Object Oriented Programming

[1. OOP là gì? 2](#_Toc475007234)

[2. Tại sao cần dùng OOP? 2](#_Toc475007235)

[3. Lợi ích khi dùng OOP 2](#_Toc475007236)

[4. Các đặc tính của OOP 3](#_Toc475007237)

[4.1. Tính đóng gói (encapsulation) 3](#_Toc475007238)

[4.2. Tính trừu tượng (abstraction) 3](#_Toc475007239)

[4.3. Tính kế thừa (inheritance) 3](#_Toc475007240)

[4.4. Tính đa hình (polymorphism) 3](#_Toc475007241)

[5. Một số khái niệm trong OOP 3](#_Toc475007242)

[5.1. Object 3](#_Toc475007243)

[5.2. Class 4](#_Toc475007244)

[5.3. Inheritance (kế thừa) 4](#_Toc475007245)

[5.4. Interface 4](#_Toc475007246)

[5.5. Package 5](#_Toc475007247)

[5.6. Modifiers 5](#_Toc475007248)

[5.7. Overloading và Overriding 7](#_Toc475007251)

[6. Abstract Class và Interface Class 7](#_Toc475007252)

1. OOP là gì?

Lập trình hướng đối tượng là lập trình hỗ trợ công nghệ đối tượng (OOP) giúp tăng năng xuất và đơn giản hóa công việc xây dựng phần mềm *(giảm các thao tác viết mã cho người lập trình)*.

Phương pháp lập trình hướng đối tượng lấy **dữ liệu làm nền tảng.**

Chương trình được thiết kế theo cách tiếp cận từ **dưới lên (bottom- up).**

Chương trình được chia thành các lớp đối tượng và dữ liệu được đóng gói, che giấu, bảo vệ.

**Ví dụ:**

* + Khi bạn xây dựng một căn nhà, dựa vào số tiền có được bạn sẽ lên kế hoạch xây dựng căn nhà cho mình. **→ chương trình.**
  + Bạn dự tính được lượng nguyên vật liệu (cát, đá gạch, sắt, xi măng,…). **→ đối tượng dữ liệu.**
  + Với lượng nguyên vật liệu đó bạn dùng cách gì cho chúng liên lạc nhau, chúng kết dính lại được để tạo nên một căn nhà. **→**  **Các hàm, thủ tục kết nối các đối tượng dữ liệu.**
    - Nguyên lý kết dính các nguyên vật liệu lại với nhau được hiểu theo nghĩa các chương trình con tác động lên dữ liệu làm thay đổi dữ liệu.

1. Tại sao cần dùng OOP?

Tăng hiệu suất làm việc, giảm thiểu các thao tác viết mã lệnh vì có thể tái sử dụng code.

Dễ bảo trì và sửa đổi mà không gây ảnh hưởng tới các chức năng khác của hệ thống.

Giảm thiểu sự lẫn lộn trong code.

Dễ dàng chia hệ thống thành từng phần nhỏ để giao cho các nhóm phát triển. Các lập trình viên không cần phải biết mọi khía cạnh cách chương trình làm việc.

1. Lợi ích khi dùng OOP

**Điểm mạnh:** là sự kế thừa, có thể sử dụng lại những gì đã có trước đó và phát triển thêm dựa trên những gì có sẵn. Code dễ quản lí và mở rộng, tốc độ thực thi nhanh, hạn chế sự rắc rối trong code.

**Điểm yếu:** Các hàm trong OOP phân biệt bằng tham số truyền vào nên khiến người dùng tốn nhiều thời gian để hiểu chức năng của nó.

1. Các đặc tính của OOP
   1. Tính đóng gói (encapsulation)

Có thể gói dữ liệu *(data, biến, trạng thái)* và mã chương trình *(code, phương thức)* thành một lớp *(class)* để dễ quản lí. Không cho người không có trách nhiệm truy cập trực tiếp tới data, nên thường ta sẽ che dấu data đi, chỉ hiển thị phương thức ra ngoài.

Trong OOP, tính đóng gói thể hiện phạm vi sử dụng của các class, phương thức và thuộc tính. Các phạm vi này thể hiện qua các access modifier: public, protected, private,…

**Ví dụ:** Phòng nhân sự họ sẽ **đóng gói (ẩn)** thông tin về các nhân viên. Bất kỳ yêu cầu nào về dữ liệu nhân viên *(cập nhật dữ liệu)* phải được thông qua bởi họ.

* 1. Tính trừu tượng (abstraction)

Khi viết chương trình theo phong cách hướng đối tượng, khi thiết kế các đối tượng, ta cần rút tỉa ra những đặc trưng của chúng, rồi trừu tượng hóa thành các interface và thiết kế xem chúng sẽ tương tác với nhau như thế nào.

