

## **Buổi 1: Tổng quan & So sánh hai phương pháp lập trình**

### **Mục tiêu:**

- Hiểu được phương pháp tiếp cận truyền thống và hướng đối tượng
- So sánh sự khác nhau giữa hai phương pháp
- Làm quen với cú pháp Java và môi trường lập trình Phân bổ nội dung:

### **◆ Giới thiệu học phần & Tổng quan lập trình hướng đối tượng**

- Mục tiêu học phần & phương pháp đánh giá
- Lập trình truyền thống: Lập trình tuyến tính và lập trình cấu trúc
- Lập trình hướng đối tượng: Khái niệm và đặc điểm

### **◆ So sánh lập trình truyền thống và lập trình hướng đối tượng**

- Bảng so sánh chi tiết hai phương pháp
- Thực hành: Viết chương trình đơn giản theo hai cách tiếp cận

### **◆ Xu hướng lập trình & Thực hành**

- Xu hướng lập trình hướng đối tượng & lập trình hướng thành phần
- Hướng dẫn cài đặt Java & VS Code
- Bài tập thực hành: Chạy chương trình Java đầu tiên

## **Buổi 2: Các khái niệm & Kỹ thuật lập trình hướng đối tượng**

### **Mục tiêu:**

- Nắm bắt các khái niệm cơ bản trong OOP
- Hiểu và sử dụng các từ khóa quan trọng (this, super, final, static)
- Sử dụng đúng phạm vi truy cập

### **Phân bổ nội dung:**

### **◆ Các khái niệm cơ bản trong OOP**

- Đối tượng, lớp, kế thừa, đa hình, trừu tượng, đóng gói
- Ví dụ minh họa từng khái niệm

### **◆ Các từ khóa quan trọng trong lập trình hướng đối tượng**

- Giới thiệu từ khóa this, super, final, static
- Thực hành: Viết chương trình áp dụng từ khóa this và super

### **◆ Phạm vi truy cập & Bài tập tổng hợp**

- Phạm vi truy cập của biến, phương thức, constructor, lớp
- Thực hành: Viết chương trình minh họa phạm vi truy cập
- Bài tập tổng hợp: Xây dựng một lớp đơn giản áp dụng các từ khóa đã học

## PHẦN 1: CÁC KHÁI NIỆM CƠ BẢN TRONG OOP

- **Giới thiệu OOP** – Tại sao cần lập trình hướng đối tượng?
- **Các đặc trưng chính của OOP** – Tổng quan về **Đóng gói, Kế thừa, Đa hình, Trừu tượng**
- **Lớp và Đối tượng** – Cách khai báo, tạo và sử dụng **Constructor và Destructor** – Cách sử dụng trong Java
- **Từ khóa `this` và `super`** – Cách dùng trong thực tế **Các khái niệm khác** – Interface, Abstract Class

## PHẦN 2: CÁC TỪ KHÓA QUAN TRỌNG TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

- **Cách rút gọn:** Giảm số lượng từ khóa trình bày, tập trung vào từ khóa quan trọng nhất.
- **Từ khóa `static` và `final`** – Ứng dụng trong OOP
- **Từ khóa `abstract` và `interface`** – So sánh, khi nào dùng cái nào
- **Từ khóa `this`, `super`** – Tóm tắt cách dùng ngắn gọn
- **Từ khóa `extends`, `implements`** – Minh họa bằng code ngắn gọn
- **Tổng hợp & So sánh nhanh** – Đưa ra bảng tóm tắt

## PHẦN 3: PHẠM VI TRUY CẬP & BÀI TẬP TỔNG HỢP

- **Các mức độ phạm vi truy cập (`public`, `private`, `protected`, `default`)** – Bảng so sánh chi tiết
- **Ứng dụng phạm vi truy cập trong kế thừa** – Code minh họa
- **Tầm quan trọng của phạm vi truy cập trong bảo mật dữ liệu** – Ví dụ thực tế
- **Bài tập thực hành trên lớp** – Code & giải thích
- **Tình huống thực tế: Thiết kế một hệ thống quản lý đơn giản bằng OOP**
- **Giao bài tập về nhà** – Đề bài và hướng dẫn sinh viên tự thực hành

## 1 PHẦN 1: CÁC KHÁI NIỆM CƠ BẢN TRONG OOP

### 1.1 Giới thiệu OOP – Tại sao cần lập trình hướng đối tượng?

#### 1. Vấn đề của lập trình truyền thống (Procedural Programming)

- ◆ Mã nguồn dài, khó bảo trì.
- ◆ Dữ liệu và hàm xử lý tách biệt, dễ gây lỗi.
- ◆ Khó tái sử dụng code, khó mở rộng chương trình.

#### 2. Sự ra đời của Lập trình Hướng Đối Tượng (OOP)

- ✓ Giải quyết các hạn chế của lập trình truyền thống.
- ✓ Tổ chức chương trình theo mô hình **lớp (class)** và **đối tượng (object)**.

✓ Giúp mã nguồn dễ hiểu, dễ bảo trì và mở rộng.

## Lợi ích của OOP

✦ **Tái sử dụng (Reusability):** Dùng lại code thông qua kế thừa.

✦ **Bảo mật (Encapsulation):** Hạn chế truy cập trực tiếp vào dữ liệu.

✦ **Mở rộng (Extensibility):** Dễ dàng nâng cấp và phát triển hệ thống.

✦ **Quản lý dữ liệu tốt hơn:** Giảm lỗi, tăng tính nhất quán.

## ✦ Kết luận:

☞ OOP là phương pháp lập trình hiện đại, giúp phát triển phần mềm hiệu quả và bền vững hơn!

## 1.2 Các đặc trưng chính của OOP – Tổng quan về Đóng gói, Kế thừa, Đa hình, Trừu tượng

### 1.2.1 Slide 1: Các đặc trưng chính của OOP – Tổng quan

✦ **Lập trình hướng đối tượng (OOP) dựa trên 4 đặc trưng chính:**

1. Đóng gói (Encapsulation)

2. Kế thừa (Inheritance)

3. Đa hình (Polymorphism)

4. Trừu tượng (Abstraction)

✦ **Mục tiêu:** Giúp mã nguồn **gọn gàng, dễ bảo trì, mở rộng và tăng tính bảo mật.**

### 1.2.2 Slide 2: Đóng gói (Encapsulation)

#### ◆ Khái niệm:

✓ Che giấu dữ liệu bên trong đối tượng, chỉ cho phép truy cập qua phương thức được cung cấp.

#### ◆ Lợi ích:

✓ □ Bảo vệ dữ liệu khỏi truy cập trái phép.

✓ □ Dễ bảo trì, sửa đổi mà không ảnh hưởng đến phần còn lại của chương trình.

#### ◆ Ví dụ trong Java:

```
class Student {  
    private String name; // Dữ liệu bị đóng gói  
    public void setName(String newName) {  
        name = newName; }  
}
```

```
    public String getName() {  
        return name; }  
}
```

☞ **Kết luận:** Đóng gói giúp kiểm soát tốt hơn việc truy cập và chỉnh sửa dữ liệu trong chương trình.

