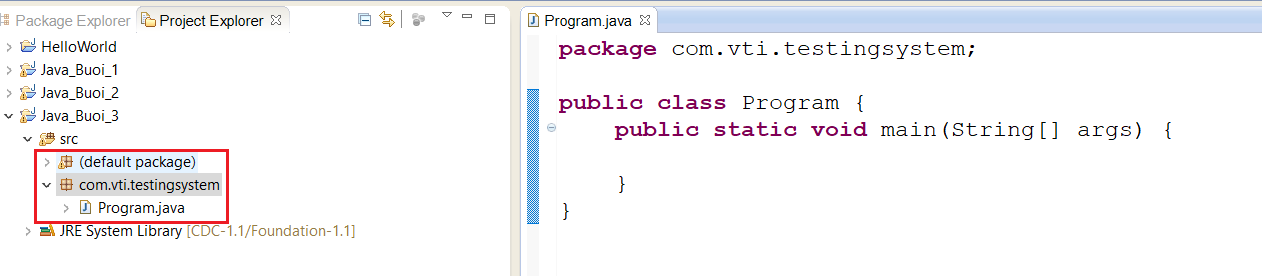
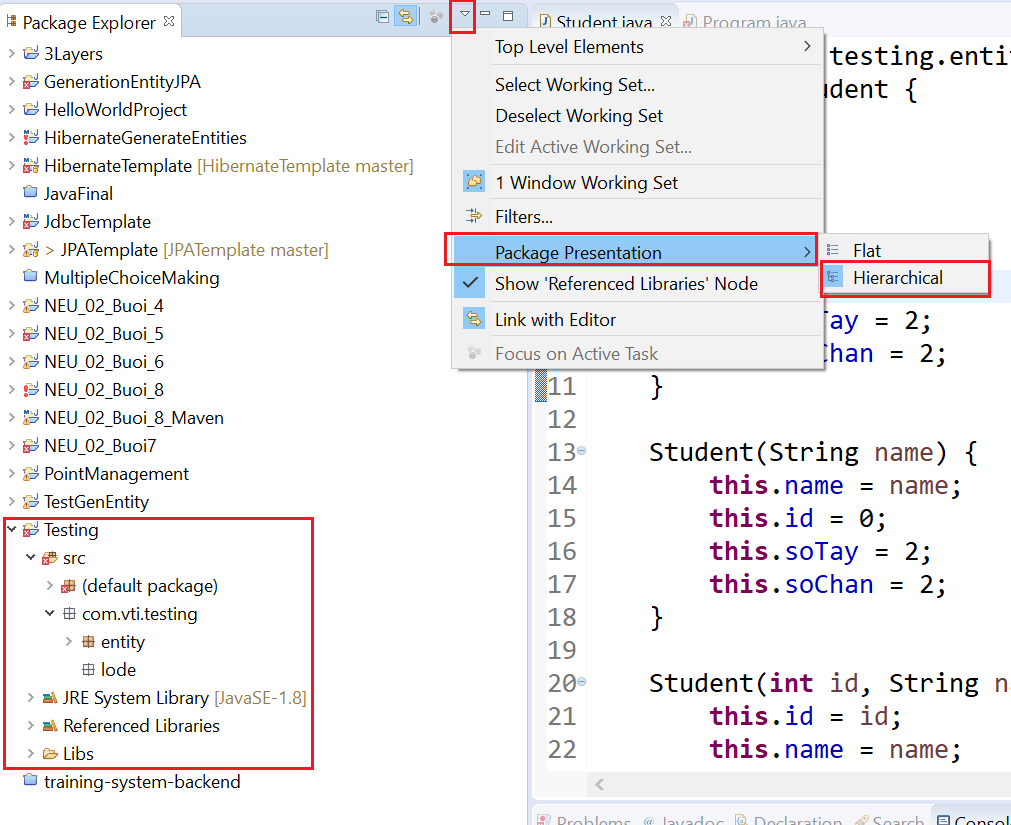
1. **Package**

Các object cùng chung 1 mục đích thì sẽ nằm cùng trong 1 package

Mục đích: để quản lý dễ hơn

****

****

*Hiển thị package theo dạng thư mục*

1. **Constructor**

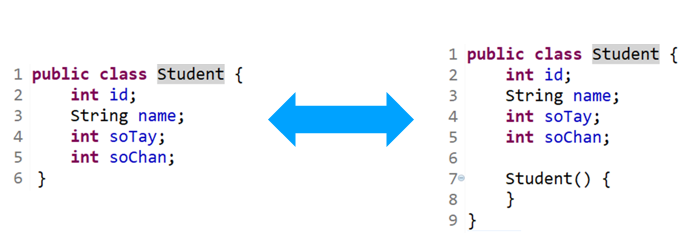
* Sử dụng để khởi tạo các property của object
* Được gọi khi khởi tạo object

VD: new Department()

