Contents

[**1. Java Core Basic** 2](#_Toc207287674)

[**1.1. Primitive and Object data type** 2](#_Toc207287675)

[**1.2. String** 2](#_Toc207287676)

[**1.3. Static & Final** 2](#_Toc207287677)

[**1.4. OOP** 2](#_Toc207287678)

[**1.5. Memory** 2](#_Toc207287679)

[**1.6. Exception handle** 2](#_Toc207287680)

[**2. Java Collections** 2](#_Toc207287681)

[**2.1. Array** 2](#_Toc207287682)

[**2.2. List** 5](#_Toc207287683)

[**2.3. Queue** 10](#_Toc207287684)

[**2.4. Set** 12](#_Toc207287685)

[**2.5. Map** 16](#_Toc207287686)

[**3. Java Thread** 21](#_Toc207287687)

[**3.1. Synchronous vs Asynchronous** 21](#_Toc207287688)

[**3.2. Thread** 23](#_Toc207287689)

[**4. Design Patterns** 27](#_Toc207287690)

[**4.1. Tổng quan** 27](#_Toc207287691)

[**4.2. Nhận biết** 31](#_Toc207287692)

[**4.3. Đào sâu vào Signleton, Factory method** 35](#_Toc207287693)

[**5. Database** 44](#_Toc207287694)

[**5.1. Query** 44](#_Toc207287695)

[**5.2. Index** 59](#_Toc207287696)

[**5.3. Partition** 63](#_Toc207287697)

[**5.4. Stored Procedure** 65](#_Toc207287698)

# **1. Java Core Basic**

## **1.1. Primitive and Object data type**

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

**Primitive types (kiểu dữ liệu nguyên thủy)**

* Bao gồm 8 loại: byte, short, int, long, float, double, char, boolean
* **Lưu trữ** trực tiếp giá trị trong stack hoặc trong local variable slots.
* **Nhanh, hiệu năng cao**, tiêu hao bộ nhớ thấp
* **Không thể null**, luôn có giá trị hợp lệ.
* **Không thể dùng với Collections hoặc generics** (vd: không thể dùng List<int>) — vì Collections yêu cầu kiểu tham chiếu (object)

**Wrapper classes** (kiểu object - reference types)

* Mỗi primitive có lớp tương ứng: Integer, Double, Boolean, Character, v.v.
* Lưu trữ **tham chiếu** đến object nằm trên heap, chứa một giá trị primitive
* Có thể **bằng null**, có thể dùng trong Collections, generics, reflection,...
* Là **immutable objects** (giá trị không thay đổi)
* Có nhiều phương thức hữu ích: equals(), toString(), parseInt(), intValue(),...

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

**Autoboxing** và **unboxing** là cơ chế Java hỗ trợ tự động chuyển đổi tại compile-time.

**Autoboxing**: primitive → wrapper

VD  
int x = 5;

Integer y = x;

**Unboxing**: wrapper → primitive

Vd

Integer a = new Integer(10);

int b = a;

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

So sánh **Primitive** vs **primitive** dùng ==

So sánh **Wrapper** vs **Wrapper** dùng == để so sánh **tham chiếu**, không phải giá trị, dùng **.equals()** để so sánh nội dung

**Primitive** vs **wrapper**:

Khi dùng ==, wrapper sẽ được **unboxed** thành primitive rồi so sánh giá trị.

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

**Primitive types**: có giá trị mặc định

* int, byte, short, long: 0
* float: 0.0f
* double: 0.0d
* char: '\u0000' (null character)
* boolean: false

**Wrapper types / reference types**: mặc định là null

Với **Local Variables** không có giá trị mặc định, phải tạo trước khi sử dụng, nếu không sẽ bị compile-time error

## **1.2. String**

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

**String** là Lớp bất biến (**Immutable**), sau khi tạo thì nội dung không đổi. Mọi thay đổi sẽ tạo đối tượng mới. Nó là đại diện chuỗi **Unicode** (nội bộ UTF-16), final, triển khai **CharSequence** và **Comparable**<String>

**Đặc điểm quan trọng**

* **Immutable:** mọi “thay đổi” tạo đối tượng mới → an toàn luồng, có thể dùng làm key trong Map, hashCode() ổn định.
* **Unicode / UTF-16:** một số ký tự (emoji, ký tự hiếm) là **surrogate pair** (chiếm 2 char). Đếm “ký tự thực” dùng: s.codePointCount(0, s.length()). Duyệt ký tự thực: s.codePoints().
* **Nối chuỗi (+):** Với **hằng số**: gộp ở **compile-time** hoặc với biểu thức runtime: compiler dùng **StringBuilder** cho **một** biểu thức; **trong vòng lặp** bạn nên tự dùng StringBuilder.
* **Compact Strings (JDK 9+):** tối ưu bộ nhớ bên trong (không đổi API).

**Lưu ý nhanh (pitfalls)**

* Không dùng == để so sánh nội dung.
* Tránh nối chuỗi bằng + trong vòng lặp dài.
* Cẩn thận với emoji/ký tự ghép khi cắt/đếm.
* Khi chuyển byte ↔ String, **luôn chỉ định charset** (vd. UTF-8).

Phương thức

* length() -> độ dài
* charAt(int i) -> ký tự tại vị trí
* substring(int, int) -> cắt chuỗi
* indexOf(String/char), lastIndexOf(...) -> tìm vị trí
* startsWith(), endsWith(), contains() -> kiểm tra
* equals(), equalsIgnoreCase(), compareTo() -> so sánh
* toLowerCase(), toUpperCase() trim() -> bỏ khoảng trắng 2 đầu (cũ); strip()/isBlank() (các phương thức hiện đại hơn)
* replace(), replaceAll() (regex), replaceFirst()
* split(String regex) -> chia chuỗi
* format(...) / String.format(...) -> format kiểu C
* join(CharSequence, CharSequence...) -> nối từ
* collection/varargs toCharArray(), getBytes() -> chuyển đổi
* intern() -> đưa vào pool và trả tham chiếu pool
* matches(regex) -> kiểm tra regex
* lines() -> tách theo dòng (trả Stream trong Java hiện đại)
* repeat(int) -> lặp lại chuỗi

1. Có bao nhiêu cách để tạo 1 biến String

**Literal** (vào String Pool) — *khuyến nghị mặc định*

String a = "duc";

**new String(...)** (tạo đối tượng mới trên heap)

String b = new String("duc"); // cách này ít dùng

**Từ mảng ký tự**

char[] cs = {'J','a','v','a'};

String s = new String(cs);

**Từ mảng byte + charset** (chuẩn khi đọc I/O)

byte[] bytes = ...;

String s = new String(bytes, StandardCharsets.UTF\_8);

Từ **builder**

String s = new StringBuilder().append("A").append(1).toString();

Từ các **Hàm**

String s1 = String.valueOf(123);

String s2 = String.format("Hi %s", "Lan");

String s3 = String.join(", ", "a","b","c");

String s4 = "x".repeat(3);

1. Làm sao để so sánh hai chuỗi trong java

So sánh **nội dung** (đúng chuẩn)

* equals, equalsIgnoreCase
* **Thứ tự từ điển (Unicode):** compareTo, compareToIgnoreCase
* **So khớp vùng:** regionMatches(ignoreCase, toffset, other, ooffset, len)
* **Với CharSequence khác:** contentEquals (so sánh với StringBuilder, v.v.)
* **Tiền/hậu tố:** startsWith, endsWith
* **Regex:** matches, replaceAll, split (dùng khi thật cần vì tốn kém)
* **Theo ngôn ngữ (locale-aware):** dùng Collator khi cần quy tắc bản địa (tiếng Việt, v.v.)

So sánh **tham chiếu**

* **== chỉ so sánh tham chiếu** (cùng object?). Dùng để kiểm tra nhận diện, **không** dùng để so nội dung.

1. Tìm hiểu về String pool?

Là vùng lưu trữ dùng chung cho **chuỗi hằng**. Các literal giống nhau trỏ cùng một entry để **tiết kiệm bộ nhớ**.

Chuỗi **literal** và **hằng ghép tại compile-time** được đặt vào pool.

String x = "ab"; // vào pool

String y = "a" + "b"; // cũng "ab" trong pool (compile-time)

Chuỗi tạo **runtime** không tự vào pool

String z = "a" + b; // b là biến → runtime concat → object mới, không vào pool

Đưa chuỗi runtime vào pool bằng **intern()**

String r = new String("java");

String p = "java";

r == p; // false

r.intern() == p; // true (cùng entry trong pool)

## **1.3. Static & Final**

1. 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 (thuộc tính, phương thức, khối static, lớp lồng static) **thuộc về lớp** chứ **không** thuộc về từng đối tượng. Tồn tại **một bản duy nhất** (per class loader) và được khởi tạo khi lớp được **initialize**.

**Dùng khi nào?**

* **Hằng số & tiện ích**: public static final cho constant; các hàm tiện ích (như Math.sqrt).
* **Trạng thái chia sẻ**: bộ đếm, cache dùng chung (phải lo thread-safety).
* **Factory / helper**: static factory methods.
* **static nested class**: lớp lồng không cần tham chiếu tới đối tượng ngoài (hay dùng cho Builder).

**Cách truy cập:**

* Chuẩn: ClassName.member. (Gọi qua instance được nhưng **không khuyến nghị**.)
* Bên trong phương thức static **không dùng được** this/super và **không truy cập trực tiếp** thuộc tính/ phương thức **không-static**.

Vd

public class Counter {

private static int liveCount = 0; // dùng chung

public Counter() { liveCount++; } // tăng khi tạo đối tượng mới

public static int getLiveCount() { // gọi: Counter.getLiveCount()

return liveCount;

}

}

public class MathUtil {

public static int sum(int a, int b) { return a + b; } // gọi: MathUtil.sum(1,2)

}

**Static khác gì override?**

* **Phương thức static không thể override**; subclass chỉ **hides** (che khuất) phương thức static cùng chữ ký. Gọi phương thức static quyết định theo **kiểu biên dịch**, không phải runtime.

**Lưu ý/ Pitfalls với static:**

* **Biến static mutable** là tài nguyên chia sẻ → cần đồng bộ (synchronized, Lock, Atomic\*, …).
* **Rò rỉ bộ nhớ**: tham chiếu tĩnh giữ đối tượng sống “mãi”.
* **Testability**: lạm dụng static (singletons, trạng thái global) làm khó test → ưu tiên **DI** khi có thể.

1. 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 áp dụng cho **biến**, **phương thức**, **lớp** với ý nghĩa “không thay đổi ở khía cạnh nhất định”.

**final cho biến**

* **Local/field** final: **gán đúng 1 lần**.
  + Field final gán trong **khai báo** hoặc **constructor** (gọi là *blank final*).
  + static final phải gán trong khai báo hoặc **static block**.
* Với **tham chiếu object** final: **không đổi tham chiếu**, **nhưng có thể đổi trạng thái nội bộ** nếu object là mutable.

final int x = 10; // OK, không thể gán lại x = 20;

final int y; // blank final

public MyClass(int v) { y = v; } // gán đúng 1 lần trong ctor

final java.util.List<String> names = new java.util.ArrayList<>();

names.add("A"); // OK: đổi NỘI DUNG

// names = new ArrayList<>(); // LỖI: không được gán tham chiếu mới

**final cho phương thức**

* **Không thể bị override** ở subclass (nhưng có thể overload).

class A { public final void f(){} }

class B extends A { /\* void f(){} // lỗi \*/ }

**final cho lớp**

* **Không thể bị kế thừa** (vd. String là final).