* 1. Tính kế thừa (inheritance)

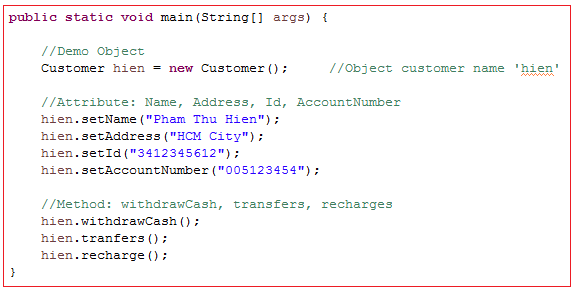
Kế thừa là việc tạo một lớp con thừa hưởng lại tất cả các đặc tính của lớp cha **ngoại trừ phương thức khởi tạo**. Cho phép kế thừa 1 interface, abstract class hoặc 1 class bình thường.

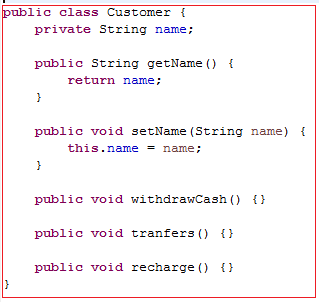
Khả năng truy cập: **private, public, final, protected.**

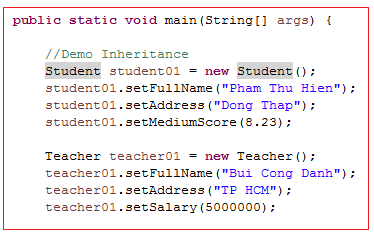
* 1. Tính đa hình (polymorphism)

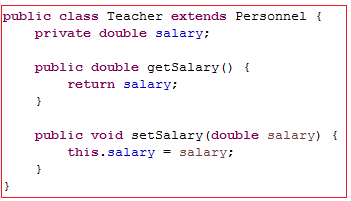
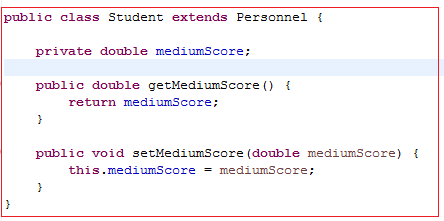
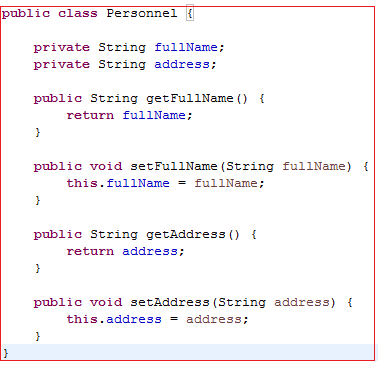
Đa hình là việc lớp B thừa kế các đặc tính từ lớp A nhưng có thêm một số cài đặt riêng.

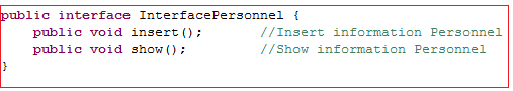
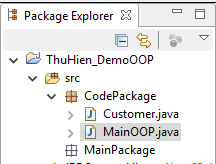
1. Một số khái niệm trong OOP
   1. Object

* Biểu diễn 1 đối tượng trong thế giới thực.
* Mỗi đối tượng được đặc trưng bởi các **thuộc tính (*attribute / Properties*)** và các **hành vi (*method*)**  riêng của nó.
* **Ví dụ:** Đối tượng **Khách hàng** của ngân hàng
  + *Thuộc tính:* họ tên, địa chỉ, cmnd, số tài khoản,…
  + *Hành vi:* rút tiền từ tài khoản, nạp tiền vào tài khoản, yêu cầu chuyển khoản.

* 1. Class
* Class mô tả cho những thực thể có chung tính chất và hành vi. 1 Class có thể chứa nhiều Object.
* 1 Class bao gồm:
  + *Biến thành viên (Field):* lưu trữ các thông tin mô tả về đối tượng
  + *Thuộc tính (Property):* định nghĩa chi tiết thao tác truy cập các biến thành viên
  + *Phương thức khởi tạo (Constructor)*
  + *Phương thức, hàm (Method)*
* **Ví dụ:** Lớp các đối tượng khách hàng
  1. Inheritance (kế thừa)
* Lớp cho phép lớp khác kế thừa được gọi là **lớp cơ sở (base - lớp cha)**. Lớp kế thừa gọi là **lớp dẫn xuất (derived - lớp con).**
* **Lớp dẫn xuất** sẽ **thừa hưởng được tất cả** các **phương thức** và **biến thành viên** của **lớp cơ sở**, thậm chí còn thừa hưởng cả các thành viên mà **lớp cơ sở** đã thừa hưởng trước đó, **ngoại trừ hàm khởi tạo**.
* **Lớp dẫn xuất** **không thể bỏ đi** các **thành phần** đã được khai báo trong **lớp cơ sở**.
* Các **hàm trong** **lớp dẫn xuất** **không được truy cập trực tiếp** đến các **thành viên** có mức độ truy cập là **private** trong **lớp cơ sở**.
* Dùng từ khóa **extends** khi kế thừa lớp cơ sở, dùng từ khóa **implements** khi kế thừa từ **interface.**
* **Ví dụ:** Lớp cha là Personnel class. Lớp con là Student, Teacher.