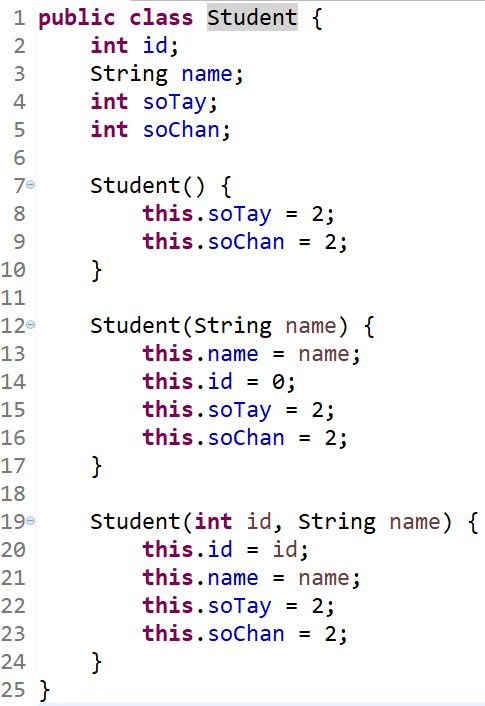
Object obj = new Object() 🡺 sẽ gọi method constructor của Object

* Mỗi 1 object luôn có 1 constructor default (không parameter)

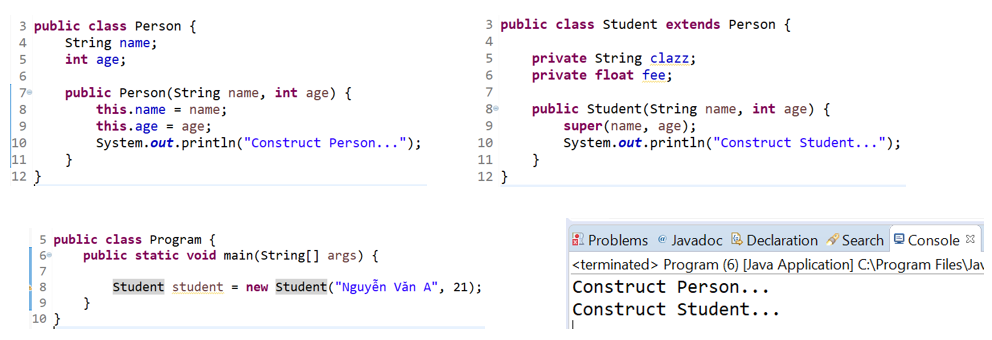
Khi khai báo constructor khác thì constructor default sẽ mất đi



* Có thể có nhiều Constructor trong 1 Object (overloading)



* Nếu có multi level inheritance thì sẽ run từ constructor của parent trước rồi mới tới child

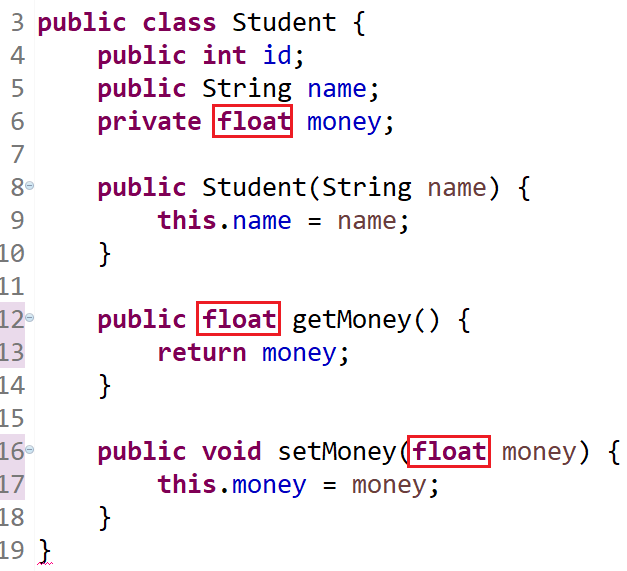
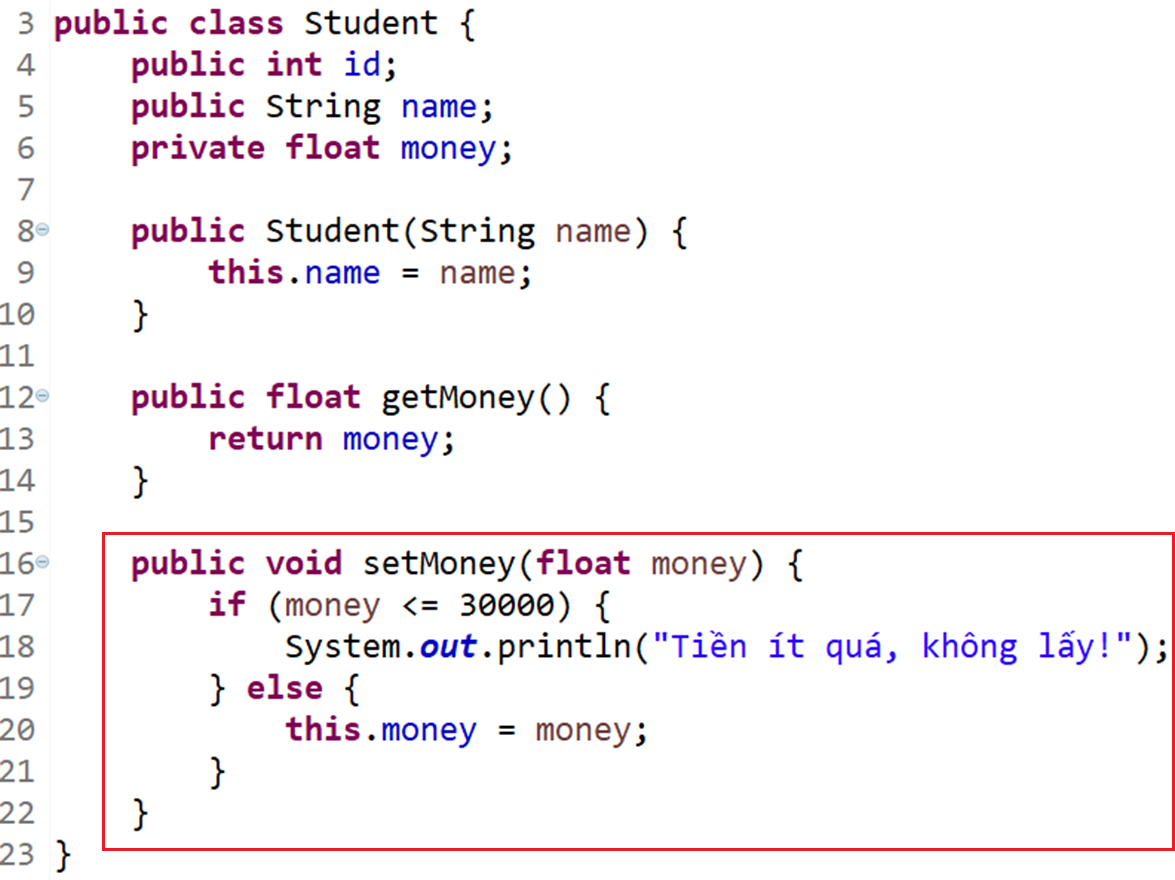


