



## CHƯƠNG 2: LỚP (CLASS) & ĐỐI TƯỢNG (OBJECT) – MODERN JAVA APPROACH

Chào mừng bạn đến với hành trình khám phá trái tim của lập trình hướng đối tượng! Trong chương này, chúng ta sẽ cùng nhau từng bước xây dựng nền tảng vững chắc về Class và Object - hai khái niệm cốt lõi giúp bạn tư duy và thiết kế phần mềm theo cách chuyên nghiệp.



# MỤC TIÊU HỌC TẬP

Sau khi hoàn thành chương này, bạn sẽ tự tin làm chủ những kỹ năng then chốt của lập trình Java hiện đại:



## Class & Object Mastery

Hiểu sâu bản chất của Class (khuôn mẫu) và Object (thực thể), biết cách thiết kế và sử dụng chúng hiệu quả



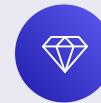
## Cơ chế Bộ nhớ

Nắm vững cách Java quản lý Stack và Heap, hiểu rõ vòng đời của đối tượng từ lúc sinh ra đến khi được thu gom



## Encapsulation Pro

Thực hành đóng gói dữ liệu chuyên nghiệp, sử dụng thành thạo Constructors, Getters/Setters và Access Modifiers



## Modern Java Features

Áp dụng Java Records cho Data Classes và thiết kế Immutable Objects theo chuẩn công nghiệp

# KIẾN THỨC CÂN CÓ (PREREQUISITES)

## ⚠ QUAN TRỌNG

Trước khi bắt đầu cuộc hành trình này, hãy đảm bảo bạn đã trang bị đầy đủ những kiến thức nền tảng sau. Nếu chưa vững, đừng ngại quay lại Chương 1 để ôn tập!

1

### Cú pháp Java cơ bản

#### Chương 1, phần 0.2

- Hiểu cấu trúc class, method, main
- Biết cách compile và chạy chương trình
- Nắm được flow điều khiển cơ bản

2

### Kiểu dữ liệu Java

#### Chương 1, phần 0.3

- Primitive types: int, double, boolean, char...
- Reference types: String, arrays
- Phân biệt được value vs reference

3

### Method cơ bản

#### Chương 1, phần 0.8

- Cách định nghĩa và gọi method
- Tham số và return type
- Static method (đã học cơ bản)

4

### Reference & null

#### Chương 1, phần 0.3, 0.15

- Hiểu reference types hoạt động thế nào
- Biết về null và NullPointerException
- Từ khóa new để tạo object

 **Lưu ý:** Chương này xây dựng trên nền tảng Chương 1. Nếu bạn cảm thấy bất kỳ khái niệm nào còn mơ hồ, hãy dành 15-20 phút ôn tập trước khi tiếp tục. Điều này sẽ giúp bạn học hiệu quả gấp nhiều lần!

## 2.1. CLASS, OBJECT & VÒNG ĐỜI ĐỐI TƯỢNG

### 2.1.1. Class (Lớp) - Khuôn mẫu

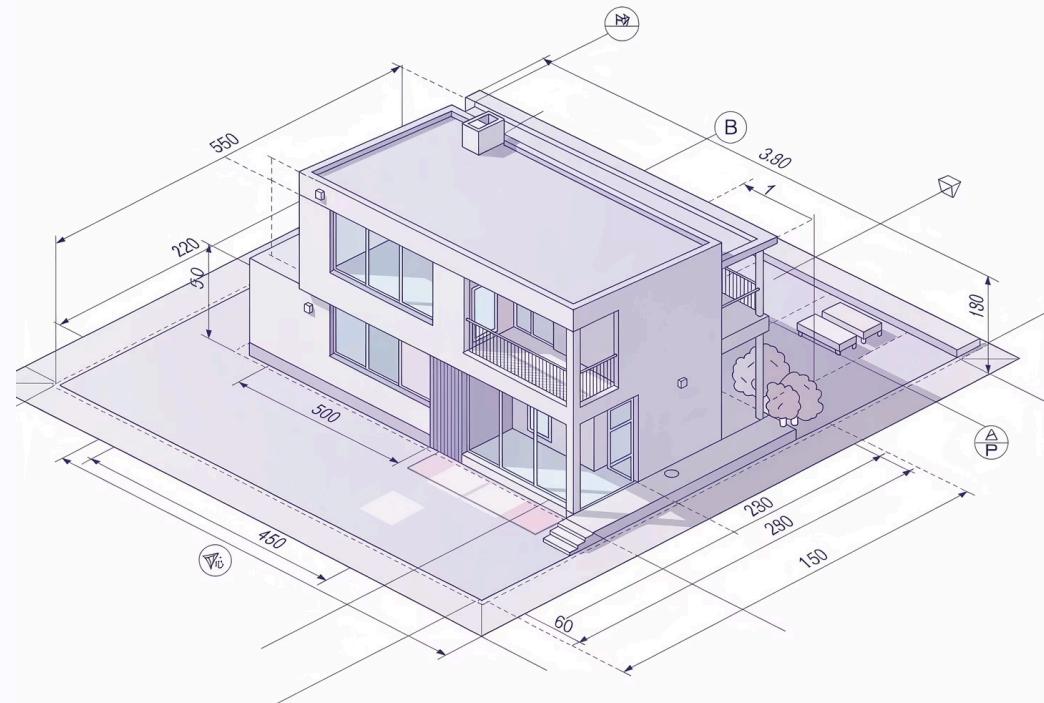
#### Class là gì?

**Class** là bản thiết kế, khuôn mẫu (template) để tạo ra các đối tượng.

Giống như một bản vẽ kiến trúc mô tả chi tiết ngôi nhà, Class định nghĩa:

- **Attributes (Thuộc tính):** Dữ liệu mà đối tượng sẽ lưu trữ
- **Methods (Phương thức):** Hành vi mà đối tượng có thể thực hiện

Class là trừu tượng - nó chỉ mô tả, chưa tồn tại thực sự trong bộ nhớ cho đến khi bạn tạo object từ nó.



## Attributes (Thuộc tính)

Thuộc tính (Attributes), hay còn gọi là **trường (fields)** hoặc **biến thành viên (member variables)**, là các đặc điểm hoặc dữ liệu mà một đối tượng thuộc về lớp đó sẽ sở hữu. Chúng mô tả trạng thái của đối tượng.

Quay lại ví dụ **Class Car**, các thuộc tính của nó có thể bao gồm:

### brand (Thương hiệu)

Kiểu dữ liệu: String

Ví dụ: "Toyota"

### model (Mẫu xe)

Kiểu dữ liệu: String

Ví dụ: "Camry"

### color (Màu sắc)

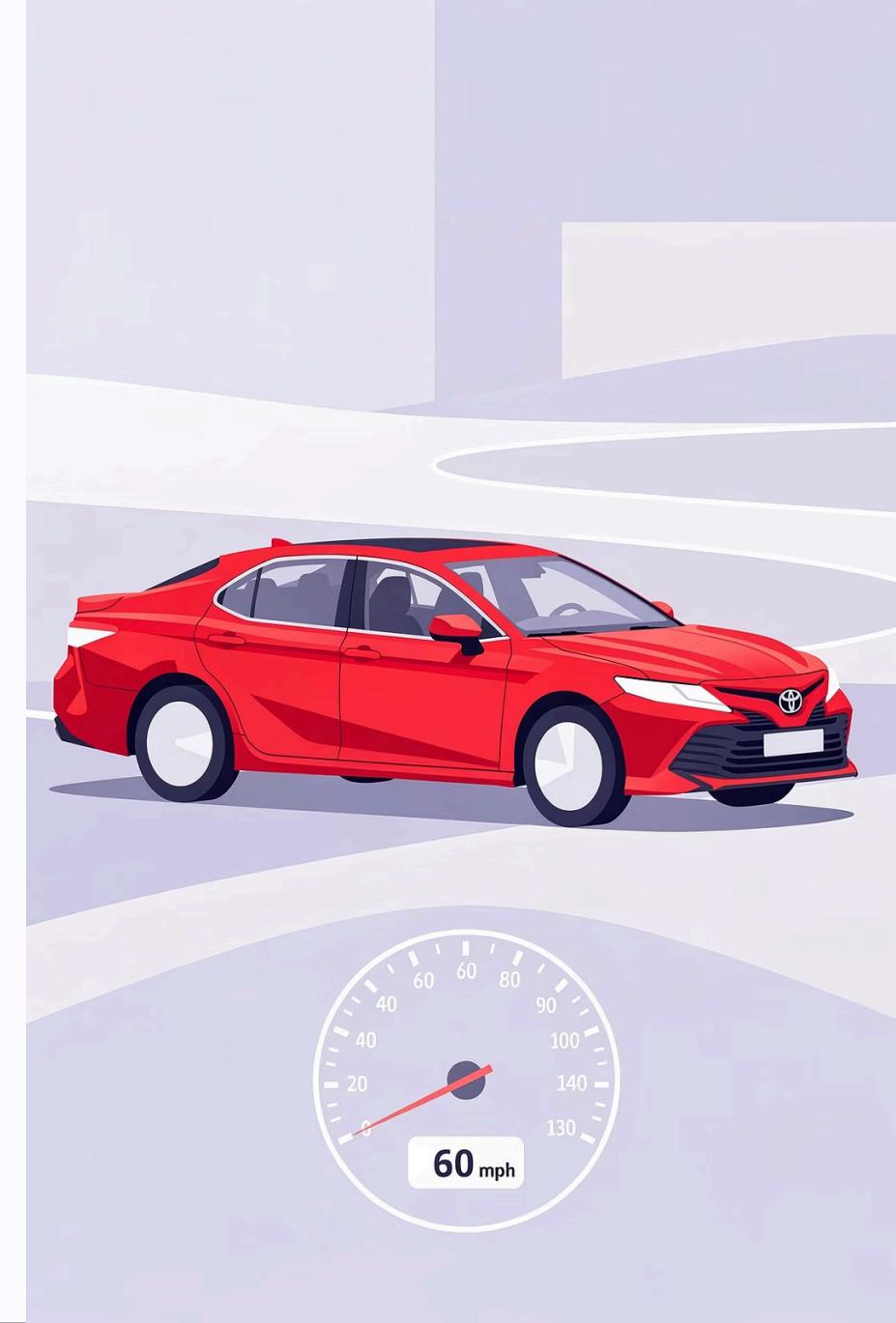
Kiểu dữ liệu: String

Ví dụ: "Đỏ"

### currentSpeed (Tốc độ)

Kiểu dữ liệu: int

Ví dụ: 0



# Methods (Phương thức) - Hành vi của Đối tượng

## Methods là gì?

**Methods (Phương thức)** là các hành động hoặc chức năng mà một đối tượng có thể thực hiện. Chúng định nghĩa hành vi của đối tượng, cho phép đối tượng tương tác với các thuộc tính của nó và thực hiện các logic nghiệp vụ.