**final vs static (khác nhau gì?)**

* static: **phạm vi** là lớp (dùng chung 1 bản).
* final: **ràng buộc gán/ghi đè/kế thừa** (tùy ngữ cảnh).
* Kết hợp phổ biến **public static final** cho **hằng số**:

public static final double PI = 3.141592653589793;

public static final String APP\_NAME = "MyApp";

| **Tiêu chí** | **static** | **final** |
| --- | --- | --- |
| Thuộc về | Lớp (class), 1 bản dùng chung | Ràng buộc “không đổi”: biến gán 1 lần; method không override; class không kế thừa |
| Truy cập | ClassName.member | Không ảnh hưởng cách truy cập |
| Mục đích chính | Chia sẻ, tiện ích, hằng số, nested class, factory | Bất biến cục bộ/tham chiếu, khóa kế thừa/ghi đè, hằng số |
| Kết hợp | Thường đi với final cho constant | Thường đi với static cho constant |

## **1.4. OOP**

**1) Các tính chất quan trọng của OOP**

* **Encapsulation (Đóng gói):** Ẩn chi tiết dữ liệu/hành vi bên trong lớp; chỉ lộ API cần thiết qua method công khai.  
  *Lợi ích:* giảm kết dính, kiểm soát bất biến (invariant), thay đổi nội bộ không ảnh hưởng bên ngoài.
* **Abstraction (Trừu tượng):** Mô tả *làm gì* thay vì *làm thế nào*; triển khai qua **abstract class** hoặc **interface**.
* **Inheritance (Kế thừa):** Tái sử dụng và mở rộng hành vi theo quan hệ *is-a*; dùng đúng chỗ, tránh “kế thừa cho có”.
* **Polymorphism (Đa hình):** Cùng lời gọi nhưng hành vi khác nhau tùy *kiểu thực thi*; thực hiện qua **overriding** (ghi đè).

Quy tắc: Ưu tiên **composition** (*has-a*) hơn inheritance nếu không thực sự cần đa hình.

**2) Access modifier trong Java**

| **Mức** | **Truy cập được từ đâu** | **Ghi chú nhanh** |
| --- | --- | --- |
| public | Mọi nơi | Dùng cho API công khai |
| *(default)* = package-private | Trong **cùng package** | Top-level class có thể là public hoặc *default* |
| protected | Cùng package **và** subclass khác package | Subclass **khác package** chỉ được truy cập **qua chính nó** (không qua đối tượng “người khác”) |
| private | Chỉ **trong chính lớp** | Không thừa kế; không override |

Ví dụ khác package với protected:

// package a

public class Base { protected int v = 1; }

// package b

class Sub extends Base {

void ok() { this.v = 2; } // OK: qua chính Sub (là subclass)

void notOk(Base x){ x.v = 3; } // LỖI: khác package & không phải qua Sub

}

**3) Phân biệt Class và Instance**

* **Class:** “Bản thiết kế” — khai báo field, method, constructor, static members.
* **Instance (Object):** “Thực thể” cụ thể của class; có trạng thái riêng (các field không-static).

**4) Abstract vs Interface (khi nào dùng, va chạm chữ ký)**

**Abstract class**

* Có **state**, **constructor**, method thường + **abstract method**; cho phép mọi access modifier.
* Phù hợp khi các lớp con **chia sẻ dữ liệu/hành vi** đáng kể và bạn muốn **định khung xử lý** (Template Method).
* **Một lớp chỉ extends 1 lớp (abstract hoặc concrete).**

**Interface**

* “Hợp đồng” hành vi: **abstract**, **default**, **static** (đều mặc định public); (Java 9+) có **private** method dùng nội bộ.
* **Field trong interface là public static final** (hằng).
* Dùng khi cần **đa kế thừa kiểu** (một lớp implements nhiều interface), hoặc làm **SPI/Plugin/Strategy**.

**Khi nào dùng?**

* Cần **state chung + khung xử lý + helper protected** ⇒ **abstract class**.
* Cần **hợp đồng** cho nhiều hệ phân cấp khác nhau, muốn **đa kế thừa kiểu** ⇒ **interface**.

**Nếu 2 interface (hoặc 1 abstract + 1 interface) có cùng tên method?**

1. **Cùng chữ ký & cùng kiểu trả về (abstract/abstract):** implement **1 lần** là đủ.
2. **Cùng chữ ký nhưng kiểu trả về *covariant*** (một kiểu là *subtype* của kiểu kia): implement bằng **kiểu hẹp hơn** (subtype) để thỏa cả hai.

interface A { Number m(); }

interface B { Integer m(); } // Integer ⊂ Number

class C implements A, B { public Integer m(){ return 1; } } // OK

1. **Cùng chữ ký nhưng kiểu trả về không tương thích** ⇒ **lỗi biên dịch**; phải đổi thiết kế (tách interface, đổi tên method…).

interface A { Integer m(); }

interface B { String m(); }

class C implements A, B { /\* compile error: incompatible return types \*/ }

1. **Va chạm default methods (A.default m, B.default m):** lớp **bắt buộc override** để phân giải; có thể gọi cụ thể: A.super.m().

interface A { default void hi(){ System.out.println("A"); } }

interface B { default void hi(){ System.out.println("B"); } }

class C implements A, B { @Override public void hi(){ A.super.hi(); } }

1. **Class vs interface default:** “**Class thắng interface**”.
   * Nếu **superclass** có method cùng chữ ký (dù abstract), **default** trong interface **không cứu**; subclass **phải tự cài**.

**5) Overriding vs Overloading**

**Overriding (ghi đè)**

* Xảy ra ở **subclass**: cùng **tên + tham số**; kiểu trả về **covariant** được phép.
* Quy tắc:
  + **Không** được thu hẹp **phạm vi truy cập** (có thể mở rộng: protected → public).
  + **Checked exceptions**: chỉ được ném **ít hơn/hẹp hơn** so với method cha.
  + Là **đa hình runtime** (dynamic dispatch).
* Mẹo: luôn dùng @Override để trình biên dịch kiểm tra.

**Overloading (nạp chồng)**

* Cùng **tên** nhưng **khác danh sách tham số** trong cùng lớp (hoặc qua kế thừa).
* Phân giải **lúc compile-time** dựa trên **kiểu tham số**.
* **Return type và throws không tạo ra overload**.
* Cẩn thận với **autoboxing**, **varargs**, và **type promotion** gây mơ hồ.

**6) private / static có overriding được không?**

* **private:** *Không*. Method private **không được thừa kế**, nên “cùng chữ ký” ở subclass thực chất là **method mới**.
* **static:** *Không*. static là **hiding** (che khuất), **không** đa hình. Gọi tĩnh phân giải theo **kiểu tại biên dịch**:

class A { static void hi(){ System.out.println("A"); } }

class B extends A { static void hi(){ System.out.println("B"); } }

A a = new B();

A.hi(); // "A"

B.hi(); // "B"

a.hi(); // "A" (theo kiểu tham chiếu A), không phải overriding

**7) final có kế thừa/ghi đè được không?**

* **final method:**
  + **Kế thừa:** *Có* (nếu không private).
  + **Ghi đè:** *Không thể* override.
* **final class:** *Không thể* bị kế thừa (đồng nghĩa mọi method trong đó không thể bị override vì không có subclass).
* **final field:** gán **một lần**; với object reference thì không đổi được **tham chiếu**, nhưng **nội dung** object có thể đổi nếu nó mutable.

**8) Phân biệt this và super**

* **this**
  + Tham chiếu **đối tượng hiện tại**.
  + Dùng để phân biệt field bị che khuất (this.x), hoặc gọi **constructor cùng lớp**: this(...) (phải đứng **dòng đầu** của constructor).
* **super**
  + Truy cập **thành phần của superclass** (field/method bị override/hide).
  + Gọi **constructor cha**: super(...) (phải đứng **dòng đầu** của constructor; không thể vừa this(...) vừa super(...) trong cùng constructor).
  + Không dùng được trong **ngữ cảnh static**.

Ví dụ:

class Base {

int x = 1;

void hi() { System.out.println("Base.hi"); }

Base(int a){ System.out.println("Base("+a+")"); }

}

class Sub extends Base {

int x = 2;

Sub() {

super(10); // gọi ctor cha (dòng đầu)

// this(5); // LỖI nếu đặt cạnh super(...) trong cùng ctor

}

@Override void hi() {

System.out.println("Sub.hi, x="+this.x); // 2

super.hi(); // gọi bản ở Base

System.out.println("Base.x = "+super.x); // 1

}

}

**Bẫy thường gặp & mẹo thực hành**

* **Đừng gọi method có thể bị override trong constructor** (dễ chạm trạng thái “nửa khởi tạo”).
* **equals()/hashCode()**: nếu override equals() thì **phải** override hashCode() để dùng an toàn với HashMap/HashSet.
* **Field hiding ≠ overriding**: truy cập field phân giải theo **kiểu compile-time** (không đa hình).
* **Top-level class** chỉ có thể là public hoặc *default* (package-private).
* Dùng @Override cho mọi ghi đè để tránh lỗi chính tả/khác chữ ký.

**Tóm tắt “nhớ nhanh”**

* **Cần state chung + khung xử lý** → **abstract class**.
* **Cần hợp đồng/mix-in đa hệ phân cấp** → **interface** (+ default khi cần).
* **Overriding** = runtime, cùng chữ ký; **Overloading** = compile-time, khác tham số.
* private/static **không override**; final method **không** override nhưng **có thể** kế thừa; final class **không** cho kế thừa.
* this = bản thân đối tượng; super = phần cha; super(...)/**this(...)** phải ở **dòng đầu** constructor (và chỉ 1 cái).

## **1.5. Memory**

**1) Cấp phát tĩnh (static allocation) vs cấp phát động (dynamic allocation)**

**Cấp phát tĩnh là gì?**

* **Định nghĩa chung:** kích thước và *vòng đời bộ nhớ* được quyết định **trước khi chương trình chạy** (thường ở compile-time hoặc khi lớp được nạp).
* **Trong Java:**
  + **Trường static** của lớp được cấp phát một lần khi **class được initialize** (tồn tại theo vòng đời class loader).
  + **Hằng số compile-time** (ví dụ public static final int X = 10;) có thể được **inline** vào bytecode.
  + **String literal** được đặt vào **String Pool** (tạo khi lớp chứa literal được nạp).
  + **Metadata của lớp** (bảng phương thức, constant pool, …) nằm trong vùng **Metaspace** (quản lý riêng, không phải Java heap ứng dụng).
* **Đặc trưng:** không cần new, số lượng/ kích thước đã “chốt”; vòng đời dài; thuận tiện cho **hằng số** và **trạng thái dùng chung**.

**Cấp phát động là gì?**

* **Định nghĩa chung:** bộ nhớ được xin **khi chương trình đang chạy** theo nhu cầu thực tế.
* **Trong Java:** mọi **đối tượng/ mảng** tạo bằng new → cấp phát **trên heap** (JIT có thể tối ưu đặc biệt, nhưng về mặt mô hình bạn coi như nằm ở heap). Bộ nhớ được **GC** thu hồi khi không còn tham chiếu.
* **Đặc trưng:** linh hoạt, kích thước/ số lượng quyết định lúc chạy; vòng đời phụ thuộc tham chiếu; chi phí quản lý cao hơn cấp phát tĩnh.

Lưu ý: ngoài “tĩnh/động”, còn có “**tự động**/stack” (automatic allocation) cho **biến cục bộ** và **khung ngăn xếp** (stack frame) của lời gọi hàm. Đây **không phải** “tĩnh”; nó được cấp phát/thu hồi **tự động** khi vào/ra phương thức.