1. **Access modifier**

Áp dụng cho variable, method, constructor

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **Class** | **Package** | **Subclass** | **World** |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| default | Y | Y | N | N |
| private | Y | N | N | N |

* **Private** 
  + Chỉ truy cập được trong class
  + Muốn lấy/thay đổi giá trị của thuộc tính Private thì thông qua phương thức getter/setter (có thể chỉnh sửa logic ở đây)

* **Protected** 
  + Chỉ xảy ra trong quan hệ kế thừa (có thể khác package) và trong cùng 1 package
* **Default**
  + Không khai báo

VD: int a;

* + Dùng trong 1 package
* **Public**
  + Sử dụng ở mọi nơi trong project

1. **OOP**
   1. **Phân tích bài toán lập trình**

* Lập trình hướng đối tượng là
  + Phân tích bài toán thành các đối tượng

Mỗi đối tượng sẽ gồm các thuộc tính và phương thức.

* + Để giải quyết bài toán chúng ta xây dựng mối quan hệ giữa các đối tượng
  + Và cho các đối tượng thực thi phương thức của chính nó
* Đối tượng là 1 thực thể sống, hành vi(method) của mình, đặc điểm của mình( thuộc tính)
  1. **Xây dựng các mối quan hệ giữa các đối tượng**
* **Is-a**

Kế thừa hoặc đa hình thể hiện cho is-a

VD: A is-a B // Tức là Object A extends (implement) Object B

* **Has-a (phụ thuộc)**

Nếu đối tượng A phụ thuộc vào đối tượng B thì đối tượng B trở thành thuộc tính đối tượng A

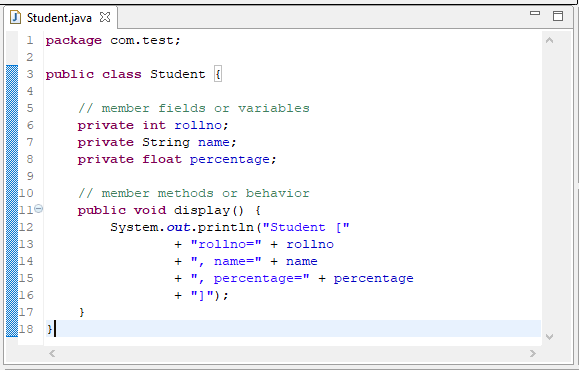
VD: A Has-a B // Tức là Object B được khai báo trong Object A