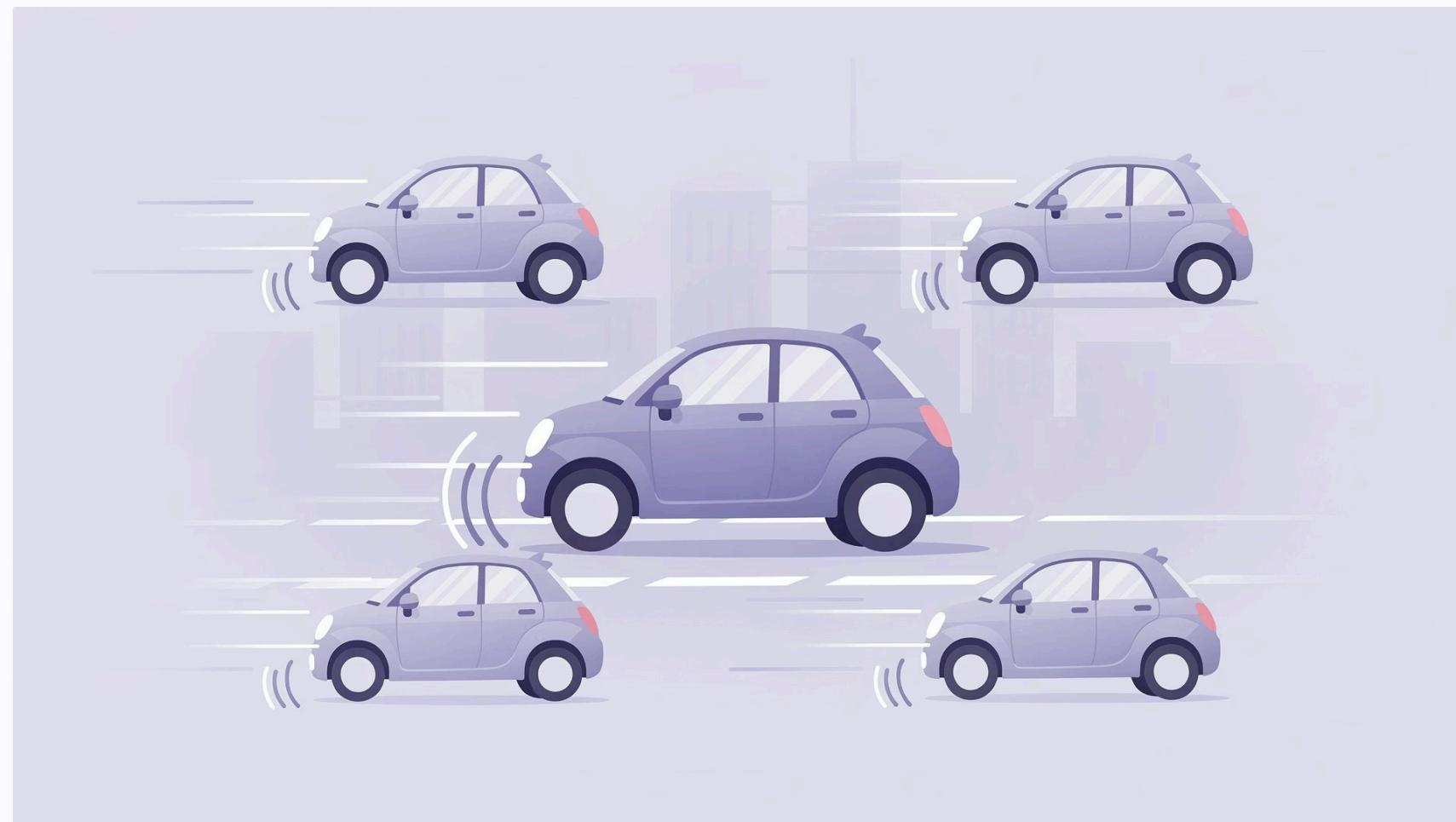
Giống như việc tài xế thực hiện các hành động trên chiếc xe (khởi động, tăng tốc, phanh), các phương thức giúp đối tượng "làm việc" và thay đổi trạng thái của chính nó hoặc tương tác với các đối tượng khác.

## Cấu trúc cơ bản của một Method:

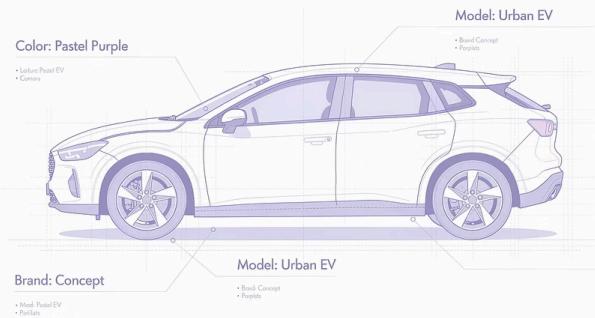
```
<Kiểu trả về> <tên method>(<Danh sách tham số>) {  
    // Thân method: Các câu lệnh thực hiện hành vi  
}
```

## Ví dụ với Class Car:

```
public class Car {  
    String brand;  
    int currentSpeed;  
  
    // Method để khởi động xe  
    public void startEngine() {  
        System.out.println(brand + " engine started!");  
    }  
  
    // Method để tăng tốc  
    public void accelerate(int speedIncrease) {  
        currentSpeed += speedIncrease;  
        System.out.println(brand + " accelerating to " + currentSpeed + " km/h.");  
    }  
}
```



## Ví dụ thực tế: Class Car



Class Car = Bản thiết kế

Định nghĩa các đặc điểm (attributes) và hành vi (methods) chung.



Objects = Các xe cụ thể

Tạo ra nhiều đối tượng xe khác nhau từ một bản thiết kế duy nhất.



Tính độc lập của Objects

Mỗi đối tượng có trạng thái riêng, thay đổi một xe không ảnh hưởng đến xe khác.

# Cú pháp Class trong Java

## Cấu trúc tổng quát

```
[access modifier] class ClassName {  
    // Fields (Thuộc tính)  
    [access modifier] dataType fieldName;  
  
    // Constructor (Hàm khởi tạo)  
    [access modifier] ClassName(parameters) {  
        // Khởi tạo đối tượng  
    }  
  
    // Methods (Phương thức)  
    [access modifier] returnType methodName() {  
        // Thực hiện hành vi  
    }  
}
```

## Ví dụ cụ thể: Class Student

```
public class Student {  
    // Fields - Dữ liệu của sinh viên  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Constructor - Khởi tạo sinh viên  
    public Student(String name, int age, double gpa) {  
        this.name = name;  
        this.age = age;  
        this.gpa = gpa;  
    }  
  
    // Method - Hiển thị thông tin  
    public void displayInfo() {  
        System.out.println("Name: " + name +  
            ", Age: " + age + ", GPA: " + gpa);  
    }  
  
    // Method - Lấy điểm GPA  
    public double getGpa() {  
        return gpa;  
    }  
}
```

## 2.1.2. Object (Đối tượng) - Thực thể

Object là gì?

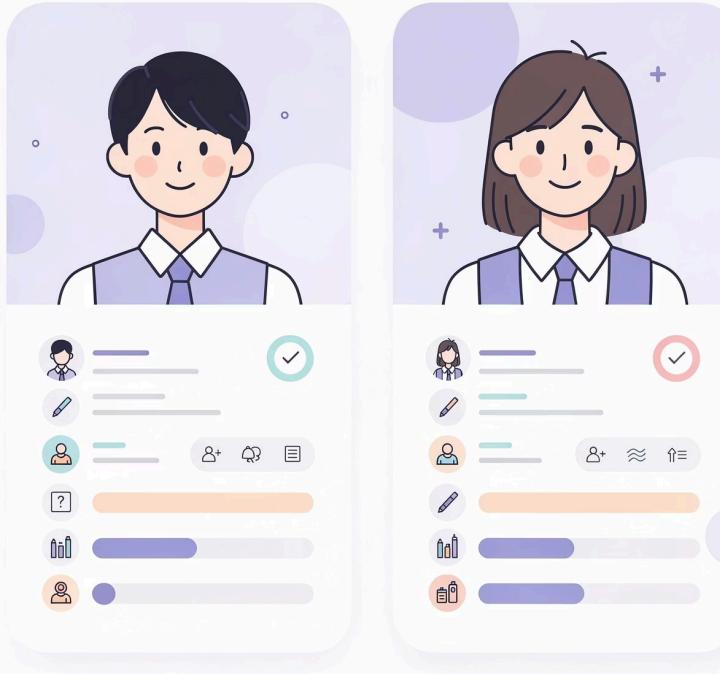
**Object** là một thực thể cụ thể được tạo ra từ class. Nếu Class là bản vẽ, thì Object là sản phẩm thực tế. Mỗi object có đời sống riêng với trạng thái (giá trị fields) và hành vi (methods) độc lập.

