thiết lập mối quan hệ "IS-A" (ví dụ: Dog IS-A Animal).

Ví dụ thực tế: Animal và Dog

Hãy xem một ví dụ cụ thể để hiểu rõ hơn về cách hoạt động của inheritance:

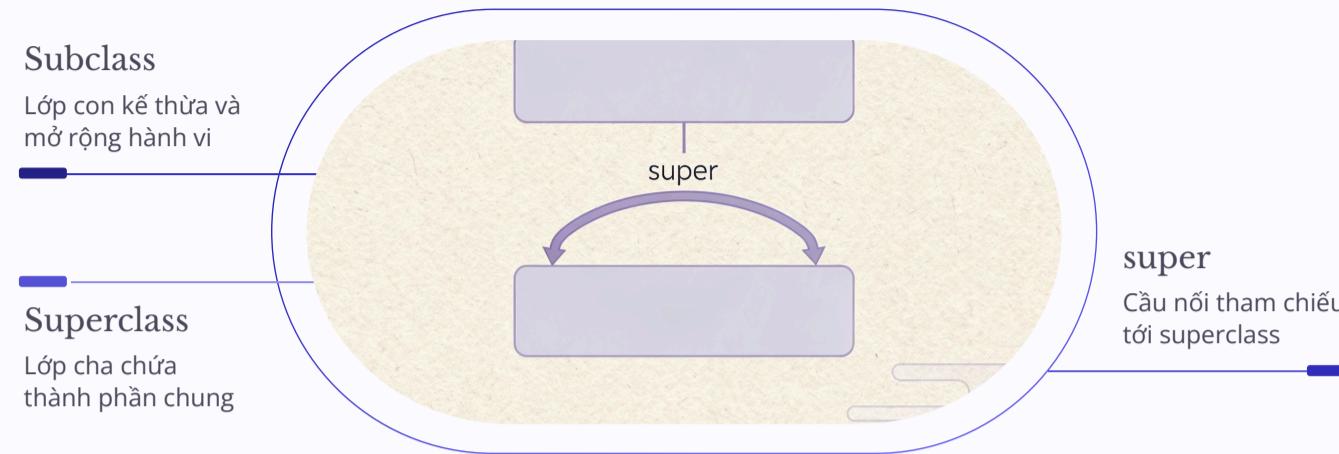
```
// Superclass (Parent class)  
public class Animal {  
    protected String name;  
    protected int age;  
  
    public Animal(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public void eat() {  
        System.out.println(name + " is eating");  
    }  
  
    public void sleep() {  
        System.out.println(name + " is sleeping");  
    }  
}  
  
// Subclass (Child class)  
public class Dog extends Animal {  
    private String breed;  
  
    public Dog(String name, int age, String breed) {  
        super(name, age); // Gọi constructor của Animal để khởi tạo name và age  
        this.breed = breed;  
    }  
  
    // Method riêng của Dog  
    public void bark() {  
        System.out.println(name + " is barking: Woof! Woof!");  
    }  
  
    // Override hoặc thêm phương thức để hiển thị thông tin cụ thể của Dog  
    public void displayInfo() {  
        System.out.println("Dog: " + name +  
            ", Age: " + age +  
            ", Breed: " + breed);  
    }  
}
```

```
// Sử dụng các lớp  
public class Main {  
    public static void main(String[] args) {  
        Dog dog = new Dog("Buddy", 3, "Golden Retriever");  
        dog.eat(); // Kế thừa từ Animal  
        dog.sleep(); // Kế thừa từ Animal  
        dog.bark(); // Method riêng của Dog  
        dog.displayInfo(); // Method riêng của Dog (hoặc override)  
    }  
}
```

- 💡 **Quan sát quan trọng:** Class Dog không cần định nghĩa lại eat() và sleep(). Nó tự động kế thừa từ Animal và có thể sử dụng ngay. Đây chính là sức mạnh của inheritance, giúp tái sử dụng code và dễ dàng mở rộng!

Từ Khóa super – Giao Tiếp với Superclass

`super` là một reference đặc biệt trong Java, trỏ đến superclass (class cha) của object hiện tại. Nó là cầu nối giúp subclass tương tác với superclass.



1. Gọi Constructor của Superclass

Đây là cách sử dụng phổ biến nhất của `super`:

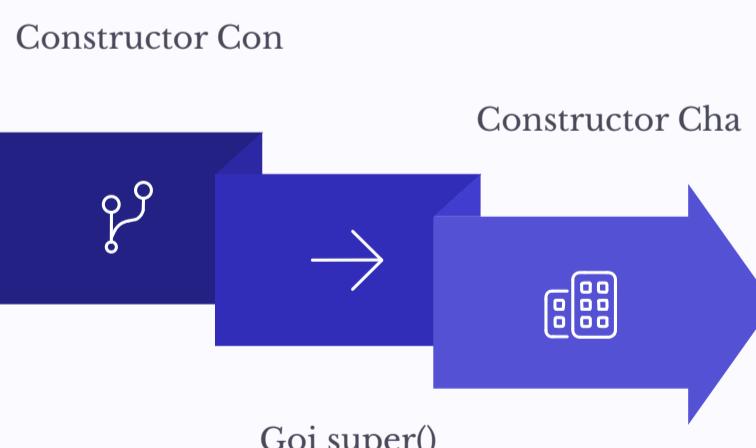
```
// Superclass (Parent class)
public class Animal {
    protected String name;
    protected int age;

    public Animal(String name, int age) { // Constructor của Superclass
        this.name = name;
        this.age = age;
    }
}

// Subclass (Child class)
public class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age); // Gọi constructor của Animal để khởi tạo name và age
        this.breed = breed;
    }
}
```

Khi một object của `Dog` được tạo, constructor của `Dog` sẽ gọi constructor của `Animal` thông qua `super(name, age)` để khởi tạo các thuộc tính kế thừa.



⚠ Quy tắc quan trọng

`super()` phải là dòng đầu tiên trong constructor của subclass.

🔄 Gọi tự động

Nếu bạn không gọi `super()` rõ ràng, Java sẽ tự động gọi `super()` (constructor không tham số) của superclass.

❗ Lưu ý

Nếu superclass không có constructor không tham số, bạn **bắt buộc** phải gọi `super(...)` với tham số phù hợp.

☒ ✗ Lỗi thường gặp: Quên gọi super()

Nếu Superclass `Animal` chỉ có constructor có tham số (`Animal(String name, int age)`) mà không có constructor không tham số, và bạn quên gọi `super(name, age)` trong constructor của `Dog`, trình biên dịch sẽ báo lỗi. Điều này là do Java cố gắng tự động gọi `super()` mặc định, nhưng không tìm thấy constructor không tham số trong `Animal`.

Các Cách Sử Dụng super (Tiếp Theo)

2. Truy cập Fields và Methods của Superclass

Ngoài việc gọi constructor, từ khóa super còn cho phép subclass truy cập trực tiếp vào các thuộc tính (fields) và phương thức (methods) của superclass, đặc biệt hữu ích khi có sự trùng tên hoặc ghi đè.



2.1. Truy cập Field của Superclass

Bạn có thể sử dụng `super.fieldName` để truy cập một field của superclass, ngay cả khi subclass có một field cùng tên (shadowing).

```
// Superclass
public class Animal {
    protected String name = "Generic Animal"; // Field của Superclass

    public String getName() {
        return name;
    }

// Subclass
public class Dog extends Animal {
    protected String name = "Buddy the Dog"; // Field của Subclass (che phủ field của Superclass)

    public void displayNames() {
        System.out.println("Subclass's name: " + this.name); // Truy cập field của Dog
        System.out.println("Superclass's name: " + super.name); // Truy cập field của Animal
    }

    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.displayNames();
    }
}
```

Kết quả chạy ví dụ trên:

Subclass's name: Buddy the Dog
Superclass's name: Generic Animal



2.2. Gọi Method của Superclass

`super.methodName()` cho phép bạn gọi phiên bản method của superclass, điều này rất quan trọng khi subclass đã ghi đè (override) method đó.

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

    public void eat() {
        System.out.println("Animal is eating.");
    }
}

// Subclass
public class Cat extends Animal {
    @Override // Ghi đè phương thức makeSound()
    public void makeSound() {
        System.out.println("Meow!");
    }

    public void performActions() {
        // Gọi phương thức makeSound() của Cat (đã bị ghi đè)
        System.out.println("--- Calling subclass's method ---");
        makeSound(); // hoặc this.makeSound();

        // Gọi phương thức makeSound() của Animal (phiên bản của superclass)
        System.out.println("--- Calling superclass's method ---");
        super.makeSound();
    }

    // Gọi phương thức eat() của Animal (không bị ghi đè)
    System.out.println("--- Calling inherited method ---");
    eat(); // hoặc super.eat(); hoặc this.eat();
}

public static void main(String[] args) {
    Cat myCat = new Cat();
    myCat.performActions();
}
```

Kết quả chạy ví dụ trên:

--- Calling subclass's method ---
Meow!
--- Calling superclass's method ---
Animal makes a sound
--- Calling inherited method ---
Animal is eating.

Trong ví dụ Cat, khi bạn gọi `makeSound()` (hoặc `this.makeSound()`), Java sẽ thực thi phiên bản đã ghi đè trong lớp Cat. Tuy nhiên, khi bạn cần truy cập cụ thể phiên bản method của lớp cha, bạn phải sử dụng `super.makeSound()`.

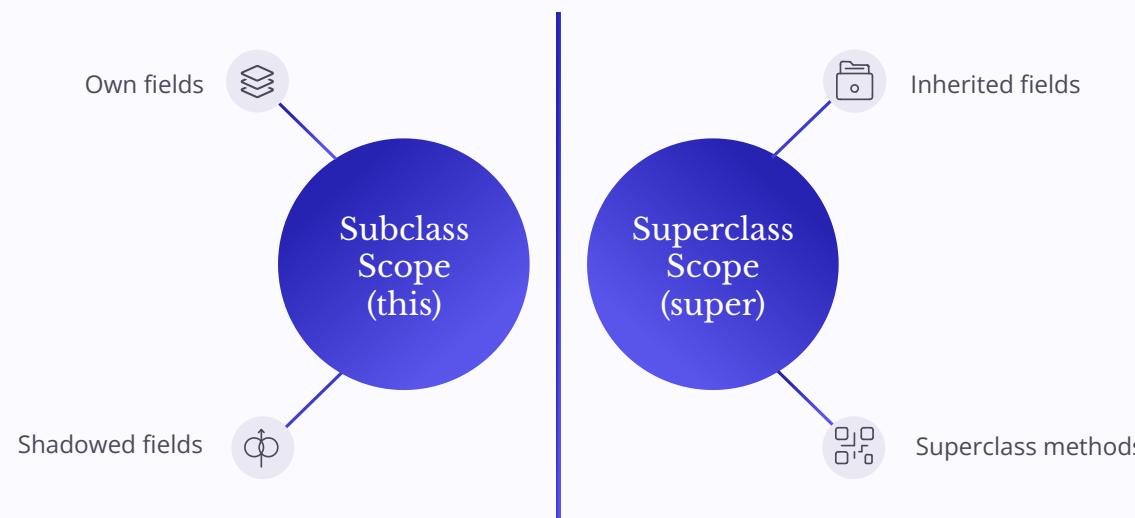
Điểm khác biệt quan trọng:

Sử dụng `super.method()` là cần thiết khi bạn muốn gọi phiên bản method của superclass mà đã bị ghi đè trong subclass. Nếu method đó không bị ghi đè, việc dùng `method()` hay `super.method()` sẽ cho kết quả tương tự, nhưng `super.method()` giúp làm rõ ý định của bạn là đang truy cập từ lớp cha.

Phân Biệt Field/Method của Subclass và Superclass

3. Khi Subclass có Field/Method trùng tên

Đôi khi subclass có thể định nghĩa field hoặc method với cùng tên như trong superclass. Trong trường hợp này, super giúp bạn phân biệt rõ ràng:



Hãy xem ví dụ sau để hiểu rõ hơn về khái niệm che phủ (shadowing) trường và cách sử dụng super để truy cập chúng:

```
// Superclass
public class Animal {
    protected String name = "Animal"; // Field của Superclass
    public String animalSound = "Roar";

    public void displayAnimalInfo() {
        System.out.println("Animal's name: " + name);
    }
}

// Subclass
public class Dog extends Animal {
    private String name = "Dog"; // Field trùng tên (shadows Animal's name)
    public String dogSound = "Woof";

    public Dog(String dogName) {
        // Có thể gọi super() constructor ở đây nếu Animal có constructor riêng
        this.name = dogName; // Gán giá trị cho field 'name' của Dog
    }

    public void showNames() {
        System.out.println("Subclass's name (direct access): " + name); // "Dog" - của subclass
        System.out.println("Subclass's name (with this): " + this.name); // "Dog" - của subclass
        System.out.println("Superclass's name (with super): " + super.name); // "Animal" - của superclass
    }
}

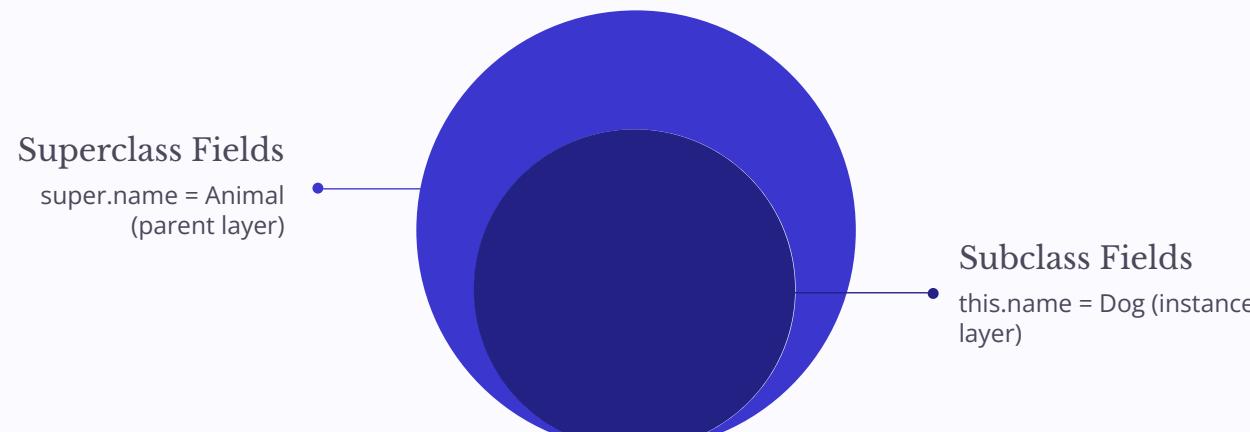
// Ví dụ về method shadowing (không phải override)
public void displayAnimalInfo() { // Đây là một method mới, không ghi đè Animal's displayAnimalInfo
    System.out.println("Dog's info: " + this.name + ", " + dogSound);
}

public static void main(String[] args) {
    Dog myDog = new Dog("Buddy");
    myDog.showNames();
    System.out.println("---");
    myDog.displayAnimalInfo(); // Gọi method của Dog
    // Để gọi method displayAnimalInfo() của Animal, bạn cần một tham chiếu Animal hoặc gọi tường minh từ bên trong Dog.
}
```

Subclass's name (direct access): Buddy
Subclass's name (with this): Buddy
Superclass's name (with super): Animal

Dog's info: Buddy, Woof

Trong Java, khi subclass định nghĩa một trường có cùng tên với trường trong superclass, trường của superclass sẽ bị che phủ (shadowed) bởi trường của subclass. Điều này có nghĩa là khi bạn truy cập trường bằng tên từ trong subclass, Java sẽ ưu tiên trường của subclass.



1

Truy cập không có tiền tố

name → Tìm trong scope gần nhất (subclass trước). Nếu subclass có, nó sẽ được sử dụng.

2

Truy cập với this

this.name → Chỉ rõ là field của đối tượng hiện tại (luôn trả đến field của subclass nếu có, hoặc field được kế thừa nếu subclass không có field cùng tên).

3

Truy cập với super

super.name → Chỉ rõ là field của superclass, bỏ qua bất kỳ trường che phủ nào trong subclass.

⚠ **Best Practice:** Tránh việc đặt tên trùng giữa field của subclass và superclass để tránh gây nhầm lẫn và khó debug. Nếu bắt buộc phải có trường trùng tên (do thiết kế hệ thống hoặc legacy code), hãy luôn sử dụng super.fieldName và this.fieldName một cách rõ ràng để biểu thị bạn đang truy cập trường nào. Đối với phương thức, super.methodName() là cần thiết để gọi phiên bản của lớp cha khi phương thức đó đã bị ghi đè trong lớp con.

Khi Nào Nên Dùng Inheritance?

✓ Quan hệ "IS-A" thực sự

Inheritance chỉ nên được sử dụng khi có quan hệ "IS-A" (là một) thực sự giữa hai class. Điều này có nghĩa là lớp con phải là một dạng chuyên biệt hóa của lớp cha, kế thừa các đặc tính chung và thêm các đặc điểm riêng. Hãy xem các ví dụ sau:

NÊN DÙNG Inheritance

```
// Dog IS-A Animal
class Animal {
    String species;
    void eat() { /*...*/ }
}

class Dog extends Animal {
    String breed;
    void bark() { /*...*/ }
}

// Car IS-A Vehicle
class Vehicle {
    int maxSpeed;
    void start() { /*...*/ }
}

class Car extends Vehicle {
    String model;
    void drive() { /*...*/ }
}

// Student IS-A Person
class Person {
    String name;
    int age;
}

class Student extends Person {
    String studentId;
    void study() { /*...*/ }
}
```

KHÔNG NÊN DÙNG Inheritance

```
// Circle HAS-A Point (không phải IS-A)
// class Circle extends Point { // SAI!
//     double radius;
// }

// Giải thích: Một hình tròn có một điểm trung tâm, nhưng bản thân nó không phải là một điểm. Mỗi
// quan hệ ở đây là "có một" (HAS-A).

// A HAS-A B
// class Engine extends Car { // SAI!
//     void startEngine() { /*...*/ }
// }

// Giải thích: Một chiếc ô tô có một động cơ, động cơ không phải là một loại ô tô.