* **Song Song**

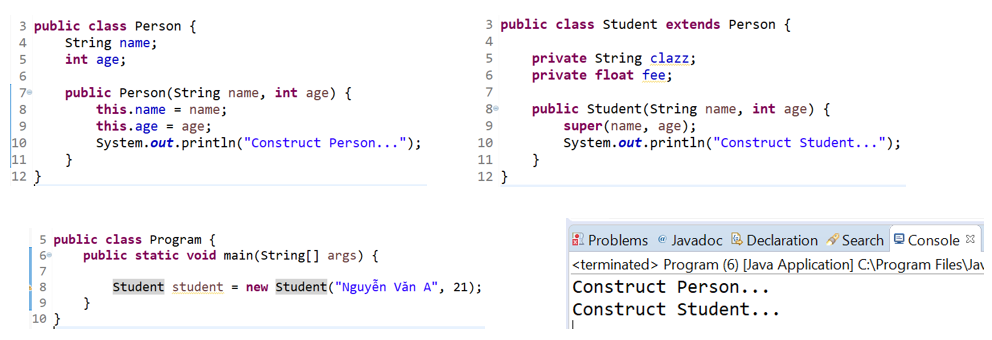
Xảy ra khi các đối tượng không liên quan gì tới nhau cả nhưng lại muốn lấy thông tin của nhau thì phải qua 1 đối tượng trung gian

1. **Các tính chất của OOP**

* **Encapsulation** (Tính đóng gói,che giấu thông tin):
  + Thể hiện qua access modifier **Private**
  + Không cho phép người sử dụng các đối tượng thay đổi trực tiếp trạng thái của đối tượng. Chỉ có các phương thức **getter/setter** của đối tượng đó mới cho phép thay đổi trạng thái của nó. 🡺 bảo đảm tính toàn vẹn của đối tượng



* **Inheritance** (Kế thừa)
  + Thể hiện qua **Extend**
  + Java không hỗ trợ multi inheritance, mà chỉ có multi level inheritance (chỉ có thể extend được 1 class cha)
  + Trong bài toán lập trình hướng đối tượng khi có 2 hoặc nhiều đối tượng giống nhau về bản chất (cùng chung 1 số thuộc tính và phương thức). Khi đó nảy sinh mối quan hệ kế thừa từ 1 đối tượng cha sẽ bao gồm thuộc tính và phương thức chung của các đối tượng còn lại.
* Những thuộc tính chung ở Super Class thì Sub Class **không được** viết vào
* Các class con thừa kế tất cả các phương thức và thuộc tính của lớp cha cho phép
* Có thể gọi tới các thuộc tính và method của Super Class bằng keyword super.method();
  + Khi khởi tạo thì sẽ khởi tạo từ constructor của cha rồi tới constructor của con



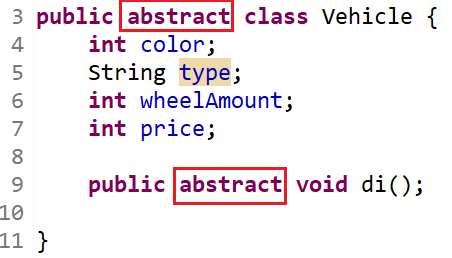
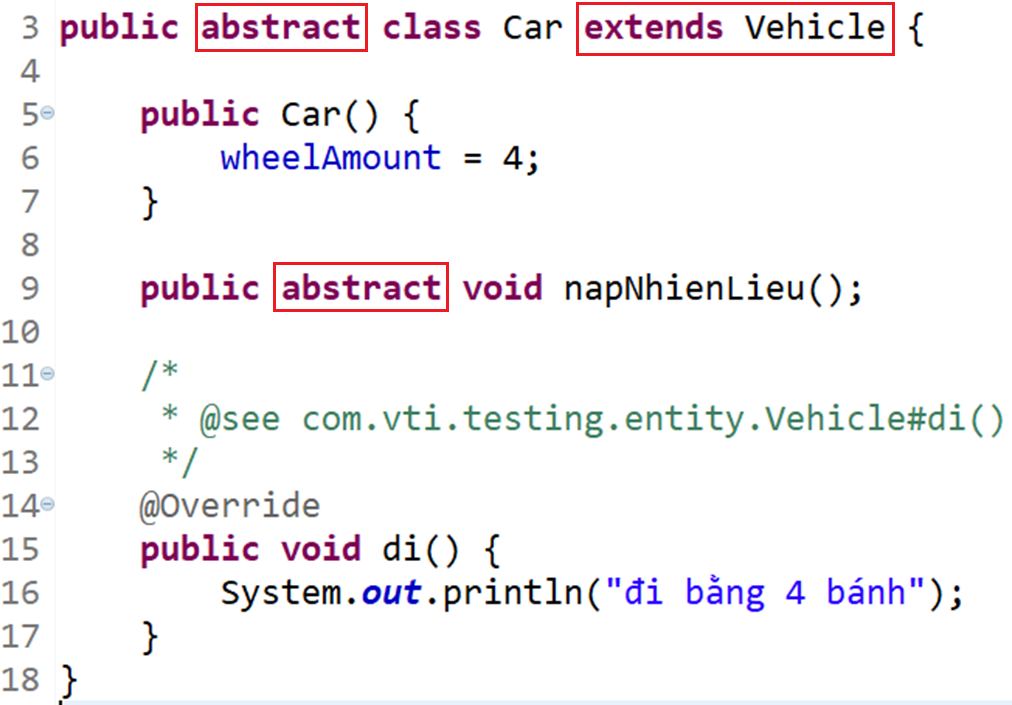
* **Abstraction** (Tính trìu tượng hóa)
  + Thể hiện qua **Abstract và interface**
  + Việc bỏ qua các đặc điểm riêng của các đối tượng để tập chung làm nổi bật các đặc điểm chung của các đối tượng
* Sau khi áp dụng trìu tượng hóa chúng ta có
  + - Mối quan hệ kế thừa giữa các đối tượng
    - Danh sách các đối tượng duy nhất mỗi đối tượng
  + Bao gồm có
* **Abstract class**
  + Về bản chất vẫn là 1 class 🡺 có các tính chất của 1 class bình thường nhưng không thể khởi tạo được instance 🡺 chỉ có class con cuối cùng mới khởi tạo được
  + Dùng để quản lý các object giống nhau về bản chất (có chung property và 1 số method)
  + Nếu 1 class có ít nhất 1 abstract method thì class đó phải là abstract
  + Abstract có thể không có abstract method nào
  + Không thể khai báo final
* **Abstract method**
  + Là method chưa hoàn thành
  + Class con cuối cùng thì tất cả các abstract method đều phải được implement (override)
* **Interface**
  + Không phải là object, chỉ quan tâm tới method
  + Chỉ quản lý các method, chia các nhóm method cho dễ quản lý
  + Dùng để quản lý các object khác nhau về bản chất nhưng lại giống nhau về 1 số method (kiểu object có chung mục đích sử dụng)
  + Default khai báo là abstract **interface**

