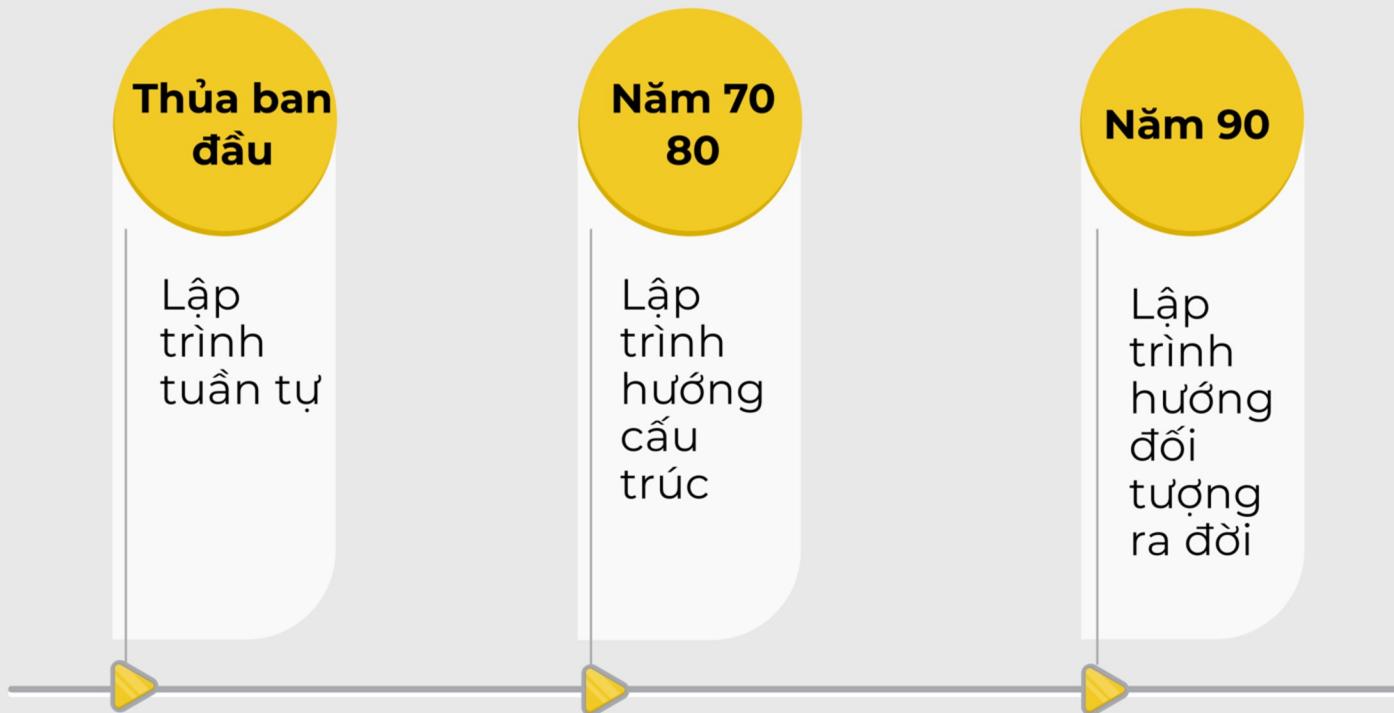


# Java Core #2

Lục Ngọc

# OOP

# Lịch sử phát triển



# Object Oriented Programming (OOP)

Đối  
tượng

Java

Python

C++

Ruby

Lớp

Khi viết chương trình theo phương pháp hướng đối tượng ta  
phải trả lời các câu hỏi:

Chương trình liên quan tới những lớp đối tượng nào?

Mỗi đối tượng cần có những dữ liệu và thao tác nào?

Các đối tượng quan hệ với nhau như thế nào trong  
một chương trình



# Các nguyên lý cơ bản của OOP

Tính đóng gói

Tính đa hình

OOP

Tính kế thừa

Tính trừu tượng



## Ưu điểm của OOP

Tính đóng gói làm giới hạn phạm vi sử dụng của các biến, nhờ đó việc quản lý giá trị của biến dễ dàng hơn, việc sử dụng mã an toàn hơn.

Phương pháp này làm cho tốc độ phát triển các chương trình mới nhanh hơn vì mã được tái sử dụng và cải tiến dễ dàng, uyển chuyển.

Phương pháp này tiến hành tiến trình phân tích, thiết kế chương trình thông qua việc xây dựng các đối tượng có sự tương hợp với các đối tượng thực tế.



## Nhược điểm của OOP

Các chương trình hướng đối tượng có xu hướng chậm hơn và sử dụng nhiều bộ nhớ

Quá khái quát

Các chương trình được xây dựng theo mô hình này có thể mất nhiều thời gian hơn

# Lớp và đối tượng



# Đối tượng

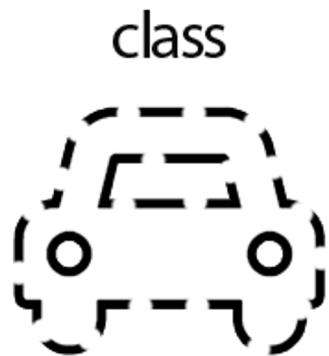
Đối tượng là một thực thể mang tính vật lý





# Lớp

Lớp là tập hợp các đối tượng có cùng trạng thái, hành vi hay là một nhóm các đối tượng có các thuộc tính chung.



Car



objects





# Lớp

Một lớp trong java có thể chứa:

Thành viên dữ liệu

Constructor

Phương thức

Khối lệnh

Lớp và interface



# Tạo lớp

Cú pháp:

```
<access modifiers> class <Tên class> {  
}
```

```
public class Person {  
    public String name;  
    public int age; }  
Các thuộc tính  
  
public void eat(){  
    System.out.println("Method eat() is called!!!");  
}  
  
public void study(String subject){  
    System.out.println("Method study() is called!!!");  
    System.out.println(subject);  
}  
}  
  
Phương thức  
không có tham số  
  
Phương thức  
có tham số
```



# Tạo đối tượng

Cú pháp khởi tạo đối tượng:

```
<Tên class> <Tên biến tham chiếu> = new <Tên class>();
```

Gán giá trị cho thuộc tính:

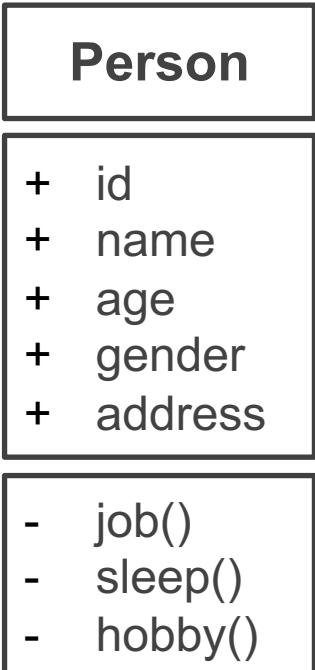
```
<Tên biến tham chiếu>.<Thuộc tính> = <Giá trị>;
```

Gọi phương thức:

```
<Tên biến tham chiếu>.<Tên phương thức>();
```

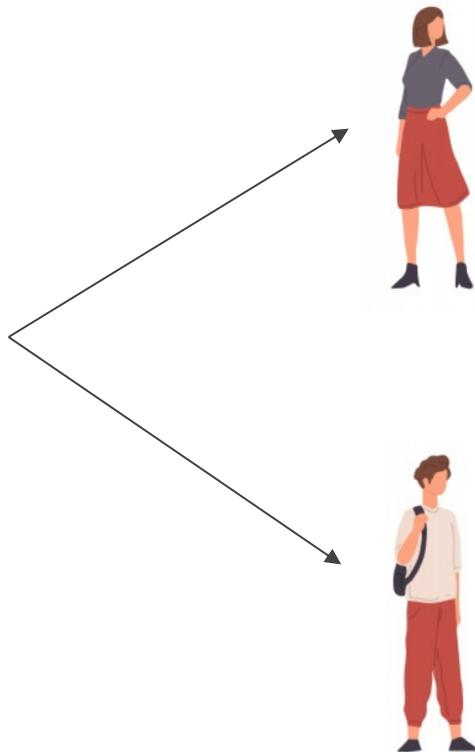
```
public class App {  
    public static void main(String[] args) {  
        Person person = new Person(); → Tạo đối tượng person  
        person.name = "Ngoc"; } } Gán giá trị cho thuộc tính  
        person.age = 25; }  
        System.out.println(person.name+", "+ person.age);  
  
        person.eat(); } } Gọi tới 2 phương thức  
        person.study("English"); }
```

