

HANDOUT – TÍNH ĐA HÌNH (POLYMORPHISM)

Buổi: 12 / 15

Mục tiêu bài học:

- Hiểu khái niệm và vai trò của tính đa hình trong lập trình hướng đối tượng.
- Phân biệt được đa hình lúc biên dịch (compile time) và đa hình lúc chạy (runtime).
- Vận dụng tính đa hình để giải quyết các bài toán lập trình bằng Java.

LÝ THUYẾT

1. Giới thiệu về Tính Đa Hình (Polymorphism)

Khái niệm

- **Polymorphism** có nguồn gốc từ tiếng Hy Lạp, nghĩa là "**nhiều hình thái**".
- Trong lập trình hướng đối tượng (**OOP**), đa hình cho phép **cùng một lời gọi phương thức** có thể **thực hiện các hành vi khác nhau** tùy thuộc vào **đối tượng cụ thể** thực hiện nó.

Ví dụ đơn giản

```
Animal animal1 = new Dog();  
  
Animal animal2 = new Cat();  
  
animal1.sound(); // Output: "Woof!"  
animal2.sound(); // Output: "Meow!"
```

Cùng là lời gọi sound() nhưng kết quả khác nhau vì đối tượng thực thi khác nhau (Dog, Cat).

Cơ chế hoạt động

- Đa hình có thể được **thực hiện theo hai cách**:
 - **Compile-time polymorphism (tĩnh)**: qua **method overloading**.

- **Runtime polymorphism (động):** qua **method overriding**, thường sử dụng với **kế thừa** và **tính trừu tượng**.

Vai trò và lợi ích của tính đa hình

| Lợi ích | Mô tả |
|--|---|
| Tăng tính mở rộng | Có thể thêm đối tượng mới (lớp con) mà không cần sửa lại logic ở lớp cha. |
| Dễ bảo trì | Khi cần thay đổi hành vi của một loại đối tượng cụ thể, chỉ cần thay đổi trong lớp đó. |
| Tái sử dụng mã | Viết một đoạn mã tổng quát cho lớp cha, lớp con có thể sử dụng lại và tùy biến hành vi. |
| Tăng tính trừu tượng và linh hoạt | Giúp chương trình tập trung vào "cái gì làm" thay vì "làm như thế nào". |

Tính đa hình là một **trụ cột quan trọng** trong lập trình hướng đối tượng, giúp viết mã ngắn gọn, **dễ mở rộng**, **dễ bảo trì**, đồng thời tăng khả năng **tái sử dụng** và **linh hoạt** hóa ứng dụng.

2. Đa hình lúc Compile (Compile-time Polymorphism)

Khái niệm

- **Compile-time polymorphism** còn gọi là **đa hình tĩnh**, vì việc lựa chọn phương thức phù hợp sẽ được xác định **ngay tại thời điểm biên dịch**.
- Hình thức phổ biến nhất của đa hình tĩnh là **nạp chồng phương thức (method overloading)**.

Method Overloading là gì?

- **Overloading** là việc khai báo **nhiều phương thức cùng tên** trong **cùng một lớp**, nhưng **khác nhau về danh sách tham số**:
 - Số lượng tham số.
 - Kiểu dữ liệu tham số.
 - Thứ tự tham số.

 **Lưu ý:** Khác biệt về kiểu trả về **không đủ điều kiện** để tạo overloading.

Ví dụ minh họa

```
public class Calculator {  
  
    // Phương thức cộng hai số nguyên  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    // Phương thức cộng hai số thực  
    public double add(double a, double b) {  
        return a + b;  
    }  
  
    // Phương thức cộng ba số nguyên  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    public static void main(String[] args) {  
        Calculator calc = new Calculator();  
  
        System.out.println(calc.add(2, 3));        // Output: 5  
        System.out.println(calc.add(2.5, 3.1));    // Output: 5.6  
        System.out.println(calc.add(1, 2, 3));     // Output: 6  
    }  
}
```

Lợi ích của method overloading

| Lợi ích | Mô tả |
|--------------------------|--|
| Tăng tính linh hoạt | Cho phép sử dụng cùng tên phương thức cho nhiều mục đích khác nhau. |
| Cải thiện tính dễ đọc | Dễ hiểu hơn thay vì tạo nhiều tên phương thức khác nhau. |
| Gắn logic theo ngữ nghĩa | Các phương thức cùng thực hiện một ý tưởng chung (ví dụ: “add”) nhưng với dữ liệu khác nhau. |

- **Method overloading** là biểu hiện cụ thể của **đa hình tĩnh** trong Java.
- Nó giúp chương trình trở nên linh hoạt và thân thiện hơn với người đọc, đồng thời hỗ trợ tốt cho việc xử lý các kiểu dữ liệu khác nhau mà không phải thay đổi tên hàm.