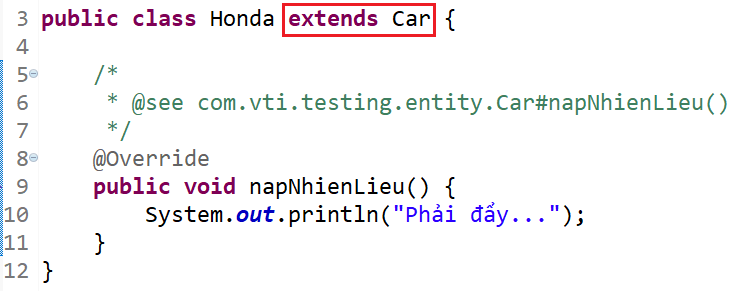
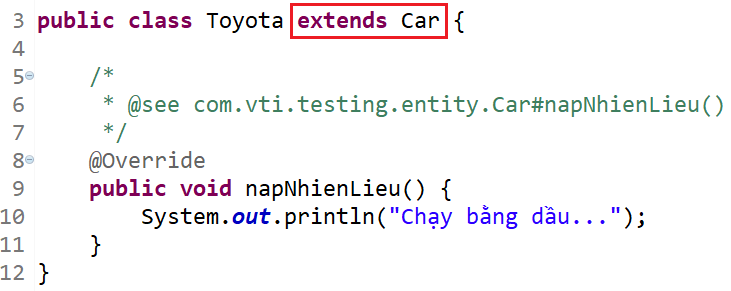
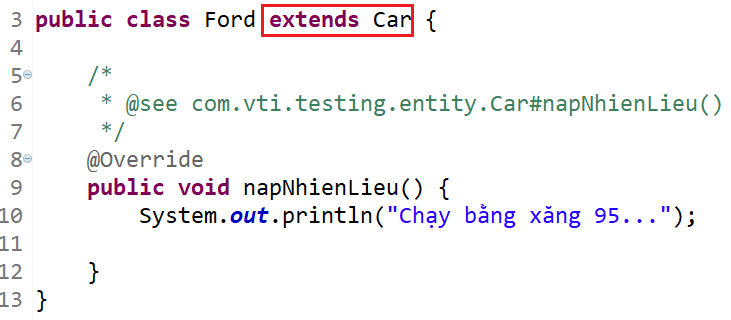