### Cú pháp tạo Object

```
ClassName objectName = new ClassName(arguments);
```

#### Ví dụ tạo 2 sinh viên

```
// Tạo sinh viên thứ nhất  
Student student1 = new Student(  
    "Nguyen Van A", 20, 3.5  
);  
  
// Tạo sinh viên thứ hai  
Student student2 = new Student(  
    "Tran Thi B", 21, 3.8  
);  
  
// Mỗi object có dữ liệu riêng  
student1.displayInfo();  
// Name: Nguyen Van A, Age: 20, GPA: 3.5  
  
student2.displayInfo();  
// Name: Tran Thi B, Age: 21, GPA: 3.8
```



#### Phân tích quan trọng:

- student1 và student2 là **2 object hoàn toàn khác nhau** trong bộ nhớ
- Cùng được tạo từ class Student nhưng **lưu trữ dữ liệu khác nhau**
- Gọi cùng method displayInfo() nhưng **kết quả khác nhau** vì state khác nhau
- Thay đổi student1 **không ảnh hưởng** đến student2

## 2.1.3. Cơ chế Cấp phát Bộ nhớ: Stack vs Heap

Java quản lý bộ nhớ trong 2 vùng chính với vai trò rất khác biệt. Hiểu rõ sự khác biệt này sẽ giúp bạn viết code hiệu quả và tránh nhiều lỗi tinh vi.

### Stack - "Danh sách công việc"

Tưởng tượng Stack như một **danh sách việc cần làm** trên bàn làm việc. Bạn ghi việc ra giấy note, làm xong việc nào thì vò nát tờ giấy ném đi ngay.

- **Lưu trữ:** Biến local, tham số method, reference đến object
- **Đặc điểm:** Cực nhanh, tự động dọn dẹp khi method kết thúc
- **Kích thước:** Nhỏ, cố định (thường vài MB)
- **Cơ chế:** LIFO (Last In First Out - Vào sau ra trước)

### Heap - "Nhà kho khổng lồ"

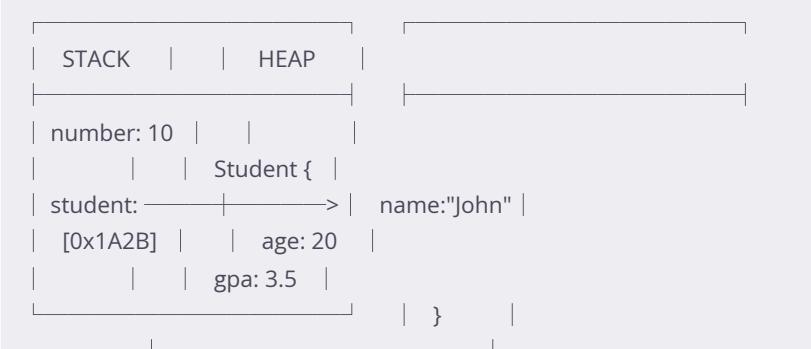
Heap giống như một **nhà kho rộng lớn**. Khi mua đồ lớn (TV, tủ lạnh - tức Object), bạn không để lên bàn mà cất vào kho. Trên bàn chỉ giữ địa chỉ kho.

- **Lưu trữ:** Objects, arrays (những thứ to lớn, phức tạp)
- **Đặc điểm:** Chậm hơn Stack, được dọn dẹp bởi Garbage Collector
- **Kích thước:** Lớn, động (có thể tăng giảm)
- **Cơ chế:** Truy cập ngẫu nhiên qua reference

## Ví dụ minh họa Stack vs Heap

```
public class MemoryExample {  
    public static void main(String[] args) {  
        // Biến primitive lưu trực tiếp trong Stack  
        int number = 10;  
  
        // Object được tạo trong Heap  
        // Reference 'student' lưu trong Stack  
        // Object thực tế lưu trong Heap  
        Student student = new Student("John", 20, 3.5);  
  
        // Khi method kết thúc:  
        // - Stack tự động xóa 'number' và 'student'  
        // - Object trong Heap chờ Garbage Collector  
    }  
}
```

### Sơ đồ bộ nhớ



### 🧠 Memory Trace - Dòng chảy bộ nhớ

Dòng Code	Stack	Heap	Giải thích
<code>int number = 10;</code>	<code>number: 10</code>	(Trống)	Primitive lưu trực tiếp
<code>new Student(...)</code>	(Chờ ref)	Student object	Tạo object trong Heap
<code>Student student = ...</code>	<code>student: 0x1A2B</code>	Student object	Lưu địa chỉ vào Stack
(Kết thúc method)	Xóa sạch	Chờ GC dọn	Stack tự hủy

## Reference (Tham chiếu) - Chìa khóa quan trọng

**Reference** là địa chỉ trỏ đến object trong Heap. Hiểu đúng về reference giúp bạn tránh nhiều bug khó hiểu!

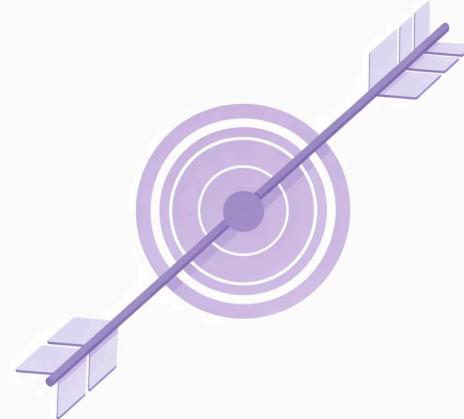
### Ví dụ: 2 Reference trỏ cùng Object

```
Student student1 = new Student("A", 20, 3.5);

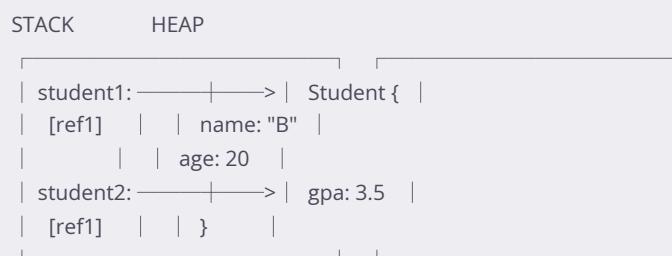
// student2 trỏ đến CÙNG object với student1
Student student2 = student1;

// Thay đổi qua student2
student2.setName("B");

// Ảnh hưởng đến student1!
System.out.println(student1.getName());
// Output: "B" (không phải "A")!
```



### Sơ đồ bộ nhớ



#### ⚠ Lưu ý quan trọng:

- student1 và student2 là 2 **reference khác nhau**
- Nhưng cả 2 đều **trỏ đến cùng 1 object**
- Thay đổi qua reference nào cũng **Ảnh hưởng lẫn nhau**
- Đây là nguồn gốc của nhiều bug nếu không cẩn thận!

### So sánh Objects: == vs equals()

```
Student s1 = new Student("A", 20, 3.5);
Student s2 = new Student("A", 20, 3.5);

// So sánh reference (địa chỉ)
System.out.println(s1 == s2); // false - 2 object khác nhau trong Heap

// So sánh nội dung (cần override equals())
System.out.println(s1.equals(s2)); // true - nếu đã override equals() đúng cách
```

## 2.1.4. Vòng đời của Object

Mỗi object trong Java trải qua 4 giai đoạn rõ ràng từ lúc sinh ra cho đến khi biến mất. Hiểu vòng đời này giúp bạn quản lý bộ nhớ hiệu quả.

### 1. Tạo (Creation)

```
Student student = new Student("John", 20, 3.5);
```

- Java cấp phát bộ nhớ trong Heap
- Khởi tạo fields với giá trị mặc định (null, 0, false)
- Gọi constructor để set giá trị thực
- Reference được gán địa chỉ object

### 2. Sử dụng (Usage)

```
student.displayInfo();  
student.setGpa(3.8);  
double gpa = student.getGpa();
```

- Object được truy cập qua reference
- Có thể đọc/ghi dữ liệu
- Có thể gọi các methods

### 3. Mất tham chiếu (Unreferenced)

```
student = null; // Cắt đứt reference
```

- Object vẫn tồn tại trong Heap
- Nhưng không thể truy cập được nữa
- Trở thành "rác" chờ được dọn

### 4. Thu gom (Garbage Collection)

Tự động, không cần code

- Garbage Collector tự động phát hiện object rác
- Giải phóng bộ nhớ trong Heap
- Không thể biết chính xác khi nào GC chạy

## 2.2. ĐÓNG GÓI (ENCAPSULATION)

### 2.2.1. Encapsulation là gì?

#### Định nghĩa

**Encapsulation (Đóng gói)** là một trong 4 trụ cột của OOP, dựa trên 3 nguyên lý:

1. **Gom dữ liệu và hành vi** lại với nhau trong một class
2. **Che giấu chi tiết bên trong**, chỉ expose những gì cần thiết
3. **Bảo vệ dữ liệu** khỏi truy cập/thay đổi trái phép

Encapsulation giống như việc bạn đóng gói món hàng vào hộp - bên ngoài chỉ thấy hộp đẹp, bên trong phức tạp được che giấu.