3. Đa hình lúc Runtime (Runtime Polymorphism)

Khái niệm

- **Runtime polymorphism** còn gọi là **đa hình động**, vì việc lựa chọn phương thức để thực thi được quyết định **tại thời điểm chạy chương trình**.
- Được thực hiện thông qua **ghi đè phương thức (method overriding)** và sử dụng **các đối tượng kế thừa từ một lớp cha chung**.
- Cơ chế này còn được gọi là **dynamic dispatch** – chọn phương thức phù hợp tại thời điểm chương trình thực thi.

Method Overriding là gì?

- Khi một **lớp con định nghĩa lại** một phương thức đã được khai báo trong **lớp cha** với:
 - Cùng tên phương thức.
 - Cùng kiểu trả về.
 - Cùng danh sách tham số.

- ◆ Ghi đè thể hiện mối quan hệ **"is-a"** giữa lớp con và lớp cha.
- ◆ Trong Java, ta thường dùng annotation `@Override` để đảm bảo tính đúng đắn khi ghi đè.

Ví dụ minh họa

```
class Animal {  
    public void sound() {  
        System.out.println("Some sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    public void sound() {  
        System.out.println("Woof!");  
    }  
}  
  
class Cat extends Animal {  
    @Override  
    public void sound() {  
        System.out.println("Meow!");  
    }  
}  
  
public class Demo {  
    public static void main(String[] args) {  
        Animal a1 = new Dog();  
    }  
}
```

```
Animal a2 = new Cat();

a1.sound(); // Output: Woof!

a2.sound(); // Output: Meow!

}

}
```

📌 Mặc dù a1 và a2 đều được khai báo là Animal, nhưng khi gọi sound(), Java sử dụng phương thức tương ứng với đối tượng thực (Dog hoặc Cat) tại **runtime**.

🔧 Cơ chế Dynamic Dispatch

- Khi gọi phương thức từ một tham chiếu đến lớp cha, JVM sẽ xác định **đối tượng thực sự** (instance) của tham chiếu đó và gọi **phiên bản phương thức phù hợp** trong lớp con (nếu có override).
- Đây là chìa khóa giúp hiện thực hóa tính đa hình ở cấp độ đối tượng.

🎯 Lợi ích của Runtime Polymorphism

| Lợi ích | Mô tả |
|--|---|
| Hỗ trợ mở rộng | Cho phép thêm nhiều lớp con mà không cần thay đổi code xử lý logic chính. |
| Tăng tính linh hoạt | Xử lý đối tượng thông qua interface hoặc lớp cha chung. |
| Áp dụng trong thiết kế hướng interface | Viết chương trình theo hướng giao diện và trừu tượng , không bị ràng buộc bởi kiểu cụ thể. |

💡 Tổng kết

- Runtime polymorphism giúp chương trình **thay đổi hành vi linh hoạt tùy theo loại đối tượng thực sự**, dù đang thao tác qua tham chiếu của lớp cha.
- Đây là một trong những **cơ chế cốt lõi** của OOP, đặc biệt khi kết hợp với **interface**, **abstract class**, và **collection framework** trong Java.

Phân biệt giữa (method overloading). và (method overriding)

Dưới đây là bảng so sánh giúp bạn **phân biệt rõ giữa Method Overloading và Method Overriding** trong Java:

| Tiêu chí | Method Overloading | Method Overriding |
|--------------------------|--|--|
| Khái niệm | Định nghĩa nhiều phương thức cùng tên nhưng khác tham số trong cùng một lớp | Định nghĩa lại phương thức của lớp cha trong lớp con |
| Thời điểm xảy ra | Compile-time (Đa hình lúc biên dịch – static) | Runtime (Đa hình lúc chạy – dynamic) |
| Tham số | Phải khác nhau về số lượng, kiểu dữ liệu hoặc thứ tự | Phải giống hệt (tên, kiểu, thứ tự tham số) |
| Kế thừa | Không cần kế thừa, xảy ra trong cùng một lớp | Cần kế thừa , xảy ra giữa lớp cha và lớp con |
| Annotation hỗ trợ | Không sử dụng @Override | Nên dùng @Override để thông báo đang override |
| Mục đích sử dụng | Tăng tính linh hoạt trong cách gọi cùng tên phương thức | Tùy biến lại hành vi của phương thức lớp cha |
| Ví dụ | java void print(int a); void print(String s); | java class A { void show() {} } class B extends A { @Override void show() {} } |

Ví dụ minh họa

Method Overloading (Nạp chồng phương thức):

```
public class Print {  
  
    void show(int a) {  
        System.out.println("int: " + a);  
    }  
  
    void show(String s) {  
        System.out.println("String: " + s);  
    }  
}
```

```
}  
  
}
```

Method Overriding (Ghi đè phương thức):



```
class Animal {  
    void sound() {  
        System.out.println("Animal sound");  
    }  
}  
  
class Dog extends Animal {  
    @Override  
    void sound() {  
        System.out.println("Woof!");  
    }  
}
```

4. Tổng kết: Ý nghĩa của Tính Đa Hình

Tính đa hình là gì, nhắc lại?