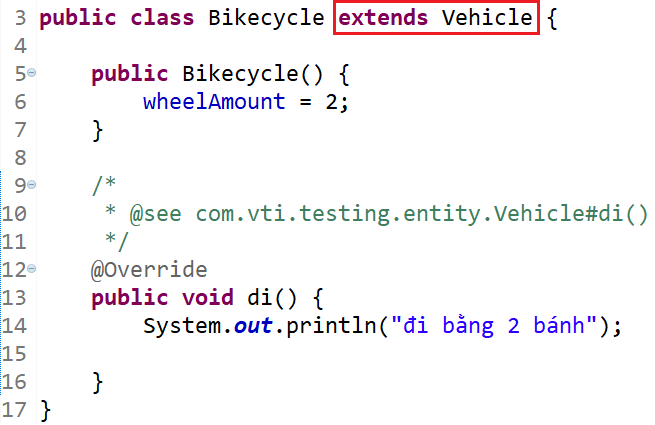
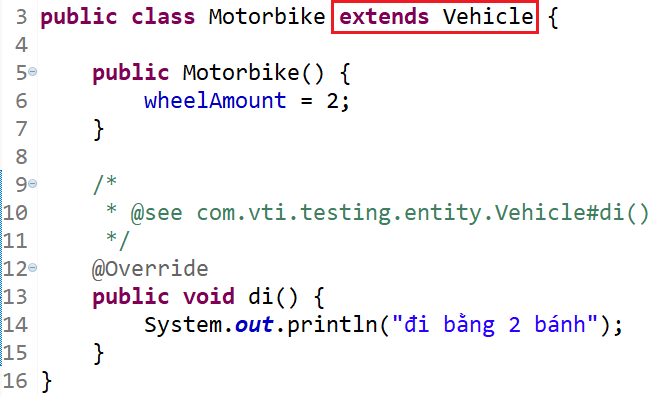
* + Method được khai báo trong interface là viết tắt của **public abstract** method
  + Biến khai báo trong interface là constant (public static final)
  + Có thể implement được nhiều interface
  + **Phân biệt abstract class & interface**

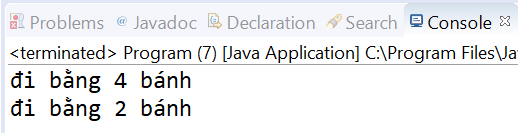
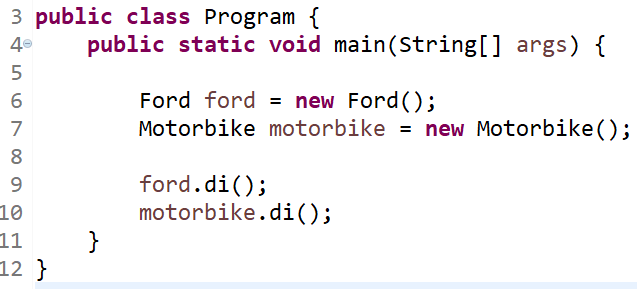
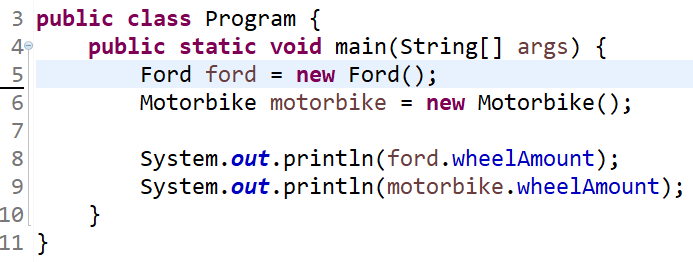
|  |  |
| --- | --- |
| **Interface** | **Abstract class** |
| Hỗ trợ đa kế thừa  (1 lớp implements từ nhiều interface ) | Không hỗ trợ đa kế thừa |
| Không chứa các property của Object  Biến được khai báo trong interface là Constant | Chứa data member |
| Không có phương thức khởi tạo | Có phương thức khởi tạo |
| Mức truy cập public | Tất cả |

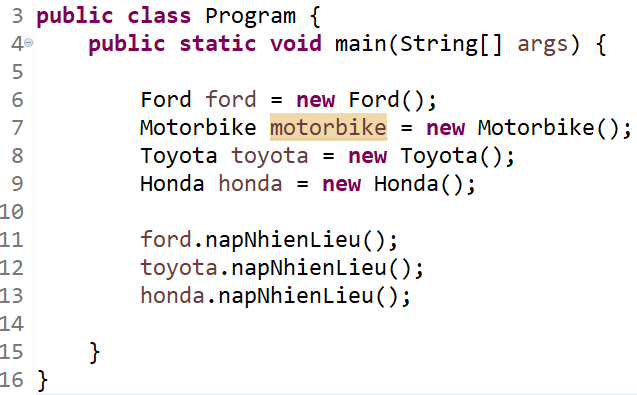
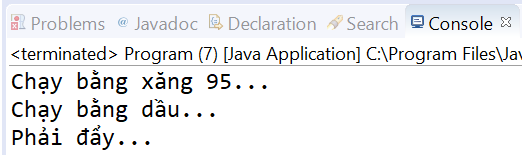
**VD1: abstract class**

