# OOP

# **1. Các tính chất quan trọng của hướng đối tượng**

**Tính đóng gói (Encapsulation)**

* **Định nghĩa**: Đóng gói là việc gom nhóm các thuộc tính (biến) và phương thức (hàm) liên quan vào một đối tượng, đồng thời che giấu thông tin chi tiết bên trong đối tượng với bên ngoài.
* **Lợi ích**: Bảo vệ dữ liệu, chỉ cho phép truy cập thông qua các phương thức công khai. Điều này giúp ngăn chặn việc thay đổi dữ liệu một cách không mong muốn hoặc không hợp lệ.

### **Tính kế thừa (Inheritance)**

* **Định nghĩa**: Kế thừa cho phép một lớp (class) con thừa hưởng các thuộc tính và phương thức của một lớp cha (superclass), giúp tái sử dụng mã và giảm thiểu việc viết lại mã.
* **Lợi ích**: Giúp giảm sự lặp lại mã và tổ chức chương trình theo cách dễ quản lý hơn.

### **Tính đa hình (Polymorphism)**

* **Định nghĩa**: Đa hình là khả năng một đối tượng có thể biểu hiện dưới nhiều dạng khác nhau. Nó cho phép một phương thức có thể hoạt động khác nhau tùy theo đối tượng gọi nó.
* **Lợi ích**: Giúp mở rộng và thay đổi hành vi của các đối tượng mà không cần thay đổi mã gốc.
* **Hai loại đa hình chính**:
  + **Đa hình lúc biên dịch (Compile-time polymorphism)**: Được thực hiện thông qua nạp chồng phương thức (method overloading).
  + **Đa hình lúc chạy (Runtime polymorphism)**: Được thực hiện thông qua ghi đè phương thức (method overriding).

### **Tính trừu tượng (Abstraction)**

* **Định nghĩa**: Trừu tượng là khả năng ẩn các chi tiết phức tạp và chỉ hiển thị các thông tin cần thiết của đối tượng. Nó cho phép tập trung vào bản chất cốt lõi của vấn đề mà không cần biết chi tiết bên trong.
* **Lợi ích**: Giúp đơn giản hóa chương trình và tăng tính linh hoạt trong việc phát triển và bảo trì.

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

**Access Modifiers** (Bộ điều khiển truy cập) được sử dụng để kiểm soát mức độ truy cập của các lớp, phương thức và biến. Có 4 loại access modifier trong Java: private, default, protected, và public

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Modifier | Phạm vi truy cập (Access level) | Truy cập từ cùng lớp (Same class) | Truy cập từ cùng package | Truy cập từ lớp con (Subclass) | Truy cập từ lớp khác (Other class) |
| private | Chỉ có thể truy cập trong chính lớp khai báo. | Có | Không | Không | Không |
| default | Truy cập được từ các lớp trong cùng package (gói) (Không có từ khóa cụ thể). | Có | Có | Không (nếu khác package) | Không |
| protected | Có thể truy cập từ các lớp trong cùng package và lớp con ở package khác. | Có | Có | Có | Không |
| public | Truy cập từ bất kỳ đâu (từ mọi lớp, mọi package). | Có | Có | Có | Có |

# **3. Phân biệt class và instance?**

### **Class (Lớp)**

* **Định nghĩa**: Class là một bản thiết kế (blueprint) hoặc mẫu chung để tạo ra các đối tượng (instances). Nó xác định các thuộc tính (biến) và hành vi (phương thức) mà các đối tượng của lớp đó sẽ có.
* **Vai trò**: Lớp đóng vai trò như khuôn mẫu để định nghĩa thuộc tính và hành vi của các đối tượng.
* **Tính chất**:
  + Là một khái niệm tĩnh, không tồn tại trực tiếp trong bộ nhớ.
  + Lớp chỉ được định nghĩa một lần, và có thể được sử dụng nhiều lần để tạo ra nhiều đối tượng (instances).
  + Lớp có thể chứa các biến (thuộc tính), phương thức, constructor, và các lớp lồng nhau.