**2) Phân biệt bộ nhớ heap và stack (trong Java)**

|  |  |  |
| --- | --- | --- |
| **Tiêu chí** | **Stack** | **Heap** |
| Mục đích | Lưu **stack frame** cho từng lời gọi method: tham số, biến cục bộ, địa chỉ trả về | Lưu **đối tượng** và **mảng** được tạo bằng new (kể cả các field primitive bên trong đối tượng) |
| Phạm vi | **Mỗi thread có một stack riêng** → không chia sẻ trực tiếp | **Chia sẻ giữa các thread** |
| Vòng đời | Tạo khi **vào** method, hủy khi **ra** method (khung hiện hành) | Tồn tại cho đến khi **không còn tham chiếu** → **GC thu hồi** |
| Tốc độ | Rất nhanh (push/pop khung) | Chậm hơn; quản lý bởi GC |
| Lỗi điển hình | StackOverflowError (đệ quy sâu, frame quá lớn) | OutOfMemoryError: Java heap space (rò rỉ tham chiếu, đối tượng quá nhiều) |
| Nội dung điển hình | Tham chiếu tới object, biến cục bộ primitive, địa chỉ trả về | Bản thân **object/array**, bao gồm **field primitive** bên trong object |
| Thread-safety | An toàn tự nhiên (mỗi thread một stack) | Phải đồng bộ khi nhiều thread cùng truy cập |
| Phân mảnh | Không đáng kể (mô hình LIFO) | Có thể phân mảnh; GC xử lý (copy/compact tùy thuật toán) |

Lưu ý

* Biến cục bộ *primitive* nằm trong **stack frame**; **field primitive** bên trong **object** thì nằm **trên heap** (vì nó là một phần của object).
* Coi **trường static thuộc “bộ nhớ cấp phát tĩnh” theo vòng đời lớp**, nhưng chúng **không nằm trong stack** và **không phụ thuộc từng instance**.

## **1.6. Exception handle**

**1) Phân biệt throw và throws**

|  |  |  |  |
| --- | --- | --- | --- |
| **Từ khóa** | **Dùng ở đâu** | **Ý nghĩa** | **Ví dụ** |
| throw | **Trong thân** method/khối lệnh | **Thực sự ném** một đối tượng exception (đã được tạo) | if (x < 0) throw new IllegalArgumentException("x<0"); |
| throws | **Trong khai báo** method/constructor | **Khai báo** rằng method **có thể ném** (propagate) những checked exception nào để caller biết mà xử lý hoặc tiếp tục khai báo | public void read() throws IOException { ... } |

Lưu ý nhanh:

* throw chấm dứt luồng điều khiển tại điểm đó; code sau throw là *unreachable*.
* throws không ném gì cả — chỉ là **hợp đồng** ở chữ ký method.
* Có thể *rethrow* trong catch: catch (IOException e) { throw e; } (giữ stack gốc); hoặc “wrap” kèm cause: throw new RuntimeException("failed", e);

**2) Checked vs Unchecked exception**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Loại** | **Kế thừa từ** | **Đặc điểm** | **Ví dụ điển hình** | **Cách xử lý** |
| **Checked** | Exception (trừ RuntimeException) | Compiler **bắt buộc** bạn **bắt** (*catch*) **hoặc** **khai báo throws** | IOException, SQLException, ParseException | try-catch **hoặc** thêm vào throws |
| **Unchecked** | RuntimeException (và các subclass) | **Không** bắt buộc khai báo/bắt; thường là lỗi lập trình hoặc tiền điều kiện không thỏa | NullPointerException, IllegalArgumentException, IndexOutOfBoundsException, ArithmeticException | Sửa logic, validate đầu vào; chỉ catch khi thật sự có ý nghĩa khôi phục |

3) try-catch-finally vs **try-with-resources**

**a) try-catch-finally (cổ điển)**

* Dùng khi **không** có tài nguyên cần đóng tự động hoặc bạn muốn **toàn quyền** logic finally.
* Bạn **tự đóng** tài nguyên (file, socket, stream) trong finally

**b) try-with-resources (JDK 7+)**

* Dùng khi tài nguyên **implements AutoCloseable** (Closeable cũng được).
* Tài nguyên được **đóng tự động** theo **thứ tự ngược** khi thoát try, kể cả khi có exception.
* Exception trong close() sẽ được **gắn** vào exception chính dưới dạng **suppressed** (xem getSuppressed()).

**4) Tạo custom exception như thế nào?**

Quy tắc:

* **Checked** exception → extends Exception (caller **phải** bắt/khai báo).
* **Unchecked** exception → extends RuntimeException (caller **không bắt buộc** bắt/khai báo).
* Tối thiểu nên có các constructor: *(message)*, *(message, cause)*, *(cause)*.

Ví dụ 1: **Checked** exception (nghiệp vụ bắt buộc xử lý)

public class InsufficientBalanceException extends Exception {

public InsufficientBalanceException() { }

public InsufficientBalanceException(String message) { super(message); }

public InsufficientBalanceException(String message, Throwable cause) { super(message, cause); }

public InsufficientBalanceException(Throwable cause) { super(cause); }

}

public void withdraw(long amount) throws InsufficientBalanceException {

if (amount > balance) throw new InsufficientBalanceException("Not enough funds");

balance -= amount;

}

Ví dụ 2: **Unchecked** exception (lỗi tiền điều kiện/logic)

public class InvalidUserInputException extends RuntimeException {

public InvalidUserInputException(String message) { super(message); }

public InvalidUserInputException(String message, Throwable cause) { super(message, cause); }

}

public void setAge(int age) {

if (age < 0) throw new InvalidUserInputException("age must be non-negative");

this.age = age;

}

Tips khi thiết kế custom exception:

* **Tên** rõ nghĩa theo domain: OrderNotFoundException, QuotaExceededException.
* **Không lạm dụng checked**: chỉ dùng khi caller **thực sự có thể** và **nên** khôi phục.
* **Giữ nguyên cause** khi wrap: throw new MyBizException("...", e);
* Tránh “nuốt” exception (catch xong không log/không rethrow).

# **2. Java Collections**

## **2.1. Array**

**1) Array là gì? Sinh ra để làm gì?**

**Array (mảng)** là cấu trúc dữ liệu lưu **tập hợp các phần tử cùng kiểu** (int, double, String, …) trong **vùng nhớ liên tiếp**, truy cập bằng **chỉ số (index)** bắt đầu từ 0.  
**Mục đích**: cung cấp cách lưu trữ dữ liệu **cố định kích thước** với **truy cập ngẫu nhiên O(1)** cực nhanh theo vị trí, rất phù hợp cho tính toán hiệu năng cao, xử lý dãy số, bảng, ma trận…

**2) Đặc điểm & ưu điểm**

* **Bộ nhớ liên tiếp** → truy cập a[i] là hằng số **O(1)** nhờ công thức địa chỉ: base + i \* sizeof(T).
* **Cố định kích thước** (với “mảng tĩnh” như int[] trong Java/C). Thêm/bớt giữa mảng phải **dịch phần tử**.
* **Tối ưu CPU cache**: tuần tự trong RAM → **locality tốt**, dễ **vectorization** → nhanh hơn cấu trúc có con trỏ rời rạc (như linked list).
* **Đơn giản, ít overhead** (đặc biệt mảng kiểu nguyên thủy).

**Nhược điểm**

* **Kích thước cố định** (khó chèn/xóa linh hoạt).
* **Chèn/xóa giữa mảng tốn O(n)** do phải dịch phần tử.
* **Quản lý biên (index) dễ lỗi**.

**Cơ chế cốt lõi tạo ưu điểm tốc độ**: vùng nhớ **liên tục** + **địa chỉ hóa theo offset** + **cache locality** ⇒ truy cập chỉ số là hằng số, vòng lặp tuần tự rất nhanh.

**3) Dùng array để giải quyết gì trong thực tế?**

* **Xử lý tín hiệu/thời gian**: dãy mẫu, rolling window.
* **Ma trận/ảnh**: pixel, convolution, DP table.
* **Thuật toán & thi đấu lập trình**: prefix sum, two-pointers, binary search (khi đã sắp xếp).
* **Cấu trúc bậc cao**: triển khai **stack/queue vòng** (circular buffer), **heap**, **hash table buckets**.
* **Dữ liệu cố định kích thước**: top-N điểm, bảng tra cứu (lookup table), mapping ID→object.

**4) Những lỗi sai thường gặp (kèm ví dụ Java)**

Giữ code Java vì bạn hay copy chạy thẳng.

**a) Vượt chỉ số (IndexOutOfBounds)**

int[] a = {10, 20, 30};

int x = a[3]; // LỖI: chỉ số hợp lệ là 0..2

**Khắc phục**: luôn kiểm tra 0 <= i < a.length.

**b) So sánh nội dung mảng bằng equals thay vì Arrays.equals**

int[] a = {1,2}, b = {1,2};

System.out.println(a.equals(b)); // false (so sánh tham chiếu)

System.out.println(java.util.Arrays.equals(a,b)); // true (so sánh nội dung)

**c) Sao chép nông (aliasing)**

int[] a = {1,2};

int[] b = a; // b cùng tham chiếu với a

b[0] = 9; // a cũng đổi

**Đúng**:

int[] b = java.util.Arrays.copyOf(a, a.length);

**d) Mảng đối tượng vs mảng nguyên thủy**

* int[] rất gọn/nhanh.
* Integer[] có **auto-boxing** → tốn bộ nhớ, chậm hơn.

**e) Mảng đa chiều trong Java là “mảng của mảng”**

int[][] m = new int[2][];

m[0] = new int[3];

m[1] = new int[2];

// m[0] và m[1] có độ dài khác nhau (ragged). Truy cập chưa khởi tạo sẽ NPE.

**f) Tính “covariant” gây lỗi ở runtime**

Number[] arr = new Integer[3];

arr[0] = 1.5; // ArrayStoreException vì thực chất là Integer[]

**g) Quên sắp xếp trước khi binary search**

int[] a = {5,1,4};

int pos = java.util.Arrays.binarySearch(a, 4); // KHÔNG ĐẢM BẢO ĐÚNG nếu chưa sort

**h) Chèn/xóa giữa mảng không dùng System.arraycopy**

int[] a = {1,2,3,4,5};

int pos = 2; // chèn 9 vào giữa

int[] b = new int[a.length+1];

System.arraycopy(a, 0, b, 0, pos);

b[pos] = 9;

System.arraycopy(a, pos, b, pos+1, a.length - pos);

**5) Độ phức tạp các thao tác (mảng tĩnh & mảng động)**

**Ký hiệu nhanh**

* **n**: số phần tử
* **k**: số phần tử thao tác (ví dụ số phần tử copy)
* **Amortized**: trung bình dài hạn (mảng động thỉnh thoảng mới phải **resize**)

**Bảng “cheat sheet”**

| **Thao tác** | **Mảng tĩnh T[]** | **Mảng động (vd. ArrayList<T>)** | **Ghi chú** |
| --- | --- | --- | --- |
| Truy cập a[i] / Cập nhật a[i]=x | **O(1)** | **O(1)** | Nhờ địa chỉ hóa theo offset |
| Tìm tuyến tính (unsorted) | Best **O(1)**, Avg/Worst **O(n)** | Tương tự | Dừng sớm nếu thấy sớm |
| Tìm nhị phân (sorted) | Best **O(1)**, Avg/Worst **O(log n)** | Tương tự | Cần **đã sắp xếp** |
| Thêm cuối | Không thể (phải tạo mảng mới) → **O(n)** copy | **Amortized O(1)**, Worst **O(n)** khi resize | Mảng động tăng kích thước ~1.5–2x khi đầy |
| Xóa cuối | **O(n)** nếu cần tạo mảng mới | **O(1)** | ArrayList giảm size, không copy |
| Chèn/Xóa tại vị trí i | **O(n)** | **O(n)** | Dịch phần tử sau i |
| Chèn/Xóa đầu mảng | **O(n)** | **O(n)** | Dịch toàn bộ |
| Sao chép/slice k phần tử | **O(k)** | **O(k)** | System.arraycopy rất nhanh |
| Sắp xếp so sánh | **O(n log n)** | **O(n log n)** | Tuỳ thuật toán sort |
| Duyệt toàn mảng | **O(n)** | **O(n)** | Locality tốt → rất nhanh |

**Tình huống best/avg/worst** (tiêu biểu):

* **Tìm tuyến tính**: best khi phần tử nằm ở a[0] (**O(1)**), worst khi không có (**O(n)**).
* **Binary search**: best nếu trúng giữa (**O(1)**), avg/worst **O(log n)**.
* **Thêm cuối của ArrayList**: phần lớn lần thêm là **O(1)**; **worst O(n)** đúng lúc resize (copy sang mảng lớn hơn).
* **Chèn giữa**: luôn **O(n)** do phải dồn/giãn.

