

# Object Oriented Programming in Java

## Cheat Sheet - Tóm lược nội dung

TÓM LƯỢC NỘI DUNG MÔN HỌC JAVA OOP, ĐẠI HỌC FPT-HCM

BIÊN SOẠN BỞI GIÁO-LÀNG (FB/GIAO.LANG.BIS)

PHIÊN BẢN 19.0729.22 (TIẾP TỤC CẬP NHẬT...),

DOWNLOAD TẠI [HTTPS://GITHUB.COM/DOIT-NOW](https://github.com/doit-now)

### NỘI DUNG

- Tổng quan môi trường Java (Getting Started)
- Thành phần ngôn ngữ (Java Language)
- Lớp đối tượng (Class)
- Số và Chuỗi (Numbers and Strings)
- Collection Framework
- Tập tin (File)
- Ngày tháng (Date-Time API)
- Biểu thức Lambda
- Stream API
- Đóng gói (Deployment)
- Ngoại truyện

## Lời phi lộ

© 2019 giáo-làng | DoIT-Now

Bạn đang cầm trên tay bản Tóm lược nội dung môn học Java OOP (Java OOP Cheat Sheet) thuộc series bài viết **Cheat Sheet for...** với mục đích giúp các bạn sinh viên tổng kết kiến thức các môn học cuối học kỳ.

Tài liệu này là một phần của dự án **DoIT-Now** do giáo-làng chủ biên, với mong muốn chia sẻ miễn phí các kiến thức về IT (Do IT), về kĩ năng sống (Do it) một cách đơn giản, dễ hiểu, vui vẻ, làm được ngay và luôn.

Mọi kiến thức ghi ra ở đây đến từ sự trải-nghiệm cá nhân, đến từ việc học-hỏi, sưu-tầm, tham-khảo từ đồng nghiệp, bạn bè, các thế hệ sinh viên, và từ nguồn chia sẻ "khùng" trên Internet của "bá tánh".

Đa tạ tất cả!

Xin nhận mọi gạch đá, phản biện để có được bản tài liệu ngày càng hoàn thiện hơn ngõ hầu phục vụ quý bạn đọc gần xa.

Trân trọng,

giáo-làng | [fb/giao.lang.bis](https://fb/giao.lang.bis) | <https://github.com/doit-now>

**PS:** Bạn không cần cảm ơn giáo-làng. Nếu thấy tài liệu này có chút hữu ích, hãy chia sẻ nó **miễn phí** tới mọi người...

## Tổng quan môi trường Java (Getting Started)

Sau đây là một số định nghĩa xoay quanh môi trường Java khi lần đầu tiên tiếp xúc với nó.

### XUẤT XỨ TÊN GỌI JAVA

Tên hòn đảo trồng nhiều café ở Indonesia.

### NGÔN NGỮ LẬP TRÌNH JAVA

Là tên gọi của một ngôn ngữ lập trình hướng đối tượng được công bố vào năm 1995. Tên gọi lấy cảm hứng từ loại café (đến từ hòn đảo mà ai cũng biết) nhóm phát triển ra ngôn ngữ này nghiện uống.

### ĐẶC ĐIỂM NỔI BẬT

Ngôn ngữ mức cao/high-level language có các đặc điểm nổi bật (characteristics):

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic linking
- Architecture neutral
- Portable
- High performance
- Robust
- Secure

### PLATFORM LÀ GÌ?

Tên gọi chung của một tổ hợp được tạo nên bởi phần cứng, phần mềm, hệ điều hành mà một chương trình máy tính chạy trên nó. Nó cung cấp một nền tảng, môi trường/không gian để app có thể chạy.

## JAVA PLATFORM

Bao gồm máy ảo Java Virtual Machine (JVM) và tập hợp các hàm/công cụ (Application Programming Interface - API) giúp lập trình viên viết code, tạo app chạy trên/và tương tác với máy ảo Java.

### Các bước viết và chạy app Java

- Lập trình/viết app, tạo ra mã nguồn của app (source code), là các tập tin **.java**
- Dịch (compile) bởi tool **javac** (trong bộ JDK) ra mã **byte-code**, là các tập tin **.class** (không được xem là file nhị phân hoàn chỉnh kiểu .exe bên Windows)
- Máy ảo Java hiểu mã byte-code và chuyển dịch thành lệnh tương ứng với hệ điều hành mà app đang chạy

### Platform-independence

Độc lập môi trường vận hành nghĩa là ngôn ngữ Java được thiết kế để app viết bằng ngôn ngữ này – file **.java**; và sau đó nó dịch thành file file **.class**, có thể chạy không phụ thuộc vào hệ điều hành (OS) và phần cứng ứng với hệ điều hành đó bởi nó đã được “che” bởi Java Virtual Machine (JVM – máy ảo Java). Vậy khi chạy app Java ta chỉ cần máy ảo tương ứng với OS mà app Java sẽ chạy trên đó.

Hiện có máy ảo cho MacOS, Linux, Windows... Khi đó lập trình viên chỉ quan tâm viết code bằng ngôn ngữ Java và dịch code ra mã byte-code, phần thực thi với OS nào do máy ảo “đảm nhiệm”, do đó lập trình viên không cần viết nhiều phiên bản app tương ứng với các OS khác nhau, khái niệm này gọi là “**write once, run anywhere**”.

## JRE

Để app Java chạy được trên máy bạn, ta cần cài đặt các “đồ chơi” cần thiết bao gồm máy ảo Java – JVM và

phụ kiện (thư viện, library, dependency). Đám “đồ chơi” này gọi là môi trường thực thi Java – Java Runtime Environment.

## JDK

Để viết code và chạy thử nghiệm app Java, lập trình viên cần nhiều đồ chơi hơn nữa, dĩ nhiên phải bao gồm luôn JRE, bộ đồ “khủng” này được gọi là JDK (Java Development Kit – bộ công cụ phát triển phần mềm Java). Do đó  $JDK = JRE + \dots + \dots$

**Chốt hạ: Muốn chạy app Java cần tối thiểu JRE. Muốn viết app Java bắt buộc cần JDK.**

## Hàm main()

Entry point of a Java program - cửa chính để đi vào app Java. CPU sẽ tìm các câu lệnh ở trong hàm main() này để bắt đầu thực thi app Java.

Cú pháp hợp lệ của hàm main():

```
public static void main(String[] args) {...}
static public void main(String[] args) {...}
```

**args** chính là mảng các tham số đưa vào khi chạy app ở chế độ dòng lệnh. Ta có quyền sử dụng các giá trị truyền vào cho hàm main() ở mức gọi từ dấu nhắc hệ điều hành. Hãy xem app Hello được chạy ở dấu nhắc lệnh (cmd)

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("1st input: " + args[0]);
    }
}
```

```
D:\Hello\build\classes>java hello.Hello Ahihi Ahuhu
1st input: Ahihi
```

-oOo-

# Thành phần ngôn ngữ (Java Language)

## Kiểu dữ liệu (DATA TYPES)

Java cung cấp hai cách thức “lớn” để lưu trữ dữ liệu – 2 kiểu dữ liệu: nguyên thủy (primitive data types) và tham chiếu (reference data types, object data types).

### Primitive Data Types

Kiểu dữ liệu nguyên thủy dùng một lượng nhỏ bộ nhớ để biểu diễn một giá trị đơn, không thể chia nhỏ. Kiểu dữ liệu này được đặt tên bằng cụm chữ thường. Có 8 loại dữ liệu nguyên thủy: byte (1 byte), short (2 byte), int (4 byte), long (8 byte), float (4 byte), double (8 byte), char (2 byte hỗ trợ Unicode), boolean (true/false, không chỉ định rõ kích thước, hoặc 1 bit).

Kiểu dữ liệu	Số bit	Giá trị nhỏ nhất	Giá trị lớn nhất
byte	8	-128 ( $-2^7$ )	127 ( $2^7-1$ )
short	16	-32,768 ( $-2^{15}$ )	32,767 ( $2^{15}-1$ )
int	32	-2,147,483,648 ( $-2^{31}$ )	2,147,483,647 ( $2^{31}-1$ )
long	64	-9,223,372,036,854,775,808 ( $-2^{63}$ )	9,223,372,036,854,775,807 ( $2^{63}-1$ )
float	32	$-3.4028235 \times 10^{38}$	$3.4028235 \times 10^{38}$
double	64	$-1.7976931348623157 \times 10^{308}$	$1.7976931348623157 \times 10^{308}$
boolean	false	true	
char	16	'\u0000' (0)	'\uffff' (65,535).

Mỗi primitive có tương ứng một kiểu tham chiếu gọi là Wrapper Class [\[Xem thêm ở mục Wrapper Class\]](#).

**LƯU Ý:** Khi nói về giá trị mặc định (default value) ở biến kiểu primitive, ta sẽ nói đó là giá trị 0 (với kiểu số, chữ), giá trị false (với kiểu boolean).

### Object Data Types

Các primitive có thể được gom lại với nhau để tạo ra những kiểu dữ liệu phức tạp hơn, gọi là object data types. Object data types chứa bên trong nhiều value phức hợp, ví dụ kiểu dữ liệu Student (do người dùng

tự định nghĩa) sẽ chứa bên trong nó những thông tin như mã số, tên, ngày sinh, địa chỉ... Tương tự cho các kiểu dữ liệu Person - người dùng tự tạo; File, Scanner – có sẵn trong Java... “Sờ, chạm” đến thông tin bên trong vùng phức hợp này qua dấu chấm, ví dụ:

```
Student s1 = new Student("SE12345", "An
Nguyễn", 2000, 9.5);
System.out.println(s1.getName());
//lấy và in ra tên của sinh viên An Nguyễn
```

**LƯU Ý:** Biến object khi không muốn lưu trữ giá trị gì cả, hoặc thiết lập giá trị mặc định cho nó, ta sẽ nói nó trở về null.

[\[Xem thêm ở mục Class\]](#)

## JDK10 - LOCAL VARIABLE TYPE INFERENCE

Từ JDK10, Java hỗ trợ khả năng khai báo biến cục bộ trong hàm mà không cần chỉ rõ data type một cách tường minh – type inference (suy luận kiểu) bằng cách sử dụng từ khoá **var**. Java Compiler sẽ dựa trên cách khai báo biến và gán giá trị để suy luận ra kiểu dữ liệu của biến.

### Một số lưu ý khi dùng var

> Bắt buộc phải gán giá trị cho biến khi khai báo có dùng **var**

```
var x; //bị báo lỗi
var y = "Do IT Now"; //hợp lệ
```

> Không thể gán giá trị null cho biến lúc khai báo

```
var x = null; //bị báo lỗi
```

> Không đổi kiểu dữ liệu ban đầu của biến sang kiểu khác

```
var x = 10;
x = "Hello"; //bị báo lỗi tương thích kiểu
dữ liệu do x đã được xác định là int
```

> Không dùng **var** để khai báo đặc điểm/field/attribute của class. **var** chỉ được dùng làm biến cục bộ trong hàm, trong vòng lặp.

```
public class Var {
    public static void main(String[] args) {
        var javaVer = "Java 10"; //biến javaVer có kiểu là String
        var nameList = new ArrayList<String>(); //biến nameList chứa 1 danh sách chuỗi
        var n = 6869; //biến n bản chất là biến int

        var mixedList = new ArrayList(); //biến mixedList chứa 1 danh sách Object
        mixedList.add(68);
        mixedList.add("Do IT now");
        mixedList.add(3.14);
        mixedList.add('$');

        System.out.println("Java version: " + javaVer);
        System.out.println("n has value of: " + n);

        System.out.println("The list has the following items");
        for (var x : mixedList) {
            System.out.println(x); //trình biên dịch sẽ tự suy ra kiểu của từng
            //Object được add() vào trong list
        }
    }
}
```

## Sử dụng Primitive Data Types

Mặc định Java ưu tiên số kiểu **int**, **double**. Do đó nếu muốn dùng số kiểu long, float phải lưu ý hậu tố/suffix.

Câu lệnh hợp lệ khi dùng kiểu long với số lớn

```
long n1 = 3000000000L;
//mặc định mọi số nguyên là int. Số 3 tỷ
tràn miền int nên phải ghi hậu tố L ám chỉ
đang nói số long. Có thể dùng 1 thường
```

Câu lệnh hợp lệ khi dùng kiểu float

```
float n2 = 3.14F;
//mặc định mọi số thập phân là double 8
byte, bị ép về 4 byte để gán vào float thì
phải nói rõ. Có thể dùng f thường
```

Câu lệnh hợp lệ khi dùng số lớn

```
long n3 = 3_000_000_000L;
//Dấu gạch dưới _ dùng phân cách phần ngàn
để dễ đọc code. Không xuất hiện khi in ra
màn hình
```

Câu lệnh hợp lệ với số hệ thập lục phân (hex)

```
int n4 = 0xFA;
//0x là tiền tố/prefix ám chỉ con số nguyên
ghi dưới dạng hệ 16 (hexa)
```

Câu lệnh hợp lệ với số hệ bát phân (oct)

```
int n5 = 077;
//0 là tiền tố ám chỉ con số nguyên ghi dưới
dạng hệ 8 (octal)
```

Câu lệnh bị báo lỗi

```
int n6 = 091;
//báo lỗi vì 9 không thể xuất hiện trong hệ
8. Hệ 8 chỉ dùng 0, 1, ... 7 để biểu diễn các
con số
```

## TOÁN JỬ (OPERATORS)

### Math Operators (phép toán truyền thống)

Các phép toán số học giống như bên C nên không đề cập ở đây.

```
+, -, *, /, % (lấy dư), ++, --, +=, -=, *=,
/=...
```

### Shift Operators <<, >>

Một con số trong hệ thập phân có thể được tăng giảm giá trị thông qua phép đẩy các bit của nó khi xét nó được biểu diễn dưới dạng số nhị phân.

Có hai phép toán đẩy/dịch bit. Phép dịch bit >> đẩy sang phải, << đẩy sang trái một số lần các bit nhị phân của con số nào đó trong hệ 10. Đẩy mãi thì con số gốc sẽ còn lại toàn bit 0, tức là số gốc sẽ tiến về 0.

Nguyên tắc đẩy là hụt thì bù 0, lỏ/té thì mất luôn.

Làm dạng bài tập đẩy bit bắt buộc phải đổi con số hệ 10 sang dạng nhị phân, và biết trước tổng số bit để biểu diễn con số này trong hệ 10. Biết tổng số bit để ta còn bù 0 khi bị hụt, cho bit “té” đi mất khi bị dồn đẩy ra ngoài biên.

**Xem kết quả đẩy con số 7 (hệ 10) sang phải 2 lần sẽ được con số mấy? Biết rằng dùng 4 bit để lưu con số 7.**

```
7 >> 2
```

Bài giải:

Số 7 biểu diễn dưới dạng nhị phân 4 bit có dạng 0111. Số 7 cần đẩy bit của nó sang phải 2 lần. Khi đẩy sang phải 2 lần, bit nào bên phải nhất bị té khỏi vùng 4 bit thì mất luôn, song song việc bị té thì bit bên trái bị hụt, sẽ được bù 0. Vậy sau hai lần đẩy sang phải 0111 còn lại 0001 nhị phân (hai số 11 bị té, hụt hai số 00). Đổi 0001 nhị phân thành số 1 trong hệ 10. Vậy 7 đẩy sang phải 2 bit sẽ thành con số 1.

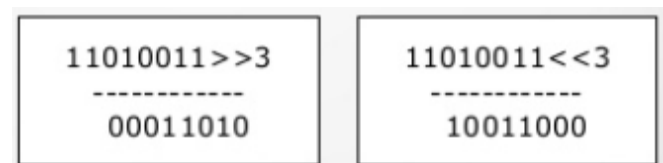
**Xem kết quả đẩy con số 7 sang trái 2 lần sẽ được con số mấy? Biết rằng dùng 4 bit để lưu con số 7.**

```
7 << 2
```

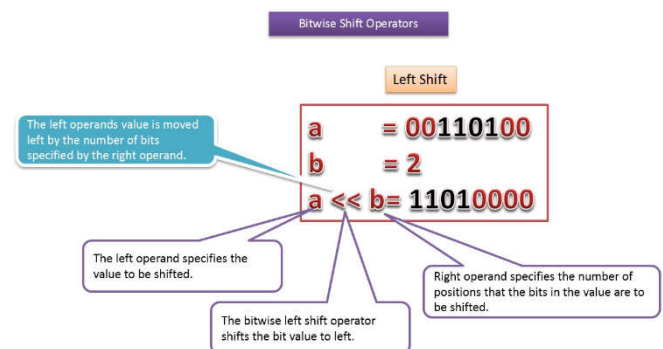
Bài giải:

Số 7 biểu dưới dạng nhị phân 4 bit có dạng 0111. Số 7 cần đẩy bit của nó sang trái 2 lần. Khi đẩy sang trái 2 lần, bit nào bị té (bên trái) khỏi vùng 4 bit thì mất luôn, song song bị té thì 2 bit bên phải lúc này lại bị hụt, được bù 00. Vậy sau hai lần đẩy sang trái 0111 ta có 1100 nhị phân, tức là số 12 trong hệ 10. Vậy 7 đẩy sang trái 2 bit sẽ thành con số 12.

Xem thêm ví dụ dưới đây



<https://www.slideshare.net/tusharkute/0-module-bitwise-operators>



Source: <https://ramj2ee.blogspot.com/2015/11/java-tutorial-java-bitwise-left-right.html>

**Bitwise (&, |, ^, ~)**

Là các phép tính toán khi thực hiện sẽ thao tác trên mức bit, tức là 2 mức 0, 1.

Muốn làm dạng bài này phải đổi các toán hạng/con số (operand) từ dạng thập phân sang dạng nhị phân, và tiến hành làm phép toán trên từng cặp bit tương ứng về vị trí. Kết quả cuối cùng đổi ngược lại về thập phân. Phải cho biết trước số bit dùng biểu diễn con số dưới dạng nhị phân và thao tác các bit theo bảng chân trị dưới đây.

Bảng chân trị của phép toán Bitwise

a	b	a&b	a b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

**MỌI NHỚ:**

- **& (AND):** chỉ cần nhớ kết quả của phép toán & là 1 nếu cả hai toán hạng là 1, các trường hợp còn lại kết quả là 0
- **| (OR):** chỉ cần nhớ kết quả của phép toán | là 1 nếu có ít nhất một trong hai toán hạng là 1, cả 2 toán hạng là 1 càng tốt
- **^ (XOR):** kết quả phép toán là 1 nếu hai toán hạng khác nhau, các trường hợp còn lại cho ra kết quả 0
- **~ (NOT):** nghịch đảo, đảo của 1 là 0, đảo của 0 là 1

**5 & 5 là mấy (biểu diễn bằng 4 bit)?**

**Bài giải:**

5 ở dạng nhị phân 4 bit là 0101, vậy

```

      0 1 0 1
    &
      0 1 0 1
    -----
      0 1 0 1
  
```

Kết quả là 0101, tức là 5 & 5 trả kết quả là 5 ở hệ 10.

Làm tương tự cho các phép toán còn lại dựa vào bảng chân trị ở trên.

Xem thêm ví dụ dưới đây

```

~11010011
-----
00101100

11010011
&
10001100
-----
10000000

11010011
|
10001100
-----
11011111

11010011
^
10001100
-----
01011111
  
```

<https://www.slideshare.net/tusharkute/0-module-bitwise-operators>

**MẢNG (ARRAY)**

Là kĩ thuật gom nhiều dữ liệu cùng kiểu vào chung một chỗ, đặt sát nhau như “cá mèi đóng hộp”. Nói cách khác, mảng là kĩ thuật khai báo nhiều biến cùng một lúc, cùng một kiểu, ở sát nhau, và chung một tên.

Có hai loại mảng: mảng primitive và mảng object.

Tên mảng là một biến object, biến đối tượng, biến tham chiếu, cho dù trong mảng có thể đang gom đám primitive hay đám object khác.

Khai báo mảng không hợp lệ

```

int[10] a1;
int a2[10];

//kích thước mảng chỉ được đưa vào qua toán tử new
  
```

Một vài khai báo mảng hợp lệ

```
int[] a1;
a1 = new int[10];
//khai báo mảng trước, xin kích thước sau

int a2[] = new int[10];
//vừa khai báo mảng vừa xin kích thước

int[] a3 = {1,2,3,4,5};
//vừa khai báo mảng vừa gán giá trị. Kích
thước mảng bằng chính số giá trị đã gán vào
mảng

int a4[] = {1,2,3,4,5};
//dấu [] đặt ngay sau kiểu dữ liệu hoặc sau
tên biến đều được
```

Phải new số phần tử, số biến của mảng trước khi sử dụng mảng. Nếu không new thì phải gán giá trị cho mảng ngay lúc khai báo mảng.

Phần tử trong mảng tính từ/đếm từ 0.

Tên mảng, ví dụ a1, a2 ở trên, luôn là biến tham chiếu, nghĩa là có thể chấm và “bung lụa”.

```
a1.length sẽ nhận về số 10 là kích thước
mảng, nghĩa là số biến/số phần tử có trong
mảng
```

Mảng chứa các đối tượng, ví dụ mảng Student, sẽ có thêm một tầng dấu chấm – đi vào sâu bên trong object. Nói cách khác, mảng object chính là mảng của các con trỏ. Ví dụ

```
Student s[] = new Student[100];
//có 100 Sinh viên sẽ được lưu trữ

//các lệnh gán giá trị cho mảng ở đây, ví dụ
s[0] = new Student("SE9999", "Ngọc Trinh",
4.9);

s[0].getName();
//lấy tên của Sinh viên đầu tiên, là Ngọc
Trinh
```

## CHUỖI KÍ TỰ (STRING)

Là kiểu dữ liệu object. Có nhiều cách khai báo và gán chuỗi. Các lệnh khai báo chuỗi sau đều hợp lệ

```
String x = null; //chuỗi null, trỏ vùng null

//4 lệnh dưới đây đều tạo vùng object String
trong RAM
String x = ""; //chuỗi rỗng, không chứa gì
String x = "Ahihi"; //dùng literal, POOL
String x = new String(); //chuỗi rỗng
String x = new String("Ahuhu");
```

**LƯU Ý:** Biến chuỗi khi không muốn lưu trữ giá trị gì cả, hoặc thiết lập giá trị mặc định, ta sẽ cho nó trỏ về null.

### Ghép chuỗi

Dùng dấu + để ghép các chuỗi con thành chuỗi tổng

```
String msg = "Hello" + " Java";
//kết quả có chuỗi "Hello Java"
```

### Lấy kí tự tại vị trí thứ x trong chuỗi

```
msg.charAt(0);
//lấy ra kí tự/chữ H. Thứ tự kí tự trong
chuỗi tính từ 0
```

### So sánh chuỗi

```
msg.equals(chuỗi khác)
msg.compareTo(chuỗi khác)
```

So sánh chuỗi lưu ý kèm thêm có quan tâm/phân biệt chữ hoa chữ thường hay không – IgnoreCase

### In chuỗi

```
System.out.println("Do It Now");
System.out.println(msg);
System.out.println("Hello" + " Java");
```

[\[Xem thêm ở mục Số và Chuỗi\]](#)

-oOo-

## Lớp đối tượng (Class)

### NGUYÊN LÝ THIẾT KẾ OOP

Lập trình hướng đối tượng dựa trên 4 nguyên lý sau

- Abstraction (trừu tượng hoá)
- Encapsulation (đóng gói)
- Inheritance (kế thừa)
- Polymorphism (đa xạ, đa hình)

### CLASS/OBJECT

Đối tượng, hay object là những cái gì cụ thể, thấy được, đếm được, sờ được, nhìn được, nhận dạng được, ví dụ trong hình dưới đây có 3 đối tượng cờ-hó – là 3 con vật cụ thể

Các đối tượng khả nghi (object) cần được theo dõi/lưu trữ thông tin, theo dõi hành vi		
		
<b>Đặc điểm nhân dạng (characteristics)</b> <ul style="list-style-type: none"> <li>Name: Ngáo Cỏ</li> <li>Weight: 50.0 kg</li> <li>Hair Color: Hung đỏ</li> <li>Breed: Ngao</li> </ul> <b>Hành động (methods)</b> <ul style="list-style-type: none"> <li>Rượt mèo(): rượt mèo rừng</li> <li>Sủa(): nhẹ nhẹ</li> </ul>	<b>Đặc điểm nhân dạng (characteristics)</b> <ul style="list-style-type: none"> <li>Name: Ngáo Đá</li> <li>Weight: 60.0 kg</li> <li>Hair Color: Đen-vàng</li> <li>Breed: Ngao</li> </ul> <b>Hành động (methods)</b> <ul style="list-style-type: none"> <li>Rượt mèo(): rượt mèo nhà</li> <li>Sủa(): dữ dằn</li> </ul>	<b>Đặc điểm nhân dạng (characteristics)</b> <ul style="list-style-type: none"> <li>Name: Chi Hu Hu</li> <li>Weight: 1.0 kg</li> <li>Hair Color: Café sữa</li> <li>Breed: ChiHuaHua</li> </ul> <b>Hành động (methods)</b> <ul style="list-style-type: none"> <li>Rượt mèo(): rú mèo đi chơi</li> <li>Sủa(): không ra hơi</li> </ul>
object	object	object

Dog
- Name: ???
- Weight: ???
- Hair Color: ???
- Breed: ???
- ...
+ Rượt mèo(): ???
+ Sủa(): ???
+ ...

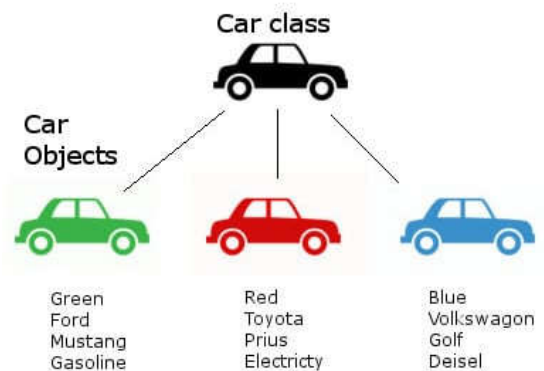
Class

Giữa một đám lộn xộn đối tượng gồm đồ vật, con vật, sự kiện, con người...ta nhận ra có những đám object thuộc về một nhóm nào đó bởi vì giữa chúng có sự tương đồng/đồng dạng/tựa tựa nhau. Ta có thể phân nhóm, phân cụm, phân hạng, chia nhóm một đám object dựa trên sự tương đồng, nhóm này đặt cho một cái tên chung, vậy tên chung này được gọi là class đại diện cho nhóm đó.

Đám vài chục “đứa” chạy bằng 4 chân, kêu gâu-gâu chứ không phải kêu miu-miu, cứ gập con miu-miu là rượt, cắn cổ, bao gồm cả 3 object ở hình trên, đích thị đám này thuộc nhóm, thuộc cụm, thuộc class Dog.

Theo chiều ngược lại, class Dog là một khái Khuôn chung để đúc ra đám con gâu-gâu Chi Hu Hu, Ngáo Đá (chó Ngao), Misa, Milu, Vàng O’i. Đám con gâu-gâu này gọi là object, đối tượng, instance, thể hiện, hiện-thân, xuất thân từ thể giới nhà Dog.

**Chốt hạ: Class giống như cái Khuôn. Object là đối tượng/bức tượng được đúc ra từ Khuôn. Class là sự chia nhóm, gom nhóm các đối tượng tương đồng nhau về đặc điểm và hành vi.**



Source: <https://breatheco.de/en/lesson/object-oriented-programming/>

### CONSTRUCTOR (HÀM KHỞI TẠO)

Class như cái Khuôn thì constructor như cái phễu dùng để đổ nguyên vật liệu vào trong Khuôn để đúc ra được một bức tượng, đồ vật, một object/instance.

Một class có thể có nhiều constructor, khác nhau ở phần tham số đầu vào, tương ứng với cùng một cái Khuôn, ta có nhiều cách chế/rót nguyên vật liệu đổ vào trong Khuôn.

Nếu một class không tạo sẵn constructor, Java tự động tạo một constructor rỗng/default lúc chạy app, có dạng



```
public class Person {
    private String id;
    private String name;
    public Person() {
    }
}
```

Tên constructor phải giống y chang tên class, và không có giá trị trả về

Ví dụ constructor hợp lệ

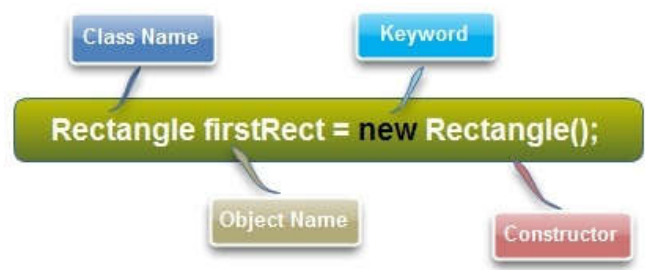
```
public class Person {
    private String id;
    private String name;
    public Person() {
    }
    public Person(String id, String
name) {
        this.id = id;
        this.name = name;
    }
}
```

## KHỞI TẠO ĐỐI TƯỢNG

Dùng toán tử new kèm việc gọi phễu/constructor để tạo ra một đối tượng, một vật thể, đúc một bức tượng, “to initialize an object, to create an instance of a class”, thông qua việc truyền tham số đầu vào cho hàm constructor, giống như việc dùng phễu để đổ vật liệu vào bên trong cái Khuôn.

Hãy nhớ đặt cho đối tượng vừa tạo một cái định danh, một tên gọi, còn gọi là biến đối tượng.

```
Person chiPu = new Person("CHI PU", "Nguyễn
Thùy Chi");
//chiPu là biến object, con trỏ, biến tham
chiếu
//new Person(...); chính là vùng object, hiện
thân của em Chi Pu, sẽ được trỏ bởi, quản lí
bởi biến object chiPu
//ngoài đời thực chiPu chính là tên gọi tắt,
tham chiếu, định danh về “bé/object” Chi Pu
```



Source: <http://ecomputernotes.com/java/what-is-java/constructor-in-java>

## STATIC

Là từ khoá áp dụng cho khai báo hàm, biến, nhóm lệnh, và class lồng (nested class). Hàm và biến static sẽ nằm riêng ở một vùng nhớ, là vùng nhớ dùng chung cho mọi object của class đó.

Truy xuất static thông qua tên class chấm (nếu được quyền truy xuất).

Ví dụ lệnh hợp lệ truy xuất static

```
int n1 = MyToys.getAnInteger();
int n2 = Integer.parseInt("6789");
```

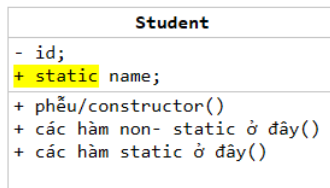
Static chơi với static. Static còn được gọi là class-level.

Biến, hàm non-static, tức là không chứa keyword static, thì chúng sẽ thuộc về object, còn gọi là object-level.

Không dùng biến static để lưu trữ các thông tin của riêng từng object.

Đồ static thường dùng cho các hàm, biến tiện ích, thư viện dùng chung – UTILITIES, TOOLS, TOYS.

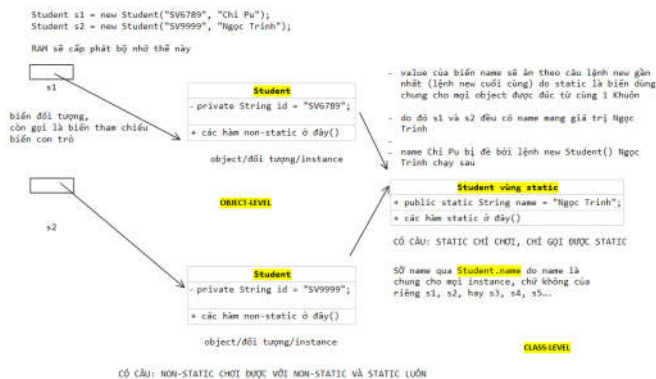
Hình dưới đây giải thích cơ chế biến/hàm static và non-static nằm trong RAM như thế nào.



```
public class Student {
    private String id;
    public static String name;

    public Student(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public void showInfo() {
        System.out.println("ID: " + id + "; Name: " + name);
    }
}
```



## INHERITANCE (KẾ THỪA)

Một Con chỉ có tối đa (extends) một Cha (single inheritance).

Nếu class không có class Cha, Cha mặc định là Object. Object là class ông-tổ của muôn loài.

Câu lệnh `super()` gọi constructor Cha nếu xuất hiện trong constructor Con, thì phải là câu lệnh đầu tiên trong constructor Con.

“Con bắt chước”, những gì của Cha (hàm, đặc tính) sẽ là của Con (kế thừa). Những gì của Con tạo dựng (riêng Con) sẽ là của Con, Cha không biết, không thấy.

Cha che giấu thông tin/field với người ngoài (private), thoải mái cho người nhà (Con), gọi là protected.

Một class chứa hàm abstract thì class bắt buộc phải là abstract. Đừng quan tâm điều ngược lại.

Vùng nhớ Con chứa:

- Vùng nhớ Cha qua câu lệnh `super()` ở trong constructor của Con
- Vùng extends, vùng Con mở rộng từ Cha chứa các hàm, đặc tính của riêng Con và các hàm `@override` từ Cha

Con của một class Cha abstract thì Con bắt buộc phải viết code/implement cho tất cả các hàm abstract đang có trong Cha abstract. Nếu không chịu viết hết code cho các abstract của Cha, Con cũng phải là abstract

```
10 Child is not abstract and does not override abstract method showInfo() in Parent
11 (Alt-Enter shows hints)
12 public class Child extends Parent {
13     Implement all abstract methods
14     Make class Child abstract
}
```

Không dùng toán tử `new` để tạo object với Abstract

Class. Ngoan cổ `new` sẽ tạo object dạng

**ANONYMOUS CLASS**, màn hình code sẽ bung rộng để cài đặt/viết code cho các hàm abstract của lớp Cha.

**LƯU Ý:** Đừng quên dấu ; ở vị trí thần thánh.

**Override:** Con có hàm trùng 100% tên và tham số và giá trị trả về với hàm của Cha.

**Overload:** xảy ra trong class bất kì, khi có nhiều hàm trùng tên nhau nhưng khác phần tham số (khác data type, không quan tâm tên tham số)

Thông qua kĩ thuật override, overload, ta thể hiện được nguyên lí đa xạ/đa hình **POLYMORPHISM**, cùng một tên hàm, ánh xạ nhiều cách thực thi/implement khác nhau – một tên hàm, nhiều cách xử lí mà không cần dùng đến câu lệnh IF cho mỗi cách xử lí.

Với mối quan hệ kế thừa Cha Con (và Thánh Thần ☺)  
ta có những cách khai báo, khởi tạo object/instance  
như sau:

```
Khai báo Cha new Cha();
```

```
//Cha không là abstract
```

```
Khai báo Con new Con();
```

```
Khai báo Cha new Con();
```

```
//dùng kế thừa, nhớ kĩ thuật DRIFT ép kiểu
```

```
Khai báo Con new Cha();
```

```
//GÃY, NGƯỢC ĐỜI
```

```
//Nếu Cha là abstract class thì
```

```
Khai báo Cha new Cha() {
```

```
    //code để implement  
    các hàm abstract của Cha abstract
```

```
class                //kĩ thuật anonymous
```

```
    };
```

```
    //chấm phẩy thần thánh
```

```
//Bung rộng/extends vùng nhớ Cha để có chỗ  
viết code/implement cho các hàm abstract của  
Cha ở vùng nhớ của object Con
```

### Drift, ép kiểu Cha Con

Khai báo Cha new Con(...), thì khi chấm chỉ xổ ra  
những gì của Cha, không xổ ra những gì của riêng  
Con

Để xổ ra những gì của chính Cha (thừa kế) và của  
riêng Con tạo dựng, ta dùng các kĩ thuật drift/ép kiểu  
như minh hoạ dưới đây

### Con chấm xổ ra hết của Cha và Con

```
package drift;

/**
 *
 * @author giao-lang
 */
public class Parent {

    public void sayHi() {
        System.out.println("Say Hi from Parent");
    }

}

public class Child extends Parent{

    public void sayHiFromChild() {
        System.out.println("Hi, I'm a child");
    }

}

public class Inheritance {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Child c1 = new Child(); //khai báo Con new Con()
        c1.                //xổ ra tất cả các hàm của Cha + Con
    }

    }

    equals(Object obj)    boolean
    getClass()            Class<?>
    hashCode()            int
    notify()              void
    notifyAll()           void
    sayHi()               void
    sayHiFromChild()      void
    toString()            String
```

### Con chấm chỉ xổ ra những gì của Cha

```
public class Inheritance {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Child c1 = new Child(); //khai báo Con new Con()
        c1.sayHiFromChild();    //xổ ra tất cả các hàm của Cha + Con

        Parent c2 = new Child(); //khai báo Cha new Con()
        c2.                //xổ ra chỉ những hàm của Cha
    }

    equals(Object obj)    boolean
    getClass()            Class<?>
    hashCode()            int
    notify()              void
    notifyAll()           void
    sayHi()               void
    toString()            String
    wait()                void
    wait(long timeout)    void
```

### Drift để trả lại những gì của Con qua hai kĩ thuật ép kiểu

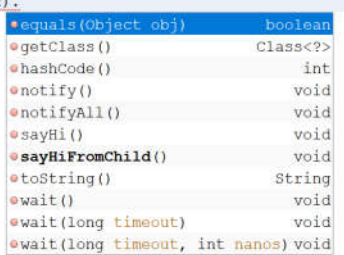
Ép “tạm” qua hai tầng dấu ngoặc:

```
public class Inheritance {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Child c1 = new Child(); //khai báo Con new Con()
        c1.sayHiFromChild(); //xỏ ra tất cả các hàm của Cha + Con

        Parent c2 = new Child(); //khai báo Cha new Con()
        //c2. //xỏ ra chỉ những hàm của Cha

        //drift, tở lái con trở, cho Con trở lại làm Con, xỏ tất cả
        ((Child)c2).

    }
}
```



Ép “vĩnh viễn” vào biến kiểu Con:

```
public class Inheritance {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Child c1 = new Child(); //khai báo Con new Con()
        c1.sayHiFromChild(); //xỏ ra tất cả các hàm của Cha + Con

        Parent c2 = new Child(); //khai báo Cha new Con()
        //c2. //xỏ ra chỉ những hàm của Cha

        //drift, tở lái con trở, cho Con trở lại làm Con, xỏ tất cả
        ((Child)c2).sayHiFromChild();

        Child tmp = (Child)c2; //cách khác, ép c2 trở vào Child
        tmp.sayHiFromChild(); //chụp lấy con trở này thả vào tmp
        //sẵn có kiểu là Child
    }
}
```

## INTERFACE (HỘI NHÓM ĂN CHƠI)

Interface là dạng class “Cha” đặc biệt, gom những đứa “Con” không quan tâm nguồn gốc xuất thân, chỉ quan tâm chung lí tưởng, hành động, giao tiếp, interface.

Là hình thức của Câu lạc bộ, Hội nhóm.

Chỉ chứa hàm abstract (trước JDK 8).

Không tạo constructor cho interface.

Biến, field trong interface mặc định là public static final, và phải được gán giá trị ngay khi khai báo biến.

Mọi hàm trong interface tầm truy xuất là public hoặc default (trước JDK8).

Không dùng toán tử new để tạo object với Interface.

Nếu ngoan cố dùng new sẽ tạo object dạng

### ANONYMOUS CLASS.

Một class “Con”, hay gọi là “Hội viên” có thể tham gia/implements nhiều interface, nhưng chắc chắn chỉ có tối đa 1 Cha kiểu kế thừa/inheritance – **đơn kế thừa, đa interface**.

Hội viên của một interface phải có trách nhiệm viết code/implement tất cả các hàm abstract của interface.

**Implements:** từ khoá dùng để báo hiệu một class

“Con” muốn ra nhập hội interface. Nếu ra nhập nhiều hội, tên các hội/interface cách nhau bằng dấu phẩy (,).

**Implement:** là hành động yêu cầu lập trình viên phải viết code cho các hàm abstract được khai báo bên abstract class hay interface.

```
public class Student implements DeadRacer, Shishaer, Phuoter {
    // ...
}
```

## JDK8 – INTERFACE CÓ CODE

> Hàm trong interface được dùng private và có code bên trong (JDK9)

> Hàm public trong interface được quyền có code nhưng phải mang keyword default hoặc static

```
default public void sayDefault() {
    System.out.println("Say Hi from a
    default method in an interface");
}

static void sayStatic() {
    System.out.println("Say Hi from a
    static method in an interface");
}
```

## JDK8 - FUNCTIONAL INTERFACE

> Là một interface chứa duy nhất trong nó một abstract method. Tuy thế một functional interface vẫn có thể có những hàm default khác (hàm có code).

> **@FunctionalInterface** Annotation được bổ sung vào đầu interface để đảm bảo rằng interface này không thể có nhiều hơn 1 abstract method. Ghi chú này là không bắt buộc.

> Để tạo object từ functional interface, ta có cách truyền thống là tạo inner class hoặc anonymous class. Tuy nhiên, “sành điệu” ta nên dùng biểu thức **Lambda**

```
package data;

/**
 *
 * @author giao-lang
 */
@FunctionalInterface //ép interface này không cho có quá một hàm abstract
public interface DeadRacer {

    public double runToDeath(); //mỗi đua thủ có một cách đua khác nhau
}
```

```
package data;

import java.util.Random;

/**
 *
 * @author giao-lang
 */
public class Race {

    public static void main(String[] args) {

        //dùng anonymous class
        DeadRacer miNhanNgu = new DeadRacer() {
            @Override
            public double runToDeath() {
                return new Random().nextDouble() * 100;
            }
        }; //VIP statement

        //dùng biểu thức Lambda
        DeadRacer chiHuHu = () -> new Random().nextDouble() * 50;
    }
}
```

[Xem thêm ở mục Biểu thức Lambda]

-oOo-

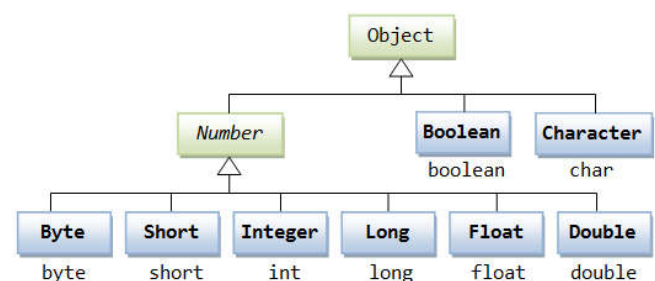
## Số và Chuỗi (Numbers & Strings)

### WRAPPER CLASS

Là những class mà object xuất phát từ nó “gói” các primitive data types. Nói cách khác, các object kiểu wrapper sẽ “bao, đóng hộp” các giá trị/dữ liệu có kiểu primitive int, long, double...bên trong nó. Ý nghĩa của việc này là giúp các kiểu dữ liệu trong thế giới Java hoàn toàn thuộc về kiểu đối-tượng.

Ứng với mỗi kiểu dữ liệu primitive nguyên thủy sẽ có một class bao wrapper tương ứng.

- Character (bao cho char)
- Byte (bao cho byte)
- Short (bao cho short)
- Integer (bao cho int)
- Long (bao cho long)
- Float (bao cho float)
- Double (bao cho double)
- Boolean (bao cho boolean)



### Khai báo:

> **Cách 1:** Theo đúng chuẩn OOP, dùng toán tử new, điều này không cần thiết (bỏ hẳn từ JDK9-Deprecated)

```
Integer n1 = new Integer(10);
```

> **Cách 2:** Gán trực tiếp giá trị, cách được ưa dùng

```
Integer n2 = 10;
```

Cách 1 và 2 đều sẽ tạo ra object n1, n2 trở vào vùng RAM “bộ” mà lõi/core của nó là 1 biến primitive int chứa giá trị 10. Con số 10 đã được “gói - BOXING” thành một object.

### Sử dụng:

Dùng biến kiểu wrapper bình thường như mọi kiểu object khác.

```

/**
 *
 * @author giao-lang
 */
public class Wrapper {
    public static void main(String[] args) {

        Integer n1 = new Integer(10); //bỏ hẳn từ JDK9

        Integer n2 = 10; //AUTO-BOXING
        System.out.println("n2 is: " + n2);

        n2 += 10; //UNBOXING và AUTO-BOXING
        System.out.println("n2 now is: " + n2);

        int n3 = n2; //UNBOXING - mở hộp n2 lấy lõi
        //là primitive 20 gán vào biến primitive n3
        System.out.println("n3 is: " + n3);
    }
}

```

Test (run) x

```

run:
n2 is: 10
n2 now is: 20
n3 is: 20

```

Trong hình ở trên, lưu ý lệnh `int n3 = n2;` nên được hiểu như sau: do n2 có kiểu là object, trở vùng new Integer() “bộ” có lõi là value primitive 20, mà n3 thì có kiểu là primitive, Java sẽ “mở hộp – UNBOXING” n2 để lấy value lõi là con số 20 gán vào biến primitive n3.

**Boxing:** đóng gói 1 primitive value thành object; hay gán 1 biến primitive vào biến wrapper.

**Unboxing:** mở hộp wrapper lấy ra value primitive bên trong; hay gán 1 biến object wrapper vào biến primitive.

**Immutable Class/Object:** Điều đặc biệt ở các wrapper class là chúng được thiết kế mà không có hàm setter(), nghĩa là ta không thể thay đổi value lõi

primitive bên trong vùng boxing. Muốn thay đổi giá trị lõi bên trong một vùng wrapper, ta phải cấp vùng nhớ boxing mới. Các wrapper class thuộc vào nhóm **IMMUTABLE CLASS**.

**Hàm tiện ích:** `parseInt()` (“chuỗi dạng số”), nhận vào một chuỗi dạng số và biến đổi chuỗi này thành con số kiểu primitive tương ứng. Ví dụ

```
Integer.parseInt("6789") //sẽ trả về giá trị 6789;
```

**Integer Pool:** [cập nhật sau...]

## STRINGS (CHUỖI KÍ TỰ)

Là một dãy liên tiếp các kí tự, được “gói” trong một vùng RAM để trở thành một object. String là kiểu dữ liệu đối tượng. String là IMMUTABLE CLASS, không chứa hàm setter(), một khi bạn tạo ra chuỗi trong vùng RAM rồi thì không thể thay đổi giá trị của các kí tự trong chuỗi. Muốn thay đổi bạn chỉ có cách tạo ra object String mới.

### Khai báo:

> **Cách 1:** Theo đúng chuẩn OOP, dùng toán tử new, điều này không cần thiết

```
String s1 = new String("Do It Now");
```

> **Cách 2:** Dùng chuỗi giá trị cho trước gán trực tiếp mà không cần new, gọi là dùng string literal. Và tình huống này sử dụng kĩ thuật String Pool.

```
String s1 = "Do It Now";
```

**String Pool:** khi khai báo và gán giá trị cho biến chuỗi dùng string literal (gán trực tiếp), không gán qua toán tử new, đó là lúc chuỗi được lưu vào 1 vùng nhớ chung gọi là Pool. Pool phân biệt chữ hoa chữ thường, do đó chuỗi “Hello” và “hello” sẽ nằm ở 2 vùng Pool khác nhau. Pool dùng cho mục đích tăng tốc việc cấp

phát lưu trữ chuỗi và tiết kiệm bộ nhớ do chuỗi là loại data type hay được sử dụng.

```
String s1 = new String("Do It Now"); //câu lệnh này không xài Pool
```

```
String s2 = "Do It Now"; //một Pool được tạo ra chứa chuỗi "Do It Now"
```

```
String s3 = "Do It Now"; //xài ké Pool trên, nghĩa là biến s3 và s2 đang trỏ cùng một vùng nhớ chứa object/chuỗi "Do It Now"
```

//HẾT SỨC LƯU Ý: s1 không xài Pool nên đang trỏ vùng object khác dù bên trong 2 vùng object đều mang chuỗi "Do It Now"

```
String s4 = "do it now"; //xài Pool mới, Pool này chứa chuỗi "do it now"
```

//các lệnh gán chuỗi sau này nếu xài literal, và nếu trùng chuỗi "do it now" sẽ cùng tấm chung, trỏ chung vùng RAM object "do it now"

### Các hàm chính:

**.toUpperCase():** trả ra chuỗi mới đã được đổi sang chữ hoa.

**.toLowerCase():** trả ra chuỗi mới đã được đổi sang chữ thường.

Do String là IMMUTABLE CLASS, không cho sửa nội dung chuỗi bên trong object chứa chuỗi, nên các hàm xử lý chuỗi sẽ tạo ra chuỗi mới, vùng nhớ mới chứa String đã được hàm xử lý.

```
String s1 = "Do It Now";
```

```
s1.toUpperCase(); //Lệnh này sẽ tạo ra một object chuỗi mới, một vùng mới RAM chứa chuỗi "DO IT NOW"
```

//Cần kết hợp lệnh này với xử lý khác. Lưu ý s1 vẫn giữ nguyên giá trị "Do It Now" như cũ

//Câu lệnh dưới đây KHÔNG LẠ, do hàm trả về object, và object thì chấm tiếp...

```
s1.toUpperCase().toLowerCase().toUpperCase();  
; //chấm MÃI MÃI
```

//Câu lệnh dưới đây KHÔNG LẠ, do literal là một object, và object thì chấm được

```
"Do It Now".toUpperCase();
```

```
System.out.println("Do It  
Now".toUpperCase()); //màn hình có DO IT NOW
```

//chuỗi "Do It" nằm trong Pool và bản chất là một object, thì CHẤM BUNG LỤA là chuyện dễ hiểu

### chuỗi-tớ.compareTo(chuỗi-cậu)

**chuỗi-tớ.compareToIgnoreCase(chuỗi-cậu):** so sánh hai chuỗi theo kiểu lớn-bé-bằng với 2 cơ chế: có phân biệt chữ cái hoa và chữ cái thường - compareTo(), không phân biệt hoa thường - compareToIgnoreCase() (nghĩa là chữ A và chữ a xem là giống nhau). Hàm trả về ba giá trị kiểu int -1, 0, 1 (hoặc âm, 0, dương) để biết chuỗi-tớ nhỏ thua, bằng, thắng chuỗi-cậu.

### chuỗi-tớ.equals(chuỗi-cậu)

**chuỗi-tớ.equalsIgnoreCase(chuỗi-cậu):** so sánh 2 chuỗi theo kiểu tớ có bằng cậu không? Kết quả trả về là giá trị đúng sai kiểu boolean (true/false). Có hai dạng so sánh là không phân biệt và có phân biệt chữ cái hoa hay thường.

```
String s1 = "Do It Now";
```

```
String s2 = "DO IT NOW";
```

```
System.out.println("s1 vs. s2: " +  
s1.compareToIgnoreCase(s2)); //0 - bằng nhau
```

```
System.out.println("s1 = s2? " +  
s1.equalsIgnoreCase(s2)); //true - bằng nhau
```

**CÁM TUYỆT ĐỐI** so sánh hai chuỗi dùng các toán tử số học dạng >, >=, <, <=, !=, == vì như vậy là đang so sánh 2 địa chỉ vùng nhớ, so sánh hai số nhà đang nằm trong hai biến chuỗi – vô nghĩa.



```
String s1 = "Do It Now";
String s2 = "DO IT NOW";

System.out.println("s1 = s2? " + (s1 == s2));

//phép so sánh trên hết sức “ngáo”, vô
nghĩa, vì bản chất s1 và s2 là hai biến đối
tượng, đang lưu địa chỉ, số nhà 2 vùng RAM,
2 căn nhà đầu đó trong RAM chứa 2 chuỗi "Do
It Now" và "DO IT NOW"

//KẾT QUẢ NÀY VÔ NGHĨA, MÌNH SO SÁNH SỐ NHÀ
CỦA NHAU ĐỂ LÀM CHI???
```

```
String s1 = "Do It Now";
String s2 = "Do It Now";

System.out.println("s1 = s2? " + (s1 == s2));

//s1 và s2 đang trỏ cùng 1 vùng object, một
vùng POOL, 2 biến nhưng chỉ có 1 chuỗi mà
thôi, phép so sánh == trong tình huống này
cũng vô nghĩa
```

## SO SÁNH STRING, STRINGBUILDER, STRINGBUFFER

String vs. StringBuffer Vs. StringBuilder

Property	String	StringBuffer	StringBuilder
Storage Area	When String is created <b>without using new operator</b> , JVM will create string in <b>string pool</b> area of heap.  When String is created using <b>new operator</b> , it will force JVM to create new string in <b>heap</b> (not in string pool).  <b>Example -&gt;</b> String s1 = "abc"; > In string pool area of heap. String s2 = new String("abc"); > In heap	StringBuffer is created in <b>heap</b> .	StringBuilder is created in <b>heap</b> .
Modifiable	No(Immutable)  This is the main drawback of strings. Immutable means that the string objects cannot be modified. Whenever an attempt to modify the string, a new object will be created.	Yes(Mutable)  StringBuffer objects can be modified without creating a new object.	Yes(Mutable)  StringBuilder objects can be modified without creating a new object.

Source: [https://ramj2ee.blogspot.com/2016/05/java-tutorial-java-string-vs\\_10.html](https://ramj2ee.blogspot.com/2016/05/java-tutorial-java-string-vs_10.html)

Factor / Class	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread Safety	Not thread safe	Thread safe	Not thread safe
Performance	Very high	Moderate	Very high

Source: <https://www.startertutorials.com/corejava/strings-in-java.html>

Khác một chút so với String là IMMUTABLE CLASS, StringBuilder và StringBuffer là Mutable class, có nghĩa là chuỗi lưu trong 2 object này có thể bị sửa đổi thông

qua việc gọi các hàm nội tại bên trong object đó. Xem ví dụ dưới đây

```
StringBuilder msg = new StringBuilder("Do It Now");

System.out.println("Before: " + msg);
//trước khi bị sửa

msg.append(" - Ngay và luôn");

System.out.println("After: " + msg);
//sau khi gọi hàm, msg đã bị thay đổi giá trị

//Màn hình sẽ có kết quả sau:
Before: Do It Now
After: Do It Now - Ngay và luôn
```

## STRINGTOKENIZER

Lớp java.util.StringTokenizer giúp bạn tách một chuỗi thành nhiều miếng nhỏ hơn, dựa trên dấu hiệu bạn muốn cắt. Dấu hiệu muốn cắt có thể là dấu cách, xỏ đứng |, dấu #, gọi là delimiter.

```
StringTokenizer st;

st = new StringTokenizer("SE12345|An
Nguyễn|2000|9.9", "|");

System.out.println("All of elements after
splitting");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}

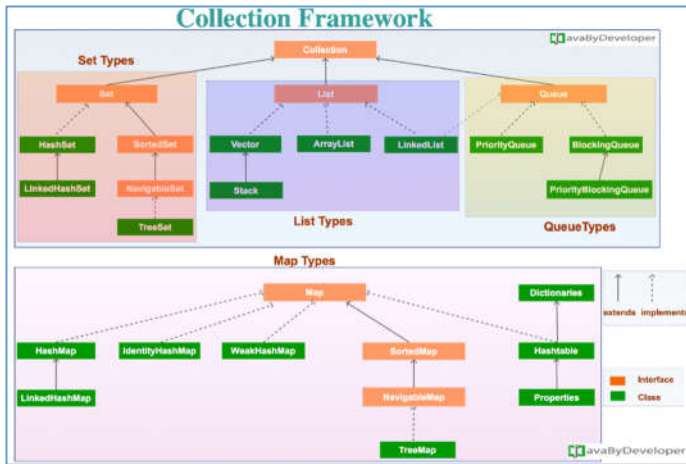
//Kết quả khi chạy chương trình:

All of elements after splitting
SE12345
An Nguyễn
2000
9.9
```

-oOo-



## Collection Framework (bộ sưu tập)



Source: <https://javabydeveloper.com/collection-framework-key-interfaces/>

**Collection** (tạm dịch – bộ sưu tập) là kĩ thuật gom một nhóm các object lại với nhau thành một đơn vị. Collection giống như cái giỏ, cái túi, cái ba-lô chứa bên trong nhiều món đồ. Ngoài đời có nhiều loại túi/giỏ khác nhau được thiết kế để đựng các món đồ khác nhau: giỏ chuyên đựng laptop, ba-lô đi phượt đựng đủ món... Tương ứng trong Java cũng có nhiều loại túi/giỏ khác nhau, giỏ đựng tùm lum món ArrayList, giỏ không được đựng trùng món HashSet...

**Java Collection Framework** cung cấp nhiều loại class và interface khác nhau (nhiều loại giỏ khác nhau) để đựng các object. Khác một chút so với ngoài đời, bên trong các loại giỏ của Java chỉ đựng con trỏ trỏ đến object/món đồ. Ví dụ túi đi chợ thì object đồ ăn, trái cây đựng trực tiếp trong giỏ. Trong lập trình khác một chút, ta dùng cái giỏ List, Set, Map khi đựng 1 object/món đồ, thì bản thân object có vùng nhớ riêng, do đó “nhét” món đồ vào giỏ được làm bằng cách tạo 1 sợi dây một đầu cột món đồ, đầu kia “móc”, “thả” vào trong giỏ. Giỏ trong Java chứa 1 đám biến con trỏ.

Hai interface Collection (`java.util.Collection`) và Map (`java.util.Map`) là hai giỏ “gốc” để từ đó “fake” ra một loạt các loại giỏ chuyên dụng khác nhau.

Kể từ phiên bản 8.0 trở đi, Java cung cấp thêm tính năng Java Stream API, cung cấp cách tiếp cận functional programming để giúp xử lý Collection một cách hiệu quả, trực quan hơn.

**[Stream API sẽ được viết thành mục riêng]**

## ARRAY LIST

**List/ArrayList:** cái giỏ chứa danh sách con trỏ bên trong nó, các con trỏ có thể trỏ trùng vào vùng new object. Thử tự vào giỏ cũng là thử tự lấy ra

```
ArrayList<Integer> bag = new ArrayList();
bag.add(1); //boxing, tương đương
bag.add(5); //new Integer(?)
bag.add(5);
bag.get(0) //sẽ lấy được con trỏ trỏ vùng
new Integer(1)

System.out.println(bag.get(0));
//gọi thăm tên em

//tương đương kết quả với
System.out.println(bag.get(0).toString());
```

## SET

**Set/HashSet:** cái giỏ chứa danh sách con trỏ bên trong nó, các con trỏ KHÔNG thể trỏ trùng vào cùng 1 vùng new object. Bỏ vào giỏ sẽ lộn xộn thứ tự. Thứ tự vào và lấy ra là không giống nhau

**Set/TreeSet:** cái giỏ chứa danh sách con trỏ bên trong nó, các con trỏ KHÔNG thể trỏ trùng vào cùng 1 vùng new object. Món đồ/object bỏ vào giỏ sẽ

được sắp xếp dựa theo đặc tính đã khai báo trong  
hàm compareTo() cài đặt cho interface Comparable

## File (Tập tin)

### MAP

**Map/HashMap:** cái giỏ chứa danh sách cặp con trỏ bên trong. Một món đồ/một object/một vùng new nhét vào giỏ sẽ được nhét theo hình thức sau:

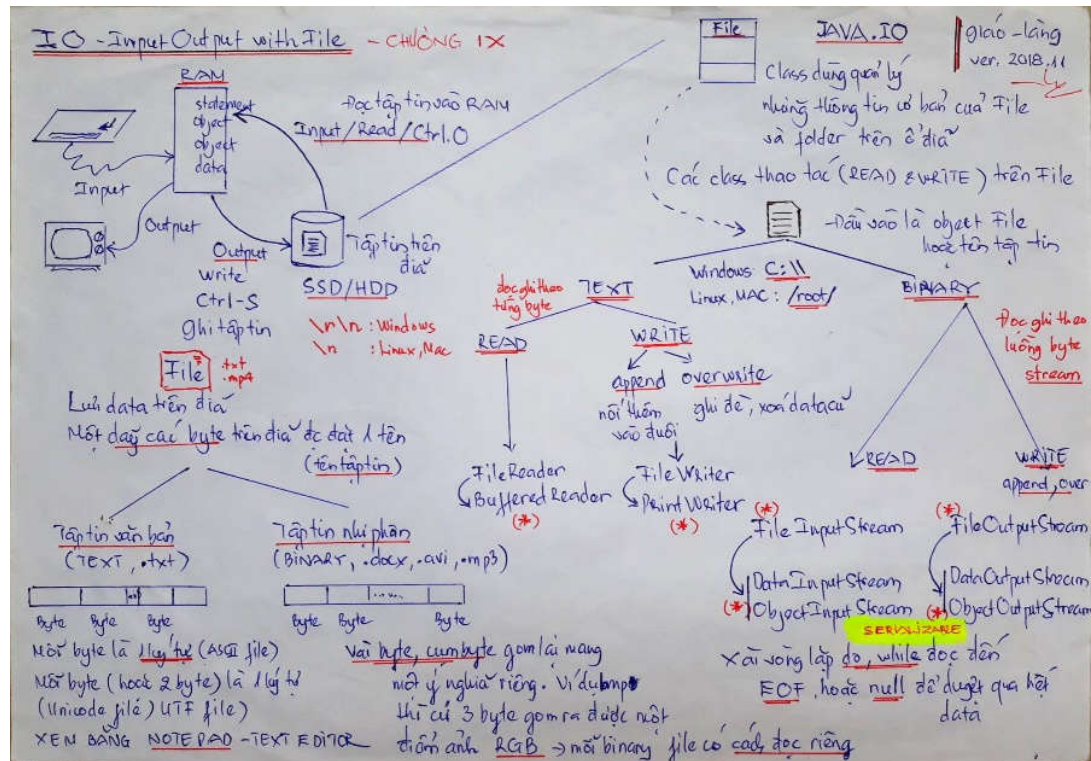
Một con trỏ trỏ vào object được bỏ vào giỏ, con trỏ này gọi là VALUE. Song song với con trỏ này, cần 1 con trỏ khác được gọi là dấu hiệu nhận dạng duy nhất con trỏ VALUE kia, cũng được add vào. Con trỏ song

song này gọi là KEY, vậy trong Map luôn chứa 1 cặp <KEY-VALUE>, key dùng nhận dạng, tìm ra value, thông qua value trỏ đến vùng object thật.

HashMap thì KEY cấm trùng, và cặp KEY, VALUE nằm lộn xộn không theo thứ tự add vào

**Map/TreeMap:** Nguyên tắc giống HashMap, chỉ có điều KEY sẽ được tự động sort khi add vào

[Phần này sẽ được viết chi tiết sau, đón xem...]



//CÒN TIẾP, ĐÓN XEM....

## Tài liệu tham khảo

Đang cập nhật...



Biên soạn bởi **Nguyễn Thế Hoàng (giáo-làng)**, giảng viên CNTT Đại học FPT, TP HCM

Kẻ gàn dở - yêu thơ văn - và giảng dạy IT.

Với gã, *"Code này đang chảy trong huyết quản. Cày code không cày rank/cày view"*. Với gã phải là *"Trên thông IT - Dưới tường showbiz..."*.

Cực đoan hơn, gã hay lẩm bầm, *"Lòng anh nghĩ đến em - Cả trong mơ còn code"* - Mong thi sĩ Xuân Quỳnh lượng thứ.

Và còn hơn thế nữa, *"Ta thường tới bữa quên ăn, nửa đêm gõ code, ruột đau như cắt, nước mắt đầm đìa, chỉ cảm tức chưa giải được thuật toán, vẽ được diagram, fix được bug để hoàn tất chương trình"* - Lạy cụ Trần Hưng Đạo tha thứ vì con đã mạo phạm văn thơ cụ.

Gặp gã ở đây: [hoang.nguyenthe@gmail.com](mailto:hoang.nguyenthe@gmail.com) | [fb/giao.lang.bis](https://fb/giao.lang.bis)