### **Instance (Đối tượng)**

* **Định nghĩa**: Instance là một đối tượng cụ thể của lớp. Khi bạn tạo một đối tượng từ một lớp, đối tượng đó được gọi là instance của lớp.
* **Vai trò**: Instance đại diện cho một thể hiện cụ thể của lớp với các giá trị riêng biệt của các thuộc tính.
* **Tính chất:**
  + Instance được tạo ra bằng cách sử dụng từ khóa new.
  + Mỗi instance có một bản sao riêng biệt của các thuộc tính của lớp, có thể có giá trị khác nhau so với các instance khác.
  + Instance tồn tại trong bộ nhớ và chiếm không gian bộ nhớ.

# **4. Contructor trong Java là gì? Có những loại constructor nào? Contructor có thể bị ghi đè (overload) không?**

Constructor trong Java là một phương thức đặc biệt được sử dụng để khởi tạo các đối tượng của một lớp. Nó có cùng tên với lớp và được gọi tự động khi một đối tượng mới được tạo. Constructor được sử dụng để gán các giá trị ban đầu cho các thuộc tính của đối tượng.

### Đặc điểm của Constructor:

* Tên trùng với tên lớp: Constructor phải có cùng tên với lớp của nó.
* Không có kiểu trả về: Constructor không có kiểu trả về, kể cả void.
* Tự động gọi khi tạo đối tượng: Constructor được gọi tự động khi bạn sử dụng từ khóa new để tạo đối tượng.
* Mặc định có constructor: Nếu bạn không định nghĩa bất kỳ constructor nào, Java sẽ tự động cung cấp một constructor mặc định (default constructor) không tham số.
* Có thể có tham số: Constructor có thể có tham số để khởi tạo các thuộc tính với giá trị tùy ý.

Default Constructor (Constructor mặc định):

* Là constructor không có tham số, được Java cung cấp tự động nếu bạn không định nghĩa bất kỳ constructor nào.

Parameterized Constructor (Constructor có tham số):

* Là constructor có các tham số, cho phép khởi tạo đối tượng với các giá trị ban đầu cụ thể.

### Constructor có thể bị ghi đè (Overload) không?

Constructor có thể được overload, nhưng không thể bị override.

* Constructor Overloading: Là việc định nghĩa nhiều constructor trong cùng một lớp nhưng với các tham số khác nhau. Điều này giúp bạn tạo đối tượng theo các cách khác nhau.

Constructor không thể bị override:

* Constructor không thể bị override bởi lớp con vì constructor không phải là phương thức kế thừa.
* Override chỉ áp dụng cho các phương thức mà lớp con có thể ghi đè lại phương thức của lớp cha. Nhưng constructor là phương thức dùng để khởi tạo đối tượng của chính lớp đó, nên nó không bị override.

# **5. Trình tự khởi tạo đối tượng trong Java diễn ra như thế nào?**

### **1. Tải lớp vào bộ nhớ (Class Loading):**

* Trước khi tạo bất kỳ đối tượng nào, lớp của đối tượng phải được tải vào bộ nhớ.
* Nếu lớp chưa được tải, Java ClassLoader sẽ tải lớp đó vào bộ nhớ (bao gồm việc kiểm tra và xác minh mã bytecode).

### **2. Cấp phát bộ nhớ (Memory Allocation):**

* Sau khi lớp được nạp, hệ thống Java cấp phát vùng nhớ cho đối tượng trong heap. Vùng nhớ này chứa tất cả các biến instance của đối tượng, và ban đầu chúng sẽ có giá trị mặc định theo kiểu dữ liệu (ví dụ, số nguyên là 0, kiểu tham chiếu là null).

### **3. Khởi tạo biến tĩnh (Static Variable Initialization):**

* Nếu lớp chứa các biến static, chúng sẽ được khởi tạo trước khi bất kỳ constructor nào được gọi.
* **Khối static** cũng sẽ được thực thi trong quá trình này.
* Các biến static được khởi tạo chỉ **một lần** khi lớp được tải vào bộ nhớ.

### **4. Khởi tạo biến instance (Instance Variable Initialization):**

* Sau khi các biến static được khởi tạo (nếu có), các biến instance của đối tượng sẽ được khởi tạo.
* Ban đầu, các biến instance được gán các giá trị mặc định theo kiểu dữ liệu, sau đó chúng được khởi tạo bằng các giá trị mà bạn đã định nghĩa trong lớp hoặc thông qua constructor.
* Nếu có các **khối khởi tạo instance** (initializer blocks), chúng sẽ được thực thi trong bước này.