**7) Khi nào chọn Array vs ArrayList?**

* **Array (int[], double[], …)**:
  + Biết **kích thước cố định**, ưu tiên **tốc độ** và **bộ nhớ thấp**.
  + Xử lý dữ liệu **nguyên thủy** khối lượng lớn, pipeline hiệu năng.
* **ArrayList<T>**:
  + **Kích thước linh hoạt**, thường xuyên thêm/xóa ở cuối.
  + API tiện dụng (thêm/xóa/tìm kiếm), chấp nhận overhead và auto-boxing (nếu dùng Integer…).

**8) Ghi chú hiệu năng “đi vào việc”**

* Duyệt tuần tự một chiều **rất nhanh** nhờ cache; gom thao tác thành khối lớn (**System.arraycopy**) tốt hơn nhiều vòng lặp lẻ tẻ.
* Với dữ liệu **đã sắp xếp**, hãy **binary search** thay vì tuyến tính.
* Thao tác chèn/xóa nhiều ở giữa → cân nhắc **list liên kết**, **deque**, hoặc **cấu trúc chuyên biệt** (gap buffer, rope) tùy bài toán.
* Tránh Integer[] nếu có thể dùng int[] (tiết kiệm đáng kể bộ nhớ & GC).

## **2.2. List**

**1) List là gì? Sinh ra để làm gì?**

**List** là cấu trúc dữ liệu **tuyến tính có thứ tự**, cho phép lưu **phần tử trùng lặp**, truy cập theo **chỉ số (index)**.  
**Mục đích**: quản lý tập phần tử **kích thước linh hoạt**, **giữ thứ tự thêm vào** (insertion order) và cung cấp API thao tác tiện lợi (thêm/xóa/tìm kiếm/duyệt).

Trong Java, List là **interface**; các triển khai tiêu biểu: ArrayList, LinkedList, CopyOnWriteArrayList, (legacy: Vector). Ngoài ra còn List.of(...) (bất biến/immutable).

**2) Đặc điểm, ưu điểm & “cơ chế” đứng sau**

* **Giữ thứ tự**: phần tử có **vị trí**; get(i), set(i, v) rõ ràng.
* **Cho phép trùng lặp**: khác Set.
* **API giàu**: add, addAll, remove, removeIf, indexOf, sort, subList, v.v.
* **Linh hoạt kích thước**: thêm/xóa không cần khai báo trước (khác mảng tĩnh).

**Cơ chế & ưu điểm theo từng triển khai**

* **ArrayList** (nền tảng là **mảng động liên tiếp**):
  + **Truy cập ngẫu nhiên O(1)** nhờ công thức địa chỉ (base + i\*sizeOf).
  + **Cache locality tốt** → duyệt tuần tự rất nhanh.
  + Tăng kích thước theo **hệ số ~1.5x** khi đầy (đổi mảng + copy).
* **LinkedList** (danh sách móc nối **hai chiều**):
  + Chèn/xóa **tại vị trí đã có con trỏ** là **O(1)** (chỉ nối/unlink nút).
  + **Truy cập theo index O(n)** vì phải lần theo nút (ít lợi thế cache).
* **CopyOnWriteArrayList** (mảng copy-on-write):
  + **Đọc/duyệt rất an toàn khi đa luồng** (snapshot),
  + **Mọi ghi (add/remove/set) là O(n)** do **copy toàn bộ mảng**.
  + **Không cho phép null**.

**3) Dùng List để giải quyết gì trong thực tế?**

* **Danh sách có thứ tự**: giỏ hàng, playlist, lịch sự kiện theo thời gian thêm.
* **Tập dữ liệu thay đổi linh hoạt**: thêm/xóa nhiều ở **cuối** (ArrayList); thêm/xóa ở **đầu/cuối** (LinkedList) trong vai trò deque.
* **Duyệt & thao tác hàng loạt**: lọc (removeIf), sắp xếp (sort), transform (Stream).
* **Đa luồng đọc nhiều–ghi ít**: CopyOnWriteArrayList cho các bảng tra cứu nhỏ, listener list, cache cấu hình tải một lần.

**4) Những lỗi sai thường gặp (kèm ví dụ & cách sửa)**

**a) Dùng == thay vì equals để so sánh nội dung**

List<String> a = new ArrayList<>(List.of("A", "B"));

System.out.println(a.get(0) == "A"); // có thể false (so sánh tham chiếu)

System.out.println(a.get(0).equals("A")); // đúng: so sánh nội dung

**b) Nhầm remove(int index) với remove(Object o)**

List<Integer> lst = new ArrayList<>(List.of(1, 2, 3));

lst.remove(1); // XÓA phần tử ở index=1 -> còn [1,3]

lst.remove(Integer.valueOf(1)); // XÓA phần tử có giá trị 1 -> còn [2,3]

**c) Xóa khi đang for-each → ConcurrentModificationException**

List<Integer> a = new ArrayList<>(List.of(1,2,3,4));

// Sai:

for (Integer x : a) {

if (x % 2 == 0) a.remove(x); // có thể ném CME

}

// Đúng:

a.removeIf(x -> x % 2 == 0); // an toàn & gọn

// hoặc dùng Iterator:

for (Iterator<Integer> it = a.iterator(); it.hasNext();) {

if (it.next() % 2 == 0) it.remove();

}

**d) Dùng LinkedList cho truy cập ngẫu nhiên**

List<Integer> l = new LinkedList<>();

// Sai: truy cập theo index trong vòng lặp -> O(n^2)

for (int i = 0; i < l.size(); i++) use(l.get(i));

// Đúng: dùng Iterator

for (Integer v : l) use(v);

**e) Quên đặt initialCapacity cho ArrayList khi biết trước kích thước**

var list = new ArrayList<String>(expectedSize); // giảm số lần resize + copy

**f) Lạm dụng CopyOnWriteArrayList cho workload ghi nhiều**

* Mỗi lần ghi **copy cả mảng** → tốn CPU & bộ nhớ. Chỉ dùng khi **đọc nhiều–ghi rất ít**.

**g) Hiểu sai subList**

var base = new ArrayList<>(List.of(1,2,3,4,5));

var view = base.subList(1, 4); // [2,3,4] - là VIEW, không phải bản sao

base.add(99); // thay đổi cấu trúc base

// Tiếp tục dùng view có thể ném ConcurrentModificationException

Nếu cần bản sao độc lập: new ArrayList<>(base.subList(1,4)).

**h) Bất biến (immutable) không cho sửa**

var imm = List.of(1,2,3); // immutable, không cho null

imm.add(4); // UnsupportedOperationException

**5) Độ phức tạp thao tác – “cheat sheet” so sánh**

**Ký hiệu**

* **n**: số phần tử; **amortized**: trung bình dài hạn.
* **Best/Avg/Worst**: nơi phù hợp.

| **Thao tác** | **ArrayList** | **LinkedList** | **CopyOnWriteArrayList** | **Ghi chú** |
| --- | --- | --- | --- | --- |
| Truy cập get(i) / set(i) | **O(1)** | **O(n)** (best O(1) ở đầu/cuối qua getFirst/Last) | **O(1)** đọc | LL phải lần nút; COWAL đọc từ snapshot |
| Thêm cuối add(e) | **Amortized O(1)** (worst **O(n)** khi resize) | **O(1)** (thêm tail) | **O(n)** (copy-on-write) | AL tăng ~1.5x khi đầy |
| Xóa cuối | **O(1)** | **O(1)** | **O(n)** |  |
| Chèn/Xóa ở giữa (index i) | **O(n)** (dịch đuôi) | **O(n)** (traverse tới i, unlink O(1)) | **O(n)** |  |
| Thêm/Xóa đầu | **O(n)** | **O(1)** | **O(n)** | LL làm deque rất hợp |
| contains, indexOf | **O(n)** | **O(n)** | **O(n)** | Tuyến tính |
| Duyệt tuần tự | **O(n)** (rất nhanh nhờ cache) | **O(n)** | **O(n)** (snapshot) |  |
| Sort | **O(n log n)** | **O(n log n)** | **O(n log n)** | TimSort cho list so sánh |

**Best/Avg/Worst nổi bật**

* ArrayList.add(e) ở cuối: **best/avg amortized O(1)**; **worst O(n)** khi resize.
* LinkedList.get(i): **best O(1)** ở đầu/cuối; **avg/worst O(n)** ở giữa.
* Xóa khi duyệt:
  + ArrayList dùng Iterator.remove() vẫn phải **dịch mảng** → **O(n)** mỗi lần xóa ở giữa.
  + LinkedList dùng iterator **unlink O(1)** (sau khi đã tới nút).

**6) Chọn List nào cho bài toán nào?**

* **Thêm cuối + truy cập ngẫu nhiên nhiều** → **ArrayList**.
* **Thêm/xóa đầu/cuối thường xuyên** hoặc **xóa nhiều trong lúc duyệt** → **LinkedList** (nhưng cân nhắc ArrayDeque cho stack/queue).
* **Đọc cực nhiều, ghi rất hiếm, đa luồng** → **CopyOnWriteArrayList**.
* **Cần bất biến** → **List.of(...)** hoặc Guava ImmutableList.

**7) Mẹo hiệu năng & thực hành tốt**

* Với ArrayList, **đặt trước initialCapacity** (hoặc ensureCapacity) nếu ước lượng được kích thước.
* Duyệt **tuần tự** sẽ tận dụng **cache** tốt (đặc biệt với ArrayList).
* Dùng **removeIf/replaceAll** để thao tác hàng loạt, gọn và ít lỗi CME.
* Với LinkedList, **đừng lặp bằng get(i)**; hãy dùng **iterator/for-each**.
* Cẩn thận **auto-boxing** khi dùng List<Integer> thay vì mảng int[] (bộ nhớ & GC)

## **2.3. Queue**

**1) Queue là gì? Khi nào dùng?**

**Queue** là cấu trúc **hàng đợi** – mặc định **FIFO**: phần tử vào trước ra trước. Dùng khi:

* Xử lý tuần tự theo thứ tự đến (request, log, task…).
* Duyệt theo lớp (BFS), xử lý theo “luồng công việc”.
* Cần “đệm” giữa **producer** và **consumer** (đa luồng).

Trong Java, Queue<E> là **interface** con của Collection<E>. Hai họ chính:

* **FIFO Queue/Deque**: ArrayDeque, LinkedList, LinkedBlockingQueue, …
* **Priority Queue (heap)**: PriorityQueue, PriorityBlockingQueue (xếp ưu tiên theo comparator, **không** FIFO).

**2) Các triển khai chủ chốt**

| **Lớp** | **Kiểu** | **Đặc điểm & Use-case** | **Độ phức tạp (xấp xỉ)** |
| --- | --- | --- | --- |
| **ArrayDeque** | Deque (non-blocking) | Mảng vòng tự giãn; **rất nhanh** cho đầu/cuối; không thread-safe; **không** cho null | offer/poll/peek: O(1) amortized |
| **LinkedList** | Deque & List | Danh sách móc nối; hỗ trợ thao tác List; chậm khi truy cập ngẫu nhiên | đầu/cuối O(1), giữa O(n) |
| **PriorityQueue** | Min-heap | Luôn lấy **phần tử “nhỏ nhất”** theo comparator; **không** FIFO; không thread-safe; không null | offer: O(log n); poll/peek: O(log n)/O(1) |
| **ConcurrentLinkedQueue** | Lock-free | Hàng đợi **không chặn**, thread-safe, thông lượng cao | O(1) trung bình |
| **ArrayBlockingQueue** | Blocking (bounded) | Dung lượng **cố định**; hỗ trợ put/take chặn; công bằng optional | put/take: O(1) |
| **LinkedBlockingQueue** | Blocking (bounded/unbounded) | Mặc định unbounded; put/take chặn; phổ biến producer-consumer | O(1) |
| **PriorityBlockingQueue** | Blocking heap | Giống PriorityQueue nhưng thread-safe; **unbounded** | offer: O(log n); take: O(log n) |
| **SynchronousQueue** | Rendezvous | **Không đệm**: producer phải “bàn giao” trực tiếp cho consumer | — |
| **LinkedTransferQueue** | Transfer | Cho phép tryTransfer/transfer tối ưu trao tay | — |
| **DelayQueue** | Delay | Lấy phần tử **sau một khoảng delay** (yêu cầu Delayed) | take: chặn đến hạn |

**Deque** (double-ended queue) cho phép thao tác cả đầu **và** cuối. Trong Java, **Deque được khuyến nghị** thay cho Stack.

