JAVA 1

Lập trình hướng đối tượng (gọi tắt là OOP - Object-Oriented Programming) là kĩ thuật lập trình hỗ trợ công nghệ đối tượng. OOP được xem là giúp tăng năng suất, đơn giản hóa độ phức tạp khi bảo trì cũng như mở rộng phần mềm bằng cách cho phép lập trình viên tập trung vào các đối tượng phần mềm ở bậc cao hơn. Tóm lại, đây là một nền tảng lập trình rất quan trọng để các bạn phát triển trở thành một lập trình viên chuyên nghiệp sau này.

Khái niệm lập trình hướng đối tượng căn bản là trừu tượng (abstraction), Kế thừa (inheritance), đóng gói (encapsulation) và đa hình (polymorphism) .

IDE( INTERGRATED DEVELOPMENT ENVIRONMENT) – Môi trường phát triển tích hợp. ( Eclipse) thường được dùng cho Java.

Cau lenh khai bao cho Java: ***public static void main (String[] args ) { }***

Cau lenh if :

***if ( Boolean\_expression) {***

***execute if the boolean expression is true***

***}***

There are three kinds of variables in Java −

* **Local variables**
* **Instance variables**
* **Class/Static variables**

**Local variables** are visible only within the declared method, constructor, or block.

**Instance Variables**

* Instance variables are declared in a class, but outside a method, constructor or any block.
* When a space is allocated for an object in the heap, a slot for each instance variable value is created.
* Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
* Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
* Instance variables can be declared in class level before or after use.
* Access modifiers can be given for instance variables.
* The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
* Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
* Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

Declare Array :

Ex: kieu\_du\_lieu[] ten\_mang = {…,…,…,…};

int[] mang\_1 = { 1,2,3,4,5};

**For Loop :**

**2** types of for loop :

*The second is* : **int**[] x = {12,659,958,-959,956251};

**for** (**int** i : x) {

System.***out***.println(i);

}

**Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

**Class** − A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

In java, you can use the keyword FINAL to denote a constant. For instance:

public class Constants  
{  
public static void main(String[] args)  
{  
**final** double CM\_PER\_INCH = 2.54;  
double paperWidth = 8.5;  
double paperHeight = 11;  
System.out.println("Paper size in centimeters: "  
+ paperWidth \* CM\_PER\_INCH + " by " + paperHeight \* CM\_PER\_INCH);  
}  
}

More common in Java to create a constant :

public class Constants2  
{  
public static final double CM\_PER\_INCH = 2.54;  
public static void main(String[] args)  
{  
double paperWidth = 8.5;  
double paperHeight = 11;  
System.out.println("Paper size in centimeters: "  
+ paperWidth \* CM\_PER\_INCH + " by " + paperHeight \* CM\_PER\_INCH);  
}  
}

**Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

**Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

**Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

## Ví Dụ Nhập Dữ Liệu Kiểu Chuỗi

import java.util.Scanner;;

public class MyFirstClass {

    public static void main(String[] args) {

        // TODO Auto-generated method stub

        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter your name here: ");

        String name = scanner.nextLine();

        System.out.println("Hello! " + name);

    }

}

Ví Dụ Nhập Dữ Liệu Kiểu int

System.out.println("How old are you? ");

int age = scanner.nextInt();

System.out.println("Your age is: " + age);

## Ví Dụ Nhập Dữ Liệu Kiểu Float

System.out.println("How old are you? ");

int age = scanner.nextFloat();

System.out.println("Your age is: " + age);

***switch case***

switch (đối\_tượng\_so\_sánh) {

case giá\_trị\_1:

các\_câu\_lệnh\_1;

break;

case giá\_trị\_2:

các\_câu\_lệnh\_2;

break;

...

default:

các\_câu\_lệnh\_n;

break;

}

Inheritance( Tinh thua ke) :

Public class Name\_class\_1 extends Name\_class\_2 {

}

Ex: class Super {

.....

.....

}

class Sub extends Super {

.....

.....

}

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

* It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
* It is used to **invoke the superclass** constructor from subclass.

## Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

class Superclass {

int age;

Superclass(int age) {

this.age = age;

}

public void getAge() {

System.out.println("The value of the variable named age in super class is: " +age);

}

}

IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

the **implements** keyword is used with classes to inherit the properties of an interface. Interfaces can never be extended by a class.

public interface Animal {

}

public class Mammal implements Animal {

}

public class Dog extends Mammal {

}

## The instanceof Keyword

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal.

interface Animal{}

class Mammal implements Animal{}

public class Dog extends Mammal {

public static void main(String args[]) {

Mammal m = new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

**Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java −

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

## Benefits of Encapsulation

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.

Tính đa hình (Polymophism) – có nhiều form khác nhau và bất cứ khi nào bạn định nghĩa lớp đối tượng và các thuộc tính của nó thỏa mãn điều kiện tồn tại ở nhiều dạng khác nhau được gọi là đa hình.