### **5. Gọi Constructor của lớp cha (Superclass Constructor Invocation):**

* Nếu đối tượng là một instance của một lớp con (subclass), trước khi khởi tạo các thuộc tính và phương thức của lớp con, constructor của lớp cha sẽ được gọi.
* Điều này đảm bảo rằng tất cả các thành phần được thừa kế từ lớp cha được khởi tạo trước.
* Nếu bạn không gọi constructor của lớp cha một cách rõ ràng, trình biên dịch sẽ tự động gọi constructor mặc định của lớp cha bằng từ khóa super().
* Quá trình này diễn ra theo thứ tự kế thừa từ lớp cha cao nhất (gần Object) đến lớp con.

### **6. Khởi tạo biến instance và khối khởi tạo của lớp con (Subclass Initialization):**

* Sau khi gọi constructor của lớp cha, các biến instance và các khối khởi tạo của lớp con sẽ được thực thi.
* Các khối khởi tạo instance sẽ được thực thi **trước** khi constructor của lớp con được gọi.

### **7. Gọi Constructor của lớp con (Subclass Constructor Execution):**

* Sau khi tất cả các khối khởi tạo instance đã được thực thi, constructor của lớp con sẽ được gọi. Đây là nơi bạn có thể khởi tạo các giá trị cụ thể cho các biến instance của đối tượng, truyền các tham số hoặc thực hiện các hành động cần thiết khi đối tượng được tạo.

### **8. Trả về đối tượng đã khởi tạo:**

* Sau khi hoàn thành quá trình khởi tạo, hệ thống Java sẽ trả về một tham chiếu tới đối tượng mới được khởi tạo. Đối tượng này bây giờ có thể được sử dụng trong chương trình.

# **6. Phân biệt Abstract Class và Interface trong Java**

#### **1. Abstract Class (Lớp trừu tượng)**

* **Định nghĩa**: Lớp trừu tượng là lớp không thể tạo đối tượng trực tiếp. Nó chứa các phương thức trừu tượng (không có thân hàm) và các phương thức cụ thể (có thân hàm).  
  **Đặc điểm chính:**
  + Có thể chứa cả phương thức **trừu tượng** và **cụ thể**.
  + Có thể có **biến instance**, **biến static**, và **constructor**.
  + Một lớp chỉ có thể **kế thừa** một lớp trừu tượng (vì Java không hỗ trợ đa kế thừa với lớp).
  + Các phương thức trừu tượng phải được **override** trong các lớp con.
* **Khi nào sử dụng**:
  + Dùng khi muốn chia sẻ một số phương thức cụ thể giữa các lớp con nhưng vẫn muốn các lớp con cung cấp triển khai riêng cho các phương thức trừu tượng.
  + Khi bạn có một "phần chung" nhưng một số phương thức yêu cầu lớp con triển khai theo cách riêng của chúng.

#### **2. Interface**

* **Định nghĩa**: Interface là một tập hợp các phương thức mà một lớp có thể thực thi (implement). Interface không chứa thân phương thức (cho đến Java 8 trở đi, có thể có phương thức default và static với thân hàm).  
  **Đặc điểm chính:**
  + Interface chỉ chứa các **phương thức trừu tượng** (trước Java 8) và các **hằng số** (biến static final).
  + Từ Java 8 trở đi, interface có thể chứa các phương thức **default** (với thân hàm) và **static**.
  + Một lớp có thể **kế thừa nhiều interface** (đa kế thừa với interface).
  + Các lớp triển khai phải **override** tất cả các phương thức của interface, trừ khi lớp đó là abstract.
* **Khi nào sử dụng**:
  + Khi cần xác định một **hợp đồng** giữa các lớp, mà không quan tâm đến cách thức triển khai chi tiết.
  + Khi cần sử dụng **đa kế thừa**, bạn có thể dùng interface để cho phép lớp thực thi nhiều interface cùng lúc.