**3) API cốt lõi & cặp phương thức**

| **Nhóm** | **Ném Exception** | **Special Value** |
| --- | --- | --- |
| **Chèn** | add(e) | offer(e) → false nếu không thêm được |
| **Xem đầu** | element() | peek() → null nếu rỗng |
| **Lấy + xóa đầu** | remove() | poll() → null nếu rỗng |

Với **blocking queues** còn có put(e) (chặn đến khi có chỗ) và take() (chặn đến khi có phần tử), kèm phiên bản timeout: offer(e, t, unit), poll(t, unit).

**4) Độ phức tạp & hiệu năng thực chiến**

* ArrayDeque thường **nhanh nhất** cho FIFO/stack đơn luồng nhờ mảng vòng & locality tốt.
* PriorityQueue là heap: offer/poll O(log n), peek O(1); **iteration không theo thứ tự**.
* LinkedList tiện vì vừa Deque vừa List, nhưng **ít khi nhanh nhất**.
* Blocking queues có chi phí đồng bộ hóa; chọn đúng loại (bounded vs unbounded) quan trọng cho **back-pressure**.

**5) Bloking Queue**

**BlockingQueue** là hàng đợi thread-safe có **phương thức chặn**:

* put(e): chặn nếu **đầy** (với queue có giới hạn).
* take(): chặn nếu **rỗng**.

 Điều này tạo **back-pressure** tự nhiên: Producer sẽ **đợi** khi Consumer chưa xử lý kịp, tránh tràn bộ nhớ và không phải tự viết wait/notify.

 Phối hợp dừng hệ thống bằng 2 cách phổ biến:

1. **Poison pill (sentinel)**: nhét “viên thuốc độc” (đối tượng đặc biệt) vào queue; Consumer nhận được thì thoát vòng.
2. **Interrupt**: consumerThread.interrupt() → take() ném InterruptedException; Consumer dọn dẹp rồi thoát (nhớ **khôi phục cờ interrupt** nếu cần).

## **2.4. Set**

**1) Set là gì? Sinh ra để làm gì?**

**Set** là cấu trúc dữ liệu lưu **tập phần tử không trùng lặp** (mỗi giá trị xuất hiện nhiều nhất 1 lần).  
**Mục đích**:

* Khử trùng (de-duplicate) nhanh.
* **Kiểm tra thuộc tập** (contains) hiệu quả.
* Đại diện các **tập tính chất/nhãn/quyền** (tags, permissions, visited…).

Java: Set là **interface**. Thực thi thường dùng: HashSet, LinkedHashSet, TreeSet, EnumSet, CopyOnWriteArraySet, ConcurrentHashMap.newKeySet()/ConcurrentSkipListSet.

**2) Đặc điểm, ưu điểm & “cơ chế” đứng sau**

**HashSet (mặc định)**

* **Cơ chế**: **bảng băm** (hash table) → phần tử rơi vào **bucket** theo hashCode().
* **Ưu điểm**: add/contains/remove **trung bình O(1)**; khử trùng rất nhanh.
* **Java 8+**: khi một bucket quá dài có thể **chuyển sang cây đỏ–đen** → tra cứu trong bucket ~**O(log n)**, giảm thoái hoá.
* **Thứ tự duyệt**: **không đảm bảo**.

**LinkedHashSet**

* HashSet + **danh sách liên kết hai chiều** để **giữ thứ tự chèn** (insertion order).
* Hữu ích khi **vừa cần khử trùng vừa cần giữ thứ tự**.

**TreeSet**

* **Cây đỏ–đen** theo **thứ tự sắp xếp của phần tử** (natural order hoặc Comparator).
* **Ưu điểm**: first/last, **range query** (subSet, headSet, tailSet), ceiling/floor.
* **Độ phức tạp**: thao tác chính **O(log n)**, duyệt theo **thứ tự tăng**.

**EnumSet**

* Lưu các giá trị của **một enum** bằng **bitset** → **rất nhanh & tiết kiệm bộ nhớ**.
* Thao tác thường **O(1)**, duyệt theo thứ tự khai báo enum.

**Set cho đa luồng**

* **Nhiều đọc, ít ghi**: CopyOnWriteArraySet (đọc an toàn, ghi tốn kém O(n)).
* **Ghi/đọc đồng thời**: ConcurrentHashMap.newKeySet() (**avg O(1)**) hoặc ConcurrentSkipListSet (**sorted, O(log n)**).

**Cơ chế tạo sức mạnh:** bảng băm (phân phối đều hash) cho **O(1)** trung bình; cây cân bằng cho **O(log n)** + hỗ trợ **thứ tự/biên**.

**3) Dùng Set để giải quyết gì trong thực tế?**

* **Khử trùng** ID/email/sản phẩm trong giỏ, tập quyền người dùng.
* **Membership test** nhanh: chặn IP, danh sách từ khóa cấm, whitelist.
* **Đánh dấu đã thăm** trong đồ thị/DFS/BFS.
* **Tập nhãn** (tags), tính union/intersection/difference.
* **Duy trì tập có thứ tự** (TreeSet) cho bài toán **xếp lịch**, **top-k theo ngưỡng**, **range query**.

**4) Lỗi sai thường gặp (và cách tránh)**

**a) Không override equals/hashCode cho phần tử tùy biến (HashSet/LinkedHashSet)**

class User { String id; User(String id){this.id=id;} } // thiếu equals/hashCode

var s = new java.util.HashSet<User>();

s.add(new User("A"));

s.add(new User("A")); // TƯỞNG trùng, nhưng vẫn tồn tại 2 phần tử!

**Đúng**: dùng record hoặc override chuẩn:

record User(String id) {}

**b) Thay đổi trạng thái ảnh hưởng đến hashCode sau khi đã add**

class Box { int x; public int hashCode(){return x;} public boolean equals(Object o){return o instanceof Box b && b.x==x;} }

var set = new java.util.HashSet<Box>();

var b = new Box(); b.x = 1;

set.add(b);

b.x = 2; // đổi hash -> phần tử "lạc bucket"

System.out.println(set.contains(b)); // có thể false

**Tránh**: phần tử trong Set nên **bất biến** (immutable) hoặc **không đổi field dùng cho equals/hashCode**.

**c) Hiểu sai thứ tự**

* HashSet: **không có thứ tự**; **đừng** dựa vào thứ tự duyệt.
* Cần giữ thứ tự chèn → **LinkedHashSet**. Cần sắp xếp → **TreeSet**.

**d) TreeSet với Comparator không nhất quán với equals**

* Nếu compare(a,b)==0 mà !a.equals(b), TreeSet **coi là trùng** và **loại bớt**.  
  **Giải pháp**: đảm bảo comparator **phân biệt đủ** hoặc chấp nhận quy tắc “bằng nhau theo comparator thì trùng”.

**e) null**

* HashSet/LinkedHashSet: cho **một** null.
* TreeSet: thường **không cho null** (không so sánh được).
* Concurrent sets: **không cho null**.

**f) Xóa khi đang for-each → ConcurrentModificationException**

var s = new java.util.HashSet<>(java.util.List.of(1,2,3,4));

// Sai:

// for (var x : s) if (x%2==0) s.remove(x);

// Đúng:

s.removeIf(x -> x % 2 == 0);

**g) Lạm dụng CopyOnWriteArraySet cho workload ghi nhiều**

* Mỗi add/remove **copy mảng** → tốn CPU/bộ nhớ. Chỉ dùng **đọc nhiều – ghi rất ít**.

**5) Độ phức tạp thao tác – “cheat sheet”**

Ký hiệu: **n** = số phần tử.

| **Thao tác** | **HashSet** | **LinkedHashSet** | **TreeSet** | **EnumSet** | **CopyOnWriteArraySet** | **Concurrent keySet** |
| --- | --- | --- | --- | --- | --- | --- |
| add(x) | **Avg O(1)**; Worst **O(n)** (bucket xấu/rehash; tree bin ~**O(log n)**) | Như HashSet | **O(log n)** | **O(1)** | **O(n)** (copy) | **Avg O(1)** |
| contains(x) | **Avg O(1)**; Worst **O(n)** (tree bin ~**O(log n)**) | Như HashSet | **O(log n)** | **O(1)** | **O(n)** | **Avg O(1)** |
| remove(x) | **Avg O(1)** | Như HashSet | **O(log n)** | **O(1)** | **O(n)** | **Avg O(1)** |
| Duyệt | **O(n)** (không thứ tự) | **O(n)** (thứ tự chèn) | **O(n)** (có sắp xếp) | **O(n)** | **O(n)** (snapshot) | **O(n)** |
| first/last/ceiling/floor | — | — | **O(log n)** | — | — | ConcurrentSkipListSet: **O(log n)** |

**Best/Avg/Worst tiêu biểu**

* HashSet.contains: **best/avg O(1)** (băm tốt); **worst O(n)** nếu va chạm dồn; với bucket “tree hoá”: ~**O(log n)**.
* TreeSet.add/remove: luôn **O(log n)**.
* EnumSet: phần lớn **O(1)** (bit thao tác).
* CopyOnWriteArraySet: **đọc nhanh, ghi chậm O(n)**.

**6) Chọn Set nào cho bài toán nào?**

* **Khử trùng & membership nhanh, không cần thứ tự** → **HashSet**.
* **Vừa khử trùng vừa giữ thứ tự chèn** → **LinkedHashSet**.
* **Cần sắp xếp/range query** → **TreeSet**.
* **Tập giá trị enum (quyền/flags)** → **EnumSet** (siêu nhanh, gọn).
* **Đa luồng**:
  + Đọc nhiều – ghi hiếm → **CopyOnWriteArraySet**.
  + Đọc/ghi đồng thời → **ConcurrentHashMap.newKeySet()**; cần sắp xếp → **ConcurrentSkipListSet**.

**8) Mẹo hiệu năng & thực hành tốt**

* Phần tử trong Set nên **bất biến** (record, String, enum).
* Với HashSet, nếu biết trước quy mô: **khởi tạo initialCapacity** (giảm rehash).
* Cần **thứ tự** → chọn đúng loại (LinkedHashSet/TreeSet), **đừng** trông chờ vào HashSet.
* TreeSet → đảm bảo Comparator **nhất quán** với khái niệm “trùng” của bạn.
* Đa luồng → ưu tiên **concurrent set** sẵn có, **không** tự khoá thô quanh HashSet trừ khi buộc phải làm.

## **2.5. Map**

**1) Map là gì? Sinh ra để làm gì?**

**Map** lưu các cặp **(key, value)** với ràng buộc **key là duy nhất**.  
**Mục đích**: tra cứu (lookup) **theo khóa** cực nhanh, thay vì phải tìm tuyến tính trong danh sách.

Trong Java, Map là **interface**. Triển khai phổ biến: HashMap, LinkedHashMap, TreeMap, ConcurrentHashMap; ngoài ra còn EnumMap, WeakHashMap, IdentityHashMap, Map.of(...) (bất biến).

**2) Đặc điểm, ưu điểm & “cơ chế” phía sau**

**HashMap (mặc định)**

* **Cơ chế**: bảng băm (**hash table**) → **bucket** theo hash(key).
* **Ưu điểm**: get/put/containsKey/remove **trung bình O(1)**; rất nhanh, bộ API phong phú.
* **Java 8+**: khi một bucket quá dài, **chuyển sang cây đỏ–đen** (tree bin) → truy cập bucket **O(log n)**, giảm nguy cơ thoái hóa.
* **Thứ tự duyệt**: **không ổn định** (không giữ insertion order).

