Lưu ý về mảng:

```
X[] arr = new Y[2]; // Y là con của X
arr[0] = new Y(); // OK
arr[1] = new X(); // Lỗi: không thể gán X vào mảng Y


int [] a = {1, 2, 3};
a[3] = 4; // Lỗi: ArrayIndexOutOfBoundsException
Khi khai báo mảng phải cho biết số lượng phần tử ở vế phải


Widening từ byte → short → int → long → float → double luôn hợp lệ, không
mất dữ liệu đáng kể (theo định nghĩa Java).
```

Lưu ý về **try-catch**:

```
try {…
}
catch(IOException e) {
    System.out.println("Lỗi IO: " + e.getMessage());
}
catch (Exception e) {
    System.out.println("Lỗi khác: " + e.getMessage());
}     //đảo ngược 2 cái catch này sẽ báo lỗi
```

StringBuilder // thường dùng để đảo ngược chuỗi

ArrayList

```java
public static void main(String[] args) {
        ArrayList<String> danhSach = new ArrayList<>();

        // Thêm phần tử
        danhSach.add("An");
        danhSach.add("Bình");
        danhSach.add("Chi");

        // Thêm vào vị trí cụ thể
        danhSach.add(1, "Dũng");
```

```
        // Sửa phần tử
        danhSach.set(0, "Anh");

        // Xoá phần tử
        danhSach.remove("Chi");
        danhSach.remove(1);

        // Lấy phần tử
        String firstElement = danhSach.get(0);

        // Kiểm tra
        System.out.println("Có An không? " + danhSach.contains("An"));
        System.out.println("Danh sách rỗng? " + danhSach.isEmpty());
        System.out.println("Số phần tử: " + danhSach.size());

        // Duyệt
        System.out.println("\n--- Danh sách ---");
        for (String ten : danhSach) {
            System.out.println(ten);
        }

        // Sắp xếp
        Collections.sort(danhSach);
        System.out.println("\nSau khi sắp xếp: " + danhSach);

        // Chuyển sang mảng
        String[] arr = danhSach.toArray(new String[0]);
        System.out.println("Phần tử đầu tiên trong mảng: " + arr[0]);

        // Xoá toàn bộ
        danhSach.clear();
        System.out.println("Danh sách rỗng? " + danhSach.isEmpty());
    }
```

HashMap

```
public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();

        // Thêm phần tử
        map.put("An", 9);
        map.put("Bình", 10);
        map.put("Chi", 8);

        // Lấy giá trị
        System.out.println("Điểm của Bình: " + map.get("Bình"));
```

```
        // Kiểm tra tồn tại
        System.out.println("Có Chi không? " + map.containsKey("Chi"));

        // Thay thế giá trị
        map.replace("Chi", 9);

        // Duyệt map
        System.out.println("\n--- Duyệt HashMap ---");
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            System.out.println(entry.getKey() + ": " +
entry.getValue());
        }
         //chỉ lấy value

        for(Integer a : map.values()) {
            System.out.println("Giá trị: " + a);
        }


        // Xóa phần tử
        map.remove("An");

        // Kiểm tra kích thước và rỗng
        System.out.println("\nSố phần tử còn lại: " + map.size());
        System.out.println("Map rỗng? " + map.isEmpty());

        // Xóa toàn bộ
        map.clear();
        System.out.println("Sau clear, map rỗng? " + map.isEmpty());
    }
```

//Các chương trình muốn truy cập sử dụng trực tiếp được phải sử dụng static

Chương 2: Lập trình hướng đối tượng

1. Giới thiệu về OOP

- Khái niệm lập trình hướng đối tượng
- Định nghĩa từ Grady Booch

2. Lịch sử phát triển của ngôn ngữ lập trình

Ngôn ngữ máy và hợp ngữ

- Ngôn ngữ máy

- Là tập hợp các **lệnh** hoặc **chỉ thị** mà bộ xử lý (CPU) của máy tính hiểu và thực thi trực tiếp.
- Được biểu diễn bằng **dãy số nhị phân** gồm 0 và 1.
- **Không gần gũi với ngôn ngữ con người**, nên:
  - Khó đọc
  - Khó ghi nhớ
  - Khó viết chương trình phức tạp

- Hợp ngữ (Assembly Language)

- Là phiên bản **trừu tượng hóa** của ngôn ngữ máy trên từng nền tảng cụ thể.
- Cung cấp **ký hiệu gần gũi hơn với con người**, ví dụ như `MOV`, `ADD`, `SUB`... thay vì những dãy bit dài.
- Vẫn cần **hiểu cấu trúc phần cứng** để viết chương trình.
- Được sử dụng phổ biến trong giai đoạn đầu khi máy tính chủ yếu phục vụ **tính toán đơn giản**.

Ngôn ngữ lập trình cấp cao

- Khái niệm

- Còn được gọi là **ngôn ngữ ra lệnh**, vì lập trình viên **viết mã bằng lệnh** gần gũi với ngôn ngữ con người hơn là ngôn ngữ máy.
- Là sự **trừu tượng hóa từ hợp ngữ**, giúp giảm độ phức tạp và khó hiểu khi lập trình.

- Đặc điểm

- Người lập trình vẫn phải suy nghĩ **dưới dạng cấu trúc máy tính** vì chưa có đủ công cụ để mô hình hoá thế giới thật dễ dàng.
- Cần phải **thiết lập mối quan hệ giữa hai mô hình**:
  - 🖥️ **Mô hình máy**: môi trường giải quyết vấn đề, là hệ thống máy tính.
  - 🌍 **Mô hình vấn đề**: thể hiện thực tế, như hành vi hoặc dữ liệu từ thế giới thật.

- Ứng dụng

- Trong giai đoạn này, máy tính **bắt đầu được dùng để giải quyết các vấn đề thực tế** chứ không chỉ đơn thuần là tính toán.
- Là nền tảng cho các ngôn ngữ sau này như C, Pascal, và dẫn tới **lập trình hướng đối tượng**.

Ngôn ngữ lập trình hướng đối tượng

- Khái quát: **Lập trình Hướng đối tượng:**

- Cung cấp các công cụ (khái niệm) cho phép người lập trình **mô hình hóa thế giới thật trong không gian giải quyết vấn đề** một cách dễ dàng.
- Mô hình mà Lập trình hướng đối tượng chọn lựa là biểu diễn vấn đề trong không gian giải pháp như các **"sự vật"** hay **"đối tượng"** (object).
- Đây là một sự trừu tượng hóa mạnh mẽ và linh hoạt vì **bản chất của thế giới là sự tương tác giữa các "sự vật".**
  ⇒ Nó cho phép **mô tả vấn đề dưới dạng vấn đề,** thay vì dưới dạng máy tính (nơi giải pháp sẽ *chạy*). Giúp mô hình hóa giống ngoài đời thật thay vì tập trung vào cách máy tính xử lý

- Ý tưởng: là các **sự vật:**

+ Chương trình là một tập các sự vật **tương tác lẫn nhau**

+ Sự vật trong OOP là sự tái hiện của các sự vật trong thế giới thật: Mỗi sự vật có những **đặc tính** (properties/characteristics) và **khả năng** (capacities) riêng. Cung cấp nhiều quy định ràng buộc về biến và đối tượng

Lịch sử hình thành OOP

- OOP là phương pháp lập trình chính hiện **nay:**

+ Simula 1967, **Smalltalk 1972**

➜ Simula được xem là ngôn ngữ đầu tiên hỗ trợ lập trình hướng đối tượng. Smalltalk là ngôn ngữ giúp phát triển mạnh các khái niệm OOP như: kế thừa, đa hình.

+ Giữa thập niên 90's, OOP mới bắt đầu được chú ý và sử dụng rộng rãi.

➜ Do sự bùng nổ của các phần mềm lớn, phức tạp, cần mô hình hóa theo thế giới thực để dễ bảo trì, mở rộng.

+ Hầu hết các ngôn ngữ lập trình hiện đại đều hướng đối tượng:

➜ Ví dụ: **C++**, **Java**, **.NET**, **Python**, **C#**, **Ruby**, **Swift**,...

➜ Những ngôn ngữ này đều hỗ trợ các tính năng như: đóng gói, kế thừa, đa hình, trừu tượng hóa.

3. So sánh các phương pháp lập trình

Lập trình cổ điển vs. Lập trình hướng đối tượng

Tiêu chí	Lập trình cổ điển (Hướng thủ tục)	Lập trình hướng đối tượng (OOP)
Khái niệm cơ bản	Tập trung vào các bước (thủ tục, hàm) để giải quyết vấn đề.	Tập trung vào các đối tượng (sự vật) và mối quan hệ giữa chúng.
Cách tổ chức chương trình	Dựa vào các hàm, chia thành các bước xử lý tuần tự.	Dựa vào các lớp (class) và đối tượng (object).
Mô hình hóa vấn đề	Mô hình hóa theo cách máy tính hiểu (dữ liệu và thuật toán riêng biệt).	Mô hình hóa theo thế giới thực (dữ liệu và hành vi gắn kết).

Tiêu chí	Lập trình cổ điển (Hướng thủ tục)	Lập trình hướng đối tượng (OOP)
Dữ liệu và chức năng	Dữ liệu và hàm xử lý tách rời nhau.	Dữ liệu và hàm xử lý được đóng gói trong đối tượng.
Tính mở rộng	Khó mở rộng, dễ phát sinh lỗi khi thay đổi chương trình.	Dễ mở rộng nhờ cơ chế kế thừa và đa hình.
Tính tái sử dụng	Khó tái sử dụng do thiết kế cứng nhắc.	Dễ tái sử dụng nhờ các đối tượng và kế thừa.
Tính bảo mật dữ liệu	Thấp, vì dữ liệu thường có thể bị truy cập trực tiếp.	Cao, nhờ cơ chế đóng gói, hạn chế quyền truy cập.
Ví dụ ngôn ngữ	C, Pascal, Fortran, Assembly,...	C++, Java, Python, C#, Ruby, Swift,...

Trừu tượng hóa trong OOP

- Là khả năng **che giấu** những chi tiết **phức tạp** bên trong đối tượng và chỉ **hiển thị** những thông tin **cần thiết** cho người sử dụng

- Xác định tính hành vi của đối tượn có liên quan.

- Thể hiện bằng **interface** và **abstract class**

Tại sao lại OOP

- Liên kết chặt chẽ giữa dữ liệu và thao tác của một đối tượng, cho phép tập trung vào **vấn đề** hơn là các **chi tiết bên trong vấn đề**

- Các dữ liệu và thao tác được "**bao gói**" trong một đối tượng, có thể **che giấu** những dữ liệu cần thiết, hoặc chỉ cho phép **truy xuất qua các hàm**

4. Đặc trưng của OOP

Mô Hình Hóa Vấn Đề

- Vấn đề được mô hình hóa như là tập hợp các đối tượng hoạt động cộng tác với nhau:

  + Chương trình: tương tác của nhóm các đối tượng

  + Tương tác giữa các đối tượng: là việc gửi các thông điệp/yêu cầu cho nhau

- Trong OOP, các đối tượng có thể được "**nhân hóa**". VD: một "cái cửa" có thể tự mở cửa, một "menu" có thể tự "hiển thị"

- Tuy nhiên, các đối tượng phải được "**yêu cầu**" khi nào cần thực hiện thao tác gì bởi một đối tượng khác → Các đối tượng trong chương trình phải "**hợp tác**" với nhau để giải quyết một vấn đề

Những đặc trưng của OOP:

Đặc trưng	Ý nghĩa bổ sung

Mọi thứ đều là đối tượng	Mô hình hóa thế giới thực
Đối tượng tương tác	Thông qua gửi thông điệp (gọi phương thức)
Bộ nhớ riêng	Đóng gói dữ liệu và bảo vệ dữ liệu, có thể chứa đối tượng khác
Đều có kiểu (lớp)	Các đối tượng cùng lớp sẽ cùng cấu trúc và hành vi
Nhận cùng thông báo	Phản hồi theo cách riêng của mình. Hỗ trợ đa hình

5. Các khái niệm quan trọng

Đối tượng (Object)

- Khái niệm và ví dụ minh họa: Khái niệm quan trọng

+ Trong OPP, mọi thứ đều là **đối tượng**

+ Là một thực thể được sử dụng bởi máy tính, là **"cái mà ứng dụng muốn đề cập đến"**

+ Mô tả cho **một sự vật hoặc khái niệm trong thực tế.** Có thể là:

- Một đối tượng thật: **có thể sờ, nhìn thấy hay cảm nhận** được
- Một đối tượng khái niệm: được **trừu tượng hóa**
- Một đối tượng phần mềm

+ Đối tượng cũng là **1 biến** trong chương trình

- Cấu trúc:

+ Thuộc tính (property, attribute): mô tả các đặc điểm, trạng thái của đối tượng.