Class



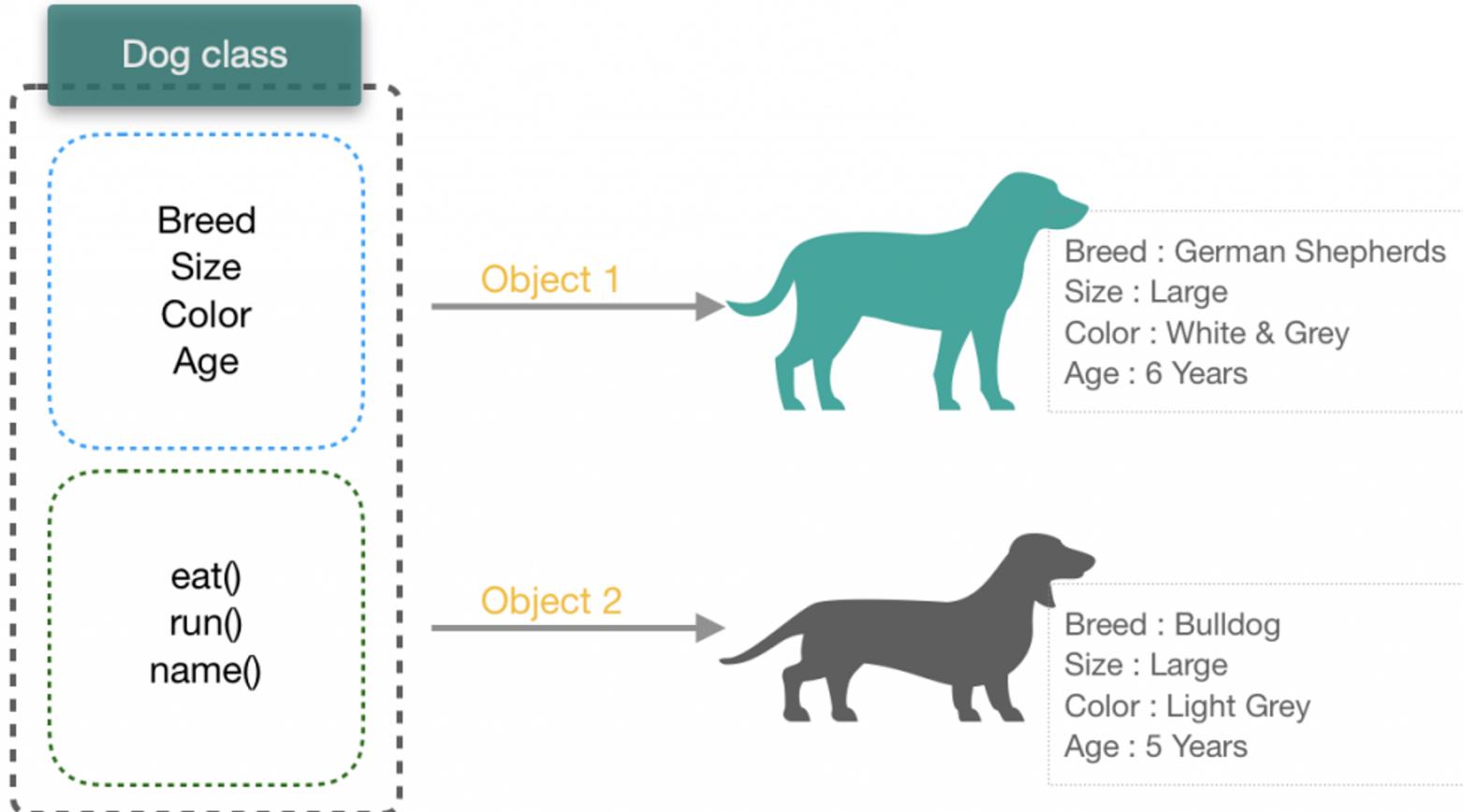
Data  
Members

Methods



Name: Jane  
Age: 25  
Address: HN  
Gender: female

Name: John  
Age: 30  
Address: HCM  
Gender: male



# DOG

breed
size
age
color
eat() sleep() run()

Breed: Neapolitan Mastiff  
Size: Large  
Age: 5 year  
Color: black



Breed: Maltese  
Size: Small  
Age: 2 year  
Color: white



Breed: Chow chow  
Size: Midium  
Age: 3 year  
Color: Brown



# Car

- + model
  - + color
  - + brand
- 
- speed()
  - size()
  - Brand()



Model: Ertiga  
Color : Red  
Brand: Maruti



Model: XUV 500  
Color : Black  
Brand: Mahindra



Model: CX5  
Color : White  
Brand: Mazda



# Constructor

Constructor trong java là một phương thức đặc biệt được sử dụng để khởi tạo các đối tượng. Constructor được gọi khi một đối tượng của một lớp được tạo. Nó có thể được sử dụng để đặt các giá trị ban đầu cho các thuộc tính của đối tượng.

Quy tắc chính của các constructor là chúng có cùng tên như lớp đó

Có 2 loại constructor:

### **Constructor mặc định**

Là constructor không có tham số, nhằm mục đích cung cấp các giá trị mặc định cho các đối tượng như 0, null, ... tùy thuộc vào kiểu dữ liệu.

### **Constructor có tham số**

Được sử dụng để cung cấp các giá trị khác nhau cho các đối tượng riêng biệt



# Constructor

Cú pháp:

```
<Access modifies> <Tên lớp>(<Tham số truyền vào>) {  
}
```

```
public class Person {  
    public String name;  
    public int age;
```

```
public Person() {  
}
```

Constructor mặc định

```
public Person(String name, int age) {  
    this.name = name;  
    this.age = age;  
}
```

Constructor có tham số



# Từ khóa static

Từ khoá static trong java được sử dụng để quản lý bộ nhớ. Chúng ta có thể áp dụng từ khoá static với các biến, các phương thức, khối và các lớp được lặp





# Biến static

Khi khai báo một biến là static, thì biến đó là biến tĩnh hay biến static. Biến static là biến mà ta có thể sử dụng mà không cần khởi tạo đối tượng. Biến static được sử dụng để tham chiếu thuộc tính chung của tất cả đối tượng.

Ví dụ: Tên trường học của các sinh viên

```
public static String school = "Techmaster";
```



# Phương thức static

Nếu bạn áp dụng từ khoá static với bất cứ phương thức nào, thì phương thức đó được gọi là phương thức static. Một phương thức static thuộc lớp chứ không phải đối tượng của lớp.

Phương thức static có thể truy cập thành viên dữ liệu static và có thể thay đổi giá trị của nó. Tuy nhiên, phương thức static không thể sử dụng biến non-static hoặc gọi trực tiếp phương thức non-static.



# Phương thức static

```
public static void change(){  
    school = "CNTT";  
}
```

Person.change();

Tên lớp



# Khối static

Khối static trong java được sử dụng để khởi tạo thành viên dữ liệu static. Nó được thực thi trước phương thức main tại lúc tải lớp

```
static {  
    System.out.println("Khối static: hello !");  
}
```



# Từ khóa this

Từ khóa this để cập tới một đối tượng hiện tại trong một phương thức hoặc constructor.

Thông thường, từ khóa this nhằm loại bỏ sự nhầm lẫn giữa các thuộc tính lớp và các tham số có cùng tên. Ngoài ra từ khóa this còn được sử dụng để:

Gọi constructor của lớp hiện tại

Gọi phương thức của lớp hiện tại

Trả về đối tượng của lớp hiện tại

Truyền một đối số trong lệnh gọi phương thức

Truyền một đối số trong lời gọi constructor

```
public class Person {  
    public String name;  
    public int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
public class App {  
    public static void main(String[] args) {  
        Person person = new Person("Ngoc", 25);  
        System.out.println(person.name+", "+ person.age);  
    }  
}
```

Viết chương trình quản lý học viên của Techmaster, thực hiện các công việc sau:

Tạo class Student chứa các thuộc tính: id, tên, điểm lý thuyết, điểm thực hành, trường

Tạo phương thức để nhập thông tin các học viên

Tạo phương thức để tính điểm trung bình (biết điểm trung bình = (điểm lý thuyết + điểm thực hành)/2)

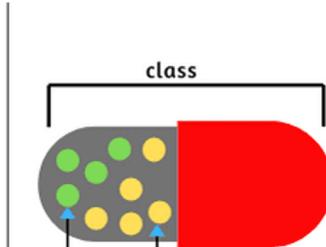
Tạo phương thức để in thông tin ra màn hình

# Tính đóng gói (Encapsulation)

# Tính đóng gói

Tính đóng gói là một trong 4 nguyên lý cơ bản của OOP  
Đóng gói trong Java là một cơ chế gói biến và phương thức  
lại với nhau thành một đơn vị duy nhất  
Với tính đóng gói, các biến của một lớp sẽ bị ẩn khỏi các  
lớp khác và chỉ có thể truy cập thông qua các phương thức  
của lớp hiện tại của chúng

```
class
{
    data members
    +
    methods (behavior)
}
```



Variables  
Methods

Để đạt được tính đóng gói trong Java, ta cần phải:

Khai báo các biến của một lớp là private  
Cung cấp các phương thức setter và getter để sửa đổi và xem các giá trị của biến

Các phương  
thức setter  
và getter

```
public class Person {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

Tính chất đóng gói có những đặc điểm sau:

Tạo ra cơ chế ngăn ngừa việc gọi phương thức của lớp này hay truy xuất dữ liệu của đối tượng thuộc về lớp khác  
Dữ liệu riêng của mỗi đối tượng được bảo vệ khỏi sự truy xuất không hợp lệ từ bên ngoài

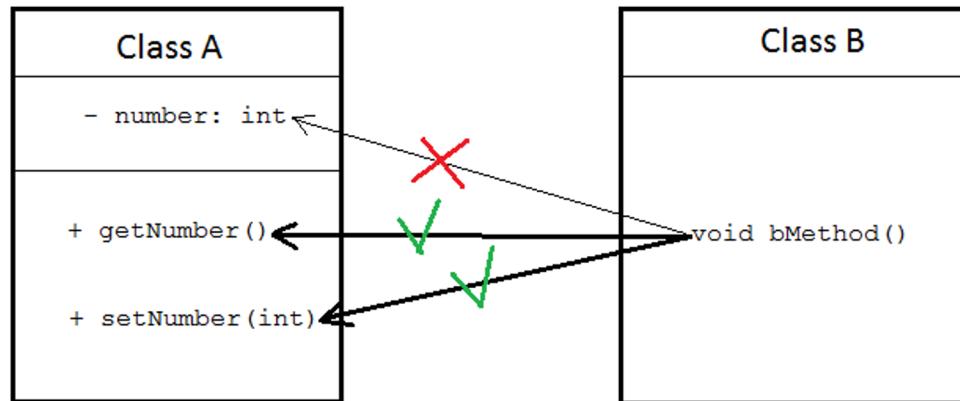
Người lập trình có thể dựa vào cơ chế này để ngăn ngừa sự gán giá trị không hợp lệ vào thành phần dữ liệu của mỗi đối tượng

Cho phép thay đổi cấu trúc bên trong của mỗi lớp mà không làm ảnh hưởng đến những lớp bên ngoài có sử dụng lớp đó



# Getter & Setter

Getter và Setter là hai phương thức sử dụng để lấy ra hoặc cập nhật giá trị thuộc tính, đặc biệt dành cho các thuộc tính ở phạm vi private





# Getter

Phương thức Getter là phương thức truy cập vào thuộc tính của đối tượng và trả về các thuộc tính của đối tượng

Cú pháp:

```
public <Kiểu dữ liệu trả về> get<Tên thuộc tính>() {  
    return <Tên thuộc tính>;  
}
```

Ví dụ:

```
public String getName() {  
    return name;  
}
```

```
public int getAge() {  
    return age;  
}
```

```
System.out.println(person.getName() + ", " + person.getAge());
```



# Setter

Phương thức Setter là phương thức truy cập vào thuộc tính của đối tượng và gán giá trị cho các thuộc tính của đối tượng đó

Cú pháp :

```
public void set<Tên thuộc tính>(<Tham số giá trị mới>) {  
    this.<Tên thuộc tính> = <Tham số giá trị mới>;  
}
```

Ví dụ:

```
public void setName(String name) {  
    this.name = name;  
}  
  
public void setAge(int age) {  
    this.age = age;  
}
```

```
person.setName("Ngoc");  
person.setAge(25);
```



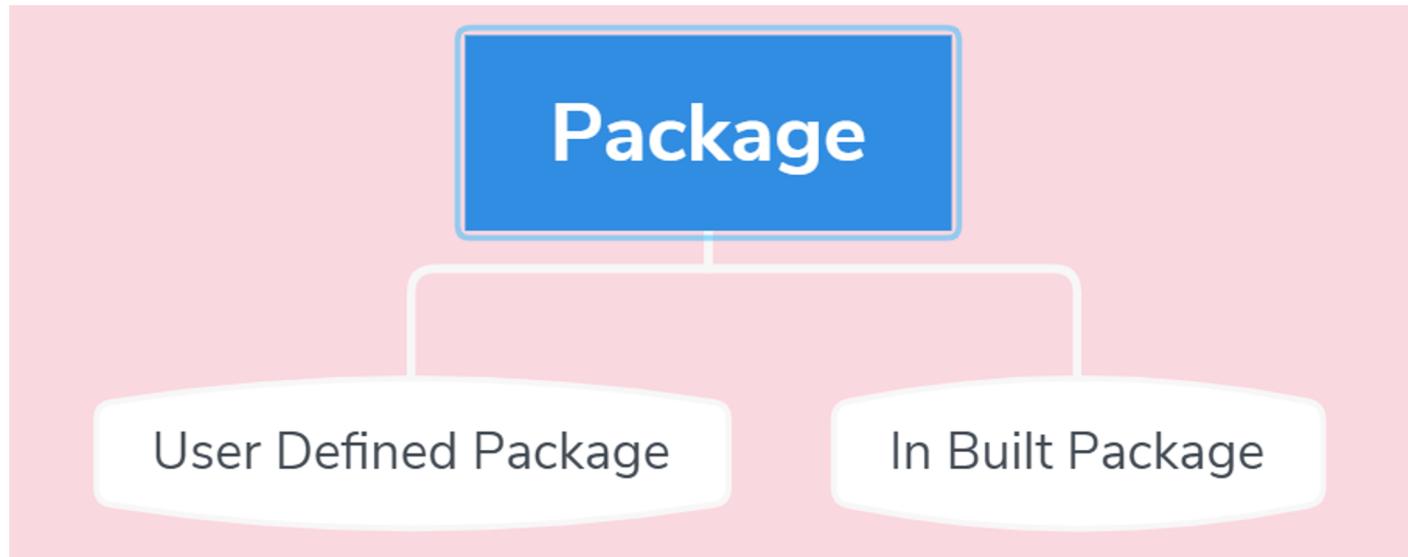
# Package

Một package trong java được sử dụng để nhóm các lớp liên quan. Ta có thể coi nó như một thư mục.

Việc sử dụng package nhằm tránh xung đột về tên và code có thể dễ dàng bảo trì hơn



Package được chia làm 2 loại:





## Built - in Package

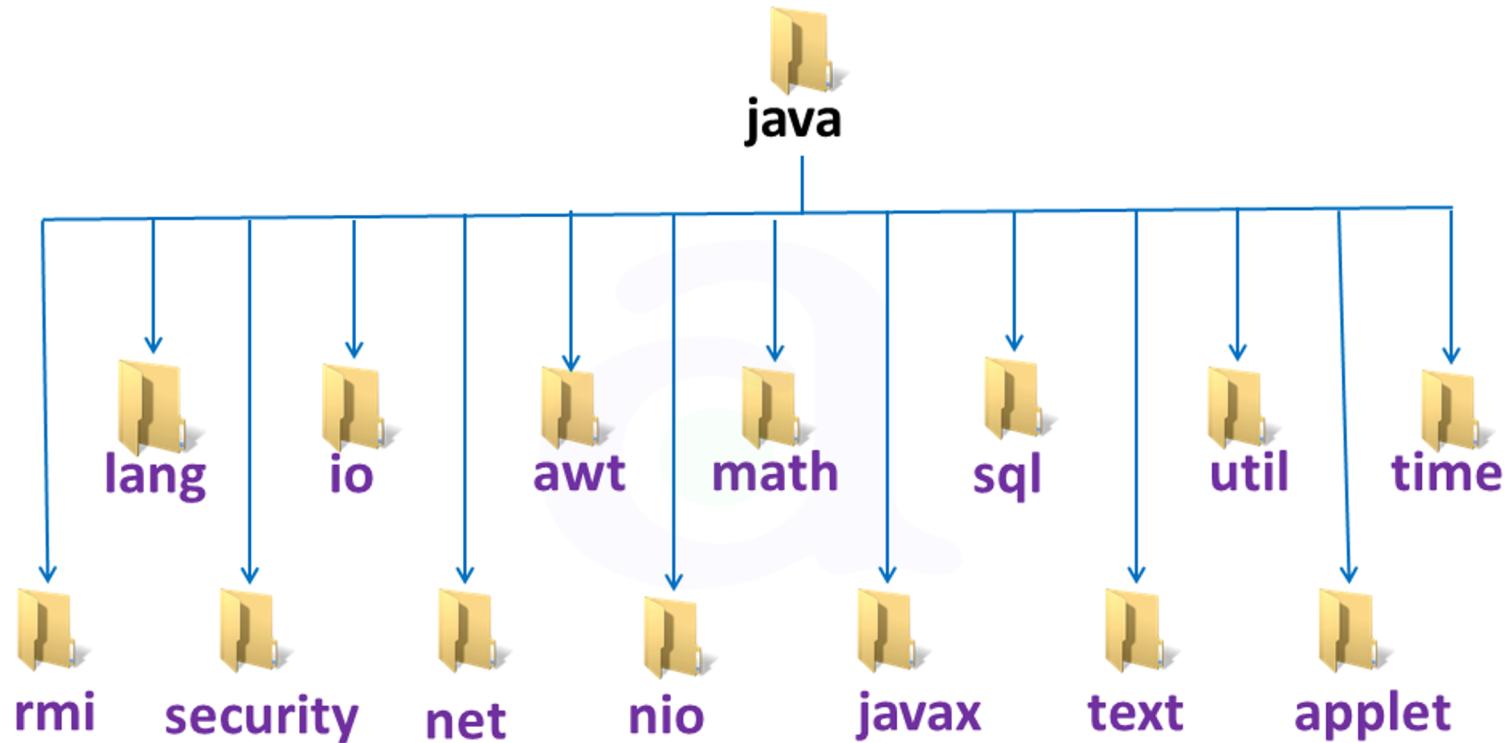
Java API là một thư viện các lớp được viết sẵn, được sử dụng miễn phí.

Thư viện này chứa các thành phần để quản lý đầu vào, cơ sở dữ liệu, ...

Có thể tham khảo tại <https://docs.oracle.com/javase/8/docs/api/>

Thư viện này được chia thành các package và các lớp. Có nghĩa là ta có thể truy cập một lớp duy nhất hoặc toàn bộ package chứa tất cả các lớp thuộc package đó. Ví dụ:

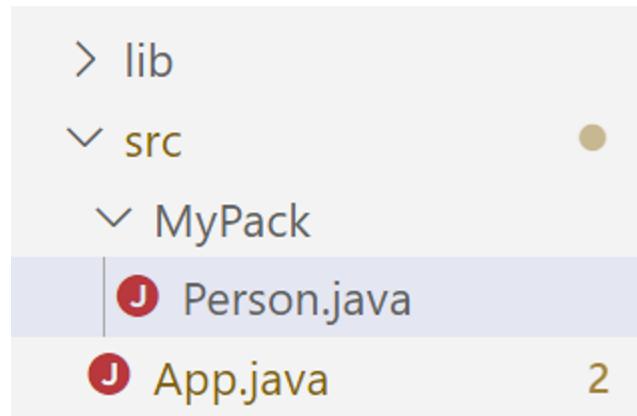
```
import java.util.Scanner; //Truy cập lớp Scanner  
import java.util.*; //Truy cập vào package
```





# User – defined Package

Để tạo package của riêng bạn, bạn cần hiểu rằng Java sử dụng một thư mục hệ thống tệp để lưu trữ chúng. Cũng giống như các thư mục trên máy tính của bạn



Để tạo package, ta sử dụng từ khóa *package*:

```
package MyPack;

public class Person {
    private String name;
    private int age;
}
```

```
import MyPack.*; //hoặc import.MyPack.Person;

public class App {
    public static void main(String[] args) {
        //TODO
    }
}
```



# Access Modifiers

	public	private	protected	default
class	Allowed	Not allowed	Not allowed	Allowed
constructor	Allowed	Allowed	Allowed	Allowed
variable	Allowed	Allowed	Allowed	Allowed
method	Allowed	Allowed	Allowed	Allowed

```
public class Person  
class Person
```



```
private class Person  
protected class Person
```





# Access Modifiers

	class	subclass	package	outside
private	Allowed	Not allowed	Not allowed	Not allowed
protected	Allowed	Allowed	Allowed	Not allowed
public	Allowed	Allowed	Allowed	Allowed
default	Allowed	Not allowed	Allowed	Not allowed

# Tính kế thừa (Inheritance)



# Inheritance (IS-A)

Tính kế thừa là một trong 4 nguyên lý cơ bản của OOP

Trong Java, kế thừa đề cập đến việc tất cả các thuộc tính và phương thức của lớp này có thể kế thừa bởi một lớp khác.

Với kế thừa ta có thêm 2 thuật ngữ:

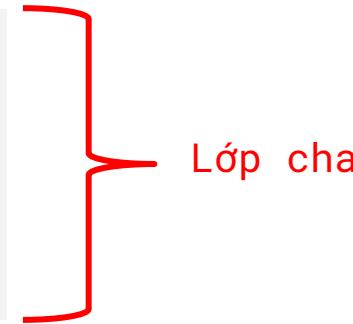
Subclass: Lớp con

Superclass: Lớp cha



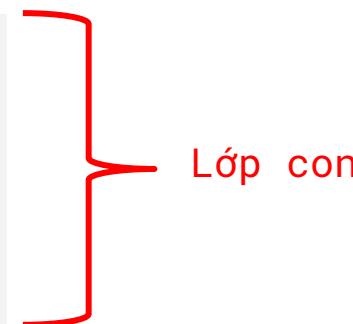
Để kế thừa từ một lớp, ta sử dụng từ khóa `extends`