### 1.2.3 Slide 3: Kế thừa (Inheritance) & Đa hình (Polymorphism)

#### 1. Kế thừa (Inheritance)

◆ **Khái niệm:** Lớp con có thể kế thừa thuộc tính và phương thức từ lớp cha.

◆ **Lợi ích:**

✓□ Tái sử dụng mã nguồn, giúp tiết kiệm thời gian.

✓□ Mở rộng tính năng mà không cần chỉnh sửa lớp cha.

◆ **Ví dụ trong Java:**

```
class Animal {  
    void makeSound() { System.out.println("Some sound..."); } }  
class Dog extends Animal {  
    void makeSound() { System.out.println("Bark!"); } }
```

☞ Lớp Dog kế thừa từ Animal và có thể thay đổi hoặc mở rộng chức năng.

#### 2. Đa hình (Polymorphism)

◆ **Khái niệm:** Một phương thức có thể có nhiều cách triển khai khác nhau.

◆ **Lợi ích:**

✓□ Tăng tính linh hoạt khi lập trình.

✓□ Giảm số lượng mã lặp lại, dễ bảo trì.

◆ **Ví dụ trong Java:**

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

👉 Gọi `makeSound()` trên các đối tượng `Dog`, `Cat` sẽ cho kết quả khác nhau.

## 1.2.4 Slide 4: Trừu tượng (Abstraction) & Tổng kết

### 1. Trừu tượng (Abstraction)

◆ **Khái niệm:** Ẩn đi chi tiết thực thi, chỉ hiển thị phần quan trọng cho người dùng.

◆ **Lợi ích:**

✓ □ Đơn giản hóa lập trình, tập trung vào chức năng chính.

✓ □ Giúp xây dựng hệ thống **mở rộng** dễ dàng hơn.

◆ **Ví dụ trong Java:**

```
abstract class Animal {  
    abstract void makeSound();  
}  
class Dog extends Animal {  
    void makeSound() {  
        System.out.println("Bark!");  
    }  
}
```

👉 Lớp `Animal` chỉ định nghĩa phương thức `makeSound()`, nhưng không triển khai chi tiết!

### Tổng kết 4 đặc trưng của OOP:

✓ **Đóng gói:** Bảo vệ dữ liệu, tránh truy cập trái phép.

✓ **Kế thừa:** Tái sử dụng mã nguồn, dễ mở rộng.

✓ **Đa hình:** Một hành vi có thể có nhiều hình thái khác nhau.

✓ **Trừu tượng:** Ẩn đi chi tiết không cần thiết, giúp lập trình dễ dàng hơn.

👉 **Kết luận:** Hiểu rõ 4 đặc trưng này giúp bạn viết mã hiệu quả, dễ bảo trì và mở rộng!  
🚀

## 1.3 Lớp và Đối tượng – Cách khai báo, tạo và sử dụng

### 1.3.1 Slide 1: Lớp và Đối tượng – Giới thiệu

✦ **Lập trình hướng đối tượng xoay quanh hai khái niệm chính:**

◆ **Lớp (Class)** – Mẫu thiết kế (blueprint) cho các đối tượng.

◆ **Đối tượng (Object)** – Thực thể cụ thể được tạo ra từ lớp.

💡 **Ví dụ thực tế:**

**Lớp:** Bản thiết kế của một chiếc ô tô.

**Đối tượng:** Một chiếc ô tô cụ thể được sản xuất từ bản thiết kế đó.

### 🔍 Mô hình tổng quát:

Lớp (Class) → Định nghĩa thuộc tính & hành vi → Tạo ra nhiều đối tượng (Objects)

## 1.3.2 Slide 2: Khai báo lớp và tạo đối tượng trong Java

◆ **Khai báo lớp:** Một lớp trong Java bao gồm:

✓□ **Thuộc tính (fields):** Biến lưu trữ trạng thái của đối tượng.

✓□ **Phương thức (methods):** Hành vi của đối tượng.

◆ **Ví dụ khai báo lớp:**

```
class Student {  
    String name;  
    int age;  
    void displayInfo() {  
        System.out.println(name + " - " + age + " tuổi");  
    }  
}
```

◆ **Tạo đối tượng từ lớp:**

```
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student(); // Tạo đối tượng  
        s1.name = "Minh";  
        s1.age = 20;  
        s1.displayInfo();  
    }  
}
```

👉 **Kết quả:** Minh - 20 tuổi

💡 **Lưu ý:** Mỗi đối tượng có trạng thái riêng, nhưng đều tuân theo thiết kế của lớp.

## 1.3.3 Slide 3: Tương tác với đối tượng & Tổng kết

◆ **Truy cập và cập nhật dữ liệu đối tượng**

✓□ Dữ liệu trong đối tượng có thể được truy xuất và thay đổi thông qua phương thức:

```
s1.name = "An";  
System.out.println(s1.name);
```

✓☐ Có thể tạo nhiều đối tượng từ cùng một lớp:

```
Student s2 = new Student();  
s2.name = "Lan";  
s2.age = 22;  
s2.displayInfo();
```

### ✦ Tổng kết:

✓ **Lớp:** Định nghĩa chung về dữ liệu và hành vi.

✓ **Đối tượng:** Thực thể cụ thể của lớp, có trạng thái riêng.

✓ **Tạo và sử dụng đối tượng trong Java:** Sử dụng từ khóa new để khởi tạo.

✍ **Hiểu rõ về lớp và đối tượng là bước quan trọng để lập trình hướng đối tượng hiệu quả!**

## 1.4 Constructor và Destructor – Cách sử dụng trong Java

### 1.4.1 Slide 1: Constructor – Cách sử dụng trong Java

**Constructor là gì?**

◆ **Constructor** là một phương thức đặc biệt trong lớp, được gọi khi đối tượng được tạo.

◆ Có nhiệm vụ **khởi tạo giá trị ban đầu** cho đối tượng.

◆ **Tên constructor** trùng với tên lớp và **không có kiểu trả về**.

◆ **Ví dụ về Constructor trong Java:**

```
class Student {  
    String name; int age;  
    Student(String n, int a) { // Constructor  
        name = n;  
        age = a; }  
    void display() {  
        System.out.println(name + " - " + age + " tuổi");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student("Minh", 20); // Gọi constructor  
        s1.display();  
    }  
}
```

}

☞ Khi tạo đối tượng `new Student("Minh", 20)`, constructor tự động chạy.

💡 Lưu ý:

✓ Có thể có nhiều constructor (Overloading constructor).

✓ Nếu không định nghĩa constructor, Java sẽ tự tạo một constructor mặc định không tham số.

## 1.4.2 Slide 2: Destructor & Tổng kết

✦ Destructor trong Java là gì?

⇒ Java không có Destructor như C++!

⇒ Java sử dụng **Garbage Collector (GC)** để tự động thu hồi bộ nhớ của đối tượng không còn được tham chiếu.