+ Phương thức (behavior, method): mô tả các thao tác, các hoạt động mà đối tượng có thể thực hiện **"khả năng", "chức năng"** của đối tượng đó

+ Định danh (object identifier): dùng để phân biệt giữa các đối tượng

Lớp (Class)

- Định nghĩa:

+ Còn được gọi là **loại/kiểu (cấu trúc dữ liệu)** của đối tượng

+ Là một **đặc tả** (specification) của một kiểu dữ liệu mới

+ Lớp là một khuôn mẫu để tạo ra các đối tượng cùng kiểu.

+ Định nghĩa các thuộc tính và phương thức (hành vi) chung cho tất cả các đối tượng cùng lớp

+ Mỗi đối tượng được gọi là một thể hiện của một lớp và giá trị thuộc tính của chúng có thể khác nhau

- Thể hiện (instance):

+ Các thuộc tính được xác định bằng giá trị cụ thể

+ Một đối tượng cụ thể được gọi là một **thể hiện**

+ VD: Chó là một lớp, Mino và Lulu là những thể hiện của lớp Chó

- Thuộc tính & phương thức (hành vi):

+ **Thuộc tính:** mô tả trạng thái của đối tượng

+ **Phương thức:** Thể hiện cho khả năng của một đối tượng có thể thực hiện được những hành vi gì. Đối với việc **truyền thông điệp**:

- Trong một chương trình OOP, các đối tượng hoạt động cộng tác với nhau qua việc truyền thông điệp cho nhau
- Thông điệp (message): là một yêu cầu thực hiện một thao tác, hoạt động. Gồm có:
  - Tên thông điệp (tên phương thức)
  - Các tham số của thông điệp (tham số của phương thức)
- Muốn truyền thông điệp thì bao gồm:
  - Đối tượng nhận thông điệp
  - Thông điệp

6. Đối tượng và lớp

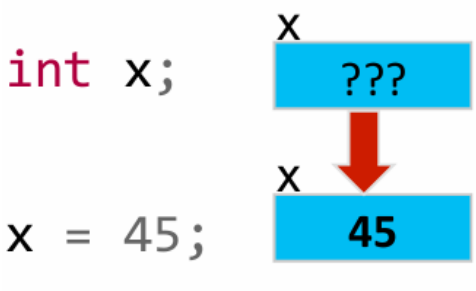
Đối tượng: Các loại biến

- Biến kiểu dữ liệu **cơ bản** (biến **nguyên thủy**):

+ Chỉ chứa dữ liệu, không có hàm để gọi

+ Có thể gọi từ một đối tượng khác

+ Lưu trực tiếp dữ liệu vào vùng nó được cấp phát (stack)



+ float pi = 3.14**f (phải có f)**

+ **char = ‘a’;**

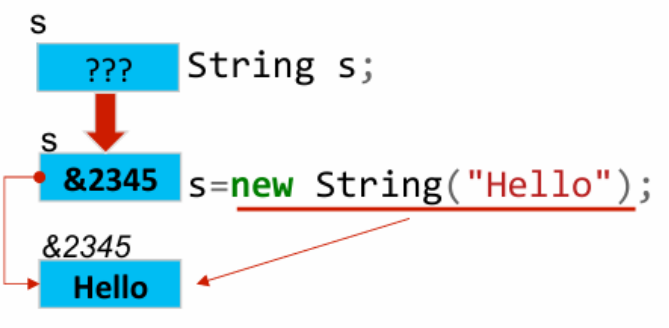
+ **String b = “phu”;**

+ **Không thể là null (có giá trị mặc định):**

- Giá trị mặc định của biến char là: \u0000
- Giá trị mặc định của biến boolean là: false

+ **Biến cục bộ (trong một hàm)** thì java không có tạo giá trị mặc định

- Biến **đối tượng** (biến **tham chiếu**):

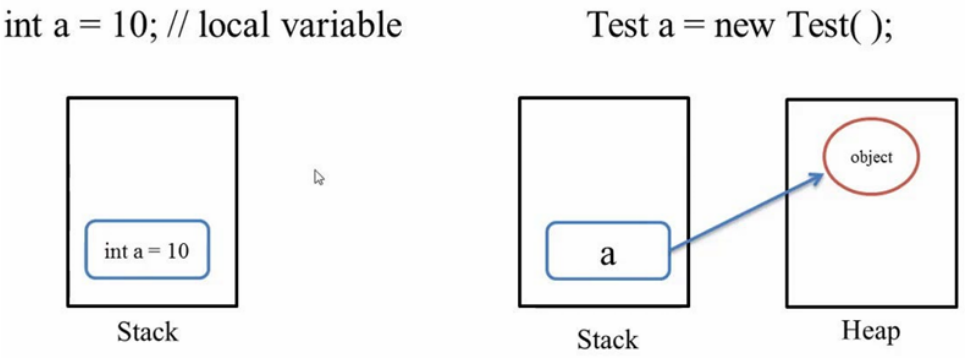


+ Chứa cả dữ liệu và **phương thức** trên dữ liệu. VD: String, Scanner, ArrayList, SinhVien, etc.

+ Tác động lên dữ liệu của đối tượng: gọi các phương thức của chính đối tượng.

+ Biến **không** lưu giá trị trực tiếp, mà chỉ lưu **địa chỉ tham chiếu** đến vùng nhớ heap

+ Nếu String s; thì s sẽ là biến tham chiếu ở trong Stack và trỏ đến **null**



Tạo lớp căn bản (lưu ý về việc hàm main và file):

- Một lớp còn được gọi là một **kiểu dữ liệu mới**

- Một lớp có thể là:

+ Chỉ có thuộc tính: chỉ để chứa dữ liệu

+ Chỉ có phương thức: dùng như công cụ/tính toán

+ Có cả phương thức và thuộc tính: **phổ biến**

+ Lớp không có phương thức và thuộc tính nào là lớp trừu tượng (không thể tạo ra đối tượng được) hoặc placeholder

```
// Lớp placeholder trừu tượng (abstract)

public abstract class Animal {
    // Chưa có phương thức nào, lớp con sẽ triển khai sau
}

// Lớp placeholder không trừu tượng (ít dùng)
public class PlaceholderClass {
    // Rỗng, có thể thêm logic sau
}
```

```
public interface Database {
    void connect(); // Placeholder, triển khai ở lớp cụ thể
}

public abstract class Shape {
    public abstract void draw(); // Placeholder, lớp con phải
override
}
```

- Cú pháp:

```
[phạm vi truy cập] class <tên lớp> {

//các thành phần của lớp: thuộc tính + phương thức
}
```

- Trong một tập tin có thể định nghĩa nhiều lớp, tuy nhiên chỉ được có nhiều nhất một lớp có phạm vi truy cập là public, và có thể lớp có có hàm **main (tên file trùng với lớp public)**
- Nếu **không có lớp nào public** thì có thể đặt tên **file tùy ý**

Cấu trúc khai báo lớp:

```
<access modifier> class classname {
    // declare instance variables
    <access modifier> type var1;
    <access modifier> type var2;
    // declare methods
    <access modifier> type method1(parameters)  {
    //  body of method
    }
    <access modifier> type method2 (parameters) {
    //  body of method
    }
}
```

- Có thể xen lẫn giữa phương thức là thuộc tính lẫn lộn
- Không **thể** dùng **private, protected** cho lớp ở cấp cao nhất. **Inner class** có thể là **private, protected**

Thuộc tính của lớp

- Kiểu dữ liệu: có thể là nguyên thủy hoặc lớp

- Tên thuộc tính: Tên có thể gồm nhiều từ, từ thứ hai trở đi viết hoa chữ cái đầu (camelCase). Phải có ý nghĩa

- Nếu thuộc tính **private** thì việc truy xuất sẽ thông qua các **phương thức**

7. Phương thức

Khái niệm và cú pháp khai báo

- Phương thức trong java là một nhóm các câu lệnh được đặt tên và có thể gọi để thực thi đơn giản bằng cách gọi tên đó

- Phương thức **public** có thể được gọi từ bất cứ nơi đâu trong chương trình

- static: phương thức chỉ có một thể hiện trên một lớp (không cần thiết phải tạo đối tượng để gọi nó). Mặc nhiên phương thức là không phải static

- Cách viết tên phương thức: tương tự như biến.

```
<access modifier> [static] type method1(parameters)  {
    //  body of method
}
```

Phạm vi truy cập (Tương tự thuộc tính)

Mức truy cập	Class	Package	Sub-class	World
private	✓			
no access	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

- **private static** thường dùng cho các hàm **trợ giúp nội bộ (**không cho bên ngoài dùng, và do không cần truy cập đến thuộc tính của đối tượng => dùng static**)**.

```
public class StringUtils {

    public static String normalize(String input) {
        input = removeSpecialChars(input); // gọi hàm trợ giúp nội bộ
        return input.trim().toLowerCase();
    }
    private static String removeSpecialChars(String str) {
        return str.replaceAll("[^a-zA-Z0-9 ]", "");
    }
}
```

- Phương thức không có thân hàm sẽ báo lỗi ngoại trừ

Kiểu dữ liệu trả về

- Thường thì một phương thức sẽ **trả về một giá trị**

- Hàm có thể trả về một giá trị có kiểu dữ liệu nguyên thủy (int, float, …) hoặc là một đối tượng.

- Cú pháp: `return value;`

- Hàm không có kiểu dữ liệu trả về là hàm **void**. Không được phép **return value; (có thể return;)**

Tham số của hàm (Parameters)

- Một số hàm cần phải được cung cấp dữ liệu đầu vào để thực hiện

- Tham số: các dữ liệu truyền vào hàm, các số cách nhau bằng dấu phẩy

- Về bản chất, tham số chính là biến cục bộ của phương thức

- Tham số hình thức: là các tham số trong định nghĩa hay khai báo hàm

VD:
```
void greet(String name) {  // "name" là tham số hình thức
        System.out.println("Hello, " + name);
    }
```
- Tham số thực tế: các biến, biểu thức được truyền vào trong lời gọi hàm

VD: `greet("Alice");  // "Alice" là tham số thực tế`

- Tham số **kiểu dữ liệu nguyên thủy**:

+ Giá trị của tham số thực tế được truyền vào cho tham số hình thức → truyền bằng **giá trị**

+ **Không làm** thay đổi giá trị thực tế

- Tham số **kiểu dữ liệu tham chiếu (đối tượng):**

+ Địa chỉ của đối tượng được tham chiếu bởi tham số thực tế được truyền vào cho tham số hình thức → truyền bằng **tham chiếu**

+ Cả tham số hình thức và thực tế tham chiếu đến cùng một đối tượng

→ Có thể **thay đổi giá trị (thuộc tính) của đối tượng** được tham chiếu bởi tham số thực tế thông qua tham số hình thức

→ Là một cách truyền giá trị trả về cho lời gọi hàm

+ **Không thể tạo** 2 tham số trùng tên nhau được

+ Có thể sử dụng varargs (số lượng tham số không cố định) (Trong một phương thức chỉ có 1 varargs). Cú pháp: `public void printNames(String... names){ … }`

+ Tham số thực tế là đối tượng bằng cách sử dụng toán tử `new Lớp(biến)` sẽ tạo một địa chỉ tham chiếu khác, từ đó không làm thay đổi giá trị ở bên ngoài đối tượng

+ **Đặc biệt:** Nếu trong phương thức một biến tham chiếu được truyền vào thì nếu biến đó sử dụng toán tử **new** thì sẽ mất vùng nhớ ban đầu, tức là không thể truy cập được địa chỉ ban đầu của tham số thực tế nữa (trở thành **truyền tham trị)**

```
void modifyArray(int[] arr) {
    arr[0] = 100;  // Ảnh hưởng đến mảng gốc
```

```
        arr = {1, 2, 3}; // lỗi biên dịch (cú pháp này chỉ để dùng
    để khai báo mảng)
        arr = new int[]{1, 2, 3};  // Không ảnh hưởng
    }
```

+ **String** là bất biến (immutable): khác với class khác là không thay đổi giá trị khi bỏ dô hàm

+ Có thể sử dụng **final** để ngăn thay đổi tham số (nếu cần). VD:

```
void process(final List<String> list) {...} // bị lỗi khi cố thay
```

đổi địa chỉ tham chiếu (vẫn có thể thay đổi giá trị bên trong biến)

VD:

```
public class Main {
    public static void changeStudent(Student s) {
        // Gán giá trị mới cho thuộc tính name → ảnh hưởng đến đối tượng ban đầu
        s.name = "Alice";
        // Tạo đối tượng mới và gán cho s → chỉ thay đổi bản sao của tham chiếu
        s = new Student("Bob");
        System.out.println("Trong phương thức: " + s.name); // Bob
    }
    public static void main(String[] args) {
        Student student = new Student("John");
        changeStudent(student);
        System.out.println("Sau phương thức: " + student.name); // Alice
    }
}
```

+ Ưu điểm của việc truyền tham chiếu:

- Tiết kiệm bộ nhớ: chỉ chứa địa chỉ được truyền vào, không phụ thuộc kích thước của đối tượng
- Có thể sử dụng để **trả về nhiều** hơn một **giá trị** cho lời gọi phương thức
- Tăng tốc độ hàm: chỉ sao chép tham chiếu chứ không sao chép toàn bộ đối tượng (dữ liệu)

Đệ qui:

- Có hỗ trợ đệ qui

```
public static int factorial(int n) {
    if (n == 0 || n == 1) { // Điều kiện dừng
        return 1;
    } else {
        return n * factorial(n - 1); // Gọi đệ quy
    }
}
```

- Nhược điểm:

+ Tốn bộ nhớ do mỗi lần gọi đệ quy lưu trữ trạng thái vào **Stack**.

+ Nguy cơ **Stack Overflow** nếu đệ quy quá sâu (vd: factorial(100000)).

+ Hiệu suất thường **kém** hơn vòng lặp (do overhead gọi hàm).

8. Khai báo & tạo đối tượng

- Khai báo đối tượng: `<tên lớp> <tên đối tượng>;` (Chỉ khai báo biến tham chiếu student1 (chưa trỏ đến đối tượng nào)), giá trị mặc định là `null`

- Tạo đối tượng: `<tênĐốiTượng> = new <TênLớp>();` (Tạo một đối tượng mới trong Heap Memory. Cấp phát bộ nhớ cho các thuộc tính của lớp, sau đó gán địa chỉ vào biến mới tạo

- Đối tượng cũng là biến, cú pháp giống khai báo biến, trong đó tên lớp đóng vai trò kiểu dữ liệu

- Tạo đối tượng: dùng toán tử new: `new <tên lớp>`

- Khi tạo ra đối tượng thì đối tượng cũng sẽ có một bản sao các thuộc tính của lớp

- Khởi tạo đối tượng với **Initializer Block**

```
class Student {
    String name;
    int age;

    // Initializer block (chạy trước constructor)
    {
        name = "Default";
        age = 18;
    }
}
```

Lưu ý:

- Một biến tham chiếu được gán là **null** thì không thể dùng các phương thức của đối tượng lên trên nó (sẽ bị NullPointerException), còn "" (String) hay `new <tên lớp>()` thì **được**.

Các lỗi liên quan đến khởi tạo đối tượng:

- Không Khởi Tạo Đối Tượng Trước Khi Sử Dụng

```
Student student;
student.displayInfo(); // Lỗi: variable student might not have been initialized
```

- Gọi Phương Thức Trên null

```
Student student = null;
System.out.println(student.getName()); // NullPointerException
```

- Nhầm Lẫn Giữa Khai Báo Và Khởi Tạo

```
Student student; // Chỉ khai báo
student = new Student(); // Khởi tạo
```

9. Khởi tạo và hủy đối tượng

Hàm xây dựng (Constructor)

- Là phương thức **đặc biệt** của lớp

- Dùng để **khởi tạo** đối tượng khi nó được tạo ra bằng từ khóa new

- Có 2 loại:

+ Hàm khởi tạo không có tham số, còn được gọi là hàm xây dựng mặc nhiên (default constructor)

+ Các hàm khởi tạo có tham số

- Cú pháp:

+ **Phạm vi truy cập:** public, private, no modifier, protected (bất kỳ)

+ **Tên hàm:** trùng với tên lớp

+ **Kiểu dữ liệu trả về:** không có, kể cả void

- Hàm xây dựng mặc định được cung cấp khi chưa khai báo hàm xây dựng

- Khi **đã khai báo** hàm xây dựng, thì hàm xây dựng **mặc định sẽ bị xóa**

- Nếu trong những thuộc tính của lớp có một thuộc tính là biến **Đối tượng**, thì bắt buộc phải khai báo constructor để tránh bị lỗi

- Một số tính chất:

+ Hàm xây dựng sẽ **tự động được gọi** khi đối tượng được tạo ra.

+ Có thể khởi tạo biến, gọi phương thức non-static và static

+ Hàm xây dựng **cũng là 1 phương thức** của lớp nên nó có thể được **tái định nghĩa.**

+ Trong trường hợp có nhiều hàm xây dựng, hàm nào được gọi là tùy vào **danh sách tham số truyền** vào trong câu lệnh tạo đối tượng.

+ Có thể có nhiều constructor (Overloading). Lớp có bao nhiêu hàm xây dựng, có bấy nhiêu cách để tạo đối tượng.

+ Hàm xây dựng không thể được gọi trực tiếp mà chỉ có thể tạo khi khai báo đối tượng mới.

Tiêu chí	Hàm xây dựng không tham số	Hàm xây dựng có tham số	Hàm xây dựng sao chép
Định nghĩa	Hàm xây dựng không nhận tham số	Hàm xây dựng nhận một hoặc nhiều tham số	Hàm xây dựng nhận một đối tượng cùng kiểu

Mục đích	Tạo đối tượng với giá trị mặc định	Tạo đối tượng với giá trị khởi tạo theo yêu cầu	Tạo đối tượng mới sao chép dữ liệu từ đối tượng khác
Ví dụ	`Vehicle() { }`	`Vehicle(int p, int f, int m) { }`	`Vehicle(Vehicle v) { }`
Cách gọi	`Vehicle v = new Vehicle();`	`Vehicle v = new Vehicle(4, 10, 20);`	`Vehicle v2 = new Vehicle(v1);`
Tính linh hoạt	Thường dùng để khởi tạo mặc định	Cho phép khởi tạo theo nhiều cách khác nhau	Tạo bản sao nhanh chóng của đối tượng
Tính đặc biệt	Java sẽ tự tạo nếu không khai báo	Phải tự định nghĩa	Phải tự định nghĩa

Lưu ý: Copy constructor trong kế thừa

- Constructor không được kế thừa (không thể public LớpCha trong một lớp con)

- Lớp con phải gọi constructor của lớp cha (dùng super).

- Nếu lớp con không có constructor mặc định thì bắt buộc phải gọi super(tham số)

Trường hợp	Có thể copy?	Ghi chú
Copy đối tượng **cha → con**	❌ Không an toàn	Không đủ thông tin
Copy đối tượng **con → cha**	✅ Có thể	Nhưng chỉ giữ dữ liệu lớp cha
Copy đối tượng **con → con**	✅ Tốt nhất	Dùng copy constructor lớp con

Hàm hủy đối tượng (desctructor)

- Trong java, hàm hủy được gọi là **finalizer (**hàm kết thúc**)**

- Cú pháp hàm hủy:

+ Không có đối số và kiểu dữ liệu trả về, kể cả void.

+ Một lớp chỉ được phép có **tối đa 1 hàm** kết thúc.

+ Tên của hàm hủy: finalize.

+ Được thực hiện **trước** khi Garbage Collector (GC) thực hiện công việc này

+ Có thể ghi đè từ lớp Object

- Trong Java, thông thường hàm này ít được sử dụng vì JVM đã cài đặt cơ chế **Garbage Collection** để giải phóng các đối tượng không cần thiết.

- GC giúp người lập trình không cần phải viết code để giải phóng đối tượng do đó không bao giờ quên giải phóng đối tượng

- GC sử dụng cơ chế đếm

+ Mỗi đối tượng có một số đếm đến các tham chiếu trỏ tới

+ Giải phóng đối tượng khi số đếm **bằng 0**

- Giải phóng các đối tượng khi không còn sử dụng

+ Kiểm tra tất cả các tham chiếu

+ Giải phóng các đối tượng không còn được tham chiếu

10. Phép gán đối tượng

- Phép gán đối tượng còn được gọi là phép sao chép cạn:

+ Cái được sao chép chính là tham chiếu: phép gán `obj1 = obj2` làm cho `obj1` và `obj2` tham chiếu đến cùng 1 đối tượng

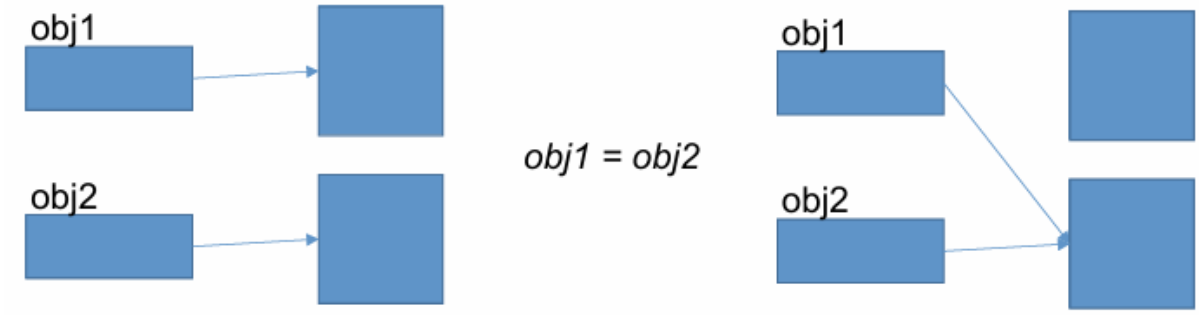
+ Điều này có thể dẫn đến các lỗi ngoài mong đợi trong chương trình: `obj1` và `obj2` luôn có **cùng 1 giá trị ,** gây ra những lỗi không mong muốn (thay đổi giá trị của biến này ở trên biến khác)

- Khắc phục:

+ Sử dụng toán tử new với sao chép sâu

+ Sử dụng clone() nếu lớp triển khai Cloneable (phải bắt buộc triển khai)

+ Sử dụng thư viện sao chép đối tượng (SerializationUtils.clone())



11. Sao chép sâu (deep copy)

- Muốn thực hiện sao chép dữ liệu của đối tượng thay vì tham chiếu, ta phải tự viết các phương thức để sao chép dữ liệu

- Thao tác sao chép đó được gọi là sao chép sâu

- Có 2 phương pháp

+ Sử dụng constructor sao chép:

- Đối với thuộc tính là Class, thì new Class(biến);
- Đối với mảng 1 chiều (nguyên thủy hoặc đối tượng: dùng clone() hoặc Copyof()
- Đối với ArrayList sử dụng đối tượng: **thủ công** từng giá trị một

+ Phương thức: makeCopy();

- Cách copy: s2.makeCopy(s1)

- Lưu ý: s2 khởi tạo bắt buộc phải có new Class();
- Nội dung của phương thức (như hàm xây dựng sao chép)

+ Phương thức: getCopy();

- Cách copy: s2 = s1.getCopy();
- Lưu ý: s2 khởi tạo **không** bắt buộc phải có new Class();
- Nội dung của phương thức:

```
public TenLop getCopy() {
   return new TenLop(this); // Gọi constructor sao chép
}
```

+ Sao chép mảng:

- Dùng:

```
this.authors = authors.clone(); // nếu không cần sao chép sâu
this.authors = Arrays.copyOf(authors, authors.length);
```

+ Mảng 2 chiều: thủ công

Lưu ý về sao chép:

- Nếu là mảng kiểu nguyên thủy: => chỉ cần .clone() (nhớ cần **implement** hoặc là **ép kiểu** về chính lớp đó) là sao chép sâu

- Nếu là mảng đối tượng => Cần chạy vòng lặp for và .clone() hoặc new đối tượng từng phần tử trong mảng

- final field không thể gán qua clone() (final fields lại **được yêu cầu** phải khởi tạo hợp lệ tại lúc khởi tạo đối tượng trong khi cơ chế clone lại không có)

12. Toán tử quan hệ

- So sánh nội dung của các dữ liệu **nguyên thủy**

- So sánh nội dung của **tham chiếu** chứ không so sánh nội dung của đối tượng do tham chiếu trỏ tới

- Muốn có thể so sánh được giữa 2 tham chiếu phải dùng đến phương thức equals() (không dùng cho biến mảng)

Equals()

- Lưu ý khi override equals()

- Phải kiểm tra:

+ this == obj: cùng tham chiếu

+ obj == null: kiểm tra null

+ getClass() != obj.getClass(): đảm bảo đúng loại

+ So sánh từng thuộc tính, nếu là String dùng .equals(), nếu là số thì dùng ==.

+ Nếu lớp override equals(), nên override luôn hashCode() để dùng trong các cấu trúc như HashMap, HashSet (theo nguyên tắc: bằng nhau → hashCode phải bằng nhau). (record hỗ trợ sẵn)

Phân biệt getClass và instanceof

- getClass(): "Bạn là chính xác lớp nào?" → Kiểm tra class chính xác.

```
Object obj = "Hello"; // obj đại diện cho lớp cụ thể là String
System.out.println(obj.getClass() == String.class); // true
System.out.println(obj.getClass() == Object.class); // false (dù
String kế thừa Object)
```

Biến	Thực tế new	instanceof Animal	instanceof Dog	getClass() == Animal.class	getClass() == Dog.class
`Animal a = new Animal()`	Animal	✅ true	❌ false	✅ true	❌ false
`Animal a = new Dog()`	Dog	✅ true	✅ true	❌ false	✅ true
`Dog d = new Dog()`	Dog	✅ true	✅ true	❌ false	✅ true

Phương pháp: nhìn theo bên phải

- instanceof: "Bạn có phải là kiểu này không?" → Kiểm tra kiểu + kế thừa.

```
Object obj = "Hello";
System.out.println(obj instanceof String);  // true
System.out.println(obj instanceof Object); // true (vì String kế
thừa Object)
System.out.println(obj instanceof CharSequence); // true (vì String
implements CharSequence)
```

Biến	Khởi tạo thực tế	instanceof Dog	instanceof Animal	Giải thích
`Animal a1`	`new Animal()`	❌ false	✅ true	`a1` là `Animal`, không phải `Dog`
`Animal a2`	`new Dog()`	✅ true	✅ true	`a2` là `Dog`, kế thừa `Animal`
`Dog d1`	`new Dog()`	✅ true	✅ true	`d1` là `Dog`, cũng là `Animal`
`Dog d2`	`new Animal()` ❌	❌ Compilation error	❌ Compilation error	Không thể gán `Animal` vào biến `Dog` mà không ép kiểu

Phương pháp: nhìn cha nó và con nó

- `Instanceof` an toàn **với null** (không ném lỗi) => Luôn ra false
- `instanceof` chỉ sử dụng được nếu như 2 vế có quan hệ kế thừa

13. Thành phần tĩnh (static members)

- Là thuộc tính dùng dung cho tất cả các đối tượng của lớp. Không thuộc về từng đối tượng riêng lẻ, mà thuộc về lớp.

- Từ khóa static.

- Được gọi là **thành phần** "của lớp", dính với lớp

- Tồn tại độc lập với đối tượng: có bao nhiêu đối tượng được tạo ra thì vẫn chỉ có 1 bản (copy) của thành phần tĩnh trong bộ nhớ.

- Được truy xuất trực tiếp thông qua lớp: `<tên lớp/ tên đối tượng>.<tên thành phần>`

Thuộc tính tĩnh:

- Thuộc tính được khai báo với từ khóa static là thuộc tính tĩnh.

```
public class StaticExample {
    //  ◇  Phương pháp 1: Khởi tạo trực tiếp
    static int a = 10;
    //  ◇  Phương pháp 2: Khởi tạo bằng static block
    static int x;
    static int y;
    static {
        x = 5;
        y = 15;
    }
}
```

- Static với inner Class:

```
    class Outer {
        private static int x;
        static class Inner {  // Static inner class
            void access() {
                System.out.println(x);  // Truy cập được static của
    Outer
            }
        }
    }
```

- Có thể import được từ lớp khác (nếu phạm vi truy cập cho phép)

```
    import static java.lang.Math.PI;
    import static java.lang.Math.pow;

    public class Circle {
        double area(double radius) {
            return PI * pow(radius, 2);  // Dùng trực tiếp PI và pow
        }
    }
```

Chú ý:

+ Có thể thay đổi giá trị của biến static thông qua một đối tượng cụ thể (`biến.tên_biến_static`)

+ Có thể tương tác và thay đổi biến static thông qua **phương thức non-static (kể cả constructor)**

+ **Có thể** sử dụng **this** với static. Vì this tham chiếu đến đối tượng hiện tại, trong khi static thuộc về lớp

Phương thức tĩnh (static)

- Các phương thức tĩnh **chỉ được truy xuất** các thuộc tính tĩnh (không truy xuất được thuộc tính non-static và phương thức non-static do nó là qua đối tượng)

- Không thể sử dụng **this hoặc super** (**bao gồm cả hàm main**)(vì không gắn với đối tượng cụ thể).

- Không thể ghi đè (tương tự như **final**) nhưng có thể nạp chồng (ví dụ hàm add)

- Phương thức **không tĩnh có thể** truy xuất **thuộc tính tĩnh.**

- Phương thức **không tĩnh không thể** truy cập qua **lớp**

- Có thể được gọi thông qua một đối tượng cụ thể (tương tự như thuộc tính tĩnh)

- Có thể nhầm lẫn giữa override và method hiding (**static và biến non-static không** hỗ trợ override)

VD:

```
public class StaticTest {
    static int staticCount = 100;  // Biến tĩnh (thuộc về lớp)
    int instanceCount = 5;         // Biến thường (thuộc về đối tượng)
    // Phương thức tĩnh
    public static void staticMethod() {
        System.out.println("Trong staticMethod:");
        System.out.println("staticCount = " + staticCount); // ☑ OK
        // System.out.println("instanceCount = " + instanceCount); ❌ Lỗi: 
không thể truy cập biến thường
        // ❗ Không thể dùng this hoặc biến instance tại đây
        System.out.println(this);  // Lỗi: 'this' cannot be referenced from 
static context
    }// Phương thức không tĩnh
    public void instanceMethod() {
        System.out.println("Trong instanceMethod:");
        System.out.println("staticCount = " + staticCount);   // ☑ OK: 
Truy cập biến static
        System.out.println("instanceCount = " + instanceCount); // ☑ OK
    }
}
```

14. Biến tham chiếu this

- Trỏ đến **đối tượng hiện tại** đang được thao tác

- Chỉ sử dụng được trong **phương thức không tĩnh** và **constructor** (mặc định là ngầm sử dụng)

- Trong tất cả các phương thức không tĩnh, tất cả các lệnh truy xuất vào các dữ liệu thành viên không tĩnh sẽ được tự động thêm vào tham chiếu this.

- Cú pháp: `this.<tên thuộc tính>` hoặc `this()`

- `this()` **chỉ được dùng** trong constructor (bắt buộc ở dòng đầu tiên)

- Java cung cấp tham chiếu this để trỏ tới **chính đối tượng** đang hoạt động

- **Không thể** gọi this ở trong hàm main

- Mục đích:

+ Tham chiếu tường minh đến thuộc tính và phương thức của đối tượng tránh trùng tên (nếu dùng name = name sẽ bị che khuất thuộc tính)

+ Truyền đối tượng hiện tại làm tham số `printJob(this);`

+ Truyền tham số và giá trị trả về của phương thức

+ Dùng để gọi constructor trong **cùng** 1 lớp (`this (1, 1)  // gọi constructor có 2 tham số`). Lưu ý:

- Chỉ được sử dụng **1 lần** trong **1 phương thức xây dựng**
- Lệnh gọi this() phải đứng ở **dòng đầu tiên** của constructor (tương tự như super()) và đương nhiên 2 cái này không được xuất hiện cùng lúc trong một hàm
- Constructor chỉ được gọi bên trong một constructor khác và chỉ được gọi 1 lần ở thời điểm đầu tiên

```
class Outer {

    int x = 10;
    class Inner {
        int x = 20;
        void print() {
            System.out.println(x);         // 20 (của Inner)
            System.out.println(this.x);    // 20 (của Inner)
            System.out.println(Outer.this.x); // 10 (của Outer)
        }
    }
}
```

15. Kết tập (Aggregation)

Định nghĩa

- Kết tập (Aggregation) là một **mối quan hệ "has-a"** giữa hai lớp trong lập trình hướng đối tượng, trong đó:

+ Một đối tượng này *có chứa* (sở hữu) đối tượng khác, nhưng cả hai vẫn có thể tồn tại độc lập.

+ **Là dạng đặc biệt** của **Association** với tính chất "whole-part" (toàn thể - bộ phận)

📌 **Đặc điểm của kết tập:**

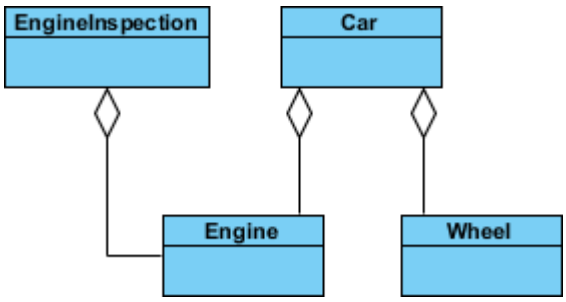
- Mối quan hệ có hướng: A *has-a* B.

- Đối tượng chứa (A) **không sở hữu hoàn toàn** đối tượng bị chứa (B). Tức đối tượng bị chứa có thể chia sẽ cho nhiều đối tượng

- Nếu A bị hủy thì B **vẫn có thể tồn tại** độc lập.

- Một đối tượng B có thể thuộc về **nhiều** đối tượng A

- Thể hiện sự **gắn kết lỏng lẻo** hơn so với *composition (thành phần)*.



16. Thành phần (Composition)

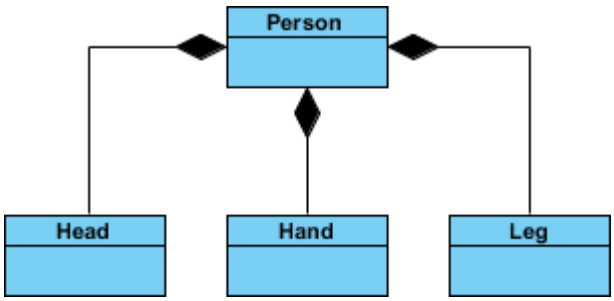
Định nghĩa

- Composition là một mối quan hệ "**has-a**" giống như **kết tập (aggregation)**, nhưng **mạnh hơn và chặt chẽ hơn**. Trong composition:

- Một đối tượng A *sở hữu hoàn toàn* đối tượng B, và **vòng đời của B phụ thuộc vào A** — nếu A bị huỷ thì B cũng bị huỷ theo.

🔍 **Đặc điểm của Composition:**

Đặc điểm	Giải thích
A has-a B	A chứa B, nhưng **A sở hữu hoàn toàn B**
B không thể tồn tại độc lập	Nếu A bị hủy, B cũng bị hủy
A kiểm soát vòng đời của B	B được tạo ra bên trong A
Gắn kết chặt chẽ (tight coupling)	Các đối tượng gắn bó mật thiết, thay đổi A sẽ ảnh hưởng đến B
Không dùng chung	B chỉ thuộc về 1 A duy nhất, không chia sẻ



Lưu ý:

Trường hợp	Aggregation	Composition
`this.nv = nv;` – dùng tham chiếu đã có	✅	❌
`this.nv = new NhanVien(nv);` – tạo bản sao	❌	✅

17. Phụ lục

Sơ đồ lớp (Class Diagram)

✅ Khái niệm:

Sơ đồ lớp mô tả cấu trúc tĩnh của một hệ thống theo mô hình hướng đối tượng.

Nó thể hiện các lớp, thuộc tính, phương thức, và các mối quan hệ giữa các lớp trong hệ thống.

🧱 Cấu trúc sơ đồ lớp bao gồm:

1. Lớp (Class):

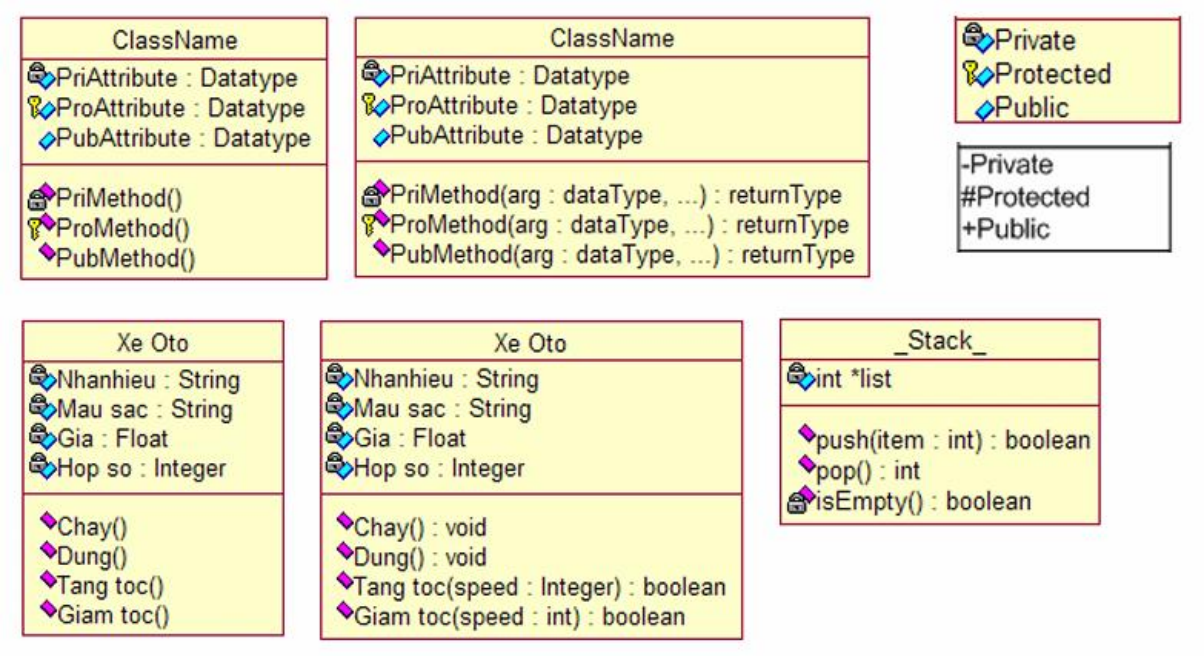
- Biểu diễn một thực thể trong hệ thống.

- Gồm 3 phần:

  + Tên lớp

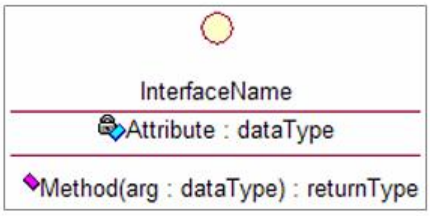
  + Các thuộc tính (fields)

  + Các phương thức (methods)



2. Giao diện (Interface):

Biểu diễn một nhóm hành vi mà các lớp có thể thực hiện (implement).



Ký hiệu bằng hình chữ nhật giống lớp, có thêm từ khóa `<<interface>>`.

3. Các mối quan hệ (Relationships):

a. Kết hợp (Association):

Quan hệ đơn giản giữa hai lớp.

Ví dụ: `Student` -- `Course`: một sinh viên học nhiều khóa học.

b. Tập hợp (Aggregation):

Quan hệ *"has-a"* lỏng lẻo.

Lớp A chứa B, nhưng B vẫn tồn tại độc lập.

Ký hiệu: Hình thoi rỗng (◊——).

c. Tổng hợp (Composition):

Quan hệ *"has-a"* chặt chẽ.

Lớp A tạo và sở hữu hoàn toàn B, nếu A mất thì B cũng mất.

Ký hiệu: Hình thoi tô đen (◆——).

d. Hiện thực hóa (Realization):

Giao diện được một lớp hiện thực (implement).

Ký hiệu: Mũi tên đứt nét từ lớp đến giao diện.

e. Thừa kế / Tổng quát hóa (Generalization):

Quan hệ cha – con giữa các lớp.

Lớp con kế thừa thuộc tính và hành vi từ lớp cha.

Ký hiệu: Mũi tên rỗng hướng về lớp cha.

📌 Tóm tắt ký hiệu UML phổ biến trong sơ đồ lớp:

Mối quan hệ	Ký hiệu UML	Mô tả
Association	———————	Quan hệ thông thường
Aggregation	◊———————	Quan hệ "has-a", lỏng lẻo
Composition	◆———————	Quan hệ "has-a", chặt chẽ
Inheritance	△———————	Thừa kế
Realization	△——— — —	Giao diện được cài đặt

Chương 3: Thừa kế và Đa hình

1. Tính bao gói

Khái niệm

- Tính bao gói (encapsulation): là một trong những đặc điểm **quan trọng** của OOP

- Thể hiện ở:

+ Sự kết hợp chặt chẽ giữa dữ liệu và thao tác của cùng một đối tượng ⇒ Chi tiết của một đối tượng được che dấu đi, giảm sự phức tạp.

+ Việc nhóm chung dữ liệu và phương thức của một đối tượng

+ Ngăn chặn sự tác động từ đối tượng khác lên dữ liệu ⇒ xác định và giới hạn truy cập đến các thành phần của một đối tượng thông qua mức độ phạm vi (Chỉ có thể truy cập qua getter/setter)

Thuộc tính truy cập:

Access Modifier	Truy cập từ cùng lớp	Truy cập từ cùng package	Truy cập từ lớp con (subclass)	Truy cập từ bên ngoài
`private`	✔ Có	✘ Không	✘ Không	✘ Không
`default`	✔ Có	✔ Có	✘ Không	✘ Không
`protected`	✔ Có	✔ Có	✔ Có	✘ Không
`public`	✔ Có	✔ Có	✔ Có	✔ Có

Mối liên hệ mật thiết giữa tính bao gói và trừu tượng hóa

- Cùng mục tiêu: Ẩn giấu thông tin cần thiết, chỉ cho phép người dùng làm việc với giao diện bên ngoài

- Trừu tượng hóa tập trung vào những đặc điểm thuộc về bản chất của đối tượng trong khi sự bao gói tập trung vào sự cài đặt những tính chất đó.

- Bao gói là **công cụ để thực hiện trừu tượng hóa**

+ Bao gói giúp che dữ liệu → dữ liệu không bị truy cập trực tiếp

+ Giúp che đi các chi tiết không cần thiết hoặc không phải là các thuộc tính không phải là bản chất của đối tượng

- Bao gói đảm bảo rằng trừu tượng hóa luôn được giữ vững: Người dùng chỉ có thể tương tác qua các phương thức được thiết kế sẵn

- Càng lên cao càng trừu tượng hóa

2. Thừa kế (Inheritance)

Khái niệm về thừa kế

- Thừa kế (Inheritance) là một trong những đặc điểm **quan trọng**, khả năng định nghĩa một lớp mới dựa trên một lớp đã có (lớp cơ sở - superclass) và mở rộng hoặc sửa đổi các đặc điểm của lớp đó.

- Một lớp (subclass) nhận tất cả thuộc tính và phương thức từ lớp khác (superclass).

- Mở rộng hoặc sửa đổi chức năng mà không cần viết lại từ đầu

- Dùng để mô hình hóa mối quan hệ "là" ("is a") giữa các lớp/đối tượng với nhau:

+ Đối tượng "thừa kế" là một đối tượng **đã có sẵn khác**, với những thuộc tính và phương thức **"tương tự"** nhau.

+ Thừa kế sử dụng "sự tương tự" (similarities) và "sự khác nhau" (differences) để mô hình một nhóm các đối tượng có liên quan với nhau.

- Lớp được thừa kế: Lớp cha (superclass), lớp cơ sở (based class)

- Lớp con (subclass), lớp dẫn xuất (deriverd class) sẽ kế thừa tất cả các thuộc tính và phương thức của lớp cha.

Lợi ích của thừa kế

- Tái sử dụng mã nguồn.

- Giảm thiểu việc lặp lại mã.

- Tăng khả năng mở rộng và bảo trì chương trình.

- Tận dụng lại các thuộc tính chung, phương thức tương tự

- Thiết kế lớp gọn nhẹ, đơn giản hơn

- Ghi đè phương thức: Thay đổi hành vi phương thức kế thừa

- Đa hình (Polymorphism): Sử dụng đối tượng lớp con thay thế lớp cha

Cấu trúc và cú pháp thừa kế

Lớp con sử dụng từ khóa `extends` để kế thừa lớp cha.

```
public class A {
    // Lớp cha
}
public class B extends A {
    // Lớp con kế thừa A
}
```

Phân loại kế thừa

- Đơn kế thừa: Một lớp chỉ kế thừa một lớp cha duy nhất

```
class Vehicle {}
class Car extends Vehicle {}  // OK
```

- Kế thừa đa cấp: Chuỗi kế thừa nhiều tầng

```
class Animal {}
class Mammal extends Animal {}
class Dog extends Mammal {}
```

- Kế thừa phân cấp: Nhiều lớp con cùng kế thừa một lớp cha

```
class Shape {}
class Circle extends Shape {}
class Square extends Shape {}
```

Tính chất thừa kế

- Java không hỗ trợ thừa kế đa lớp (sử dụng interface thay thế)

- Lớp con kế thừa:

+ Các thuộc tính và phương thức **public và protected** của lớp cha.

+ Không kế thừa thành viên private và constructor

+ Các thuộc tính và phương thức có phạm vi **default** nếu cùng **package**.

+ Không kế thừa các thuộc tính **private** (có thể kế thừa gián tiếp qua phương thức public).

- Lớp con có thể:

+ Dùng lại các phương thức/thuộc tính của lớp cha (nếu cho phép)

+ Ghi đè các phương thức của lớp cha (Override). (phương thức non-static, **tuy nhiên** phương thức ghi đè static chỉ có thể hide phương thức cha)

+ Khối static không được kế thừa

```
class Cha {
    static {
        System.out.println("Khối static trong lớp Cha");
    }
    public Cha() {
        System.out.println("Constructor của lớp Cha");
    }
}
class Con extends Cha {
```

```
    static {
        System.out.println("Khối static trong lớp Con");
    }
    public Con() {
        System.out.println("Constructor của lớp Con");
    }
}
public class Main {
    public static void main(String[] args) {
        Con c = new Con();
    }
}
//Khối static trong lớp Cha
//Khối static trong lớp Con
//Constructor của lớp Cha
//Constructor của lớp Con
```

+ Viết thêm các phương thức và thuộc tính riêng. **(Khai báo A a = new B() sẽ không sử dụng được trừ khi phải down casting lại)**

- Biến static chia sẻ giá trị của biến cho lớp con, và giá trị đó có thể che giấu biến của lớp cha.

- Biến non-static: tương tự

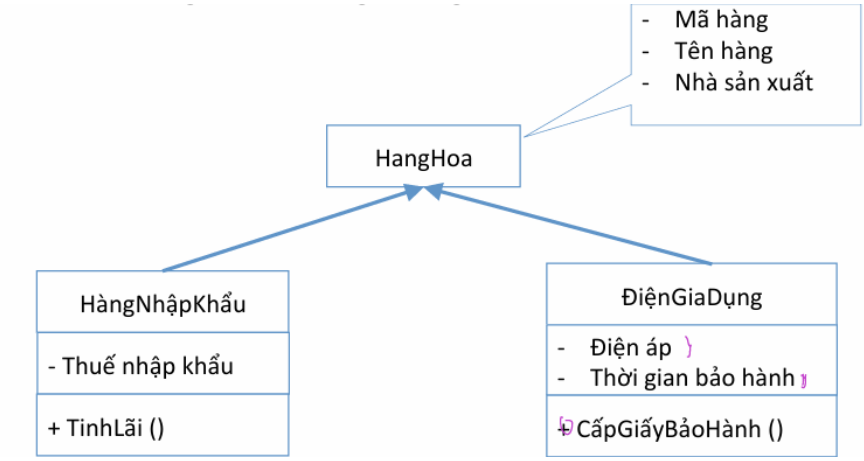
Ghi chú quan trọng

- Java không hỗ trợ đa thừa kế bằng lớp. Một lớp chỉ có thể kế thừa trực tiếp từ một lớp cha.

- Java hỗ trợ đa thừa kế thông qua interface.

- Tránh thiết kế sai về mặt ý nghĩa: VD: lớp XeHoi kế thừa từ lớp BanhXe là sai

Ví dụ minh hoạ (Circle – Cylinder, Person – Student)



Phân biệt kết tập và kế thừa

Tiêu chí	Kết tập (Aggregation)	Kế thừa (Inheritance)
Khái niệm	Là mối quan hệ **có (has-a)** giữa các đối tượng. Một đối tượng chứa đối tượng khác như một thành phần.	Là mối quan hệ **là (is-a)** giữa các lớp. Một lớp con kế thừa thuộc tính và hành vi của lớp cha.
Cú pháp	Sử dụng thuộc tính là đối tượng của lớp khác.	Sử dụng từ khóa `extends` trong Java.
Mối quan hệ	"A có B" (A has a B).	"A là B" (A is a B).
Ràng buộc	Quan hệ lỏng lẻo. Hai đối tượng có thể tồn tại độc lập.	Quan hệ chặt chẽ. Lớp con gắn chặt với lớp cha.
Sơ đồ UML	Dùng đường thẳng có hình thoi rỗng.	Dùng đường thẳng với mũi tên rỗng (tam giác).

Quy tắc trong thừa kế

- Lớp con thừa kế có tất cả các thành phần của lớp cha **trừ** phương thức xây dựng

- Lớp con chỉ có thể truy xuất các thành phần public, protected, no modifier của lớp cha

- Lớp con có thể có thêm các thuộc tính, các phương thức mới

- Lớp con có thể nạp đè (overriding) các phương thức của lớp cha

- Lớp cha không thể truy xuất được các thành phần của lớp con

- Các thành phần của lớp con sẽ che đi các thành phần trùng tên trong lớp cha

    + Nếu trùng tên đó sẽ truy xuất thành phần của lớp con

    + Muốn truy xuất cha thì phải chỉ rõ: super.<tên hàm/biến>

- Final methods: không thể override

- Final classes: không thể kế thừa

- **Trùng tên biến (**dù khác kiểu dữ liệu**)** cũng có thể gây ra che biến của lớp cha

Hàm xây dựng trong thừa kế

- Lớp con không kế thừa các phương thức khởi tạo lớp cha

- Có hai phương pháp gọi phương thức xây dựng của lớp cơ sở:

    + Sử dụng phương thức khởi tạo mặc định (Trình biên dịch tự chèn super(); vào đầu **constructor** của Child.)

    + Gọi hàm xây dựng một cách tường minh (super(<tham số>) để dùng hàm xây dựng có tham số, **bắt buộc** phải là dòng đầu tiên)

- Việc khởi tạo thuộc tính của lớp cơ sở **nên giao phó** cho constructor của lớp cơ sở (lớp cha nên tự khởi tạo thuộc tính của nó thông qua constructor)

- Sử dụng từ khóa **super** để gọi constructor của lớp cơ sở:

+ Lớp cơ sở (lớp cha) bắt buộc phải được thực hiện đầu tiên

+ Nếu lớp cơ sở không có thì tự java sẽ tự động gọi constructor mặc định

+ Nếu cả lớp con và lớp cơ sở đều không có constructor thì cả 2 lớp đều sẽ tự động có (kể cả có super ở lớp con)

VD:

```
public class Cylinder extends Circle {
    // Các dữ liệu thành viên (chưa hiển thị, nhưng có thể là:
private int height;)
    public Cylinder() {
        super(); // Gọi constructor mặc định của lớp cha
(Circle)
        height = 0; // Khởi tạo chiều cao mặc định cho Cylinder
    }
    public Cylinder(int r, int height) {
        super(r); // Gọi constructor tham số của cha (Circle),
truyền bán kính r
        this.height = height; // Gán chiều cao cho Cylinder
    }
    // Các hàm thành viên khác...
}
```

Lưu ý: phân biệt tham số với thuộc tính cùng tên

```
class M {
    int i;
    public M(int i){
        this.i = --i;
    }
}

 class N extends M{
    public N(int i) {
        super(++i);
        System.out.println(i);   // i tham số
        System.out.println(this.i);  // i của lớp M
    }
}
```

Từ khóa `super`

- Là một tham chiếu đến lớp cha của một lớp

- Chỉ hoạt động trong **phương thức không tĩnh và constructor**

- Giúp truy xuất đến thành phần của lớp cha:

+ super.([đối số]): hàm xây dựng (chỉ được dùng trong hàm xây dựng)

+ super.<phương thức|thuộc tính>: truy xuất đến **thành viên** của cha (nếu khác private)

- Chú ý:

+ super() phải là lệnh **đầu tiên** trong constructor, dùng được 1 lần trong contructor

+ Nếu không viết super(), Java tự động chèn super()

+ Nếu cha không có constructor mặc định, thì bắt buộc phải gọi super(…) tường minh // nếu không sẽ bị lỗi

+ Không super 2 lần được (gọi ông)

+ **Không thể** gọi super ở ngoài hàm main

```
class Parent {
    Parent(int x) {}
}

class Child extends Parent {
    Child() {} // Lỗi: No default constructor in Parent
}
```

- Trường hợp đặc biệt:

+ Kết hợp cả this và super(): không thể sử dụng đồng thời trong cùng constructor

Thành phần `protected` và `final`

- Thành phần `protected`: Đây là các thành phần dành cho các lớp con cháu

- Thành phần `final`:

+ phương thức: các thành phần không được nạp đè trong lớp con (cố gắng nạp đè sẽ báo lỗi)

+ Được sử dụng để đảm bảo thành phần này chỉ được sử dụng bởi các lớp con hơn là thay đổi (nạp đè chúng)

+ **Không nên dùng** kết hợp với **phương thức abstract, vì** lớp **abstract** có **phương thức abstract** cần phải **ghi đè lên**

+ Khi dùng với thuộc tính, thì nó là const

+ Khi là tham số, không được thay đổi giá trị của tham chiếu

+ Lớp final: không định nghĩa được lớp dẫn xuất, **không kế thừa được**

3. Đa hình (Polymorphism)

Khái niệm và mục đích:

- Các loại đối tượng khác nhau có thể có cách ứng xử khác nhau cho cùng một thông điệp

Kỹ thuật thực hiện:

- Dùng hàm ảo kết hợp với nạp đè hàm (overriding): Khai báo phương thức ảo trong lớp cha chung. Mỗi lớp con sẽ cài đặt theo những cách khác nhau

- Dùng tái định nghĩa hàm (overloading): Định nghĩa các hàm trùng tên với nhau.

Nạp chồng (overloading): là tái định nghĩa phương thức

Yếu tố	Yêu cầu
Tên phương thức	✔ Phải giống nhau
Danh sách tham số	✔ Phải khác nhau (về số lượng, kiểu dữ liệu, hoặc thứ tự)
Kiểu trả về	❌ Không ảnh hưởng (có thể giống hoặc khác)
Phạm vi truy cập	Không quan trọng
Từ khóa `static` hay `instance`	Không quan trọng

```
class MathUtils {
    // Đây là overload - nạp chồng: cùng tên, khác tham số
    public int add(int a, int b) {
        return a + b;
    }
    // Đây là overload - nạp chồng: cùng tên, khác kiểu dữ liệu
tham số
    public double add(double a, double b) {
        return a + b;
    }
    // Đây là overload - nạp chồng: cùng tên, khác số lượng tham số
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    // ❌ Đây KHÔNG PHẢI là overload: khác tên phương thức
    public int sum(int a, int b) {
        return a + b;
    }
    // ❌ Đây KHÔNG PHẢI là overload: cùng tên, cùng tham số, chỉ
khác kiểu trả về → LỖI BIÊN DỊCH
    // public double add(int a, int b) {
```

```
        //     return (double) (a + b);
        // }
        // 👉 Java sẽ báo lỗi: method add(int, int) is already defined.
    }
```

- Ta phải cho java biết cần phải gọi phương thức nào để thực hiện dựa vào sự khác nhau về số lượng đối số cũng như kiểu dữ liệu của các đối số này để phân biệt các phương thức trùng tên.

- Ta **không thể** sử dụng giá trị trả về để phân biệt sự khác nhau giữa 2 phương thức overload

- Overloading có thể xảy ra giữa lớp cha và lớp con

- Java làm gì khi không tìm thấy chính xác?

+ Java sẽ tìm kiếm phương thức gần khớp nhất theo quy tắc sau:

+ Kiểm tra phương thức trùng khớp chính xác.

+ Nếu không có, tự động ép kiểu (widening conversion) để tìm phương thức phù hợp.

- Lưu ý:

```
public class OverloadTest {
    public void print(Object obj) {
        System.out.println("Object version");    // Ngoài String ra thì ưu tiên cái này
    }
    public void print(String str) {             // Ưu tiên với tham số thực tế là String
        System.out.println("String version");
    }
    public static void main(String[] args) {
        OverloadTest test = new OverloadTest();
        String s = "Hello";
        Object o = "Hello";
        CharSequence cs = "Hello";
        test.print(s);  // String
        test.print(o);  // Object
        test.print(cs); // Object (vì không có print(CharSequence))
    }
}
```

- Chú ý: Lỗi overloading không rõ ràng

```
    void print(int x, double y) { ... }
    void print(double x, int y) { ... }
```

```
print(10, 10); // Lỗi: ambiguous - không xác định được phiên bản nào
```

Nạp đè (overriding):

- Phải có cùng chữ ký hàm: tên hàm + đối số

Điều kiện	Giải thích
✔ **Tên phương thức phải giống nhau**	Phương thức ở lớp con và lớp cha phải cùng tên.
✔ **Danh sách tham số phải giống nhau (same signature)**	Số lượng, kiểu dữ liệu, và thứ tự tham số phải giống nhau.
✔ **Kiểu trả về phải giống hoặc kiểu con (covariant return type)**	Ví dụ: nếu lớp cha trả về `Animal`, lớp con có thể trả về `Dog` nếu `Dog` kế thừa `Animal`.
✔ **Phạm vi truy cập phải bằng hoặc rộng hơn**	Không được thu hẹp phạm vi. Ví dụ: nếu lớp cha là `public` thì lớp con không được dùng `protected` hay `private`.
✔ **Lớp con không thể ném ngoại lệ rông hơn lớp cha**	Không được mở rộng các loại ngoại lệ ra ở lớp con
✔ **Không được override nếu phương thức là `final`, `static`, hoặc `private`**	Những phương thức này không cho phép ghi đè. (static: không phải thuộc về đối tượng, mà thuộc về class)
✔ **Phương thức của lớp cha không được là constructor**	Constructor không thể override.

- Lưu ý: Nguyên tắc đa hình chỉ áp dụng cho:

+ Override phương thức non-static

+ **Không áp dụng** cho biến và phương thức static (bị che khi bị override)

Tính tương thích giữa tham chiếu và đối tượng

- Một tham chiếu lớp cha có thể tham chiếu đến:
  + Đối tượng thuộc lớp cha
  + Đối tượng thuộc lớp con
- Một tham chiếu thuộc lớp con chỉ có thể tham chiếu đến đối tượng thuộc lớp con

Liên kết tĩnh vs. liên kết động

- Tính đa hình được thực hiện bởi **liên kết động** (dynamic binding)

+ Liên kết giữa lời gọi hàm và định nghĩa hàm sẽ được thực hiện **lúc thực thi chương trình** (runtime)

+ Liên kết động chỉ được áp dụng cho các phương thức bị nạp đè

VD:

```
class Animal {
    public static int count = 0;

    public Animal() {
        count++;
    }
    public void displayCount() {
        System.out.println(count);
    }
}
class Monkey extends Animal {
    public static int count = 0;
    public Monkey() {
        count++;

    public void displayCount() {
        System.out.println(count);
    }
}
class BabyDog extends Animal {
    public static int count = 0;
    public BabyDog() {
        count++;
    }
    public void displayCount() {
        System.out.println(count);
    }
}
public class Main {
    public static void main(String[] args) {
        Animal a[] = new Animal[3];
        a[0] = new Animal();
        a[1] = new Monkey();
        a[2] = new BabyDog();
        for (int i = 0; i < 3; i++) {
            a[i].displayCount();  // 3 1 1
        }
      for(int i=0; i<3; i++){

            System.out.println(a[i].count); //
3 3 3
      }

        System.out.println(Animal.count); // 3
        System.out.println(Monkey.count); // 1
        System.out.println(BabyDog.count);// 1
    }
}
```

```
class Parent {
    static void print() { System.out.println("Parent static"); }
    void show() { System.out.println("Parent instance"); }
}

class Child extends Parent {
    static void print() { System.out.println("Child static"); }
    @Override
    void show() { System.out.println("Child instance"); }
}

// Test:
Parent p = new Child();
p.print(); // "Parent static" (không đa hình)
p.show();  // "Child instance" (đa hình)
```

- Nếu không có nạp đè thì được gọi là **liên kết tĩnh**

+ Được thực hiện lúc biên dịch

+ Áp dụng cho cả thuộc tính:

+ Được sử dụng cho:

- Phương thức static:
- Thuộc tính static và không static: việc truy xuất thuộc tính luôn **được xác định** theo kiểu của **biến tham chiếu** tại thời điểm biên dịch
- Phương thức final
- Phương thức private
- Nạp chồng phương thức

VD:

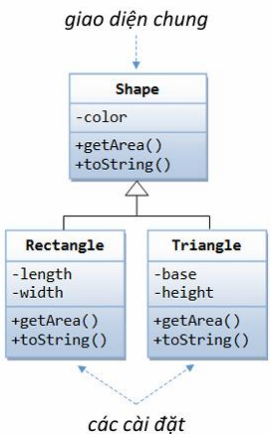
```
class Animal {
    public static int count = 0;
    public Animal() {
        count++;
    }
}
class Monkey extends Animal {
    public static int count = 0;
    public Monkey() {
        count++;
    }
}
class BabyDog extends Animal {
    public static int count = 0;
    public BabyDog() {
        count++;
    }
}
public class Main {
    public static void
main(String[] args) {
        Animal a[] = new
Animal[3];
        a[0] = new Animal();
        a[1] = new Monkey();
        a[2] = new BabyDog();
        for (int i = 0; i < 3;
i++) {
            System.out.println(a[i].
count);  // 3 3 3 Do xác định theo
Animal
        }
    }
}
```

Lưu ý: Một tham chiếu kiểu lớp cha: **Chỉ có thể** truy xuất đến các **thành phần** của lớp cha

VD:

```
class Animal {
    public void speak() {
        System.out.println("Animal speaks");
    }
}
class Dog extends Animal {
    public void bark() {
        System.out.println("Dog barks");
    }
    @Override
    public void speak() {
```

```
        System.out.println("Dog speaks");
    }
}
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog(); // Tham chiếu kiểu lớp cha
        a.speak();  // Gọi được vì có trong lớp cha (và sẽ gọi đúng
phiên bản Dog do override)
        // a.bark(); // LỖI: Không thể gọi vì phương thức bark không có
trong lớp cha
    }
}
```



Ứng dụng thực tế của đa hình

- Cho phép nhiều người lập trình cùng tham gia vào giải quyết một vấn đề phức tạp dựa trên một "giao diện" đã định nghĩa sẵn.

- Cho phép quản lý các đối tượng trong chương trình một cách hiệu quả hơn.

4. Lớp trừu tượng & Phương thức trừu tượng

Khái niệm lớp abstract

- Là một dạng lớp đặc biệt, có thể chứa các phương thức trừu tượng (chỉ có phần khai báo) và cả phương thức đã cài đặt (bình thường).

- Khai báo: [public/default] abstract class <tên lớp> {…}

- Không thể nào là private/protected (không thể đè được) (bị **lỗi biên dịch) (giống như lớp thường)**

- Abstract **không thể** tạo ra đối tượng, nhưng có thể tạo biến tham chiếu

- Thường sử dụng trong thừa kế

- Một lớp trừu tượng có **thêm** từ khóa **abstract**

- Lớp trừu tượng có thể kế thừa **lớp bình thường, lớp abstract, hoặc triển khai interface.**

- **Lớp abstract có thể có:** thuộc tính, phương thức bình thường, phương thức abstract, constructor

- Lớp abstract có thể triển khai các phương thức từ interface mà **không cần override tất cả**. (lớp con sẽ tiếp tục thực hiện nốt)

- Một lớp trừu tượng **có thể** không có phương thức trừu tượng

- Lớp kế thừa lớp trừu tượng phải override tất cả phương thức abstract (nếu không thì lớp đó phải là lớp abstract)

- Vẫn có thể có hàm main trong lớp abstract

Phương thức trừu tượng

- Cú pháp: [public/protected/default] abstract <kiểu dữ liệu trả về> <tên phương thức> ([<các tham số>]);

- Nếu có phương thức abstract thì lớp đó phải là abstract.

- Lớp kế thừa từ lớp trừu tượng thì:

+ Ghi đè tất cả các phương thức abstract của lớp cha

+ Lớp đó phải là lớp abstract

- Các phương thức trừu tượng **bắt buộc phải** được tái định nghĩa lại trong các **lớp cụ thể** kế thừa lớp trừu tượng đó

- Lớp abstract **không bắt buộc** phải có phương thức abstract.

- Phương thức abstract **không thể có thân.**

- Do thành phần static không phải của đối tượng nên **đương nhiên** không thể **abstract static** được

- Một phương thức trừu tượng **không thể** được khai báo **final (**không thể được nạp đè**) hay private (**không thấy bởi lớp con nên không đè được**)**

- Có thể ném ngoại lệ được

<đối tượng> instanceof <Tên lớp hoặc interface>

// Kiểm tra xem đối tượng bên trái có phải là (hoặc là con của) lớp hoặc interface bên phải hay không.

Vai trò trong thiết kế hệ thống

5. Packages và Giao diện (Interfaces)

Packages

- Là một nhóm các lớp có liên quan với nhau

- Mặc định là default package

- Bắt buộc khai báo ở đầu file

- Phải khai báo package: package pkname;

- Mỗi package nằm ở thư mục riêng

- Các file java khác package muốn liên kết thì phải khai báo:

import pkname.fileName; import pkname.*; // tất cả các lớp

Định nghĩa và đặc điểm đa thừa kế

- Là lớp con thừa kế **nhiều lớp cha (đa thừa kế cài đặt)**

- Java **không hỗ trợ đa thừa kế:**

+ Tránh xung đột thuộc tính của lớp cha (diamond problem)

+ Tính đơn giản của ngôn ngữ

- Tuy nhiên có **đa thừa kế kiểu**

+ Cú pháp: public class <classname> **implements** <interface names>

+ Vừa thừa kế, vừa interface: public class <classname> extends <superclass> **implements** <interface names>

Khai báo và cài đặt giao diện

- Lớp interface **không được** cài đặt ở phạm vi truy cập ở phạm vi truy cập public/nomodifier (tương tự như class và abstract class)

- Được xem là một lớp hoàn toàn ảo (full abstract), **tất cả các phương thức** đều **không được cài đặt (mặc định tất cả là public, không được thay đổi)**

+ Khai báo của phương thức và truy xuất (nếu không có **default** mà có thân hàm thì sẽ bị lỗi) => lớp con override phải khai báo là **public** (do interface pt luôn public)

+ Hoặc các **hằng số tĩnh/phương thức tĩnh** (public static final) (**có thân hàm như thường**)

+ Có thể khai báo chỉ bằng: kieudulieu temp = gia_tri; (không cần ghi public static final) như hàm của nó (ghi kieudulieu temp; **thì lại sai nha**)

- Đóng vai trò như một "cam kết", có thể làm được gì

- Qui ước đặt tên: -able

- Không thể tạo đối tượng là giao diện, **nhưng** có thể tạo **tham chiếu** thuộc kiểu giao diện: Animal myAnimal = new Dog(); // Animal là interface

Những thành phần KHÔNG ĐƯỢC trong Interface

Loại thành phần	Lý do	Ví dụ sai
Instance variables	Chỉ cho phép constants	`int count;`
Constructors	Không thể khởi tạo interface	`Vehicle() {}`

Loại thành phần	Lý do	Ví dụ sai
Instance initializer blocks	Liên quan đến khởi tạo đối tượng	`{ System.out.println("Init"); }`
Non-public abstract methods	Mặc định là public	`protected abstract void internal();`
Final methods	Mâu thuẫn với bản chất interface (**bị lỗi biên dịch**)	`final void doTask();`
Synchronized methods	Không phù hợp với mục đích interface	`synchronized void process();`

Ví dụ minh hoạ (`Comparable`)

```
public class PhanSo implements Comparable {
    private int tu, mau; // numerator and denominator
    public int compareTo(Object o) {
        return Float.compare(giatrituc(), ((PhanSo)o).giatrituc());
    …
    }
}
public class TestMovable {
    public static void main(String[] args) {
    …
        Arrays.sort(ds); // Sắp xếp mảng theo compareTo()
        for (int i = 0; i < 5; i++)
            ds[i].hienthi();
    }
}
```

Chú ý:

```
PhanSo[] ds = new PhanSo[n]; // Khởi tạo mảng đối tượng
Arrays.sort(ds); // Sắp xếp mảng đối tượng

ArrayList<PhanSo> dsList = new ArrayList<>(); // Khởi tạo danh sách đối tượng
Collection.sort(dsList); // Sắp xếp danh sách đối tượng
```

Phương thức mặc định (Java 8+)

- Từ java 8, giao diện có thể có các **phương thức mặc định**

- Là phương thức có phần triển khai **(có thân hàm)** được định nghĩa ngay bên trong interface, và được khai báo với từ khóa **default**.

+ Nếu không cài đặt default: bắt buộc phải thừa kế phương thức mặc định

+ Nếu cài đặt: không cần thiết phải nạp đè phương thức mặc định của giao diện

```
public interface MyInterface {
    // Phương thức abstract (bắt buộc class con override)
    void methodA();
    // Phương thức mặc định (không bắt buộc override)
    default void methodB() {
        System.out.println("This is a default method in
interface.");
    }
}
```

So sánh interface và abstract class

Tiêu chí	Abstract Class	Interface
Từ khóa	`abstract class`	`interface`
Kế thừa	`extends`	`implements`
Kế thừa đa cấp	❌ Chỉ kế thừa **1 abstract class**	✅ Có thể implement **nhiều interface**
Có thể có thuộc tính (biến thường)?	✅ Có (cả biến thường và hằng)	❌ Chỉ có **hằng số** (`public static final`)
Phương thức có thân (có code sẵn)?	✅ Có thể	✅ Từ Java 8 trở đi với `default method`
Có thể có constructor?	✅ Có	❌ Không
Phạm vi truy cập	Có thể là `public`, `protected`, `private`	Tất cả phương thức là **public** (mặc định)
Mục đích chính	Dùng khi các class con **có chung thuộc tính và hành vi**	Dùng khi các class con **cam kết thực hiện các hành vi giống nhau**

Một số lưu ý:

Trường hợp	Ưu tiên	Ghi chú
Cả class cha và interface đều có phương thức cùng tên	Ưu tiên **phương thức của class cha**	Class cha có thứ tự ưu tiên cao hơn interface.
Class cha, interface và class con đều có phương thức cùng tên	Ưu tiên **phương thức của class con**	Đây là cơ chế **đa hình** (polymorphism), luôn gọi theo object thực tế.
Nhiều interface cùng có default method trùng tên	Class con **bắt buộc phải override** để giải quyết xung đột (super)	Trường hợp đặc biệt khi implement nhiều interface có phương thức mặc định trùng nhau.