- **Polymorphism (đa hình)** cho phép các đối tượng có **giao diện giống nhau** nhưng **hành vi khác nhau**.
- Là một trong **bốn trụ cột chính** của lập trình hướng đối tượng (OOP), bên cạnh **tính đóng gói, kế thừa, và trừu tượng**.

Ý nghĩa và lợi ích của tính đa hình

|  Lợi ích |  Giải thích |
|---|--|
| 1. Dễ mở rộng | Khi cần thêm lớp mới (ví dụ: loài động vật mới), chỉ cần tạo lớp con mới và override phương thức – không cần sửa mã gốc. |
| 2. Tái sử dụng mã tốt hơn | Logic xử lý chung có thể viết ở lớp cha hoặc thông qua interface, các lớp con chỉ cần kế thừa và tùy biến hành vi. |
| 3. Tăng tính trừu tượng | Người lập trình chỉ cần quan tâm đến “ giao diện hành vi ” mà không cần biết chi tiết lớp con cụ thể. |
| 4. Tăng tính linh hoạt | Các thuật toán, cấu trúc chương trình hoạt động tốt với các đối tượng đa dạng – dễ áp dụng cho các mô hình mở rộng. |
| 5. Hỗ trợ thiết kế theo interface | Cho phép lập trình theo hợp đồng (contract) , dễ test, dễ bảo trì, dễ thay đổi. |

Ví dụ minh họa lại một lần nữa

```
// Interface mô tả một hành vi
interface Shape {
    void draw();
}

// Lớp con triển khai hành vi khác nhau
class Circle implements Shape {
    public void draw() {
        System.out.println("Vẽ hình tròn");
    }
}


class Rectangle implements Shape {
    public void draw() {
```

```
        System.out.println("Vẽ hình chữ nhật");
    }
}

// Hàm sử dụng tính đa hình
public class DrawingApp {
    public static void render(Shape s) {
        s.draw();
    }

    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape rectangle = new Rectangle();

        render(circle);    // Vẽ hình tròn
        render(rectangle); // Vẽ hình chữ nhật
    }
}
```

 Nhờ tính đa hình, render() có thể làm việc với **bất kỳ đối tượng nào** thuộc kiểu Shape – mà không cần biết đó là hình tròn hay chữ nhật.

Tổng kết ngắn gọn

Tính đa hình:

- Là **linh hồn của OOP**.
- Giúp chương trình **dễ mở rộng, dễ bảo trì, dễ hiểu và tái sử dụng**.
- Là nền tảng cho các kỹ thuật thiết kế phần mềm như **Dependency Injection, Interface-based Programming, Design Patterns** (Factory, Strategy, v.v.).

BÀI TẬP CÓ HƯỚNG DẪN

Ví dụ 1: Hệ thống quản lý nhân viên – Tính lương theo loại nhân viên

Mô tả bài toán

Một công ty có nhiều loại nhân viên với cách tính lương khác nhau, bao gồm:

- **Nhân viên toàn thời gian (FullTimeEmployee):** lương = lương cơ bản + phụ cấp.
- **Nhân viên bán thời gian (PartTimeEmployee):** lương = số giờ làm \times đơn giá giờ.

Yêu cầu:

1. Thiết kế hệ thống các lớp biểu diễn các loại nhân viên.
2. Viết một hàm có thể tính và in lương của **danh sách nhân viên** bất kỳ.
3. Áp dụng **tính đa hình** để xử lý danh sách nhân viên mà **không cần kiểm tra loại nhân viên cụ thể**.

Phân tích bài toán và thiết kế lớp

Lớp Employee (abstract class):

- Là lớp cha của tất cả các loại nhân viên.
- Có tên và phương thức **abstract** để tính lương.

Lớp FullTimeEmployee kế thừa Employee:

- Có thêm: basicSalary, allowance.
- Ghi đè phương thức calculateSalary().

Lớp PartTimeEmployee kế thừa Employee:

- Có thêm: hoursWorked, hourlyRate.
- Ghi đè phương thức calculateSalary().