💡 Cách hủy đối tượng trong Java:

✓☐ Gán null cho đối tượng:

```
Student s1 = new Student("Nam", 21);  
s1 = null; // Đối tượng sẽ bị GC dọn dẹp
```

✓☐ Gọi `System.gc()`; để yêu cầu GC chạy:

```
System.gc(); // Không đảm bảo GC sẽ chạy ngay lập tức
```

✓☐ Dùng phương thức `finalize()` (ít dùng):

```
protected void finalize() {  
    System.out.println("Đối tượng bị hủy!");  
}
```

✦ Tổng kết:

✓ Constructor giúp khởi tạo đối tượng khi được tạo bằng `new`.

✓ Java không có Destructor, mà sử dụng **Garbage Collector** để quản lý bộ nhớ.

✓ Không cần lo về quản lý bộ nhớ thủ công như trong C++.

🔗 Hiểu rõ constructor giúp tối ưu hóa code và đảm bảo chương trình chạy hiệu quả hơn!



## Từ khóa this và super – Cách dùng trong thực tế

### 1.4.3 Slide 1: Từ khóa this – Cách dùng trong thực tế

#### ✦ Từ khóa **this** là gì?

- ✦ **this** là một biến tham chiếu đặc biệt, trỏ đến đối tượng hiện tại.
- ✦ Được sử dụng bên trong lớp để **truy xuất biến, gọi phương thức hoặc constructor** của chính đối tượng đó. **!** Các cách sử dụng **this**:

#### ✓□ 1. Phân biệt biến instance và biến local (tránh xung đột tên):

```
class Student { String name;
    Student(String name) {
        this.name = name; // 'this.name' là biến instance, 'name' là
        tham số
    }
}
```

#### ✓□ 2. Gọi constructor khác trong cùng một lớp:

```
class Student {
    String name;
    int age;
    Student() {
        this("Không tên", 18); // Gọi constructor có tham số
    }
    Student(String name, int age) {
        this.name = name;
        this.age = age;}
}
```

#### ✓□ 3. Truy xuất phương thức của lớp hiện tại:

```
class Student {
    void show() {
        System.out.println("Hello Java");}
    void display() {
        this.show(); // Gọi phương thức show()
    }
}
```

📌 **Lưu ý:** Nếu không có sự trùng tên giữa biến instance và biến local, có thể bỏ **this**.

#### 1.4.4 Slide 2: Từ khóa super – Cách dùng trong thực tế

##### ✈ *Từ khóa super là gì?*

- ✦ super dùng để tham chiếu trực tiếp đến **lớp cha (superclass)**.
- ✦ Giúp gọi **constructor**, **phương thức**, hoặc **biến** của **lớp cha** từ lớp con.

##### 💡 **Các cách sử dụng super:**

###### ✓□ 1. *Gọi constructor của lớp cha:*

```
class Person {  
    Person() { System.out.println("Constructor lớp Person");}  
}  
  
class Student extends Person { Student() {  
    super(); // Gọi constructor của Lớp cha  
    System.out.println("Constructor lớp Student");}  
}
```

###### ✓□ 2. *Gọi phương thức của lớp cha (khi bị ghi đè - override):*

```
class Person {  
    void show() { System.out.println("Lớp cha"); }  
}  
  
class Student extends Person { void show() {  
    super.show(); // Gọi phương thức của Lớp cha  
    System.out.println("Lớp con");}  
}
```

###### ✓□ 3. *Truy xuất biến của lớp cha:*

```
class Person {  
    String name = "Người cha";  
}  
class Student extends Person { String name = "Người con";
```

```
        vo
id
showName
e() {
    S
    ys
    te
    m.
    ou
    t.
    pr
    in
    tl
    n(
    su
    pe
    r.
    na
    me
    );
    }/
    /
    Lã
    y
    gi
    á
    tr
    ì
    từ
    Lớ
    p
    ch
    a
```

}

### ✈️ Tổng kết:

- ✓ **this** trỏ đến đối tượng hiện tại, giúp truy xuất biến, phương thức hoặc gọi constructor trong cùng lớp.
- ✓ **super** trỏ đến lớp cha, giúp truy cập constructor, phương thức hoặc biến của lớp cha.
- ✈️ Hiểu rõ **this** và **super** giúp code dễ đọc, tránh xung đột và tận dụng tốt tính kế thừa của OOP!

## 1.5 Các khái niệm khác – Interface, Abstract Class

### 1.5.1 Slide 1: Abstract Class – Lớp trừu tượng là gì?

#### ✈️ Abstract Class là gì?

- ◆ Lớp trừu tượng (Abstract Class) là lớp **không thể tạo đối tượng trực tiếp**, chỉ dùng để làm nền tảng cho các lớp con.
- ◆ Có thể chứa **phương thức trừu tượng (không có phần thân)** và **phương thức thông thường (có phần thân)**.
- ◆ Giúp **định nghĩa chung** cho các lớp con, nhưng vẫn có thể có **chức năng riêng**.

#### 💡 Ví dụ Abstract Class trong Java:

```
abstract class Animal {  
    abstract void makeSound(); // Phương thức trừu tượng  
    void sleep() { System.out.println("Đang ngủ..."); } //  
    Phương thức thông thường  
}  
  
class Dog extends Animal {  
    void makeSound() { System.out.println("Gâu gâu!"); }  
}  
  
public class Main {  
    public static void main(String args[]) {  
        Dog d = new Dog();  
        d.makeSound(); // Gâu gâu!  
        d.sleep(); // Đang ngủ...}}}
```

### ✈️ Tóm tắt:

- ✓ Dùng **abstract** để khai báo lớp trừu tượng và phương thức trừu tượng.

✓ Không thể tạo đối tượng từ Abstract Class, nhưng có thể kế thừa và triển khai phương thức trong lớp con.

### 1.5.2 Slide 2: Interface – Giao diện là gì?

#### 🚀 Interface là gì?

◆ **Interface** là một dạng **hợp đồng (contract)** quy định lớp nào triển khai phải thực hiện đầy đủ các phương thức được định nghĩa.

◆ **Tất cả phương thức trong Interface đều là phương thức trừu tượng (mặc định từ**

**Java 7 trở về trước).**

◆ Từ **Java 8+**, **Interface** có thể có **phương thức mặc định (default methods)** và **phương thức tĩnh (static methods)**.

#### 💡 Ví dụ Interface trong Java:

```
interface Animal {  
    void makeSound();} // Mặc định là phương thức trừu tượng  
class Dog implements Animal {  
    public void makeSound() { System.out.println("Gâu gâu!");}}  
public class Main {  
    public static void main(String args[]) {  
        Dog d = new Dog();  
        d.makeSound();}} // Gâu gâu!
```

#### 🚀 Tóm tắt:

✓ Dùng interface để khai báo một giao diện.

✓ Lớp Dog sử dụng implements để triển khai **đầy đủ các phương thức của Interface**.

✓ Một lớp có thể implements nhiều Interface (hỗ trợ đa kế thừa).