## Ví dụ thực tế: Chiếc xe hơi



### Giao diện đơn giản (Public API)

Bạn chỉ cần biết: Nhấn ga → Xe chạy,  
Đạp phanh → Xe dừng. Không cần hiểu  
động cơ hoạt động thế nào.



### Chi tiết phức tạp (Private Implementation)

Động cơ, hộp số, hệ thống phanh được  
"đóng gói" bên trong. Bạn không thể (và  
không cần) truy cập trực tiếp.

## Áp dụng vào Code: Class BankAccount

### ✗ Không Encapsulation (Nguy hiểm!)

```
public class BankAccount {  
    public double balance; // Ai cũng truy cập được!  
}  
  
// Ở nơi khác - Có thể làm bất cứ điều gì!  
BankAccount account = new BankAccount();  
account.balance = -1000; //
```

## 2.2.2. Access Modifiers - Các mức Truy cập

Java cung cấp 4 mức độ kiểm soát truy cập, giống như 4 mức độ riêng tư trong đời sống:

Modifier	Trong class	Trong package	Trong subclass	Mọi nơi
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

 **private** - Nhật ký cá nhân   
Chỉ mình bạn đọc được. Bí mật tuyệt đối!

 **protected** - Gia sản dòng họ   
Gia đình và con cháu (subclass) được dùng.

 **default** - Giấy note tủ lạnh   
Chỉ gia đình (cùng package) đọc được.

 **public** - Biển quảng cáo   
Ai đi qua cũng thấy. Công khai hoàn toàn!

## 1. private - Mức riêng tư tuyệt đối

**private** có nghĩa là "*Chỉ mình tôi biết, không ai khác được xem*". Chỉ code trong cùng class mới truy cập được.

### Định nghĩa private members

```
public class Student {  
    // Field private  
    private String name;  
  
    // Method private  
    private void secretMethod() {  
        // Logic bí mật  
    }  
  
    // Method public có thể dùng private  
    public void publicMethod() {  
        name = "John"; //
```

## 2. default (Package-private) - Cấp gia đình

**default** (không ghi modifier nào) có nghĩa là "*Mọi người trong nhà (package) đều biết*". Ít dùng trong thực tế.

```
// File: com.example.Student.java
package com.example;

class Student { // default class
    String name; // default field

    void display() { // default method
        System.out.println(name);
    }
}
```

```
// File: com.example.Main.java (cùng package)
package com.example;

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.name = "John"; //
```

### 3. protected - Dành cho con cháu

**protected** có nghĩa là "*Gia đình và con cháu được biết*". Truy cập được trong cùng package và tất cả subclass (kể cả ở package khác).

```
// File: com.example.Animal.java
package com.example;

public class Animal {
    protected String name; // protected field

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

#### ✓ Truy cập từ subclass (cùng package)

```
package com.example;

public class Dog extends Animal {
    public void bark() {
        name = "Buddy"; //
```

## 4. public - Công khai toàn cầu

**public** có nghĩa là "Ai cũng biết, ai cũng dùng được". Không có hạn chế truy cập.



Giống như một quảng trường công cộng, mọi thành phần được đánh dấu **public** đều có thể được truy cập và sử dụng từ bất cứ đâu trong chương trình, dù là trong cùng một package hay từ các package hoàn toàn khác.

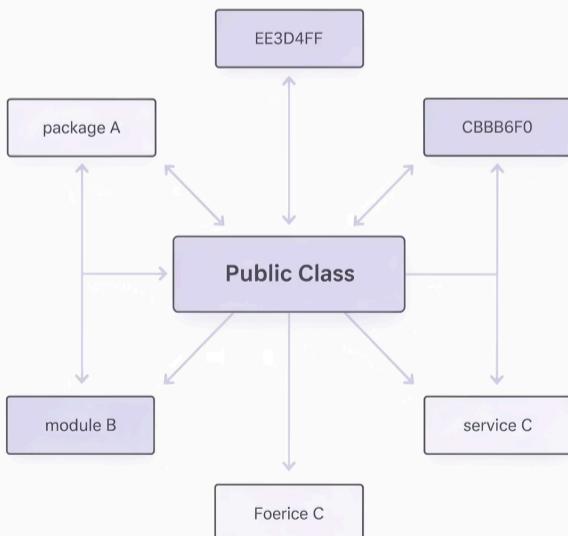
### Định nghĩa Public Class và Thành viên

Dưới đây là ví dụ về một lớp **Student** với các thành viên **public**:

```
public class Student {  
    public String name; // Public field - truy cập từ mọi nơi  
    public int age;  
  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public void displayInfo() {  
        System.out.println("Name: " + name + ", Age: " + age);  
    }  
}
```

### Truy cập từ các Package Khác

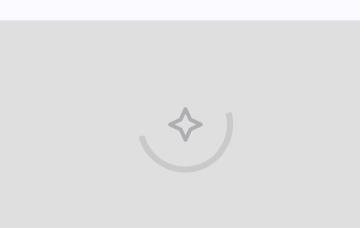
Để minh họa khả năng truy cập không giới hạn, đây là cách sử dụng lớp **Student** từ một package khác:



```
// File: com.other.Main.java  
package com.other;  
  
import com.example.Student; // Import từ package khác  
  
public class Main {  
    public static void main(String[] args) {  
        Student s = new Student("An", 20);  
  
        // Truy cập public members từ package khác - OK!  
        System.out.println(s.name); // OK  
        s.age = 21; // OK  
        s.displayInfo(); // OK  
    }  
}
```

### Khi nào và khi nào không nên dùng public?

- **Public class:** Class chính trong file, muốn dùng ở package khác.
- **Public methods:** API methods mà bên ngoài cần gọi (getters, setters, business methods).
- **Public constants:** final static fields dùng chung (Math.PI, Integer.MAX\_VALUE).



**⚠ Cảnh báo quan trọng:** KHÔNG nên dùng **public** cho các fields (biến) thông thường! Luôn khuyến nghị sử dụng **private** kết hợp với các phương thức **public getters/setters** để kiểm soát truy cập và bảo vệ dữ liệu.

## 2.2.3. Data Hiding - Che giấu Dữ liệu

Tại sao PHẢI che giấu dữ liệu?

✗ Vấn đề: Không che giấu

```
public class BankAccount {  
    public double balance; // Nguy hiểm!  
}  
  
// Ở nơi khác - Thảm họa!  
BankAccount account = new BankAccount();  
  
//
```

## Best Practice: Luôn dùng private cho Fields

### ✗ TÔI - Public fields

```
public class Student {  
    public String name; // Nguy hiểm!  
    public int age; // Không kiểm soát!  
}
```

```
// Ai cũng làm gì cũng được  
Student s = new Student();  
s.age = -5; //
```

## 2.3. CONSTRUCTORS, GETTERS/SETTERS & TỪ KHÓA THIS

### 2.3.1. Constructor - Hàm khởi tạo

Constructor là gì?

**Constructor** là method đặc biệt với những đặc điểm độc nhất:

- **Tên trùng với tên class** (phân biệt hoa thường)
- **Không có return type** (không phải void, không có gì cả!)
- **Tự động được gọi** khi tạo object bằng `new`
- **Vai trò:** Khởi tạo giá trị ban đầu cho object

Constructor giống như "lễ chào đón" của object - đây là cơ hội duy nhất để setup trạng thái ban đầu.



Cú pháp và Ví dụ

Cú pháp

```
[access modifier] ClassName(parameters) {  
    // Khởi tạo fields  
}
```

Ví dụ cụ thể

```
public class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Constructor  
    public Student(String name, int age, double gpa) {  
        this.name = name;  
        this.age = age;  
        this.gpa = gpa;  
    }  
  
    // Sử dụng  
    Student student = new Student("John", 20, 3.5);  
    // Constructor tự động chạy, khởi tạo object
```

## Default Constructor - Constructor mặc định

Nếu bạn **không định nghĩa constructor nào**, Java tự động tạo một **default constructor** (không tham số, không làm gì).

### Class không có constructor

```
public class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Không định nghĩa constructor nào  
}  
  
// Java tự động tạo (ẩn):  
// public Student() {}  
  
// Có thể dùng:  
Student student = new Student(); //
```

## Constructor Overloading - Nhiều Constructor

Bạn có thể định nghĩa **nhiều constructor** với tham số khác nhau (số lượng hoặc kiểu khác nhau). Điều này giúp linh hoạt khi tạo object.

```
public class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Constructor 1: Đầy đủ tham số  
    public Student(String name, int age, double gpa) {  
        this.name = name;  
        this.age = age;  
        this.gpa = gpa;  
    }  
  
    // Constructor 2: Chỉ name và age  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
        this.gpa = 0.0; // Mặc định  
    }  
  
    // Constructor 3: Chỉ name  
    public Student(String name) {  
        this.name = name;  
        this.age = 0; // Mặc định  
        this.gpa = 0.0; // Mặc định  
    }  
  
    // Constructor 4: Không tham số  
    public Student() {  
        this.name = "Unknown";  
        this.age = 0;  
        this.gpa = 0.0;  
    }  
}
```

### Sử dụng linh hoạt

```
// Dùng constructor 1  
Student s1 = new Student("John", 20, 3.5);  
  
// Dùng constructor 2  
Student s2 = new Student("Jane", 21);  
  
// Dùng constructor 3  
Student s3 = new Student("Bob");  
  
// Dùng constructor 4  
Student s4 = new Student();
```

#### Quy tắc Overloading:

- Các constructor phải **khác nhau về số lượng hoặc kiểu tham số**
- **Không thể** có 2 constructor giống hệt nhau
- Tên phải trùng với tên class

## Constructor Chaining - Gọi constructor khác

Dùng `this()` để gọi constructor khác trong cùng class. Kỹ thuật này giúp tránh code lặp lại (DRY - Don't Repeat Yourself).