Lớp con sẽ kế thừa các thuộc tính của lớp cha như: fullName, address mà không cần khai báo lại.

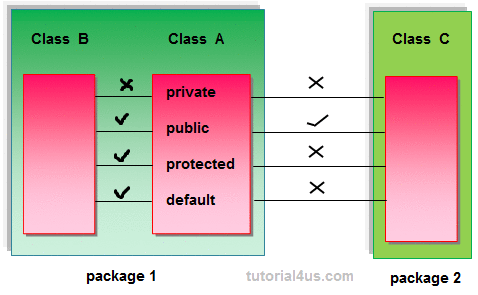


* 1. Interface
* **Interface** là tập các hàm khai báo sẵn mà không có cài đặt. Các lớp thực thi interface có nhiệm vụ cài đặt các hàm này .
* **Mục đích của việc cài đặt interface:** cho phép một lớp dẫn xuất có thể kế thừa thực thi **nhiều interface**. Điều này đã khắc phục được hạn chế của lớp abstract (một lớp dẫn xuất chỉ có thể thực thi một lớp abstract).
* Không thể khởi tạo 1 object của interface.
* Trong interface không có biến thành viên.
* **Ví dụ:**
  1. Package
* Tùy ngôn ngữ mà còn gọi là module, namespace.
* Package cho phép nhóm một tập hợp các class hoặc các interface có quan hệ với nhau để dễ dàng quản lý, bảo trì, phân phối…
  1. Modifiers

Có 2 loại modifier trong java là: **access modifiers** và **non-acccess modifiers.**

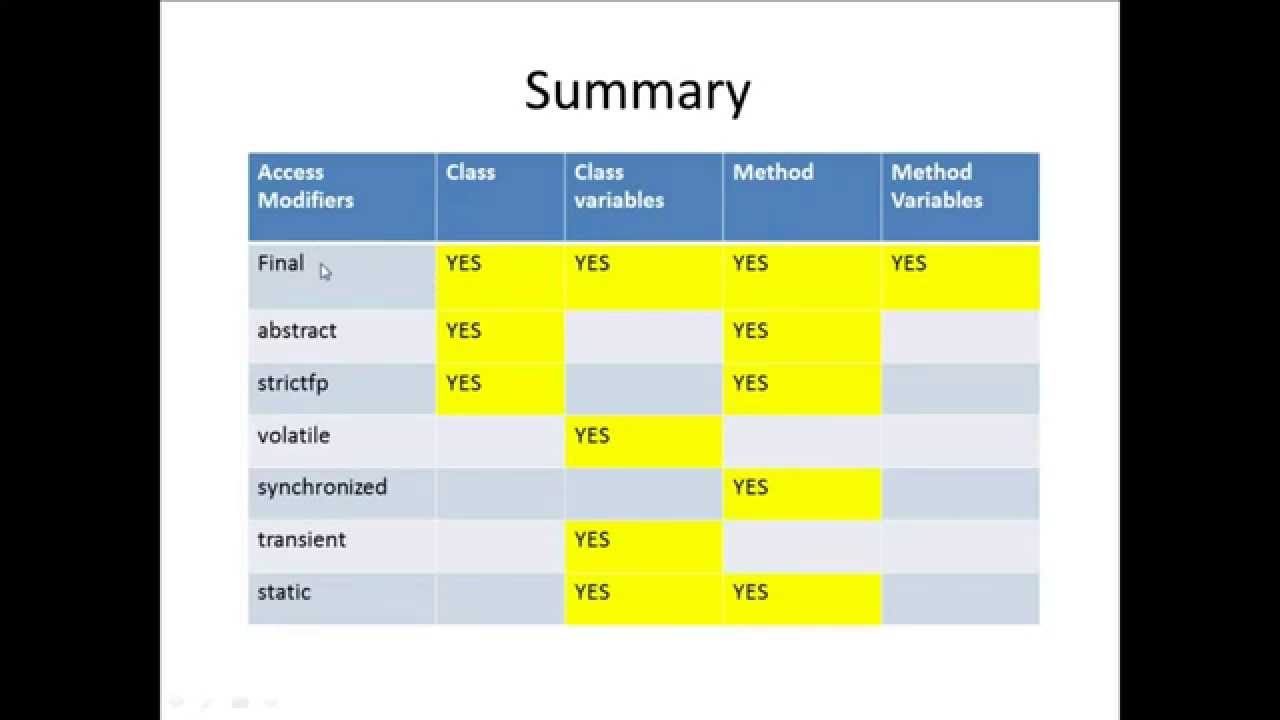
**Java Access Modifiers:** Dùng để xác định mức truy cập vào dữ liệu của các thuộc tính, phương thức, constructor hoặc class. Có 4 loại như sau:

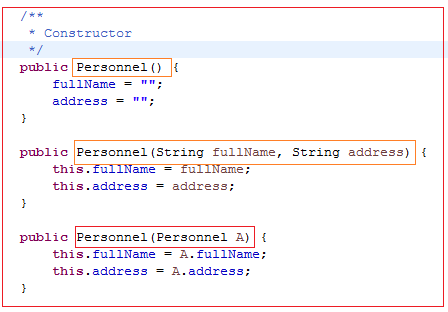
* + **public**: Có thể truy cập mọi nơi.
  + **protected**: Chỉ truy cập trong phạm vi package hoặc bên ngoài package nhưng phải thông qua tính kế thừa.
  + **default**: Chỉ truy cập trong phạm vi package.
  + **private**: Chỉ truy cập được trong phạm vi class.

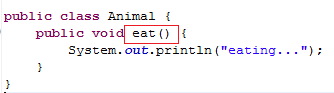
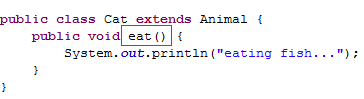


**Non-Access Modifiers:** Java cung cấp một số loại non-access modifier để thực hiện tính năng khác.

* + **Static modifier:** để tạo phương thức lớp và các biến.
  + **Final modifier:**
    - Chỉ được khởi tạo duy nhất một lần và có thể không bao giờ được gán giá trị mới.
    - Final để kết thúc sự thi hành của các lớp, các phương thức, các biến. Final modifier thường được sử dụng với static để tạo một hằng số trong class.
  + **Abstract modifier:** để tạo các lớp và các phương thức trừu tượng. Nếu một lớp được khai báo là abstract thì lớp này là để kế thừa.
  + **Synchronized modifier và Volatile modifier:**
    - Được sử dụng cho các Thread.
    - *Synchronized* được sử dụng để chỉ rằng một phương thức có thể được truy cập chỉ một Thread tại một thời điểm.
    - *Volatile modifier* được sử dụng để chỉ cho JVM biết rằng một thread đang truy cập biến đó phải luôn luôn sáp nhập bản sao biến private của riêng nó với bản sao master trong bộ nhớ.



* 1. Overloading và Overriding
* **Overloading (Nạp chồng):** Là các phương thức nằm trong cùng 1 lớp, có cùng tên nhưng khác nhau số tham số.
* **Overriding (Ghi đè):** Là khi có hai phương thức cùng tên và cùng tham số, một phương thức xuất hiện trong lớp cha và một phương thức nằm ở lớp con.



1. Abstract Class và Interface Class

|  |  |  |
| --- | --- | --- |
|  | **Abstract Class** | **Interface** |
| 1 | Có thể chứa phương thức abstract lẫn non-abstract. | Chỉ có thể chứa phương thức abstract. |
| 2 | Không hỗ trợ đa kế thừa. | Hỗ trợ **đa kế thừa.** |
| 3 | Abstract class có thể chứa biến **final, non-final, static và non-static.** | Chỉ cho phép chứa biến **static** và **final.** |
| 4 | Có thể chứa phương thức static, phương thức main và constructor. | Không thể chứa phương thức static, phương thức main và constructor. |
| 5 | Có thể kế thừa interface. | Không cho phép kế thừa abstract class. |
| 6 | Khai báo bằng cách sử dụng từ khóa abstract. | Khai báo bằng cách sử dụng từ khóa interface. |
| 7 | **Ví dụ:** public abstract class Shape {  public abstract void draw(); } | **Ví dụ:** public interface Drawable {  void draw(); } |

**Khi nào nên sử dụng Abstract Class**

* Khi nhiều object có chung các thuộc tính, và chung các phương thức.
* Những đối tượng thường xuyên thay đổi hành vi.

**Khi nào nên sử dụng Interface Class**

* Khi nhiều object không có chung thuộc tính nhưng có chung các phương thức.
* Khi cần sử dụng đa kế thừa.