* *Mảng cũng có một bất lợi là bạn phải khai báo độ lớn, tức khai báo sẵn số lượng phần tử mà Mảng này sẽ sử dụng*

Co 2 cach khai bao mang nhu sau:

1. *ten\_kieu\_du\_lieu[] ten\_mang =*

2. *ten\_kieu\_du\_lieu ten\_mang[] =*

Để cấp phát bộ nhớ cho mảng, ta dung từ khoá NEW

:

Ten\_mang = new kieu\_du\_lieu[kich\_co\_mang];

Khai báo gộp như sau:

Kieu\_du\_lieu[] ten\_mang = new kieu\_du\_lieu[kich\_co\_mang]

Ex: int[] soThuc = new int[20];

– Không được dùng foreach để remove một phần tử nào đó khỏi Danh Sách (mình dùng từ “Danh Sách” chứ không phải “Mảng”, vì Mảng không cho phép bạn thêm hay bớt một phần tử trong nó, chỉ có Danh Sách mới cho phép, và vì foreach cũng sẽ làm việc trên Danh Sách tương tự như Mảng nên mình nhắc đến ý này ở đây, về sau khi nói đến Danh Sách mình sẽ nhắc lại cho bạn nhớ).

– Vì foreach tự duyệt tuần tự trên các phần tử mảng, nên nó rất dở trong việc xác định vị trí (chỉ số) của từng phần tử. Do đó đừng bắt foreach hoạt động khi bạn muốn truy xuất nhanh đến vị trí của bất kỳ phần tử nào đó.

Mang 2 chieu : tên\_mảng = new kiểu\_dữ\_liệu[số\_lượng\_dòng][số\_lượng\_cột];

So sánh chuỗi:

### equals()

Nếu có 2 chuỗi thì để so sánh, cú pháp như sau: *chuoi1.equals(chuoi2),* kết quả trả về sẽ là hàm Boolean: True or False.(có phân biệt in hoa hay thường).

### equalsIgnoreCase()

Giống với hàm equals(), hàm equalsIgnoreCase() sẽ so sánh không phân biệt in hoa hay in thường.

*Chuoi1.equalsIgnoreCase(chuoi2)*

### compareTo()

Phương thức so sánh này khá đặc biệt, nó sẽ lấy từng ký tự ra so sánh, với mỗi ký tự nó sẽ lấy giá trị Unicode ra rồi tiến hành phép trừ các ký tự này, một khi đã phát hiện ta ký tự khác biệt (phép trừ cho ra kết quả âm hay dương) thì sẽ ngừng việc trừ lại.

Cụ thể như ví dụ chuoi1.compareTo(chuoi2) thì.

– Kết quả trả về là một số âm khi chuoi1 có thứ tự ký tự trong Unicode nhỏ hơn chuoi2.

– Kết quả trả về là số 0 khi chuoi1 giống hệt chuoi2.

– Kết quả trả về là một số dương khi chuoi1 có thứ tự ký tự trong Unicode lớn hơn chuoi2.

Nối chuỗi: 2 cách, sử dụng toán tử “+” hoặc hàm concat

Ex:  *chuoi1.concat(chuoi2)*

### Rút Trích Chuỗi Con - ****subString(int startIndex)****

Nếu bạn gọi chuoi.subString(startIndex) thì kết quả sẽ trả về một Chuỗi con với nội dung bắt đầu từ chỉ số startIndex của chuoi, lưu ý là giống như với Mảng, chỉ số của các ký tự trong chuỗi được bắt đầu từ 0.

**subString(int startIndex, int endIndex) : lấy ra chuỗi con có vị trí index đầu cuối dc xác định trong ngoặc.**

## Một Số Phương Thức Thông Dụng Khác Của Chuỗi

### trim() : loại bỏ khoảng trắng ở đầu và cuối chuỗi(nếu có).

startsWith() Và endsWith() : trả về hàm Boolean với cú pháp để biết về vị trí bắt đầu, kết thúc có trùng với chuỗi trong ngoặc hay k.

charAt(): Phương thức này trả về ký tự của Chuỗi tại chỉ mục được truyền vào.

### replace() : Phương thức này có hai tham số truyền vào chính là hai Chuỗi, như này chuoi.replace(chuoi1, chuoi2). Nếu chuoi1 có tồn tại trong chuoi thì tất cả những nơi nào xuất hiện chuoi1 trong chuoi sẽ bị thay bằng chuoi2.

### indexOf() : Hàm này sẽ tìm trong Chuỗi vị trí xuất hiện đầu tiên của ký tự được truyền vào trong hàm.

# STRINGBUFFER AND STRINGBUILDER :

StringBuffer strBuffer\_1 = new StringBuffer(); // Khởi tạo một StringBuffer rỗng, với khả năng chứa đựng ban đầu là 16 ký tự

StringBuffer strBuffer\_2 = new StringBuffer(50); // Khởi tạo một StringBuffer rỗng, với khả năng chứa đựng ban đầu do bạn định nghĩa

StringBuffer strBuffer\_3 = new StringBuffer("Hello World!");

## Nối Chuỗi – append() : Giống như với hàm concat() trong String

## Chèn chuỗi – insert(vị\_trí\_chèn,chuỗi chèn) :

## Thay Thế – replace(vị\_trí\_bd, vị\_trí\_kthuc, chuỗi chèn)

## Xóa Chuỗi – delete(vị\_trí\_bd, vị\_trí\_kthuc)

# Đảo Ngược Chuỗi – reverse() : str.reverse();

## Kiểm Tra Dung Lượng Bộ Đệm – capacity() : Nếu bạn khởi tạo các đối tượng này rỗng, dung lượng mặc định ban đầu là 16.

## Lập trình hướng đối tượng:

## 1. Trạng Thái Của Đối Tượng

Trạng thái của đối tượng nói lên các đặc thù, hay đặc trưng của đối tượng đó.