|  |  |  |
| --- | --- | --- |
| Đặc điểm | Abstract Class | Interface |
| Triển khai | Có thể chứa phương thức trừu tượng và phương thức cụ thể. | Chỉ chứa phương thức trừu tượng (Java 7 trở về trước). Từ Java 8, có thể có phương thức cụ thể với default hoặc static. |
| Kế thừa | Một lớp chỉ có thể kế thừa một lớp trừu tượng. | Một lớp có thể thực thi nhiều interface (đa kế thừa). |
| Biến | Có thể chứa biến instance, biến static và hằng số. | Chỉ chứa hằng số (static final). |
| Constructor | Có constructor. | Không có constructor. |
| Từ khóa sử dụng | abstract class và extends. | interface và implements. |
| Tính chất kế thừa | Được dùng khi cần chia sẻ hành vi chung. | Được dùng khi chỉ cần quy định một hợp đồng/tiêu chuẩn. |
| Tốc độ | Abstract class thường nhanh hơn (vì không có overhead của nhiều interface). | Có thể chậm hơn do hỗ trợ đa kế thừa. |

### Trường hợp sử dụng cụ thể

Abstract Class: Dùng khi bạn muốn chia sẻ mã giữa các lớp có mối quan hệ chặt chẽ. Ví dụ: lớp Animal có các phương thức cụ thể như eat() và các phương thức trừu tượng như sound(), và các lớp con như Dog và Cat sẽ triển khai sound().  
Interface: Dùng khi các lớp không liên quan cần thực thi một số hành vi chung. Ví dụ: Flyable interface có thể được thực thi bởi cả Bird và Airplane, mặc dù chúng không liên quan đến nhau theo hướng đối tượng.

### **Nếu 2 interface hoặc 1 abstract và 1 interface có một phương thức cùng tên:**

Giả sử, nếu một lớp kế thừa từ một **abstract class** và thực thi một hoặc nhiều **interface** có **phương thức trùng tên** (có thể cùng hoặc khác kiểu trả về), thì điều gì sẽ xảy ra:

#### **Trường hợp 1: Trùng tên và cùng kiểu trả về**

* Nếu cả **interface** và **abstract class** đều có một phương thức cùng tên và cùng kiểu trả về, lớp con sẽ cần phải override phương thức đó. Điều này được yêu cầu vì:
* Nếu không override, Java sẽ không biết thực thi phương thức nào (từ interface hay abstract class), dẫn đến lỗi.

#### **Trường hợp 2: Trùng tên nhưng khác kiểu trả về**

* Nếu các phương thức trùng tên nhưng **khác kiểu trả về**, điều này sẽ gây ra lỗi biên dịch, vì một phương thức không thể được override với kiểu trả về khác nhau.
* Nếu **phương thức trùng tên** và **cùng kiểu trả về** từ abstract class và interface, lớp con phải override phương thức đó.
* Nếu **phương thức trùng tên** nhưng **khác kiểu trả về**, điều này sẽ gây lỗi biên dịch.

# **7. Thế nào là Overriding và Overloading**

### **1. Overriding (Ghi đè)**

**Định nghĩa**: Overriding xảy ra khi một lớp con định nghĩa lại một phương thức đã được định nghĩa trong lớp cha. Mục đích của overriding là để cung cấp một triển khai cụ thể cho phương thức trong lớp con.

#### **Đặc điểm chính:**

* **Cùng tên và cùng tham số**: Phương thức trong lớp con phải có cùng tên, cùng kiểu trả về (return type) và cùng danh sách tham số (parameter list) với phương thức trong lớp cha.
* **Phải là lớp con**: Chỉ xảy ra giữa lớp cha và lớp con (hoặc giữa các lớp kế thừa).
* **Tính đa hình**: Overriding cho phép thực hiện tính đa hình (polymorphism) trong Java. Khi một đối tượng của lớp con được xử lý qua tham chiếu của lớp cha, phương thức của lớp con sẽ được gọi.
* **Sử dụng từ khóa @Override**: Mặc dù không bắt buộc, nhưng thường được khuyến cáo sử dụng để làm rõ rằng bạn đang ghi đè một phương thức.

### **2. Overloading (Nạp chồng)**

**Định nghĩa**: Overloading xảy ra khi một lớp định nghĩa nhiều phương thức có cùng tên nhưng khác nhau về số lượng tham số, kiểu tham số hoặc cả hai. Mục đích của overloading là để cải thiện khả năng sử dụng và linh hoạt của phương thức.

