JAVA CORE BASIC

1. Primitive and Object data type

- Phân biệt kiểu dữ liệu nguyên thủy và kiểu dữ liệu object.

+ **Kiểu nguyên thủy (Primitive types)**

* Bao gồm: byte, short, int, long, float, double, char, boolean.
* Lưu giá trị trực tiếp trong stack (truy cập nhanh).
* Không phải object, không có phương thức đi kèm.
* Kích thước cố định (ví dụ int = 4 bytes).

+ **Kiểu object (Wrapper/Object types)**

* Là các lớp trong Java như: Integer, Long, Double, Character, Boolean, hoặc các lớp bất kỳ.
* Lưu tham chiếu (reference) đến vùng nhớ heap, không lưu giá trị trực tiếp.
* Có nhiều phương thức tiện ích (ví dụ: Integer.parseInt("123")).
* Có thể là null (primitive thì không).

- Có thể chuyển đổi giữa hai kiểu dữ liệu này không ?

**Có**: Java hỗ trợ **Boxing** và **Unboxing**.

* **Boxing**: Tự động chuyển primitive → object wrapper.
* **Unboxing**: Chuyển object wrapper → primitive.

+ **Primitive → Wrapper (Boxing thủ công)**

Dùng các phương thức **factory** có sẵn trong wrapper class:

int a = 42;

// Cách 1: dùng valueOf (tối ưu, dùng cache nếu có thể)

Integer b = Integer.valueOf(a);

// Cách 2: dùng constructor (deprecated sau Java 9)

Integer c = new Integer(a); // không khuyến khích

+ **Wrapper → Primitive (Unboxing thủ công)**

Dùng các phương thức xxxValue() của wrapper:

Integer x = Integer.valueOf(99);

// Chuyển về primitive

int i = x.intValue();

double d = x.doubleValue();

+ **Chuỗi (String) ↔ Primitive/Object**

Ngoài ra hay gặp trường hợp phải chuyển đổi qua **String**:

* String → Primitive/Object

String s = "123";

int num = Integer.parseInt(s); // về primitive int

Integer obj = Integer.valueOf(s); // về object Integer

* Primitive/Object → String

int a = 456;

String s1 = String.valueOf(a); // "456"

String s2 = Integer.toString(a); // "456"

- Có thể so sánh hai kiểu dữ liệu này với nhau không?

* Với **primitive**: so sánh bằng toán tử == → so sánh giá trị.
* Với **object (wrapper)**:
  + == → so sánh địa chỉ tham chiếu trong heap.
  + equals() → so sánh nội dung (ví dụ: Integer.equals()).

Vì vậy, khi so sánh giá trị giữa primitive và wrapper, tốt nhất dùng equals().

Integer a = 128;

Integer b = 128;

System.out.println(a == b); // false (vì khác object)

System.out.println(a.equals(b)); // true

Integer c = 100;

Integer d = 100;

System.out.println(c == d); // true (vì caching -128 đến 127)

- Giá trị khi khởi tạo biến với hai loại kiểu dữ liệu này là gì?

* **Primitive**: có **default value** (khi là biến instance trong class).
  + int/byte/short/long = 0
  + float/double = 0.0
  + char = '\u0000' (ký tự rỗng)
  + boolean = false
* **Object (wrapper)**: giá trị mặc định = null.

⚠️ Lưu ý: nếu là **local variable trong method**, Java **không gán mặc định** → phải gán trước khi dùng.

2. String

- Tìm hiểu về các đặc điểm và tính chất của String trong java

 String là **class** trong Java (nằm trong java.lang).

 String là **immutable** (bất biến):

* Một khi tạo ra, giá trị của nó **không thể thay đổi**.
* Mọi thao tác như concat, replace, substring đều tạo ra **chuỗi mới** chứ không thay đổi chuỗi cũ.

 String implements CharSequence, có thể duyệt từng ký tự.

 Dễ dàng nối chuỗi bằng + hoặc concat().

 Có hỗ trợ nhiều method tiện ích: length(), charAt(), equals(), split(), toLowerCase()…

- Có bao nhiêu cách để tạo 1 biến String

Có 2 cách chính:

**+ String literal**

String s1 = "Java";

 Chuỗi này sẽ được tạo và lưu trong **String Pool**.

 Nếu có chuỗi "Java" khác đã tồn tại trong pool, nó sẽ dùng lại, không tạo mới.