## 2. Hành Vi Của Đối Tượng

Thì Hành vi chính là các hành động của Đối tượng đó, hay có thể hiểu đó là các hoạt động mà đối tượng đó có trách nhiệm phải thực hiện.

## Hình Dung Về Lớp: Tên lớp, các thuộc tính, các phương thức.

# Một số nguyên tắc khi khai báo ***kiểu\_trả\_về*** của một phương thức:

– Nếu như phương thức có khai báo ***kiểu\_trả\_về***, thì hiển nhiên phương thức đó *phải kết thúc với một hoặc nhiều câu lệnh* ***return***. Nếu không có ***return*** đối với phương thức này, hệ thống sẽ báo lỗi đấy nhé.

– ***kiểu\_trả\_về*** *của phương thức phải cùng kiểu với biểu thức (hoặc giá trị) sau câu lệnh* ***return***, như bạn đã thấy ở ví dụ trên. Nếu khác nhau về kiểu, sẽ bị báo lỗi luôn.

– Bạn có thể *[ép kiểu tường minh](http://yellowcodebooks.com/2016/10/25/java-bai-6-ep-kieu-comment-source-code/" \t "_blank)* một biểu thức (hoặc giá trị) trước khi ***return*** nó. Mình làm tạm một ví dụ sau.

## Nếu một phương thức có ***kiểu\_trả\_về*** là ***void***, thì có nghĩa là nó không có kiểu trả về.

## Tham số truyền vào cho từng phương thức chính là “cửa ngõ” cho các thành phần bên ngoài lớp có cơ hội truyền dữ liệu vào bên trong một lớp, giúp lớp đó có “nguyên liệu” để thực hiện các logic.

# Phương thức khởi tạo ( CONSTRUCTOR):

## Phương thức khởi tạo, hay gọi Hàm khởi tạo cũng được, bạn cũng có thể gọi là Constructor, mình thì mình sẽ dùng constructor luôn cho ngắn gọn. Thực chất thì constructor này cũng là một phương thức, nhưng nó đặc biệt ở chỗ là, ngay khi mà bạn khởi tạo một đối tượng bằng từ khóa new, thì constructor của đối tượng đó sẽ lập tức được gọi đến một cách tự động.

# Khai Báo Constructor

## [khả\_năng\_truy\_cập] tên\_phương\_thức () {

## // Các dòng code

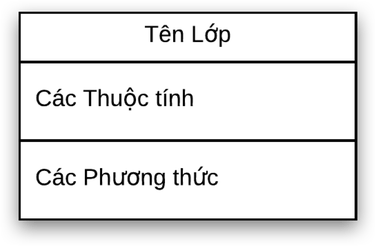
## }

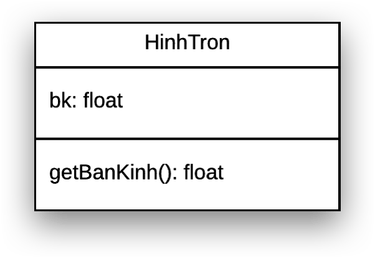
# Kế Thừa Là Gì?( Inheritance)

### Kế thừa trong lập trình hướng đối tượng ám chỉ đến một mối quan hệ giữa các đối tượng, có người thì nói mối quan hệ này là Cha-Con, có người thì nói là quan hệ Mở rộng. Người ta có vẻ thích cái khái niệm Cha-Con hơn, nhưng mình thấy Mở rộng, và sát nhất nên hiểu là Kế thừa luôn, thì sẽ hay hơn. Bởi vì nó thực chất là một sự dùng lại, một số trường hợp là mở rộng hơn các đặc tính, của một đối tượng từ một đối tượng nào đó khác.

### Mục đích chính mà kế thừa mang lại, đó là việc tận dụng lại, và mở rộng hơn các thuộc tính và phương thức có sẵn từ một đối tượng nào đó.

### Một lớp chỉ được phép kế thừa từ một và chỉ một lớp cha mà thôi.





# This and Super:

### Sử Dụng this Khi Gọi Đến Một Constructor Khác Bên Trong Lớp.

Có một điều lưu ý rằng, từ khóa **this()** dùng trong trường hợp này chỉ được dùng trong các constructor, để mà tận dụng lại các constructor khác như ví dụ trên, nếu bạn để this() này vào các phương thức bình thường khác, sẽ có báo lỗi xảy ra từ hệ thống. Thêm một điều nữa, là this() nếu có, thì nó phải là dòng code đầu tiên bên trong một constructor, trường hợp dưới đây là sai, hệ thống sẽ báo lỗi vì trước this() có các dòng code khác.