```
public class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Constructor chính (Master Constructor)  
    public Student(String name, int age, double gpa) {  
        this.name = name;  
        this.age = age;  
        this.gpa = gpa;  
    }  
  
    // Constructor gọi constructor chính  
    public Student(String name, int age) {  
        this(name, age, 0.0); // Gọi constructor 3 tham số  
    }  
  
    // Constructor gọi constructor 2 tham số  
    public Student(String name) {  
        this(name, 0); // Gọi constructor 2 tham số  
    }  
  
    // Constructor gọi constructor 1 tham số  
    public Student() {  
        this("Unknown"); // Gọi constructor 1 tham số  
    }  
}
```



### ⚠ Quy tắc bắt buộc:

- `this()` phải là **dòng đầu tiên** trong constructor
- **Không thể** gọi `this()` và `super()` cùng lúc

## Luồng chạy Constructor Chaining

Hãy theo dõi "cuộc chạy tiếp sức" khi bạn gọi `new Student()`:



### Tại sao lại làm thế (The Why)?

Thay vì copy-paste code gán giá trị ở cả 4 constructor, chúng ta dồn hết logic vào **một constructor chính** (Master Constructor). Các constructor khác chỉ là "cò mồi" để gọi về constructor chính với giá trị mặc định. Code gọn hơn, sửa lỗi dễ hơn (chỉ cần sửa 1 chỗ).

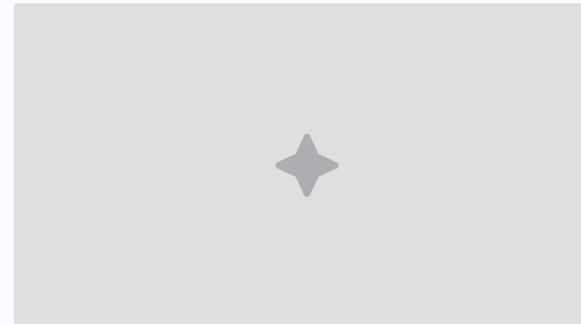
## 2.3.2. Từ khóa this - Reference đến chính mình

this là gì?

**this** là một reference đặc biệt trỏ đến **object hiện tại** (object đang thực thi method/constructor).

Tưởng tượng this giống như từ "**tôi**" trong câu nói:

- "Tôi tên là John" → this.name = "John"
- "Tôi 20 tuổi" → this.age = 20



## 4 cách sử dụng this

1

### Phân biệt field và parameter

```
public class Student {  
    private String name; // Field  
  
    public void setName(String name) { // Parameter  
        this.name = name;  
        // this.name = field của object  
        // name = parameter của method  
    }  
}
```

Đây là cách dùng phổ biến nhất!

2

### Gọi constructor khác

```
public Student(String name) {  
    this(name, 0); // Gọi constructor khác  
}
```

Đã học ở phần Constructor Chaining.

3

### Truyền object hiện tại

```
public class Student {  
    public void register(Course course) {  
        course.addStudent(this);  
        // Truyền chính object Student này  
    }  
}
```

Hữu ích khi cần truyền object sang class khác.

4

### Return object hiện tại (Method chaining)

```
public class Student {  
    public Student setName(String name) {  
        this.name = name;  
        return this; // Trả về chính object này  
    }  
  
    public Student setAge(int age) {  
        this.age = age;  
        return this;  
    }  
  
    // Method chaining  
    Student s = new Student()  
        .setName("John")  
        .setAge(20);
```

Kỹ thuật nâng cao, tạo API dễ đọc.

## 2.3.3. Getters và Setters - Cửa ra vào của Object

Tại sao cần Getters/Setters?

### Vấn đề

- Fields `private` → Không thể truy cập từ bên ngoài
- Nhưng đôi khi **CẦN** cho phép đọc/ghi giá trị
- Làm thế nào để vừa bảo vệ vừa cho phép truy cập?

### Giải pháp

- Getter (Accessor):** Method `public` để **đọc** giá trị field
- Setter (Mutator):** Method `public` để **ghi** giá trị field
- Getters/Setters là "cửa ra vào" được kiểm soát của object



## Getter (Accessor) - Đọc giá trị

### Cú pháp Getter

```
public returnType getFieldName() {  
    return fieldName;  
}
```

### Quy ước đặt tên

- **get** + Tên field (viết hoa chữ cái đầu)
- **Boolean**: Có thể dùng **is** thay vì **get**

```
// Boolean field  
private boolean active;  
  
public boolean isActive() {  
    return active; // is thay vì get  
}
```

### Ví dụ đầy đủ

```
public class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Getter cho name  
    public String getName() {  
        return name;  
    }  
  
    // Getter cho age  
    public int getAge() {  
        return age;  
    }  
  
    // Getter cho gpa  
    public double getGpa() {  
        return gpa;  
    }  
  
    // Sử dụng  
    Student student = new Student("John", 20, 3.5);  
    String name = student.getName(); // "John"  
    int age = student.getAge(); // 20  
    double gpa = student.getGpa(); // 3.5
```

## Setter (Mutator) - Ghi giá trị

### Cú pháp Setter

```
public void setFieldName(DataType fieldName) {  
    this.fieldName = fieldName;  
}
```

### Quy ước đặt tên

- **set** + Tên field (viết hoa chữ cái đầu)
- Return type: **void**
- **Nên có validation!**

### Ví dụ với Validation

```
public class Student {  
    private String name;  
    private int age;  
    private double gpa;  
  
    // Setter cho name (có validation)  
    public void setName(String name) {  
        if (name != null && !name.trim().isEmpty()) {  
            this.name = name;  
        } else {  
            throw new IllegalArgumentException(  
                "Name cannot be null or empty");  
        }  
    }  
  
    // Setter cho age (có validation)  
    public void setAge(int age) {  
        if (age >= 0 && age <= 150) {  
            this.age = age;  
        } else {  
            throw new IllegalArgumentException(  
                "Age must be between 0 and 150");  
        }  
    }  
  
    // Setter cho gpa (có validation)  
    public void setGpa(double gpa) {  
        if (gpa >= 0.0 && gpa <= 4.0) {  
            this.gpa = gpa;  
        } else {  
            throw new IllegalArgumentException(  
                "GPA must be between 0.0 and 4.0");  
        }  
    }  
}
```

## Quy tắc thiết kế Getter/Setter thông minh

1

Không phải field nào cũng cần setter

```
public class Student {  
    private String id; // ID không đổi  
    private String name; // Name có thể đổi  
    private Date createdAt; // Ngày tạo không đổi  
  
    //
```

## 2.4. JAVA RECORDS (JAVA 14+)

### 2.4.1. Record là gì?



```
1 public record ExampleRecord(String name, int age) {  
2     //...  
3 }  
4  
5     Jina care: "afoprCorei";  
6     fittareofe: "nos"(%;"  
7 }  
8  
9     public recordExampleReor(Stringname, int age);  
10    fincore:= "oogis";  
11    fublice="dodori(flogcore)"  
12    {  
13        dstarincane: "deatord(?)";  
14        fite=finue(intaie){  
15            estrlicane("ffogcive");  
16            itf(necofe);  
17            scanplie(danper(inCace));  
18        }  
19    }  
20 }  
21  
22 }
```

### Định nghĩa

**Record** là một loại class đặc biệt trong Java (từ Java 14, chính thức ở Java 16):

- Dùng để tạo **immutable data classes** (class chỉ chứa dữ liệu, không đổi)
- **Tự động generate:** constructor, getters, equals(), hashCode(), toString()
- **Giảm thiểu boilerplate code** (code lặp lại nhảm chán)
- Là **best practice** cho Data Transfer Objects (DTOs)

#### ☐ Khi nào dùng Record:

- Data classes đơn giản (DTO, Value Objects)
- Immutable objects (không cần thay đổi sau khi tạo)
- Truyền dữ liệu giữa layers
- API responses, database entities đơn giản

## Vấn đề với POJO truyền thống

### ✗ POJO truyền thống (Nhiều code!)

```
public class Point {  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass())  
            return false;  
        Point point = (Point) o;  
        return x == point.x && y == point.y;  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(x, y);  
    }  
  
    @Override  
    public String toString() {  
        return "Point{x=" + x + ", y=" + y + '}';  
    }  
}
```

### ✓ Giải pháp: Java Record (1 dòng!)