### 1.5.3 Slide 3: So sánh Abstract Class và Interface

Tiêu chí	Abstract Class	Interface
Mục đích	Dùng làm nền tảng chung cho các lớp con	Định nghĩa quy tắc chung cho các lớp triển khai
Từ khóa	abstract class	interface

<b>Chứa phương thức</b>	Cả phương thức có thân (default) và không có thân (abstract)	Mặc định chỉ chứa phương thức không có thân (trừ khi có default hoặc static từ Java 8)
<b>Chứa biến</b>	Có thể có biến instance (biến có giá trị)	Chỉ chứa hằng số (biến final static)
<b>Kế thừa</b>	Lớp con kế thừa bằng extends (chỉ được kế thừa 1 lớp)	Lớp triển khai bằng implements (có thể implements nhiều interface)
<b>Tạo đối tượng</b>	Không thể tạo trực tiếp	Không thể tạo trực tiếp

### ✈️ *Khi nào dùng Abstract Class & Interface?*

- ✓ **Dùng Abstract Class** khi có các **phương thức dùng chung** mà không muốn lớp con viết lại.
- ✓ **Dùng Interface** khi muốn **định nghĩa hành vi chung** mà **nhiều lớp** có thể thực hiện được.
- ✈️ **Hiểu rõ sự khác biệt** giúp bạn **thiết kế chương trình** một cách **tối ưu và dễ mở rộng!**

## 2 CÁC TỪ KHÓA QUAN TRỌNG TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

### 2.1 Từ khóa static và final – Ứng dụng trong OOP

#### 2.1.1 Slide 1: Từ khóa static – Ứng dụng trong OOP

##### ✈️ *Từ khóa static là gì?*

- ⬢ static được sử dụng để khai báo **biến, phương thức hoặc khối lệnh** thuộc về lớp, không phải thuộc về đối tượng.
- ⬢ Khi một thành phần là static, nó được **chia sẻ bởi tất cả các đối tượng** của lớp đó.

##### 💡 **Ứng dụng của static trong OOP:**

##### ✓❑ 1. *Biến tĩnh (static variable) – Dùng chung cho tất cả đối tượng*

```
class Student {  
    static String school = "Đại học ABC"; // Biến static  
    String name;  
    Student(String n) { name = n; }  
    void display() { System.out.println(name + " - " + school);  
}  
}  
  
public class Main {  
    public static void main(String args[]) { Student s1 = new  
        Student("Minh"); Student s2 = new Student("Lan");  
        s1.display(); // Minh - Đại học ABC  
        s2.display(); } // Lan - Đại học ABC  
}
```

💡 **Không cần tạo đối tượng vẫn có thể truy cập biến static:**

```
System.out.println(Student.school);
```

✓□ 2. *Phương thức tĩnh (static method) – Có thể gọi mà không cần tạo đối tượng*

```
class Utility {  
    static void showMessage() {  
        System.out.println("Xin chào từ phương thức static!");  
    }  
  
    public class Main {  
        public static void main(String[] args) {  
            Utility.showMessage();  
            // Gọi phương thức static mà không cần tạo đối tượng  
        }  
    }  
}
```

✈ **Tóm tắt:**

- ✓ static giúp tạo biến và phương thức dùng chung cho tất cả đối tượng.
- ✓ Có thể truy cập static mà không cần tạo đối tượng  
(ClassName.staticMember).

## 2.1.2 Slide 2: Từ khóa final – Ứng dụng trong OOP

✈ **Từ khóa final là gì?**

- ⇒ final được sử dụng để **định nghĩa giá trị không thể thay đổi** hoặc **ngăn chặn ghi đè, kế thừa**.
- ⇒ Có thể áp dụng cho **biến, phương thức, và lớp**.

💡 **Ứng dụng của final trong OOP:**

✓□ 1. *Biến final – Định nghĩa hằng số*

```
class Config {  
    final static double PI = 3.14159; } // Hằng số không thể thay đổi  
  
    public class Main {  
        public static void main(String[] args) {
```



```
        System.out.println(Config.PI); } // 3.14159
    }
```

💡 **Lưu ý:** Biến `final` phải được khởi tạo ngay khi khai báo hoặc trong constructor.

### ✓❏ 2. Phương thức *final* – Ngăn ghi đè (override)

```
class Parent {
    final void show() {
        System.out.println("Phương thức không thể bị ghi
        đè!"); }
}

class Child extends Parent {
    // void show() { System.out.println("Lỗi biên dịch!"); }
    // Không thể ghi đè
}
```

### ✓❏ 3. Lớp *final* – Ngăn kế thừa (inheritance)

```
final class Vehicle { }

// class Car extends Vehicle { } // Lỗi biên dịch: Không thể kế
// thừa lớp final
```

#### ✈️ Tóm tắt:

- ✓ `final` dùng để ngăn chặn thay đổi giá trị, ghi đè phương thức và kế thừa lớp.
- ✓ Dùng `final` cho biến để tạo hằng số, giúp mã nguồn an toàn và dễ bảo trì hơn.
- ✈️ Hiểu rõ `static` và `final` giúp tối ưu hóa bộ nhớ và kiểm soát kế thừa trong OOP!

## 2.2 Từ khóa *abstract* và *interface* – So sánh, khi nào dùng cái nào

### 2.2.1 Slide 1: Từ khóa *abstract* & *interface* – Định nghĩa và cách sử dụng

#### ✈️ Từ khóa *abstract* là gì?

◆ Dùng để khai báo lớp trừu tượng (**abstract class**) hoặc phương thức trừu tượng

(**abstract method**).

◆ Lớp trừu tượng có thể chứa cả phương thức có thân và phương thức trừu tượng

(chưa có phần thân).

✦ **Không thể tạo đối tượng từ lớp abstract.**

💡 *Ví dụ về **abstract class** trong Java:*

```
abstract class Animal {  
    abstract void makeSound(); // Phương thức trtượng (phải được ghi đè)  
    void sleep() { // Phương thức có phần thân  
        System.out.println("Đang ngủ...");  
    }  
}  
class Dog extends Animal {  
    void makeSound() { System.out.println("Gâu gâu!"); }  
}  
public class Main {  
    public static void main(String args[]) {  
        Dog d = new Dog();  
        d.makeSound(); // Gâu gâu!  
        d.sleep(); // Đang ngủ...  
    }  
}
```

✦ ***Khi nào dùng **abstract**?***

- ✓ Khi cần tạo một lớp **mô tả chung** nhưng vẫn cho phép các lớp con mở rộng và triển khai chi tiết.
- ✓ Khi muốn một số phương thức phải được ghi đè (**abstract method**), nhưng vẫn có thể có phương thức thông thường.

## 2.2.2 Slide 2: Interface – Định nghĩa, So sánh với **abstract class** & Khi nào dùng?

✦ ***Interface là gì?***

- ✦ Chỉ chứa phương thức trừu tượng (mặc định từ Java 7 trở về trước).
- ✦ Từ Java 8+, có thể chứa phương thức mặc định (**default method**) và phương thức tĩnh (**static method**).
- ✦ Một lớp có thể implements nhiều interface (hỗ trợ đa kế thừa).