Sơ đồ quan hệ giữa các lớp

[Employee] (abstract)

/ \

/ \

[FullTimeEmployee] [PartTimeEmployee]

Code chương trình Java

// Lớp cha trừu tượng

```
abstract class Employee {
```

```
    protected String name;
```

```
    public Employee(String name) {
```

```
        this.name = name;
```

```
    }
```

```
    public abstract double calculateSalary();
```

```
    public void printSalary() {
```

```
        System.out.println(name + " có lương: " + calculateSalary() + " VND");
```

```
    }
```

```
}
```

// Lớp nhân viên toàn thời gian

```
class FullTimeEmployee extends Employee {
```

```
    private double basicSalary;
```

```
    private double allowance;
```

```
public FullTimeEmployee(String name, double basicSalary, double allowance) {
    super(name);
    this.basicSalary = basicSalary;
    this.allowance = allowance;
}

@Override
public double calculateSalary() {
    return basicSalary + allowance;
}
}

// Lớp nhân viên bán thời gian
class PartTimeEmployee extends Employee {
    private int hoursWorked;
    private double hourlyRate;

    public PartTimeEmployee(String name, int hoursWorked, double hourlyRate) {
        super(name);
        this.hoursWorked = hoursWorked;
        this.hourlyRate = hourlyRate;
    }

    @Override
    public double calculateSalary() {
        return hoursWorked * hourlyRate;
    }
}
```

```

}

// Chương trình chính
public class PayrollSystem {
    public static void main(String[] args) {
        Employee[] employees = new Employee[] {
            new FullTimeEmployee("Nguyễn Văn A", 5000000, 2000000),
            new PartTimeEmployee("Trần Thị B", 80, 50000),
            new FullTimeEmployee("Phạm Văn C", 6000000, 2500000)
        };

        System.out.println("== Bảng lương nhân viên ==");
        for (Employee e : employees) {
            e.printSalary(); // Gọi phương thức đa hình
        }
    }
}

```

Giải thích chương trình

- **Đa hình động (runtime polymorphism)** được thể hiện qua việc:
 - Danh sách Employee[] có thể chứa các đối tượng FullTimeEmployee và PartTimeEmployee.
 - Khi gọi printSalary(), Java tự động chọn phương thức calculateSalary() phù hợp với **đối tượng thực**.
- Giúp hệ thống dễ mở rộng: thêm loại nhân viên khác chỉ cần tạo lớp con mới và override phương thức calculateSalary().

Tổng kết

- Đây là một ví dụ thực tế và dễ hiểu về đa hình trong Java.
- Kỹ thuật này giúp chương trình **mở rộng dễ dàng, giảm phụ thuộc vào if-else, và đóng gói tốt logic tính lương**.

Ví dụ 2: Hệ thống vẽ hình học – Sử dụng giao diện và đa hình

Mô tả bài toán

Xây dựng một hệ thống cho phép người dùng làm việc với các hình học cơ bản như:

- **Hình tròn (Circle)**
- **Hình chữ nhật (Rectangle)**
- **Hình tam giác (Triangle)**

Mỗi hình có thể:

1. **Vẽ ra màn hình** (draw()).
2. **Tính diện tích** (calculateArea()).

Yêu cầu:

- Tạo một **giao diện (interface)** để đại diện cho các hành vi chung của hình học.
- Viết chương trình có thể xử lý **một danh sách các hình** và gọi phương thức draw() và calculateArea() **mà không cần biết đối tượng là loại hình gì**.
- Áp dụng **tính đa hình runtime thông qua interface**.

Phân tích và thiết kế lớp

Interface Shape

- Định nghĩa hai hành vi:
 - void draw();
 - double calculateArea();

Các lớp Circle, Rectangle, Triangle đều triển khai (implements) giao diện Shape.

Sơ đồ quan hệ giữa các lớp

[Shape] (interface)

/ | \

/ | \

[Circle] [Rectangle] [Triangle]

Code chương trình Java

// Giao diện chung

```
interface Shape {
```

```
    void draw();
```

```
    double calculateArea();
```

```
}
```

// Lớp hình tròn

```
class Circle implements Shape {
```

```
    private double radius;
```

```
    public Circle(double radius) {
```

```
        this.radius = radius;
```

```
    }
```

```
    public void draw() {
```

```
        System.out.println("Vẽ hình tròn bán kính " + radius);
```

```
    }
```

```
    public double calculateArea() {
```

```
        return Math.PI * radius * radius;
```



```
}  
}  
  
// Lớp hình chữ nhật  
class Rectangle implements Shape {  
    private double width, height;  
  
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    public void draw() {  
        System.out.println("Vẽ hình chữ nhật " + width + "x" + height);  
    }  
  
    public double calculateArea() {  
        return width * height;  
    }  
}  
  
// Lớp hình tam giác  
class Triangle implements Shape {  
    private double base, height;  
  
    public Triangle(double base, double height) {  
        this.base = base;
```

```
        this.height = height;
    }

    public void draw() {
        System.out.println("Vẽ hình tam giác đáy " + base + ", cao " + height);
    }

    public double calculateArea() {
        return 0.5 * base * height;
    }
}

// Chương trình chính
public class DrawingApp {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new Circle(3.0),
            new Rectangle(4.0, 5.0),
            new Triangle(6.0, 4.0)
        };

        for (Shape s : shapes) {
            s.draw();
            System.out.println("Diện tích: " + s.calculateArea() + "\n");
        }
    }
}
```

Phân tích chương trình

- **Giao diện Shape** giúp mô tả hành vi chung của mọi hình học.
- **Các lớp cụ thể** (Circle, Rectangle, Triangle) đều **triển khai giao diện**, và cung cấp cách xử lý riêng cho draw() và calculateArea().
- Trong main(), biến Shape[] shapes chính là nơi thể hiện rõ **tính đa hình runtime**: ta thao tác với đối tượng qua interface, nhưng hành vi thực sự là của lớp con cụ thể.