```
public class Animal {  
    public void eat(){  
        System.out.println("Eating...");  
    }  
}
```



Sử dụng để thể hiện sự kế thừa

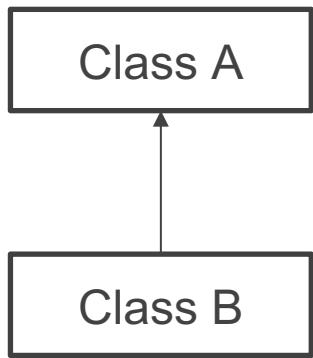
```
public class Cat extends Animal {  
    public void sound(){  
        System.out.println("Meow...");  
    }  
}
```



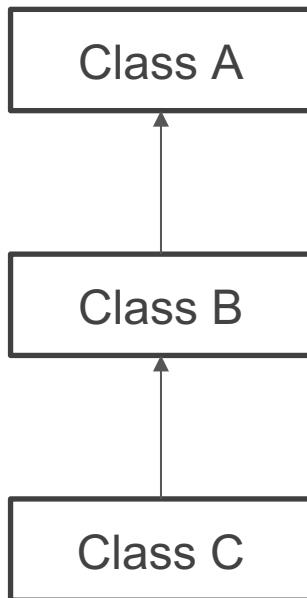
```
public class Main {  
    public static void main(String[] args) {  
        Cat cat = new Cat();  
        cat.eat();  
        cat.sound();  
    }  
}
```

Truy cập tới phương  
thức của lớp cha

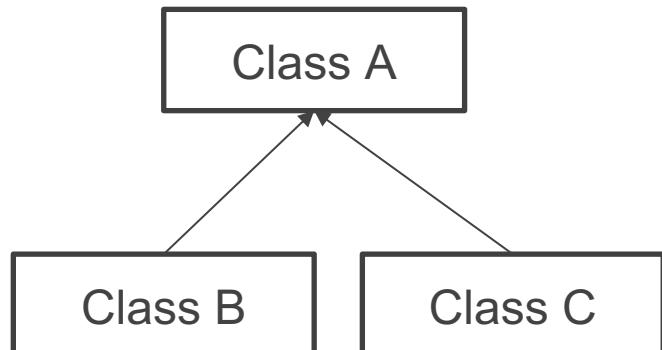
# Các loại kế thừa trong java



Đơn kế thừa



Kế thừa nhiều cấp



Kế thừa thứ bậc

Đơn kế thừa

```
public class ClassA {  
}  
}
```

Class A là superclass  
Class B kế thừa Class A

```
public class ClassB extends ClassA {  
}  
}
```

Kế thừa nhiều cấp

Class A là superclass  
Class B kế thừa ClassA  
ClassC kế thừa ClassB

```
public class ClassA {  
}
```

```
public class ClassB extends ClassA {  
}
```

```
public class ClassC extends ClassB {  
}
```

Kế thừa thứ bậc

Class A là superclass  
Cả ClassB và ClassC  
đều kế thừa từ ClassA

```
public class ClassA {  
}
```

```
public class ClassB extends ClassA {  
}
```

```
public class ClassC extends ClassA {  
}
```

Ngoài ra ta còn đa kế thừa, tuy nhiên đa kế thừa không được hỗ trợ thông qua lớp mà chỉ được hỗ trợ thông qua interface

```
public class ClassA {  
    void msg(){System.out.println("Hello!!!");}  
}
```

```
public class ClassB {  
    void msg(){System.out.println("Hi!!!");}  
}
```

```
public class ClassC extends ClassA, ClassB {  
    public static void main(String[] args) {  
        C c = new C();  
        c.msg();  
    }  
}
```

 Không biết phương thức của lớp nào được gọi

Viết chương trình quản lý thư viện. Thực hiện các yêu cầu sau:  
Tạo class Library chứa:

Các thuộc tính: Mã sách, tên sách, nhà xuất bản, năm xuất bản,  
số lượng

Các phương thức để nhập và xuất thông tin.

Tạo class SchoolBook kế thừa class Library chứa:

Các thuộc tính: Mã sách, tên sách, nhà xuất bản, năm xuất bản,  
số lượng, số trang, tình trạng, số lượng mượn.

Các phương thức để nhập xuất thông tin, phương thức đưa ra vị  
trí và tồn kho



## Aggregation (HAS-A)

Nếu một lớp có một tham chiếu thực thể, thì nó được biết đến như là một lớp có quan hệ HAS-A

Sử dụng quan hệ HAS-A giúp làm tăng tính tái sử dụng của code. Và khi không có mối quan hệ IS-A, thì quan hệ HAS-A là lựa chọn tốt nhất.

```
public class Address {  
    String district, city, country;  
  
    public Address(String district, String city, String country) {  
        this.district = district;  
        this.city = city;  
        this.country = country;  
    }  
}
```

```
public class Person {  
    String name;  
    int age;  
    Address address;  
  
    public Person(String name, int age, Address address) {  
        this.name = name;  
        this.age = age;  
        this.address = address;  
    }  
  
    public void display(){  
        System.out.print(name + " - " + age + " - ");  
        System.out.print(address.district + ", " +address.city+ ", " +address.country);  
    }  
}
```

Address là một lớp

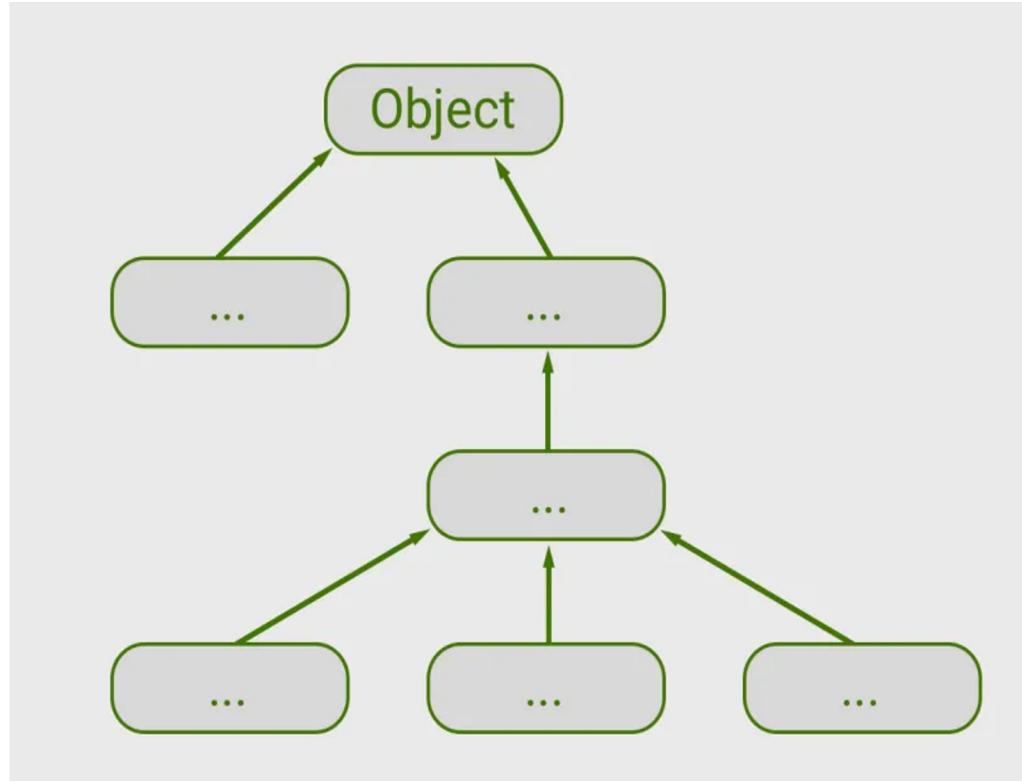
```
public static void main(String[] args) throws Exception {  
    Address address = new Address("Nam Tu Liem", "Ha Noi", "Viet Nam"  
);  
    Person person = new Person("Ngoc", 25, address);  
  
    person.display();  
}
```

Ngoc - 25 - Nam Tu Liem, Ha Noi, Viet Nam



# Lớp Object

Lớp Object là lớp cha của tất cả các lớp trong Java



Phương thức	Miêu tả
public final Class getClass()	Trả về đối tượng lớp Class của đối tượng này. Lớp Class có thể được sử dụng để lấy metadata của lớp này
public int hashCode()	Trả về hashcode cho đối tượng này
public boolean equals(Object obj)	So sánh đối tượng đã cho với đối tượng này
protected Object clone() throws CloneNotSupportedException	Tạo và trả về bản sao (bản mô phỏng) của đối tượng này
public String toString()	Trả về biểu diễn chuỗi của đối tượng này
public final void notify()	Thông báo Thread đơn, đợi trên monitor của đối tượng này
public final void notifyAll()	Thông báo tất cả Thread, đợi trên monitor của đối tượng này

<code>public final void wait(long timeout) throws InterruptedException</code>	Làm cho Thread hiện tại đợi trong khoảng thời gian là số mili giây cụ thể, tới khi Thread khác thông báo (triệu hồi phương thức <code>notify()</code> hoặc <code>notifyAll()</code> )
<code>public final void wait(long timeout, int nanos) throws InterruptedException</code>	Làm cho Thread hiện tại đợi trong khoảng thời gian là số mili giây và nano giây cụ thể, tới khi Thread khác thông báo (triệu hồi phương thức <code>notify()</code> hoặc <code>notifyAll()</code> )
<code>public final void wait() throws InterruptedException</code>	Làm Thread hiện tại đợi, tới khi Thread khác thông báo ( <code>invokes notify() or notifyAll() method</code> ).
<code>protected void finalize() throws Throwable</code>	Được triệu hồi bởi Garbage Collector trước khi đối tượng bị dọn rác

Kiểm tra một đối tượng có phải là thể hiện của một kiểu dữ liệu cụ thể không

```
@Override  
public boolean equals(Object obj) {  
    if(obj instanceof Person){  
        if(((Person)obj).name.equals(this.name)){  
            return true;  
        }  
    }  
    return false;  
}
```

So sánh dựa vào tên