+ Sử dụng **new** keyword

String s2 = new String("Java");

Luôn tạo một object mới trong heap, **không dùng lại** từ String Pool (trừ khi gọi intern()).

- Tìm hiểu về String pool?

 Là một vùng nhớ đặc biệt trong **Heap**, nơi Java lưu trữ các chuỗi literal.

 Ưu điểm: **tiết kiệm bộ nhớ** vì các chuỗi literal trùng nhau sẽ dùng chung một object.

String a = "Hello";

String b = "Hello";

System.out.println(a == b); // true (cùng object trong pool)

String c = new String("Hello");

System.out.println(a == c); // false (object khác trong heap)

+ Có thể dùng intern() để đưa chuỗi trong heap vào pool:

String d = c.intern();

System.out.println(a == d); // true

- Làm sao để so sánh hai chuỗi trong java

Có 2 cách:

1. **Dùng ==**
   * So sánh tham chiếu (có trỏ cùng object không).
   * Không nên dùng để so sánh nội dung chuỗi.
2. **Dùng equals()**
   * So sánh **nội dung** của chuỗi.
   * Đây là cách chuẩn để so sánh chuỗi.

3. Static & final

- Thế nào là static ? Phương thức, thuộc tính khai báo bằng từ khóa static được sử dụng khi nào ? Làm thế nào để truy cập được tới phương thức, thuộc tính static

**Static là gì?**

* Gắn với **class** chứ **không** gắn với từng object.
* Có thể áp dụng cho **biến**, **phương thức**, **khối khởi tạo**, **lớp lồng nhau (static nested class)**.

**Dùng static khi nào?**

* **Tiện ích/dịch vụ không cần trạng thái của đối tượng**: Math.max, UUID.randomUUID().
* **Hằng số dùng chung**: public static final int PORT = 8080;
* **Đếm/ghi nhận dùng chung** giữa mọi instance: bộ đếm, cache tĩnh.
* **Điểm vào chương trình**: public static void main(String[] args).

**Truy cập static như thế nào?**

* Ưu tiên qua **tên lớp**: ClassName.field, ClassName.method().
* Có thể truy cập qua instance nhưng **không nên** (dễ gây nhầm).
* **Từ ngữ cảnh static** (ví dụ trong main):
  + **Không** dùng được this/super.
  + **Không** truy cập trực tiếp biến/phương thức **instance** (vì thiếu đối tượng); muốn dùng phải có **object** rồi gọi qua object đó.

- Thế nào là final ? Khai báo 1 biến final khác gì với static, biến khai báo bằng final có thể chỉnh sửa được không ? Nếu được cho ví dụ minh họa.

**final là gì?**

final có thể áp dụng cho **biến**, **phương thức**, **lớp**:

* **Biến final**:
  + Gán **một lần** (phải gán tại chỗ, trong constructor, hoặc trong initializer).
  + Nếu là **kiểu tham chiếu**: **không đổi được tham chiếu**, **nhưng** đối tượng mà nó trỏ tới **vẫn có thể thay đổi trạng thái** (nếu object mutable).
* **Phương thức final**: **không cho override** ở subclass.
* **Lớp final**: **không cho kế thừa** (ví dụ java.lang.String).

**Khác gì với static?**

* static nói về **phạm vi gắn với class** (dùng chung).
* final nói về **bất biến ở mức tham chiếu/kế thừa/ghi đè**.
* Có thể kết hợp: **static final** → **hằng số** cấp lớp (đặt tên SNAKE\_CASE).

So sánh nhanh & cách dùng chuẩn

| **Tình huống** | **Dùng gì** | **Ghi nhớ** |
| --- | --- | --- |
| Giá trị dùng chung cho mọi object | static | Truy cập qua ClassName.member |
| Hằng số | static final | Đặt tên UPPER\_SNAKE\_CASE |
| Phương thức tiện ích | static | Không dùng/không cần this |
| Cấm ghi đè | final method | Ổn định API |
| Cấm kế thừa | final class | Bảo toàn bất biến, bảo mật |
| Biến chỉ gán 1 lần | final variable | Cho phép mutable state *bên trong* object nếu object mutable |

4. OOP

- Nêu ra các tính chất quan trọng của hướng đối tượng

 **Encapsulation (Đóng gói)**

* Che giấu chi tiết bên trong, chỉ expose qua getter/setter/public method.
* Giúp bảo mật dữ liệu, dễ bảo trì.