So sánh với Ví dụ 1

| Tiêu chí | Ví dụ 1: Nhân viên | Ví dụ 2: Hình học |
|------------------|--|--|
| Kiểu đa hình | Ghi đè phương thức (Overriding) qua kế thừa | Ghi đè phương thức thông qua giao diện (interface) |
| Mục đích | Tính lương | Vẽ hình & tính diện tích |
| Đặc điểm | Quan hệ cha-con | Không có quan hệ kế thừa trực tiếp – chỉ thông qua interface |
| Hành vi gọi động | calculateSalary() từ lớp Employee | draw(), calculateArea() từ Shape |

Tổng kết

- Ví dụ 2 giúp sinh viên nhận ra rằng: **đa hình không chỉ đến từ kế thừa**, mà còn có thể đến từ **interface**.
- Điều này rất phổ biến trong các hệ thống lớn, nơi ta không luôn có quan hệ cha-con, nhưng **vẫn có thể thiết kế theo hành vi (behavior)** chung.

Ví dụ 3: Máy tính đơn giản – Nạp chồng phương thức (Method Overloading)

Mô tả bài toán

Xây dựng một lớp Calculator thực hiện các phép tính cộng (add) với các loại dữ liệu và số lượng tham số khác nhau:

- Cộng hai số nguyên.
- Cộng hai số thực.

- Cộng ba số nguyên.

Yêu cầu:

- Thiết kế một lớp duy nhất chứa các phương thức add() với nhiều chữ ký khác nhau.
- Khi gọi phương thức, Java **chọn phiên bản phù hợp tại thời điểm biên dịch** dựa vào **kiểu dữ liệu và số lượng tham số**.

Phân tích và thiết kế

- Sử dụng **method overloading**: cùng tên phương thức, nhưng **khác tham số (số lượng, kiểu dữ liệu)**.
- Không cần dùng kế thừa hay interface.
- Không có ghi đè (override), nên đa hình này là **đa hình tĩnh**.

Code chương trình Java

```
public class Calculator {  
  
    // Cộng hai số nguyên  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    // Cộng hai số thực  
    public double add(double a, double b) {  
        return a + b;  
    }  
  
    // Cộng ba số nguyên  
    public int add(int a, int b, int c) {
```

```

        return a + b + c;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();





        // Java tự chọn phương thức phù hợp dựa vào kiểu dữ liệu và số lượng tham số
        System.out.println("Cộng 2 số nguyên: " + calc.add(5, 10)); // Gọi add(int, int)
        System.out.println("Cộng 2 số thực: " + calc.add(3.5, 2.5)); // Gọi add(double, double)
        System.out.println("Cộng 3 số nguyên: " + calc.add(1, 2, 3)); // Gọi add(int, int, int)
    }
}

```

Phân tích chương trình

- **Đa hình compile-time (tĩnh):** tất cả quyết định xảy ra **trước khi chạy chương trình**.
- Trình biên dịch Java xác định **phiên bản chính xác của phương thức add()** dựa vào:
 - **Kiểu dữ liệu tham số:** int hay double.
 - **Số lượng tham số:** 2 hay 3.

So sánh với Overriding (đa hình runtime)

| Tiêu chí | Method Overloading | Method Overriding |
|------------------------------|---|---|
| Thời điểm quyết định hành vi | Khi biên dịch (compile-time) | Khi chạy chương trình (runtime) |
| Cần kế thừa? |  Không |  Có (lớp cha – lớp con) |
| Cùng tên phương thức? |  Có |  Có |

| Tiêu chí | Method Overloading | Method Overriding |
|--------------------|---------------------------------------|------------------------------|
| Tham số khác nhau? | ✅ Phải khác | ❌ Phải giống |
| Mục đích | Tiện lợi: cùng tên – nhiều dạng xử lý | Tuỳ biến hành vi của lớp con |
| Từ khóa @Override | ❌ Không cần | ✅ Nên dùng |

💬 Tổng kết

- Đây là **đa hình tĩnh** – tiện lợi để viết nhiều phương án xử lý trong cùng một lớp.
- Sinh viên dễ nhầm với overriding, nên cần **phân biệt rõ dựa trên**:
 - Thời điểm thực hiện
 - Cấu trúc kế thừa
 - Sự khác biệt về tham số