**LinkedHashMap**

* **HashMap + danh sách liên kết** để **giữ thứ tự chèn** (hoặc **thứ tự truy cập** nếu bật accessOrder=true).
* Phù hợp cho **LRU cache** nhỏ gọn.

**TreeMap**

* **Cây đỏ–đen** theo **thứ tự tăng dần của key** (natural order hoặc Comparator).
* **Ưu điểm**: hỗ trợ **range query** (subMap, headMap, tailMap), **ceiling/floor**…
* **Độ phức tạp**: thao tác chính **O(log n)**, thứ tự duyệt **có sắp xếp**.

**ConcurrentHashMap**

* Thiết kế cho **đa luồng**: **đọc không khóa**, ghi dùng **CAS/lock tinh** theo bucket.
* **Iterator** “weakly consistent” (không ném CME), **không cho null key/value**.
* Cung cấp **toán tử hợp nhất** an toàn (compute, merge, computeIfAbsent).

**3) Dùng Map để giải quyết gì trong thực tế?**

* **Bộ nhớ đệm (cache)/LRU**: map khóa→đối tượng tính toán đắt.
* **Đếm tần suất** (word count), **grouping** (nhóm theo khóa).
* **Từ điển, bảng tra cứu** (ID→User, mã sp→giá, mã sân bay→thông tin).
* **Chỉ mục nhanh** cho dữ liệu đã tải (ví dụ map email→profile).
* **Range query/ordered feed** → TreeMap.

**4) Những lỗi sai thường gặp (và cách tránh)**

**a) Dùng key có trạng thái biến đổi (mutable)**

record Point(int x, int y) {} // nên dùng record/immutable

// Sai: nếu key là 1 đối tượng mutable (vd. List) và bạn sửa nó sau khi put,

// hashCode() thay đổi → Map "mất" key.

**Tránh**: dùng key **bất biến** (immutable) hoặc **không đổi trạng thái** sau khi đưa vào Map.

**b) Không override equals/hashCode cho key tùy biến**

class User { String id; }

// Thiếu equals/hashCode → hai User("A") khác nhau trở thành 2 key khác nhau.

**Đúng**: dùng record User(String id) {} hoặc tự override **đúng chuẩn**.

**c) Nhầm lẫn null giữa các Map**

* HashMap, LinkedHashMap: **cho phép 1 null key**, và **nhiều null value**.
* TreeMap: **không cho null key** (so sánh cần comparator); null value **được**.
* ConcurrentHashMap: **không cho null** ở cả key lẫn value.  
  → Xác định rõ loại map để tránh NullPointerException/UnsupportedOperationException.

**d) Hiểu sai thứ tự duyệt**

* HashMap: thứ tự **không xác định**.
* LinkedHashMap: **thứ tự chèn** (mặc định) hoặc **thứ tự truy cập** (access order).
* TreeMap: **thứ tự sắp xếp** theo key.

**e) Lạm dụng HashMap trong đa luồng mà không đồng bộ**

* HashMap **không thread-safe** → có thể hỏng cấu trúc.  
  **Giải pháp**: ConcurrentHashMap hoặc Collections.synchronizedMap(...) (cả map bị khóa thô).

**f) Hiệu năng kém do resize liên tục**

* HashMap có **capacity** & **loadFactor** (mặc định 0.75). Khi vượt ngưỡng → **rehash** (sao chép tốn kém).  
  **Mẹo**: nếu biết trước kích thước, khởi tạo new HashMap<>(expectedSize \* 4/3 + 1).

**g) Nhầm giữa putIfAbsent vs computeIfAbsent**

* putIfAbsent(k,v) luôn **tính trước v**, đôi khi tốn kém.
* computeIfAbsent(k, f) chỉ **gọi hàm f** khi **chưa có key** → tiết kiệm.

**5) Độ phức tạp thao tác – “cheat sheet”**

**Ký hiệu**

* **n**: số phần tử; **amortized**: trung bình dài hạn.
* Với HashMap Java 8+: bucket có thể **treeify** → tra cứu bucket **O(log n)**.

| **Thao tác** | **HashMap** | **LinkedHashMap** | **TreeMap** | **ConcurrentHashMap** |
| --- | --- | --- | --- | --- |
| get(k) | **Avg O(1)**; **Worst O(n)** (tree bin: O(log n)) | Tương tự HM | **O(log n)** | **Avg O(1)** |
| put(k,v) | **Avg O(1)**; **Worst O(n)** (rehash/collision) | Tương tự HM | **O(log n)** | **Avg O(1)** (có đồng bộ tinh) |
| remove(k) | **Avg O(1)** | Tương tự HM | **O(log n)** | **Avg O(1)** |
| containsKey | **Avg O(1)** | **O(1)** | **O(log n)** | **O(1)** |
| Duyệt entry | **O(n)** (không thứ tự) | **O(n)** (giữ thứ tự) | **O(n)** (có sắp xếp) | **O(n)** (weakly consistent) |
| Bộ nhớ/overhead | Thấp–vừa | Vừa (thêm linked) | Vừa–cao (node cây) | Vừa (metadata đồng bộ) |

**Best/Avg/Worst ví dụ**

* HashMap.get: **best O(1)** (băm đều), **avg O(1)**, **worst O(n)** (mọi key rơi vào 1 bucket & chưa treeify) hoặc **O(log n)** trong tree bin.
* TreeMap.put/get/remove: luôn **O(log n)** (cân bằng cây).
* LinkedHashMap: tương tự HashMap + chi phí liên kết hai chiều để giữ thứ tự.
* ConcurrentHashMap: trung bình **O(1)**; ghi có thể chậm hơn HashMap do điều phối đồng thời.

**6) Chọn Map nào cho bài toán nào?**

* **Tra cứu nhanh, không cần thứ tự** → **HashMap**.
* **Cần giữ thứ tự chèn / LRU** → **LinkedHashMap** (với accessOrder=true).
* **Cần thứ tự sắp xếp / range query** → **TreeMap**.
* **Đa luồng, đọc nhiều–ghi vừa** → **ConcurrentHashMap**.
* **Key là enum** → **EnumMap** (nhanh, tiết kiệm bộ nhớ).
* **Muốn key “tự rơi” khi không còn tham chiếu** → **WeakHashMap**.
* **So sánh key theo địa chỉ (==) thay vì equals** → **IdentityHashMap** (trường hợp đặc biệt).

**7) Ví dụ gọn – đúng “đi vào việc”**

**a) Đếm tần suất (HashMap)**

var words = List.of("a","b","a","c","b","a");

var freq = new java.util.HashMap<String, Integer>(words.size()\*4/3+1);

for (var w : words) freq.merge(w, 1, Integer::sum);

System.out.println(freq); // {a=3, b=2, c=1}

**b) LRU cache 5 phần tử (LinkedHashMap)**

var lru = new java.util.LinkedHashMap<Integer, String>(16, 0.75f, true) {

protected boolean removeEldestEntry(java.util.Map.Entry<Integer, String> e) {

return size() > 5; // quá 5 thì loại phần tử ít dùng nhất

}

};

lru.put(1,"A"); lru.put(2,"B"); lru.get(1); lru.put(3,"C"); // ...

**c) Range query (TreeMap)**

var tm = new java.util.TreeMap<Integer, String>();

tm.put(10,"A"); tm.put(20,"B"); tm.put(30,"C");

System.out.println(tm.subMap(15, true, 30, true)); // {20=B, 30=C}

**d) Cache đa luồng (ConcurrentHashMap + computeIfAbsent)**

var cache = new java.util.concurrent.ConcurrentHashMap<String, byte[]>();

byte[] avatar = cache.computeIfAbsent(userId, id -> loadAvatarFromDB(id)); // atomic

**e) Lỗi điển hình: key mutable / thiếu equals-hashCode**

// Sai: key mutable

var map = new java.util.HashMap<java.util.List<Integer>, String>();

var k = new java.util.ArrayList<>(java.util.List.of(1,2));

map.put(k, "value");

k.add(3); // thay đổi hashCode -> "mất" key

System.out.println(map.get(java.util.List.of(1,2))); // null

// Đúng: dùng key bất biến

record Pair(int x, int y) {}

var good = new java.util.HashMap<Pair, String>();

good.put(new Pair(1,2), "ok");

System.out.println(good.get(new Pair(1,2))); // "ok"

**8) Mẹo hiệu năng & thực hành tốt**

* **Key nên bất biến** (record, String, enum) & có **equals/hashCode** chuẩn.
* Với HashMap, **đặt initialCapacity** nếu biết trước số phần tử để giảm **rehash**.
* Với LinkedHashMap(accessOrder=true) + removeEldestEntry → **LRU** đơn giản.
* TreeMap cho **tìm kiếm theo khoảng**; đừng cố mô phỏng trên HashMap.
* Đa luồng: ưu tiên **ConcurrentHashMap**; tránh tự khóa HashMap trừ khi thật cần.
* Tránh null trong bối cảnh đa luồng & API phức tạp; dùng Optional/sentinel nếu cần.

**Tóm tắt**

* **Map** giải quyết bài toán **tra cứu theo khóa** hiệu quả.
* **HashMap**: nhanh, **O(1)** trung bình; **LinkedHashMap**: giữ thứ tự (LRU); **TreeMap**: có **thứ tự sắp xếp, range query**; **ConcurrentHashMap**: **thread-safe** hiệu quả.
* **Chìa khóa**: **key bất biến**, equals/hashCode chuẩn, chọn **đúng loại Map** theo yêu cầu **thứ tự/đa luồng/phạm vi**; tinh chỉnh **capacity/loadFactor** khi cần.

# **3. Java Thread**

## **3.1. Synchronous vs Asynchronous**

**1) Phân biệt synchronous vs asynchronous**

**Khái niệm**

* **Synchronous (đồng bộ)**: luồng hiện tại **đợi** tác vụ hoàn tất rồi mới đi tiếp. Dòng lệnh chạy **tuần tự**.
* **Asynchronous (bất đồng bộ)**: khởi động tác vụ và **không đợi** nó xong; luồng hiện tại tiếp tục làm việc khác. Kết quả sẽ đến qua **callback**, **Future/CompletableFuture**, event, v.v.

Lưu ý: **Async ≠ song song (parallel)**. Async là mô hình “không chặn”; chạy song song hay không tùy cách triển khai (thread khác, I/O non-blocking, v.v.). Bạn có thể viết async nhưng vẫn chạy tuần tự nếu chỉ có 1 thread.

**2) Khi nào dùng? Ưu/nhược điểm của sync và async**

**Khi dùng synchronous**

* Luồng xử lý **ngắn**, đơn giản, **phụ thuộc thứ tự** (giao dịch tuần tự).
* Code chạy **CPU-bound** nhỏ lẻ (tốn ít thời gian), không đáng để tách luồng.
* Môi trường hạn chế tài nguyên, không muốn sinh nhiều thread.

**Ưu điểm**

* Dễ đọc, dễ debug, stack trace rõ ràng.
* Tránh được các bẫy đồng bộ phức tạp.

**Nhược điểm**

* **Chặn** luồng gọi → UI dễ “đơ”, server tốn **thread/blocking** (N threads ~ N request).
* Khó tận dụng đa nhân khi có nhiều tác vụ độc lập.

**Khi dùng asynchronous**

* **I/O-bound** (HTTP, DB, file, network) – thời gian chủ yếu là **chờ** I/O.
* Nhiều tác vụ **độc lập** có thể chạy song song; cần **độ trễ thấp**, **thông lượng cao**.
* Ứng dụng **UI** cần giữ giao diện mượt mà; microservice cần không chặn.

**Ưu điểm**

* Tăng **throughput** và **độ phản hồi**, tận dụng tài nguyên tốt hơn (đặc biệt với I/O).
* Ghép nhiều tác vụ song song, **giảm thời gian tổng**.