#### **Đặc điểm chính:**

* **Cùng tên nhưng khác tham số**: Các phương thức phải có cùng tên nhưng khác nhau về kiểu dữ liệu hoặc số lượng tham số.
* **Cùng lớp hoặc khác lớp**: Overloading có thể xảy ra trong cùng một lớp hoặc giữa các lớp khác nhau (khi sử dụng kế thừa).
* **Kiểu trả về không liên quan**: Không thể chỉ dựa vào kiểu trả về để phân biệt giữa các phương thức, tức là, bạn không thể có hai phương thức cùng tên và cùng tham số nhưng khác kiểu trả về.

|  |  |  |
| --- | --- | --- |
| Đặc điểm | Overriding | Overloading |
| Khái niệm | Định nghĩa lại phương thức của lớp cha trong lớp con. | Nhiều phương thức cùng tên trong một lớp nhưng khác tham số. |
| Số lượng phương thức | Một phương thức trong lớp cha được ghi đè bởi một phương thức trong lớp con. | Nhiều phương thức trong cùng một lớp hoặc lớp khác với cùng tên. |
| Tham số | Phải cùng tên, cùng kiểu trả về và cùng tham số. | Cùng tên nhưng khác số lượng hoặc kiểu tham số. |
| Tính đa hình | Hỗ trợ đa hình (polymorphism). | Không hỗ trợ đa hình. |
| Sử dụng từ khóa | Sử dụng từ khóa @Override (khuyến cáo). | Không cần từ khóa. |

# **8. Một function có access modifier là private or static có thể overriding được không?**

* Phương thức có access modifier là private chỉ có thể được truy cập từ trong chính lớp đó. Vì vậy, lớp con không thể thấy được phương thức private của lớp cha, và do đó, không thể **overriding** phương thức này.
* Phương thức static thuộc về lớp, không thuộc về đối tượng của lớp. Vì vậy, các phương thức static không tham gia vào quá trình **dynamic binding** (liên kết động) và do đó, không thể bị overriding.

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

* **Kế thừa**: Lớp con có thể **sử dụng** phương thức final của lớp cha như bình thường.
* **Không thể overriding**: Lớp con **không thể cung cấp một triển khai mới** cho phương thức đó. Phương thức final không thể bị thay đổi hay ghi đè lại trong bất kỳ lớp con nào.

# **10. Sự khác biệt giữa kế thừa đơn và đa kế thừa? Java có hỗ trợ đa kế thừa không?**

* Trong kế thừa đơn, một lớp chỉ có thể kế thừa từ **một lớp cha duy nhất**. Điều này đảm bảo rằng một lớp con chỉ nhận các thuộc tính và phương thức từ một lớp cha, tránh sự phức tạp và mâu thuẫn trong kế thừa.
* Trong đa kế thừa, một lớp có thể kế thừa từ **nhiều lớp cha** cùng một lúc. Điều này có nghĩa là lớp con sẽ nhận các thuộc tính và phương thức từ nhiều lớp cha khác nhau. Tuy nhiên, điều này có thể dẫn đến một số vấn đề như **"diamond problem"**, trong đó một lớp con có thể nhận cùng một phương thức từ nhiều lớp cha và gây ra xung đột.

# **11. Phân biệt hai từ khóa This và Super**

### **Từ khóa this**

* **this** được sử dụng để tham chiếu đến **đối tượng hiện tại** của lớp.
* Dùng để phân biệt giữa các biến instance của đối tượng và các tham số của phương thức hay constructor khi chúng có cùng tên.
* this có thể được sử dụng trong các trường hợp:
  + **Tham chiếu đến các biến instance** trong lớp hiện tại.
  + **Gọi constructor khác** trong cùng một lớp.
  + **Truyền đối tượng hiện tại** làm tham số cho một phương thức hoặc constructor.
  + **Gọi phương thức của lớp hiện tại** từ trong lớp đó.

### **Từ khóa super**

* **super** được sử dụng để tham chiếu đến **lớp cha (superclass)** của lớp hiện tại.
* Dùng khi cần truy cập các thuộc tính, phương thức hoặc constructor của lớp cha mà đã bị lớp con ghi đè (overriding) hoặc để gọi constructor của lớp cha.
* super có thể được sử dụng trong các trường hợp:
  + **Gọi phương thức của lớp cha** bị lớp con ghi đè.
  + **Tham chiếu đến biến instance của lớp cha** nếu lớp con có biến cùng tên.
  + **Gọi constructor của lớp cha** từ constructor của lớp con.