OOP

1. Nêu ra các tính chất quan trọng của hướng đối tượng

- Tính đóng gói (Encapsulation): là việc gom các thuộc tính (dữ liệu) và phương thức (hành vi) liên quan vào một đơn vị duy nhất gọi là lớp (class). Mục đích chính bảo vệ dữ liệu bên trong của đối tượng khỏi sự truy cập và thay đổi trái phép từ bên ngoài. Thay vì cho phép truy cập trực tiếp, cung cấp các phương thức công khai (public methods) để tương tác với dữ liệu, đảm bảo tính toàn vẹn và bảo mật.

- Tính kế thừa (Inheritance): cho phép một lớp mới (lớp con) có thể kế thừa các thuộc tính và phương thức từ một lớp đã có (lớp cha). Điều này giúp tái sử dụng mã nguồn, tránh việc phải viết lại những đoạn code giống nhau.

- Tính đa hình (Polymorphism): là khả năng một đối tượng có thể có nhiều hình thái khác nhau hoặc một phương thức có thể được thực hiện theo nhiều cách khác nhau tùy thuộc vào đối tượng gọi nó. Tính chất này thường được thể hiện qua việc ghi đè phương thức (method overriding)

- Tính trừu tượng (Abstraction): hiển thị những thông tin cần thiết và che giấu đi những chi tiết phức tạp, không quan trọng bên trong. Nó giúp đơn giản hóa hệ thống và cho phép người dùng chỉ cần tương tác với giao diện đã được định nghĩa sẵn, mà không cần biết cách thức hoạt động bên trong.

2. Access modifier trong java có những loại nào ? Nêu đặc điểm của từng loại

- public

Phạm vi: Rộng nhất. Các thành phần được khai báo là **public** có thể được truy cập từ bất cứ đâu trong chương trình, không giới hạn bởi lớp hay package.

Đặc điểm: Thường được sử dụng cho các phương thức hoặc biến cần được sử dụng rộng rãi, ví dụ như các phương thức của một API công cộng.

- protected

Phạm vi: Hẹp hơn public. Các thành phần protected có thể được truy cập:

- Trong cùng một lớp.
- Trong cùng một package.
- Trong các lớp con (subclass), kể cả khi lớp con đó ở một package khác.

Đặc điểm**:** Rất hữu ích trong việc kế thừa, cho phép các lớp con truy cập và sử dụng các thành phần của lớp cha mà không cần phải làm chúng trở nên công khai hoàn toàn.

- **default** (mặc định)

Phạm vi: Khi một thành phần không được khai báo với bất kỳ access modifier nào, nó sẽ có quyền truy cập mặc định (default). Các thành phần này chỉ có thể được truy cập trong cùng một package.

Đặc điểm: Giúp kiểm soát truy cập trong nội bộ một module hoặc package, đảm bảo các lớp bên ngoài không thể can thiệp vào.

- private

Phạm vi: Hẹp nhất. Các thành phần private chỉ có thể được truy cập từ bên trong **chính lớp đó**.

Đặc điểm: Đây là công cụ chính để đạt được tính đóng gói. Nó bảo vệ dữ liệu nội bộ của lớp khỏi sự truy cập trái phép từ bên ngoài, đảm bảo tính toàn vẹn và an toàn của dữ liệu.

3. Phân biệt class và instance

- Class (Lớp)

Định nghĩa: Một class là một bản thiết kế hoặc khuôn mẫu để tạo ra các đối tượng. Nó định nghĩa các thuộc tính (dữ liệu) và phương thức (hành vi) chung cho tất cả các đối tượng được tạo ra từ nó.

Tính chất:

- Trừu tượng: Một class chỉ là một ý tưởng trừu tượng, không tồn tại vật lý trong bộ nhớ cho đến khi bạn tạo ra một đối tượng từ nó.
- Không chiếm bộ nhớ: Bản thân class không chiếm dung lượng bộ nhớ khi chương trình chạy.

Ví dụ:

```
Class Car{
private string color;
private string brand;}
```

- Instance (Đối tượng)

Định nghĩa: Một instance là một thực thể cụ thể được tạo ra từ một class. Nó là một bản sao độc lập của class, với các giá trị thuộc tính riêng biệt và khả năng thực hiện các phương thức đã được định nghĩa trong class đó.

Tính chất:

- **Cụ thể:** Mỗi instance là một thực thể vật lý, tồn tại trong bộ nhớ khi chương trình chạy.
- **Chiếm bộ nhớ:** Khi bạn tạo một instance, nó sẽ chiếm một vùng bộ nhớ để lưu trữ các giá trị của thuộc tính.