```
public record Point(int x, int y) {  
    // Chỉ cần 1 dòng!  
}
```

#### Tự động có:

- ✓ Constructor: Point(int x, int y)
- ✓ Getters: x(), y() (không phải getX(), getY())
- ✓ equals() và hashCode()
- ✓ toString()
- ✓ Immutable (tất cả fields là final)

#### Sử dụng

```
Point p1 = new Point(10, 20);  
Point p2 = new Point(10, 20);  
  
System.out.println(p1.x()); // 10  
System.out.println(p1.y()); // 20  
System.out.println(p1.equals(p2)); // true  
System.out.println(p1);  
// Output: Point[x=10, y=20]
```

Vấn đề: 30+ dòng code cho 2 fields!

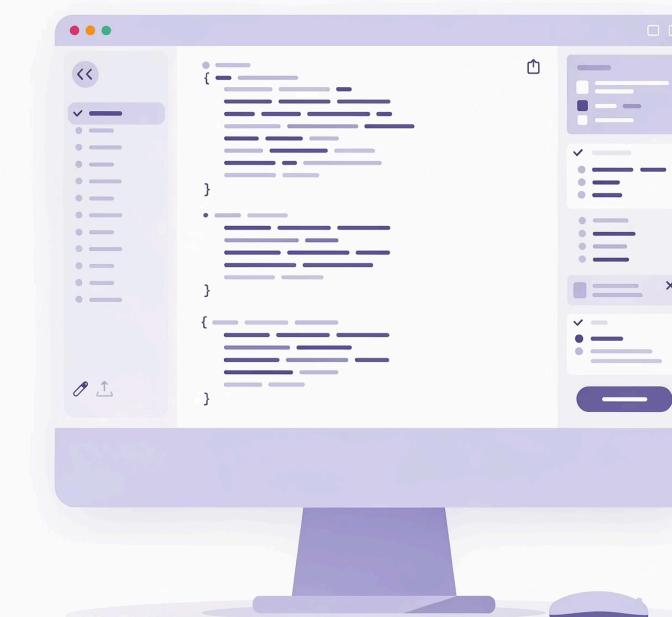
## 2.4.2. Cú pháp Record

### Cú pháp cơ bản

```
public record RecordName(type1 field1, type2 field2, ...) {  
    // Body (tùy chọn)  
}
```

### Ví dụ đơn giản

```
// Record Student  
public record Student(String name, int age, double gpa) {  
    // Không cần code gì thêm!  
}  
  
// Sử dụng  
Student student = new Student("John", 20, 3.5);  
  
// Getters (không có prefix get!)  
System.out.println(student.name()); // "John"  
System.out.println(student.age()); // 20  
System.out.println(student.gpa()); // 3.5  
  
// toString tự động  
System.out.println(student);  
// Output: Student[name=John, age=20, gpa=3.5]
```



#### ⚠️ Lưu ý quan trọng:

- Getters của Record **KHÔNG CÓ PREFIX** get: `name()` thay vì `getName()`
- Tất cả fields là `final` → **Immutable**
- Không thể thay đổi giá trị sau khi tạo

## Record với Validation (Compact Constructor)

Bạn có thể thêm **compact constructor** để validation mà không cần viết toàn bộ constructor. Java tự động gán giá trị sau khi chạy validation.

```
public record Student(String name, int age, double gpa) {  
  
    // Compact constructor - chỉ validation  
    public Student {  
        // Kiểm tra age  
        if (age < 0 || age > 150) {  
            throw new IllegalArgumentException(  
                "Age must be between 0 and 150");  
        }  
  
        // Kiểm tra gpa  
        if (gpa < 0.0 || gpa > 4.0) {  
            throw new IllegalArgumentException(  
                "GPA must be between 0.0 and 4.0");  
        }  
  
        // Không cần gán this.age = age;  
        // Java tự động gán sau validation!  
    }  
}  
  
// Sử dụng  
Student s1 = new Student("John", 20, 3.5); //
```

## Record với Methods

Record có thể có methods như class thông thường, giúp thêm logic nghiệp vụ vào data.

```
public record Point(int x, int y) {  
  
    // Instance method  
    public double distanceToOrigin() {  
        return Math.sqrt(x * x + y * y);  
    }  
  
    // Instance method  
    public Point add(Point other) {  
        return new Point(x + other.x, y + other.y);  
    }  
  
    // Static method  
    public static Point origin() {  
        return new Point(0, 0);  
    }  
  
    // Static method  
    public static Point fromString(String str) {  
        String[] parts = str.split(",");  
        int x = Integer.parseInt(parts[0]);  
        int y = Integer.parseInt(parts[1]);  
        return new Point(x, y);  
    }  
}
```

### Sử dụng instance methods

```
Point p1 = new Point(3, 4);  
  
// Instance method  
double distance = p1.distanceToOrigin();  
System.out.println(distance); // 5.0  
  
Point p2 = new Point(1, 2);  
Point p3 = p1.add(p2);  
System.out.println(p3); // Point[x=4, y=6]
```

### Sử dụng static methods

```
// Static method  
Point origin = Point.origin();  
System.out.println(origin);  
// Output: Point[x=0, y=0]  
  
// Static factory method  
Point p = Point.fromString("10,20");  
System.out.println(p);  
// Output: Point[x=10, y=20]
```

### 2.4.3. So sánh Record vs POJO

Đặc điểm	POJO	Record
Code	Nhiều dòng (30+)	Ít dòng (1+)
Immutable	Phải tự implement	Mặc định immutable
Getters	getX(), getY()	x(), y()
equals/hashCode	Phải tự override	Tự động
toString	Phải tự override	Tự động
Kế thừa	Có thể extends	Không thể (final)
Thêm fields	Dễ dàng	Không thể (immutable)
Validation	Trong constructor	Compact constructor

#### ✓ Khi nào dùng Record

- Data classes đơn giản (chỉ chứa dữ liệu)
- Immutable objects (không cần thay đổi)
- DTOs (Data Transfer Objects)
- Value objects (địa chỉ, tọa độ, tiền tệ...)
- API responses/requests
- Configuration objects

#### ✓ Khi nào dùng POJO

- Cần kế thừa (extends)
- Cần mutable (có thể thay đổi)
- Logic nghiệp vụ phức tạp
- Cần kiểm soát chi tiết getters/setters
- Làm việc với frameworks cũ (không hỗ trợ Record)

## 2.5. IMMUTABLE OBJECTS - ĐỐI TƯỢNG BẤT BIẾN

### 2.5.1. Immutable là gì?



#### Định nghĩa

**Immutable Object** là object **không thể thay đổi** sau khi được tạo:

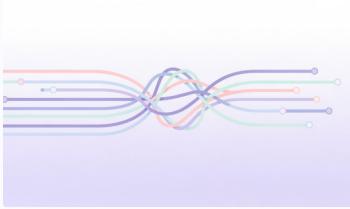
- Không thể thay đổi giá trị của fields
- Nếu cần "thay đổi", phải **tạo object mới**
- Giống như số 5 - nó luôn là 5, không thể "sửa" 5 thành 6

#### Ví dụ: String là Immutable

```
String str = "Hello";  
  
// toUpperCase() không thay đổi str  
str.toUpperCase();  
System.out.println(str); // Vẫn là "Hello"  
  
// Để "thay đổi", phải gán object mới  
str = str.toUpperCase();  
System.out.println(str); // Bây giờ là  
"HELLO"
```

String trong Java là immutable - đây là lý do tại sao các thao tác String luôn trả về String mới!

## 2.5.2. Lợi ích của Immutable Objects



### 1. Thread Safety - An toàn đa luồng

Đây là lý do quan trọng nhất! Immutable objects tự động thread-safe mà không cần synchronization.

```
// Mutable - NGUY HIỂM trong đa luồng
public class Counter {
    private int count;
    public void increment() {
        count++; // Race condition!
    }
}

// Immutable - AN TOÀN
public record Counter(int count) {
    public Counter increment() {
        return new Counter(count + 1);
    }
}

// Mỗi thread có object riêng → Không xung đột
```

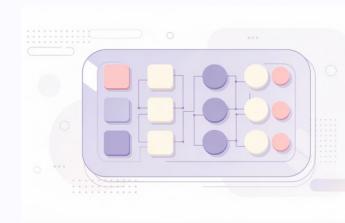


### 2. Dễ hiểu và Debug

Một khi object được tạo, bạn biết chắc giá trị của nó không đổi. Không còn "ma quỷ" thay đổi dữ liệu bí mật.

```
// Mutable - Khó debug
Point p = new Point(10, 20);
someMethod(p); // p có thể bị thay đổi!
System.out.println(p); // Không biết giá trị

// Immutable - DỄ debug
Point p = new Point(10, 20);
someMethod(p); // p KHÔNG thể thay đổi
System.out.println(p); // Vẫn là (10, 20)
```



### 3. An toàn làm Key trong Map

Immutable objects có thể dùng làm key trong HashMap vì hashCode không bao giờ thay đổi.

```
// Mutable - NGUY HIỂM
Map map = new HashMap<>();
Point p = new Point(10, 20);
map.put(p, "Value");
p.setX(30); // Thay đổi key!
map.get(p); //
```

## Thêm 3 lợi ích khác của Immutable

### 4. Caching an toàn

Có thể cache và reuse objects mà không lo bị thay đổi. String pool trong Java là ví dụ điển hình.

### 5. Không có Side Effects

Truyền object vào method không sợ bị thay đổi. Functional programming yêu thích immutability!

### 6. Failure Atomicity

Nếu constructor thất bại, object không tồn tại. Không có trạng thái "nửa vời" nguy hiểm.

### 2.5.3. Kỹ thuật thiết kế Immutable Class

Để thiết kế một class immutable đúng cách, bạn cần tuân thủ **5 quy tắc vàng:**

01

Tất cả fields là final và private

final đảm bảo không thể gán lại, private đảm bảo không thể truy cập trực tiếp từ bên ngoài.

02

Class phải là final

Ngăn không cho subclass override methods và phá vỡ tính immutable.

03

Không có setter

Không có method nào thay đổi state. Chỉ có getters để đọc.

04

Constructor phải tạo defensive copy

Nếu nhận mutable object, tạo copy thay vì lưu reference trực tiếp.

05

Getter phải trả về defensive copy

Không trả về reference trực tiếp đến mutable object bên trong.

## Ví dụ: Immutable Class với Primitive Fields

```
public final class ImmutablePoint {  
    private final int x; // final + private  
    private final int y;  
  
    public ImmutablePoint(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    // Chỉ có getters  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    //
```

## Ví dụ: Immutable Class với Mutable Fields (Khó hơn!)

Khi class chứa mutable objects (List, Date, Array...), bạn phải cẩn thận hơn với **defensive copying**.