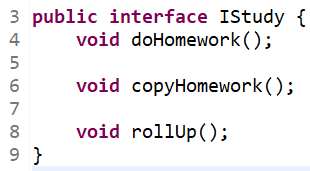
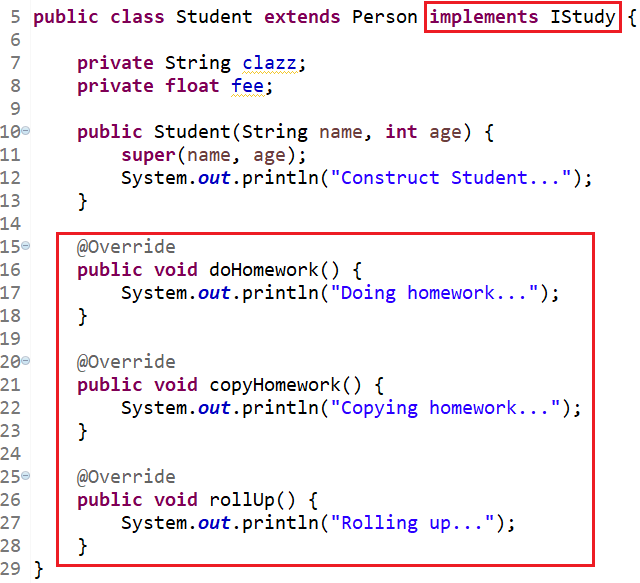




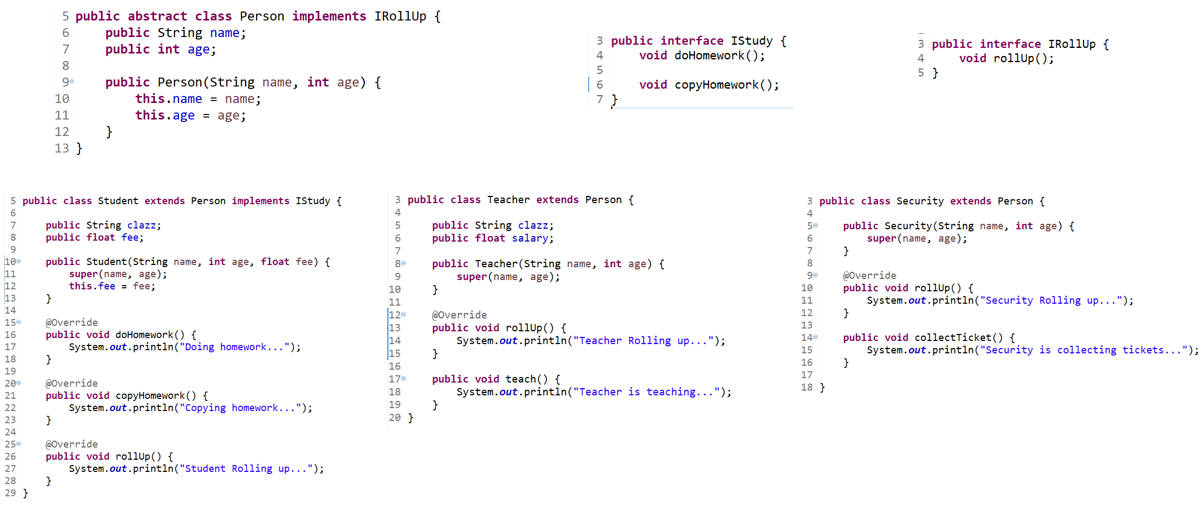
**VD2: interface**

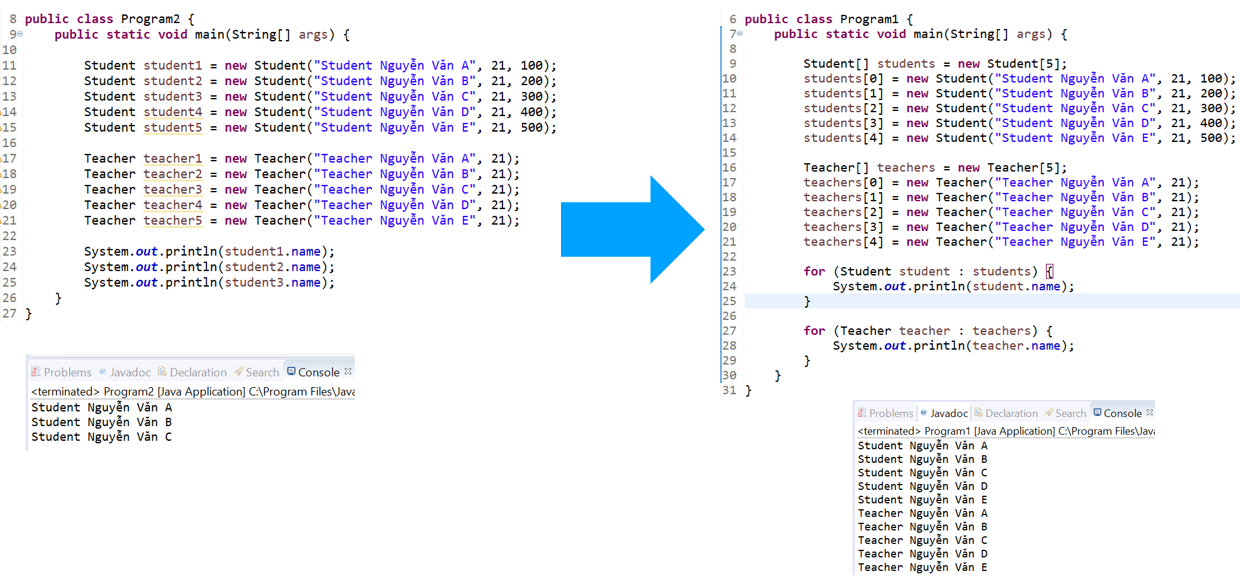
 

* **Polymorphism** (Đa hình)
  + Thể hiện qua **Override và Overload**
  + Quy về thằng cha để quản lý, khi cần thì có thể ép kiểu dữ liệu về thằng con