Ví dụ:

```
Car myCar = new Car(“red”, “Honda”);
Car yourCar = new Car(“blue”, “Toyota”);
```

4. Phân biệt **Abstract** và **Interface** , Nêu trường hợp sử dụng cụ thể. Nếu 2 interface hoặc 1 abstract và 1 interface có 1 function cùng tên, có thể cùng hoặc khác kiểu trả về cùng được kế thừa bởi một class, chuyện gì sẽ xảy ra?

Đặc điểm	Abstract class	Interface
Bản chất	Một lớp có thể chứa cả phương thức đã được định nghĩa và phương thức trừu tượng (abstract).	Một khuôn mẫu để định nghĩa một tập hợp các phương thức trừu tượng (đến Java 8) và các hằng số.
Kế thừa	Chỉ extends **1 class abstract**	Có thể implements **nhiều interface**
Constructor	Có thể có constructor.	Không thể có constructor

Thành phần chứa được	Thuộc tính, constructor, phương thức thường, abstract methods, non-abstract methods	Chỉ hằng số, phương thức abstract (Java 8+ cho phép default, static method)
Mục đích	Chia sẻ code giữa các lớp có quan hệ **"is-a"** (là một), ví dụ Dog là một Animal	Định nghĩa các hành vi chung cho các lớp không có quan hệ "is-a" rõ ràng, ví dụ Flying là một hành vi của Bird hay Airplane.
Từ khóa	Dùng từ khóa **abstract** và **extends**	Dùng từ khóa **interface** và **implements**

- Trường hợp sử dụng cụ thể:

- **Abstract class:** Sử dụng abstract class khi bạn muốn tạo một lớp cha cung cấp một phần chức năng chung (shared code) cho các lớp con.
  Ví dụ: Animal là một abstract class. Nó có thể chứa một phương thức trừu tượng sound() và một phương thức không trừu tượng nameAnimal(). Các class con Dog và Cat sẽ kế thừa Animal và phải tự định nghĩa sound của mình, nhưng có thể sử dụng lại phương thức nameAnimal() đã có sẵn.
- **Interface:** Sử dụng interface khi bạn muốn định nghĩa một khuôn mẫu hành vi mà các lớp khác nhau, không có chung một lớp cha, phải tuân theo.
  Ví dụ: Một interface Movable với phương thức move(). Các lớp như Car, Bird, và Person có thể implement interface này, mỗi lớp sẽ định nghĩa cách di chuyển riêng của mình. Bằng cách này, chúng ta có thể gọi phương thức move() trên bất kỳ đối tượng nào implement Movable mà không cần biết đối tượng đó thuộc lớp nào.

- Trường hợp: Nếu 1 abstract và 1 interface có function cùng tên, cùng kiểu trả về

Nếu một lớp triển khai (implements) hai interface và cả hai đều có một phương thức với cùng tên và cùng kiểu trả về, sẽ không có xung đột. Lớp đó chỉ cần triển khai phương thức một lần. Trình biên dịch hiểu rằng phương thức này đáp ứng yêu cầu từ cả

hai interface. Đây là một lợi thế của Interface vì nó giải quyết vấn đề "Deadly Diamond of Death" trong đa kế thừa, nơi các lớp cha có cùng phương thức và gây ra sự mơ hồ.

Ví dụ:

- Giả sử bạn có hai interface: CanFly và CanWalk, cả hai đều có một phương thức move() trả về void.
- Một lớp SuperHero triển khai cả hai interface này. Nó chỉ cần định nghĩa phương thức move() một lần, và phương thức này sẽ được coi là đã đáp ứng yêu cầu của cả CanFly và CanWalk.

- Trường hợp: Nếu 2 interface có function cùng tên, cùng kiểu trả về

Nếu cả phương thức trừu tượng trong abstract class và phương thức trong interface đều có cùng tên và cùng kiểu trả về, không có xung đột xảy ra. Lớp con chỉ cần ghi đè (override) phương thức một lần duy nhất.

Ví dụ:

- Lớp trừu tượng Animal có một phương thức trừu tượng eat().
- Interface Herbivore cũng có phương thức eat().
- Lớp Cow kế thừa Animal và triển khai Herbivore. Lớp Cow chỉ cần triển khai phương thức eat() một lần, và phương thức này sẽ đáp ứng cả hai yêu cầu.