**Vd dung:**

 // Constructor

    public HinhChuNhat(float dai, float rong) {

        this(); // Gọi đến HinhChuNhat()

        this.dai = dai;

        this.rong = rong;

    }

**Vd sai:**

// Constructor

public HinhChuNhat(float dai, float rong) {

    this.dai = dai;

    this.rong = rong;

    this();

}

### SUPER lại giúp tham chiếu đến lớp cha, mà là lớp cha gần nhất của nó. Tức là bạn có thể hiểu vui là super không thể được sử dụng để tham chiếu đến lớp “ông nội” được nhé. Dưới đây tiếp tục là một vài vai trò của super, bạn có thể so sánh với vai trò của this trên kia để xem sự khác biệt, và để dễ nhớ nữa.

### Mục đích đầu tiên của super này là để phân biệt đâu là giá trị của lớp con và đâu là giá trị của lớp cha gần nhất khi chúng bị trùng tên.

# Object clone():

Phương thức này giúp khởi tạo và trả về một bản sao của đối tượng được gọi. Thực chất phương thức này mình chỉ nêu ra mà không có ví dụ cụ thể cho nó, vì hiện tại lớp Object đang khai báo Khả năng truy cập của phương thức này là protected. Bạn sẽ hiểu loại khả năng truy cập này sau, nhưng đại loại là nó được bảo vệ và chỉ hữu dụng khi lớp con của nó override phương thức này thôi.

# String toString():

Phương thức này giúp trả về một kiểu Chuỗi diễn đạt cho đối tượng này. Nội dung của nó là sự kết hợp của chuỗi (***getClass().getName() + “@” + Integer.toHexString(hashCode())***).

Có một điều thú vị rằng là, ***phương thức này có thể không cần phải gọi đến một cách tường minh***.

# – *Nếu private được chỉ định cho một thuộc tính*, thuộc tính đó sẽ không được phép truy cập, hay chỉnh sửa từ các lớp khác, trừ khi bạn xây dựng các phương thức *getter* và *setter* cho thuộc tính đó mà mình sẽ nói sau.

# – Còn *nếu private được chỉ định cho một phương thức*, phương thức đó sẽ không được truy cập hay kế thừa từ lớp khác.

# – Và *nếu private được chỉ định cho constructor*, constructor này sẽ không dùng được để khởi tạo đối tượng cho lớp đó.

# Khả Năng Truy Cập protected:

Khả năng này mang ý nghĩa bảo vệ các giá trị của lớp cho các lớp con của nó dùng lại hoặc *[ghi đè](https://yellowcodebooks.com/2017/07/27/java-bai-23-tinh-phu-quyet-overriding-trong-ke-thua/" \t "_blank)*. Do đó có thể hiểu, khả năng ***protected*** sẽ trao quyền sử dụng hoàn toàn tự do cho các lớp con, dù có ở cùng hay khác package. Khả năng này chỉ giới hạn với các lớp không phải lớp con của nó và nằm ngoài package mà thôi.

( su dung khi phai co tinh ke thua)

## Thuộc Tính final Trống:

## *khi bạn khai báo chúng là final, bạn không cần thiết phải chỉ định ngay giá trị cho thuộc tính hay biến* đó, mà có thể để trống, để sau này khi có giá trị cụ thể, thì bạn gán vào các thuộc tính hay biến final đó sau, và chỉ gán một lần duy nhất thôi nhé. Điều này giúp cho *final* tưởng như rất cứng nhắc này lại trở nên linh động hơn một chút.

## protected final float tinhChuVi() {

## final float PI; //chua can gan gia tri voi

## PI = tinhPI(); // gan cho PI mot constructor

## return 2 \* PI \* banKinh;

## }

# bạn có thể chỉ định ***final*** cho bất cứ phương thức nào. Nhưng, bạn nên lưu ý rằng, bạn ***không thể chỉ định final cho một constructor nhé***, vì constructor có bao giờ bị override đâu nào.

# ***final*** đến với lớp tương tự như tác dụng của em ấy đến với phương thức. Đó là ***bạn sẽ không thể kế thừa từ bất cứ lớp nào được khai báo là final***.

# Tổ Chức Các Phương Thức Getter Và Setter Như Thế Nào?

– Định nghĩa *[khả năng truy cập](https://yellowcodebooks.com/2017/08/11/java-bai-25-kha-nang-truy-cap-vao-cac-gia-tri-lop/" \t "_blank)* cho các thuộc tính trong một lớp về ***private***.  
– Xây dựng phương thức *getter* hay *setter* cho từng thuộc tính ***private*** đó.

public class ToaDo {

    private int x;

    private int y;

    // getter của thuộc tính x

    public int getX() {

        return x;

    }

    // setter của thuộc tính x

    public void setX(int x) {

        this.x = x;

    }

# Getter, Setter Và Những Tùy Biến:

 public void setTen(String ten) {

        if (ten == null || ten.isEmpty()) {

            // Nếu biến ten chưa khởi tạo (mang giá trị null), hoặc biến ten có nội dung rỗng

            // Thì hãy lưu với tên là "Không biết"

            this.ten = "Không biết";

        } else {

            this.ten = ten;

        }

    }

    public String getTuoi() {

        if (Integer.valueOf(tuoi) != -1) {

            // Tuổi hợp lệ

            return tuoi;

        } else {

            return "Tuổi không hợp lệ";

        }

    }

# Chuyển đổi 1 chuỗi sang số:

Dùng cấu trúc như sau:

double number = Double.parseDouble(String);

int number = Integer.parseInt(String);

float number = Float.parseFloat(String);

vd: chuoi = “2018”;

int number = Integer.parseInt(chuoi);