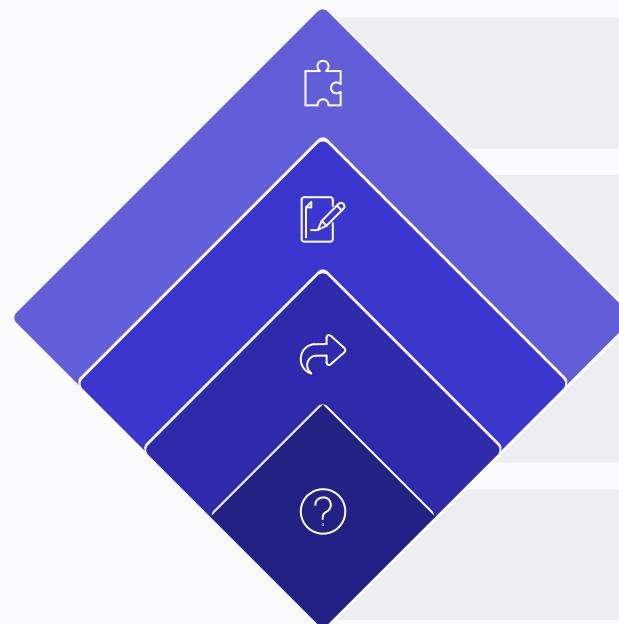
// Utility HAS-A Feature
// class DatabaseUtil extends Connection { // SAI!
//     void connect() { /*...*/ }
//     void query() { /*...*/ }
// }

// Giải thích: Một lớp tiện ích có thể sử dụng (HAS-A) một đối tượng Connection, nó không phải là một
// loại Connection.
```

Trong các ví dụ này, việc sử dụng inheritance sẽ tạo ra mối quan hệ không tự nhiên hoặc sai ngữ nghĩa. "Một hình tròn là một điểm" hay "Một động cơ là một chiếc ô tô" là những mệnh đề sai. Trong những trường hợp này, nên sử dụng composition (tổng hợp) để các đối tượng có thể chứa các đối tượng khác làm thuộc tính của chúng.

Các trường hợp này thể hiện mối quan hệ phân cấp rõ ràng, nơi lớp con là một dạng đặc biệt của lớp cha. Lớp con kế thừa các thuộc tính và hành vi chung từ lớp cha và có thể thêm các đặc điểm riêng mà không làm thay đổi bản chất của lớp cha.

Cây Quyết Định Sử Dụng Inheritance



No: Consider Composition

Prefer composition or interfaces

Does subclass need to override?

Modify superclass behavior?

Yes: Use Inheritance

Share common behavior

Is there a clear IS-A relationship?

Confirm specialization

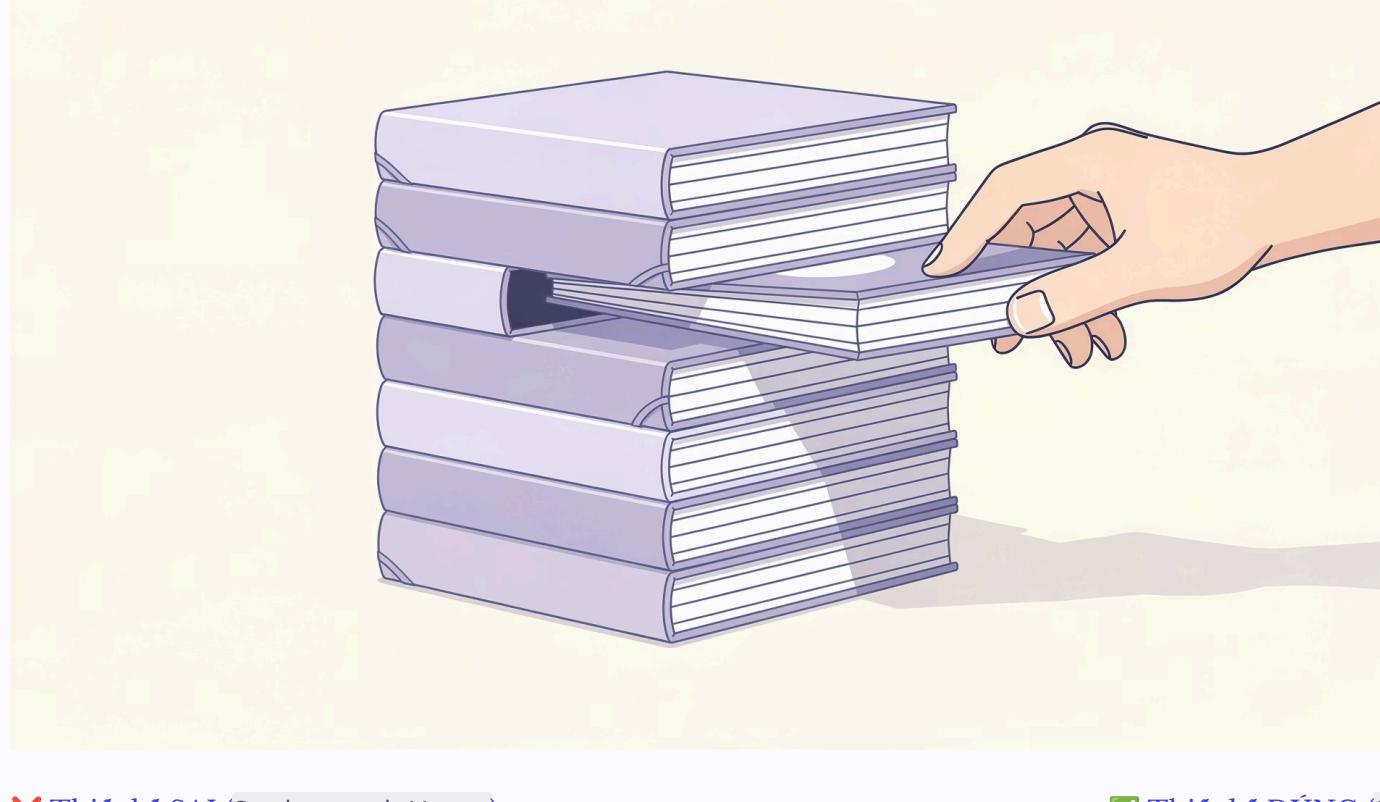
Sử dụng cây quyết định trên để đánh giá liệu inheritance có phải là lựa chọn phù hợp cho thiết kế lớp của bạn hay không. Luôn ưu tiên composition (tổng hợp) hoặc interfaces (giao diện) nếu mối quan hệ "IS-A" không rõ ràng hoặc không bền vững theo thời gian.

Phân Tích Các Trường Hợp Đặc Biệt trong Inheritance

Không phải lúc nào mối quan hệ "IS-A" theo logic thông thường cũng phù hợp để áp dụng Inheritance trong lập trình hướng đối tượng. Chúng ta cần xem xét kỹ lưỡng các hệ quả của việc kế thừa để tránh những lỗi thiết kế tiềm ẩn.

Ví dụ 1: Stack extends Vector? ❌ (Sai làm trong thiết kế thư viện cũ)

Đây là một thiết kế sai trong thư viện Java cũ, nơi Stack kế thừa từ Vector. Mặc dù về mặt *implementation*, Stack có thể dùng Vector bên trong để lưu trữ các phần tử, nhưng Stack KHÔNG PHẢI "LÀ MỘT" Vector theo đúng nghĩa.



✗ Thiết kế SAI (Stack extends Vector)

```
// Stack không phải là Vector
class Stack extends Vector {
    // Kế thừa Vector
    // Expose tất cả methods của Vector
    // ví dụ: add(index, element), remove(index)
    // không phù hợp với Stack (LIFO)
}
```

Vấn đề: Khi Stack kế thừa từ Vector, nó sẽ **kết thừa tất cả các phương thức công khai** của Vector. Điều này bao gồm các phương thức cho phép thao tác ở giữa danh sách như add(index, element) hoặc remove(index). Một Stack thực sự chỉ nên cho phép thêm (push) và bóc (pop) phần tử từ đỉnh (nguyên tắc LIFO - Last-In, First-Out). Việc expose các phương thức của Vector làm phá vỡ tính toàn vẹn của cấu trúc dữ liệu Stack, cho phép người dùng thực hiện các thao tác không hợp lệ đối với một Stack.

✓ Thiết kế ĐÚNG (Stack HAS-A Vector)

```
// Stack HAS-A Vector (Composition)
class Stack {
    private Vector<Object> elements; // Hoặc List/Deque

    public Stack() {
        elements = new Vector<>(); // Khởi tạo Vector
    }

    public void push(Object o) {
        elements.add(o); // Thêm vào cuối (đỉnh Stack)
    }

    public Object pop() {
        if (elements.isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return elements.remove(elements.size() - 1); // Bóc từ cuối
    }

    public boolean isEmpty() {
        return elements.isEmpty();
    }
}
```

Giải pháp: Sử dụng Composition (tổng hợp). Thay vì kế thừa, Stack "có một" (HAS-A) Vector (hoặc tốt hơn là một List hay Deque) làm thuộc tính riêng tư bên trong. Bằng cách này, Stack chỉ expose các phương thức push() và pop(), đảm bảo tính nhất quán của cấu trúc dữ liệu LIFO và không để lộ các phương thức không mong muốn của Vector.

Ví dụ 2: Square extends Rectangle? ⚠ (Vi phạm Liskov Substitution Principle)

Đây là một trường hợp phức tạp hơn. Về mặt toán học, hình vuông (Square) là một trường hợp đặc biệt của hình chữ nhật (Rectangle). Tuy nhiên, trong lập trình OOP, việc cho Square kế thừa Rectangle có thể gây ra những vấn đề nghiêm trọng và vi phạm Nguyên tắc Thay thế Liskov (Liskov Substitution Principle - LSP).

```
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    // Phương thức setWidth và setHeight bị ghi đè để duy trì hình vuông
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width); // Phải giữ hình vuông, thay đổi chiều cao
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height); // Phải giữ hình vuông, thay đổi chiều rộng
    }
}
```

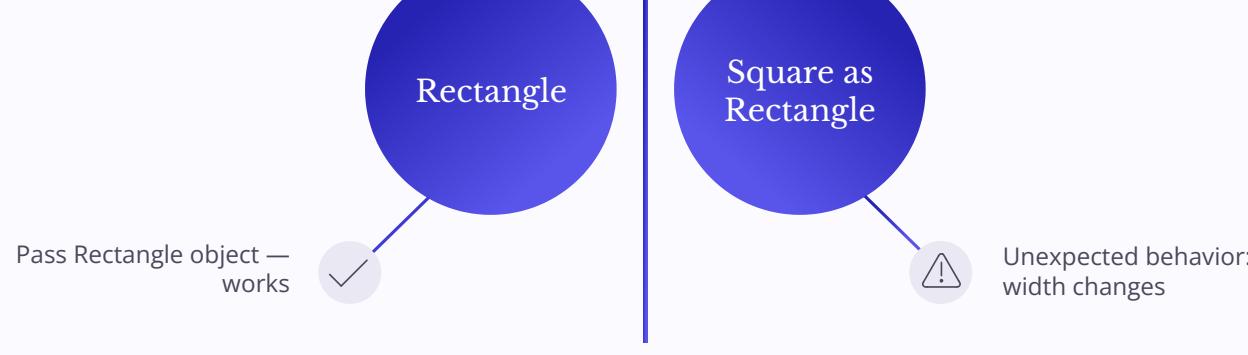
Vấn đề phát sinh:

```
// Giả sử có một hàm nhận đối tượng Rectangle
public void manipulateRectangle(Rectangle r) {
    r.setWidth(5);
    r.setHeight(10);
    // Chúng ta mong đợi diện tích là 5 * 10 = 50
    // Nhưng nếu r là một đối tượng Square...
}

Rectangle rect = new Rectangle(); // Một hình chữ nhật thực sự
manipulateRectangle(rect); // Diện tích sẽ là 50 (5x10)
```

```
Rectangle sq = new Square(); // Một hình vuông được coi là hình chữ nhật
manipulateRectangle(sq); // Sau khi gọi, Square bây giờ sẽ là 10x10!
// Diện tích là 100, không phải 50.
// Điều này vi phạm nguyên tắc LSP!
```

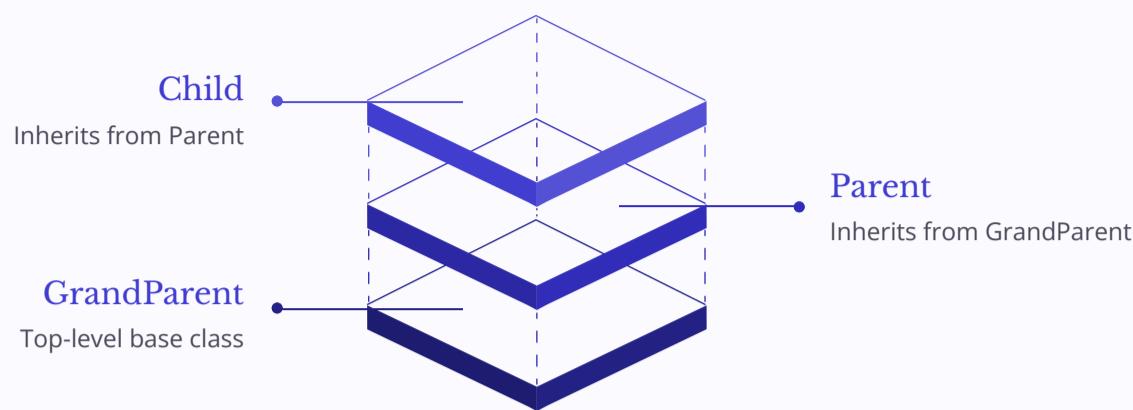
Trong ví dụ trên, khi chúng ta truyền một đối tượng Square vào hàm manipulateRectangle (mong đợi một Rectangle), hành vi của nó thay đổi không mong muốn. Một đối tượng Square không thể thay thế một đối tượng Rectangle mà không làm thay đổi tính đúng đắn của chương trình, vì phương thức setHeight trên Square cũng thay đổi width, điều mà Rectangle không làm.



Multi-Level Inheritance

Kế thừa nhiều cấp độ

Multi-level Inheritance xảy ra khi có nhiều cấp độ kế thừa nối tiếp nhau, tạo thành một chuỗi kế thừa, nơi một lớp con kế thừa từ một lớp cha, mà lớp cha đó lại kế thừa từ một lớp khác nữa.



01

Level 1: GrandParent Class

Lớp cơ sở nhất, định nghĩa các đặc điểm và hành vi chung.

02

Level 2: Parent Class

Kế thừa từ GrandParent, thêm các đặc điểm và hành vi cụ thể hơn.

03

Level 3: Child Class

Kế thừa từ Parent, mang tất cả đặc điểm của GrandParent và Parent, đồng thời có các đặc điểm riêng của mình.

Hãy xem xét ví dụ cụ thể với chuỗi kế thừa **Vehicle -> Car -> ElectricCar**:

```
// Level 1: Vehicle
public class Vehicle {
    protected String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    public void start() {
        System.out.println(brand + " is starting.");
    }
}
```

```
// Level 2: Car extends Vehicle
public class Car extends Vehicle {
    protected String model;

    public Car(String brand, String model) {
        super(brand); // Gọi constructor của Vehicle
        this.model = model;
    }

    public void drive() {
        System.out.println(model + " is driving.");
    }
}
```

```
// Level 3: ElectricCar extends Car
public class ElectricCar extends Car {
    private int batteryCapacity;

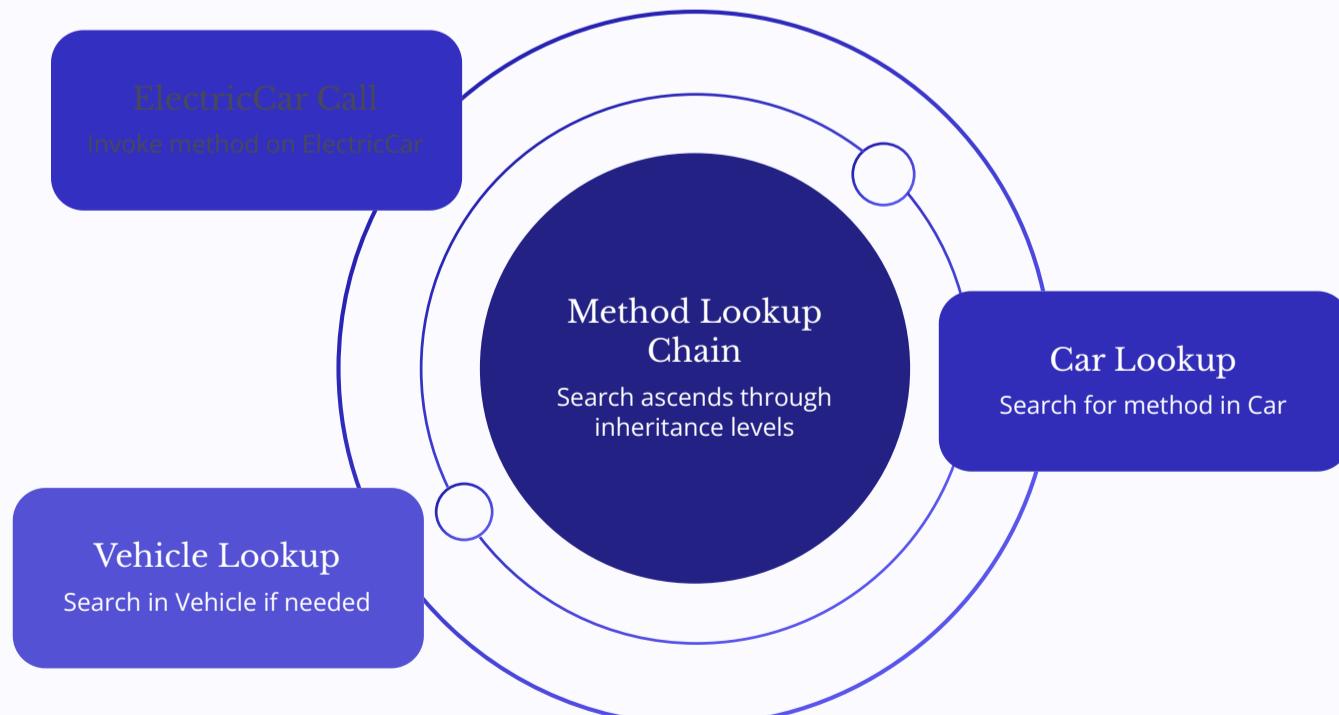
    public ElectricCar(String brand, String model, int capacity) {
        super(brand, model); // Gọi constructor của Car
        this.batteryCapacity = capacity;
    }

    public void charge() {
        System.out.println(model + " is charging with " +
batteryCapacity + " kWh.");
    }
}
```

Một đối tượng ElectricCar kế thừa gián tiếp từ Vehicle và trực tiếp từ Car, do đó có thể sử dụng tất cả các phương thức từ cả ba lớp:

```
ElectricCar myCar = new ElectricCar("Tesla", "Model S", 100);
myCar.start(); // Từ Vehicle (cấp 1)
myCar.drive(); // Từ Car (cấp 2)
myCar.charge(); // Riêng của ElectricCar (cấp 3)
```

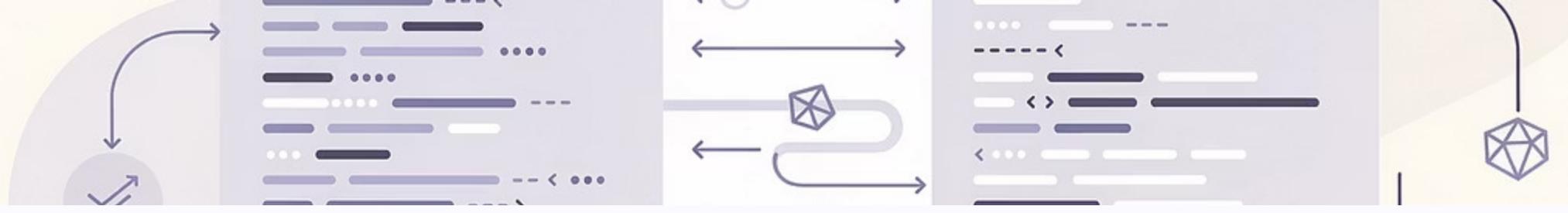
Khi một phương thức được gọi trên đối tượng ElectricCar, JVM sẽ tìm kiếm phương thức đó theo chuỗi kế thừa, bắt đầu từ lớp hiện tại và đi dần lên các lớp cha:



💡 Khi nào nên sử dụng Multi-Level Inheritance?

Kế thừa đa cấp phù hợp khi có một hệ thống phân cấp IS-A rõ ràng và tự nhiên, nơi mỗi cấp độ bổ sung thêm các đặc điểm hoặc hành vi cụ thể hơn. Ví dụ: Động vật -> Động vật có vú -> Chó. Tuy nhiên, nếu chuỗi kế thừa trở nên quá sâu (> 3-4 cấp độ) hoặc các lớp không có mối quan hệ IS-A mạnh mẽ, nó có thể dẫn đến sự phức tạp, khó bảo trì và dễ vi phạm các nguyên tắc thiết kế như LSP.

⚠️ Lưu ý về Multiple Inheritance: Java không hỗ trợ Multiple Inheritance (một class extends nhiều class cùng lúc). Bạn chỉ có thể extends một class duy nhất. Tuy nhiên, bạn có thể implement nhiều interface (chúng ta sẽ học ở phần sau).



PHẦN 3.2

Overriding vs Overloading

Overriding (Ghi đè)

Overriding là một khái niệm quan trọng trong lập trình hướng đối tượng, cho phép một lớp con (subclass) cung cấp triển khai cụ thể cho một phương thức đã được định nghĩa trong lớp cha (superclass) của nó. Điều này giúp các lớp con tùy biến hành vi kế thừa, mang lại tính đa hình (polymorphism) mạnh mẽ cho ứng dụng của bạn.

- **Cùng tên** method
- **Cùng tham số** (số lượng, kiểu dữ liệu, thứ tự)
- **Cùng return type** (hoặc subtype - covariant return type)

🎮 Tưởng tượng: Điều khiển đa năng

Bạn có một chiếc điều khiển với nút "**Power**". Khi chĩa vào **TV**, nút Power làm TV bật. Khi chĩa vào **Máy lạnh**, nút Power làm máy lạnh chạy. **Nút bấm giống nhau**, nhưng **hành động khác nhau** tùy đối tượng. Đó chính là Overriding và Polymorphism!

Ví dụ cụ thể

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

public class Dog extends Animal {
    @Override // Annotation đánh dấu override
    public void makeSound() {
        System.out.println("Dog barks: Woof! Woof!");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows: Meow! Meow!");
    }
}
```

```
// Sử dụng
Animal animal = new Animal();
animal.makeSound(); // "Animal makes a sound"

Dog dog = new Dog();
dog.makeSound(); // "Dog barks: Woof! Woof!"

Cat cat = new Cat();
cat.makeSound(); // "Cat meows: Meow! Meow!"
```

Quy Tắc Overriding

Khi ghi đè (override) một phương thức, bạn phải tuân thủ một số quy tắc quan trọng để đảm bảo tính đúng đắn của hành vi kế thừa và đa hình. Dưới đây là các quy tắc chính:

1. Access Modifier không thể hép hơn

Phạm vi truy cập của phương thức ghi đè (ở lớp con) không được hép hơn phương thức được ghi đè (ở lớp cha). Nó có thể giống hoặc rộng hơn (ví dụ: từ protected sang public).