Trường hợp	Ưu tiên	Ghi chú
Interface có default method và class cha có phương thức cùng tên	Ưu tiên **phương thức của class cha**	Class cha luôn ưu tiên hơn default method trong interface.
Class con override phương thức của class cha	Ưu tiên **phương thức class con**	Đặc trưng của tính đa hình.
Tham chiếu kiểu cha, object kiểu con	**Phương thức được gọi theo kiểu object thực tế (class con)**	Đây là bản chất của **đa hình động** (runtime polymorphism).

```
interface Vehicle {
    void start(); // Phương thức abstract (tự động public)
}
abstract class Car implements Vehicle {
    void stop() {
        // ... body method
    }
}
```

→ Vẫn chạy bình thường (Không nhất thiết phải override ở class Car vì do là abstract)

6. Đa thừa kế trong Java

Khái niệm và lý do Java không hỗ trợ

Cách Java hỗ trợ thông qua `interface`

Tất cả các lỗi cú pháp liên quan đến OOP Java

chỉ bổ sung những thông tin chưa được liệt kê và những lưu ý khác gây ra lỗi

Dưới đây là tổng hợp các lỗi cú pháp và logic phổ biến liên quan đến OOP trong Java, **không bao gồm các lỗi cú pháp cơ bản của Java** (ví dụ: thiếu `;`, `{}`,...):

1. Lỗi về Class và Object

❌ Quên khởi tạo đối tượng trước khi sử dụng

java

```
Person p;
p.getName(); // NullPointerException
```

Fix:

java
```
Person p = new Person();
```

❌ Gọi phương thức/trường từ lớp cha không có trong lớp con

java
```
Person p = new Student();
p.study(); // Lỗi biên dịch nếu `study()` không có trong `Person`
```

Fix: Ép kiểu hoặc khai báo đúng kiểu:

java
```
((Student)p).study();
```

2. Lỗi về Kế thừa (Inheritance)

❌ Ghi đè (Override) sai phương thức

java
```
class Parent {
    void show() { System.out.println("Parent"); }
}
class Child extends Parent {
    void Show() { System.out.println("Child"); } // Viết hoa 'S' → Không ph
ải override
}
```

Fix: Đảm bảo đúng tên và tham số:

java
```
@Override
void show() { System.out.println("Child"); }
```

❌ Không gọi `super()` trong constructor lớp con

java
```
class Parent {
    Parent(int x) { }
}
```

```java
class Child extends Parent {
    Child() { } // Lỗi: No default constructor in Parent
}
```

Fix: Gọi constructor cha phù hợp:

```java
Child() { super(0); }
```

3. Lỗi về Đóng gói (Encapsulation)

❌ Truy cập trực tiếp trường `private` từ bên ngoài

```java
class Person {
    private String name;
}
Person p = new Person();
p.name = "Alice"; // Lỗi biên dịch
```

Fix: Dùng getter/setter:

```java
p.setName("Alice");
```

4. Lỗi về Đa hình (Polymorphism)

❌ Ép kiểu sai (ClassCastException)

```java
Person p = new Person();
Student s = (Student)p; // Runtime Error
```

Fix: Kiểm tra `instanceof` trước:

```java
if (p instanceof Student) {
    Student s = (Student)p;
}
```

5. Lỗi về Interface/Abstract Class

❌ Không triển khai phương thức abstract

java
```java
abstract class Animal {
    abstract void sound();
}
class Dog extends Animal { } // Lỗi biên dịch
```

Fix: Triển khai đủ phương thức abstract:

java
```java
class Dog extends Animal {
    @Override
    void sound() { System.out.println("Gau gau"); }
}
```

❌ Implement interface nhưng thiếu phương thức

java
```java
interface Runnable {
    void run();
}
class Robot implements Runnable { } // Lỗi biên dịch
```

Fix: Thêm phương thức từ interface:

java
```java
class Robot implements Runnable {
    @Override
    public void run() { }
}
```

6. Lỗi về Ghi đè `equals()`/`hashCode()`

❌ Ghi đè `equals()` nhưng không ghi đè `hashCode()`

java
```java
class Person {
    String name;
```

```
    @Override
    public boolean equals(Object obj) { ... }
    // Thiếu hashCode() → Gây lỗi khi dùng trong HashMap/HashSet
}
```

Fix: Luôn ghi đè cả hai:

java
```
@Override
public int hashCode() {
    return Objects.hash(name);
}
```

❌ So sánh đối tượng bằng `==` thay vì `equals()`

java
```
String s1 = new String("A");
String s2 = new String("A");
if (s1 == s2) { ... } // Sai logic (so sánh tham chiếu)
```

Fix: Dùng `equals()`:

java
```
if (s1.equals(s2)) { ... }
```

7. Lỗi về Static/Non-static

❌ Gọi phương thức non-static từ static context

java
```
class Test {
    void show() { }
    public static void main(String[] args) {
        show(); // Lỗi biên dịch
    }
}
```

Fix: Khởi tạo đối tượng hoặc đổi thành static:

java
```
new Test().show();
// hoặc
static void show() { }
```

8. Lỗi về Constructor

❌ Đệ quy vô hạn trong constructor

java
```java
class Test {
    Test() {
        this(); // Lỗi: Recursive constructor invocation
    }
}
```

Fix: Gọi constructor khác có tham số:

java
```java
Test() {
    this(0);
}
Test(int x) { }
```

9. Lỗi về Generic Type

❌ Dùng sai kiểu generic

java
```java
List<Person> list = new ArrayList<Student>(); // Lỗi biên dịch
```

Fix: Dùng wildcard (`? extends`):

java
```java
List<? extends Person> list = new ArrayList<Student>();
```

10. Lỗi về Ghi đè (Overriding) và Che giấu (Hiding)

❌ Nhầm lẫn giữa `@Override` và khai báo phương thức mới

java
```java
class Parent {
    static void print() { System.out.println("Parent"); }
}
class Child extends Parent {
```

```
    void print() { System.out.println("Child"); } // Không phải ghi đè (vì Parent.print() là static)
}
```

Fix:

- Để ghi đè phương thức, cha và con phải cùng là **instance method** hoặc cùng **static method**.
- Thêm `@Override` để compiler kiểm tra:

java
```
@Override
static void print() { System.out.println("Child"); } // Hiding (che giấu phương thức static)
```

11. Lỗi về Thứ tự Khởi tạo Đối tượng

❌ Khởi tạo trường `static` sau khi sử dụng

java
```
class Test {
    static int x = getValue(); // x = 0 (do getValue() chạy trước khi y được khởi tạo)
    static int y = 10;
    static int getValue() { return y; }
}
```

Fix: Đảm bảo thứ tự khởi tạo:

java
```
static int y = 10;
static int x = getValue(); // x = 10
```

12. Lỗi về `final` và Kế thừa

❌ Cố gắng ghi đè phương thức `final`

java
```
class Parent {
    final void show() { System.out.println("Parent"); }
```

```
}
class Child extends Parent {
    void show() { } // Lỗi biên dịch
}
```

Fix: Không thể ghi đè phương thức `final`.

❌ Gán lại giá trị cho trường `final`

java

```
final int x = 10;
x = 20; // Lỗi biên dịch
```

13. Lỗi về `super` và `this`

❌ Gọi `super()` hoặc `this()` không phải dòng đầu tiên trong constructor

java

```
class Parent {
    Parent(int x) { }
}
class Child extends Parent {
    Child() {
        System.out.println("Child"); // Lỗi
        super(10); // Phải là lệnh đầu tiên
    }
}
```

Fix:

java

```
Child() {
    super(10);
    System.out.println("Child");
}
```

14. Lỗi về `instanceof` và Mảng

❌ Dùng `instanceof` với kiểu không liên quan

```java
String s = "Java";
if (s instanceof Integer) { } // Lỗi biên dịch (String và Integer không có quan hệ kế thừa)
```

❌ Truy cập phần tử mảng không tồn tại

```java
int[] arr = new int[5];
System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
```

15. Lỗi về `clone()` và `Cloneable`

❌ Gọi `clone()` mà không implement `Cloneable`

```java
class Person {
    String name;
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // Runtime Error nếu Person không implements Cloneable
    }
}
```

Fix:

```java
class Person implements Cloneable {
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // Hợp lệ
    }
}
```

16. Lỗi về `Comparable` và `Comparator`

❌ Triển khai `compareTo` không đối xứng

```java
class Student implements Comparable<Student> {
```

```
    int age;
    @Override
    public int compareTo(Student s) {
        return this.age - s.age; // Nguy hiểm nếu age âm (tràn số)
    }
}
```

Fix: Dùng `Integer.compare`:

java

```
@Override
public int compareTo(Student s) {
    return Integer.compare(this.age, s.age);
}
```

17. Lỗi về `enum`

❌ So sánh `enum` bằng `==` và `equals()` không nhất quán

java

```
enum Color { RED, GREEN }
Color c1 = Color.RED;
Color c2 = Color.RED;
System.out.println(c1 == c2); // true
System.out.println(c1.equals(c2)); // true (nhưng nên dùng == để tránh Null
PointerException)
```

18. Lỗi về `try-with-resources`

❌ Quên implement `AutoCloseable`

java

```
class Resource {
    void close() { }
}
try (Resource r = new Resource()) { } // Lỗi biên dịch
```

Fix:

java

```
class Resource implements AutoCloseable {
    @Override
    public void close() { }
}
```

Tổng kết Các Lỗi OOP Phổ Biến

Nhóm Lỗi	Ví dụ Điển Hình	Nguyên Tắc Fix
Ghi đè/Hiding	Nhầm `static` và non-static	Dùng `@Override` để kiểm tra
Khởi tạo	Thứ tự khởi tạo sai	Ưu tiên khởi tạo biến `static` trước
`final`	Ghi đè phương thức `final`	Không thể ghi đè
`super`/`this`	Gọi không đúng thứ tự	Luôn đặt ở dòng đầu constructor
`instanceof`	Kiểm tra kiểu không liên quan	Chỉ dùng với lớp/interface có quan hệ
`clone()`	Quên `implements Cloneable`	Luôn kiểm tra `CloneNotSupportedException`
`Comparable`	`compareTo` gây tràn số	Dùng `Integer.compare`
`enum`	So sánh không nhất quán	Ưu tiên dùng `==`
`try-with-resources`	Quên `AutoCloseable`	Implement đúng interface

Lưu ý: Các lỗi này đều liên quan đến **thiết kế OOP**, không phải lỗi cú pháp Java cơ bản. Để tránh lỗi, luôn tuân thủ nguyên tắc SOLID và kiểm tra bằng `@Override`, `@FunctionalInterface`,...

Mục lục lưu ý:
- Nếu không có constructor nào thì mặc định sẽ gọi constructor mặc định (bị lỗi nếu như có thuộc tính là đối tượng)

- Nếu constructor không có super() thì sẽ mặc định gọi super() không tham số => gài lỗi

- Lỗi chưa implements mà đòi sài đối với đối tượng tự tạo

```
public class DiemM extends Diem {
    private String color;

    // ...

    public void nhapDiemM() {
        Scanner sc = new Scanner(...);
        nhap(); // Diem.nhap()
        color = sc.nextLine();
    }

    public void in() { // DiemM.in()
        in(); // Gọi chính nó => lỗi đệ quy vô hạn, cần super().in
    }
}
```

Nếu trong một phương thức có một biến trùng tên (có thể khác kiểu dữ liệu) thì các biến cục bộ sẽ che đi các thuộc tính

```
class A {
```

```
    public void in() {
        System.out.println("In from A (ông)");
    }
}

class B extends A {
    // KHÔNG có phương thức in()
}

class C extends B {
    @Override
    public void in() {
        System.out.println("In from C (con)");
    }

    public void callSuper() {
        super.in(); // Gọi đến lớp A vì lớp B không override in()
    }
}
```

Nếu lớp B có lớp in thì từ C không bao giờ gọi được phương thức từ lớp A nữa

Khi tạo đối tượng `Child`, constructor `Parent` được gọi. Trước khi constructor `Parent` chạy, khối khởi tạo của nó (`I_P`) chạy. Sau đó constructor `Parent` (`P`) chạy. Tiếp theo, khối khởi tạo của `Child` (`I_C`) chạy, và cuối cùng là constructor `Child` (`C`).