=🡺 number = 2018

# Abstract : là một lớp khai báo chưa hoàn thiện. Để hoàn thiện lớp này thì cần một subclass làm việc đó. Với Abstract thì không thể khai báo các object con.

trừu tượng ở chỗ, nó ***không thể được dùng để tạo ra các đối tượng như những lớp bình thường khác***. Lớp *Trừu tượng* khi này chỉ là cái *“xác không hồn”*, hay bạn có thể hiểu ***nó chỉ là một cái sườn, để mà bạn có thể tạo ra các lớp con của nó dựa vào sự ràng buộc từ cái sườn này***.

abstract class tên\_lớp {

các\_thuộc\_tính;

các\_phương\_thức;

}

Một khi có một lớp nào đó kế thừa từ lớp Trừu tượng này, thì lớp kế thừa đó buộc phải ***hiện thực*** (tiếng Anh gọi là ***implement***) nội dung cho ***tất cả*** các phương thức Trừu tượng này, bằng cách [override](https://yellowcodebooks.com/2017/07/27/java-bai-23-tinh-phu-quyet-overriding-trong-ke-thua/) lại chúng.

Lớp trừu tượng có thể chứa cả các abstract constructor(constructor không thể triển khai) và concret constructor(constructor có thể triển khai).

# Interface: được định nghĩa và tiêu chuẩn hóa theo cách làm trong đó những thứ như con người và hệ thống có thể tương tác với nhau.

Interface có thể được xem như thứ gì đó có tính kế thừa( multiple inheritance).

Chúng chứa các hàm không có nội dung trong thân.

Bạn không thể tạo ra các đối tượng của một interface nhưng có thể khai báo các biến kiểu interface.

Các subclass phải cung cấp sự thực thi của các hàm interface hoặc bản thân chúng được khai báo là abstract.

Chỉ sử dụng các biến constant trong interface, không sử dụng các biến instance.

Tất cả các biến trong một interface đều tự động là public static final.

Interface không có constructor nên không khai báo được constructor cho chính nó và cho class.

Khi muốn kế thừa từ 1 interface cho 1 class: dùng khai báo implements

Khi muốn kế thừa 1 interface cho 1 interface : dùng khai báo extends

interface chính là nơi nhận tương tác từ một đối tượng đến các đối tượng đang lắng nghe sự tương tác đó. Ngoài ra interface còn giúp định nghĩa sẵn các hành động, để các đối tượng bên ngoài biết đường tương tác, và các đối tượng bên trong hệ thống sẵn sàng đáp ứng các tương tác đó thông qua việc hiện thực sẵn các hành động đã được định nghĩa đó.

Bạn nên biết, một lớp có thể triển khai nhiều interface một lúc. Các interface được triển khai vẫn được khai báo sau từ khóa ***implements***, nhưng cách nhau bởi dấu phẩy (***,***). Điều này khác với tính chất của kế thừa, khi mà một lớp chỉ được kế thừa đến một lớp cha mà thôi, nếu quên thì bạn xem lại [bài 21](https://yellowcodebooks.com/2017/06/14/java-bai-21-ke-thua-lam-quen-voi-ke-thua/" \t "_blank) nhé. Ví dụ như, bạn có thể khai báo một lớp ***HinhTru*** triển khai từ nhiều interface như sau ***public class HinhTru extends HinhTron implements InterfaceA, InterfaceB***.

 Như bài thực hành trên bạn có thể thấy nếu một lớp triển khai từ interface mà là lớp trừu tượng, thì lớp đó không còn buộc phải hiện thực các phương thức trừu tượng từ interface đó.

Đầu tiên bạn có thể thấy rằng, interface giống với lớp trừu tượng ở chỗ nó giúp bạn tạo ra ràng buộc đối với các đối tượng bên trong hệ thống. Và nếu bạn cần một ràng buộc toàn vẹn, thì interface là một giải pháp tốt.  
– Bạn có thể tận dụng tính chất một lớp có thể triển khai nhiều interface trên kia để mở rộng hơn cho một số mục đích kế thừa.  
– Và cũng với ý mà mình nói từ đầu bài học, interface là một giao diện nhận tương tác từ bên ngoài, nên nếu bạn có xây dựng UI cho ứng dụng, hoặc bạn xây dựng các ứng dụng Android bằng Java như mình có nhắc đến, thì interface sẽ được tận dụng để xây dựng các sự kiện tương tác giữa người dùng và hệ thống, như sự kiện nhấn lên Button, sự kiện nhấn chọn một item list,..

# INNER CLASS:

Lớp Lồng Là Gì?

# là kiến thức về việc khai báo một lớp ở bên trong một lớp khác.

Lớp mà chứa các lớp khác bên trong nó người ta gọi là **Outer Class**, có thể hiểu theo tiếng Việt là lớp **Bao**.

Còn lớp ở bên trong ***Outer Class*** thì được chia ra làm hai loại khác nhau.

– Một là *lớp không-static (non-static class)*, người ta gọi loại này là ***Inner Class***.  
– Loại còn lại là *lớp static (static class)*, người ta gọi loại này là ***Static Nested Class***.

Khoan hãy nói đến vai trò của từng loại ***Inner Class*** và ***Static Nested Class***, chúng ta hãy đến với cú pháp khai báo của từng loại như sau.