Ví dụ đúng:

```
// Lớp cha  
public class Vehicle {  
    protected void startEngine() { // protected access  
        System.out.println("Engine started (Vehicle)");  
    }  
}
```

```
// Lớp con (Access Modifier rộng hơn)  
public class Car extends Vehicle {  
    @Override  
    public void startEngine() { // public > protected => OK  
        System.out.println("Car engine started with ignition!");  
    }  
}
```

Ví dụ sai:

```
// Lớp cha  
public class Animal {  
    public void move() { // public access  
        System.out.println("Animal moves");  
    }  
}
```

```
// Lớp con (Access Modifier hép hơn => LỖI BIÊN ĐỊCH)  
public class Dog extends Animal {  
    // @Override // Dòng này sẽ báo lỗi nếu bỏ comment  
    private void move() { // private < public => LỖI!  
        System.out.println("Dog runs on four legs");  
    }  
}
```

☒ **Lỗi thường gặp:** Cố gắng làm cho phương thức ở lớp con ít truy cập hơn lớp cha. Hãy nhớ: "**Mở rộng, không thu hẹp**" phạm vi truy cập khi ghi đè.

2. Return Type phải tương thích (Covariant)

Kiểu trả về của phương thức ghi đè phải giống hệt hoặc là một kiểu con (subtype - covariant return type) của kiểu trả về của phương thức ở lớp cha.

Ví dụ đúng:

```
// Lớp cha  
public class Shape {  
    public Object create() { // Trả về Object  
        return new Shape();  
    }  
}
```

```
// Lớp con (Kiểu trả về là subtype của Object)  
public class Circle extends Shape {  
    @Override  
    public Circle create() { // Circle là subtype của Object => OK  
        return new Circle();  
    }  
}
```

Ví dụ sai:

```
// Lớp cha  
public class Processor {  
    public String process() { // Trả về String  
        return "Processed String";  
    }  
}
```

```
// Lớp con (Kiểu trả về không tương thích => LỖI BIÊN ĐỊCH)  
public class NumberProcessor extends Processor {  
    // @Override // Dòng này sẽ báo lỗi nếu bỏ comment  
    public Integer process() { // Integer không phải là String hay subtype của String => LỖI!  
        return 123;  
    }  
}
```

☒ **Lỗi thường gặp:** Thay đổi kiểu trả về sang một kiểu không liên quan. Đảm bảo rằng kiểu trả về của phương thức con phải "**ít nhất cũng là**" kiểu của phương thức cha hoặc một kiểu con của nó.

3. Chữ ký phương thức phải giống hệt

Tên phương thức và danh sách tham số (số lượng, kiểu dữ liệu, thứ tự) của phương thức ở lớp con phải giống hệt với phương thức ở lớp cha. Nếu khác, đó sẽ là quá tải (overloading) chứ không phải ghi đè.

Ví dụ đúng:

```
// Lớp cha  
public class Payment {  
    public void execute(double amount) { // Tham số: double  
        System.out.println("Processing generic payment of " + amount);  
    }  
}
```

```
// Lớp con (Chữ ký giống hệt)  
public class CreditCardPayment extends Payment {  
    @Override  
    public void execute(double amount) { // Tham số: double => OK  
        System.out.println("Processing credit card payment of " + amount);  
    }  
}
```

Ví dụ sai (đây là Overloading):

```
// Lớp cha  
public class Calculator {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

```
// Lớp con (Đây là Overloading chứ không phải Overriding)  
public class AdvancedCalculator extends Calculator {  
    // Không phải override vì chữ ký khác (thêm tham số c)  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
}
```

☒ **Lỗi thường gặp:** Nhầm lẫn giữa ghi đè và quá tải. Sử dụng annotation `@Override` giúp trình biên dịch phát hiện lỗi nếu chữ ký không khớp.

4. Không ghi đè phương thức final hoặc static

Các phương thức được đánh dấu là `final` không thể bị ghi đè. Các phương thức `static` không thể bị ghi đè (chúng có thể được "che giấu" - method hiding, nhưng không phải đa hình).

Ví dụ sai (final method):

```
// Lớp cha  
public class Parent {  
    public final void printMessage() { // Phương thức final  
        System.out.println("This is a final message.");  
    }  
}
```

```
// Lớp con (Không thể ghi đè phương thức final => LỖI BIÊN ĐỊCH)  
public class Child extends Parent {  
    // @Override // Dòng này sẽ báo lỗi nếu bỏ comment  
    // public void printMessage() { // LỖI! Không thể ghi đè final method  
    //     System.out.println("Child's message.");  
    // }  
}
```

Ví dụ sai (static method):

```
// Lớp cha  
public class BaseClass {  
    public static void display() { // Phương thức static  
        System.out.println("Static method in BaseClass");  
    }  
}
```

```
// Lớp con (Không thể ghi đè static method)  
public class DerivedClass extends BaseClass {  
    // @Override // Dòng này sẽ báo lỗi nếu bỏ comment  
    public static void display() { // Không phải ghi đè, mà là method hiding  
        System.out.println("Static method in DerivedClass");  
    }  
}
```

☒ **Lưu ý quan trọng:** Phương thức static thuộc về lớp, không phải đối tượng, nên không thể thể hiện tính đa hình thông qua ghi đè.

☒ Tóm Tắt Các Quy Tắc Ghi Đè (Overriding)

Để ghi đè một phương thức thành công và đúng cách, hãy nhớ:

- **Access Modifier:** Không được hép hơn (public > protected > default > private).
- **Return Type:** Phải giống hệt hoặc là kiểu con (covariant) của kiểu trả về phương thức cha.
- **Chữ ký:** Tên phương thức và danh sách tham số (số lượng, kiểu, thứ tự) phải giống hệt.
- **final / static:** Không thể ghi đè phương thức final hoặc static.
- Sử dụng annotation `@Override` để trình biên dịch kiểm tra giúp bạn.

Annotation @Override



Tại sao nên dùng @Override?

Mặc dù @Override không bắt buộc, nhưng sử dụng nó mang lại nhiều lợi ích quan trọng, hoạt động như một "lưới an toàn" cho mã của bạn:



Compiler Kiểm Tra Chặt Chẽ

Compiler sẽ đảm bảo method này đúng là ghi đè một method từ superclass. Nếu không phải, nó sẽ báo lỗi ngay lập tức, ngăn ngừa các lỗi khó phát hiện.



Tránh Lỗi Tinh Vi

Giúp bạn phát hiện ngay lập tức các lỗi đánh máy (typo) hoặc sự thay đổi chữ ký phương thức ở lớp cha. Compiler sẽ cảnh báo, thay vì âm thầm tạo ra một phương thức mới mà bạn không hề mong muốn.



Code Rõ Ràng & Dễ Đọc

Người đọc code biết ngay đây là một phương thức được ghi đè, không phải là một phương thức mới. Điều này cải thiện đáng kể khả năng đọc hiểu và bảo trì mã nguồn.

Ví dụ thực tế: @Override giúp phát hiện lỗi như thế nào?

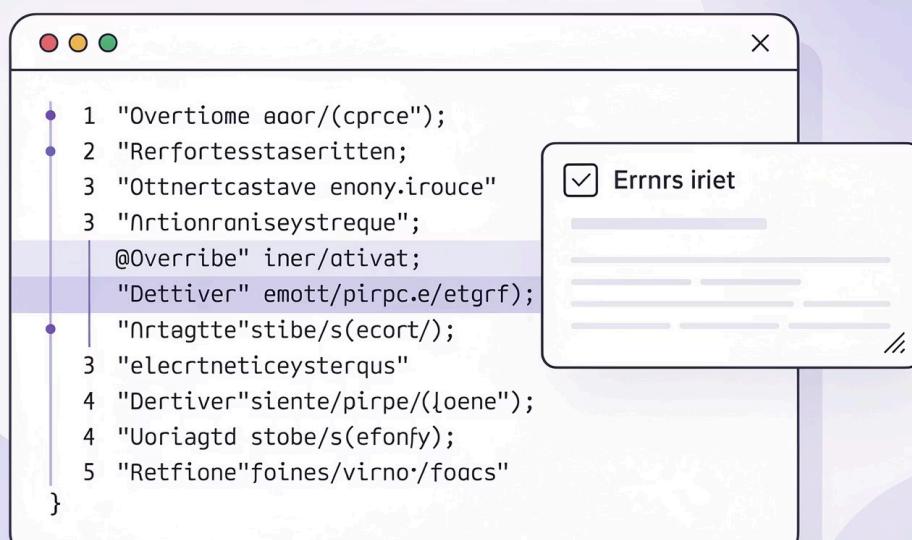
Hãy xem xét ví dụ dưới đây về một lỗi đánh máy phổ biến và cách @Override cứu nguy:

✗ Không có @Override (Lỗi không được phát hiện)

```
// Giả định lớp cha Animal có phương thức 'makeSound()'  
public class Animal {  
    public void makeSound() {  
        System.out.println("Animal makes a sound");  
    }  
  
    public class Dog extends Animal {  
        // Lỗi đánh máy: 'makeSoud()' thay vì 'makeSound()'  
        public void makeSoud() { // Đây là một phương thức MỚI  
            System.out.println("Woof woof!");  
        }  
        // Compiler KHÔNG báo lỗi!  
        // Tại runtime, gọi 'dog.makeSound()' vẫn sẽ chạy phương thức của Animal.  
    }  
}
```

✓ Có @Override (Lỗi được Compiler phát hiện)

```
// Giả định lớp cha Animal có phương thức 'makeSound()'  
public class Animal {  
    public void makeSound() {  
        System.out.println("Animal makes a sound");  
    }  
  
    public class Dog extends Animal {  
        @Override // Annotation này là chìa khóa!  
        // Lỗi đánh máy tương tự  
        public void makeSoud() { // <<- LỖI BIÊN DỊCH TẠI ĐÂY!  
            System.out.println("Woof woof!");  
        }  
        // Compiler SẼ báo lỗi:  
        // "Method does not override method from its superclass"  
        // hoặc tương tự, vì 'makeSoud()' không tồn tại trong Animal.  
    }  
}
```



☐ Thực hành tốt nhất: Luôn sử dụng @Override!

Để đảm bảo tính đúng đắn và an toàn cho mã của bạn, hãy luôn sử dụng annotation @Override mỗi khi bạn có ý định ghi đè một phương thức từ lớp cha hoặc triển khai một phương thức từ interface. Nó là một công cụ mạnh mẽ giúp bạn tránh những lỗi phát sinh từ sự thiếu nhất quán trong chữ ký phương thức.

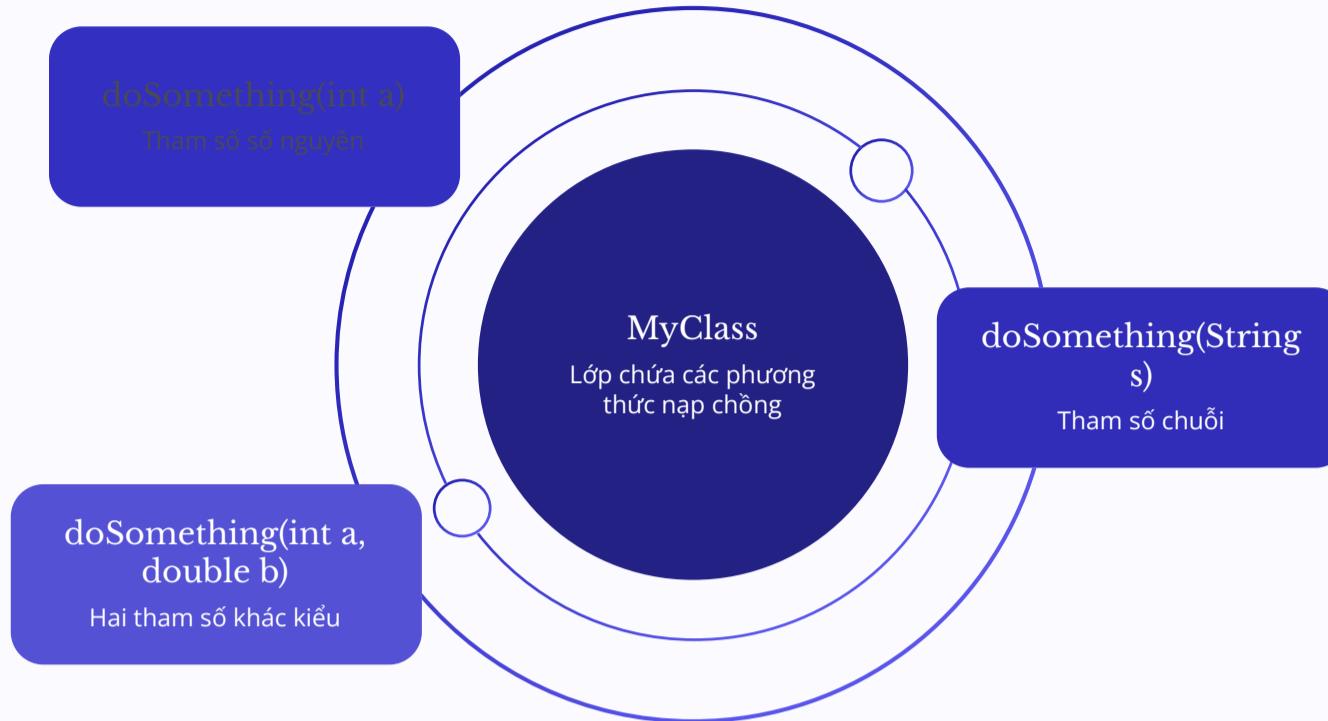
Overloading (Nạp chồng phương thức)

Định nghĩa Overloading

Overloading (Nạp chồng phương thức) là khả năng có nhiều phương thức trong cùng một lớp có **cùng tên**, nhưng **khác nhau về chữ ký phương thức (method signature)**. Compiler sử dụng chữ ký phương thức (bao gồm số lượng, kiểu và thứ tự các tham số) để quyết định phương thức nào sẽ được gọi.

Để hai phương thức được coi là nạp chồng (overloaded), chúng phải:

- Có **cùng tên**.
- Có **các danh sách tham số khác nhau** (về số lượng, kiểu dữ liệu, hoặc thứ tự của các kiểu dữ liệu).
- Kiểu trả về (return type) **có thể giống hoặc khác nhau**, nhưng không phải là yếu tố quyết định để phân biệt các phương thức nạp chồng.



So sánh các yếu tố phân biệt Overloading

Bảng dưới đây tóm tắt các yếu tố cần thiết và không cần thiết để thực hiện nạp chồng phương thức:

Tên phương thức	Phải giống nhau	Đây là đặc điểm nhận dạng chính của overloading.
Số lượng tham số	Phải khác nhau	Ví dụ: add(int a, int b) vs add(int a, int b, int c)
Kiểu dữ liệu tham số	Phải khác nhau	Ví dụ: add(int a, int b) vs add(double a, double b)
Thứ tự kiểu dữ liệu tham số	Phải khác nhau	Ví dụ: print(String s, int i) vs print(int i, String s)
Kiểu trả về (Return Type)	Không bắt buộc phải khác	Không đủ để phân biệt các phương thức nạp chồng. Compiler chỉ dựa vào chữ ký phương thức.

Ví dụ thực tế: Class Printer

Hãy xem xét một lớp Printer sử dụng nạp chồng phương thức để xử lý việc in các loại dữ liệu khác nhau:

```
public class Printer {  
    // Phương thức in chuỗi  
    public void print(String message) {  
        System.out.println("In chuỗi: " + message);  
    }  
  
    // Phương thức in số nguyên  
    public void print(int number) {  
        System.out.println("In số nguyên: " + number);  
    }  
  
    // Phương thức in số thực  
    public void print(double decimal) {  
        System.out.println("In số thực: " + decimal);  
    }  
  
    // Phương thức in boolean  
    public void print(boolean status) {  
        System.out.println("In trạng thái: " + status);  
    }  
}
```

```
// Cách sử dụng các phương thức nạp chồng:  
Printer myPrinter = new Printer();  
  
myPrinter.print("Xin chào thế giới!"); // Gọi print(String)  
myPrinter.print(123); // Gọi print(int)  
myPrinter.print(45.67); // Gọi print(double)  
myPrinter.print(true); // Gọi print(boolean)
```

Trong ví dụ trên, compiler sẽ tự động chọn phương thức print() phù hợp dựa trên kiểu dữ liệu của đối số được truyền vào. Điều này giúp code trở nên linh hoạt và dễ sử dụng hơn.

Analogy: Đầu bếp đa năng

- Hãy tưởng tượng bạn là một đầu bếp tài năng với một món ăn chủ lực, ví dụ như "Làm trứng". Bạn có thể "Làm trứng" theo nhiều cách khác nhau tùy thuộc vào nguyên liệu bạn có:
- LàmTrứng(trứng): Luộc trứng
 - LàmTrứng(trứng, sữa): Trứng khuấy
 - LàmTrứng(trứng, phô mai, rau củ): Trứng ốp lết

Mỗi phương thức LàmTrứng đều có cùng tên, nhưng chúng nhận các nguyên liệu (tham số) khác nhau, tạo ra các món ăn khác nhau. Khách hàng chỉ cần nói "LàmTrứng" và cung cấp nguyên liệu, bạn sẽ tự động biết phải làm món nào. Đây chính là cách Overloading hoạt động trong lập trình!

Quy Tắc Overloading



Phải khác về danh sách tham số

Các phương thức nạp chồng phải có danh sách tham số khác nhau về **số lượng**, **kiểu dữ liệu**, hoặc **thứ tự** của các kiểu dữ liệu.

Ví dụ hợp lệ:

```
class Calculator {  
    void add(int a, int b) /* ... */ // Hai tham số int  
    void add(int a, int b, int c) /* ... */ // Ba tham số int (khác số lượng)  
    void add(double a, double b) /* ... */ // Hai tham số double (khác kiểu dữ liệu)  
    void print(String s, int i) /* ... */ // String, int  
    void print(int i, String s) /* ... */ // int, String (khác thứ tự)  
}
```



Kiểu trả về không đủ để phân biệt

Kiểu trả về (return type) có thể giống hoặc khác nhau, nhưng nó **không phải là yếu tố quyết định** để phân biệt các phương thức nạp chồng. Compiler chỉ dựa vào chữ ký phương thức.

Ví dụ không hợp lệ:

```
class Example {  
    int getValue() { return 1; }  
    // Lỗi biên dịch: Phương thức này không thể được nạp chồng chỉ bằng cách thay đổi kiểu trả về  
    double getValue() { return 1.0; }  
}
```



Access Modifier (Phạm vi truy cập) có thể khác nhau

Các phương thức nạp chồng có thể có các **access modifier** (phạm vi truy cập) khác nhau (ví dụ: public, private, protected, default).

Ví dụ hợp lệ:

```
class MyClass {  
    public void process() /* ... */  
    private void process(String data) /* ... */ // Khác access modifier, nhưng quan trọng hơn là  
    // khác tham số  
}
```



Các ngoại lệ (Exceptions) có thể khác nhau

Các phương thức nạp chồng có thể khai báo **ném** các loại ngoại lệ khác nhau (checked hoặc unchecked) mà không ảnh hưởng đến khả năng nạp chồng.

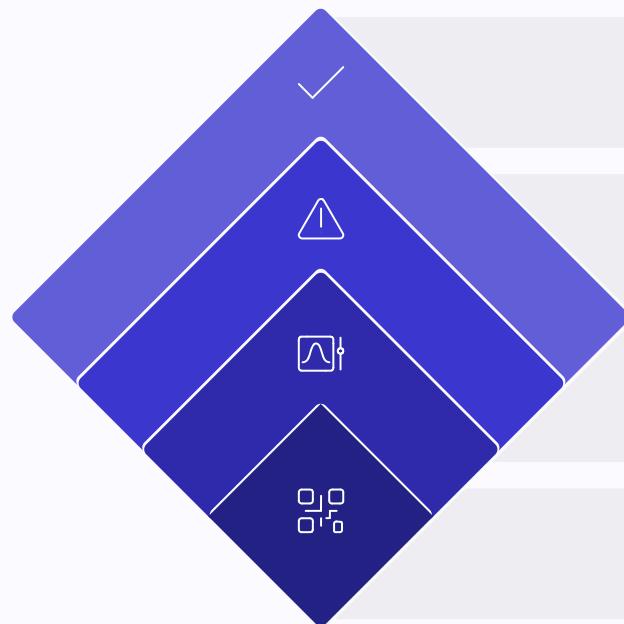
Ví dụ hợp lệ:

```
import java.io.IOException;  
  
class FileProcessor {  
    void readData() /* ... */  
    void readData(String filename) throws IOException /* ... */ // Khác tham số và khai báo ngoại  
    // lệ  
}
```

Các lỗi thường gặp khi sử dụng Overloading

- Chỉ thay đổi kiểu trả về:** Đây là lỗi phổ biến nhất. Như đã đề cập, kiểu trả về không phải là một phần của chữ ký phương thức để phân biệt các phương thức nạp chồng.
- Chỉ thay đổi access modifier:** Tương tự như kiểu trả về, access modifier không phải là yếu tố phân biệt overloading.
- Nhầm lẫn giữa Overloading và Overriding:** Overloading xảy ra trong cùng một lớp hoặc giữa các lớp không có quan hệ thừa kế rõ ràng (khác chữ ký phương thức), trong khi Overriding xảy ra giữa lớp cha và lớp con (cùng chữ ký phương thức).
- Sử dụng sai thứ tự kiểu dữ liệu tham số:** Đôi khi, nhà phát triển quên rằng thứ tự kiểu dữ liệu cũng là một yếu tố quan trọng trong chữ ký phương thức.

Flowchart: Đây có phải là Overloading hợp lệ?



Hợp lệ nạp chồng
Phù hợp theo quy tắc

Kiểm tra biên dịch
Trùng tham số => lỗi

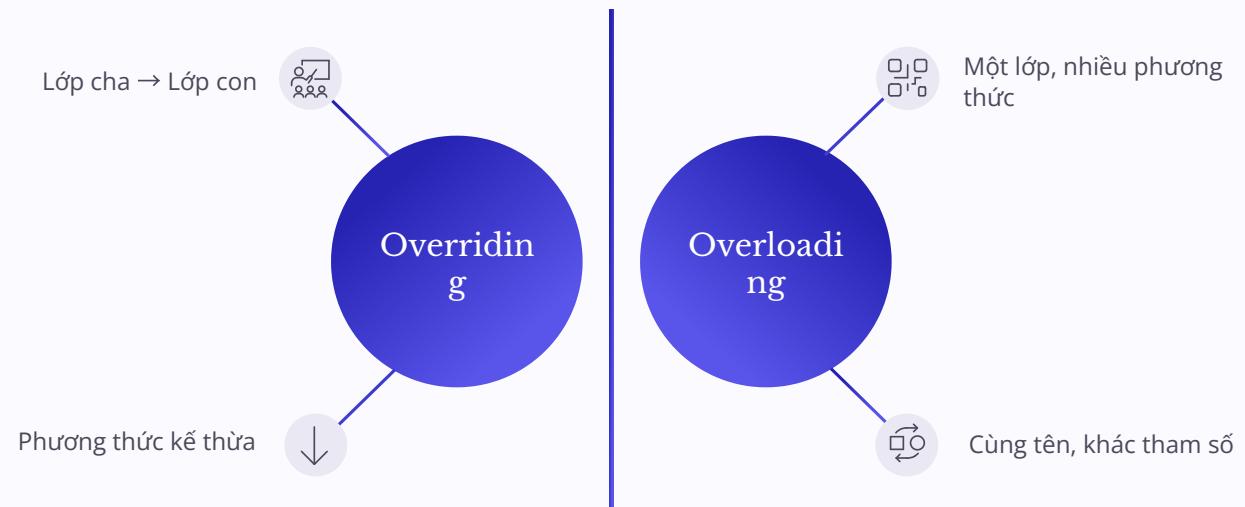
Danh sách tham số khác?
Số, kiểu hoặc thứ tự

Tên phương thức giống?
Kiểm tra tên

So Sánh Overriding vs Overloading

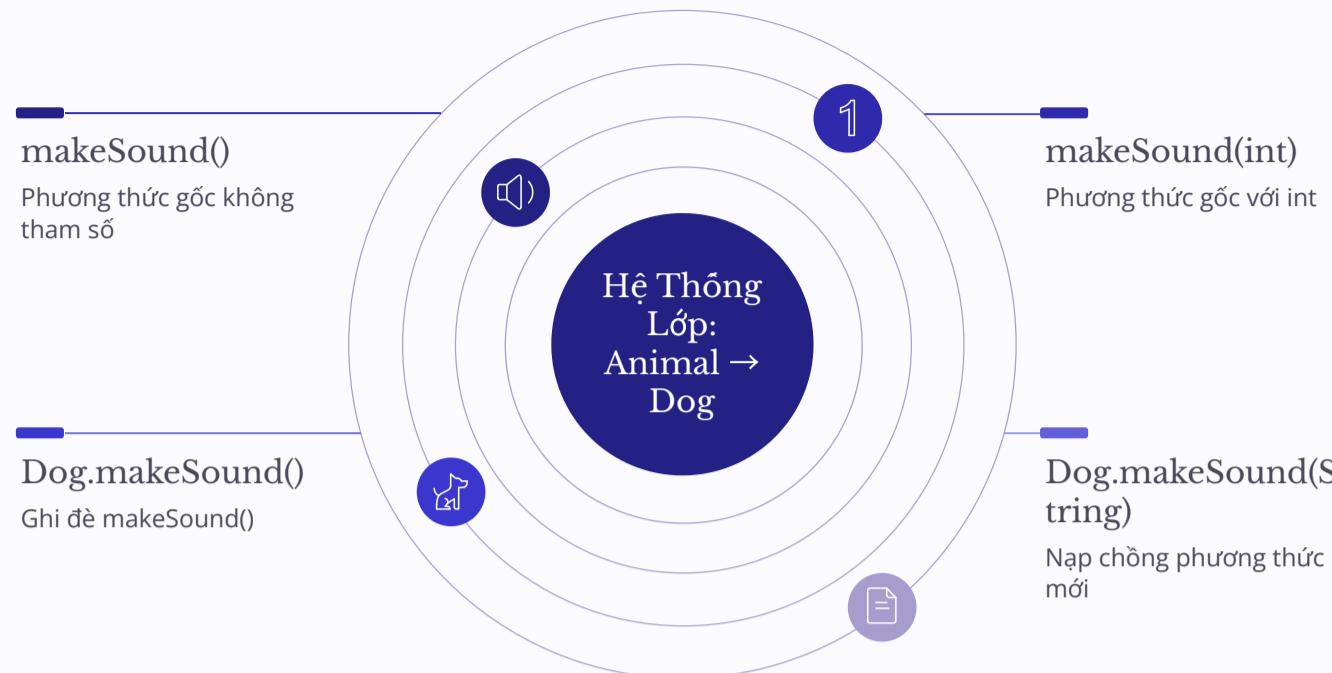
Đặc điểm	Overriding	Overloading
Mục đích	Thay đổi hành vi (cung cấp implementation mới cho phương thức đã kế thừa).	Mở rộng chức năng (cung cấp nhiều cách thực hiện cho một hành động).
Quan hệ yêu cầu	Phải có quan hệ kế thừa (giữa superclass và subclass).	Không yêu cầu kế thừa (có thể trong cùng một class hoặc giữa các lớp kế thừa).
Binding (Thời điểm quyết định)	Runtime (Dynamic Binding)	Compile-time (Static Binding)
Quy tắc chữ ký phương thức	<ul style="list-style-type: none">Tên method: Phải giống.Tham số: Phải giống.Kiểu trả về: Phải giống hoặc là subtype.Access modifier: Không được hép hơn phương thức trong superclass.	<ul style="list-style-type: none">Tên method: Phải giống.Tham số: Phải khác (về số lượng, kiểu dữ liệu, hoặc thứ tự).Kiểu trả về: Có thể khác.Access modifier: Có thể khác.
Annotation	@Override (khuyến nghị)	Không cần

Sự khác biệt cơ bản nhất: **Overriding** thay đổi implementation của method đã có, còn **Overloading** tạo ra các phiên bản mới của cùng một method với tham số khác nhau.



Ví Dụ Tổng Hợp: Overriding + Overloading

Một class có thể vừa sử dụng Overriding vừa sử dụng Overloading. Hãy xem ví dụ sau:



Class Animal (Superclass)

```
public class Animal {  
    // Phương thức gốc, có thể được ghi đè  
    public void makeSound() {  
        System.out.println("Animal makes a sound");  
    }  
  
    // Phương thức makeSound() đã được nạp chồng (overloaded)  
    // Cùng tên nhưng khác tham số (int volume)  
    public void makeSound(int volume) {  
        System.out.println("Animal makes a sound" +  
            " at volume " + volume);  
    }  
}
```

- Animal có hai phương thức makeSound: một không có tham số, một có tham số int. Đây là ví dụ về Overloading trong cùng một class.

Class Dog (Subclass)

```
public class Dog extends Animal {  
  
    // Ghi đè (Overriding) phương thức makeSound()  
    // không tham số từ Animal  
    @Override  
    public void makeSound() {  
        System.out.println("Dog barks: Woof!");  
    }  
  
    // Ghi đè (Overriding) phương thức makeSound()  
    // với tham số int từ Animal  
    @Override  
    public void makeSound(int volume) {  
        System.out.println("Dog barks loudly at volume " +  
            volume + "!");  
    }  
  
    // Nạp chồng (Overloading): Một phương thức mới  
    // với cùng tên nhưng khác tham số (String)  
    public void makeSound(String message) {  
        System.out.println("Dog says: " + message);  
    }  
}
```

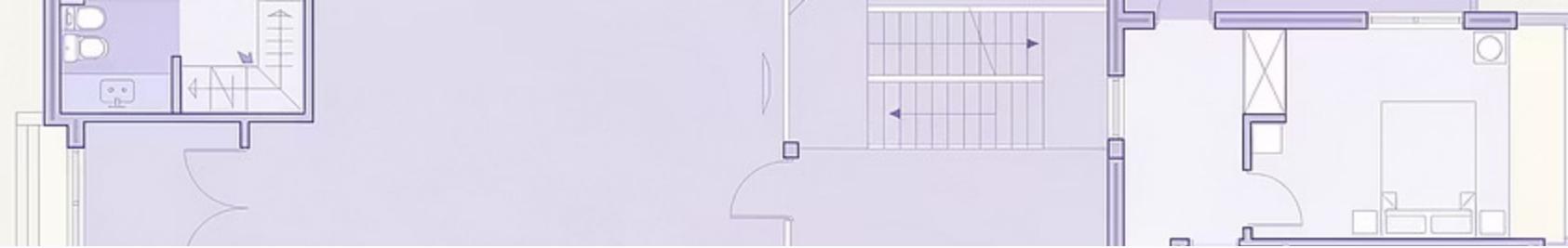
- Dog ghi đè (override) cả hai phiên bản makeSound từ Animal. Đồng thời, nó cũng nạp chồng (overload) thêm một phiên bản makeSound mới nhận tham số String.

Class Main (Test)

```
public class Main {  
    public static void main(String[] args) {  
        Dog myDog = new Dog();  
  
        System.out.print("Gọi phương thức makeSound() không tham số: ");  
        myDog.makeSound(); // Kết quả: "Dog barks: Woof!" (Overriding)  
  
        System.out.print("Gọi phương thức makeSound(int volume): ");  
        myDog.makeSound(10); // Kết quả: "Dog barks loudly at volume 10!" (Overriding)  
  
        System.out.print("Gọi phương thức makeSound(String message): ");  
        myDog.makeSound("Hello World"); // Kết quả: "Dog says: Hello World" (Overloading)  
    }  
}
```

- Ví dụ này minh họa cách các cuộc gọi phương thức được giải quyết:
 - Hai lệnh gọi đầu tiên sử dụng các phương thức đã được **ghi đè (Overriding)** trong lớp Dog.
 - Lệnh gọi thứ ba sử dụng phương thức **nạp chồng (Overloading)** mới được thêm vào trong lớp Dog.

Trong ví dụ này, Dog vừa override các method từ Animal, vừa overload thêm một version mới với tham số String.



PHẦN 3.3

Abstract Class & Phương Thức Trừu Tượng

Tìm hiểu về Abstract Class – một khuôn mẫu thiết kế mạnh mẽ cho phép định nghĩa các phương thức trừu tượng mà các lớp con phải triển khai, đảm bảo tính kế thừa và cấu trúc trong lập trình hướng đối tượng.

Abstract Class là gì?

Abstract Class là một class đặc biệt mà bạn **không thể tạo object** trực tiếp từ nó. Nó chỉ dùng để làm blueprint cho các subclass kế thừa.

Tưởng tượng: Ngôi nhà chưa hoàn thiện

Abstract Class giống như bản vẽ một ngôi nhà đã xây xong móng và tường, nhưng **chưa lợp mái**. Vì chưa hoàn thiện, bạn không thể vào ở (không thể tạo object). Bạn phải thuê thợ (Subclass) để lợp mái (implement abstract methods) thì mới thành ngôi nhà hoàn chỉnh (Concrete Class) để ở được.

Tại sao cần Abstract Class?

Vấn đề

```
// Animal có method makeSound()  
// nhưng không biết implement  
// thế nào  
public class Animal {  
    public void makeSound() {  
        // Làm gì đây???  
        // Mỗi loài kêu khác nhau!  
        System.out.println("???");  
    }  
}
```

Giải pháp

```
// Animal là abstract  
// không thể tạo object  
public abstract class Animal {  
    protected String name;  
  
    // Abstract method  
    // subclass phải implement  
    public abstract void  
        makeSound();  
  
    // Concrete method  
    public void eat() {  
        System.out.println(  
            name + " is eating"  
        );  
    }  
}
```

```
// Subclass phải implement abstract method  
public class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof! Woof!");  
    }  
}  
  
//
```

Cú Pháp Abstract Class và Abstract Method

Định nghĩa Abstract Method

Abstract method là method không có body (implementation), chỉ có khai báo:

```
[access modifier] abstract returnType methodName(parameters);
```

Để hiểu rõ hơn về các thuộc tính của abstract method, hãy xem các điểm nổi bật sau:

Không có body { }

Abstract method chỉ khai báo signature, không có implementation.

Kết thúc bằng dấu ;

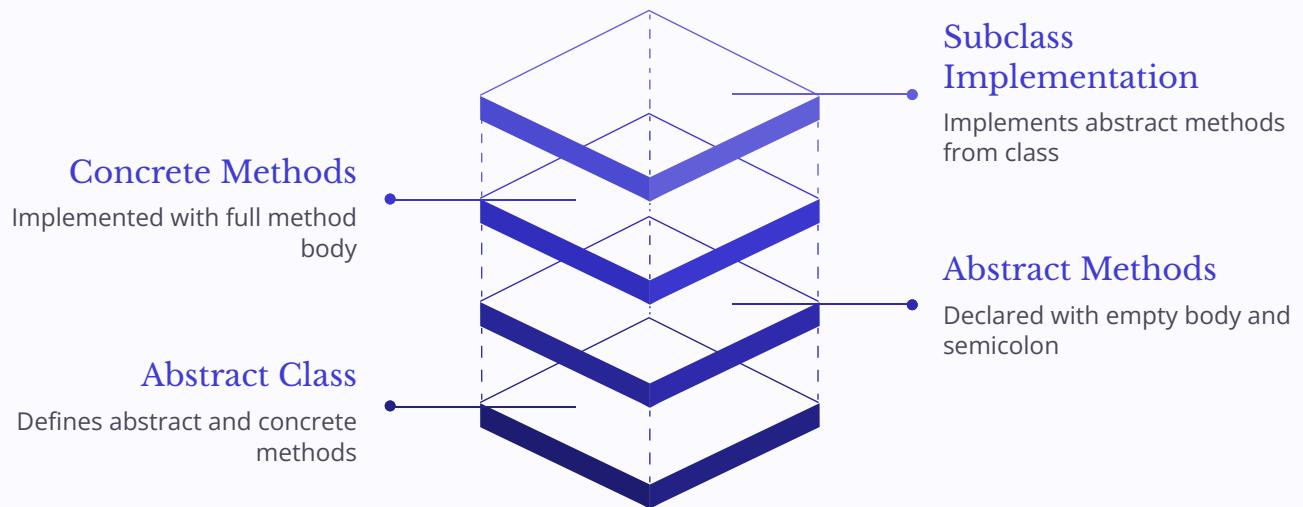
Vì không có body, abstract method kết thúc bằng dấu chấm phẩy.

Chỉ trong abstract class hoặc interface

Abstract method chỉ có thể tồn tại trong abstract class hoặc interface.

So sánh Abstract Method và Concrete Method

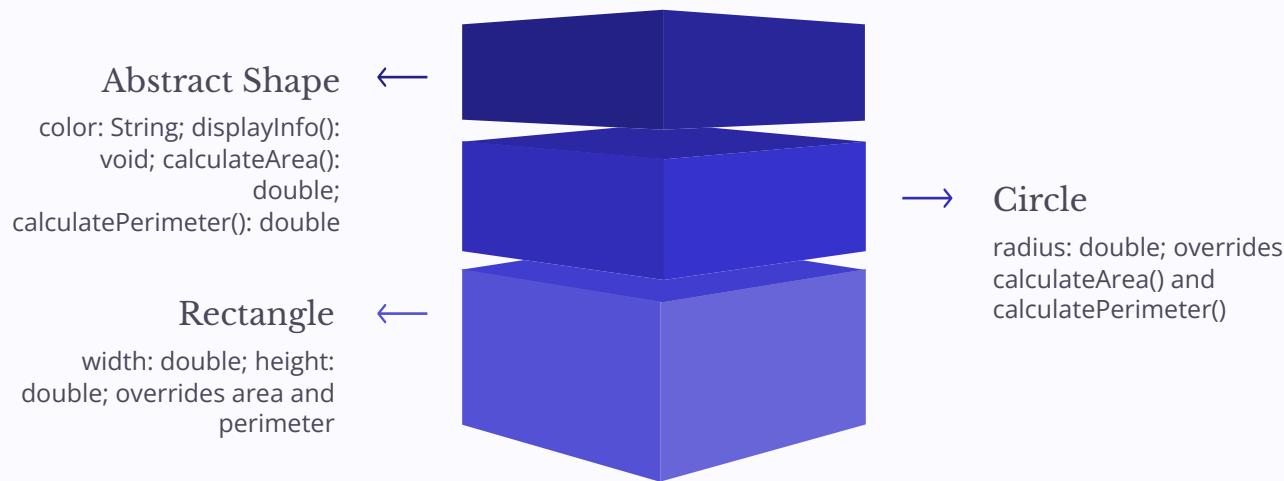
Một Abstract Class có thể chứa cả abstract methods (chưa hoàn thiện) và concrete methods (đã hoàn thiện).



Phân Tích Thiết Kế Abstract Class

Abstract Class đóng vai trò quan trọng trong việc định hình kiến trúc ứng dụng, đặc biệt khi áp dụng các Design Pattern như Template Method. Hãy cùng phân tích cách thiết kế này hoạt động hiệu quả.

Cấu trúc phân cấp các Hình học



Template Method Pattern

Đây là cách áp dụng **Template Method Pattern** trong thực tế. Người Kiến trúc sư (Abstract Class - **Shape**) quy định luật chơi chung: "Đã là một Hình thì bắt buộc phải tính được Diện tích và Chu vi. Tôi không biết công thức cụ thể, nhưng các anh (Subclass) bắt buộc phải có." Người Thợ (Concrete Classes - **Circle, Rectangle**) tuân thủ: "Tôi là Hình Tròn, công thức của tôi là πr^2 . Tôi đã điền vào chỗ trống."

✓ Lợi ích cốt lõi của thiết kế này

Tính Thông nhât & Ràng buộc

Abstract Class đảm bảo tất cả các lớp con (concrete classes) đều phải triển khai các phương thức trừu tượng đã định nghĩa. Điều này tạo ra một hợp đồng chung, giúp duy trì tính nhất quán trong cấu trúc code.

Tính Mở rộng (Extensibility)

Bạn có thể dễ dàng thêm các loại hình mới (ví dụ: Triangle, Hexagon) mà không cần thay đổi code hiện có. Chỉ cần tạo một lớp con mới kế thừa từ Shape và triển khai các phương thức trừu tượng.

Tính Linh hoạt (Flexibility) thông qua Polymorphism

Code xử lý có thể làm việc với đối tượng kiểu Shape mà không cần biết chính xác đó là Circle hay Rectangle. Mỗi đối tượng sẽ tự động gọi phương thức triển khai riêng của nó.

Minh họa Sức mạnh của Polymorphism

Bạn có thể tạo một List chứa đủ loại hình và xử lý thống nhất mà không cần quan tâm đến loại cụ thể của từng đối tượng:

```
List<Shape> shapes = new ArrayList<>();
shapes.add(new Circle("Red", 5.0));
shapes.add(new Rectangle("Blue", 4.0, 6.0));
shapes.add(new Triangle("Green", 3.0, 4.0, 5.0)); // Giả sử có lớp Triangle

// Duyệt qua và tính toán - Sức mạnh của Polymorphism!
for (Shape s : shapes) {
    s.displayInfo(); // Mỗi hình tự in thông tin của mình
    System.out.println("Diện tích: " + s.calculateArea()); // Mỗi hình tự tính diện tích theo công thức riêng
    System.out.println("Chu vi: " + s.calculatePerimeter()); // Mỗi hình tự tính chu vi theo công thức riêng
    System.out.println("----");
}
```

💡 **Đây là sức mạnh tối thượng của Polymorphism:** Code xử lý (vòng lặp for-each) không cần quan tâm đến kiểu cụ thể của object. Mỗi object tự biết cách thực hiện method của mình. Điều này làm code cực kỳ linh hoạt và dễ mở rộng.

Điều gì xảy ra nếu KHÔNG dùng Abstract Class?

Nếu không có Abstract Class và abstract method, bạn sẽ phải đổi mặt với nhiều vấn đề:



Thiếu Ràng buộc

Không có cách nào để ép buộc các lớp hình học khác phải có phương thức calculateArea() hoặc calculatePerimeter(). Developer có thể quên hoặc đặt tên khác nhau, dẫn đến không nhất quán.



Code Trùng lặp & Phức tạp

Để xử lý danh sách các hình, bạn sẽ phải dùng các câu lệnh instanceof và ép kiểu (casting), dẫn đến code dài dòng, dễ gây lỗi và khó bảo trì.

```
// Ví dụ code phức tạp và kém linh hoạt:
for (Object obj : objects) {
    if (obj instanceof Circle) {
        Circle c = (Circle) obj;
        System.out.println(c.calculateArea());
    } else if (obj instanceof Rectangle) {
        Rectangle r = (Rectangle) obj;
        System.out.println(r.calculateArea());
    }
    // ... và cứ thế thêm các loại hình mới
}
```



Khó bảo trì & Mở rộng

Mỗi khi thêm một loại hình mới, bạn sẽ phải quay lại sửa đổi tất cả các đoạn code có cấu trúc if-else if này, vi phạm nguyên tắc "Open/Closed Principle" (mở rộng nhưng đóng để chỉnh sửa).

So Sánh Abstract Class vs Concrete Class

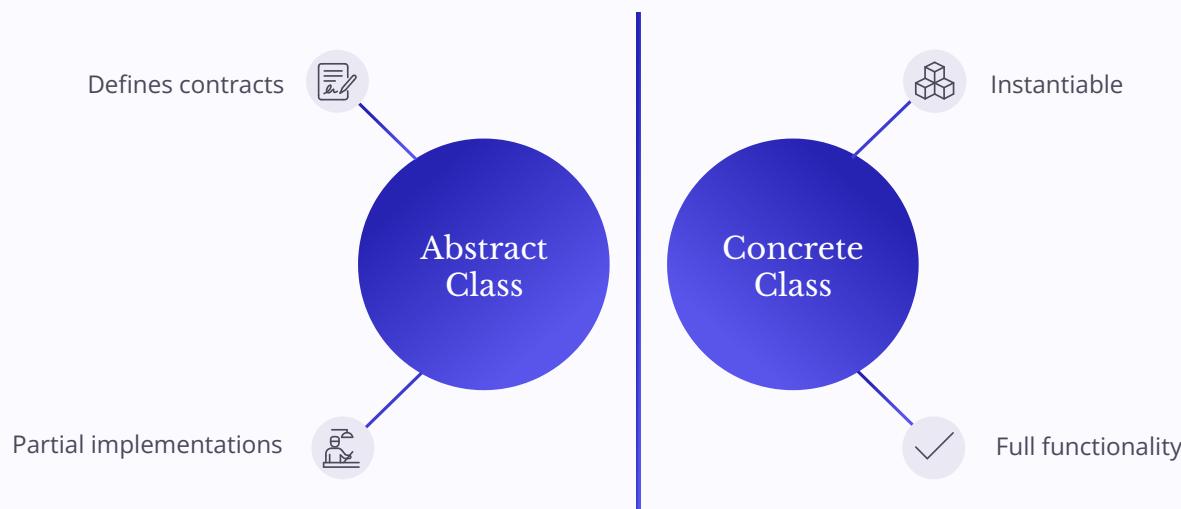
Đặc điểm	Abstract Class	Concrete Class
Mục đích chính	Định nghĩa khung sườn, hợp đồng chung	Cung cấp triển khai đầy đủ, cụ thể
Tạo object (Instantiate)	✗ Không thể (cần lớp con cụ thể)	✓ Có thể (object hoàn chỉnh)
Abstract methods	✓ Có thể có (không thân hàm)	✗ Không thể (phải có thân hàm)
Concrete methods	✓ Có thể có (có thân hàm)	✓ Có thể có (có thân hàm)
Fields (Biến)	✓ Có thể có	✓ Có thể có
Constructor	✓ Có thể có (được gọi từ lớp con)	✓ Có thể có
Kế thừa	Lớp con phải extends và triển khai abstract methods	Có thể extends lớp khác
Từ khóa	abstract	Không có

Giải thích sự khác biệt:

Một **Abstract Class** giống như một bản thiết kế chưa hoàn chỉnh; nó đặt ra các quy tắc và định nghĩa một số phần chung, nhưng để tạo ra một đối tượng thực sự, bạn cần một lớp cụ thể kế thừa nó. Ví dụ, bạn không thể tạo một "Hình" chung chung, nhưng có thể tạo một "Hình Tròn" hoặc "Hình Chữ Nhật" từ bản thiết kế "Hình" đó.

Ngược lại, một **Concrete Class** là một bản thiết kế hoàn chỉnh, sẵn sàng để được tạo thành đối tượng và sử dụng ngay lập tức. Nó đã triển khai tất cả các phương thức và không có bất kỳ "chỗ trống" nào cần được điền thêm.

Khi nào dùng Abstract Class hay Concrete Class?



Khi nào dùng Abstract Class?

✓ Có code chung

Khi nhiều subclass có code chung, đặt vào abstract class để tái sử dụng.

✓ Template Method Pattern

Khi muốn định nghĩa flow chung nhưng để subclass implement chi tiết.

✓ Cần fields và constructors

Khi cần lưu trữ state (fields) và khởi tạo thông qua constructor.

Ví dụ: Template Method Pattern

```

public abstract class DataProcessor {
    // Template method - định nghĩa flow
    public final void process() {
        readData();
        processData(); // Abstract - mỗi subclass implement khác
        saveData();
    }

    protected abstract void processData();

    protected void readData() {
        System.out.println("Reading data...");
    }

    protected void saveData() {
        System.out.println("Saving data...");
    }
}
  
```

Interface – Hợp Đồng Hành Vi

Interface đóng vai trò như một "hợp đồng hành vi" trong lập trình hướng đối tượng, chỉ định các phương thức mà một lớp phải triển khai mà không cung cấp bất kỳ chi tiết cài đặt nào. Không như **Abstract Class** có thể chứa cài đặt một phần và các trường dữ liệu, Interface hoàn toàn tập trung vào việc định nghĩa các khả năng, đảm bảo các lớp tuân thủ một chuẩn mực nhất định. Nó là bản kế hoạch thuận túy cho những gì một đối tượng có thể làm.

Đặc điểm của Interface

Chỉ định nghĩa method signatures

Không có implementation (body), chỉ khai báo tên method, tham số và return type.

Không có constructors

Vì không thể tạo object từ interface.

Không có fields thông thường

Chỉ có constants (public static final) - trước Java 8.

Class phải implement tất cả methods

Class implement interface phải cung cấp implementation cho tất cả methods.

Ví dụ cơ bản

```
// Interface định nghĩa hành vi
public interface Flyable {
    void fly();
}

public interface Swimmable {
    void swim();
}
```

```
// Class implement interface
public class Bird
implements Flyable {
@Override
public void fly() {
System.out.println(
"Bird is flying"
);
}
}
```

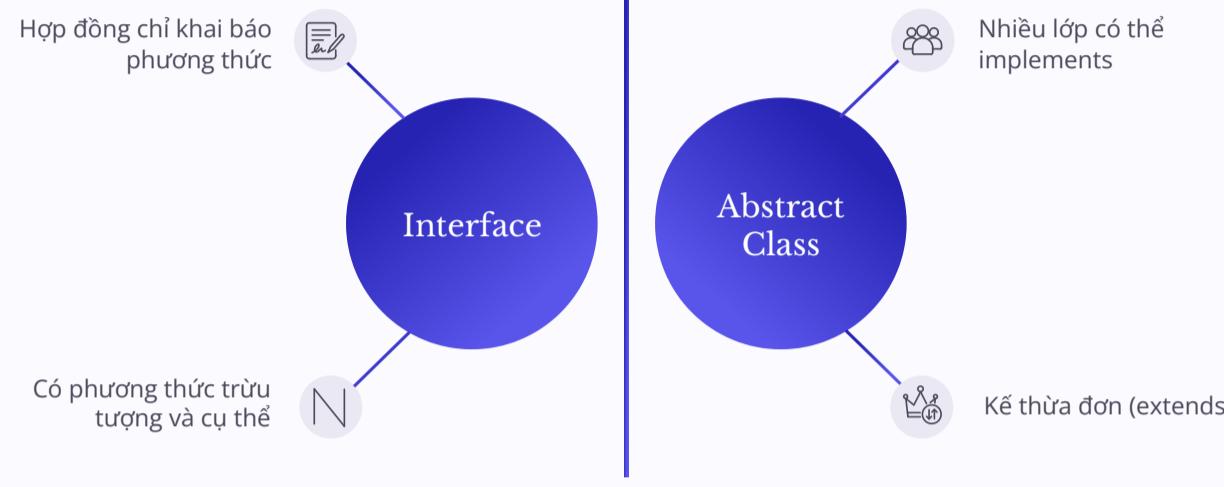
```
public class Duck implements
Flyable, Swimmable {
@Override
public void fly() {
System.out.println(
"Duck is flying"
);
}
@Override
public void swim() {
System.out.println(
"Duck is swimming"
);
}
}
```

Interface vs Abstract Class

Đặc điểm	Interface	Abstract Class
Từ khóa	interface	abstract class
Fields	Chỉ constants (public static final)	Có thể có fields (public, protected, private)
Methods	Chỉ abstract (trước Java 8), có thể có default, static, private	Abstract + Concrete
Constructor	✗ Không có	✓ Có thể có
Multiple inheritance	✓ Implement nhiều interface	✗ Chỉ extends 1 class
Mục đích	Định nghĩa hành vi	Định nghĩa cấu trúc, bản chất
Quan hệ	"CAN-DO" (có thể làm)	"IS-A" (là một)
Tính truy cập (Visibility)	Các thành viên (methods, fields) mặc định là public	Có thể có public, protected, private
Khởi tạo đối tượng	✗ Không thể	✗ Không thể
Tính đa hình	Hỗ trợ đa hình thông qua việc implement	Hỗ trợ đa hình thông qua việc kế thừa

Sự khác biệt quan trọng nhất: Interface thể hiện **khả năng/hành vi** (CAN-DO), còn Abstract Class thể hiện **bản chất/định danh** (IS-A).

So sánh trực quan



Khi nào dùng Interface hay Abstract Class?

1. Căn định nghĩa "khả năng" (CAN-DO)?
Nếu bạn muốn định nghĩa một tập hợp các hành vi mà nhiều lớp không liên quan có thể chia sẻ (ví dụ: Flyable, Saveable), hãy dùng **Interface**.
2. Căn định nghĩa "là một loại" (IS-A)?
Nếu bạn muốn tạo một lớp cơ sở với một số cài đặt mặc định và một số hành vi bắt buộc phải được triển khai bởi các lớp con, hãy dùng **Abstract Class**. (ví dụ: Shape, Vehicle).
3. Cần kế thừa nhiều loại?
Nếu một lớp cần thể hiện nhiều "khả năng" từ các nguồn khác nhau, **Interface** cho phép đa kế thừa hành vi. Abstract Class chỉ cho phép đơn kế thừa.
4. Cần chứa trạng thái (fields) hoặc constructor?
Nếu bạn cần lưu trữ trạng thái (fields) hoặc có một constructor để khởi tạo trạng thái, hãy chọn **Abstract Class**.

Ví dụ minh họa chi tiết

Abstract Class: Animal

```
// Abstract Class: Animal.java
public abstract class Animal {
    String name; // Có thể có fields

    public Animal(String name) { // Có thể có constructor
        this.name = name;
    }

    public void eat() { // Concrete method
        System.out.println(name + " is eating.");
    }

    public abstract void makeSound(); // Abstract method
}
```

Interface: Swimmable

```
// Interface: Swimmable.java
public interface Swimmable {
    void swim(); // Mặc định public abstract

    default void dive() { // Default method (Java 8+)
        System.out.println("Diving deep!");
    }
}
```

Class kế thừa Abstract Class

```
// Concrete Class: Dog.java
public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println(name + " barks.");
    }
}
```

Class implement Interface

```
// Concrete Class: Fish.java
public class Fish implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Fish is swimming.");
    }
    // Có thể không implement dive() nếu không muốn thay đổi hành vi mặc định
}
```

Class kế thừa và implement

```
// Concrete Class: Duck.java
public class Duck extends Animal implements Swimmable {
    public Duck(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println(name + " quacks.");
    }

    @Override
    public void swim() {
        System.out.println(name + " is swimming.");
    }
}
```

Sử dụng đa hình

```
// Main class
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog("Buddy");
        myDog.eat(); // Từ Animal
        myDog.makeSound(); // Từ Dog

        Swimmable myFish = new Fish();
        myFish.swim(); // Từ Fish
        myFish.dive(); // Từ default method của Swimmable

        Animal myDuck = new Duck("Daffy");
        myDuck.eat();
        myDuck.makeSound();
        ((Swimmable) myDuck).swim(); // Sử dụng khả năng Swimmable
    }
}
```

Khi Nào Dùng Interface?

✓ DÙNG INTERFACE khi



1. Định nghĩa hành vi chung (CAN-DO)

Khi bạn muốn định nghĩa một tập hợp các khả năng hoặc hành vi mà nhiều class không liên quan có thể có.

Ví dụ thực tế: Interface Flyable định nghĩa khả năng "bay" mà không quan tâm đến "ai" hay "cái gì" đang bay. Giống như một "hợp đồng" quy định những việc phải làm, bất kỳ class nào "ký hợp đồng" (implement interface) đều phải thực hiện những cam kết đó.