 **Inheritance (Kế thừa)**

* Cho phép class con tái sử dụng và mở rộng từ class cha.
* Dùng extends (class), implements (interface).

 **Polymorphism (Đa hình)**

* Nhiều hình thức thể hiện cùng một hành động.
* **Compile-time** (Overloading) & **Runtime** (Overriding).

 **Abstraction (Trừu tượng)**

* Che giấu logic chi tiết, chỉ hiển thị những gì cần thiết qua abstract class hoặc interface.

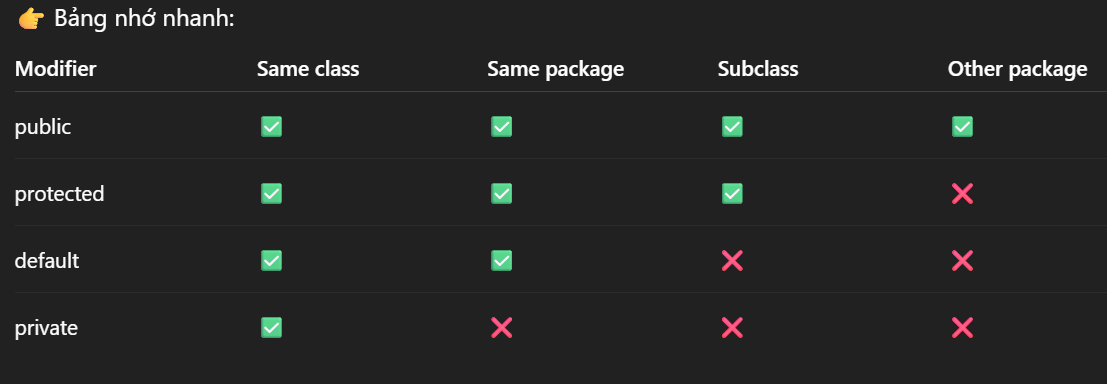
- Access modifier trong java có những loại nào ? Nêu đặc điểm của từng loại

 public: truy cập từ mọi nơi.

 protected: trong cùng package + subclass khác package.

 (default) **package-private**: không ghi gì → chỉ trong cùng package.

 private: chỉ trong chính class đó.



- Phân biệt class và instance

 **Class**: bản thiết kế (blueprint).

 **Instance (object)**: hiện thực cụ thể được tạo từ class.

- Phân biệt **Abstract**và **Interface**, Nêu trường hợp sử dụng cụ thể. Nếu 2 interface hoặc 1 abstract và 1 interface có 1 function cùng tên, có thể cùng hoặc khác kiểu trả về cùng được kế thừa bởi một class, chuyện gì sẽ xảy ra?

**+ Abstract class**:

* Có thể chứa cả **abstract methods** (chưa triển khai) và **concrete methods** (đã có code).
* Có thể có **biến instance**, **constructor**.
* Dùng extends.

**+ Interface**:

* Chỉ chứa **hợp đồng** **(contract)** (từ Java 8 có default & static methods).
* Không có constructor.
* Dùng implements.

**Khi dùng**:

* abstract class → khi có quan hệ “is-a” và muốn chia sẻ code dùng chung.
* interface → khi chỉ cần hợp đồng chung cho nhiều loại class.

**Nếu 2 interface hoặc abstract + interface có cùng method signature**:

* Không lỗi, class triển khai chỉ cần viết lại 1 lần.
* Nếu 2 interface có **default method trùng nhau** → class **phải override** để tránh ambiguity.

- Thế nào là **Overriding**và **Overloading**

 **Overloading**:

* Cùng tên method, khác tham số (số lượng hoặc kiểu).
* Xảy ra trong **compile-time**.

 **Overriding**:

* Subclass định nghĩa lại method cùng signature từ superclass.
* Xảy ra trong **runtime**.
* Yêu cầu access modifier không chặt hơn class cha.

class A {

void show(String s) { System.out.println("A: " + s); }

}

class B extends A {

@Override

void show(String s) { System.out.println("B: " + s); } // overriding

void show(int x) { System.out.println("int: " + x); } // overloading

}

- Một function có access modifier là private or static có thể overriding được không?

 private: ❌ không thể override (không thấy từ subclass).

 static: ❌ không override, mà là **method hiding**.

- Một phương thức final có thể kế thừa được không ?

final method: vẫn kế thừa xuống subclass, **nhưng không thể override**.