Đây là cú pháp của một ***Inner Class***.

class OuterClass {

...

class InnerClass {

...

}

}

Còn đây là cú pháp của một ***Static Nested Class***.

class OuterClass {

...

static class StaticNestedClass {

...

}

}

Tuy cú pháp chỉ có một ***Outer Class*** chứa một ***Inner Class*** hoặc một ***Static Nested Class*** bên trong. Nhưng bạn nên biết rằng bên trong một ***Outer Class*** có thể chứa nhiều ***Inner Class***, nhiều ***Static Nested Class***, hoặc chứa nhiều cả ***Inner Class*** lẫn ***Static Nested Class***.  
*– Và tuy cú pháp chỉ nói đến*lớp lồng*, nhưng bạn có thể lồng*[interface](https://yellowcodebooks.com/2017/10/15/java-bai-33-interface/)*vào trong lớp, hoặc lồng*interface*vào với nhau, hay lồng lớp vào trong*interface*đều được nhé*.  
– ***Inner class*** lúc này được xem như một thành phần của ***Outer Class***, chính vì vậy bạn có thể chỉ định cho nó các [khả năng truy cập](https://yellowcodebooks.com/2017/08/11/java-bai-25-kha-nang-truy-cap-vao-cac-gia-tri-lop/" \t "_blank), như ***private***, ***public***, ***protected***, hoặc ***default*** (tức là không có khai báo khả năng truy cập gì). Điều này khác với ***Outer Class*** hay các lớp mà bạn đã từng làm quen, đều chỉ có thể khai báo ***public*** hoặc ***default*** (khai báo này chỉ cho phép các lớp cùng trong một [package](https://yellowcodebooks.com/2017/05/23/java-bai-19-package/) mới có thể nhìn thấy nhau) mà thôi.

 Nếu bạn có một lớp ***A*** chỉ dùng đến một lớp ***B*** nào đó. Tức là ***B*** không được ai dùng đến cả ngoại trừ ***A*** thôi. Thì bạn có thể xem xét tổ chức theo cách ***B sẽ là lớp lồng vào trong lớp A***. Lợi ích của việc này thì bạn có thể xem các ý dưới đây.  
– Việc lồng các lớp vào nhau giúp tăng tính gói gém dữ liệu (encapsulation). Chẳng hạn với việc lớp ***B*** nằm trong lớp ***A***, thì bạn có thể khai báo ***B*** là ***private***, khi đó ngoài ***A*** ra không có bất kỳ lớp nào khác có thể biết đến sự tồn tại của ***B*** và các giá trị của nó, có thể nói rằng bạn đã “giấu” ***B*** khỏi thế giới, ngoại trừ ***A***.  
– Ở chiều ngược lại. Lớp lồng ***B*** có thể truy cập đến tất cả các thành viên (thuộc tính và phương thức) của lớp bao ***A***, ngay cả khi các thành viên của ***A***được khai báo là ***private***.  
– Tất nhiên, khi lồng các lớp vào nhau như vậy, code của bạn sẽ dễ đọc hơn (vì không phải tìm và mở quá nhiều lớp), dẫn đến việc bảo trì cũng dễ dàng hơn.

# LỚP VÔ DANH LÀ GÌ?

Tất nhiên một lớp được gọi là Vô Danh thì có nghĩa là nó sẽ không có cái tên cụ thể. Lớp Vô Danh sẽ gắn liền với kế thừa (bao gồm cả kế thừa từ một lớp [cha bình thường](https://yellowcodebooks.com/2017/06/14/java-bai-21-ke-thua-lam-quen-voi-ke-thua/) hoặc lớp [cha trừu tượng](https://yellowcodebooks.com/2017/09/26/java-bai-32-tinh-truu-tuong-abstraction/)), và gắn liền với [interface](https://yellowcodebooks.com/2017/10/15/java-bai-33-interface/) nữa. Và bởi vì lớp Vô Danh còn có cái tên Anonymous Inner Class, nên nó cũng có liên quan chút đỉnh đến kiến thức về [lớp lồng](https://yellowcodebooks.com/2017/11/01/java-bai-34-lop-long/" \t "_blank)đấy nhé.

Lớp Vô Danh thường được dùng khi bạn không muốn phải khai báo cụ thể lớp con của một lớp nào đó (bao gồm cả lớp trừu tượng và lớp bình thường mà bạn biết), kể cả khi bạn không muốn khai báo cụ thể lớp triển khai của một interface nào đó, mà vẫn muốn sử dụng các đối tượng của chúng.

## Phân Loại Lớp Vô Danh

## Lớp Vô Danh Được Tạo Ra Thông Qua Kế Thừa Từ Lớp Khác

Nếu bạn có một lớp nào đó, dù cho đó là lớp thường hay lớp trừu tượng. Thì thay vì khai báo một lớp con của nó với tên tuổi hẳn hoi, bạn có thể khai báo một lớp con *Vô Danh*. Như sau này bạn cần phải kế thừa lớp ***Thread*** từ hệ thống như sau.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void main(String[] args) {      // Ví dụ về lớp Vô Danh được tạo ra      // thông qua kế thừa từ lớp Thread      Thread t = new Thread() {            @Override          public void run() {              System.out.println("Bạn có thể thử nghiệm thực thi code này");          }      };      t.start();  } |

Lớp Vô Danh Được Tạo Ra Thông Qua Triển Khai Từ Interface Khác

Mục này tương tự như ví dụ của bài học trên kia. Khi mà bạn có một interface, thay vì khai báo một triển khai rõ ràng từ interface này, bạn có thể áp dụng lớp *Vô Danh*.