```
public interface Flyable {  
    void fly(); // Mọi thứ bay đều phải có hành động này  
}
```



2. Nhiều class không liên quan cần cùng một hành vi

Khi các class không có quan hệ kế thừa tự nhiên nhưng cần chia sẻ một hành vi chung.

Ví dụ thực tế: Chim, máy bay, và Superman đều có thể bay, nhưng chúng không có quan hệ kế thừa trực tiếp. Interface Flyable giúp nhóm các thực thể này dựa trên hành vi chung mà không cần thay đổi cấu trúc phân cấp class của chúng.



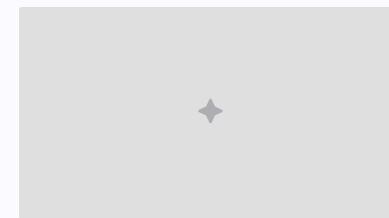
```
// Bird, Airplane, Superman đều triển khai Flyable  
public class Bird implements Flyable {  
    @Override public void fly() { System.out.println("Bird flies with  
wings"); }  
}  
public class Airplane implements Flyable {  
    @Override public void fly() { System.out.println("Airplane flies  
with engines"); }  
}  
public class Superman implements Flyable {  
    @Override public void fly() { System.out.println("Superman flies  
with superpowers"); }  
}
```



3. Cần đa kế thừa hành vi (Multiple Inheritance of Behavior)

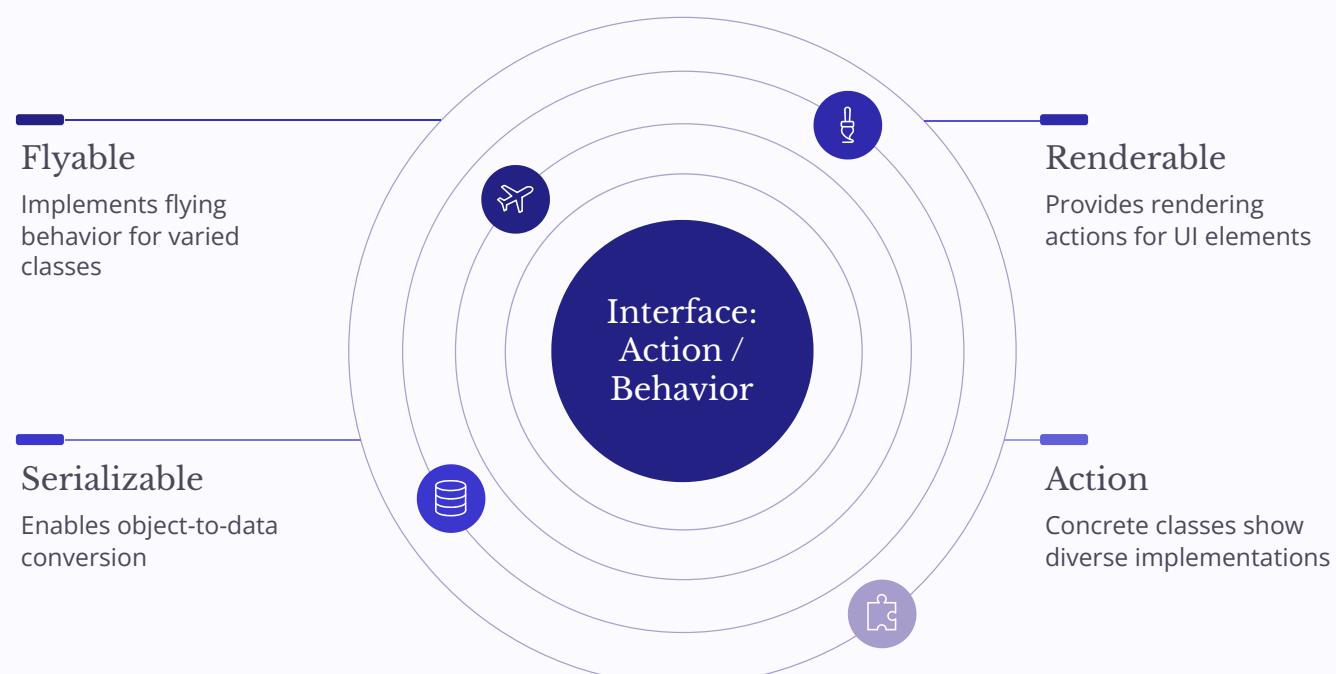
Khi một class cần "kế thừa" nhiều tập hợp hành vi từ các nguồn khác nhau, điều mà Java không cho phép với class.

Ví dụ thực tế: Một con Vịt (Duck) vừa là Động vật (Animal - thông qua kế thừa Abstract Class) vừa có khả năng bơi (Swimmable) và bay (Flyable). Interface cho phép một class có nhiều "vai trò" hoặc "khả năng" đồng thời.



```
public class Duck extends Animal implements Swimmable, Flyable  
{  
    // ... cài đặt các phương thức của Animal, Swimmable, Flyable  
}
```

Sự Linh Hoạt của Interface



Interface cho phép chúng ta xây dựng các hệ thống linh hoạt và dễ mở rộng, nơi các thành phần có thể tương tác dựa trên hành vi chung mà không cần biết chi tiết triển khai cụ thể. Điều này rất quan trọng trong thiết kế phần mềm hướng đối tượng.

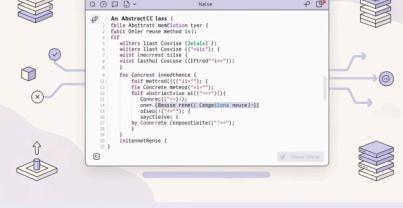
Khi Nào Dùng Abstract Class?

DÙNG ABSTRACT CLASS khi



1. Có code chung cho subclass

Khi bạn có nhiều class có chung các phương thức hoặc thuộc tính mà bạn muốn tập trung vào một nơi để tái sử dụng và tránh trùng lặp code. Abstract class cho phép định nghĩa các phương thức không có thân (abstract methods) mà subclass phải triển khai, và cả các phương thức có thân (concrete methods) có thể được kế thừa trực tiếp.



```
// Abstract class Animal với phương thức chung displayInfo()
public abstract class Animal {
    protected String name;
    protected int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

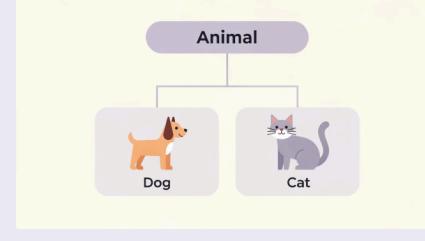
    // Phương thức cụ thể, được kế thừa và tái sử dụng
    public void displayInfo() {
        System.out.println("Tên: " + name + ", Tuổi: " + age + " năm");
    }

    public abstract void makeSound(); // Phương thức abstract,
    subclass phải triển khai
}
```



2. Quan hệ "IS-A" rõ ràng

Khi các class có một mối quan hệ phân cấp tự nhiên và logic, trong đó một class "là một loại" của class khác (ví dụ: Chó là một loại Động vật). Abstract class là nền tảng tốt để xây dựng một hệ thống phân cấp class vững chắc.



```
// Dog IS-A Animal
public class Dog extends Animal {
    public Dog(String name, int age) {
        super(name, age);
    }

    @Override
    public void makeSound() {
        System.out.println("Gâu gâu!");
    }
}

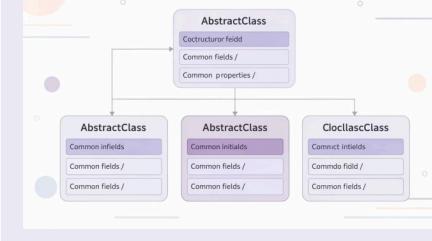
// Cat IS-A Animal
public class Cat extends Animal {
    public Cat(String name, int age) {
        super(name, age);
    }

    @Override
    public void makeSound() {
        System.out.println("Meo meo!");
    }
}
```



3. Cần fields và constructors

Khi bạn muốn định nghĩa các trường (fields) và constructor để khởi tạo trạng thái chung cho tất cả các subclass. Abstract class có thể có constructor, nhưng chúng chỉ được gọi thông qua constructor của subclass bằng super().



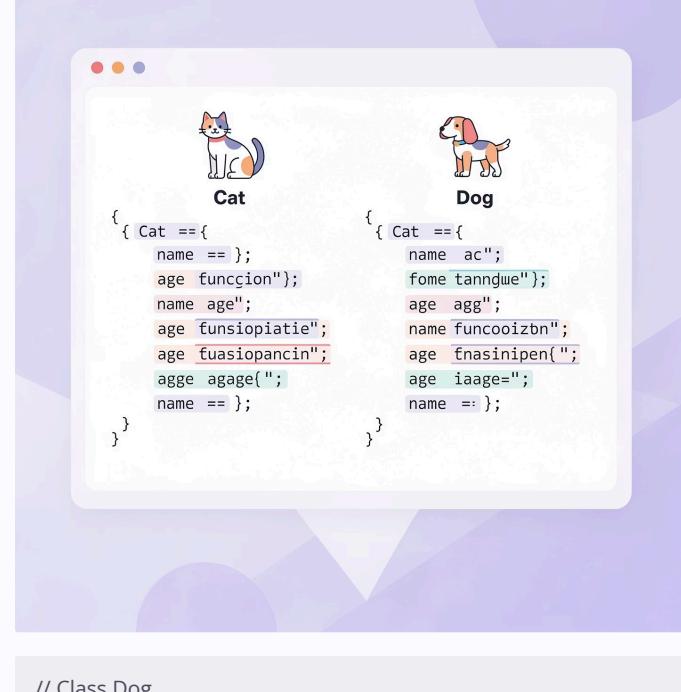
```
// Constructor của Animal khởi tạo các thuộc tính chung
public abstract class Animal {
    protected String name;
    protected int age;

    public Animal(String name, int age) { // Constructor
        this.name = name;
        this.age = age;
    }
    // ...
}
```

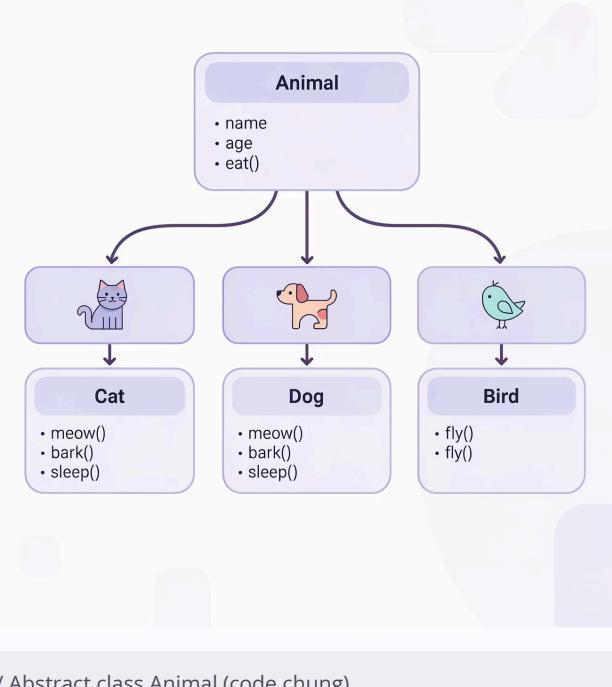
```
public class Dog extends Animal {
    private String breed;
    public Dog(String name, int age, String breed) {
        super(name, age); // Gọi constructor của lớp cha để khởi
        tạo name và age
        this.breed = breed;
    }
    // ...
}
```

So sánh: VỚI và KHÔNG có Abstract Class

Không dùng Abstract Class



Dùng Abstract Class (Animal)



```
// Class Dog
public class Dog {
    protected String name;
    protected int age;

    public Dog(String name, int age) { // Trùng lặp
        this.name = name;
        this.age = age;
    }

    public void makeSound() { System.out.println("Woof!"); }
    public void displayInfo() { // Trùng lặp
        System.out.println("Tên: " + name + ", Tuổi: " + age + " năm");
    }
}

// Class Cat
public class Cat {
    protected String name;
    protected int age;

    public Cat(String name, int age) { // Trùng lặp
        this.name = name;
        this.age = age;
    }

    public void makeSound() { System.out.println("Meow!"); }
    public void displayInfo() { // Trùng lặp
        System.out.println("Tên: " + name + ", Tuổi: " + age + " năm");
    }
}
```

Trong ví dụ trên, các trường name, age, constructor và phương thức displayInfo() bị lặp lại ở cả Dog và Cat. Điều này dẫn đến sự trùng lặp code, khó bảo trì và mở rộng.

```
// Abstract class Animal (code chung)
public abstract class Animal {
    protected String name;
    protected int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("Tên: " + name + ", Tuổi: " + age + " năm");
    }

    public abstract void makeSound();
}

// Class Dog (kế thừa Animal)
public class Dog extends Animal {
    public Dog(String name, int age) { super(name, age); }

    @Override
    public void makeSound() { System.out.println("Gâu gâu!"); }
}

// Class Cat (kế thừa Animal)
public class Cat extends Animal {
    public Cat(String name, int age) { super(name, age); }

    @Override
    public void makeSound() { System.out.println("Meo meo!"); }
}
```

Với abstract class Animal, các thuộc tính và phương thức chung được định nghĩa một lần duy nhất. Dog và Cat chỉ cần kế thừa Animal và triển khai phương thức makeSound() riêng của chúng. Điều này giúp loại bỏ trùng lặp, tăng khả năng tái sử dụng và dễ dàng quản lý code hơn.

Lợi ích của việc tái sử dụng code

Subclasses
Dog, Cat, Bird kế thừa tự động

Lợi Ích
Giảm trùng lặp, dễ bảo trì

Animal (Abstract)
Định nghĩa name, age, displayInfo()

```
"interface {
    void methodA();
    void methodB();
}
```

Modern Java: Default Methods và Static Methods

Default Methods (Java 8+)

Từ Java 8, interface có thể chứa **default methods** - các methods có implementation sẵn. Điều này giúp mở rộng chức năng của interface mà không làm ảnh hưởng đến các class đã implement trước đó.

```
public interface Flyable {
    // Abstract method (bắt buộc implement)
    void fly();

    // Default method (có implementation, không bắt buộc override)
    default void takeOff() {
        System.out.println("Ready for take off...");
    }

    default void land() {
        System.out.println("Landing gracefully...");
    }
}
```

```
// Class Bird implement Flyable, chỉ cần triển khai fly()
public class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird is soaring through the sky.");
    }

    // Bird sử dụng default takeOff() và land()
}

// Class Plane implement Flyable, có thể override default methods
public class Plane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Plane is flying at high altitude.");
    }

    @Override
    public void takeOff() {
        System.out.println("Plane is taking off from the runway.");
    }

    // Plane sử dụng default land()
}
```

```
// Sử dụng
Bird bird = new Bird();
bird.fly(); // "Bird is soaring through the sky."
bird.takeOff(); // "Ready for take off..." (default method)
bird.land(); // "Landing gracefully..." (default method)

Plane plane = new Plane();
plane.fly(); // "Plane is flying at high altitude."
plane.takeOff(); // "Plane is taking off from the runway." (overridden)
plane.land(); // "Landing gracefully..." (default method)
```

Lợi ích của Default Methods

Mở rộng không phá vỡ code cũ

Thêm methods mới vào interface mà không làm break các class đã implement trước đó, đảm bảo tính tương thích ngược.

Code chung cho tất cả implementers

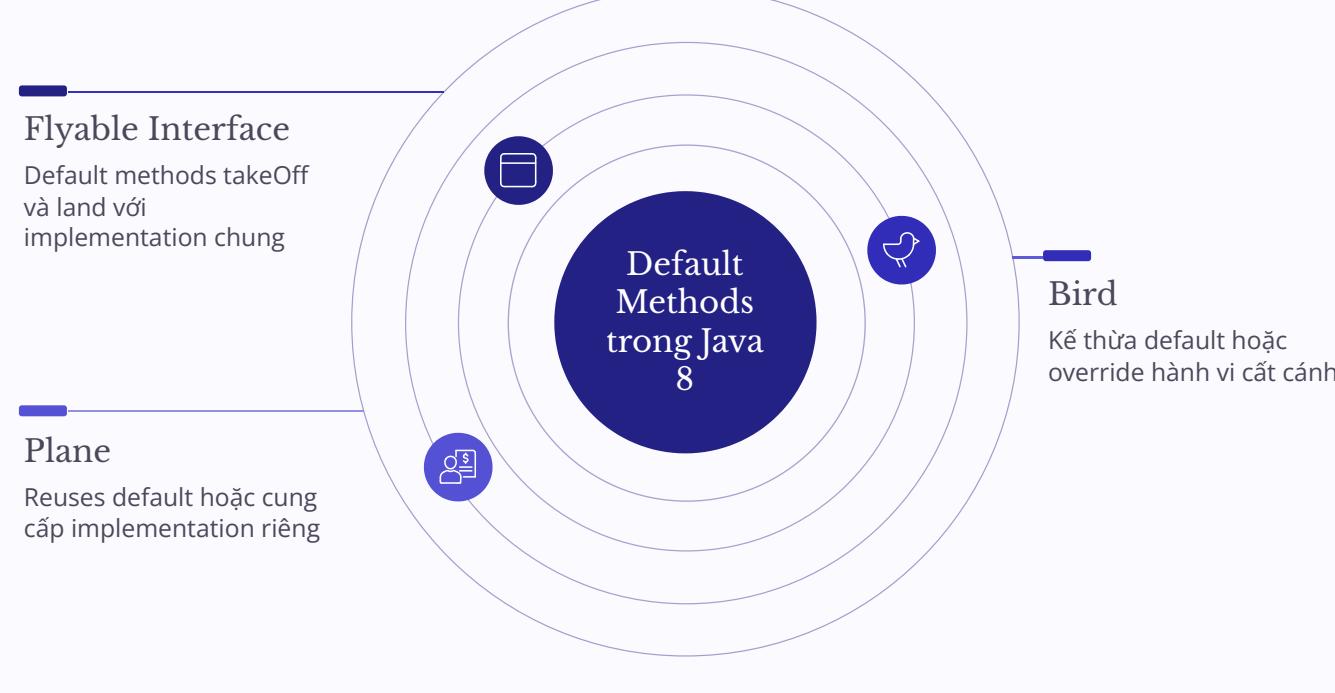
Cung cấp implementation mặc định mà tất cả implementers có thể sử dụng, giảm trùng lặp code.

Linh hoạt

Các class implement vẫn có thể override các default methods để cung cấp hành vi riêng biệt khi cần.

Tương thích ngược: Default methods được giới thiệu để giải quyết vấn đề "gây" code khi thêm phương thức mới vào interface hiện có. Chúng cho phép các interface phát triển mà không yêu cầu tất cả các lớp implement phải cập nhật ngay lập tức.

Cách Default Methods Cung cấp Linh Hoạt và Triển Khai



Ví dụ thực tế: Cập nhật thư viện API

1

Giai đoạn 1: Interface ban đầu

Một thư viện cung cấp interface PaymentGateway với phương thức processPayment().

```
public interface PaymentGateway {
    void processPayment(double amount);
}
```

2

Giai đoạn 2: Yêu cầu tính năng mới

Cần thêm tính năng xác thực giao dịch authenticateTransaction() vào interface.