TRẮC NGHIỆM

- Đa hình trong lập trình hướng đối tượng là gì?**
 - Kỹ thuật bảo mật chương trình
 - Khả năng đối tượng nhận nhiều hình dạng khác nhau
 - Cách tạo lớp trừu tượng
 - Quá trình đóng gói dữ liệu
- Phương thức nào sau đây thể hiện tính đa hình tĩnh?**
 - Ghi đè phương thức toString()
 - Tạo nhiều phương thức print() với tham số khác nhau
 - Khai báo một interface
 - Sử dụng lớp abstract
- Tính đa hình động xảy ra khi nào?**
 - Lúc biên dịch
 - Lúc gõ mã nguồn
 - Lúc chương trình chạy
 - Lúc tạo class

4. Từ khóa nào dùng để chỉ ghi đè phương thức trong Java?

- A. super
- B. overload
- C. @Override
- D. virtual

5. Phát biểu nào đúng về method overloading?

- A. Cần có tính kế thừa
- B. Các phương thức có cùng tên và tham số
- C. Xảy ra ở runtime
- D. Phân biệt nhau bởi kiểu hoặc số lượng tham số

6. Điểm khác biệt chính giữa overloading và overriding là gì?

- A. Overloading cần kế thừa, còn overriding không
- B. Overloading xảy ra ở runtime
- C. Overloading phân biệt dựa trên tham số
- D. Overriding xảy ra ở compile-time

7. Câu lệnh sau sẽ in ra gì?

```
Animal a = new Dog();  
a.sound();
```

Biết Dog ghi đè phương thức sound() của Animal.

- A. Some sound
- B. Dog sound
- C. Lỗi biên dịch
- D. Nội dung phương thức Dog.sound()

8. Method overriding giúp đạt được điều gì?

- A. Ẩn thông tin dữ liệu
- B. Đóng gói
- C. Tùy biến hành vi đối tượng ở lớp con
- D. Sử dụng nhiều phương thức giống nhau

9. Điều kiện để một phương thức được gọi là overridden là:

- A. Khác tên
- B. Khác kiểu trả về

- C. Cùng tên và tham số với phương thức lớp cha
- D. Chỉ cùng tên

10. Trong Java, khi nào nên dùng @Override?

- A. Khi nạp chồng phương thức
- B. Khi định nghĩa lớp mới
- C. Khi ghi đè phương thức từ lớp cha
- D. Khi gọi constructor

11. Cho đoạn code sau, kết quả in ra là gì?

```
class A {  
    void show() { System.out.println("A"); }  
}  
class B extends A {  
    void show() { System.out.println("B"); }  
}  
public class Test {  
    public static void main(String[] args) {  
        A obj = new B();  
        obj.show();  
    }  
}
```

- A. A
- B. B
- C. Lỗi biên dịch
- D. Không in gì

12. Vì sao ta cần ghi đè phương thức toString() trong Java?

- A. Để thay đổi cách chuyển đổi đối tượng thành chuỗi
- B. Để hỗ trợ nạp chồng
- C. Để dùng với toán tử +

D. Vì phương thức đó không tồn tại sẵn

13. Phương thức nào sau đây là ví dụ của overloading?

```
void display(int a) {}  
  
void display(double a) {}  
  
void display(int a, int b) {}
```

- A. Chỉ dòng 1 và 2
- B. Tất cả đều là overloading
- C. Không có overloading
- D. Chỉ dòng 1 và 3

14. Lớp Shape có phương thức draw(). Nếu Circle ghi đè draw(), đoạn nào gọi đúng draw() của Circle?

- A. Shape s = new Shape(); s.draw();
- B. Circle c = new Circle(); c.draw();
- C. Shape s = new Circle(); s.draw();
- D. Cả B và C

15. Trong đa hình động, phương thức được chọn tại thời điểm:

- A. Compile
- B. Khi biên dịch
- C. Runtime
- D. Trước khi tạo đối tượng

16. Giả sử Printer là interface với phương thức print(). Lớp InkjetPrinter và LaserPrinter cài đặt print(). Ta khai báo:

```
Printer p = new LaserPrinter();  
  
p.print();
```

Đây là ví dụ của:

- A. Nạp chồng
- B. Đa hình tĩnh
- C. Đa hình động
- D. Giao diện trừu tượng

17. Lợi ích chính của tính đa hình là gì?

- A. Tăng tính bảo mật
- B. Dễ mở rộng, tái sử dụng mã nguồn
- C. Giảm số lượng lớp
- D. Không cần kế thừa

18. Đa hình tĩnh không cho phép:

- A. Phân biệt phương thức theo tham số
- B. Thay đổi hành vi tại runtime
- C. Gọi phương thức bằng đối tượng lớp con
- D. Khai báo nhiều phương thức trùng tên

19. Điều gì xảy ra nếu không ghi chú @Override trong phương thức override?