💡 *Ví dụ về **Interface** trong Java:*

```
interface Animal {
```

```

    void makeSound();} // Mặc định là phương thức trừu tượng
class Dog implements Animal {
    public void makeSound() {
        System.out.println("Gâu gâu!");}}
public class Main {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.makeSound();} // Gâu gâu!
}

```

### ✦ So sánh abstract class và interface

Tiêu chí	Abstract Class	Interface
Mục đích	Làm nền tảng cho các lớp con	Định nghĩa quy tắc chung cho nhiều lớp
Từ khóa	abstract class	interface
Chứa phương thức	Cả phương thức có thân (default) và phương thức trừu tượng	Mặc định chỉ chứa phương thức không có thân (từ Java 8 có thể có default method)
Chứa biến	Có thể có biến instance	Chỉ chứa hằng số (final static)
Kế thừa	extends (chỉ kế thừa 1 lớp)	implements (có thể triển khai nhiều interface)
Tạo đối tượng	Không thể tạo trực tiếp	Không thể tạo trực tiếp

### ✦ Khi nào dùng Interface & Abstract Class?

✓ Dùng **Abstract Class** khi có các **phương thức dùng chung** mà không muốn lớp con viết lại.

✓ Dùng **Interface** khi muốn **định nghĩa hành vi chung** mà **nhiều lớp** có thể thực hiện được.

✚ Hiểu rõ sự khác biệt giúp bạn thiết kế chương trình một cách tối ưu và dễ mở rộng!

## 2.3 Từ khóa this & super – Tóm tắt cách dùng

ngắn gọn

✦ Từ khóa **this** – Tham chiếu đến đối tượng hiện tại

✓□ Dùng để phân biệt biến instance và biến local (tránh xung đột tên):

```
class Student { String name;
    Student(String name) {
        this.name = name; // 'this.name' là biến instance,
        'name' là tham số
    }
}
```

✓□ Dùng để gọi constructor khác trong cùng lớp:

```
class Student {
    Student() {
        this("Không tên", 18); } // Gọi constructor có tham số
    Student(String name, int age) { /* Gán giá trị */ }
}
```

✓□ Dùng để gọi phương thức trong cùng lớp:

```
class Student {
    void show() {
        System.out.println("Hello!"); }
    void display() {
        this.show(); } // Gọi phương thức show()
}
```

✦ Từ khóa **super** – Tham chiếu đến lớp cha (superclass)

✓□ Dùng để gọi constructor của lớp cha:

```
class Person {
    Person() { System.out.println("Lớp cha"); }
}

class Student extends Person { Student() {
```

```
        super();} // Gọi constructor Lớp cha  
    }
```

✓□ Dùng để gọi phương thức của lớp cha (khi bị ghi đè - override):

```
class Person {  
    void show() { System.out.println("Lớp cha"); }  
}  
class Student extends Person { void show() {  
    super.show();} // Gọi phương thức Lớp cha  
}
```

✓□ Dùng để truy xuất biến của lớp cha:

```
class Person {  
    String name = "Người cha";  
}  
class Student extends Person {  
    String name = "Người con";  
    void showName() {  
        System.out.println(super.name);} // Lấy giá trị từ Lớp cha  
}
```

✦ Tóm tắt:

✓ **this** → Trỏ đến đối tượng hiện tại, dùng để gọi biến, phương thức, constructor trong cùng lớp.

✓ **super** → Trỏ đến lớp cha, dùng để gọi constructor, phương thức, biến từ lớp cha.

✦ Hiểu rõ **this** và **super** giúp tận dụng tốt tính kế thừa và tổ chức code gọn gàng hơn!

## 2.4 Từ khóa extends & implements – Minh họa bằng code ngắn gọn

### 🚀 Từ khóa extends – Kế thừa từ một lớp cha

⇒ Dùng để tạo một lớp con kế thừa thuộc tính và phương thức từ lớp cha.

⇒ Một lớp chỉ có thể extends từ một lớp cha duy nhất (đơn kế thừa).

💡 Ví dụ extends trong Java:

```
class Animal {  
    void makeSound() { System.out.println("Động vật phát ra âm thanh"); }  
}  
class Dog extends Animal { // Lớp Dog kế thừa từ Animal  
    void bark() { System.out.println("Gâu gâu!"); }  
}  
public class Main {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        d.makeSound(); // Gọi phương thức từ Lớp cha  
        d.bark(); // Gọi phương thức của Lớp con  
    }  
}
```

✓ Lớp Dog có thể sử dụng cả phương thức makeSound() của Animal và phương thức riêng bark().

### 🚀 Từ khóa implements – Triển khai từ một hoặc nhiều Interface

⇒ Dùng khi một lớp muốn triển khai một hoặc nhiều interface.

⇒ Hỗ trợ đa kế thừa thông qua interface.

💡 Ví dụ implements trong Java:

```
interface Animal {  
    void makeSound(); // Mặc định là phương thức trừu tượng  
}  
class Dog implements Animal { // Lớp Dog triển khai Interface Animal  
    public void makeSound() {  
        System.out.println("Gâu gâu!");  
    }  
}
```

```

}
public class Main {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.makeSound();} // Gâu gâu!
    }
}

```

✓ Lớp Dog bắt buộc phải triển khai tất cả phương thức của Interface Animal.

### ✦ Tóm tắt:

✓ **extends** → Dùng để kế thừa một lớp cha, giúp tái sử dụng mã nguồn.

✓ **implements** → Dùng để triển khai một hoặc nhiều interface, hỗ trợ đa kế thừa.

✈ Hiểu rõ extends và implements giúp thiết kế hệ thống linh hoạt, dễ mở rộng!

## 2.5 Tổng hợp & So sánh nhanh – Đưa ra bảng tóm tắt thay vì trình bày từng từ khóa riêng lẻ

### 2.5.1 Slide 1: Tổng hợp các từ khóa quan trọng trong lập

trình hướng đối tượng

#### ✦ Các từ khóa quan trọng trong Java OOP:

✦ Quản lý đối tượng và kế thừa: this, super, extends, implements

✦ Trừu tượng hóa và giao diện: abstract, interface

✦ Phạm vi truy cập và điều khiển hành vi: static, final

💡 Những từ khóa này giúp tổ chức mã nguồn tốt hơn, tận dụng tối đa tính hướng đối tượng trong Java.

📊 Hãy cùng so sánh các từ khóa chính!

### 2.5.2 Slide 2: Bảng so sánh nhanh các từ khóa OOP trong Java

Từ khóa	Chức năng chính	Ví dụ sử dụng	Ghi chú
this	Tham chiếu đến đối tượng hiện tại	this.name = name;	Dùng trong constructor, gọi phương thức nội bộ

Từ khóa	Chức năng chính	Ví dụ sử dụng	Ghi chú
<b>super</b>	Tham chiếu đến lớp cha	<code>super.show();</code>	Gọi constructor hoặc phương thức của lớp cha
<b>extends</b>	Kế thừa một lớp cha	<code>class Dog extends Animal {}</code>	Chỉ có thể kế thừa một lớp cha (đơn kế thừa)
<b>implements</b>	Triển khai một hoặc nhiều interface	<code>class Dog implements Animal {}</code>	Hỗ trợ đa kế thừa bằng interface
<b>abstract</b>	Định nghĩa lớp/phương thức trừu tượng	<code>abstract class Animal {}</code>	Không thể tạo đối tượng trực tiếp từ abstract class
<b>interface</b>	Định nghĩa một giao diện (interface)	<code>interface Animal { void makeSound(); }</code>	Chỉ chứa phương thức trừu tượng (Java 7 trở xuống)
<b>static</b>	Thành phần tĩnh, dùng chung	<code>static int count;</code>	Có thể truy cập mà không cần tạo đối tượng
<b>final</b>	Ngăn thay đổi (hằng số, phương thức, lớp)	<code>final int MAX = 100;</code>	Không thể ghi đè hoặc kế thừa nếu dùng cho phương thức/lớp

### ✦ Tóm tắt:

✓ Hiểu rõ các từ khóa này giúp lập trình hiệu quả hơn, tận dụng tối đa sức mạnh của Java OOP!