```
public class Person implements Cloneable {  
    @Override  
    protected Object clone() throws CloneNotSupportedException {  
        // TODO Auto-generated method stub  
        return super.clone();  
    }  
}
```

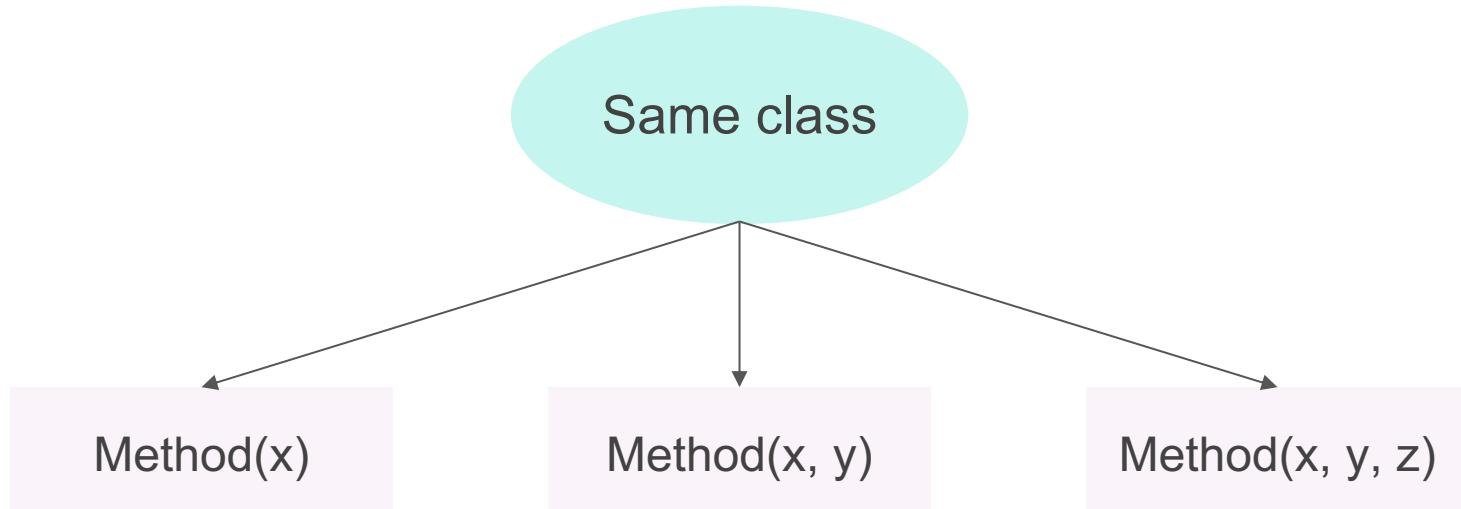
Implements interface  
Cloneable

```
public static void main(String[] args) throws Exception {  
    Person person = new Person("Ngoc", 25, "Ha Noi");  
    person.display();  
  
    Person person3 = (Person) person.clone();  
    person3.display();  
}
```



# Nạp chồng phương thức

Với nạp chồng phương thức, nhiều phương thức có thể có cùng tên nhưng lại có tham số khác nhau



Có 2 cách để nạp chồng phương thức trong Java, đó là:

O1

Thay đổi số lượng tham số

O2

Thay đổi kiểu dữ liệu của tham số

```
public class Calculation {  
    public void sum(int x, int y){  
        System.out.println(x + y);  
    }  
    public void sum(int x, int y, int z){  
        System.out.println(x + y + z);  
    }  
}
```

Có 2 tham số

Có 3 tham số

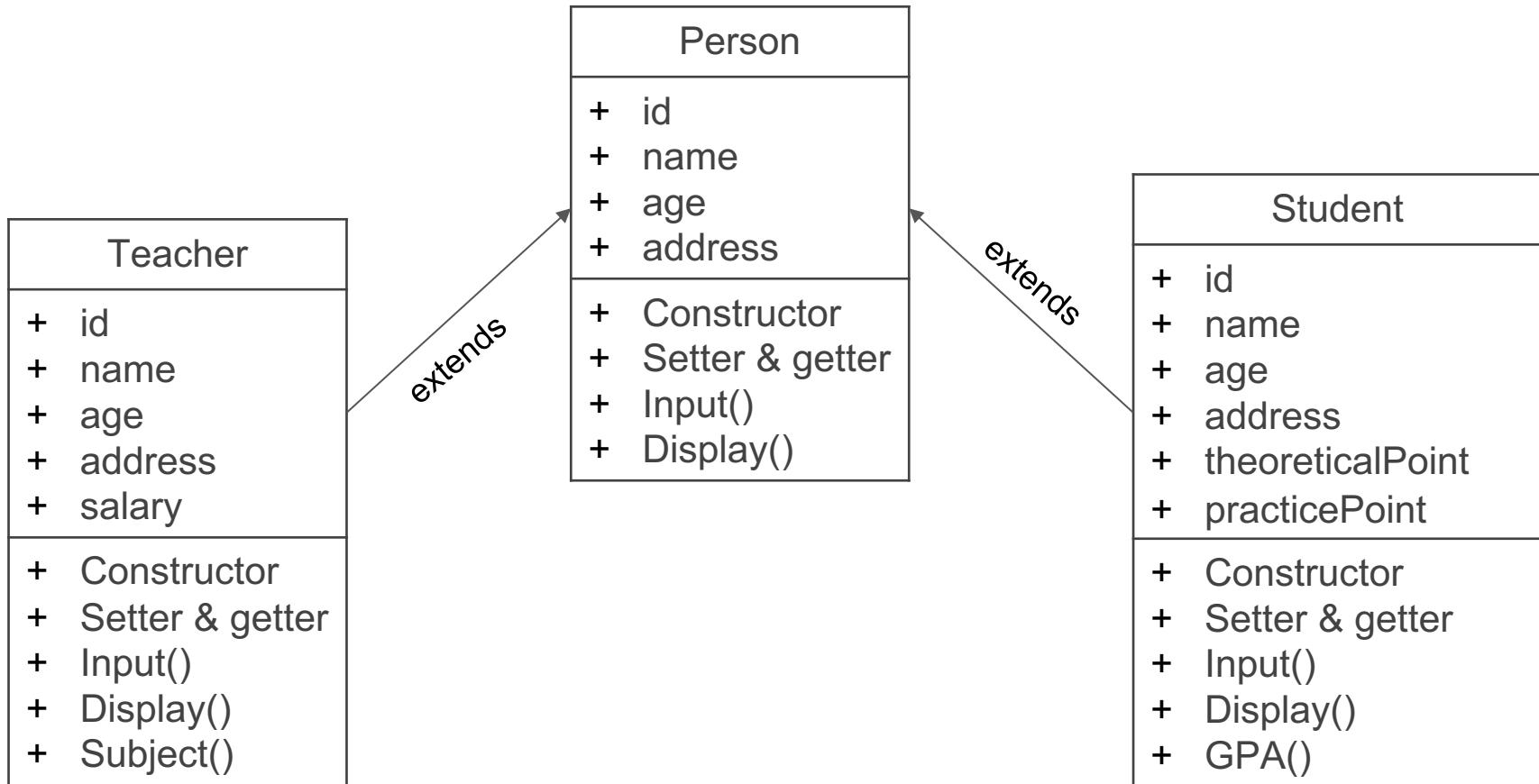
```
public static void main(String[] args) {  
    Calculation calculation = new Calculation();  
    calculation.sum(10, 5);  
    calculation.sum(10, 5, 10);  
}  
}
```

```
public class Calculation {  
    public void sum(int x, int y){  
        System.out.println(x + y);  
    }  
    public void sum(double x, double y){  
        System.out.println(x + y);  
    }  
  
    public static void main(String[] args) {  
        Calculation calculation = new Calculation();  
        calculation.sum(10, 5);  
        calculation.sum(10.5, 5);  
    }  
}
```

Kiểu dữ liệu của  
tham số là int

Kiểu dữ liệu của  
tham số là double

# Viết chương trình quản lý trường học.

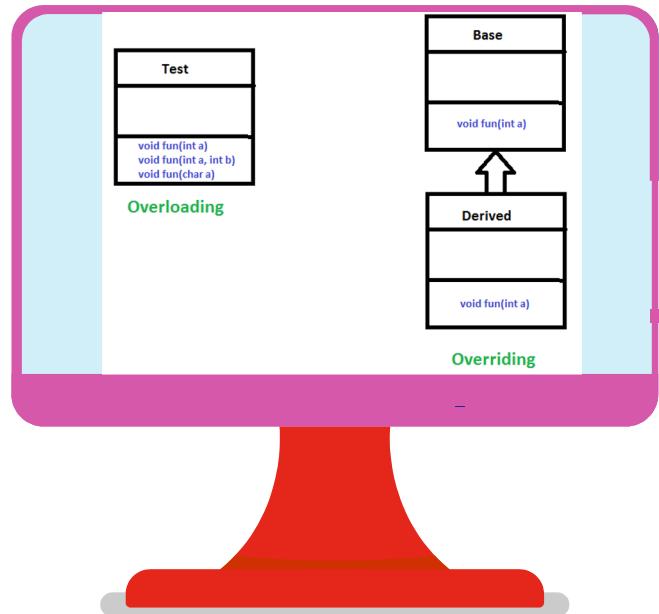


# Tính đa hình

# Tính đa hình (Polymorphism)

Tính đa hình là khả năng một đối tượng có thể thực hiện một tác vụ theo nhiều cách khác nhau.

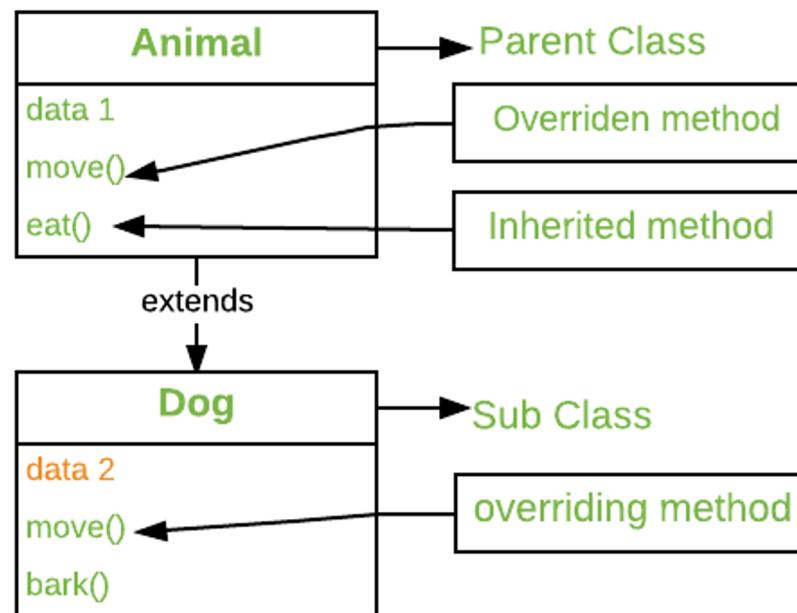
Trong Java, chúng ta sử dụng ghi đè phương thức để có tính đa hình.





# Ghi đè phương thức

Nếu lớp con có cùng phương thức được khai báo trong lớp cha, nó được gọi là ghi đè phương thức



```
public class Shape {  
    public void draw(){  
        System.out.println("Hình gì đó!!!");  
    }  
}
```

Superclass