- A. Lỗi runtime
- B. Lỗi compile
- C. Không ảnh hưởng, nhưng có thể sai logic nếu tên sai
- D. Không chạy được chương trình

20. Giả sử một lớp sử dụng cả overloading và overriding, nhận định nào sau đây đúng?

- A. Hai cơ chế này không thể cùng tồn tại
- B. Overloading chỉ hoạt động khi ghi đè xong
- C. JVM phân biệt overloading tại runtime
- D. Overloading xảy ra ở compile-time, overriding ở runtime

BÀI TẬP THỰC HÀNH

Bài 1: Nạp chồng phương thức hiển thị thông tin

Tình huống: Viết lớp Student có nhiều phương thức displayInfo() để hiển thị thông tin sinh viên, tùy theo số lượng tham số truyền vào.

Hướng dẫn:

- Sử dụng **method overloading** để viết nhiều phiên bản displayInfo() với tham số khác nhau (tên, tên và điểm, tên và tuổi...).
- Mục tiêu: hiểu được **đa hình tĩnh** và cách Java chọn phương thức phù hợp theo tham số.

Bài 2: Vẽ hình cơ bản

Tình huống: Tạo các lớp Circle, Rectangle, Triangle kế thừa từ lớp Shape, và mỗi lớp ghi đè phương thức draw().

Hướng dẫn:

- Khai báo Shape là lớp cha có phương thức draw() (có thể là abstract).
- Mỗi lớp con sẽ cài đặt draw() riêng biệt.
- Trong main(), tạo mảng Shape[] chứa các đối tượng khác nhau và gọi draw() để thấy tính **đa hình động**.

Bài 3: Máy in đa năng

Tình huống: Viết chương trình mô phỏng các loại máy in: InkjetPrinter, LaserPrinter, DotMatrixPrinter, đều kế thừa từ lớp Printer.

Hướng dẫn:

- Mỗi lớp ghi đè phương thức print().
- Trong hàm chính, viết một phương thức startPrinting(Printer p) và truyền vào các loại máy in khác nhau.
- Mục tiêu: sử dụng đa hình để xử lý hành vi tương ứng mà không cần phân biệt cụ thể kiểu đối tượng.

Bài 4: Giao diện thanh toán

Tình huống: Thiết kế một giao diện Payment với phương thức pay(). Cài đặt các lớp CreditCard, Cash, EWallet.

Hướng dẫn:

- Dùng interface Payment, và cài đặt pay() khác nhau ở mỗi lớp.
- Viết một hàm xử lý giao dịch mà chỉ dùng biến kiểu Payment, truyền vào các đối tượng khác nhau.
- Nhấn mạnh khả năng mở rộng hệ thống mà không cần sửa mã gốc.

Bài 5: Quản lý động vật

Tình huống: Tạo các lớp Cat, Dog, Bird kế thừa lớp Animal, và ghi đè phương thức makeSound().

Hướng dẫn:

- Lớp Animal có makeSound() là phương thức trừu tượng.
- Mỗi lớp con định nghĩa âm thanh riêng.
- Tạo danh sách các Animal và duyệt qua để gọi makeSound() → mục tiêu là thấy rõ cơ chế gọi đúng phương thức nhờ đa hình.

Bài 6: Giao diện nhân viên

Tình huống: Xây dựng hệ thống lương với giao diện Employee và các lớp FullTimeEmployee, PartTimeEmployee, ContractEmployee.

Hướng dẫn:

- Interface hoặc lớp trừu tượng có phương thức calculateSalary().
- Mỗi loại nhân viên có cách tính lương riêng.
- Cho người dùng nhập loại nhân viên và lương đầu vào, chương trình tính tổng lương qua đa hình.

Bài 7: Quản lý phương tiện

Tình huống: Mô phỏng một hệ thống giao thông gồm Car, Bike, Bus, tất cả kế thừa từ Vehicle và có hành vi move().

Hướng dẫn:

- Dùng kế thừa và ghi đè move() để mô tả cách di chuyển khác nhau.
- Viết một hàm simulateTraffic(Vehicle[] vehicles) để duyệt và gọi move().
- Người học hiểu rõ cách truyền mảng đối tượng kiểu cha và gọi phương thức đúng nhờ đa hình.

Bài 8: Bộ lọc tìm kiếm

Tình huống: Xây dựng các bộ lọc khác nhau cho sản phẩm: theo tên, theo giá, theo danh mục.

Hướng dẫn:

- Tạo interface Filter với phương thức apply(Product p).

- Tạo các lớp như NameFilter, PriceFilter, CategoryFilter cài đặt interface.
- Áp dụng đa hình để viết hàm lọc linh hoạt, mở rộng dễ dàng.
- Bài này hướng đến việc dùng **đa hình như một chiến lược xử lý linh hoạt** (Strategy Pattern).

Bài 9: Hệ thống thông báo

Tình huống: Tạo hệ thống gửi thông báo có thể gửi qua Email, SMS, PushNotification.