**Nhược điểm**

* Tăng độ phức tạp: callback chain, xử lý lỗi & hủy (cancellation) khó hơn.
* Dễ “**async nhưng lại chặn**” (gọi .get() trên Future ở luồng nóng → mất hết lợi ích).
* Debug/trace khó, phải quản lý pool/timeout/cancel.

**Quy tắc chọn nhanh**

* **I/O-bound** → ưu tiên **async** (ví dụ CompletableFuture, reactive, NIO).
* **CPU-bound** → cân nhắc **parallel** (ForkJoin/parallel stream) hoặc thread pool; không nhất thiết async.
* Nếu business **rất đơn giản/tuần tự** → **sync** cho gọn và an toàn.

**3) Từ khóa synchronized trong Java**

**Dùng để làm gì?**

* **Mutual exclusion (độc quyền)**: chỉ **một thread** vào vùng bảo vệ tại một thời điểm.
* **Memory visibility (nhìn thấy bộ nhớ)**: đảm bảo **happens-before** – mọi ghi của thread giữ khóa **trước khi** unlock sẽ **nhìn thấy được** bởi thread khác **sau khi** lock lại cùng monitor.

Từ Java Memory Model (JMM): monitor unlock **happens-before** monitor lock tiếp theo **trên cùng một monitor**.

**Cú pháp & phạm vi khóa**

1. **Khóa phương thức (instance)**

public synchronized void inc() { counter++; } // khóa trên this

1. **Khóa phương thức (static)**

public static synchronized void f() { /\*...\*/ } // khóa trên Class object

1. **Khóa khối (block)** – **linh hoạt & khuyến nghị**

private final Object lock = new Object();

void add(int x) {

synchronized (lock) { list.add(x); } // chỉ khóa đoạn cần thiết

}

**Tóm tắt**

* **Sync**: dễ hiểu, chặn luồng, phù hợp luồng ngắn/tuần tự.
* **Async**: không chặn, phù hợp I/O-bound & nhiều tác vụ độc lập; phức tạp hơn.
* **synchronized**: khóa **reentrant** trên monitor → **mutual exclusion + visibility**; dùng block/method; phối hợp wait/notify.
* Với yêu cầu cao cấp (timeout, fair lock, read-mostly): cân nhắc **ReentrantLock/ReadWriteLock/StampedLock** và các **concurrent collections**.

## **3.2. Thread**

**1) Process là gì? Thread là gì?**

**Process**

* **Process (tiến trình)** là một chương trình đang chạy, có **không gian địa chỉ bộ nhớ riêng**, tài nguyên riêng (handle file, socket…), và tối thiểu một luồng thực thi.
* Mỗi process được HĐH lập lịch độc lập; tiến trình này **không thể truy cập trực tiếp** bộ nhớ của tiến trình khác (trừ khi dùng cơ chế liên tiến trình IPC).

**Thread**

* **Thread (luồng)** là đơn vị thực thi nhẹ bên trong process, **chia sẻ cùng một không gian nhớ** với các thread khác của cùng process (heap, tĩnh), nhưng có **stack** riêng.
* Nhiều thread trong cùng process có thể **chạy song song** (trên nhiều core) hoặc **đan xen** (time-sliced), và **dùng chung dữ liệu** → cần đồng bộ để tránh race.

Java 21 có hai loại luồng:

* **Platform Thread**: ánh xạ 1-1 với OS thread (truyền thống).
* **Virtual Thread**: luồng nhẹ do JVM quản lý (Project Loom), **không** 1-1 với OS thread, cho phép **hàng trăm nghìn – triệu** tác vụ đồng thời I/O-bound dễ dàng.

**2) Có bao nhiêu cách để “tạo thread” trong Java? Khác nhau gì?**

**(A) Cách truyền thống (tạo *task*, chạy bằng *thread*)**

1. **extends Thread**
   * Override run() rồi new MyThread().start().
   * **Nhược**: ràng buộc kế thừa (Java chỉ có **đơn kế thừa**), tách kém “task vs thread”.
2. **implements Runnable** → truyền vào new Thread(runnable).start()
   * **Ưu**: dùng **composition**, tách “nhiệm vụ” (Runnable) khỏi “phương tiện chạy” (Thread), tái sử dụng dễ hơn.
3. **Callable<V> + Future qua ExecutorService**
   * submit(Callable) trả về Future<V> để **lấy kết quả/exception**, **timeout**, **cancel**.
   * Đây là **cách khuyến nghị** trong thực tế: **Không** tự new Thread tràn lan → dùng **thread pool**.

**(B) Bất đồng bộ hiện đại (trên nền Executor/ForkJoin)**

1. **CompletableFuture** (supplyAsync/runAsync)
   * Dễ **ghép chuỗi** (thenApply/thenCompose), xử lý lỗi (exceptionally), chạy trên **ForkJoinPool.commonPool** hoặc **Executor** bạn cung cấp.
2. **Virtual Threads** (Java 21)
   * Thread.ofVirtual().start(r) hoặc Executors.newVirtualThreadPerTaskExecutor()
   * Tạo “thread” rất rẻ cho I/O-bound; code vẫn **trông như đồng bộ** nhưng không chặn OS thread.

**Khác biệt cốt lõi**

* extends Thread vs Runnable: kế thừa **không linh hoạt**; Runnable **tách biệt** nhiệm vụ & phương tiện chạy → kết hợp với pool tốt hơn.
* Callable/Future có **giá trị trả về** và **checked exception** qua Future.get(); Runnable thì không.
* ExecutorService/pool giúp **tái sử dụng luồng**, kiểm soát **số lượng**, **hủy**, **timeout**; new Thread mỗi việc dễ gây **bùng nổ luồng**.
* CompletableFuture giúp **ghép pipeline bất đồng bộ**; **Virtual Thread** giúp **đơn giản hóa** I/O-bound theo phong cách “trông như sync”.

| * **Điểm so sánh** | **thread.run()** | **thread.start()** |
| --- | --- | --- |
| Tạo luồng mới? | **Không**. Chạy như **một hàm bình thường** trên **luồng hiện tại**. | **Có**. Yêu cầu JVM/HĐH **tạo & lập lịch một luồng mới**, rồi **gọi run() trên luồng mới**. |
| Tính đồng thời | **Không có** đồng thời/song song. | **Có thể** chạy đồng thời (tùy CPU/core & scheduler). |
| Chặn luồng gọi | **Có**: chạy xong run() mới quay lại. | **Không** (thường): trả về sớm, run() chạy trên **luồng khác**. |
| Ngoại lệ | Ném ngoại lệ **về luồng gọi** (giống hàm thường). | Ngoại lệ nằm trong **luồng mới** (không “bong” về caller). Dùng setUncaughtExceptionHandler để bắt, hoặc trả kết quả qua Future. |
| Gọi nhiều lần | Gọi run() nhiều lần **được** (chỉ là gọi hàm). | **Chỉ 1 lần**. Gọi start() lần 2 ném IllegalThreadStateException. |
| Thread.currentThread() | Trả về **luồng hiện tại** (vd: main). | Trả về **luồng mới** (vd: Thread-0). |

Tóm lại: **start() mới thực sự “chạy đa luồng”**; **run() chỉ là gọi phương thức bình thường**.

**3) Thế nào là multi-thread? Ưu/nhược điểm**

**Multi-thread**: một process có **nhiều thread** cùng lúc. Các thread có thể chạy **song song** (đa core) hoặc **đan xen**; chia sẻ tài nguyên → cần cơ chế **đồng bộ**.

**Ưu điểm**

* **Tăng throughput** và **tận dụng đa nhân** cho tác vụ **độc lập**.
* **Nâng tính phản hồi** (UI/app không bị “đơ” khi I/O dài).
* Che giấu **độ trễ I/O** (đọc/ghi mạng, DB, file).

**Nhược điểm**

* **Độ phức tạp cao**: race condition, deadlock, livelock, starvation, khó debug.
* **Overhead**: context switch, bộ nhớ (stack), đồng bộ hóa.
* **Không phải mọi bài toán** đều được lợi từ song song (đặc biệt dữ liệu nhỏ, tranh chấp khóa cao, cache-unfriendly).

**Khi nào nên**: tác vụ **độc lập**, **I/O-bound**, hoặc CPU-bound có thể chia nhỏ.  
**Khi nào tránh**: logic nhỏ lẻ, phụ thuộc thứ tự, dữ liệu chia sẻ phức tạp → chi phí đồng bộ lớn hơn lợi ích.

**4) Làm thế nào biết 1 thread/multi-thread đã hoàn thành?**

Các cách **an toàn & chuẩn**:

1. **Thread.join()**
   * Đợi **một thread** kết thúc.
2. Thread t = new Thread(r);
3. t.start();
4. t.join(); // chờ xong
5. **Future / ExecutorService**
   * Future.isDone(), Future.get() (có thể kèm **timeout**), invokeAll, invokeAny.
   * Với cả nhóm: ExecutorService.shutdown(); awaitTermination(timeout, unit).
6. **CompletableFuture**
   * fut.isDone(), fut.join(); nhóm: CompletableFuture.allOf(f1, f2, ...).
7. **Bộ đồng bộ hóa**
   * **CountDownLatch**: chờ **N** tác vụ hoàn tất.
   * **Phaser/CyclicBarrier**: đồng bộ theo **pha**.

Tránh dựa vào Thread.isAlive() hoặc Thread.activeCount() để “đoán”, vì mơ hồ và kém ổn định.

**5) Có giới hạn bao nhiêu thread trong 1 ứng dụng Java?**

* **Không có con số cố định trong Java**. Giới hạn phụ thuộc vào:
  + **HĐH** (giới hạn số luồng mỗi process, ví dụ ulimit -u trên Linux).
  + **Bộ nhớ khả dụng** và **kích thước stack mỗi thread** (-Xss, mặc định thường ~1MB trên HotSpot, tùy platform).
  + Overhead JVM/GC, thư viện native…

**Quy tắc thực tế**

* Nhiều ứng dụng **đụng trần** ở vài nghìn – vài chục nghìn **platform thread** trước khi bị:
  + java.lang.OutOfMemoryError: unable to create native thread
  + Hoặc **thrash** mạnh do context switch.
* **Giảm -Xss** (ví dụ 256k) có thể tăng số thread tối đa, nhưng **cân nhắc tràn stack** nếu call-stack sâu.
* **TỐT NHẤT**: dùng **ExecutorService** (pool) để **giới hạn** số thread; không new Thread bừa bãi.
* Với I/O-bound, cân nhắc **Virtual Threads**: số lượng “thread” có thể lên **hàng trăm nghìn** vì không 1-1 với OS thread (nhưng vẫn bị giới hạn bởi CPU, heap, scheduler).

**Ước lượng thô** (platform thread):  
Số thread tối đa ≈ *Bộ nhớ trống* / *stack mỗi thread* (cộng overhead). Ví dụ 2GB trống, stack 1MB → lý thuyết ~2000, thực tế **ít hơn**.

**Phụ lục ngắn: Đồng bộ cơ bản**

* Dùng synchronized hoặc java.util.concurrent.locks.\* để đảm bảo **mutual exclusion** & **visibility**.
* Dùng **concurrent collections** (ConcurrentHashMap, BlockingQueue…) thay vì tự wait/notify.
* Đừng khóa trên đối tượng lộ ra ngoài; dùng **monitor riêng** (private final Object lock = new Object();).

# **4. Design Patterns**

## **4.1. Tổng quan**

**1) Nhóm & vai trò**

* **Creational (Khởi tạo):** Kiểm soát cách tạo đối tượng để giảm phụ thuộc và độ phức tạp khi khởi tạo (Single Responsibility).
* **Structural (Cấu trúc):** Tổ chức quan hệ giữa lớp/đối tượng để dễ mở rộng, tái sử dụng.
* **Behavioral (Hành vi):** Điều phối luồng xử lý, trao đổi thông điệp, và thay đổi thuật toán linh hoạt.

