# Session 09: Kế Thừa & Đa Hình

Khám phá hai trụ cột của lập trình hướng đối tượng: Kế thừa giúp tái sử dụng code thông minh, Đa hình mang đến sự linh hoạt tuyệt vời cho ứng dụng.

RIKEI Academy®
where the dream begins

NỘI DUNG

# Chương Trình Học Hôm Nay

01

## Kế thừa trong Java

Mối quan hệ IS-A và từ khóa extends

02

## Ghi đè phương thức

Tùy chỉnh hành vi với @Override

03

## Từ khóa final, super

Kiểm soát kế thừa hiệu quả

04

## Đa hình cơ bản

Một giao diện, nhiều triển khai

05

## Kiểu khai báo vs thực tế

Compile-time và runtime type

06

## Overloading vs Overriding

So sánh hai cơ chế đa hình

LESSON 01

# Kế Thừa Trong Java

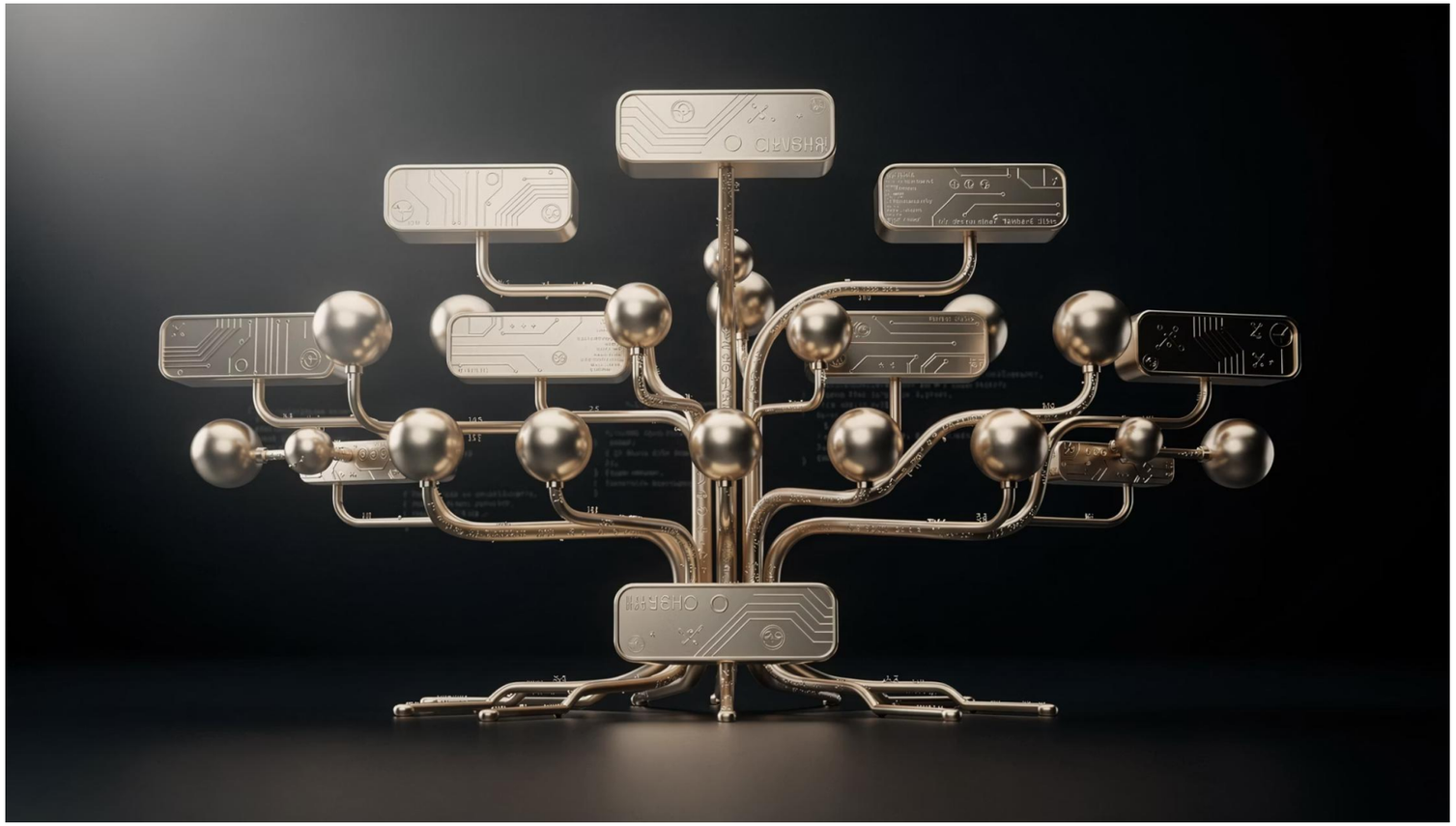
RIKKEI Academy
where the dream begins

## Kế thừa là gì?

Kế thừa cho phép lớp con kế thừa thuộc tính và phương thức từ lớp cha, tạo mối quan hệ **IS-A** (là một). Đây là nền tảng của lập trình hướng đối tượng.

## Cú pháp

```
class LopCon extends LopCha {
    // Kế thừa và mở rộng
}
```



**Tái sử dụng code**

Giảm code trùng lặp

**Mối quan hệ IS-A**

Cấu trúc phân cấp rõ ràng

**Dễ bảo trì**

Thay đổi ở cha lan tỏa xuống con

# Ba Loại Kế Thừa Trong Java

### Đơn Kế Thừa

Một lớp con kế thừa từ một lớp cha duy nhất

*Ví dụ: Student extends Person*

### Kế Thừa Nhiều Cấp

Lớp con kế thừa từ lớp cha, lớp cha kế thừa từ lớp ông

*Ví dụ: Animal → Mammal → Dog*

### Kế Thừa Phân Cấp

Nhiều lớp con cùng kế thừa từ một lớp cha

*Ví dụ: Employee → Manager, Developer*

**Lưu ý:** Java KHÔNG hỗ trợ đa kế thừa với class. Sử dụng interface để implements nhiều hành vi.

VÍ DỤ THỰC TẾ

# Hệ Thống Quản Lý Nhân Viên

**NhanVien**

tên, tuổi, lương cơ bản

**GiaoVien**

môn giảng dạy, số giờ dạy

**QuanLy**

phòng ban, số nhân viên quản lý

## Cấu trúc kế thừa

Lớp **NhanVien** chứa các thuộc tính chung: tên, tuổi, lương cơ bản.

Lớp **GiaoVien** kế thừa và bổ sung: môn giảng dạy, số giờ dạy.

Lớp **QuanLy** kế thừa và bổ sung: phòng ban, số nhân viên quản lý.

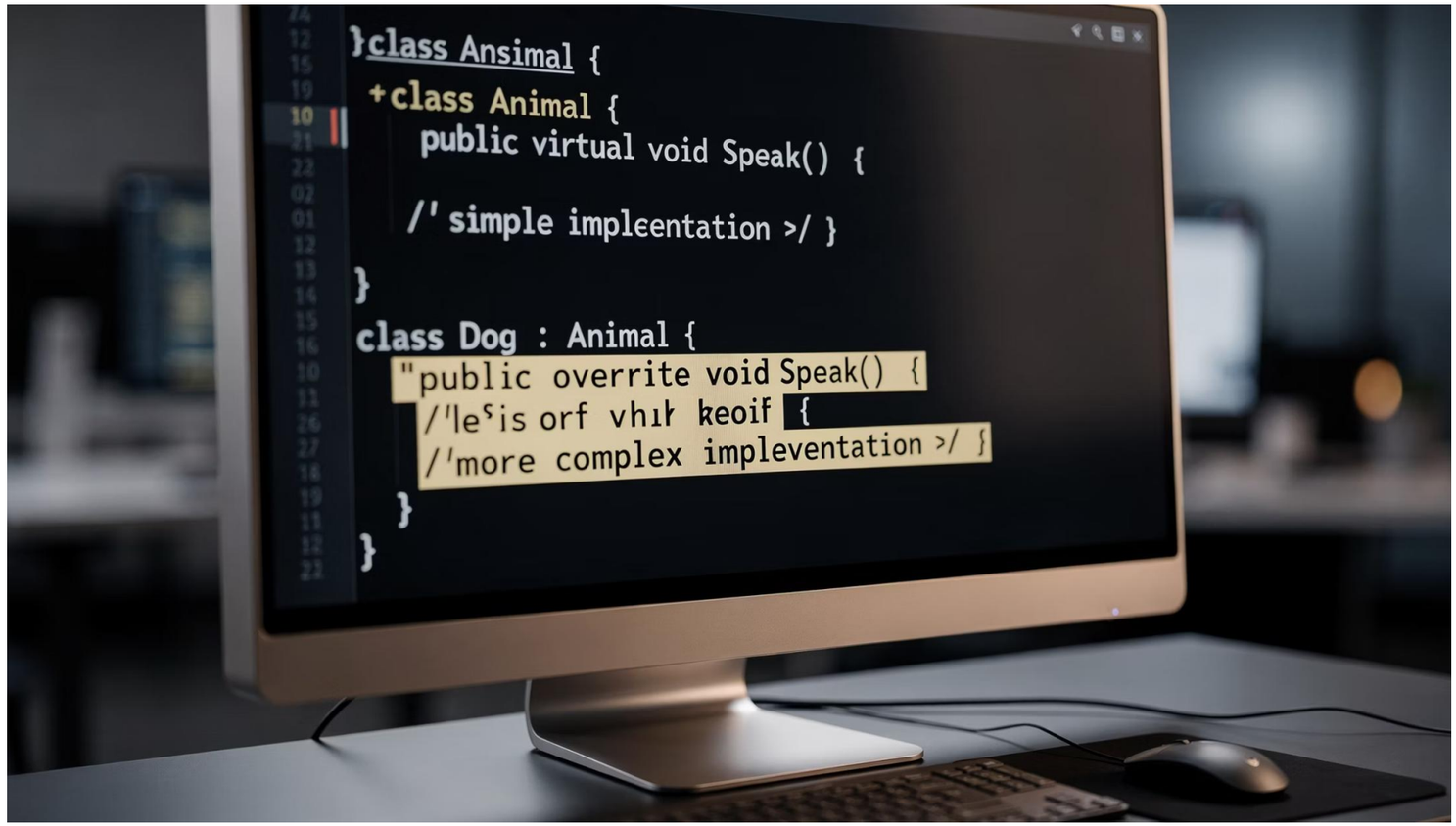
Mỗi lớp con có cách tính lương riêng phù hợp với vai trò.

LESSON 02

# Ghi Đè Phương Thức

## Overriding là gì?

Ghi đè cho phép lớp con thay đổi cách triển khai phương thức đã có trong lớp cha. Tên, tham số phải giống hệt, nhưng nội dung thực thi khác.




## Cú pháp

```
@Override
void phuongThuc() {
    // Triển khai mới
}
```

## Lợi ích @Override

- Compiler kiểm tra lỗi
- Code rõ ràng, dễ đọc
- Phát hiện lỗi sớm khi refactor

# Quy Tắc Ghi Đè Phương Thức

RIKEI Academy
where the dream begins

1

### Tên giống hệt

Tên phương thức phải giống 100%

2

### Tham số giống nhau

Số lượng, kiểu và thứ tự phải khớp

3

### Kiểu trả về tương thích

Giống hoặc là subtype của lớp cha

4

### Access modifier không hẹp hơn

Public không thể thành private

5

### Không ghi đè final

Phương thức final không thể override

### ✓ Ví dụ đúng

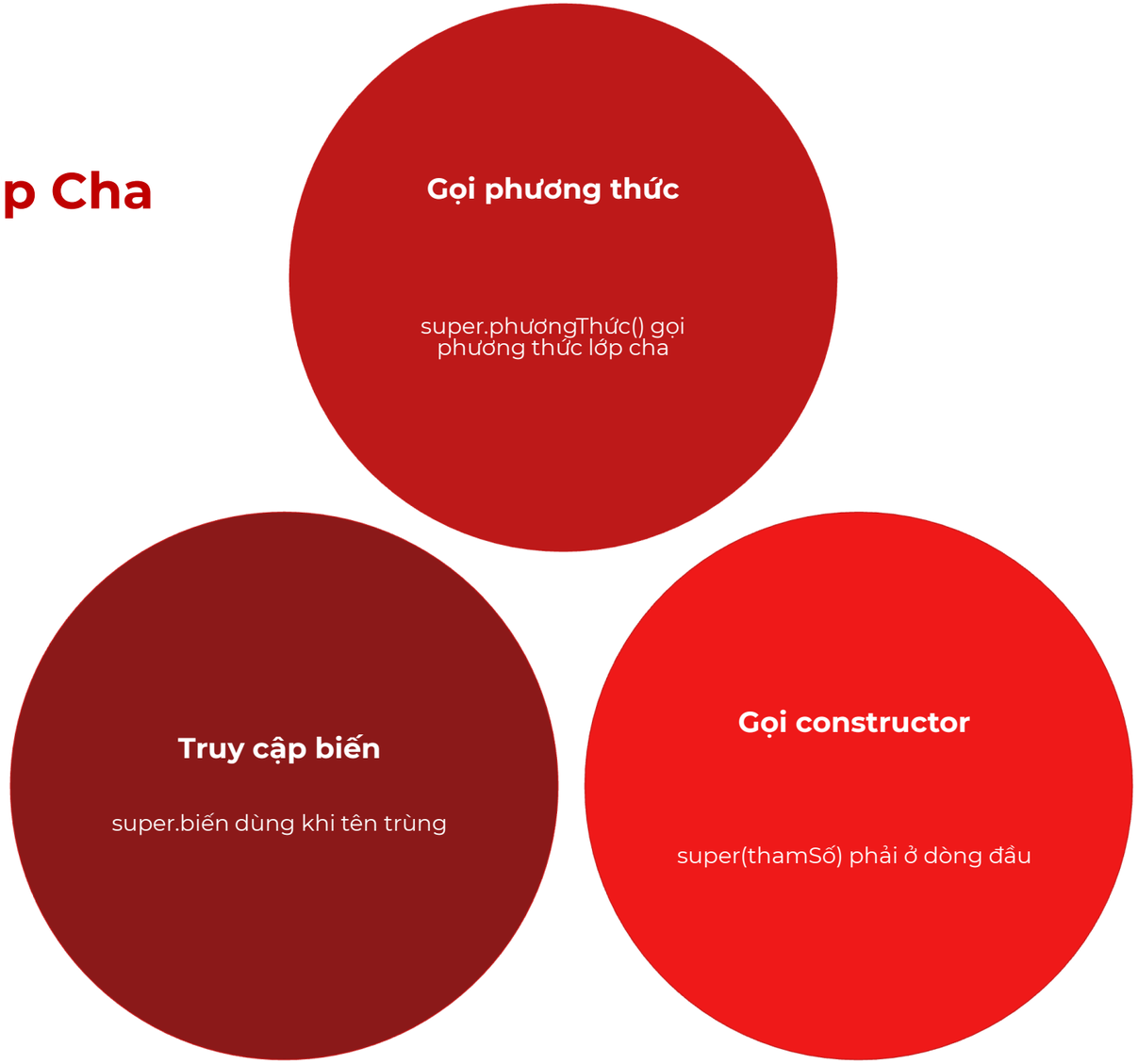
```
class ConCho extends DongVat {
    @Override
    void keu() {
        System.out.println("Gâu gâu!");
    }
}
```

### ✗ Ví dụ sai

```
class ConCho extends DongVat {
 @Override
 private void keu() { // SAI!
 // Access hẹp hơn
 }
}
```

LESSON 03

# Từ Khóa Super - Truy Cập Lớp Cha




**super.bien**
Truy cập biến của lớp cha khi bị trùng tên

**super.method()**
Gọi phương thức gốc từ lớp cha

**super(thamSo)**
Gọi constructor lớp cha (phải là dòng đầu tiên)

# Phương Thức super() Chi Tiết

## Vai trò của super()

Phương thức **super()** gọi constructor lớp cha để khởi tạo các thuộc tính kế thừa một cách chính xác.

## Quy tắc bắt buộc

- Phải là câu lệnh **đầu tiên** trong constructor
- Nếu không gọi, Java tự động gọi super() không tham số
- Truyền tham số phù hợp với constructor cha
- Compiler báo lỗi nếu vi phạm

## Ví dụ

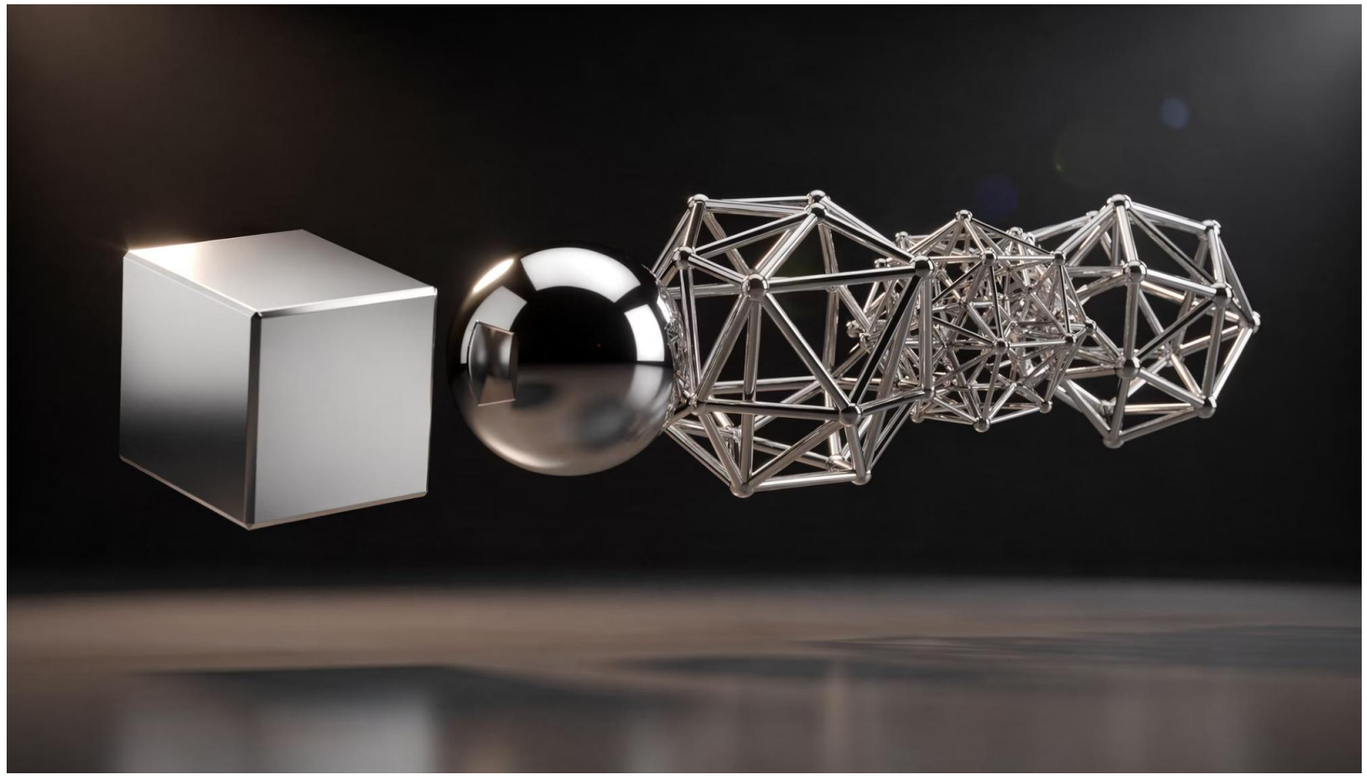
```
class NhanVien {
 protected String ten;
 protected double luong;

 public NhanVien(String ten, double luong) {
 this.ten = ten;
 this.luong = luong;
 }
}

class GiaoVien extends NhanVien {
 private String monHoc;

 public GiaoVien(String ten, double luong, String monHoc)
{
 super(ten, luong); // Khởi tạo lớp cha
 this.monHoc = monHoc;
 }
}
```

# Đa Hình - Polymorphism



## Đa hình là gì?

Đa hình có nghĩa là "nhiều hình thái" - một hành động được thực hiện theo nhiều cách khác nhau tùy đối tượng.

*Nguồn gốc: "Poly" = nhiều, "Morph" = hình thái*

## Ví dụ thực tế

Hành động "Kêu" → Chó: "Gâu gâu", Mèo: "Meo meo", Gà: "Ò ó o"

### Code linh hoạt

Thêm tính năng mới dễ dàng

### Giảm phụ thuộc

Module độc lập, dễ thay thế

### Dễ bảo trì

Thay đổi ít ảnh hưởng

### Code sạch

Tránh if-else phức tạp

# Hai Loại Đa Hình

## Compile-time Polymorphism

### OVERLOADING

Quyết định tại thời điểm **biên dịch**

Cùng tên phương thức, khác tham số

**Ví dụ**

```
int add(int a, int b)
double add(double a, double b)
int add(int a, int b, int c)
```

## Runtime Polymorphism

### OVERRIDING

Quyết định tại thời điểm **chạy**

Cùng tên, cùng tham số, khác triển khai

**Ví dụ**

```
DongVat animal = new Cho();
animal.keu();
// "Gâu gâu!" - runtime
```

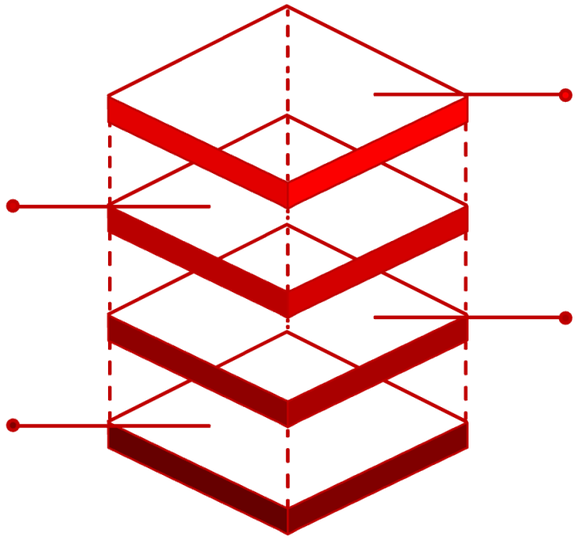
# Ứng Dụng Đa Hình Thực Tế

## Hệ thống tính lương

Mỗi loại nhân viên có cách tính lương riêng, nhưng đều thông qua phương thức `tinhLuong()`.

## Lợi ích

- Không cần if-else phân biệt loại
- Dễ thêm loại nhân viên mới
- Code sạch và dễ bảo trì



**DanhSach**

List<NhanVien> gọi tinhLuong()

**NVSanXuat**

Override tinhLuong cho sản xuất

**NVVanPhong**

Override tinhLuong cho văn phòng

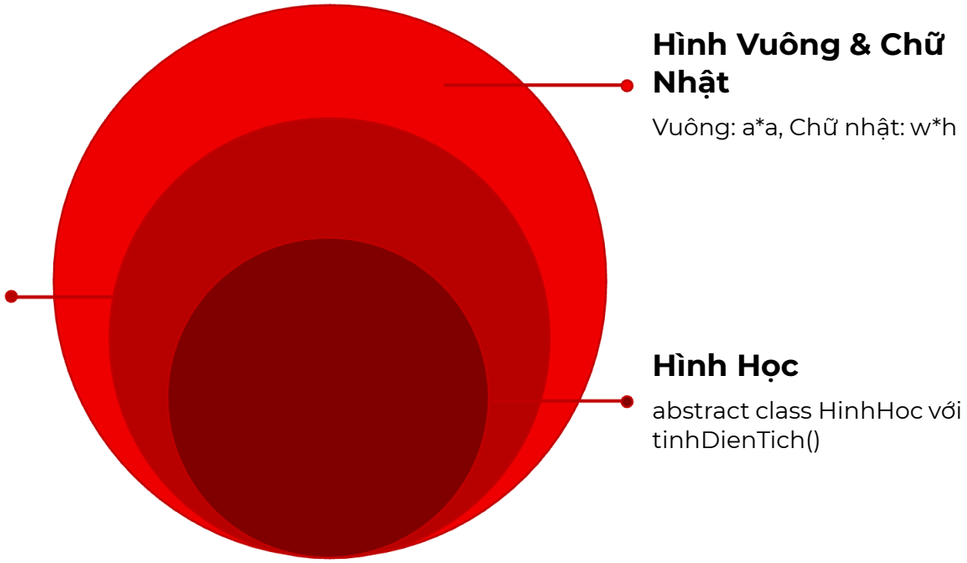
**NhanVien**

Lớp trừu tượng với tinhLuong()

```
List<NhanVien> danhSach = new ArrayList<>();
danhSach.add(new NVVanPhong());
danhSach.add(new NVSanXuat());

for (NhanVien nv : danhSach) {
 tongLuong += nv.tinhLuong(); // Mỗi loại tự biết cách tính
}
```

# Tính Diện Tích Các Hình Học




## Ứng dụng đa hình

Mỗi hình học có công thức tính diện tích riêng, nhưng đều kế thừa từ lớp **HinhHoc**.

## Cú pháp

```
abstract class HinhHoc {
    abstract double tinhDienTich();
}

class HinhTron extends HinhHoc {
    double tinhDienTich() {
        return Math.PI * r * r;
    }
}
```

Sử dụng kiểu cha để xử lý đa dạng hình con.

# Kiểu Khai Báo vs Kiểu Thực Tế

## Kiểu Khai Báo
## (Declared Type)

- Xác định tại **compile-time**
- Quyết định phương thức nào **có thể** gọi
- Dựa vào định nghĩa lớp

```
DongVat animal = new ConCho();
// Kiểu khai báo: DongVat
```

Chỉ gọi được phương thức của DongVat

## Kiểu Thực Tế
## (Actual Type)

- Xác định tại **runtime**
- Quyết định phương thức nào **thực sự** chạy
- Phụ thuộc đối tượng thực

```
DongVat animal = new ConCho();
// Kiểu thực tế: ConCho
```

Phương thức được gọi là phiên bản ConCho

**Ví dụ:** `animal.keu()` sẽ gọi phiên bản của ConCho, không phải DongVat.

# Ép Kiểu và Toán Tử instanceof

1

## UPCASTING

Lớp con → Lớp cha

**Tự động, an toàn**

```
ConCho cho = new ConCho();
DongVat animal = cho;
```

2

## DOWNCASTING

Lớp cha → Lớp con

**Tường minh, có rủi ro**

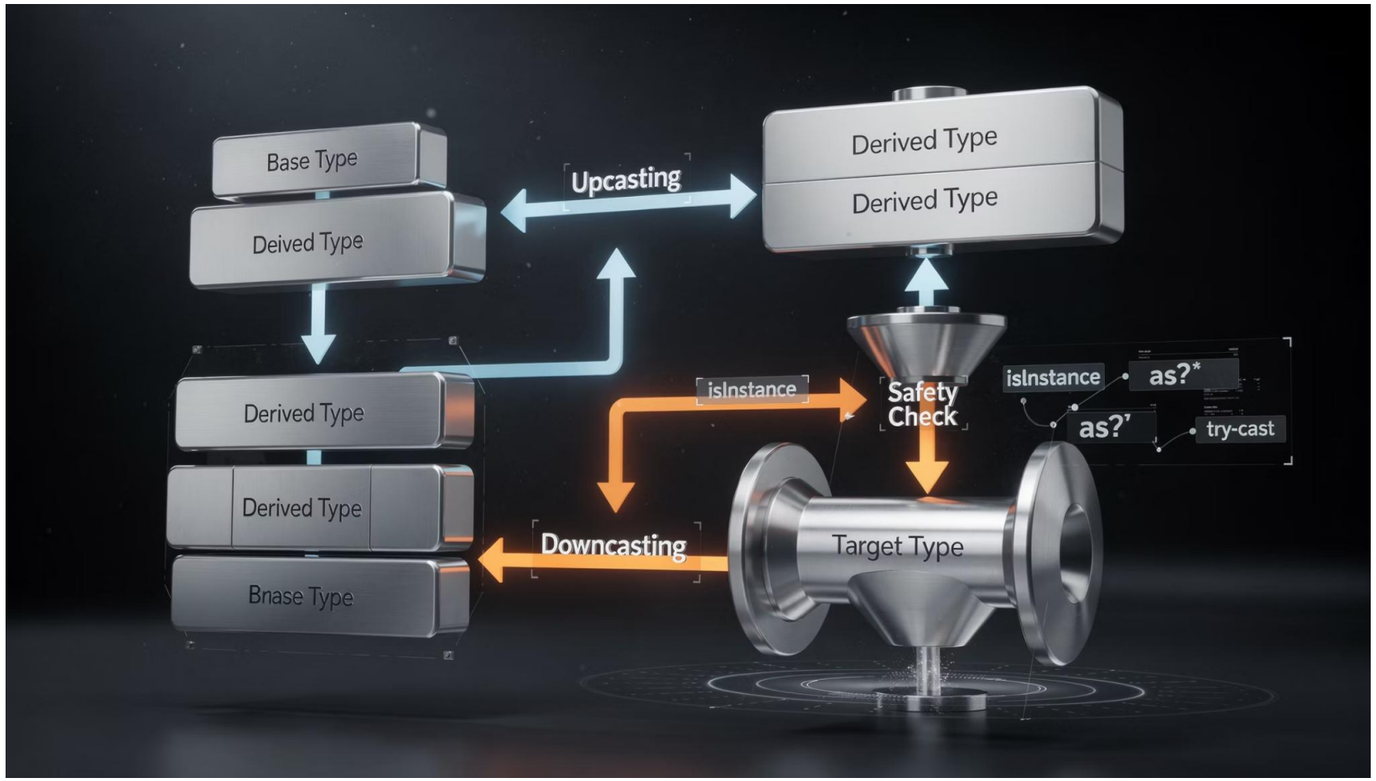
```
DongVat animal = new ConCho();
ConCho cho = (ConCho) animal;
```

## Toán tử instanceof

Kiểm tra kiểu đối tượng an toàn trước khi ép kiểu

### Cú pháp

```
if (animal instanceof ConCho) {
    ConCho cho = (ConCho) animal;
    cho.chayChaLa(); // An toàn
}
```



# Overloading - Nạp Chồng Phương Thức

## Overloading là gì?

Nhiều phương thức cùng tên nhưng khác tham số trong cùng một lớp. Đây là đa hình compile-time.

### Đặc điểm

- Cùng tên phương thức
- Khác số lượng, kiểu hoặc thứ tự tham số
- Có thể khác kiểu trả về
- Quyết định tại compile-time

### Lợi ích

Tăng tính linh hoạt, code tự nhiên, không cần đặt nhiều tên khác nhau.

### Ví dụ: Máy tính

```
class MayTinh {
 int cong(int a, int b) {
 return a + b;
 }

 double cong(double a, double b) {
 return a + b;
 }

 int cong(int a, int b, int c) {
 return a + b + c;
 }
}

// Sử dụng:
calc.cong(5, 3); // 8
calc.cong(5.5, 3.2); // 8.7
calc.cong(1, 2, 3); // 6
```

# Overriding - Ghi Đè Phương Thức

## Overriding là gì?

Lớp con cung cấp triển khai cụ thể cho phương thức đã có trong lớp cha. Đây là đa hình runtime.

### Đặc điểm

- Quan hệ kế thừa (cha-con)
- Tên và tham số giống hệt
- Kiểu trả về giống hoặc subtype
- Nội dung thay đổi phù hợp lớp con
- Quyết định tại runtime

Luôn dùng **@Override** để compiler kiểm tra

## Ví dụ: Hình học

```
class HinhHoc {
 double tinhDienTich() {
 return 0;
 }
}

class HinhTron extends HinhHoc {
 private double banKinh;

 @Override
 double tinhDienTich() {
 return Math.PI * banKinh * banKinh;
 }
}

// Sử dụng:
HinhHoc hinh = new HinhTron();
hinh.tinhDienTich(); // Gọi phiên bản HinhTron
```

# So Sánh Overloading vs Overriding

| Tiêu chí | Overloading | Overriding |
|---|---|---|
| Quan hệ | Trong cùng lớp | Giữa lớp cha-con |
| Tham số | Phải khác nhau | Phải giống hệt |
| Kiểu trả về | Có thể khác | Giống hoặc subtype |
| Thời điểm | Compile-time | Runtime |
| Mục đích | Tăng tính linh hoạt | Thay đổi hành vi kế thừa |
| Annotation | Không có | @Override |

### 💡 Ghi nhớ Overloading

"Cùng tên, khác tham số, trong một nhà"

### 💡 Ghi nhớ Overriding

"Con làm khác cha, cùng chữ ký"

TỔNG KẾT

# Tổng Kết Phần 1: Kế Thừa

**1**

### Kế Thừa (Inheritance)

Từ khóa `extends`, quan hệ IS-A, ba loại: đơn, nhiều cấp, phân cấp

**2**

### Ghi Đè (Overriding)

Annotation `@Override`, cùng tên và tham số, thay đổi nội dung

**3**

### Từ Khóa Quan Trọng

**final:** ngăn thay đổi | **super:** truy cập lớp cha | **super():** gọi constructor cha

Ba khái niệm này tạo nền tảng vững chắc cho kế thừa, cho phép xây dựng hệ thống có cấu trúc và dễ mở rộng.

TỔNG KẾT

# Tổng Kết Phần 2: Đa Hình

| Đa Hình | Kiểu Dữ Liệu | Ép Kiểu |
|---|---|---|
| Compile-time: **Overloading**<br>Runtime: **Overriding** | **Khai báo:** compile-time<br>**Thực tế:** runtime | Upcasting: tự động<br>Downcasting: tường minh + instanceof |

## Overloading

- Cùng lớp, cùng tên
- Khác tham số
- Compile-time

## Overriding

- Lớp cha-con
- Cùng chữ ký
- Runtime

# Ví Dụ Giải Bài Tập: Hệ Thống Hình Học

**HinhHoc**
Lớp trừu tượng định nghĩa giao diện

**tinhDienTich()**
Phương thức abstract trả diện tích

**tinhChuVi()**
Phương thức abstract trả chu vi

**HinhTron & HinhVuong**
Hai lớp con cài đặt phương thức

## Cấu trúc tổng quát

```
abstract class HinhHoc {
    abstract double tinhDienTich();
    abstract double tinhChuVi();
}

class HinhTron extends HinhHoc {
    @Override
    double tinhDienTich() {
        return Math.PI * r * r;
    }

    @Override
    double tinhChuVi() {
        return 2 * Math.PI * r;
    }
}
```

## Sử dụng đa hình

```
List<HinhHoc> danhSach = new ArrayList<>();
danhSach.add(new HinhTron(5));
danhSach.add(new HinhVuong(4));

for (HinhHoc hinh : danhSach) {
 tongDienTich += hinh.tinhDienTich();
 if (hinh instanceof HinhTron) {
 System.out.println("Đây là hình tròn");
 }
}
```

# Best Practices - Lập Trình Hiệu Quả

### Luôn dùng @Override

Compiler phát hiện lỗi sớm, code rõ ràng hơn

### Kiểm tra instanceof

Tránh ClassCastException khi downcasting

### Dùng abstract/interface

Định nghĩa hành vi chung, code linh hoạt

### Ưu tiên composition

Không phải lúc nào kế thừa cũng tốt nhất

### Tránh kế thừa quá sâu

Giới hạn 3-4 cấp, dễ hiểu và bảo trì

### Document đầy đủ

Giải thích rõ hành vi mong đợi của phương thức

TỔNG KẾT

# Những Điều Cần Nhớ

**Kế thừa** Tái sử dụng code hiệu quả với `extends`

**Ghi đè** Tùy chỉnh hành vi với `@Override`

**Từ khóa** `final`, `super`, `super()` để kiểm soát kế thừa

**Đa hình** Một giao diện, nhiều triển khai

**Kiểu dữ liệu** Phân biệt khai báo vs thực tế

**Overloading vs Overriding** Hai dạng đa hình khác nhau

RIKEI Academy®
where the dream begins

# KẾT THÚC

HỌC VIỆN ĐÀO TẠO LẬP TRÌNH CHẤT LƯỢNG NHẬT BẢN