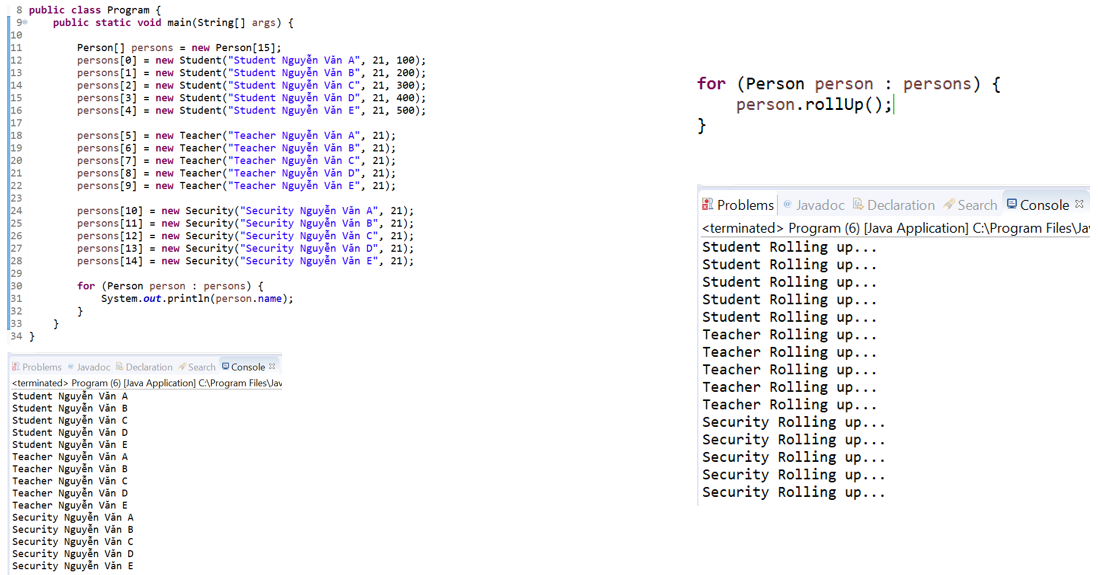
Khi kế thừa xảy ra, thay vì quản lý danh sách các đối tượng con, chúng ta quản lý danh sách đối tượng cha. Mỗi đối tượng cha sẽ là đối tượng con bất kì 🡺 khi cần ép kiểu thì phải check kiểu dữ liệu bằng keyword **instanceof**

VD1: Polymorphism





*Nếu có nhiều object 🡺 sẽ quản lý bằng nhiều array*

**

*Nếu array có nhiều datatype khác nhau thì sẽ quy về polymorphism để quản lý*

VD2: instanceof

TuGiac tuGiac1 = new HinhVuong(dinhA, 20);

// Kiểm tra hình hài lớp con

If(tuGiac1 **instanceof** HinhVuong){

// Ép kiểu tứ giác về hình vuông để gọi những phương thức riêng của Hình Vuông

((HinhVuong) tuGiac).phuongThucLopCon();

}

Chú ý: Khi cast Object từ Sub 🡪 Super Class thì không cần ép kiểu

* + **Override** (ghi đè phương thức)

mô tả việc ghi lại, ghi đè hoặc viết lại phương thức đã được xây dựng ở class cha trong class con

* Class con có thể giữ nguyên các method đã đc xây dựng trong class cha và thêm các logic khác vào

Sử dụng super.phuongThucCha();

// gọi phương thức đã đc xây dựng ở lớp cha

* Hoặc class con viết mới lại hoàn toàn method
* Runtime
* **Overload** (nạp chồng phương thức)

Là phương thức có cùng tên nhưng khác tham số truyền vào (có thể khác số lượng tham số hoặc khác datatype)

* Compile time
* **So sánh Overload & Override**

|  |  |  |
| --- | --- | --- |
|  | **Overriding** | **Overloading** |
| Name method | Phải giống | Phải giống |
| List Param | Phải giống | Phải khác |
| Datatype return | Giống hoặc là subtype của phương thức trong lớp cha | Có thể khác |
| Access Modifier | Giống hoặc mở rộng hơn | Có thể khác |
| Ngữ cảnh cài đặt | Chỉ có thể ghi đè trong lớp con | Có thể nạp chồng trong lớp con hoặc chính lớp cha |
| Sử dụng Super | Có | không |
|  | Runtime | Compiletime |

* **Reference**