```
public class Square extends Shape {  
    @Override  
    public void draw() {  
        super.draw();  
        System.out.println("Hình vuông!!!");  
    }  
}
```

Subclass



# Từ khóa super

Từ khóa super trong java là một biến tham chiếu được sử dụng để tham chiếu trực tiếp đến đối tượng của lớp cha gần nhất.

super

Có thể được sử dụng để tham chiếu đến biến instance của lớp cha

super

Có thể được sử dụng để gọi các phương thức của lớp cha

super()

Có thể được sử dụng để gọi các constructor của lớp cha

```
public class Person {  
    String name = "Ngoc";  
}
```

```
public class Student extends Person {  
    String name = "Ngoc Ban Quyen";  
  
    void display(){  
        System.out.println(super.name);  
    }  
  
    public static void main(String[] args) {  
        Student student = new Student();  
        student.display();  
    }  
}
```

Tham chiếu tới  
biến name của lớp  
Person

```
public class Person {  
    void message(){  
        System.out.println("Hôm nay là thứ 3!!!");  
    }  
}
```

```
public class Student extends Person {  
    @Override  
    void message() {  
        super.message();  
        System.out.println("Ngày mai là thứ 4!!!");  
    }  
}
```

```
public static void main(String[] args) {  
    Student student = new Student();  
    student.message();  
}
```

Gọi phương thức của  
lớp cha

```
public class Person {  
    public Person() {  
        System.out.println("Phương thức của lớp Person!!!");  
    }  
}
```

```
public class Student extends Person {  
    public Student() {  
        super(); → Gọi constructor của lớp Person  
        System.out.println("Phương thức của lớp Student!!!");  
    }  
  
    public static void main(String[] args) {  
        Student student = new Student();  
    }  
}
```



# Từ khóa final

Final trong Java được sử dụng để hạn chế thao tác của người dùng. Final có thể sử dụng trong nhiều ngữ cảnh như:

Biến final

Để tạo biến có giá trị không đổi (Hằng số)

Phương  
thức final

Để ngăn chặn ghi đè phương thức

Lớp final

Ngăn chặn sự kế thừa

```
public class Person {  
    public final void display(){  
        System.out.println("In thong tin");  
    }  
}
```

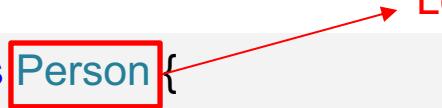
```
public class Student extends Person {  
    @Override  
    public void display() {  
        super.display();  
    }  
}
```

Lỗi

Error: LinkageError occurred while loading main class Student  
java.lang.VerifyError: class Student overrides final method Person.display()V

```
public final class Person {  
    public void display(){  
        System.out.println("In thong tin");  
    }  
}
```

```
public class Student extends Person {  
}
```



Lỗi

Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
at Student.main(Student.java:2)



# Upcasting và Downcasting

Đây là cơ chế được sử dụng để chuyển kiểu đối với kiểu dữ liệu tham chiếu

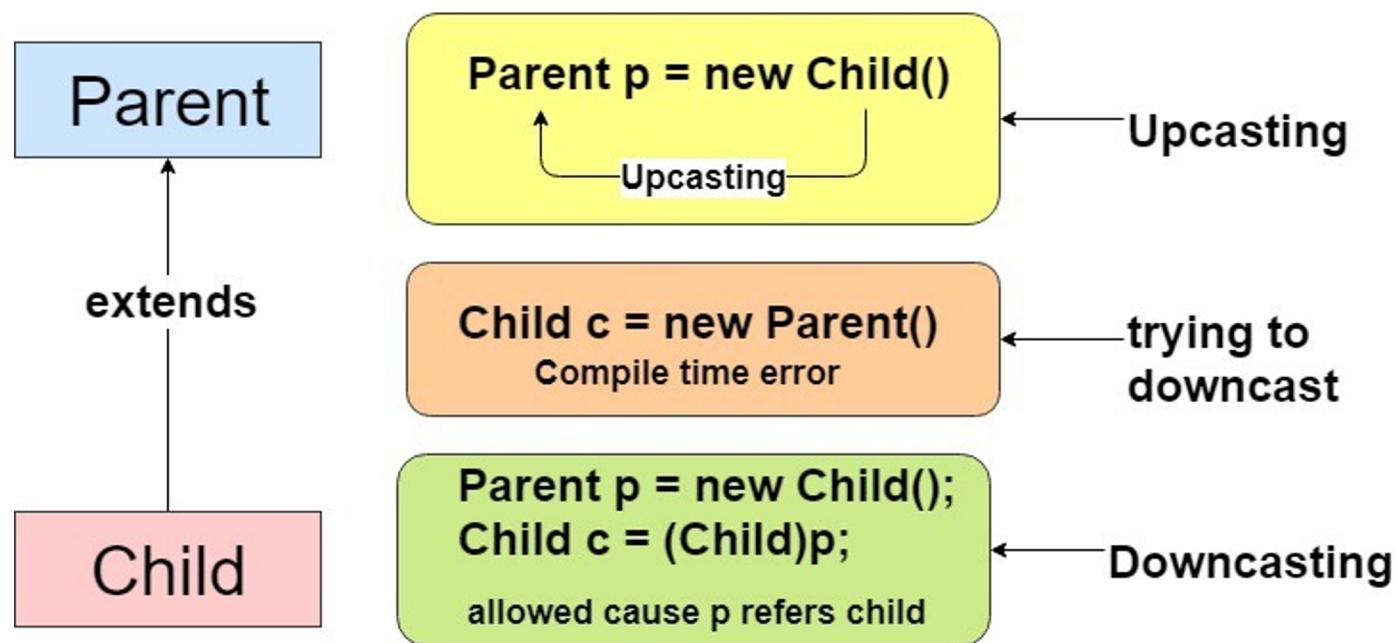
## Upcasting

Khi biến tham chiếu của lớp cha tham chiếu tới đối tượng của lớp con.

## Downcasting

Là dạng chuyển kiểu chuyển 1 đối tượng là một thể hiện của lớp cha xuống thành đối tượng là thể hiện của lớp con trong quan hệ kế thừa.

# Upcasting và Downcasting



# Tính trừu tượng



# Trùu tượng trong OOP

Tính trừu tượng trong lập trình hướng đối tượng là chỉ nêu ra vấn đề mà không hiển thị cụ thể, chỉ hiển thị tính năng thiết yếu đối với đối tượng mà không nói ra quy trình hoạt động



## Ưu điểm của tính trừu tượng

Tính trừu tượng cho phép các lập trình viên loại bỏ tính chất phức tạp của đối tượng

Tính trừu tượng giúp ta tập chung vào những cốt lõi cần thiết của đối tượng

Tính trừu tượng cung cấp nhiều tính năng mở rộng khi sử dụng kết hợp với tính đa hình và kế thừa trong OOP



# Trùu tượng trong java

## Lớp trùu tượng

Là lớp bị hạn chế,  
không thể dùng để  
tạo đối tượng (Để  
truy cập nó phải kế  
thừa từ lớp khác)

## Phương thức trùu tượng

Chỉ có thể sử dụng  
trong lớp trùu  
tượng và nó không  
có phần thân. Phần  
thân được cung cấp  
bởi lớp con



# Lớp trừu tượng

Để tạo lớp trừu tượng ta dùng từ khóa abstract trước từ khóa class  
Cú pháp:

<Phạm vi truy cập> abstract class <Tên lớp> {}

Ví dụ:

```
public abstract class Person {  
}
```



# Phương thức trừu tượng

Cú pháp:

<Phạm vi truy cập> abstract <Kiểu dữ liệu trả về>  
<Tên phương thức> (<Tham số>);

```
public abstract class Shape {
```

Lớp abstract

```
}
```

```
    public abstract void draw();
```

Phương thức abstract

```
public class Square extends Shape {
```

```
@Override
```

```
public void draw() {
```

```
    System.out.println("Ve hinh vuong!!!");
```

```
}
```

```
}
```

Phần thân được  
cung cấp bởi lớp  
con

```
public class Circle extends Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Ve hinh tron!!!");  
    }  
  