Hướng dẫn:

- Tạo interface Notifier với phương thức send(String message).
- Viết lớp NotificationService nhận Notifier như tham số để gửi thông báo.
- Cho phép thay đổi cách gửi mà không sửa code chính.
- Bài này dùng đa hình trong **mục tiêu ràng buộc lỏng (loose coupling)** – cực kỳ hữu ích cho thiết kế hướng mô-đun.

Bài 10: Trình quản lý plugin

Tình huống: Mô phỏng hệ thống plugin cho một ứng dụng. Mỗi plugin là một lớp kế thừa từ interface Plugin và thực hiện phương thức execute().

Hướng dẫn:

- Viết interface Plugin với execute().
- Cho phép đăng ký nhiều plugin khác nhau vào danh sách.
- Viết chương trình chính duyệt qua danh sách và gọi execute() để thực hiện hành vi tương ứng.
- Bài này luyện tư duy **đa hình + quản lý runtime theo mô-đun** như các framework thực tế (VD: Eclipse, IntelliJ).

BÀI TẬP DỰ ÁN

Dự án 1: Ứng dụng Quản lý Đơn hàng Online

Tình huống:

Bạn và đồng đội sẽ xây dựng một hệ thống quản lý đơn hàng cho một cửa hàng trực tuyến. Mỗi đơn hàng có thể được thanh toán bằng nhiều hình thức (COD, chuyển khoản, ví điện tử...) và vận chuyển qua nhiều dịch vụ (VNPost, Giao Hàng Nhanh, AhaMove...).

Phân công:

- **Thành viên A:** Thiết kế và cài đặt các lớp thanh toán (payment methods) và xử lý thanh toán bằng đa hình.
- **Thành viên B:** Thiết kế và cài đặt các lớp vận chuyển (shipping methods) và mô phỏng quy trình giao hàng bằng đa hình.

Yêu cầu chức năng:

1. Quản lý thông tin đơn hàng: sản phẩm, số lượng, tổng tiền.
2. Hỗ trợ nhiều hình thức thanh toán khác nhau, ví dụ:
 - CashOnDelivery, BankTransfer, EWallet.
3. Hỗ trợ nhiều đơn vị vận chuyển:
 - VNPost, GiaoHangNhanh, AhaMove.
4. Cho phép người dùng chọn phương thức thanh toán và vận chuyển.
5. Tính linh hoạt: thêm phương thức mới **mà không sửa code cũ**.

Gợi ý thiết kế:

- Dùng interface `PaymentMethod` với phương thức `pay(double amount)`.
- Dùng interface `ShippingMethod` với phương thức `ship(String address)`.
- Lớp `Order` sẽ nhận các đối tượng `PaymentMethod` và `ShippingMethod` thông qua **đa hình**.
- Kiểm thử bằng cách tạo nhiều đơn hàng khác nhau.

Kỹ năng đạt được:

- Thiết kế hệ thống mở rộng với **interface** và **đa hình**.
- Áp dụng **Strategy Pattern**.
- Tư duy phân tách trách nhiệm và ràng buộc lỏng (loose coupling).

Dự án 2: Ứng dụng Quản lý Nhân sự Công ty

Tình huống:

Xây dựng một ứng dụng Java để quản lý các loại nhân viên khác nhau (toàn thời gian, bán thời gian, hợp đồng), mỗi loại có cách tính lương và báo cáo khác nhau.

Phân công:

- **Thành viên A:** Thiết kế cấu trúc lớp nhân viên (Employee, FullTime, PartTime, Contract) và xử lý tính lương.
- **Thành viên B:** Thiết kế hệ thống báo cáo lương và xuất thông tin nhân viên.

Yêu cầu chức năng:

1. Quản lý danh sách nhân viên với các loại hình khác nhau.
2. Mỗi loại nhân viên có cách tính lương riêng:
 - FullTime: lương cơ bản + thưởng.
 - PartTime: tính theo số giờ.
 - Contract: gói trả theo tháng.
3. Mỗi nhân viên có cách báo cáo khác nhau (generateReport()).
4. Cho phép xuất tổng lương toàn bộ nhân viên.
5. Dễ dàng mở rộng hệ thống để thêm loại nhân viên mới mà không thay đổi code cũ.

Gợi ý thiết kế:

- Tạo lớp trừu tượng Employee có các phương thức:

- calculateSalary(), generateReport().
- Ghi đè phương thức ở các lớp con để mô tả đa hình.
- Lớp Company có danh sách nhân viên và xử lý tổng hợp thông tin.

Kỹ năng đạt được:

- Hiểu rõ **đa hình động** thông qua override và danh sách đối tượng kiểu cha.
- Áp dụng nguyên lý **Open/Closed** trong OOP.
- Rèn tư duy tổ chức mã và phân chia trách nhiệm nhóm.