# Các tính chất quan trọng của hướng đối tượng (OOP)

1. **Đóng gói (Encapsulation):**

* Giúp bảo vệ dữ liệu bằng cách ẩn thông tin bên trong object và chỉ cho phép truy cập thông qua các phương thức công khai (public methods).

1. **Kế thừa (Inheritance):**

* Cho phép tạo ra các class mới từ các class đã tồn tại, giúp tái sử dụng mã và xây dựng các mối quan hệ giữa các class.

1. **Đa hình (Polymorphism):**

* Cho phép một phương thức hoặc một đối tượng có thể có nhiều hình thức khác nhau. Điều này được thực hiện thông qua phương thức ghi đè (overriding) và ghi đè (overloading).

1. **Trừu tượng (Abstraction):**

* Cho phép ẩn đi các chi tiết phức tạp và chỉ cung cấp các chức năng cần thiết cho người sử dụng. Điều này có thể được thực hiện thông qua các lớp trừu tượng (abstract class) và giao diện (interface).

# Access modifier trong Java

1. **Public:**

* Có thể được truy cập từ bất kỳ lớp nào, không bị giới hạn bởi package.

1. **Protected:**

* Có thể được truy cập từ các lớp trong cùng package hoặc các lớp con (subclass) ở các package khác.

1. **Default (Package-private):**

* Nếu không chỉ định access modifier, mặc định là default. Chỉ có thể được truy cập từ các lớp trong cùng một package.

1. **Private:**

* Chỉ có thể được truy cập từ chính lớp đó. Không thể truy cập từ lớp con hay lớp khác.

# Class và Instance trong Java

1. **Class**

* Là bản thiết kế (template) cho đối tượng. Nó định nghĩa các thuộc tính (attributes) và phương thức (methods) của đối tượng.

1. **Instance**

* Là một bản sao cụ thể (concrete instance) của một class. Khi tạo một đối tượng từ class, thì cũng đồng thời đang tạo ra một instance của class đó.

# Abstract Class và Interface trong Java

1. **Abstract Class:**

* Là class không thể khởi tạo và có thể chứa cả phương thức triển khai (Implementation) và phương thức trừu tượng (abstract). Có thể có các trường dữ liệu (fields) và có thể kế thừa từ 1 class khác
* Sử dụng khi muốn nhóm các class lại với nhau mà có một số phương thức chung và có thể có một số phương thức mà class con phải triển khai.
* Ví dụ sử dụng:

abstract class Animal {

String name;

// Constructor cho lớp Animal

public Animal(String name) {

this.name = name;

}

// Phương thức trừu tượng: các class con phải triển khai

abstract void makeSound();

// Phương thức thông thường: có thể sử dụng trực tiếp

void eat() {

System.out.println(name + " is eating");

}

}

class Dog extends Animal {

public Dog(String name) {

super(name);

}

// Ghi đè phương thức trừu tượng

@Override

void makeSound() {

System.out.println(name + " says: Meow");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog("Husky");

dog.eat(); // Kế thừa phương thức từ Animal

dog.makeSound();

}

}

1. **Interface:**

* Là một tập hợp các phương thức trừu tượng (abstract) mà không có phương thức triển khai (Implementation). Class thực thi Interface phải cung cấp phương thức triển khai (Implementation) cho tất cả các phương thức của interface đó.
* Sử dụng khi bạn muốn đảm bảo rằng 1 class thực hiện một số phương thức mà không quan tâm đến phương thức triển khai.
* Ví dụ sử dụng:

interface Swim {

void swim(); // Phương thức trừu tượng

}

interface MakeSound {

void makeSound(); // Phương thức trừu tượng

}

class Dog implements Swim, MakeSound {

String name;

public Dog(String name) {

this.name = name;

}

// Ghi đè phương thức từ interface Swim

@Override

public void swim() {

System.out.println(name + " is swimming");

}

// Ghi đè phương thức từ interface MakeSound

@Override

public void makeSound() {

System.out.println(name + " says: Meow");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog("Husky");

dog.swim();

dog.makeSound();

}

}

1. **Trường hợp cùng tên:**

* Cùng tên nhưng khác kiểu trả về:
* Nếu có 2 interface (hoặc 1 abstract class và 1 interface) có phương thức cùng tên nhưng khác kiểu trả về, thì cần phải đổi tên phương thức của 1 interface hoặc abstract class, hoặc sử dụng cách gọi cụ thể cho từng interface hoặc abstract class
* Ví dụ:

// interface InterfaceA {  
// int getValue(); // Phương thức trả về kiểu int  
// }  
// interface InterfaceB {  
// double getValue(); // Phương thức trả về kiểu double  
// }

interface InterfaceA {  
 int getIntValue(); // Đổi tên phương thức  
}  
  
interface InterfaceB {  
 double getDoubleValue(); // Đổi tên phương thức  
}

class MyClass implements InterfaceA, InterfaceB {  
 @Override  
 public int getIntValue() { // Triển khai cho InterfaceA  
 return 5;  
 }  
  
 @Override  
 public double getDoubleValue() { // Triển khai cho InterfaceB  
 return 5.0;  
 }  
}

* Cùng tên và cùng kiểu trả về:
* Nếu cả 2 đều có cùng tên và cùng kiểu trả về, chỉ cần định nghĩa một phương thức trong class thực hiện.
* Ví dụ:

interface InterfaceA {

int getValue(); // Phương thức trả về kiểu int

}

interface InterfaceB {

int getValue(); // Phương thức trả về kiểu int

}

class MyClass implements InterfaceA , InterfaceB {

public int getValue() {

return 5;

}

}

# Overriding và Overloading trong Java

1. **Overriding:**

* Là việc cung cấp một phương thức triển khai mới cho phương thức trong class con mà đã được định nghĩa trong class cha. Điều này thường xảy ra khi muốn thay đổi hành vi của phương thức cha trong class con.

1. **Overloading:**

* Là việc định nghĩa nhiều phương thức cùng tên trong cùng một class, nhưng khác nhau về số lượng hoặc kiểu tham số. Điều này cho phép phương thức hoạt động với nhiều kiểu dữ liệu khác nhau.

# Overriding với private và static

1. **Private**

* Không thể được ghi đè vì private chỉ có thể được truy cập trong class mà nó được định nghĩa.

1. **Static**

* Không thể được ghi đè nhưng có thể được định nghĩa lại trong class con (method hiding).

# Một phương thức final có thể kế thừa được không ?

* Một phương thức final không thể được kế thừa và thay đổi. Vì 1 phương thức được đánh dấu là final không thể bị ghi đè trong các class con.

# This và Super trong Java

* This: Tham chiếu đến đối tượng hiện tại (instance) của class. Dùng để phân biệt giữa thuộc tính của class và tham số trong phương thức hoặc constructor.
* Super: Tham chiếu đến class cha (superclass) của class hiện tại. Dùng để gọi phương thức hoặc constructor của class cha.