Ví dụ sau đây sẽ triển khai từ interface có tên ***Runnable***. Bạn cũng sẽ được làm quen với code này ở bài học về *đa luồng (multi thread)* sau.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public static void main(String[] args) {      // Ví dụ về lớp Vô Danh được tạo ra      // thông qua triển khai từ Interface Runnable      Runnable r = new Runnable() {            @Override          public void run() {              System.out.println("Bạn có thể thử nghiệm thực thi code này");          }      };      Thread t = new Thread(r);      t.start();  } |

Lớp Vô Danh Được Dùng Như Một Tham Số Truyền Vào

Đây là phần mở rộng hơn so với hai cách trên, nếu bạn nào đang lập trình Android, sẽ thấy ứng dụng rất nhiều.

Ví dụ dưới đây cho thấy thay vì truyền ***r*** vào phương thức khởi tạo của ***Thread()***, bạn có thể tạo lớp *Vô Danh* như một tham số thay cho khai báo ***r***.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void main(String[] args) {      // Ví dụ về lớp Vô Danh được tạo ra      // như một tham số truyền vào      Thread t = new Thread(new Runnable() {            @Override          public void run() {              System.out.println("Bạn có thể thử nghiệm thực thi code này");          }      });      t.start();  } |

Các Đặc Điểm Của Lớp Vô Danh

– Nếu một lớp bình thường có thể triển khai bao nhiêu interface cũng được, thì ***lớp Vô Danh chỉ có thể triển khai từ duy nhất một interface*** mà thôi.  
– Nếu một lớp bình thường vừa có thể kế thừa từ một lớp nào đó, vừa có thể triển khai từ nhiều interface khác, thì ***lớp Vô Danh chỉ có thể hoặc là kế thừa hoặc là triển khai một lớp hay một interface khác thôi***.  
– Với một lớp bình thường, bạn có thể định nghĩa các [*constructor*](https://yellowcodebooks.com/2017/05/30/java-bai-20-phuong-thuc-khoi-tao-constructor/) tùy thích. Nhưng ***lớp Vô Danh lại không hề có bất kỳ constructor nào***. Điều này cũng dễ hiểu, vì constructor buộc phải có tên trùng với tên lớp, mà lớp *Vô Danh* thì không có tên, nên chẳng bao giờ bạn có thể định nghĩa được constructor cho nó.  
– Và vì một lớp *Vô Danh* được khai báo bên trong lớp khác, nên nó có một đặc điểm giống với *[lớp Lồng](https://yellowcodebooks.com/2017/11/01/java-bai-34-lop-long/" \t "_blank)*, ở chỗ nó có thể truy cập đến các thành viên của lớp bao của nó.

# WRAPPER LÀ GÌ?

các lớp *Wrapper* sẽ giúp chúng ta ***chuyển đổi qua lại giữa một kiểu dữ liệu nguyên thủy sang kiểu dữ liệu đối tượng và ngược lại***.

nếu với các kiểu dữ liệu nguyên thủy, bạn chỉ có một chọn lựa là tạo ra biến rồi sử dụng giá trị của nó (nếu bạn không gán giá trị thì nó vẫn được tạo một giá trị mặc định, bạn có thể xem lại ở bài *[các kiểu dữ liệu nguyên thủy](https://yellowcodebooks.com/2016/10/09/java-bai-4-bien-va-hang-trong-java/" \l "Kieu_Du_Lieu_Cua_Bien" \t "_blank)*). Còn với các kiểu đối tượng, giá trị mặc định của nó là ***null***, giá trị ***null*** này có thể được tận dụng trong một số trường hợp, như mang ý nghĩa chưa có giá trị chẳng hạn.

một số cấu trúc khác bên trong ngôn ngữ Java, như các cấu trúc về các danh sách mà chúng ta sẽ làm quen sau như ***ArrayList*** hay ***Vector*** đều chứa đựng các tập hợp kiểu dữ liệu đối tượng thay vì kiểu nguyên thủy, nên việc biết và vận dụng các lớp *Wrapper* là một bắt buộc.

Chuyển Đổi Kiểu Nguyên Thủy Sang Kiểu Wrapper

Việc chuyển đổi một kiểu nguyên thủy sang kiểu *Wrapper* của nó người ta gọi là ***Boxing***. Không phải mang ý nghĩa là môn đấm bốc đâu. Boxing ở đây mang ý nghĩa là ***đóng hộp***, tức là đóng dữ liệu nguyên thủy vào trong cái hộp *Wrapper* của nó đấy.