```
import java.util.*;  
  
public final class ImmutableStudent {  
    private final String name;  
    private final int age;  
    private final List courses; // Mutable object!  
  
    public ImmutableStudent(String name, int age, List courses) {  
        this.name = name;  
        this.age = age;  
  
        //
```

## Defensive Copying - Tại sao quan trọng?

### ✗ Không có Defensive Copy

```
// Class KHÔNG immutable thực sự
```

```
public class BadStudent {
```

```
    private final List courses;
```

```
    public BadStudent(List courses) {
```

```
        this.courses = courses; // Lưu reference
```

```
}
```

```
    public List getcourses() {
```

```
        return courses; // Trả về reference
```

```
}
```

```
}
```

```
// Sử dụng - PHÁ VỠ immutability!
```

```
List list = new ArrayList<>();
```

```
list.add("Math");
```

```
BadStudent s = new BadStudent(list);
```

```
//
```

## So sánh: Record vs Immutable Class

### ✓ Record (Tự động immutable)

```
public record Point(int x, int y) {  
  
    // Tự động final, private, immutable  
  
    public Point add(Point other) {  
        return new Point(  
            x + other.x,  
            y + other.y  
        );  
    }  
  
    public Point withX(int newX) {  
        return new Point(newX, y);  
    }  
}
```

### ✓ Immutable Class (Tự implement)

```
public final class Point {  
    private final int x;  
    private final int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
  
    public Point add(Point other) {  
        return new Point(  
            x + other.x,  
            y + other.y  
        );  
    }  
}
```

#### Ưu điểm:

- Cực kỳ gọn gàng (1 dòng!)
- Tự động immutable
- equals/hashCode/toString miễn phí

#### Nhược điểm:

- Ít linh hoạt hơn
- Không thể kế thừa

```
public Point withX(int newX) {  
    return new Point(newX, y);  
}
```

```
// Phải tự override equals, hashCode, toString  
}
```

#### Ưu điểm:

- Linh hoạt hoàn toàn
- Có thể thêm logic phức tạp

#### Nhược điểm:

- Nhiều boilerplate code

☐💡 **Khuyến nghị:** Dùng Record cho data classes đơn giản. Chỉ tự viết Immutable Class khi cần logic phức tạp hoặc tương thích với Java cũ hơn.

## 2.6. STATIC MEMBERS - THÀNH VIÊN TĨNH

### 2.6.1. Static là gì?

#### Định nghĩa

**static** là từ khóa đánh dấu thành viên (field hoặc method) thuộc về **class**, không thuộc về **object**.

#### Đặc điểm quan trọng

- **Dùng chung:** Tất cả objects của class dùng chung
- **Truy cập qua class:** ClassName.staticMember (không cần tạo object)
- **Một bản copy duy nhất:** Chỉ có 1 bản trong bộ nhớ
- **Tồn tại suốt đời chương trình:** Không bị xóa khi object mất



#### Tưởng tượng thực tế:

**Instance members** giống như **bàn làm việc cá nhân** - mỗi nhân viên (object) có một bàn riêng với đồ đạc riêng.

**Static members** giống như **phòng họp chung** - tất cả nhân viên (objects) cùng dùng một phòng. Ai cũng thấy, ai cũng dùng được.

## 2.6.2. Static Variables (Biến tĩnh)

Static variable là biến dùng chung cho tất cả objects của class. Thay đổi ở một nơi, ảnh hưởng mọi nơi!

```
public class Student {  
    // Instance variable - mỗi object có riêng  
    private String name;  
    private int age;  
  
    // Static variable - dùng chung cho TẤT CẢ  
    private static int totalStudents = 0;  
  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
        totalStudents++; // Tăng mỗi khi tạo object  
    }  
  
    // Static method để truy cập static variable  
    public static int getTotalStudents() {  
        return totalStudents;  
    }  
}  
  
// Sử dụng  
Student s1 = new Student("Alice", 20);  
Student s2 = new Student("Bob", 21);  
Student s3 = new Student("Charlie", 22);  
  
// Gọi qua CLASS, không cần object  
System.out.println(Student.getTotalStudents()); // 3
```

### Sơ đồ bộ nhớ



## So sánh Static vs Instance Variable

Đặc điểm	Instance Variable	Static Variable
Số lượng	Mỗi object 1 bản copy	Chỉ 1 bản copy cho cả class
Bộ nhớ	Trong Heap (cùng object)	Trong Method Area (Class level)
Truy cập	Qua object: obj.field	Qua class: Class.field
Tồn tại	Suốt đời object	Suốt đời chương trình
Khi nào dùng	Dữ liệu riêng của object	Dữ liệu dùng chung

### Ví dụ Instance Variable

```
public class Student {  
    private String name; // Mỗi sinh viên khác nhau  
    private int age; // Mỗi sinh viên khác nhau  
}  
  
Student s1 = new Student();  
s1.name = "Alice";  
  
Student s2 = new Student();  
s2.name = "Bob";  
  
// s1 và s2 có name riêng biệt
```

### Ví dụ Static Variable

```
public class Student {  
    private static int count = 0; // Dùng chung  
}  
  
Student s1 = new Student();  
Student.count++; // count = 1  
  
Student s2 = new Student();  
Student.count++; // count = 2  
  
// count dùng chung, không phải riêng từng object
```

### 2.6.3. Static Methods (Phương thức tĩnh)

Static method là method thuộc về class, có thể gọi mà **không cần tạo object**. Đây là cách Java implement utility functions.

#### Ví dụ: Utility Class

```
public class MathUtils {  
  
    // Static method - không cần object  
    public static int add(int a, int b) {  
        return a + b;  
    }  
  
    public static double calculateCircleArea(double radius) {  
        return Math.PI * radius * radius;  
    }  
  
    public static int max(int a, int b) {  
        return (a > b) ? a : b;  
    }  
  
    // Sử dụng - KHÔNG cần new MathUtils()  
    int sum = MathUtils.add(5, 3);    // 8  
    double area = MathUtils.calculateCircleArea(5.0);  
    int maximum = MathUtils.max(10, 20); // 20
```

#### So sánh với Instance Method

```
public class Calculator {  
    private int value; // Instance variable  
  
    // Instance method - CẦN object  
    public void setValue(int value) {  
        this.value = value;  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    // Sử dụng - PHẢI tạo object  
    Calculator calc = new Calculator();  
    calc.setValue(10);  
    int v = calc.getValue();
```



## Quy tắc quan trọng của Static Methods

1

Không thể truy cập instance variables

```
public class Student {  
    private String name; // Instance variable  
  
    public static void printName() {  
        //  
    }  
}
```

## 2.6.4. Static Block - Khởi tạo tĩnh

**Static block** là block code chạy **một lần duy nhất** khi class được load vào memory (trước cả constructor!).

```
public class DatabaseConnection {  
    private static String connectionString;  
  
    // Static block - chạy KHI CLASS ĐƯỢC LOAD  
    static {  
        System.out.println("Loading database config...");  
  
        // Có thể đọc từ file, environment, ...  
        connectionString = "jdbc:mysql://localhost:3306/mydb";  
  
        // Có thể có logic phức tạp  
        if (System.getenv("DB_URL") != null) {  
            connectionString = System.getenv("DB_URL");  
        }  
  
        System.out.println("Config loaded!");  
    }  
  
    public DatabaseConnection() {  
        System.out.println("Creating connection: " + connectionString);  
    }  
}  
  
// Sử dụng  
DatabaseConnection conn1 = new DatabaseConnection();  
// Output:  
// Loading database config...  
// Config loaded!  
// Creating connection: jdbc:mysql://localhost:3306/mydb  
  
DatabaseConnection conn2 = new DatabaseConnection();  
// Output:  
// Creating connection: jdbc:mysql://localhost:3306/mydb  
// (Static block chỉ chạy 1 lần!)
```



### Khi nào dùng Static Block:

- Load configuration phức tạp
- Khởi tạo static variables với logic phức tạp
- Load native libraries
- Initialize static collections

### Thứ tự chạy:

1. Static block (1 lần khi class load)
2. Instance initializer block (mỗi lần tạo object)
3. Constructor (mỗi lần tạo object)

## 2.6.5. Khi nào NÊN dùng Static?



### 1. Utility Methods (Phương thức tiện ích)

```
public class StringUtils {  
    public static boolean isEmpty(String str) {  
        return str == null ||  
str.trim().isEmpty();  
    }  
  
    public static String capitalize(String str) {  
        if (isEmpty(str)) return  
str;  
        return str.substring(0,  
1).toUpperCase()  
        + str.substring(1);  
    }  
  
    // Sử dụng  
    if  
    (StringUtils.isEmpty(name)  
){  
    // ...  
}
```



### 2. Constants (Hằng số)

```
public class Constants {  
    public static final  
double PI = 3.14159;  
    public static final int  
MAX_STUDENTS = 100;  
    public static final String  
DEFAULT_NAME =  
"Unknown";  
}  
  