```
public interface PaymentGateway {
    void processPayment(double amount);
    default boolean authenticateTransaction() {
        System.out.println("Default authentication logic.");
        return true;
    }
}
```

3

Lợi ích thực tế

Các class đã implement PaymentGateway như PayPalGateway, StripeGateway không bị lỗi biên dịch. Chúng tự động kế thừa logic xác thực mặc định. Những class cần logic xác thực tùy chỉnh có thể override authenticateTransaction().

Static Methods trong Interface

Static methods trong interface (Java 8+) là các utility methods liên quan đến interface, cung cấp các chức năng hữu ích mà không yêu cầu tạo một instance của interface.

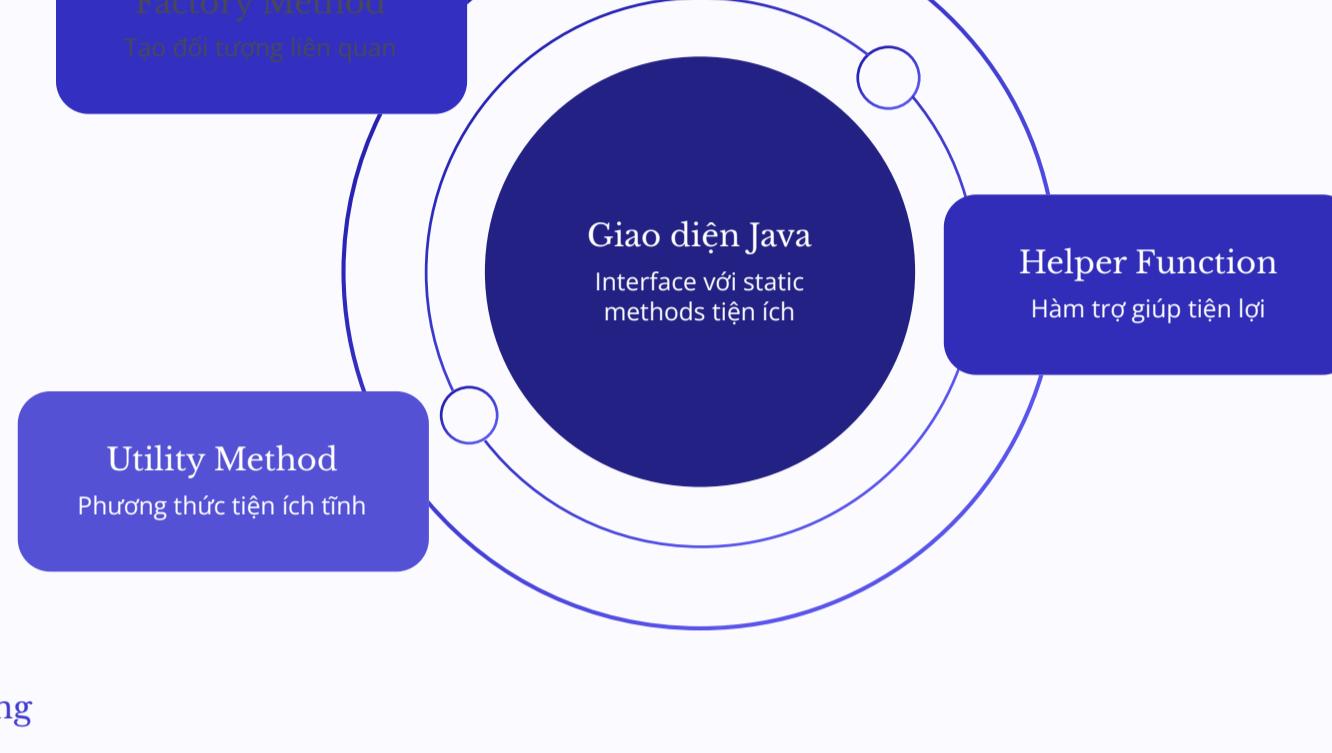
```
public interface MathOperation {  
    int calculate(int a, int b);  
  
    // Static method - gọi trực tiếp từ interface  
    static MathOperation add() {  
        return (a, b) -> a + b;  
    }  
  
    static MathOperation subtract() {  
        return (a, b) -> a - b;  
    }  
  
    static MathOperation multiply() {  
        return (a, b) -> a * b;  
    }  
}
```

```
// Sử dụng - gọi trực tiếp từ interface  
MathOperation addOp = MathOperation.add();  
int result = addOp.calculate(5, 3); // 8
```

```
MathOperation multiplyOp = MathOperation.multiply();  
int result2 = multiplyOp.calculate(4, 7); // 28
```

Static methods trong interface thường được dùng để:

- Tạo factory methods (như ví dụ trên)
- Cung cấp utility methods liên quan
- Tạo helper methods cho interface



Ví dụ thực tế và Ứng dụng

Static methods trong interface rất hữu ích cho các **factory methods**, giúp tạo và trả về các instance của interface hoặc functional interface. Như ví dụ MathOperation ở trên, chúng ta không cần viết class cụ thể cho từng phép toán mà có thể lấy trực tiếp từ interface.

Một ví dụ thực tế khác từ Java Standard Library là Comparator.comparing(), một static method trong interface Comparator dùng để tạo một Comparator từ một hàm trích xuất khóa:

```
import java.util.Arrays;  
import java.util.Comparator;  
import java.util.List;  
  
class User {  
    String name;  
    int age;  
    public User(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() { return name; }  
    public int getAge() { return age; }  
    @Override  
    public String toString() { return name + " (" + age + ")"; }  
}  
  
public class StaticMethodExample {  
    public static void main(String[] args) {  
        List<User> users = Arrays.asList(  
            new User("Alice", 30),  
            new User("Bob", 25),  
            new User("Charlie", 35)  
        );  
  
        // Sử dụng static method Comparator.comparing()  
        users.sort(Comparator.comparing(User::getName));  
        System.out.println("Sorted by name: " + users);  
        // Output: Sorted by name: [Alice (30), Bob (25), Charlie (35)]  
  
        users.sort(Comparator.comparing(User::getAge));  
        System.out.println("Sorted by age: " + users);  
        // Output: Sorted by age: [Bob (25), Alice (30), Charlie (35)]  
    }  
}
```

So sánh: Static Methods trong Interface vs Utility Class riêng biệt

Static Methods trong Interface

Các phương thức này được gắn trực tiếp với interface, làm cho chúng tự nhiên là một phần của API interface.

```
public interface MyProcessor {  
    void process(String data);
```

```
    static MyProcessor defaultProcessor() {  
        return (data) -> System.out.println("Processing: " + data);  
    }  
}
```

```
// Sử dụng: MyProcessor.defaultProcessor().process("test");
```

Ưu điểm:

- Tăng cường tính gắn kết, rõ ràng về mặt ngữ nghĩa: Các phương thức utility liên quan chặt chẽ với interface được đặt cùng chỗ.

- Không cần tạo thêm một class riêng biệt chỉ để chứa các phương thức utility.

Nhược điểm:

- Không thể chứa trạng thái (state) vì interface không có instance fields.

- Không thể bị override bởi các class implement.

Utility Class riêng biệt

Các phương thức utility được nhóm trong một class riêng, thường là final và có constructor private.

```
public final class ProcessorUtils {  
    private ProcessorUtils() {} // Ngăn không cho tạo instance
```

```
    public static MyProcessor getDefaultProcessor() {  
        return (data) -> System.out.println("Processing: " + data);  
    }  
}
```

```
// Sử dụng: ProcessorUtils.getDefaultProcessor().process("test");
```

Ưu điểm:

- Có thể chứa trạng thái (static fields) nếu cần thiết (mặc dù nên hạn chế).

- Được sử dụng rộng rãi và quen thuộc hơn trong các phiên bản Java cũ.

Nhược điểm:

- Tạo ra một class mới chỉ để chứa các phương thức, có thể làm phân tán code khi các phương thức đó rất cụ thể cho một interface nào đó.

- Ít rõ ràng hơn về mối quan hệ trực tiếp với interface.

Khi nào nên sử dụng Static Methods trong Interface?

Factory Methods

Khi bạn muốn cung cấp một cách tiêu chuẩn để tạo ra các instance của chính interface đó (đặc biệt là các functional interface), như Comparator.comparing().

Helper Utilities

Khi có các phương thức utility nhỏ, tự chứa, và gắn liền với mặt ngữ nghĩa với interface, không cần trạng thái của một instance.

API Cohesiveness

Để giữ cho các chức năng liên quan chặt chẽ với interface trong cùng một định nghĩa, làm cho API dễ khám phá và sử dụng hơn.

Lưu ý: Static methods trong interface **không thể bị override** bởi implementer. Chúng thuộc về interface và chỉ có thể được gọi thông qua tên interface.

Multiple Interfaces

Một trong những ưu điểm lớn nhất của Interface là Java cho phép một class **implement nhiều interface cùng lúc**. Đây là cách Java giải quyết vấn đề multiple inheritance:

```
public interface Flyable {  
    void fly();  
}  
  
public interface Swimmable {  
    void swim();  
}  
  
public interface Walkable {  
    void walk();  
}
```

```
// Duck implement cả 3 interface  
public class Duck implements Flyable, Swimmable, Walkable {  
    @Override  
    public void fly() {  
        System.out.println("Duck is flying");  
    }  
  
    @Override  
    public void swim() {  
        System.out.println("Duck is swimming");  
    }  
  
    @Override  
    public void walk() {  
        System.out.println("Duck is walking");  
    }  
}
```

```
// Sử dụng  
Duck duck = new Duck();  
duck.fly(); // Duck có khả năng bay  
duck.swim(); // Duck có khả năng bơi  
duck.walk(); // Duck có khả năng đi bộ
```

```
// Polymorphism với nhiều interface  
Flyable flyable = duck;  
flyable.fly();
```

```
Swimmable swimmable = duck;  
swimmable.swim();
```

Điều này cho phép Duck có nhiều "khả năng" khác nhau mà không bị giới hạn bởi single inheritance của class.

Đa Hình (Polymorphism)

Polymorphism là gì?

Polymorphism (Đa hình) - từ tiếng Hy Lạp "poly" (nhiều) và "morph" (hình dạng) - là khả năng một object có thể có nhiều hình thái khác nhau. Đây là một trong những khái niệm mạnh mẽ nhất của OOP.

Trong Java, Polymorphism nghĩa là:

Một reference, nhiều object

Cùng một kiểu reference có thể trỏ đến nhiều loại object khác nhau.

Một method call, nhiều hành vi

Cùng một method call có thể thực thi code khác nhau tùy object thực tế.

Ví dụ minh họa

```
Animal animal1 = new Dog(); // Animal reference trỏ đến Dog object  
Animal animal2 = new Cat(); // Animal reference trỏ đến Cat object
```

```
animal1.makeSound(); // "Woof!" - Gọi method của Dog  
animal2.makeSound(); // "Meow!" - Gọi method của Cat
```



Phân tích:

- Cùng kiểu reference: Animal
- Cùng method call: makeSound()
- Nhưng kết quả khác nhau tùy object thực tế (Dog hay Cat)

Đây chính là **sức mạnh của Polymorphism**: Code linh hoạt, dễ mở rộng, và dễ bảo trì.

Upcasting và Downcasting

Upcasting (Ép kiểu lên)

Upcasting là ép kiểu từ subclass lên superclass. Đây là quá trình **tự động và an toàn**:

```
Dog dog = new Dog();
Animal animal = dog; // Upcasting - tự động
```

```
// Hoặc viết trực tiếp:
Animal animal = new Dog(); // Upcasting - tự động
```

✓ Tự động

Không cần cast rõ ràng, Java tự động thực hiện.

✓ An toàn

Dog luôn luôn là một Animal, nên không bao giờ gây lỗi.

⚠ Mất quyền truy cập

Chỉ có thể gọi methods của Animal, không gọi được methods riêng của Dog.

```
Animal animal = new Dog();
animal.makeSound(); //
```

Downcasting (Ép kiểu xuống)

Downcasting là ép kiểu từ superclass xuống subclass. Đây là quá trình **phải cast rõ ràng và có thể nguy hiểm**:

✓ Downcasting hợp lệ

```
Animal animal = new Dog();
// Object thực tế là Dog
```

```
Dog dog = (Dog) animal;
// Downcasting - phải cast
```

```
dog.bark(); //
```

Dynamic Binding (Runtime Polymorphism)

Dynamic Binding là gì?

Dynamic Binding (còn gọi là Late Binding hay Runtime Polymorphism) là cơ chế Java quyết định method nào được gọi **tại runtime** (lúc chương trình chạy) dựa trên **kiểu thực tế của object**, không phải kiểu của reference.

Ví dụ minh họa

```
public class Animal {  
    public void makeSound() {  
        System.out.println("Animal makes a sound");  
    }  
}  
  
public class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Dog barks: Woof!");  
    }  
}  
  
public class Cat extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Cat meows: Meow!");  
    }  
}
```

```
// Test Dynamic Binding  
Animal animal1 = new Dog(); // Reference: Animal, Object: Dog  
Animal animal2 = new Cat(); // Reference: Animal, Object: Cat  
  
animal1.makeSound(); // "Dog barks: Woof!" - Gọi method của Dog  
animal2.makeSound(); // "Cat meows: Meow!" - Gọi method của Cat
```

1

Compile-time

Compiler chỉ biết `animal1` là `Animal`, kiểm tra xem `Animal` có method `makeSound()` không.

2

Runtime

JVM nhìn vào object thực tế (`Dog`), phát hiện `Dog` đã override `makeSound()`, nên gọi version của `Dog`.

3

Kết quả

Method được gọi phụ thuộc vào object thực tế, không phải reference type.

Phân Tích Hậu Trường: Runtime Look-up

“

JVM làm gì khi gọi method?

Tại sao animal.makeSound() lại biết gọi đúng method? Đây là cơ chế **Method Lookup** của JVM:

1. **Bước 1:** Nhìn vào **Object thực tế** trong Heap (không phải kiểu reference)
2. **Bước 2:** Tìm xem Object đó có override method này không?
3. **Nếu có:** Chạy version override
4. **Nếu không:** Tìm lên superclass, tiếp tục cho đến khi tìm thấy

”

Ví dụ "lừa tình"

```
Animal myPet = new Dog(); // Biến kiểu Animal, nhưng chứa Dog object  
myPet.makeSound();
```

?

Method nào sẽ được gọi? Của Animal hay Dog?

✓

Đáp

Method của **Dog!** Vì JVM nhìn vào "cái ruột" (new Dog()), không nhìn "cái mác" (Animal myPet).

 **Ghi nhớ:** "Reference type quyết định methods **nào có thể gọi**. Object type quyết định method **nào được thực thi**." Đây là bản chất của Dynamic Binding.

Ứng Dụng: Loại Bỏ if/else Bằng Polymorphism

✗ Vấn đề: Code có nhiều if/else

```
// TỒI: Nhiều if/else
public void processAnimal(Animal animal) {
    if (animal instanceof Dog) {
        Dog dog = (Dog) animal;
        dog.bark();
        dog.fetch();
    } else if (animal instanceof Cat) {
        Cat cat = (Cat) animal;
        cat.meow();
        cat.scratch();
    } else if (animal instanceof Bird) {
        Bird bird = (Bird) animal;
        bird.tweet();
        bird.fly();
    }
    // Thêm loại mới → Phải sửa code này
}
```

Tại sao code này tồi?

✗ Vi phạm Open/Closed

Thêm loại động vật mới phải sửa code cũ (sẽ học Chương 4).

✗ Khó mở rộng

Code ngày càng dài, phức tạp khi thêm nhiều loại.

✗ Dễ lỗi

Có thể quên handle một case nào đó.

✓ Giải Pháp: Sử Dụng Polymorphism

Polymorphism (Đa hình) là một trong những khái niệm cốt lõi của Lập trình hướng đối tượng (OOP), cho phép chúng ta thay thế chuỗi câu lệnh `if/else` dài dòng bằng cách sử dụng kế thừa và ghi đè phương thức. Thay vì kiểm tra từng loại đối tượng, chúng ta định nghĩa một giao diện chung và để mỗi đối tượng tự quyết định cách thực thi hành vi của mình.

```
// Định nghĩa một lớp trừu tượng (abstract class) Animal
abstract class Animal {
    public abstract void makeSound(); // Phương thức trừu tượng để phát ra âm thanh
    public abstract void perform(); // Phương thức trừu tượng để thực hiện hành động
}

// Lớp con Dog kế thừa từ Animal
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Gau gau!");
    }

    @Override
    public void perform() {
        makeSound();
        System.out.println("Dog is fetching the ball.");
    }
}

// Lớp con Cat kế thừa từ Animal
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meo meo!");
    }

    @Override
    public void perform() {
        makeSound();
        System.out.println("Cat is scratching the furniture.");
    }
}

// Lớp con Bird kế thừa từ Animal
class Bird extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Chiu chiu!");
    }

    @Override
    public void perform() {
        makeSound();
        System.out.println("Bird is flying high.");
    }
}

// Phương thức xử lý động vật sử dụng đa hình
public class AnimalProcessor {
    public void processAnimal(Animal animal) {
        animal.perform(); // Chỉ cần gọi phương thức perform() của Animal
    }
}

public static void main(String[] args) {
    AnimalProcessor processor = new AnimalProcessor();

    Dog myDog = new Dog();
    Cat myCat = new Cat();
    Bird myBird = new Bird();

    System.out.println("--- Processing Dog ---");
    processor.processAnimal(myDog); // Output: Gau gau! Dog is fetching the ball.

    System.out.println("\n--- Processing Cat ---");
    processor.processAnimal(myCat); // Output: Meo meo! Cat is scratching the furniture.

    System.out.println("\n--- Processing Bird ---");
    processor.processAnimal(myBird); // Output: Chiu chiu! Bird is flying high.
}
```

⌚ Lợi ích của giải pháp này:

- Code ngắn gọn, dễ đọc và dễ hiểu.
- Dễ mở rộng: Khi thêm một loại động vật mới, bạn không cần sửa đổi phương thức `processAnimal()` hiện có.
- Tuân thủ Nguyên tắc Open/Closed (Mở để mở rộng, đóng để sửa đổi).
- Tận dụng sức mạnh của Polymorphism để xử lý các đối tượng thuộc các lớp con khác nhau một cách thống nhất.

So sánh code:

✗ Trước (if/else)

15+ dòng code để xử lý 3 loại động vật, cần sửa đổi khi thêm loại mới.

Thêm con vật mới dễ dàng:

Với Polymorphism, việc thêm một loại động vật mới trở nên đơn giản mà không ảnh hưởng đến code hiện có:

```
// Thêm con vật mới - KHÔNG cần sửa processAnimal()
class Cow extends Animal {
    @Override
    public void makeSound() { System.out.println("Moo!"); }

    @Override
    public void perform() {
        makeSound();
        System.out.println("Cow is grazing");
    }
}

// Sau đó có thể dùng:
// Cow myCow = new Cow();
// processor.processAnimal(myCow); // Output: Moo! Cow is grazing
```

✓ Sau (Polymorphism)

3 dòng code (`animal.perform()`) để xử lý mọi loại động vật, không cần sửa đổi khi thêm loại mới.

Ví Dụ Thực Tế: Payment System

✗ Cách cũ: if/else rôí răm

```
public void processPayment(  
    String paymentType,  
    double amount  
) {  
    if (paymentType  
        .equals("credit")) {  
        // Process credit card  
        System.out.println(  
            "Processing credit: "  
            + amount  
        );  
    } else if (paymentType  
        .equals("paypal")) {  
        // Process PayPal  
        System.out.println(  
            "Processing PayPal: "  
            + amount  
        );  
    } else if (paymentType  
        .equals("bitcoin")) {  
        // Process Bitcoin  
        System.out.println(  
            "Processing Bitcoin: "  
            + amount  
        );  
    }  
}
```

✓ Cách mới: Polymorphism

```
public interface  
    PaymentMethod {  
        void pay(double amount);  
    }  
  
public class CreditCard  
    implements PaymentMethod {  
        @Override  
        public void pay(  
            double amount  
        ) {  
            System.out.println(  
                "Paying " + amount +  
                " with credit card"  
            );  
        }  
    }  
  
public class PayPal  
    implements PaymentMethod {  
        @Override  
        public void pay(  
            double amount  
        ) {  
            System.out.println(  
                "Paying " + amount +  
                " with PayPal"  
            );  
        }  
    }  
}
```

```
// Code xử lý cực kỳ đơn giản  
public void processPayment(PaymentMethod method, double amount) {  
    method.pay(amount); // Polymorphism!  
}  
  
// Sử dụng  
processPayment(new CreditCard(), 100.0);  
processPayment(new PayPal(), 200.0);  
processPayment(new Bitcoin(), 300.0);
```

Composition Over Inheritance

Mặt Trái của Inheritance

Mặc dù Inheritance rất mạnh mẽ, nhưng nó cũng có những hạn chế và nguy cơ tiềm ẩn. Hãy cùng tìm hiểu những vấn đề này.

Vấn đề 1: Fragile Base Class Problem

Fragile Base Class Problem xảy ra khi thay đổi superclass có thể vô tình phá vỡ subclass:

```
public class Stack {
    private List<String> elements = new ArrayList<>();

    public void push(String item) {
        elements.add(item);
    }

    public String pop() {
        return elements.remove(elements.size() - 1);
    }

    public int size() {
        return elements.size();
    }
}
```

```
public class CountingStack extends Stack {
    private int count = 0;

    @Override
    public void push(String item) {
        super.push(item);
        count++;
    }

    @Override
    public int size() {
        return count; // Override để trả về count
    }
}
```

- **⚠ Vấn đề:** Nếu Stack thay đổi implementation của `size()` hoặc cách `push()` hoạt động, CountingStack có thể bị ảnh hưởng và hoạt động sai. Subclass quá phụ thuộc vào chi tiết implementation của superclass.

Vấn Đề 2: Inheritance Không Phù Hợp

✗ Ví dụ sai: Circle extends Point

Hãy xem xét một ví dụ kinh điển về việc sử dụng inheritance sai cách khi mô hình hóa các đối tượng hình học:

```
// Lớp Point đại diện cho một điểm trong không gian 2D
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
}

// Lớp Circle kế thừa từ Point
public class Circle extends Point {
    private double radius;

    public Circle(int x, int y, double radius) {
        super(x, y); // Circle kế thừa tọa độ x, y từ Point
        this.radius = radius;
    }

    public double getRadius() { return radius; }
    public void setRadius(double radius) { this.radius = radius; }

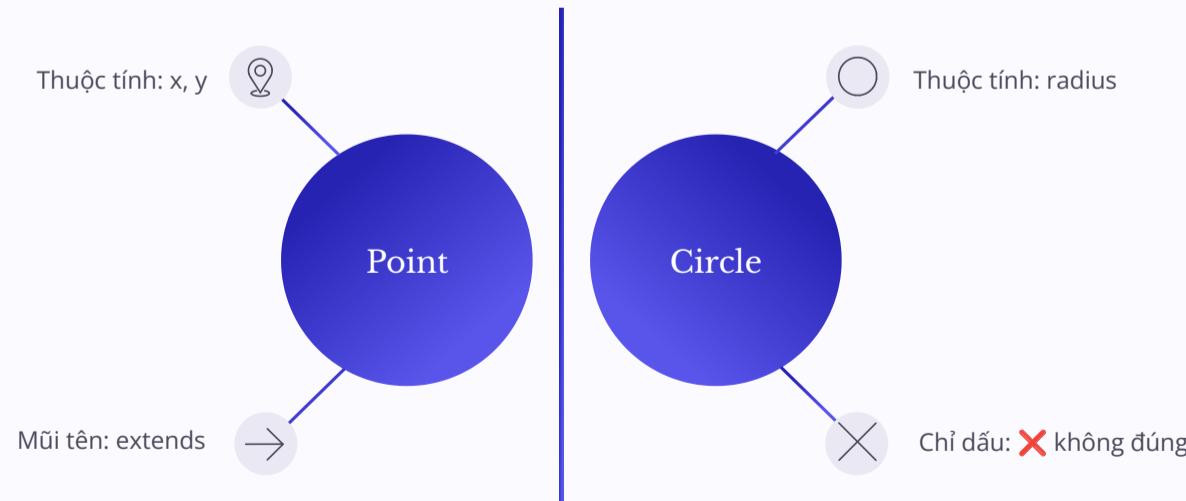
    public void draw() {
        System.out.println("Vẽ hình tròn tại (" + getX() + ", " + getY() + ") với bán kính " + radius);
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

❓ Tại sao đây là thiết kế sai?

Vấn đề chính ở đây là mối quan hệ giữa Circle và Point.

- **Circle KHÔNG PHẢI LÀ MỘT (IS-A) Point:** Theo định nghĩa, một hình tròn không phải là một điểm. Một hình tròn CÓ MỘT (HAS-A) điểm làm tâm của nó. Mối quan hệ "IS-A" là điều kiện tiên quyết cho inheritance.
- **Sự phụ thuộc không cần thiết:** Khi Circle kế thừa Point, nó kế thừa tất cả các phương thức của Point, bao gồm setX() và setY(). Điều này có nghĩa là bạn có thể gọi circle.setX(10), làm thay đổi vị trí tâm hình tròn, nhưng không rõ ràng từ bản thân lớp Circle.
- **Vi phạm nguyên tắc thiết kế:** Thiết kế này vi phạm nguyên tắc "Favor composition over inheritance" (Ưu tiên composition hơn inheritance), một nguyên tắc quan trọng trong lập trình hướng đối tượng.



⚠️ Hậu quả của thiết kế sai

- **Code khó bảo trì và mở rộng:** Bất kỳ thay đổi nào trong lớp Point (ví dụ: thay đổi cách lưu trữ tọa độ) có thể ảnh hưởng đến Circle, ngay cả khi Circle không cần những thay đổi đó.
- **Logic không rõ ràng:** Khi một Circle có các phương thức setX() và setY() trực tiếp, nó tạo ra sự mơ hồ. Các phương thức này thực sự điều khiển tâm của hình tròn, nhưng tên phương thức không phản ánh điều đó một cách trực tiếp.
- **Khó thay đổi implementation sau này:** Nếu sau này bạn muốn Circle không còn dùng tọa độ (x, y) riêng lẻ mà dùng một đối tượng CoordinateSystem phức tạp hơn cho tâm, việc này sẽ rất khó khăn do sự ràng buộc chặt chẽ của inheritance.
- **Tạo coupling chặt chẽ giữa các class:** Circle và Point trở nên phụ thuộc vào nhau một cách không cần thiết, làm giảm tính linh hoạt và khả năng tái sử dụng của code.

➡ Xem slide tiếp theo để biết cách thiết kế đúng với Composition.

Giải Pháp: Composition (Kết Hợp)

Composition là gì?

Composition là quan hệ "HAS-A" (có một): Class chứa object của class khác làm field, và object được tạo và quản lý bởi class chứa nó.

✓ Ví dụ đúng: Circle HAS-A Point

```
// Point class
public class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public double
        distanceToOrigin() {
        return Math.sqrt(
            x * x + y * y
        );
    }
}
```

```
// Circle HAS-A Point
public class Circle {
    private Point center;
    private double radius;

    public Circle(Point center,
                  double radius) {
        this.center = center;
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI *
            radius * radius;
    }

    // Expose methods của Point
    // nếu cần
    public double
        getCenterDistance() {
        return center
            .distanceToOrigin();
    }
}
```

✓ Lợi ích của Composition



Loose Coupling

Circle không phụ thuộc vào implementation chi tiết của Point. Chỉ phụ thuộc vào interface công khai.



Kiểm Soát Tốt Hơn

Circle quyết định expose methods nào của Point. Không bị ép phải expose tất cả.



Linh Hoạt Hơn

Có thể thay đổi implementation của center (dùng class khác thay Point) mà không ảnh hưởng code bên ngoài.

Khi Nào Dùng Composition?

DÙNG COMPOSITION khi

Quan hệ "HAS-A"

Khi object chứa object khác như một phần của nó (Car HAS-A Engine, Employee HAS-A Address).

Muốn tránh Fragile Base Class

Khi không muốn subclass phụ thuộc vào chi tiết implementation của superclass.

Cần linh hoạt hơn Inheritance

Khi muốn thay đổi behavior tại runtime hoặc dễ dàng swap implementation.

Ví dụ thực tế

Employee HAS-A Address

```
public class Employee {  
    private Address address;  
    private String name;  
  
    public Employee(  
        String name,  
        Address address  
    ){  
        this.name = name;  
        this.address = address;  
    }  
  
    public void relocate(  
        Address newAddress  
    ){  
        this.address =  
            newAddress;  
    }  
}
```

Car HAS-A Engine

```
public class Car {  
    private Engine engine;  
    private String model;  
  
    public Car(  
        String model,  
        Engine engine  
    ){  
        this.model = model;  
        this.engine = engine;  
    }  
  
    public void start(){  
        engine.start();  
    }  
}
```

Chiến Lược: Interface + Composition

Kết hợp Interface và Composition tạo ra thiết kế cực kỳ linh hoạt và mạnh mẽ. Hãy xem ví dụ Payment System:

```
// Interface định nghĩa hành vi  
public interface PaymentProcessor {  
    void processPayment(double amount);  
}
```

```
// Implementations  
public class CreditCardProcessor  
implements PaymentProcessor {  
  
    @Override  
    public void processPayment(  
        double amount  
    ) {  
        System.out.println(  
            "Processing credit: "  
            + amount  
        );  
    }  
}
```

```
public class PayPalProcessor  
implements PaymentProcessor {  
  
    @Override  
    public void processPayment(  
        double amount  
    ) {  
        System.out.println(  
            "Processing PayPal: "  
            + amount  
        );  
    }  
}
```

```
// Class sử dụng Composition + Interface  
public class PaymentService {  
    private PaymentProcessor processor; // Composition  
  
    // Dependency Injection - có thể thay đổi processor  
    public PaymentService(PaymentProcessor processor) {  
        this.processor = processor;  
    }  
  
    public void pay(double amount) {  
        processor.processPayment(amount);  
    }  
  
    // Có thể thay đổi processor tại runtime  
    public void setProcessor(PaymentProcessor processor) {  
        this.processor = processor;  
    }  
}
```

```
// Sử dụng - Cực kỳ linh hoạt  
PaymentProcessor creditCard = new CreditCardProcessor();  
PaymentService service = new PaymentService(creditCard);  
service.pay(100.0);  
  
// Dễ dàng thay đổi behavior  
PaymentProcessor paypal = new PayPalProcessor();  
service.setProcessor(paypal);  
service.pay(200.0);
```

So Sánh: Inheritance vs Composition

Đặc điểm	Inheritance	Composition
Quan hệ	IS-A (là một)	HAS-A (có một)
Coupling	Tight (chặt chẽ)	Loose (lỏng lẻo)
Linh hoạt	Kém (khó thay đổi)	Tốt (dễ thay đổi)
Reusability	Tốt (kết thừa code)	Tốt (tái sử dụng object)
Thay đổi runtime	✗ Không	✓ Có
Khi nào dùng	Quan hệ IS-A rõ ràng	Quan hệ HAS-A, cần linh hoạt

Nguyên tắc vàng

“

"Favor Composition over Inheritance"

Ưu tiên Composition hơn Inheritance - Đây là một trong những nguyên tắc thiết kế quan trọng nhất trong OOP. Không phải vì Inheritance xấu, mà vì Composition linh hoạt hơn trong hầu hết các trường hợp.

”

- ❑💡 **Lưu ý:** Điều này không có nghĩa "không bao giờ dùng Inheritance". Hãy dùng Inheritance khi quan hệ IS-A thực sự rõ ràng và hợp lý. Nhưng khi có nghi ngờ, hãy chọn Composition.

Tóm Tắt Chương 3



01

Inheritance và quan hệ IS-A

Hiểu rõ cơ chế kế thừa và khi nào nên áp dụng.

03

Overriding vs Overloading

Phân biệt rõ ràng hai khái niệm này và biết khi nào dùng cái nào.

05

Interface và Multiple Interfaces

Định nghĩa hành vi và implement nhiều interface cùng lúc.

07

Polymorphism và Dynamic Binding

Nắm vững đa hình và cơ chế runtime method lookup.

09

Composition over Inheritance

Ưu tiên composition để tạo thiết kế linh hoạt hơn.

02

Từ khóa extends và super

Sử dụng thành thạo cú pháp kế thừa và giao tiếp với superclass.

04

Abstract Class và Abstract Methods

Tạo template cho subclass và định nghĩa hợp đồng implementation.

06

Default Methods và Static Methods

Sử dụng các tính năng modern của Java 8+.

08

Upcasting và Downcasting

Ép kiểu an toàn và sử dụng instanceof.

10

Loại bỏ if/else bằng Polymorphism

Refactor code để dễ mở rộng và bảo trì.

Kỹ Năng Đã Được



Thiết Kế Hệ Thống

Biết cách thiết kế hệ thống phân cấp với Inheritance đúng cách, tránh các pitfall phổ biến.



Sử Dụng Abstract Class và Interface

Biết khi nào dùng abstract class, khi nào dùng interface, và khi nào kết hợp cả hai.



Áp Dụng Polymorphism

Viết code linh hoạt, dễ mở rộng bằng cách sử dụng polymorphism thay vì if/else.

Chọn Đúng Approach

Biết khi nào nên dùng Inheritance và khi nào nên dùng Composition dựa trên bản chất của vấn đề.

- ➡ **Tiếp theo:** Trong Chương 4, chúng ta sẽ học về các **SOLID Principles** - những nguyên tắc thiết kế giúp code của bạn trở nên professional và maintainable hơn. Những kiến thức trong Chương 3 là nền tảng để hiểu sâu các principles đó!



 BÀI TẬP

Bài Tập Chương 3

Các bài tập sau giúp bạn củng cố kiến thức vừa học. Hãy thử làm theo thứ tự từ dễ đến khó!

Bài 1: Inheritance Cơ Bản

🎯 Yêu cầu

01

Tạo class Vehicle (superclass)

- Fields: brand (String), year (int)
- Methods: start(), stop(), displayInfo()
- Constructor khởi tạo brand và year

02

Tạo class Car extends Vehicle

- Thêm field: numberOfDoors (int)
- Override displayInfo() để hiển thị thêm số cửa
- Constructor gọi super và khởi tạo numberOfDoors

03

Tạo class Motorcycle extends Vehicle

- Thêm field: hasSidecar (boolean)
- Override displayInfo() để hiển thị thông tin sidecar

Test code

```
Car car = new Car("Toyota", 2023, 4);
car.start();
car.displayInfo();
```

```
Motorcycle bike = new Motorcycle("Honda", 2022, false);
bike.start();
bike.displayInfo();
```

Bài 2: Overriding và Overloading

🎯 Yêu cầu

Tạo class Calculator với các methods sau:

1. **Method** add(int a, int b) - Overload với các kiểu:

- add(int a, int b) → trả về int
- add(double a, double b) → trả về double
- add(String a, String b) → nối hai chuỗi

2. **Method** multiply(int a, int b) - Sau đó tạo class ScientificCalculator extends Calculator và override method này để thêm logging hoặc xử lý đặc biệt.

Test code

```
Calculator calc = new Calculator();
System.out.println(calc.add(5, 3)); // 8
System.out.println(calc.add(5.5, 3.2)); // 8.7
System.out.println(calc.add("Hello", "World")); // "HelloWorld"
```

```
ScientificCalculator sciCalc = new ScientificCalculator();
System.out.println(sciCalc.multiply(5, 3)); // Override logic
```

Bài 3 & 4: Abstract Class và Interface

Bài 3: Abstract Class

🎯 Yêu cầu

1. Tạo abstract class Shape:
 - Abstract methods: calculateArea(), calculatePerimeter()
 - Concrete method: displayInfo()
2. Tạo subclasses: Circle, Rectangle, Triangle
3. Implement tất cả abstract methods

Test

```
Shape circle =  
    new Circle(5.0);  
circle.displayInfo();
```

```
Shape rectangle =  
    new Rectangle(4.0, 6.0);  
rectangle.displayInfo();
```

Bài 4: Interface

🎯 Yêu cầu

1. Tạo interface Drawable với method draw()
2. Tạo interface Resizable với method resize(double factor)
3. Tạo class Circle implement cả 2 interface

Test

```
Circle circle =  
    new Circle(5.0);  
circle.draw();  
circle.resize(1.5);
```

Bài 5 & 6: Polymorphism

Bài 5: Polymorphism Cơ Bản

⌚ Yêu cầu

Tạo hệ thống động vật với:

1. Abstract class `Animal` với abstract method `makeSound()`
2. Subclasses: `Dog`, `Cat`, `Bird`
3. Method `performShow(Animal animal)` sử dụng polymorphism

Test

```
Animal[] animals = {  
    new Dog("Buddy"),  
    new Cat("Fluffy"),  
    new Bird("Tweety")  
};  
  
for (Animal animal : animals) {  
    animal.makeSound(); // Polymorphism!  
}
```

Bài 6: Loại Bỏ if/else

⌚ Yêu cầu

Refactor code sau để loại bỏ if/else bằng polymorphism:

```
// Code cũ (có if/else)  
public void processPayment(String type, double amount) {  
    if (type.equals("credit")) {  
        System.out.println("Processing credit card: " + amount);  
    } else if (type.equals("paypal")) {  
        System.out.println("Processing PayPal: " + amount);  
    } else if (type.equals("bitcoin")) {  
        System.out.println("Processing Bitcoin: " + amount);  
    }  
}
```

1. Tạo interface `PaymentMethod`
2. Tạo các class: `CreditCard`, `PayPal`, `Bitcoin`
3. Refactor `processPayment()` để dùng polymorphism

Bài 7 & 8: Composition và Tổng Hợp

Bài 7: Composition

⌚ Yêu cầu

1. Tạo class Engine với method start()
2. Tạo class Car sử dụng Composition (HAS-A Engine)
3. So sánh với cách dùng Inheritance (Car extends Engine - IS-A)

Test

```
Engine engine = new Engine();
Car car = new Car(engine);
car.start(); // Gọi engine.start()
```

Bài 8: Hệ Thống Nhân Viên (Tổng Hợp)

⌚ Yêu cầu

1 Abstract class Employee

Fields: id, name, salary. Abstract method: calculateBonus(). Concrete method: displayInfo().

2 Subclasses

Manager (Bonus = 20%), Developer (Bonus = 10%), Designer (Bonus = 15%).

3 Interface Workable

Method: work(). Tất cả Employee implement interface này.

4 Class Department

Field: List<Employee> employees. Method: calculateTotalBonus() dùng polymorphism.

Yêu cầu bổ sung: Tuân thủ Clean Code, viết JavaDoc đầy đủ, sử dụng Composition khi phù hợp.



TÀI LIỆU

Tài Liệu Tham Khảo



Java Documentation

Tài liệu chính thức từ Oracle về Inheritance và Interface. Đây là nguồn tài liệu đáng tin cậy nhất để tìm hiểu chi tiết về các tính năng của Java.

[https://docs.oracle.com/
javase/tutorial/java/iandt/](https://docs.oracle.com/javase/tutorial/java/iandt/)

Effective Java - Joshua Bloch

Một trong những cuốn sách Java kinh điển nhất. Đặc biệt chú ý **Item 18: Favor composition over inheritance** để hiểu sâu hơn về khi nào nên dùng composition.

Head First Design Patterns

Cuốn sách tuyệt vời về các design patterns. Chú ý đặc biệt đến **Strategy Pattern** - một ví dụ điển hình của việc sử dụng composition và polymorphism.

Chúc Mừng! 🎉

Bạn đã hoàn thành Chương 3!

Bạn vừa chinh phục được chương học quan trọng nhất trong hành trình OOP của mình. Inheritance và Polymorphism là nền tảng để xây dựng các hệ thống phần mềm chuyên nghiệp.

10

Khái Niệm Quan Trọng

Đã nắm vững từ Inheritance đến Polymorphism

8

Bài Tập Thực Hành

Để củng cố kiến thức vừa học

100%

Sẵn Sàng

Cho Chương 4: SOLID Principles

🚀 Bước tiếp theo

Hãy dành thời gian thực hành các bài tập để kiến thức được củng cố. Khi bạn tự tin với Inheritance và Polymorphism, hãy tiếp tục sang Chương 4 để học về các nguyên tắc thiết kế SOLID - công cụ giúp bạn viết code professional và maintainable!

- 💡 **Lời khuyên cuối:** Đừng vội vàng. OOP là một hành trình, không phải đích đến. Hãy thực hành nhiều, code nhiều, và đặt câu hỏi nhiều. Mỗi dòng code bạn viết là một bước tiến trên con đường trở thành lập trình viên giỏi!

Chúc bạn học tốt! 🚀