// Các dạng Boxing

int a = 500;

Integer i = new Integer(a);

Integer j = new Integer(500);

Float f = new Float(4.5);

Double d = new Double(5);

Character ch = new Character('a');

Boolean b = new Boolean(true);

// Các dạng Autoboxing

int a = 500;

Integer i = a;

Integer j = 500;

Float f = 4.5f;

Double d = 5d;

Character ch = 'a';

Boolean b = true;

// Đây cũng là một dạng Autoboxing mà bạn sẽ được biết khi học đến bài về Collection

ArrayList<Integer> arrInt = new ArrayList<Integer>();

arrInt.add(25);

Chuyển Đổi Kiểu Wrapper Sang Kiểu Nguyên Thủy

Ngược lại với trên kia, khi bạn chuyển từ một kiểu *Wrapper* sang kiểu nguyên thủy của nó người ta gọi là ***Unboxing***, có nghĩa là mở hộp, tức là mở cái hộp *Wrapper* để lấy dữ liệu nguyên thủy ra.

Bạn có thể thực hiện việc unboxing thông qua các phương thức ***xxxValue()***. Với ***xxx***là đại diện cho từng loại dữ liệu, như với ví dụ sau.

int a = 500;

Integer i = a; // Autoboxing

int i2 = i.intValue(); // Unboxing

Integer j = 500; // Autoboxing

int j2 = j.intValue(); // Unboxing

Float f = 4.5f; // Autoboxing

float f2 = f.floatValue(); // Unboxing

Double d = 5d; // Autoboxing

double d2 = d.doubleValue(); // Unboxing

Character ch = 'a'; // Autoboxing

char ch2 = ch.charValue(); // Unboxing

Boolean b = true; // Autoboxing

boolean b2 = b.booleanValue(); // Unboxing

ArrayList<Integer> arrInt = new ArrayList<Integer>();

arrInt.add(25); // Autoboxing

int arr0 = arrInt.get(0).intValue(); // Unboxing

int a = 500;

Integer i = a;

int i2 = i; // Unboxing

Integer j = 500;

int j2 = j; // Unboxing

Float f = 4.5f;

float f2 = f; // Unboxing

Double d = 5d;

double d2 = d; // Unboxing

Character ch = 'a';

char ch2 = ch; // Unboxing

Boolean b = true;

boolean b2 = b; // Unboxing

ArrayList<Integer> arrInt = new ArrayList<Integer>();

arrInt.add(25);

int arr0 = arrInt.get(0); // Unboxing

# PACKAGE :

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

In general, a company uses its reversed Internet domain name for its package names.

**Example** − A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Set CLASSPATH System Variable

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell) −

* In Windows → C:\> set CLASSPATH
* In UNIX → % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use −

* In Windows → C:\> set CLASSPATH =
* In UNIX → % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable −

* In Windows → set CLASSPATH = C:\users\jack\java\classes
* In UNIX → % CLASSPATH = /home/jack/java/classes; export CLASSPATH

# Method Overloading:

Means providing two or more separate methods in a class with the same name but different parameters.

Method return type may or may not be different and that allows us to reuse the same method name.

Methods will be considered overloaded if both follow the following rules:

* Methods must have the same method name.
* Method must have different parameters.

# Method Overriding:

Method overriding means defining a method in a child class that already exist in the parent class with same signature (same name , same arguments)

We cant override static method, only instance methods.

Method will be considered overridden if we follow these rules:

It must have same name and same arguments

Return type an be a subclass of the return type in the parent class

It cant have a lower access modifier

For ex: if the parent method is protected the using private in the child is not allowed but using public in the child would be allowed.

Constructor and private method can not be overridden

# COMPOSITION

# Exception:

***Exception*** có thể dịch ra tiếng Việt là ***Biệt lệ*** hoặc ***Ngoại lệ***.

***Khi Exception diễn ra, nó làm cho chương trình bị lệch khỏi luồng chuẩn mà chúng ta đã lập trình ra***, dẫn đến việc ứng dụng không thể xử lý được những logic của nó, hoặc có thể bị ngưng đột ngột (tình trạng này có thể gọi là *“chết”*, *“đột tử”*, hay *“crash”*).

* *Exception* có thể xảy ra khi người dùng nhập dữ liệu sai lệch vào cho ứng dụng. Do vô tình hay cố ý.
* Hoặc khi ứng dụng tìm đọc một file nào đó mà lại không thể thấy được. Có thể file đó đã bị xóa trước đó rồi.
* Hay hệ thống bị hết bộ nhớ trong lúc ứng dụng đang chạy khiến cho ứng dụng không thể thực thi được các logic của nó.

**Java chia nó ra làm *3* loại như sau:**

* Checked Exception: Đây là những Exception không vượt qua được “cửa ải” của trình biên dịch.
* Unchecked Exception: Exception này khá nguy hiểm, khi mà trình biên dịch không thể nào kiểm tra giúp bạn sự lai lệch luồng có thể xảy đến như trên kia. Bởi vì *Exception* này xảy ra khi ứng dụng đang được thực thi, nên nó còn được gọi là ***Runtime Exception***.
* Error: Error ở đây cũng là Exception nhưng hơi khác với Exception một chút. Với hai loại Exception mà mình nói đến trên đây thực ra chưa phải là lỗi, bởi vì bạn không nên né tránh nó mà nên đối mặt và “bẻ” luồng ứng dụng sao cho khi gặp Exception này ứng dụng sẽ biết cách phản hồi đúng đắn.

# Cây Phân Cấp Các Lớp Exception Trong Java: