Nội dung

[0.Một số từ khoa final / static 2](#_Toc80539002)

[0.1 Final 2](#_Toc80539003)

[0.2 Static 4](#_Toc80539004)

[**1. Biến static trong Java** 4](#_Toc80539005)

[1 OOP 7](#_Toc80539006)

[Chú ý về khởi tạo lớp con vs lớp cha 7](#_Toc80539007)

[**Chú ý về tạo arraylist với lớp cha và lớp con** 7](#_Toc80539008)

[1.1 Kế thừa trong java 8](#_Toc80539009)

[1.2 Up-casting trong java ~ tham chiếu muộn 10](#_Toc80539010)

[1.3 Down-casting trong java 12](#_Toc80539011)

[**1.1.1 Các bước để downcasting** 13](#_Toc80539012)

[1.4 So sánh Absract class - Interface trong java 13](#_Toc80539013)

[2.Array in JAVA 16](#_Toc80539014)

[2.0 Lưu ý khi dùng với các Object trong java 16](#_Toc80539015)

[-Khi muốn tạo ra các phương thức mà liên quan đến duyệt phần tử / tìm phần tử / lấy phần tử thì phải tạo ra một cái ArrayList để chứa các phần tử này xong mới viết các phương thức để duyệt nó 16](#_Toc80539016)

[-Còn các phương thức mang tính chỉ liên quan đến đối tượng thì nên được viết ở class đối tượng 16](#_Toc80539017)

[2.1 String **Tokenizer trong JAVA** 16](#_Toc80539018)

[**Các phương thức của lớp StringTokenizer trong java** 17](#_Toc80539019)

# 0.Một số từ khoa final / static

## 0.1 Final

**Từ khóa final trong Java** được sử dụng để hạn chế người dùng. Từ khóa final có thể được sử dụng trong nhiều ngữ cảnh:

1. **Biến final:** bạn không thể thay đổi giá trị của biến final (nó sẽ là hằng số).
2. **Phương thức final:** bạn không thể ghi đè phương thức final.
3. **Lớp final:** bạn không thể kế thừa lớp final.
4. **Biến static final trống:** Một biến final mà không được khởi tạo tại thời điểm khai báo được gọi là biến final trống.
5. Ví dụ của biến final trong Java:
6. Giả sử có một biến final là MAX\_SPEED. Bạn cố ý thay đổi giá trị của biến này nhưng nó không bị thay đổi, bởi vì biến final một khi được gán giá trị thì không bao giờ thay đổi được.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class FinalExam1 {      final int MAX\_SPEED = 90; // biến final        void run() {          MAX\_SPEED = 400;      }        public static void main(String args[]) {          FinalExam1 obj = new FinalExam1();          obj.run();      }  } |

Kết quả:

Compile Time Error

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The final field FinalExam1.MAX\_SPEED cannot be assigned

**Câu hỏi:** Phương thức final có được kế thừa không?

Trả lời: Có, phương thức final được kế thừa nhưng bạn không thể ghi đè nó. Ví dụ:

[?](https://viettuts.vn/java/tu-khoa-final-trong-java)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class Bike {      final void run() {          System.out.println("running...");      }  }    class Honda2 extends Bike {      public static void main(String args[]) {          new Honda2().run();      }  } |

Kết quả:

running...

## 0.2 Static

**Từ khóa static trong Java** được sử dụng chính để quản lý bộ nhớ.

Trong java, Static có thể là:

1. **Biến static:** Khi bạn khai báo một biến là static, thì biến đó được gọi là biến tĩnh, hay biến static.
2. **Phương thức static:** Khi bạn khai báo một phương thức là static, thì phương thức đó gọi là phương thức static.
3. **Khối static:** Được sử dụng để khởi tạo thành viên dữ liệu static.

Khi bạn khai báo một biến là static, thì biến đó được gọi là biến tĩnh, hay biến

1. Biến static có thể được sử dụng để tham chiếu thuộc tính chung của tất cả đối tượng (mà không là duy nhất cho mỗi đối tượng), ví dụ như tên công ty của nhân viên, tên trường học của các sinh viên, ...
2. Biến static lấy bộ nhớ chỉ một lần trong Class Area tại thời gian tải lớp đó.

### **1. Biến static trong Java**

Khi bạn khai báo một biến là static, thì biến đó được gọi là biến tĩnh, hay biến static.

1. Biến static có thể được sử dụng để tham chiếu thuộc tính chung của tất cả đối tượng (mà không là duy nhất cho mỗi đối tượng), ví dụ như tên công ty của nhân viên, tên trường học của các sinh viên, ...
2. Biến static lấy bộ nhớ chỉ một lần trong Class Area tại thời gian tải lớp đó.

**Lợi thế của biến static**

Sử dụng biến static giúp chương trình của bạn sử dụng bộ nhớ hiệu quả hơn (tiết kiệm bộ nhớ).

Vấn đề khi không sử dụng biến static

|  |
| --- |
| public class Student8 {      int rollno;      String name;      static String college = "Bưu Chính Viễn Thông";        Student8(int r, String n) {          rollno = r;          name = n;      }        void display() {          System.out.println(rollno + " - " + name + " - " + college);      }        public static void main(String args[]) {          Student8 s1 = new Student8(111, "Thông");          Student8 s2 = new Student8(222, "Minh");            s1.display();          s2.display();      }  } |

Kết quả:

111 - Thông - Bưu Chính Viễn Thông

222 - Minh - Bưu Chính Viễn Thông

Giả sử có 1000 sinh viên trong trường đại học, bây giờ instance của các dữ liệu thành viên sẽ sự dụng bộ nhớ mỗi khi đối tượng được tạo. Tất cả sinh viên có rollno và name là thuộc tính riêng. Tuy nhiên, college là thuộc tính chung của tất cả đối tượng. Nếu chúng ta tạo nó là static, thì trường này sẽ chỉ sử dụng bộ nhớ một lần để lưu biến này.

**Chương trình đếm số không sử dụng biến static trong java**

Trong ví dụ dưới đây, chúng ta tạo một biến instance có tên count mà được tăng lên trong constructor. Khi biến instance này lấy bộ nhớ tại thời điểm tạo đối tượng, mỗi đối tượng sẽ có bản sao của biến instance đó, nếu nó được tăng lên, nó sẽ không ảnh hướng đến các đối tượng khác. Vì thế mỗi đối tượng sẽ có giá trị 1 trong biến count.

[?](https://viettuts.vn/java/tu-khoa-static-trong-java)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Counter1 {      int count = 0; // sẽ lấy bộ nhớ khi instance được tạo ra        Counter1() {          count++;          System.out.println(count);      }        public static void main(String args[]) {            Counter1 c1 = new Counter1();          Counter1 c2 = new Counter1();          Counter1 c3 = new Counter1();        }  } |

Kết quả:

1

1

1

**Chương trình đếm số có sử dụng biến static trong java**

Như bạn đã thấy ở trên, biến static sẽ lấy bộ nhớ chỉ một lần, nếu bất cứ đối tượng nào thay đổi giá trị của biến static, nó sẽ vẫn ghi nhớ giá trị của nó.

[?](https://viettuts.vn/java/tu-khoa-static-trong-java)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class Counter2 {      static int count = 0; // sẽ lấy bộ nhớ chỉ một lần        Counter2() {          count++;          System.out.println(count);      }      public static void main(String args[]) {            Counter2 c1 = new Counter2();          Counter2 c2 = new Counter2();          Counter2 c3 = new Counter2();        }  } |

Kết quả:

1

2

3

# 

# 1 OOP

# Chú ý về khởi tạo lớp con vs lớp cha

**- <lớp cha > obj = new <lớp con>**

**Đây là khởi tạo 1 tham chiếu kiểu <lớp cha > là obj tham chiếu tới 1 đối tượng <lớp con> thế này vẫn được hiểu là obj là đối tượng của lớp cha**

# **Chú ý về tạo arraylist với lớp cha và lớp con**

**khi tạo một arrayList thì dù tham chiếu tới nhiều lớp con khác nhau thì nó vẫn có kiểu dữ liệu là lớp cha**

**vd:** ArrayList<So> listnum = new ArrayList<>();

So ps6 = new PS(3,8); // PS va SoNguyen là lớp con của lớp So

So so = new SoNguyen(1);

Listnum.add(ps6);

Listnum.addNum(so);

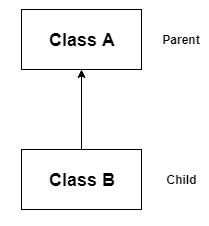
for (So x :listnum){ System.out.println(x.giaTri()+ " "); }

**phải có method giá trị để hiển thị giá trị không thì nó sẽ hiện địa chỉ**

## 1.1 Kế thừa trong java

- Java chỉ hỗ trợ đơn kế thừa:

**Đơn kế thừa**: nghĩa là một lớp chỉ được kế thừa từ đúng một lớp khác. Hay nói cách khác, lớp con chỉ có duy nhất một lớp cha.

****

- Java không hỗ trợ đa kế thừa 🡪 dùng interface để hỗ trợ đa kế thừa vì một class có thể  implements nhiều interface khác nhau một lúc.

Vd:

public interface ICoder {

String sleep(String time);

void code();

}

public interface IGamer {

String sleep(String time);

void playGame();

}

public interface IHacker {

String sleep(String time);

void hackFacebook();

}

* Lớp ItStudent sẽ implements cả ba interface đó:

public class ItStudent implements IHacker, IGamer, ICoder {

@Override

public void code() {

}

@Override

public void playGame() {

}

@Override

public void hackFacebook() {

}

@Override

public String sleep(String time) {

return time;

}

}

* Và khi muốn ItStudent là hacker, coder hay gamer các bạn chỉ cần làm như sau:

public class Main {

public static void main(String[] args) {

IGamer gamer = new ItStudent();

gamer.playGame();

gamer.sleep("no sleep :)");

ICoder coder = new ItStudent();

coder.code();

coder.sleep("I work on morning and sleep in the evening");

IHacker hacker = new ItStudent();

hacker.hackFacebook();

hacker.sleep("I work on evening and sleep in the morning");

}

}

## 1.2 Up-casting trong java ~ tham chiếu muộn

**- <lớp cha > obj = new <lớp con>**

**Đây là khởi tạo 1 tham chiếu kiểu <lớp cha > là obj tham chiếu tới 1 đối tượng <lớp con>**

-Khi lớp cha tham chiếu đến một đối tượng của lớp con ( Đối tượng của lớp con bị chuyển kiểu dữ liệu về lớp cha )

-VÌ nó truyền dữ liệu cho lớp cha nên khi khởi tạo một biến lớp con thì nó sẽ có các phương thức lớp con

**class Animal {**

**public void eat() {**

**System.out.println("eating...");//lớp cha phải có method void eat()**

**}**

**}**

**public class Cat extends Animal {**

**//@Override có hoặc ko dều được**

**public void eat() {**

**System.out.println("Eat meat");**

**}**

**public void meow() {**

**System.out.println("meowing...");**

**}**

**}**

**public class Upcasting {**

**public static void main(String[] args) {**

**Cat cat = new Cat();**

**//cả 3 cách viết giống nhau**

**Animal animal = new Cat()**

**Animal animal1 = cat; // Chuyển kiểu không tường minh**

**Animal animal2 = (Animal) cat; // Chuyển kiểu tường minh**

**cat.eat();**

**cat.meow();**

**animal1.eat();**

**animal2.eat();**

**// animal2.meow(); // Không thể gọi phương thức meow()**

**}**

**}**

**Eat meat**

**meowing...**

**Eat meat**

**Eat meat**

**Chú ý :** Khi upcasting rồi thì cái method không cần có từ khoa override để ghi đè nếu muốn gọi lớp cha thì dùng super

VD:

class CBird {

public void getInformation(){

System.out.print("has 2 legs, can fly");

}

}

public class CParrot extends CBird {

// không có override

public void getInformation(){

System.out.println("Parrot");

// using super keyword to call getInformation() of CBird

super.getInformation();

}

public static void main(String [] args){

CBird obj = new CParrot();// upcasting

obj.getInformation();

}

}

KQ: Parrot

has 2 legs, can fly

## 1.3 Down-casting trong java

- **Downcasting** là chuyển 1 đối tượng của lớp cha xuống thành đối tượng là thể hiện của lớp con trong quan hệ kế thừa

🡪Để lớp cha có thể gọi tới đối tượng của lớp con

**Thông thường, khi thực hiện dòng mã nguồn:**

|  |  |
| --- | --- |
| **1** | **Animal animal= new Cat();** |

**Ta hoàn toàn có thể gọi những phương thức đã được override của lớp Animal tại lớp Cat qua đối tượng animal. Tuy nhiên, vấn đề phát sinh khi ta muốn gọi mọi phương thức của lớp Cat thông qua việc ép kiểu đối tượng thuộc lớp Animal. Khi đó, ta sử dụng downcasting .**

**class Animal {**

**public void eat() {**

**System.out.println("eating...");//lớp cha phải có method void eat()**

**}**

**}**

**public class Cat extends Animal {**

**public void eat() {**

**System.out.println("Eat meat");**

**}**

**public void meow() { // Animal không co lop nay**

**System.out.println("meowing...");**

**}**

**}**

**public class Downcating {**

**public static void main(String[] args) {**

**Animal animal = new Cat();**

**animal.meow() 🡪 lỗi đây là vấn đề minh muốn làm**

**// các cách down-casting 2 cách:**

**Cat cat = (Cat) animal; //Cach 1 ép kiểu cho animal**

**cat.meow();**

**((Cat) animal).meow();// Cach 2 ~ animal.meow()**

**}**

**}**

### **1.1.1 Các bước để downcasting**

**BC1:ép kiểu cho animal thanh kiểu của kiểu con**

**BC2: gọi phương thức đấy ((Cat) animal).meow() ~ animal.meow()**

**Ta thấy, meow() là phương thức chỉ có ở lớp Cat. Tuy nhiên, thông qua downcasting ta hoàn toàn có thể gọi ra phương thức đó thông qua đối tượng cat mà không cần new Cat() bằng việc downcasting đối tượng animal** có kiểu Animal mà không xảy ra vấn đề trong quá trình biên dịch (compile) và thực thi (runtime).

## 1.4 So sánh Absract class - Interface trong java

\*Abstract:

- Bản chất của abstract class là một class nên nó có những đặc tính của class (có thể có biến , hàm….) **điểm duy nhất khác biệt đó là nó có các phương thức abstract không có thân mà thân được viết ở lớp kế thừa nó**

-Một abstract class nên có *ít nhất một* abstract method.

-Khi một class kế thừa một abstract class thì class đó phải implement tất cả các abstract methods của abstract class.

-không thể tạo đối tượng từ lớp trừu tượng.

Ví dụ: không thể tạo đối tượng từ lớp SV

SV s = new SV(...); // lỗi

SV s = new SVSP() // class SVSP kế thừa abstract class SV

-Lớp kế thừa từ lớp trừu tượng phải khai báo tườngminh các PTTT nếu không cũng là lớp trừu tượng

-Vẫn kế thừa được lớp abstract như bình thường vẫn có các variable như binh thường chỉ không gọi được lớp cha ở hàm main mà phải thông qua tham chiếu lớp con

VD:

public abstract class SV {  
 private String hoTen;  
 private Date ngaySinh;  
 private double dtb;  
  
 public SV(String s, Date ns, double d){  
 hoTen = s;  
 ngaySinh = ns;  
 dtb = d;  
 }

public class SVSP extends SV{  
 protected String noiTT;  
 protected double diemTT;  
  
 public SVSP(String s, Date ns, double d, String noiTT, double diemTT) {  
 super(s, ns, d);  
 this.noiTT = noiTT;  
 this.diemTT = diemTT;  
 }

\*Interface

- Hỗ trợ đa kế thừa trong java

\* Về kế thừa:

* Chúng ta biết rằng trong Java một class chỉ có thể kế thừa từ một và chỉ một class khác. Abstract class cũng không ngoại lệ, nó chỉ có thể kế thừa từ một class bình thường hoặc một abstract class khác.
* Một class có thể implement (triển khai) một hoặc nhiều interface. Abstract class cũng vậy, nó có thể implement một hoặc nhiều interface khác nhau.
* Nhưng một interface có thể extends từ nhiều interface khác.

 \*Constructor & Deconstructor

* **Interface**: Không có constructor và cả deconstructor.
* **Abstract class**: Có thể tạo constructor và deconstructor nhưng thường sẽ không sử dụng được ở lớp Entry mà thay vào = new <lớp con> (tham chiếu tới đối tượng lớp con)
* Abstract Person.

public abstract class Person {

protected String name;

protected int age;

protected Person(String name, int age){

this.name = name;

this.age = age;

}

abstract void say();

abstract void run();

abstract void walk();

}

* Nếu cố gắng tạo một instance và truyền các tham số vào constructor thì sẽ như này.

Person p = new Person("A", 20) // không được

Person p = new Student("A", 20)

* Chúng ta thường sử dụng keyword **super** ở lớp con để gọi đến constructor/ deconstructor được kế thừa abstract class. Ví dụ:

public class Student extends Person{

int id;

String address;

public Student(String name, int age, int id, String address) {

super(name, age);

this.id = id;

this.address = address;

}

@Override

public void say() {}

@Override

public void run() {}

@Override

public void walk() {}

}

# 2.Array in JAVA

## 2.0 Lưu ý khi dùng với các Object trong java

## -Khi muốn tạo ra các phương thức mà liên quan đến duyệt phần tử / tìm phần tử / lấy phần tử thì phải tạo ra một cái ArrayList để chứa các phần tử này xong mới viết các phương thức để duyệt nó

## -Còn các phương thức mang tính chỉ liên quan đến đối tượng thì nên được viết ở class đối tượng

## 2.1 String **Tokenizer trong JAVA**

**-Cho phép bạn phân tách chuỗi thành các phân tử token của dãy string đó**

**-Constructor:**

|  |  |
| --- | --- |
| **Constructor** | **Mô tả** |
| **StringTokenizer(String str)** | tạo ra một lớp StringTokenizer với chuỗi chỉ định. |
| **StringTokenizer(String str, String delim)** | tạo ra một lớp StringTokenizer dựa trên chuỗi chỉ định và dấu phân cách. |
| **StringTokenizer(String str, String delim, boolean returnValue)** | tạo ra một lớp StringTokenizer dựa trên chuỗi định, dấu phân cách và cờ hiệu. Nếu cờ hiệu là true, dấu phân cách được xem như là các phần tử token. Nếu là false, dấu phân cách không được tính là các phần tử token. |

-Các phương thức:

## **Các phương thức của lớp StringTokenizer trong java**

Các phương thức non-private của lớp StringTokenizer được tóm tắt trong bảng sau:

|  |  |
| --- | --- |
| **Phương thức Public** | **Mô tả** |
| **boolean hasMoreTokens()** | Trả về true nếu còn nhiều token trong chuỗi. |
| **String nextToken()** | Trả về token tiếp theo khi duyệt đối tượng StringTokenizer. |
| **String nextToken(String delim)** | Trả về token tiếp theo dựa trên dấu phân tách. |
| **boolean hasMoreElements()** | Giống như phương thức hasMoreTokens(). |
| **Object nextElement()** | Giống như nextToken() nhưng nó trả về một đối tượng. |
| **int countTokens()** | Trả về tổng số lượng của các token. |

## 2.2 Dùng HashMap thay cho HashTable dễ dùng hơn

-Nó cũng có các khoa và value HashMap< Key, Value>

K - the type of keys maintained by this map

V - the type of mapped values

-Các phương thức trong HashMap:

+ put ( Key, Value) : thêm Key hoặc Value vào map

+ get (Key ): cho Key vào để trả về Value

Vd: HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

System.out.println(capitalCities.get("England"));

KQ : London

+elements() : trả về một list enumaration các value

+keys() trả về một list enumaration các keys

Thường dùng với lớp Enumaration

Enumeration keys = table.keys();  
Enumeration values = table.elements();  
  
System.*out*.println("Cac tu co so lan xuat hien:");  
while (keys.hasMoreElements()) {  
 System.*out*.println(values.nextElement()+ " : " +keys.nextElement() );  
}

## 2.3 **Loop Through a HashMap Khi dùng vòng lặp**

### Cách dùng keySet() và values() nhưng chỉ lấy một key hoặc value thôi

+ keySet():method if you only want the keys

+ values() method if you only want the values

Loop through the items of a HashMap with a **for-each** loop.

**Note:** Use the keySet() method if you only want the keys, and use the values() method if you only want the values:

VD: HashMap<String, String> capitalCities = new HashMap<String, String>();

capitalCities.put("England", "London");

capitalCities.put("Germany", "Berlin");

capitalCities.put("Norway", "Oslo");

capitalCities.put("USA", "Washington DC");

for (String i : capitalCities.keySet()) {

System.out.println(i);

}

}

KQ: USA

Norway

England

Germany

### Cách sử dụng lớp Enumerations để in ra đồng thời cả hai giá trị

Table / Map / Vector có phương thức :

+elements() : trả về một list enumaration các value

+keys() trả về một list enumaration các keys

Thường dùng với lớp Enumaration

Enumeration keys = table.keys();  
Enumeration values = table.elements();  
  
System.*out*.println("Cac tu co so lan xuat hien:");  
while (keys.hasMoreElements()) {  
 System.*out*.println(values.nextElement()+ " : " +keys.nextElement() );  
}

# 3.Interface Enumerations

-Là một interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. 🡪 giúp chúng ta lấy các phần tử từ một collection of object

🡪CHỈ DÙNG VỚI NHỮNG CÁI COLLECTION MÀ CÓ NHỮNG PHƯƠNG THỨC ĐỂ LẤY CẢ MỘT LIST NHƯ elements() , keys() 🡪 Vector / HASH Table / Hash MAp….

🡪sử dụng Enumeration như một cái để chứa giá trị mình cần loop through cái list này

Vd:

Vector ds = new Vector();

Enumeration e = ds.elements();

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **boolean hasMoreElements( )**  When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated. |
| 2 | **Object nextElement( )**  This returns the next object in the enumeration as a generic Object reference. |

🡪 thường được sử dụng với Vector và HashTable vì nó không có cái hasmore…

Vd:

# 4. **Java Enums**

-Enum là viết tắt của enumerations nghĩa là danh sách đặc biệt

-Là một lớp đặc biệt tượng trưng cho group of constant (unchangeable variables, like final variables).

-Để tạo ra enum sử dụng từ khóa enum và tách cách constant bằng dấu phẩy và các constant phải được viết hoa

### **Example**

enum Level {

LOW,

MEDIUM,

HIGH

}

public class Main {

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

System.out.println(myVar);

}

}

Kq:MEDIUM

### Enums are often used in switch statements to check for corresponding values: **Example**

enum Level {

LOW,

MEDIUM,

HIGH

}

public class Main {

public static void main(String[] args) {

Level myVar = Level.MEDIUM;

switch(myVar) {

case LOW:

System.out.println("Low level");

break;

case MEDIUM:

System.out.println("Medium level");

break;

case HIGH:

System.out.println("High level");

break;

}

}

}

The output will be:

Medium level

## **Loop Through an Enum**

-Enum có method values() nó sẽ trả về một cái array chứa các constant trong list enum

### **Example**

for (Level myVar : Level.values()) {

System.out.println(myVar);

}