🔗 Nắm vững bảng so sánh này giúp bạn dễ dàng lựa chọn từ khóa phù hợp trong từng tình huống.

## 3 PHẠM VI TRUY CẬP

### 3.1 Các mức độ phạm vi truy cập (public, private, protected, default) – Bảng so sánh chi tiết

#### 3.1.1 Slide 1: Giới thiệu về các mức độ phạm vi truy cập

trong Java

#### ✦ Phạm vi truy cập là gì?

⚙️ **Phạm vi truy cập (Access Modifiers)** xác định mức độ hiển thị của biến, phương thức và lớp trong Java.



### ◆ Có 4 mức độ phạm vi chính:

- ✓ ☐ `public` – Truy cập từ mọi nơi.
- ✓ ☐ `private` – Chỉ truy cập được trong cùng một lớp.
- ✓ ☐ `protected` – Truy cập trong cùng package hoặc từ lớp con.
- ✓ ☐ (Mặc định – Default) – Truy cập trong cùng package (nếu không khai báo gì).

### ✦ Tại sao cần phạm vi truy cập?

- ✓ Giúp bảo mật dữ liệu, giảm rủi ro sửa đổi ngoài ý muốn.
- ✓ Kiểm soát quyền truy cập trong kế thừa và đóng gói.

📊 Chúng ta sẽ cùng so sánh chi tiết các phạm vi truy cập!

#### 3.1.2 Slide 2: Bảng so sánh chi tiết các mức độ phạm vi truy cập

Phạm vi truy cập	Trong cùng lớp	Trong cùng package	Lớp con (khác package)	Bên ngoài package
<code>public</code>	✓ Có thể truy cập	✓ Có thể truy cập	✓ Có thể truy cập	✓ Có thể truy cập
<code>private</code>	✓ Có thể truy cập	✗ Không thể truy cập	✗ Không thể truy cập	✗ Không thể truy cập
<code>protected</code>	✓ Có thể truy cập	✓ Có thể truy cập	✓ Có thể truy cập	✗ Không thể truy cập
(Default – Không khai báo)	✓ Có thể truy cập	✓ Có thể truy cập	✗ Không thể truy cập	✗ Không thể truy cập

### 💡 Ví dụ minh họa trong Java:

```
class Example {  
    public int a = 10; // Truy cập từ mọi nơi  
    private int b = 20; // Chỉ truy cập trong class này  
    protected int c = 30; // Truy cập trong cùng package + Lớp con  
    int d = 40; // (Mặc định) Truy cập trong cùng package
```

}

### ✦ Tóm tắt:

- ✓ Dùng **public** khi muốn cho phép truy cập từ bất kỳ đâu.
- ✓ Dùng **private** để bảo vệ dữ liệu quan trọng.
- ✓ Dùng **protected** khi muốn cho phép lớp con kế thừa nhưng vẫn giới hạn phạm vi ngoài package.
- ✓ Dùng phạm vi mặc định (default) nếu chỉ muốn truy cập trong cùng package.
- 🔗 Hiểu rõ các mức độ truy cập giúp bạn thiết kế hệ thống an toàn, dễ bảo trì hơn!

## 3.2 Ứng dụng phạm vi truy cập trong kế thừa – Code minh họa

### 3.2.1 Slide 1: Ứng dụng phạm vi truy cập trong kế thừa

#### ✦ Phạm vi truy cập ảnh hưởng như thế nào đến kế thừa?

⚙ Trong kế thừa, lớp con có thể truy cập các thành viên của lớp cha dựa vào phạm vi truy cập.

#### ⚙ Tóm tắt phạm vi truy cập trong kế thừa:

- ✓□ **public** – Lớp con có thể truy cập.
- ✓□ **protected** – Lớp con có thể truy cập, nhưng không cho phép bên ngoài package (trừ khi kế thừa).
- ✓□ **private** – Lớp con **không thể truy cập trực tiếp**, chỉ có thể thông qua phương thức **public** hoặc **protected**.
- ✓□ (*Default – Không khai báo*) – Chỉ cho phép truy cập trong cùng package.

#### 💡 Ví dụ tổng quan:

```
class Parent {  
    public int a = 10;  
    private int b = 20;  
    protected int c = 30;  
    int d = 40; } // Default  
  
class Child extends Parent { void showValues() {  
    System.out.println(a); // ✓ Được phép (public)  
    // System.out.println(b); // ✗ Không được phép (private)
```

```

        System.out.println(c); // ✓ Được phép (protected)
        System.out.println(d); } // ✓ Được phép (default, nhưng
        chỉ nếu cùng package)
    }

```

☞ **Lớp con có thể truy cập public, protected, và default (nếu cùng package), nhưng không thể truy cập private.**

### 3.2.2 Slide 2: Code minh họa kế thừa và phạm vi truy cập

✦ **Ví dụ thực tế: Phạm vi truy cập trong kế thừa**

☞ **Lớp cha (Animal) có các biến với các mức truy cập khác nhau.**

☞ **Lớp con (Dog) kế thừa Animal và thử truy cập các biến.**

```

class Animal {
    public String name = "Động vật"; private int age = 5;
    protected String type = "Thú cưng"; String sound = "Âm
    thanh";}

class Dog extends Animal {
    void showInfo() {
        System.out.println("Tên: " + name); // ✓ Có thể truy
        cập (public)
        // System.out.println("Tuổi: " + age); // ✗ Lỗi!
        (private)
        System.out.println("Loại: " + type); // ✓ Có thể truy
        cập (protected)
        System.out.println("Âm thanh: " + sound); } // ✓ Có thể
        truy cập (default, nếu cùng package)
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.showInfo();}
}

```

✦ **Tóm tắt:**

✓ **Lớp con kế thừa public và protected** – Có thể truy cập.

✓ **private không thể truy cập trực tiếp** – Cần phương thức public trong lớp cha để

truy xuất.

✓ **default chỉ truy cập được nếu cùng package** – Hữu ích khi tổ chức mã theo module.

🔪 **Kiểm soát phạm vi truy cập trong kế thừa giúp bảo vệ dữ liệu và thiết kế hệ thống an toàn hơn!**

### 3.3 Tầm quan trọng của phạm vi truy cập trong bảo mật dữ liệu – Ví dụ thực tế

#### 3.3.1 Slide 1: Tầm quan trọng của phạm vi truy cập trong

bảo mật dữ liệu

✦ **Tại sao phạm vi truy cập quan trọng trong bảo mật dữ liệu?**

⚡ Giúp **bảo vệ thông tin quan trọng** khỏi bị thay đổi ngoài ý muốn.

⚡ Ngăn **truy cập trái phép** vào các biến và phương thức nhạy cảm.

