POLYMORPHISM TRONG JAVA OOP — TOÀN DIỆN

Phần 1: Từ mục 1 đến mục 4

MỤC LỤC

1. Định nghĩa, nguồn gốc và ý nghĩa lý thuyết của đa hình

1.1. Định nghĩa

Đa hình (Polymorphism) là khả năng của một đối tượng, phương thức hoặc hàm có thể có nhiều "dạng" (nhiều hình thức biểu hiện khác nhau) tùy vào ngữ cảnh thực thi. Trong Java (và OOP nói chung), đa hình cho phép cùng một phương thức hoặc biến tham chiếu xử lý các đối tượng thuộc các lớp con khác nhau, dẫn tới hành vi khác nhau. Có hai loại đa hình chính:

- Đa hình biên dịch (static/compile-time): phương thức/hàm cùng tên nhưng khác tham số (overloading).
- Đa hình chạy (dynamic/runtime): cùng một lời gọi phương thức, nhưng thực thi khác nhau dựa vào object thực sự (overriding, interface).

Tài liệu tham khảo: Oracle Java Tutorials – Polymorphism
https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

1.2. Nguồn gốc lý thuyết

Polymorphism xuất phát từ tiếng Hy Lạp: "poly" (nhiều), "morph" (hình dạng). Xuất hiện sớm trong các ngôn ngữ OOP như Simula, Smalltalk, C++, Java — nhấn mạnh vào khả năng "giao tiếp" với nhiều kiểu đối tượng qua một giao diện chung.

1.3. Ý nghĩa của đa hình trong OOP

- Cho phép xử lý tập hợp các đối tượng cùng nhóm (kế thừa hoặc cùng interface) một cách linh hoạt.

- Giúp code “mở rộng” (extensible): Khi thêm class mới, code cũ vẫn hoạt động được.
- Là cơ sở để xây dựng những hệ thống tổng quát, reusable, dễ bảo trì.

2. Vai trò, tác dụng của tính đa hình trong phát triển phần mềm

- Linh hoạt hóa mã nguồn: Lập trình viên có thể viết code “chung” (tổng quát), nhưng vẫn xử lý chính xác từng trường hợp cụ thể khi chạy.
- Tăng khả năng mở rộng: Thêm đối tượng mới không ảnh hưởng code cũ.
- Tối ưu hóa cho thiết kế dựa trên interface/abstract class: Dễ mở rộng hệ thống mà không cần chỉnh sửa logic đang chạy.
- Hỗ trợ xây dựng các cấu trúc module mạnh mẽ (ví dụ: hệ thống plugin, kiến trúc theo event, ...).
- Tăng khả năng kiểm thử và bảo trì: Đơn vị code kiểm thử có thể thay thế các implement khác nhau (test double, mock,...).

3. Cách hiện thực hóa (triển khai) đa hình trong Java

3.1. Đa hình biên dịch (Compile-time/Static Polymorphism)

Thể hiện qua method overloading (nạp chồng phương thức): Nhiều phương thức cùng tên trong cùng một class, khác nhau về tham số (số lượng, kiểu). Trình biên dịch xác định phương thức gọi đúng ngay tại thời điểm biên dịch.

```
class MathUtils {
    int sum(int a, int b) { return a + b; }
    double sum(double a, double b) { return a + b; }
    int sum(int a, int b, int c) { return a + b + c; }
}
```

3.2. Đa hình chạy (Runtime/Dynamic Polymorphism)

Thể hiện qua method overriding (ghi đè phương thức): Lớp con định nghĩa lại phương thức của lớp cha với cùng tên, cùng tham số. Java quyết định gọi phương thức lớp cha hay lớp con dựa vào object thực tế tại thời điểm runtime (dù biến tham chiếu kiểu cha). Cơ sở cho tính đa hình thực sự: biến kiểu lớp cha (hoặc interface) có thể chứa object thuộc bất kỳ lớp con nào.

```
class Animal { void sound() { System.out.println("Some sound"); } }
class Dog extends Animal { void sound() { System.out.println("Woof"); } }
class Cat extends Animal { void sound() { System.out.println("Meow"); } }

Animal a = new Dog();
```

```
a.sound(); // "Woof"
a = new Cat();
a.sound(); // "Meow"
```

3.3. Overloading vs Overriding

- Overloading: Cùng tên phương thức, khác tham số, cùng/lớp con. Xảy ra tại compile-time.
- Overriding: Cùng tên phương thức, cùng tham số, khác class (phải có quan hệ kế thừa). Xảy ra tại runtime.

3.4. Từ khóa và cấu trúc liên quan

- @Override: Annotation thông báo ghi đè method cha.
- super: Dùng để gọi method của lớp cha từ lớp con.
- abstract class/interface: Tăng tính đa hình, cho phép nhiều lớp con thực hiện hành vi chung với implement khác nhau.

4. Ví dụ mã nguồn minh họa từ đơn giản đến nâng cao

4.1. Đa hình biên dịch (Overloading):

```
class PrintUtils {
    void print(int a) { System.out.println(a); }
    void print(String s) { System.out.println(s); }
}
PrintUtils p = new PrintUtils();
p.print(5);       // In ra: 5
p.print("Hello");  // In ra: Hello
```

4.2. Đa hình runtime (Overriding, interface):

```
class Shape {
    void draw() { System.out.println("Vẽ hình"); }
}
class Circle extends Shape {
    @Override
    void draw() { System.out.println("Vẽ hình tròn"); }
}
class Square extends Shape {
    @Override
    void draw() { System.out.println("Vẽ hình vuông"); }
}
```

```
Shape s = new Circle();
s.draw(); // Vẽ hình tròn

s = new Square();
s.draw(); // Vẽ hình vuông
```

4.3. Ví dụ nâng cao với abstract class, interface:

```
interface Animal {
    void speak();
}
class Dog implements Animal {
    public void speak() { System.out.println("Gâu gâu"); }
}
class Cat implements Animal {
    public void speak() { System.out.println("Meo meo"); }
}
void letAnimalSpeak(Animal a) {
    a.speak(); // Đa hình runtime
}
```

4.4. So sánh kết quả thực thi và phân tích code

- Khi gọi animal.speak(), Java kiểm tra object thực tế ở runtime để xác định phương thức nào sẽ được gọi.
- Việc này giúp mã dễ mở rộng, dễ kiểm thử và tuân thủ nguyên lý “open/closed principle” trong lập trình hướng đối tượng.