- Trường hợp: Xung đột khi phương thức có cùng tên nhưng khác kiểu trả về

Trình biên dịch Java sẽ báo lỗi vì không thể xác định kiểu trả về nào sẽ được sử dụng cho phương thức này. Một phương thức không thể có hai kiểu trả về khác nhau.

Ví dụ:

- Interface1 có phương thức getValue() trả về int.
- Interface2 có phương thức getValue() trả về String.

Khi một lớp cố gắng triển khai cả hai interface này, trình biên dịch sẽ báo lỗi.

5. Thế nào là **Overriding** và **Overloading**

- Overriding:

Là **việc lớp con định nghĩa lại một phương thức đã có trong lớp cha** với **cùng tên, cùng tham số và cùng kiểu trả về**, nhằm thay đổi hoặc mở rộng hành vi của phương thức đó.

Mục đích: Giúp đạt được tính **đa hình (polymorphism)** – tức là gọi phương thức qua lớp cha nhưng hành vi được quyết định bởi lớp con.

- Overloading

Là việc **trong cùng một lớp (hoặc lớp con)** định nghĩa **nhiều phương thức có cùng tên nhưng khác nhau về danh sách tham số** (số lượng, kiểu dữ liệu, hoặc thứ tự).

Mục đích: Tăng tính linh hoạt của chương trình bằng cách cho phép một hành vi hoạt động với nhiều kiểu dữ liệu đầu vào khác nhau.

6. Một function có access modifier là private or static có thể overriding được không?

- Một function có access modifier là private hoặc static **không thể overriding được**.

- Private Method Overriding

- Các phương thức private chỉ có thể được truy cập bên trong chính lớp mà chúng được định nghĩa. Vì vậy, các lớp con không thể nhìn thấy hoặc truy cập chúng, và do đó, không thể ghi đè.
- **Ví dụ:** Nếu một lớp cha có phương thức private void secretMethod(), một lớp con có thể định nghĩa một phương thức khác cũng tên là secretMethod(), nhưng đó chỉ là một phương thức hoàn toàn mới, không phải là ghi đè.

- Static Method Overriding

- Overriding là một tính năng của các phương thức liên quan đến đối tượng (instance methods), được giải quyết tại thời điểm chạy (runtime polymorphism). Ngược lại, các phương thức static thuộc về lớp chứ không phải đối tượng, và việc gọi chúng được xác định tại thời điểm biên dịch (compile time).
- **Ví dụ:** Nếu một lớp cha có phương thức static void show(), và một lớp con cũng có một phương thức static void show(), thì đó là "method hiding" chứ không phải "method overriding". Khi gọi phương thức này, Java sẽ quyết định gọi phương thức của lớp cha hay lớp con dựa trên kiểu tham chiếu, không phải kiểu đối tượng.

7. Một phương thức final có thể kế thừa được không ?

Một phương thức được đánh dấu là final có thể được **kế thừa** nhưng **không thể bị ghi đè (override)** bởi các lớp con.

- **Có thể kế thừa:** Lớp con vẫn có thể gọi và sử dụng phương thức final từ lớp cha. Chức năng của phương thức này được truyền lại cho lớp con một cách nguyên vẹn.
- **Không thể ghi đè:** Nếu một lớp con cố gắng định nghĩa lại (ghi đè) một phương thức final đã có trong lớp cha, trình biên dịch sẽ báo lỗi. Điều này giúp duy trì tính nhất quán và bảo mật của mã nguồn, đặc biệt khi phương thức đó chứa logic quan trọng không nên thay đổi.

8. Phân biệt hai từ khóa **This** và **Super**

- this: đề cập đến đối tượng hiện tại của lớp.

- super: đề cập đến đối tượng cha trực tiếp của lớp hiện tại.

Đặc điểm	This	Super
Mục đích	Dùng để truy cập các (biến và phương thức) của lớp hiện tại	Dùng để truy cập các thành viên của lớp cha trực tiếp
Trường hợp thường sử dụng	Để phân biệt giữa biến cục bộ và biến instance (khi chúng trùng tên). Dùng để gọi constructor khác trong cùng một lớp	Thường được dùng để gọi constructor của lớp cha hoặc ghi đè phương thức của lớp cha
Vị trí	Có thể sử dụng ở bất kỳ đâu trong một phương thức hoặc constructor (ngoại trừ trong một phương thức static).	Chỉ có thể sử dụng trong một lớp con, và thường là câu lệnh đầu tiên trong constructor của lớp con khi gọi constructor của lớp cha.