// Sử dụng  
double area =  
Constants.PI * radius *  
radius;
```



### 3. Counters / Shared State

```
public class Student {  
    private static int nextId  
= 1;  
    private int id;  
  
    public Student(String  
name) {  
        this.id = nextId++; //  
ID tự tăng  
        this.name = name;  
    }  
}
```



### 4. Factory Methods

```
public class Student {  
    private String name;  
  
    private Student(String  
name) {  
this.name = name;  
}  
  
// Static factory method  
public static Student  
createWithName(String name)  
{  
    return new  
Student(name);  
}  
  
public static Student  
of(String name) {  
    return new  
Student(name);  
}
```

Khi nào KHÔNG NÊN dùng Static?

✗ 1. Cần state riêng cho mỗi object

```
//
```

## Ví dụ Tổng hợp: BankAccount với Static

```
public class BankAccount {  
    // Static variable - đếm số tài khoản  
    private static int accountCount = 0;  
  
    // Static constant - lãi suất chung  
    private static final double INTEREST_RATE = 0.05;  
  
    // Static variable - tổng số tiền trong ngân hàng  
    private static double totalBankBalance = 0;  
  
    // Instance variables - dữ liệu riêng của mỗi tài khoản  
    private int accountNumber;  
    private String ownerName;  
    private double balance;  
  
    // Static block - khởi tạo khi class load  
    static {  
        System.out.println("BankAccount system initialized");  
        // Có thể load config, connect database...  
    }  
  
    // Constructor  
    public BankAccount(String ownerName, double initialBalance) {  
        accountCount++; // Tăng số lượng tài khoản  
        this.accountNumber = accountCount;  
        this.ownerName = ownerName;  
        this.balance = initialBalance;  
        totalBankBalance += initialBalance; // Cộng vào tổng  
    }  
  
    // Static method - tính lãi suất  
    public static double calculateInterest(double amount) {  
        return amount * INTEREST_RATE;  
    }  
  
    // Static method - lấy số lượng tài khoản  
    public static int getAccountCount() {  
        return accountCount;  
    }  
  
    // Static method - lấy tổng số tiền  
    public static double getTotalBankBalance() {  
        return totalBankBalance;  
    }  
  
    // Instance method  
    public void deposit(double amount) {  
        if (amount > 0) {  
            balance += amount;  
            totalBankBalance += amount;  
        }  
    }  
  
    public void withdraw(double amount) {  
        if (amount > 0 && amount <= balance) {  
            balance -= amount;  
            totalBankBalance -= amount;  
        }  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
}
```

## Sử dụng BankAccount

```
// Static method - không cần object  
double interest = BankAccount.calculateInterest(1000);  
System.out.println("Interest: " + interest); // 50.0  
  
// Tạo tài khoản 1  
BankAccount acc1 = new BankAccount("Alice", 1000);  
System.out.println("Account #" + acc1.accountNumber); // 1  
  
// Tạo tài khoản 2  
BankAccount acc2 = new BankAccount("Bob", 2000);  
System.out.println("Account #" + acc2.accountNumber); // 2  
  
// Static method - xem tổng số tài khoản  
System.out.println("Total accounts: " +  
    BankAccount.getAccountCount()); // 2  
  
// Static method - xem tổng số tiền  
System.out.println("Total balance: " +  
    BankAccount.getTotalBankBalance()); // 3000.0
```

```
// Nạp tiền vào acc1  
acc1.deposit(500);  
  
// Tổng số tiền tăng lên!  
System.out.println("Total balance: " +  
    BankAccount.getTotalBankBalance()); // 3500.0  
  
// Rút tiền từ acc2  
acc2.withdraw(300);  
  
// Tổng số tiền giảm xuống!  
System.out.println("Total balance: " +  
    BankAccount.getTotalBankBalance()); // 3200.0
```

☐💡 **Liên kết với Chương 1:** Bạn đã học về static method cơ bản ở Chương 1, phần 0.8. Nay bạn hiểu sâu hơn về static variables, static blocks, và khi nào nên dùng!

# TÓM TẮT CHƯƠNG 2

## Kiến thức đã học

- Class và Object

Khuôn mẫu và thực thể. Hiểu rõ sự khác biệt và mối quan hệ.

- Stack vs Heap

Cơ chế cấp phát bộ nhớ, reference, và vòng đời object.

- Encapsulation

Access Modifiers (private, default, protected, public) và Data Hiding.

- Constructors

Default, overloaded, constructor chaining với this().

- this keyword

4 cách sử dụng từ khóa this trong Java.

- Getters/Setters

Thiết kế thông minh với validation và defensive copying.

- Java Records

Data classes hiện đại, compact, immutable by default.

- Immutable Objects

Thiết kế đối tượng bất biến, thread-safe, và các best practices.

- Static Members

Variables, methods, blocks - dùng chung cho cả class.

## Kỹ năng đã có

- ✓ Thiết kế Class

Với encapsulation đúng cách, fields private, methods public có chọn lọc

- ✓ Constructors Pro

Overloading, chaining, validation trong constructor

- ✓ Getters/Setters

Với validation và defensive copying cho collections

- ✓ Immutable Objects

Thiết kế thread-safe objects theo best practices

- ✓ Java Records

Sử dụng thành thạo cho data classes hiện đại

- ✓ Static Members

Biết khi nào nên và không nên dùng static

# BÀI TẬP CHƯƠNG 2

Để thành thạo OOP, bạn cần **THỰC HÀNH!** Dưới đây là 7 bài tập từ cơ bản đến nâng cao. Hãy làm hết để consolidate kiến thức!

## 1 Class và Object cơ bản

**Yêu cầu:** Tạo class Rectangle với fields (width, height), constructor, và methods (calculateArea, calculatePerimeter, displayInfo).

**Test:**

1

```
Rectangle rect = new Rectangle(5.0, 3.0);
rect.displayInfo();
// Output: Width: 5.0, Height: 3.0, Area: 15.0, Perimeter: 16.0
```

## 2 Encapsulation và Validation

**Yêu cầu:** Tạo class BankAccount với fields private (accountNumber, balance), methods (deposit, withdraw, getBalance) có validation đầy đủ.

**Test:**

2

```
BankAccount account = new BankAccount("123456");
account.deposit(1000);
account.withdraw(300);
System.out.println(account.getBalance()); // 700.0
```

## 3 Constructor Overloading

**Yêu cầu:** Tạo class Student với 4 constructors khác nhau (đầy đủ tham số, 2 tham số, 1 tham số, không tham số). Sử dụng constructor chaining với this().

**Test:** Tạo 4 objects với 4 constructors và in thông tin.

3

## 4 Getters/Setters với Validation

**Yêu cầu:** Tạo class Product với fields (name, price, quantity). Tất cả setters phải có validation và throw IllegalArgumentException nếu không hợp lệ.

**Validation rules:** name không null/rỗng, price >= 0, quantity >= 0.

4

## 5 Java Records

**Yêu cầu:** Tạo record Person (name, age, email) với validation trong compact constructor. Thêm method isAdult(). So sánh với class Person truyền thống.

**Test:**

5

```
Person p = new Person("John", 25, "john@example.com");
System.out.println(p.isAdult()); // true
System.out.println(p); // Person[name=John, age=25, ...]
```

## 6 Immutable Objects

**Yêu cầu:** Tạo class ImmutableBook (final class, final fields: title, author, isbn, publishedYear). Chỉ có getters. Thêm methods withTitle(), withAuthor() trả về object mới.

**Test:**

6

```
ImmutableBook book = new ImmutableBook("Java", "Author", "123", 2023);
ImmutableBook updated = bookWithTitle("Advanced Java");
System.out.println(book.getTitle()); // "Java" (không đổi)
System.out.println(updated.getTitle()); // "Advanced Java"
```

## 7 Tổng hợp - Quản lý Sinh viên

**Yêu cầu:** Tạo hệ thống hoàn chỉnh với:

- **Class Student:** fields private, constructors, getters/setters có validation, displayInfo()
- **Class StudentManager:** List students, methods (addStudent, removeStudent, findById, displayAll, getAverageGpa)
- **Class Main:** Tạo manager, thêm ít nhất 3 sinh viên, test tất cả methods

**Yêu cầu bổ sung:** Tuân thủ Clean Code, JavaDoc đầy đủ, validation trong setters.

# 🎓 KẾT THÚC CHƯƠNG 2

Chúc mừng bạn đã hoàn thành Chương 2! 🎉

Bạn đã làm chủ những kiến thức nền tảng quan trọng nhất của OOP trong Java. Từ Class/Object cơ bản đến các kỹ thuật nâng cao như Immutable Objects và Java Records. Đây là bước đệm vững chắc cho những chương tiếp theo!

## 📚 Tài liệu tham khảo

- [Java Tutorials - Oracle](#)
- [Java Records - JEP 395](#)
- **Effective Java** - Joshua Bloch (Item 17: Minimize mutability)

## 🚀 Bước tiếp theo

Sau khi hoàn thành bài tập, hãy tiến tới **Chương 3: Kế thừa & Đa hình**, nơi bạn sẽ học về:

- Inheritance (Kế thừa)
- Polymorphism (Đa hình)
- Abstract Classes & Interfaces

## 💪 Tips học tốt

- Làm hết 7 bài tập
- Code lại các ví dụ
- Thủ nghiệm và sửa lỗi
- Đọc code của người khác
- Tham gia cộng đồng Java

Chúc bạn học tốt! 🚀

Keep coding, keep learning, keep growing!