}
```



Phần thân  
được cung cấp  
bởi lớp con

# Interface

# Interface

**Interface** là một kiểu dữ liệu tham chiếu trong Java. Nó là tập hợp các phương thức **abstract** (trừu tượng). Khi một lớp kế thừa **interface**, thì nó sẽ kế thừa những phương thức **abstract** của **interface** đó

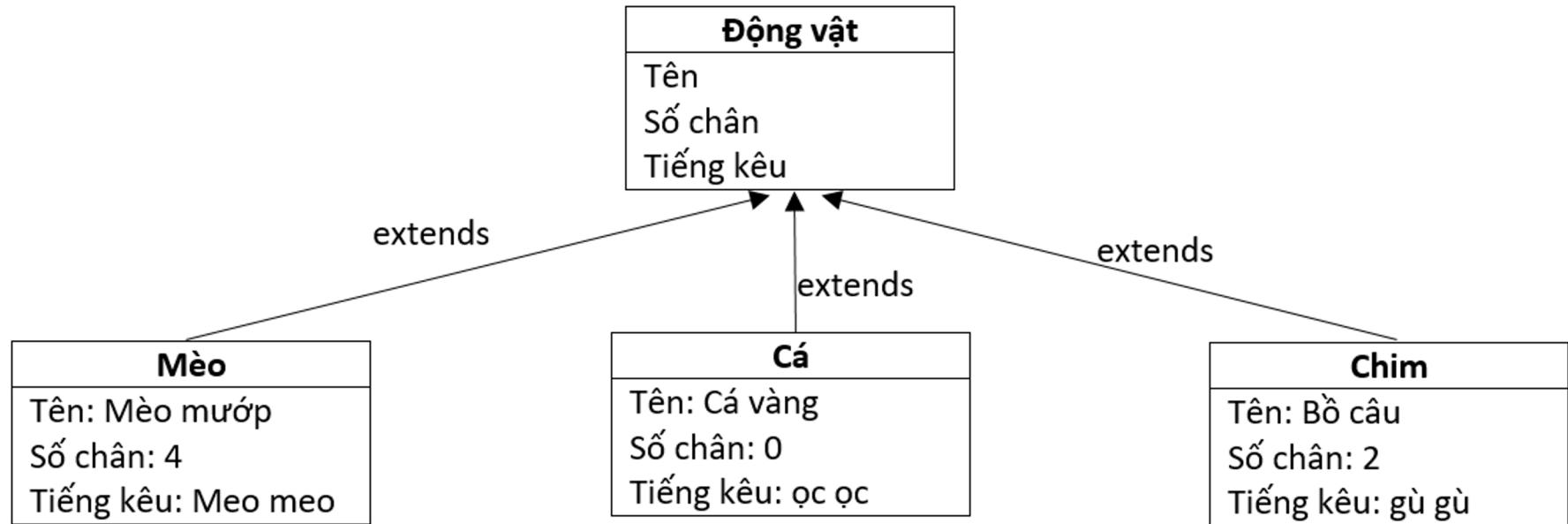


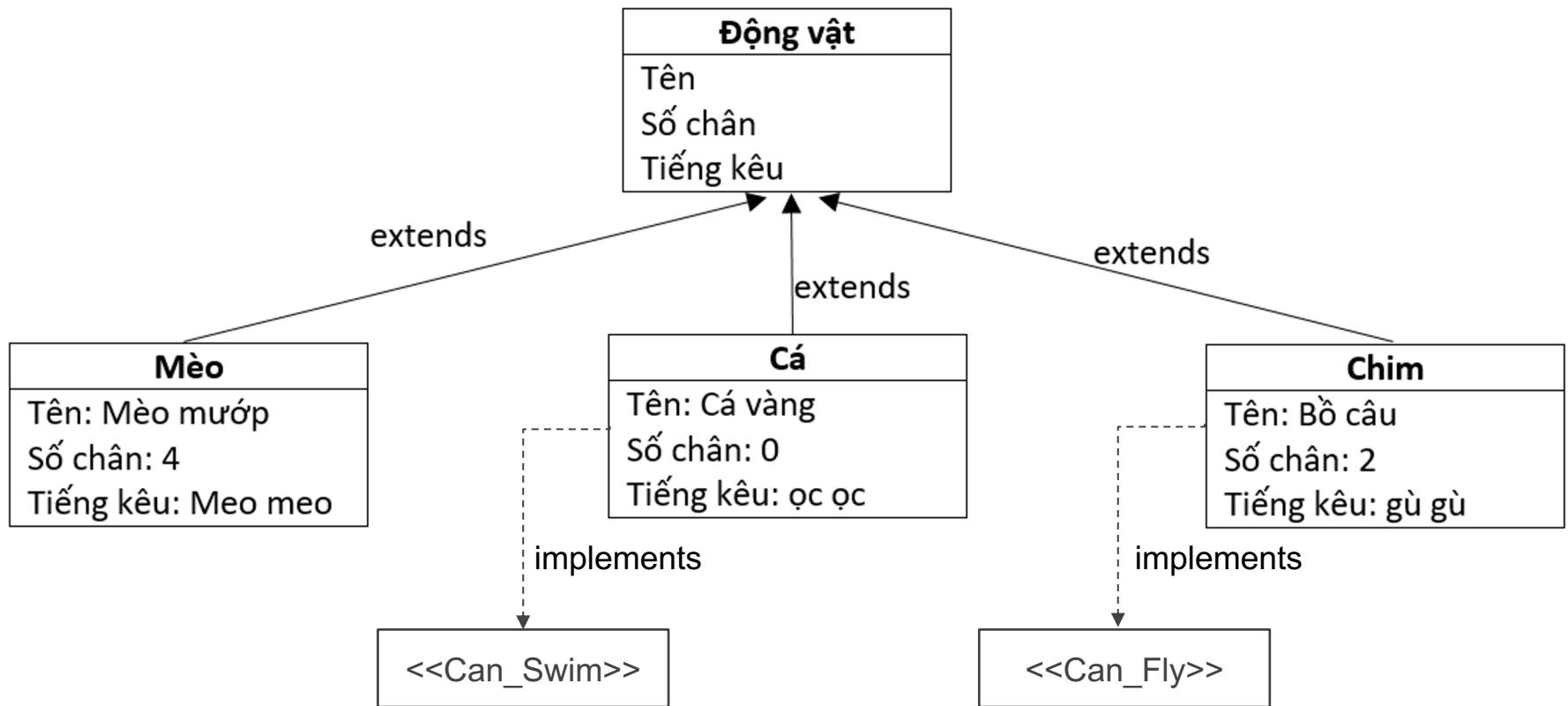


## Tại sao chúng ta lại sử dụng interface?

→ Java không hỗ trợ đa kế thừa. Do đó, ta không thể kế thừa cùng một lúc nhiều class. Để giải quyết vấn đề này interface ra đời







# Interface (Giao diện)

01

Không thể khởi tạo, nên không có phương thức khởi tạo.

02

Tất cả các phương thức trong interface luôn ở dạng public abstract mà không cần khai báo.

03

Các thuộc tính trong interface luôn ở dạng public static final mà không cần khai báo, yêu cầu phải có giá trị



# Tạo Interface

Cú pháp:

```
interface <Tên Interface>{  
    //Các thành phần bên trong interface  
}
```



# Tạo Interface

Để truy cập các phương thức interface, interface phải được thực hiện bởi một lớp khác bằng từ khóa “**implements**”

```
interface Animal {  
    void eat();  
}
```

```
public class Cats implements Animal{  
    @Override  
    public void eat() {  
        System.out.println("Fish");  
    }  
}
```

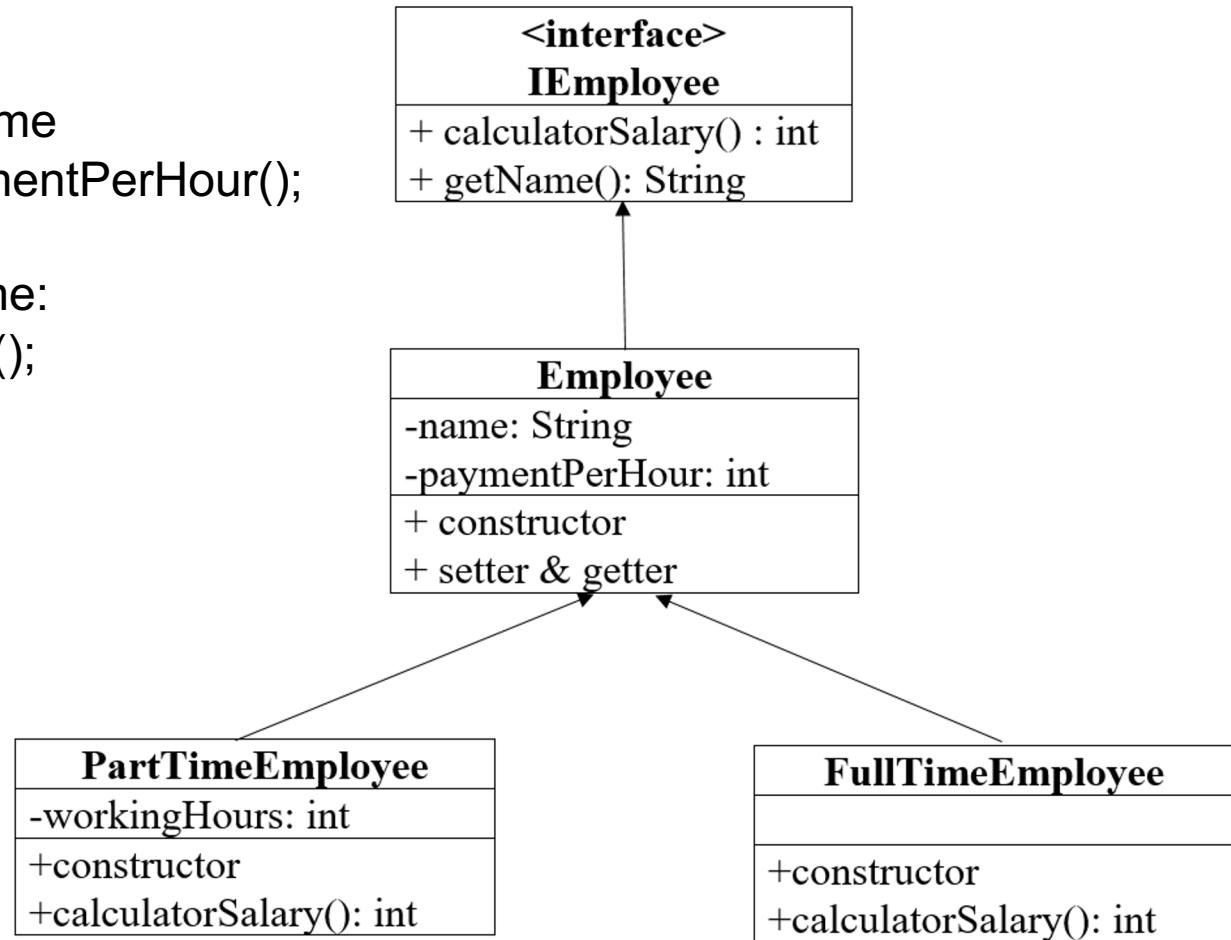
Tính lương:

Đối với nhân viên part time

workingHours \* getPaymentPerHour();

Đối với nhân viên full time:

8 \* getPaymentPerHour();

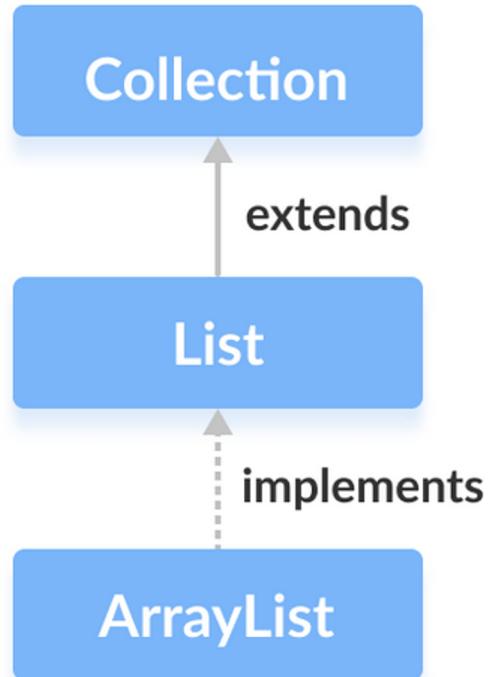




# ArrayList

Lớp **ArrayList** trong java được sử dụng như một mảng động để lưu trữ các phần tử. Nó kế thừa lớp `AbstractList` và implements giao tiếp `List`.

`ArrayList` có thể thay đổi kích thước





# ArrayList

Khởi tạo ArrayList:

```
ArrayList<Kiểu dữ liệu> <Tên> = new ArrayList<Kiểu dữ liệu>();
```

```
ArrayList<Kiểu dữ liệu> <Tên> = new ArrayList<>();
```

Ví dụ:

```
ArrayList<String> cars = new ArrayList<String>();
```

```
ArrayList<Integer> myNumbers = new ArrayList<Integer>();
```



## Một số phương thức của ArrayList

Thêm phần tử vào mảng bằng hàm add():

```
cars.add("Mazda");  
cars.add(3, "Vinfast");
```

The diagram illustrates the parameters of the `add()` method. In the first line, the string "Mazda" is highlighted with a red box and an arrow points from it to the word "Element". In the second line, the index "3" is highlighted with a red box and an arrow points from it to the word "index", while the string "Vinfast" is also highlighted with a red box.

Truy cập phần tử trong ArrayList bằng hàm get():

```
System.out.println(cars.get(1));
```

## Một số phương thức của ArrayList

Cập nhật giá trị của phần tử trong ArrayList bằng hàm  
set(index, element):

```
cars.set(3, "Audi");
```

The code shows the use of the `set` method on an `ArrayList` named `cars`. The method takes two arguments: an index (3) and an element ('Audi'). Red boxes highlight the index and element parameters, with arrows pointing to their respective values.

Xóa tất cả phần tử trong ArrayList bằng hàm clear():

```
cars.clear();
```

Xóa phần tử trong ArrayList bằng hàm remove():

```
cars.remove("Mazda");
```

The code shows the use of the `remove` method on an `ArrayList` named `cars`. The method takes one argument: the element to be removed ('Mazda'). A red box highlights the element parameter, with an arrow pointing to the string 'Mazda'.