⚡ Kiểm soát quyền truy cập trong **kế thừa và đóng gói dữ liệu**.

💡 **Ví dụ sai phạm khi không kiểm soát truy**

```
class BankAccount {  
    public double balance = 1000.0; // ✗ Không an toàn! Ai cũng  
        có thể thay đổi trực tiếp  
    void deposit(double amount) { balance += amount; }  
}  
  
public class Main {  
    public static void main(String[] args) { BankAccount acc =  
        new BankAccount();  
        acc.balance = -5000; // ✗ Lỗi hỏng bảo mật - Truy cập và  
            thay đổi trái phép!  
        System.out.println("Số dff: " + acc.balance); // -5000  
        (Không hợp Lệ!)  
    }  
}
```

📌 **Vấn đề:**

✓ Số dư có thể bị thay đổi tùy ý, gây sai lệch dữ liệu.

✓ Không có bất kỳ kiểm tra bảo mật nào khi thay đổi balance.

✦ **Giải pháp:** Sử dụng **private** để ẩn dữ liệu và cung cấp phương thức kiểm soát.

### 3.3.2 Slide 2: Cách sử dụng phạm vi truy cập để bảo mật dữ liệu

✦ **Cải thiện bảo mật bằng phạm vi truy cập hợp lý**

⇒ Dùng **private** để bảo vệ dữ liệu nhạy cảm.

⇒ Cung cấp **phương thức public hoặc protected** để truy cập dữ liệu một cách an toàn.

⇒ Kiểm tra tính hợp lệ của dữ liệu trước khi thay đổi.

💡 **Ví dụ cải tiến với private và getter/setter**

```
class BankAccount {  
    private double balance = 1000.0; // 🛡️ Bảo vệ dữ liệu  
    public double getBalance() { // ✓ Chỉ cho phép đọc dữ liệu  
        return balance;  
    }  
    public void deposit(double amount) { // ✓ Kiểm tra hợp lệ  
        // trước khi thay đổi dữ liệu  
        if (amount > 0) {  
            balance += amount;  
        } else {  
            System.out.println("Số tiền gửi phải lớn hơn 0!");  
        }  
    }  
}  
public class Main {  
    public static void main(String[] args) { BankAccount acc =  
        new BankAccount();  
        // acc.balance = -5000; // ✗ Lỗi! Không thể thay đổi  
        // trực tiếp  
        acc.deposit(500); // ✓ Hợp lệ  
        System.out.println("Số dff: " + acc.getBalance()); } //  
        1500.0  
}
```

✦ **Tóm tắt:**

- ✓ Dùng **private** để bảo vệ dữ liệu, tránh truy cập trái phép.
- ✓ Cung cấp **getter/setter** để kiểm soát truy cập, đảm bảo dữ liệu hợp lệ.
- ✓ Kiểm tra giá trị nhập vào trước khi thay đổi dữ liệu quan trọng.
- ✈️ Hiểu rõ phạm vi truy cập giúp bảo vệ dữ liệu an toàn và tránh sai sót trong lập trình!

### 3.4 Bài tập thực hành trên lớp – Code & giải thích

#### 3.4.1 Slide 1: Bài tập thực hành – Kiểm soát phạm vi truy cập trong OOP

##### ✈️ Mục tiêu bài tập:

- ◆ Áp dụng phạm vi truy cập (**private**, **protected**, **public**, **default**).
- ◆ Bảo vệ dữ liệu bằng **getter** và **setter**.
- ◆ Hiểu cách sử dụng **this** và **super** trong kế thừa.

##### 💡 Đề bài:

##### 1. Viết lớp **BankAccount** có:

Biến **private double balance** (số dư tài khoản).

Phương thức **public double getBalance()** để lấy số dư.

Phương thức **public void deposit(double amount)** để nạp tiền.

Phương thức **public void withdraw(double amount)** để rút tiền (chỉ cho phép nếu số dư đủ).

##### 2. Viết lớp **SavingAccount** kế thừa **BankAccount**, thêm: Lãi suất (**protected double interestRate**).

Phương thức **public void addInterest()** để cộng lãi suất vào số dư. 3□□ Viết chương trình **Main** để kiểm tra hoạt động.

#### 3.4.2 Slide 2: Gợi ý giải bài tập – Khai báo lớp **BankAccount**

```
class BankAccount { private double balance;
    public BankAccount(double initialBalance) { // Constructor
        this.balance = initialBalance; }
    public double getBalance() { // Getter
        return balance; }
    public void deposit(double amount) { // Nạp tiền
```

```

        if (amount > 0) { balance += amount;
            System.out.println("Nạp " + amount + ". Số dff mới: "
+ balance);
        } else {
            System.out.println("Số tiền nạp phải lớn hơn 0!"); } }
public void withdraw(double amount) { // Rút tiền
    if (amount > 0 && balance >= amount) { balance -=
        amount;
        System.out.println("Rút " + amount + ". Số dff còn
        lại: " + balance);
    } else {
        System.out.println("Rút tiền thất bại! Số dff không
        đủ."); } }
}

```

#### ✦ Giải thích:

- ✓ Dùng **private balance** để bảo vệ dữ liệu.
- ✓ Dùng **getter (getBalance())** để lấy số dư thay vì truy cập trực tiếp.
- ✓ Kiểm tra hợp lệ trước khi nạp/rút tiền.

### 3.4.3 Slide 3: Gọi ý giải bài tập – Kế thừa với

#### SavingAccount

```

class SavingAccount extends BankAccount {
    protected double interestRate;
    public SavingAccount(double initialBalance, double
interestRate) {
        super(initialBalance); // Gọi constructor lớp cha
        this.interestRate = interestRate; }
    public void addInterest() { // Cộng lãi suất vào số dư
        double interest = getBalance() * interestRate / 100;
        deposit(interest); // Gọi phương thức từ lớp cha
        System.out.println("Thêm lãi suất: " + interest); }
}

```

#### ✦ Giải thích:

- ✓ Dùng `super(initialBalance)` để gọi constructor của lớp cha.
- ✓ Dùng `protected interestRate` để lớp con có thể truy cập mà không bị lộ ra ngoài.
- ✓ Phương thức `addInterest()` tính lãi và thêm vào số dư.

### 3.4.4 Slide 4: Chạy chương trình & Kiểm tra kết quả

```
public class Main {  
    public static void main(String[] args) {  
        SavingAccount myAccount = new SavingAccount(1000, 5);  
        System.out.println("Số dff ban đầu: " +  
            myAccount.getBalance());  
        myAccount.deposit(500); myAccount.withdraw(200);  
        myAccount.addInterest();  
        System.out.println("Số dff cuối cùng: " +  
            myAccount.getBalance());  
    }  
}
```

#### ✦ Kết quả mong đợi:

Số dff ban đầu: 1000.0

Nạp 500.0. Số dff mới: 1500.0

Rút 200.0. Số dff còn lại: 1300.0 Thêm lãi suất: 65.0

Số dff cuối cùng: 1365.0

#### 🔗 Bài học rút ra:

- ✓ Dùng `private` để bảo vệ dữ liệu, `protected` để cho phép kế thừa.
- ✓ Dùng `super()` để gọi constructor của lớp cha.
- ✓ Dùng phương thức thay vì truy cập biến trực tiếp giúp kiểm soát dữ liệu tốt hơn.