- Phân biệt hai từ khóa **This**và **Super**

 **this**:

* Đại diện cho object hiện tại.
* Dùng để gọi field/method trong chính class.
* Gọi constructor khác trong cùng class: this(...).

 **super**:

* Đại diện cho **class cha**.
* Gọi field/method bị override từ superclass.
* Gọi constructor cha: super(...) (phải ở dòng đầu tiên trong constructor).

5. Memory

- Thế nào là cấp phát tĩnh và cấp phát động ?

+ Cấp phát tĩnh (Static Allocation)

 Vùng nhớ được **xác định tại thời điểm compile**.

 Kích thước cố định, không thay đổi trong suốt chương trình.

 Trong Java, các biến:

* **primitive local variable** (int, double, …)
* **static variable** trong class  
  thường được cấp phát ở **stack** (local) hoặc **method area (permgen/metaspace)** (static).

 Giải phóng khi method kết thúc (đối với local) hoặc khi JVM unload class (đối với static).

**+ Cấp phát động (Dynamic Allocation)**

* Vùng nhớ được **cấp phát tại runtime** (khi chạy chương trình).
* Thường dùng cho object (class, array, collection…).
* Được lưu trong **Heap**.
* Bộ gom rác (GC – Garbage Collector) sẽ tự động thu hồi khi object không còn được tham chiếu.

- Phân biệt bộ nhớ heap và bộ nhớ stack ?

| **Tiêu chí** | **Stack** | **Heap** |
| --- | --- | --- |
| Chứa gì? | Biến cục bộ (local variables), tham chiếu object, lời gọi hàm (method frames) | Object (instance, array), dữ liệu toàn cục |
| Thời điểm | Compile + Runtime | Runtime |
| Kích thước | Nhỏ hơn | Lớn hơn |
| Quản lý | Tự động: method kết thúc → frame bị pop | Quản lý bởi Garbage Collector |
| Tốc độ | Nhanh (LIFO) | Chậm hơn (cần GC, cấp phát động) |
| Vùng nhớ | Mỗi thread có stack riêng | Heap chia sẻ cho tất cả threads |
| Lỗi thường gặp | StackOverflowError (đệ quy vô hạn) | OutOfMemoryError (Heap full) |

6. Handle Exception

- Phân biệt **throw**và **throws**

| **Từ khóa** | **Ý nghĩa** | **Vị trí dùng** | **Ví dụ** |
| --- | --- | --- | --- |
| throw | Ném ra **một exception cụ thể** (object) | Bên trong method hoặc block code | throw new IOException("File not found"); |
| throws | Khai báo rằng method **có thể ném exception** | Ở phần khai báo method | void readFile() throws IOException |

- Thế nào là **checked**và **unchecked**exception

 **Checked Exception**

* Kiểm tra tại **compile-time**.
* Bắt buộc phải try-catch hoặc khai báo throws.
* Ví dụ: IOException, SQLException, ClassNotFoundException.

 **Unchecked Exception** (Runtime Exception)

* Kiểm tra tại **runtime**.
* Không bắt buộc try-catch.
* Thường là lỗi lập trình: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

- **try catch** , **try with resource**khác nhau như thế nào ?

**+ try-catch**

* Bắt lỗi, cho phép xử lý (logging, fallback, throw tiếp).

try {

FileReader fr = new FileReader("test.txt");

} catch (IOException e) {

e.printStackTrace();

}

Vấn đề: cần tự đóng resource (fr.close()), dễ quên gây leak.

**+ try-with-resources**

* Java 7+, cho phép khai báo resource trong try.
* Resource phải implement AutoCloseable (ví dụ: FileReader, BufferedReader, Connection JDBC).
* JVM sẽ tự gọi .close() sau khi kết thúc khối try (kể cả khi có exception).

try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {

System.out.println(br.readLine());

} catch (IOException e) {

e.printStackTrace();

}

- Làm thế nào để tạo được 1 custom exception ?

Có thể tạo **Checked** hoặc **Unchecked** exception:

* **Custom Checked Exception** → extends Exception.
* **Custom Unchecked Exception** → extends RuntimeException.

// Custom Checked Exception

class InvalidAgeException extends Exception {

public InvalidAgeException(String msg) {

super(msg);

}

}

// Custom Unchecked Exception

class BusinessException extends RuntimeException {

public BusinessException(String msg) {

super(msg);

}

}

class Test {

public void register(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Tuổi phải >= 18");

}

}

}