```
cars.remove(0);
```

The code shows the use of the `remove` method on an `ArrayList` named `cars`. The method takes one argument: the index of the element to be removed (0). A red box highlights the index parameter, with an arrow pointing to the value 0.

Phương thức	Mô tả
add(Object o)	Thêm phần tử được chỉ định vào cuối một danh sách.
add(int index, Object element)	Chèn phần tử element tại vị trí index vào danh sách.
addAll(Collection c)	Thêm tất cả các phần tử trong collection c vào cuối của danh sách, theo thứ tự chúng được trả về bởi bộ lặp iterator.
addAll(int index, Collection c)	Chèn tất cả các phần tử trong collection c vào danh sách, bắt đầu từ vị trí index.
retainAll(Collection c)	Xóa những phần tử không thuộc collection c ra khỏi danh sách.
removeAll(Collection c)	Xóa những phần tử thuộc collection c ra khỏi danh sách.
indexOf(Object o)	Trả về chỉ mục trong danh sách với sự xuất hiện đầu tiên của phần tử được chỉ định, hoặc -1 nếu danh sách không chứa phần tử này.

Phương thức	Mô tả
lastIndexOf(Object o)	Trả về chỉ mục trong danh sách với sự xuất hiện cuối cùng của phần tử được chỉ định, hoặc -1 nếu danh sách không chứa phần tử này.
toArray()	Trả về một mảng chứa tất cả các phần tử trong danh sách này theo đúng thứ tự.
toArray(Object[] a)	Trả về một mảng chứa tất cả các phần tử trong danh sách này theo đúng thứ tự.
clone()	Trả về một bản sao của ArrayList.
clear()	Xóa tất cả các phần tử từ danh sách này.
trimToSize()	Cắt dung lượng của thể hiện ArrayList này là kích thước danh sách hiện tại.
contains(element)	Kết quả trả về là true nếu tìm thấy element trong danh sách, ngược lại trả về false.



## Bài tập

Bài 1: Viết chương trình in ra tất cả các số từ 0 đến 100. Các số thu được sẽ in thành một chuỗi trên một dòng, cách nhau bằng dấu phẩy

Bài 2: Viết chương trình in ra tất cả các số chẵn từ 0 đến 100. Các số thu được sẽ in thành một chuỗi trên một dòng, cách nhau bằng dấu phẩy

# File Handling

# File JSON

JSON là viết tắt của JavaScript Object Notation, là một kiểu định dạng dữ liệu tuân theo một quy luật nhất định mà hầu hết các ngôn ngữ lập trình hiện nay đều có thể đọc được. JSON là một tiêu chuẩn mở để trao đổi dữ liệu trên web.

Định dạng JSON sử dụng các cặp *key – value* để dữ liệu sử dụng. Nó hỗ trợ các cấu trúc dữ liệu như đối tượng và mảng



```
{  
  "id": 1,  
  "first_name": "Lillian",  
  "last_name": "Skellern",  
  "email": "lskellern0@salon.com",  
  "gender": "Female"  
}
```

```
[ {  
    "id": 1,  
    "first_name": "Abbe",  
    "last_name": "Klaff",  
    "email": "aklaff0@addtoa.com",  
    "gender": "Male"  
} , {  
    "id": 2,  
    "first_name": "Yorgo",  
    "last_name": "Leversha",  
    "email": "yleve@cyber.com",  
    "gender": "Bigender"  
} ]
```



# JSON

Google Gson là một thư viện Java hỗ trợ xử lý JSON (Chuyển đổi một chuỗi JSON thành đối tượng Java hoặc ngược lại).





# Cài đặt thư viện Gson

Thêm dependency vào file pom.xml

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.9.0</version>
</dependency>
```



# Mục tiêu của Gson

- Cung cấp các phương thức `toJson()` và `fromJson()` đơn giản để chuyển các đối tượng Java sang JSON và ngược lại.
- Cho phép các **đối tượng không thể thay đổi** có sẵn được chuyển đổi sang/ từ JSON.
- Hỗ trợ rộng rãi của Java **Generics**.
- Cho phép **tùy chỉnh** cho các đối tượng để chuyển đổi sang/ từ JSON.
- Hỗ trợ các **đối tượng phức tạp** (có phân cấp thừa kế nhiều cấp và sử dụng rộng rãi các kiểu dữ liệu Generics).



## Ví dụ

```
//Tạo đối tượng Gson
Gson gson = new Gson();
Person person = new Person("Ngọc", 5, "Hà Giang");

//Chuyển đổi từ Object sang JSON
String result = gson.toJson(person);
System.out.println(result);

//Chuyển đổi từ JSON sang Object
String json = "{\"name\":\"Linh\", \"age\":8, \"address\":\"Hà Nội\"}";
Person newPerson = gson.fromJson(json, Person.class);
System.out.println(newPerson);
```

# Annotation

# Annotation

**Annotation** là một thẻ đại diện cho siêu dữ liệu tức là nó được gắn với lớp, interface, phương thức hoặc các trường để chỉ định một số thông tin bổ sung có thể được sử dụng bởi trình biên dịch java và JVM.



Các annotation được sử dụng để:

Chỉ dẫn cho trình biên dịch (Compiler)

Chỉ dẫn trong thời điểm biên dịch (Build-time)

Chỉ dẫn trong thời gian chạy (Runtime)



# Annotation

1

Các Annotation được tích hợp sẵn trong java.

2

Annotation tùy chỉnh - do người dùng định nghĩa.

# Các Annotation được tính hợp sẵn

Java Annotation được thích hợp sẵn được sử dụng trong code java.

- @Override
- @SuppressWarnings
- @Deprecated

Java Annotation được thích hợp sẵn được sử dụng trong Annotation khác

- @Target
- @Retention
- @Inherited
- @Documented

## @Override Annotation

Annotation @Override không bắt buộc phải chú thích trên method đã ghi đè method của lớp cha. Tuy nhiên, khi ghi đè một phương thức trong lớp con chúng ta nên sử dụng chú thích này để đánh dấu phương thức

```
public class Parent {  
    public void display(){  
        System.out.println("Phương thức lớp cha");  
    }  
}
```

Annotation @Override cho ta biết phương thức display đang ghi đè phương thức lớp cha

```
public class Child1 extends Parent {  
    @Override  
    public void display() {  
        System.out.println("Phương thức lớp Child1");  
    }  
}
```

```
public class Child2 extends Parent{  
    public void display() {  
        System.out.println("Phương thức lớp Child2");  
    }  
}
```

Nếu đổi tên phương thức của lớp cha sẽ xảy ra chuyện gì???

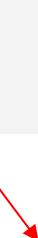
## @Deprecated Annotation

Annotation chỉ ra rằng những phần tử bị đánh dấu  
đã bị lỗi thời và không nên sử dụng nữa

```
public class Parent {  
    @Deprecated  
    public void parentMethod1(){  
        System.out.println("phương thức parentMethod1");  
    }  
}
```

Method parentMethod1 được đánh dấu  
@Deprecated để thông báo với lập trình  
viên không nên sử dụng phương thức này

```
public class Main {  
    public static void main(String[] args) {  
        Parent p = new Parent();  
        p.parentMethod1();  
    }  
}
```

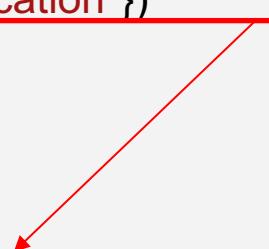


Nếu vẫn gọi tới phương thức đó

## @SuppressWarnings Annotation

Chú thích này hướng dẫn cho trình biên dịch bỏ qua những cảnh báo cụ thể

```
public class Main {  
    @SuppressWarnings({"checked", "deprecation"})  
    public static void main(String[] args) {  
        Parent p = new Parent();  
        p.parentMethod1();  
  
    }  
}
```



Khi thêm annotation @SuppressWarnings thì đã không còn cảnh báo khi sử dụng phương thức parentMethod1()

## @Retention

Annotation @Retention dùng để chú thích mức độ tồn tại của một annotation nào đó. Có 3 mức nhận thức tồn tại được chú thích



RetentionPolicy.SOURCE



RetentionPolicy.CLASS



RetentionPolicy.RUNTIME

## @Target

Annotation @Target dùng để chỉ định cho một annotation khác và annotation đó sẽ được sử dụng trong phạm vi nào

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

## **@Inherited**

Theo mặc định, các annotation không được kế thừa cho lớp con. Annotation @Inherited đánh dấu chú thích sẽ được kế thừa cho các lớp con

## **@Documented**

Annotation @Documented đánh dấu chú thích để đưa vào tài liệu



## Tạo Custom Annotations

Sử dụng **@interface** là từ khóa khai báo một Annotation, annotation khá giống một interface. Annotation có hoặc không có các phần tử trong nó.



## Tạo Custom Annotations

Đặc điểm của các phần tử (element) của annotation:

- Không có thân hàm
- Không có tham số hàm
- Khai báo trả về phải là một kiểu cụ thể:
  - Các kiểu nguyên thủy (boolean, int, float, ...)
  - Enum
  - Annotation
  - Class (Ví dụ String.class)
- Có thể có giá trị mặc định.