## 3.5 Tình huống thực tế: Thiết kế một hệ thống quản lý đơn giản bằng OOP

### 3.5.1 Slide 1: Tình huống thực tế – Thiết kế hệ thống quản lý sinh viên bằng OOP

#### ✦ Bài toán thực tế:



◆ Một trường đại học cần **hệ thống quản lý sinh viên** để theo dõi thông tin sinh viên, khóa học và điểm số.

◆ Hệ thống cần có các chức năng:

✓□ Thêm sinh viên, đăng ký khóa học.

✓□ Quản lý điểm số, tính GPA.

✓□ Hiển thị thông tin sinh viên.

💡 **Thiết kế hệ thống bằng OOP:**

◆ **Lớp Student** – Quản lý thông tin sinh viên.

◆ **Lớp Course** – Quản lý khóa học.

◆ **Lớp Enrollment** – Liên kết sinh viên với khóa học & điểm số.

✦ **Mô hình quan hệ giữa các lớp:**

Student **1** -- *n* Enrollment *n* -- **1** Course

✓ **Một sinh viên có thể đăng ký nhiều khóa học.**

✓ **Mỗi khóa học có nhiều sinh viên đăng ký.**

### 3.5.2 Slide 2: Cài đặt hệ thống quản lý sinh viên bằng Java

💡 **Bước 1: Lớp Student – Lưu trữ thông tin sinh viên**

```
class Student {  
    private String id; private String name;  
    public Student(String id, String name) {  
        this.id = id;  
        this.name = name; }  
    public String getId() { return id; }  
    public String getName() { return name; }  
    public void display() {  
        System.out.println("Sinh viên: " + id + " - " + name); }  
}
```

💡 **Bước 2: Lớp Course – Quản lý khóa học**

```
class Course {
```

```
private String courseId; private String courseName;
public Course(String courseId, String courseName) {
    this.courseId = courseId;
    this.courseName = courseName; }
public String getCourseId() { return courseId; }
public String getCourseName() { return courseName; }
}
```

### 💡 Bước 3: Lớp Enrollment – Liên kết sinh viên và khóa học

```
class Enrollment {
    private Student student;
    private Course course;
    private double grade;
    public Enrollment(Student student, Course course, double
grade) {
        this.student = student;
        this.course = course;
    }
}
```

```
        this.grade = grade; }  
    public void display() {  
        System.out.println(student.getName() + " đăng ký " +  
            course.getCourseName() + " - Điểm: " + grade); }  
    }  
}
```

#### 💡 Bước 4: Chạy chương trình

```
public class Main {  
    public static void main(String[] args) {  
        Student s1 = new Student("SV001", "Nguyễn Văn A");  
        Course c1 = new Course("CS101", "Lập trình Java");  
        Enrollment e1 = new Enrollment(s1, c1, 9.0);  
        e1.display();  
    }  
}
```

#### ✈️ Kết quả mong đợi:

Nguyễn Văn A đăng ký Lập trình Java - Điểm: 9.0

#### 🔗 Bài học rút ra:

✓ Sử dụng hướng đối tượng giúp tổ chức dữ liệu rõ ràng, dễ mở rộng.

- ✓ Dễ dàng thêm chức năng như tính GPA, danh sách sinh viên của khóa học.
- ✓ Tối ưu hệ thống bằng kế thừa và giao diện (ví dụ: `interface Manageable`).
- 💡 Bạn có muốn mở rộng bài tập này với danh sách sinh viên (`ArrayList<Student>`) không? 😊

## 3.6 Giao bài tập về nhà – Đề bài và hướng dẫn sinh viên tự thực hành

### 3.6.1 Slide 1: Giao bài tập về nhà – Đề bài

#### ✦ Mô tả bài tập:

- ✦ Xây dựng một **hệ thống quản lý thư viện** bằng OOP trong Java.
- ✦ Hệ thống cần có các chức năng sau:
  - ✓ ☐ **Quản lý sách (Book)** – Lưu trữ thông tin sách (ID, tên, tác giả).
  - ✓ ☐ **Quản lý thành viên (Member)** – Lưu thông tin thành viên (ID, họ tên).
  - ✓ ☐ **Quản lý mượn sách (Loan)** – Thành viên mượn/trả sách.

#### 💡 Yêu cầu cài đặt:

- Viết lớp `Book` chứa thông tin sách.
- Viết lớp `Member` chứa thông tin thành viên.
- Viết lớp `Loan` liên kết `Book` và `Member`, chứa thông tin ngày mượn, ngày trả. 4 ☐ ☐  
Viết chương trình `Main` để kiểm tra hệ thống.

👉 Sinh viên cần áp dụng: **private, getter/setter, ArrayList, this, super**

### 3.6.2 Slide 2: Hướng dẫn thực hiện bài tập

#### ✦ Gợi ý thiết kế hệ thống:

#### ✓ Lớp `Book` – Quản lý thông tin sách

```
class Book {  
    private String bookId;  
    private String title;  
    private String author;  
    public Book(String bookId, String title, String author) {  
        this.bookId = bookId;  
    }  
}
```

```

        this.title = title;
        this.author = author;
    }
    public String getBookId() { return bookId; }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
}

```

### ✓ Lớp Member – Quản lý thông tin thành viên

```

class Member {
    private String memberId;
    private String name;
    public Member(String memberId, String name) {
        this.memberId = memberId;
        this.name = name; }
    public String getMemberId() { return memberId; }
    public String getName() { return name; }
}

```

### 3.6.3 Slide 3: Gọi ý tiếp theo & yêu cầu nộp bài

#### ✦ Gọi ý tiếp theo:

### ✓ Lớp Loan – Quản lý thông tin mượn sách

```

import java.util.Date;
class Loan {
    private Book book;
    private Member member;
    private Date borrowDate;
    public Loan(Book book, Member member, Date borrowDate) {

```

```

        this.book = book;
        this.member = member;
        this.borrowDate = borrowDate;
    }

    public void display() {
        System.out.println(member.getName() + " mượn sách " +
            book.getTitle() + " vào ngày " + borrowDate);
    }
}

```

### ✓ Lớp Main để chạy chương trình

```

import java.util.Date;


public class Main {
    public static void main(String[] args) {
        Book book1 = new Book("B001", "Java Programming", "John
Doe");
        Member member1 = new Member("M001", "Nguyễn Văn A");
        Loan loan1 = new Loan(book1, member1, new Date());
        loan1.display();
    }
}

```

**Bài tập về nhà:** Viết lại và học thuộc tất cả các code ví dụ trong buổi học này.

### ✦ Yêu cầu nộp bài:

 **Hạn chót: 25/02/2025**

 **Hình thức nộp:** Nén tất cả các file .java thành đuôi .zip/.rar và nộp lên hệ thống LMS Canvas (Nộp bài Buổi 2).

 **Hoàn thành bài tập này giúp sinh viên hiểu rõ hơn về OOP, kế thừa và phạm vi truy cập trong Java!**