**2) Bảng nhanh: mô hình → đặc trưng → khi dùng → vì sao dùng → ví dụ thực tế**

| **Pattern** | **Đặc trưng** | **Khi dùng** | **Vì sao nên dùng** | **Ví dụ trong Java/Spring** |
| --- | --- | --- | --- | --- |
| **Singleton** | Một thể hiện duy nhất, truy cập toàn cục (có rủi ro đồng bộ) | Tài nguyên dùng chung (cache, registry…) | Giảm chi phí tạo lặp | Spring @Bean mặc định scope singleton |
| **Factory Method** | Uỷ quyền việc tạo đối tượng cho subclass | Cần chọn lớp con theo tham số/ngữ cảnh | Giảm phụ thuộc new, dễ mở rộng | Spring FactoryBean, JpaRepositoryFactory |
| **Abstract Factory** | Tạo **họ** đối tượng liên quan | Cần chuyển đổi “bộ” triển khai (MySQL vs Postgres) | Nhất quán bộ API, đổi nền tảng dễ | DataSource + EntityManagerFactory cấu hình theo profile |
| **Builder** | Tách bước dựng khỏi biểu diễn; fluent API | Đối tượng nhiều tham số tùy chọn | Code rõ ràng, bất biến | Lombok @Builder, HttpClient.newBuilder() |
| **Prototype** | Clone mẫu thay vì new | Tạo nhanh nhiều bản tương tự | Tiết kiệm chi phí khởi tạo | Object#clone (ít dùng, cân nhắc bất biến) |
| **Adapter** | “Chuyển đổi ổ cắm” giữa 2 API | Tích hợp hệ cũ/SDK khác | Tái dùng code cũ, cô lập thay đổi | UserDetailsService bọc domain User |
| **Facade** | Cửa sổ đơn giản trước hệ thống phức tạp | Expose API gọn cho module khác/UI | Giảm coupling, ẩn chi tiết | Service lớp “Application Facade” |
| **Decorator** | Bọc để thêm tính năng động | Thêm cross-cutting (logging, caching) | Mở rộng mà không sửa lõi | Java I/O BufferedInputStream, OncePerRequestFilter |
| **Composite** | Cấu trúc cây, thao tác như 1 | Dữ liệu phân cấp (menu, category) | Đơn giản hóa xử lý cây | Cây danh mục sản phẩm |
| **Proxy** | Đại diện kiểm soát truy cập | Lazy load, AOP, security | Thêm hành vi trước/sau | Spring AOP Proxy, @Transactional |
| **Flyweight** | Chia sẻ trạng thái bất biến (pool) | Rất nhiều đối tượng giống nhau | Giảm RAM | Pool Integer, String interning |
| **Strategy** | Hoán đổi thuật toán lúc chạy | Quy tắc tính phí/giảm giá | Mở rộng không đụng code cũ | PaymentStrategy, PasswordEncoder |
| **Template Method** | Khung quy trình; step override | Các bước cố định + biến thể | Tái dùng khung, ít lặp | JdbcTemplate, RestTemplate hook |
| **Observer** | Publish/Subscribe sự kiện | Xử lý sự kiện domain/UI | Tách nguồn – người nghe | Spring ApplicationEventPublisher |
| **Command** | Đóng gói yêu cầu thành đối tượng | Hàng đợi lệnh, undo/redo | Log/Retry/Queue dễ | Job object gửi qua Queue |
| **State** | Trạng thái → hành vi | Quy trình đơn hàng, ticket | Tránh if-else khổng lồ | OrderStatus định nghĩa hành vi |
| **Chain of Responsibility** | Chuỗi handler tuần tự | Pipeline xử lý request | Thay đổi/ghép bước linh hoạt | Servlet Filter, Spring Security filter chain |
| **Mediator** | Trung gian điều phối | Nhiều đối tượng tương tác chéo | Giảm phụ thuộc lẫn nhau | Event bus nội bộ |
| **Iterator** | Duyệt tập hợp ẩn cấu trúc | Bộ sưu tập tùy chỉnh | Tách duyệt khỏi cấu trúc | Iterator, Stream |

**3) Tại sao nên dùng Design Patterns?**

* **Giảm Coupling – tăng Cohesion:** mô-đun hoá, dễ bảo trì.
* **Mở rộng không sửa mã cũ (Open/Closed):** thêm tính năng bằng lớp mới.
* **Tái sử dụng & chuẩn hóa cách giải:** team dễ đọc/trao đổi hơn.
* **Testability:** mock/spy, hoán đổi chiến lược, bọc proxy để test.
* **Hiệu năng/bộ nhớ (đúng lúc):** Flyweight, Pooling.

Lưu ý: dùng đúng chỗ; over-engineering sẽ phản tác dụng.

**4) Dấu hiệu chọn pattern**

* Nhiều new rải rác, đổi loại theo cấu hình → **Factory/Abstract Factory**.
* Nhiều tham số tùy chọn, constructor dài → **Builder**.
* Cần gắn thêm tính năng logging/cache/rate-limit mà không sửa lõi → **Decorator/Proxy**.
* Luật/thuật toán thay đổi theo ngữ cảnh (giảm giá, vận chuyển) → **Strategy**.
* Pipeline xử lý request/validation nhiều bước → **Chain of Responsibility**.
* Quy trình có khung cố định, cho phép override một vài bước → **Template Method**.
* Trạng thái nghiệp vụ chi phối hành vi (NEW/PAID/CANCELLED) → **State**.
* Nhiều thành phần cần nghe sự kiện domain → **Observer**.
* API bên thứ ba không hợp giao diện domain → **Adapter**.
* Hệ thống phức tạp cần “điểm vào” gọn → **Facade**.

**5) Gắn với Spring Boot & dự án thực tế**

* **IoC/DI** của Spring giúp hiện thực hầu hết pattern hành vi/cấu trúc một cách tự nhiên (Strategy, Observer, Proxy).
* **AOP/Proxy**: @Transactional, @Cacheable, @Retryable thực chất là proxy + advice.
* **Template Method**: JdbcTemplate, RestTemplate, WebClient pattern hóa khung xử lý I/O.
* **Repository (DDD)**: tách persistence khỏi domain (gần với Facade/Adapter cho tầng dữ liệu).
* **Configuration Properties + Factory**: chuyển đổi bean theo profile (dev/test/prod).

**6) Sai lầm thường gặp**

* **Lạm dụng pattern**: thêm lớp/trừu tượng không cần thiết.
* **Singleton sai**: thread-unsafe, che giấu state toàn cục khó test.
* **Nhầm Decorator với Inheritance**: nên ưa composition.
* **Adapter biến thành God Class**: hãy giữ adapter mỏng, một trách nhiệm.

## **4.2. Nhận biết**

**Cách soi tổng quát trước khi vào từng pattern**

* **Tìm interface có ≥2 implementation** → nghi ngờ **Strategy/State/Command/Adapter/Decorator**.
* **Tìm class có field cùng kiểu với chính interface nó implement** → thường là **Decorator**.
* **Tìm chuỗi các “bước xử lý”** (list handler/filter) → **Chain of Responsibility**.
* **Tìm class “XxxFactory/Provider/Builder/…of()”** → **Factory/Abstract Factory/Builder**.
* **Tìm abstract class có method final + “hook” abstract** → **Template Method**.
* **Tìm publish/subscribe, @EventListener** → **Observer**.
* **Tìm orchestration lớp mỏng, gom nhiều service** → **Facade**.
* **Tìm private ctor + getInstance()/static field** → **Singleton** (hoặc bean Spring scope singleton).
* **Tìm cây (node có children cùng kiểu)** → **Composite**.
* **Thấy proxy CGLIB/JDK, @Transactional, @Cacheable** → **Proxy** (AOP).

**Fingerprints theo từng Design Pattern**

**1) Strategy**

* **Dấu hiệu:** 1 interface (thường single-method), nhiều @Component implement; chọn chiến lược bằng cấu hình, @Qualifier, hoặc Map<String, Strategy>.
* **Bạn sẽ thấy:** interface DiscountStrategy { double apply(double a); } + nhiều lớp NoDiscount/Percent10.
* **Grep nhanh:**  
  rg "Map<\s\*String\s\*,\s\*.\*Strategy>"  
  rg "@Qualifier\(\".\*Strategy\"\)"

**2) State**

* **Dấu hiệu:** Hành vi thay đổi theo “trạng thái” (OrderStatus/PaymentStatus). Có thể dùng **enum** override method, hoặc mỗi trạng thái là 1 class implement cùng interface.
* **Bạn sẽ thấy:** order.setState(new PaidState()); order.cancel() gọi hành vi khác nhau.
* **Grep:**  
  rg "enum .\*Status .\*{.\*@Override" -U  
  rg "class .\*State implements .\*State"

**3) Chain of Responsibility**

* **Dấu hiệu:** Handler có setNext() hoặc danh sách List<Handler> chạy tuần tự; hay filter/interceptor/pipeline.
* **Bạn sẽ thấy:** handle(req) gọi tiếp next.handle(req).
* **Grep:**  
  rg "setNext\("  
  rg "List<.\*Handler>"  
  Dấu vết Spring: HandlerInterceptor, Servlet Filter.

**4) Template Method**

* **Dấu hiệu:** abstract class định nghĩa “khung” final doProcess() gọi các “hook” stepA(), stepB() override ở subclass.
* **Bạn sẽ thấy:** AbstractMailer.send() gọi buildSubject(), buildBody().
* **Grep:**  
  rg "abstract class .\*{[^}]\*final .\*\\(" -U  
  rg "extends Abstract.\*Template"

**5) Observer (Publish/Subscribe)**

* **Dấu hiệu:** đăng ký listener, phát sự kiện; trong Spring: ApplicationEventPublisher, @EventListener.
* **Bạn sẽ thấy:** publisher.publishEvent(new OrderPaidEvent(orderId));
* **Grep:**  
  rg "@EventListener"  
  rg "ApplicationEventPublisher"

**6) Factory Method / Abstract Factory**

* **Dấu hiệu:** phương thức createXxx() quyết định lớp con trả về; **Abstract Factory** tạo **họ** đối tượng liên quan.
* **Bạn sẽ thấy:** PaymentFactory.create("paypal") → PaypalPayment.
* **Grep:**  
  rg "class .\*Factory"  
  rg "create[A-Z]"

**7) Builder**

* **Dấu hiệu:** chuỗi gọi setter fluent kết thúc bằng build(); hoặc Lombok @Builder.
* **Bạn sẽ thấy:** User.builder().name("A").age(20).build();
* **Grep:**  
  rg "@Builder"  
  rg "\\.build\\(\\)"

**8) Singleton**

* **Dấu hiệu:** private static final Instance, private constructor, getInstance(); hoặc bean Spring mặc định scope singleton (nhưng **không nhất thiết** là GoF Singleton).
* **Bạn sẽ thấy:** class ConfigRegistry { private static final ConfigRegistry I = new ConfigRegistry(); public static ConfigRegistry get(){ return I; } }
* **Grep:**  
  rg "private static .\* final .\* ="  
  rg "getInstance\\("

**9) Adapter**

* **Dấu hiệu:** lớp “bọc” 1 SDK/API lệch chuẩn thành interface domain của bạn; tên thường XxxAdapter.
* **Bạn sẽ thấy:** class PaypalAdapter implements PaymentGateway { private final PaypalSdk sdk; ... }
* **Grep:**  
  rg "class .\*Adapter"  
  rg "implements .\*Gateway"

**10) Facade**

* **Dấu hiệu:** 1 service “mặt tiền” gọi nhiều subsystem để cung cấp API gọn cho UI/Controller; tên XxxFacade hoặc XxxService rất mỏng.
* **Bạn sẽ thấy:** CheckoutFacade.placeOrder(cart, paymentInfo)
* **Grep:**  
  rg "class .\*Facade"  
  rg "@Service" -n | xargs -I{} sh -c 'sed -n "1,200p" {} | wc -l' (service “mặt tiền” thường ít logic thuần, chủ yếu orchestrate)

**11) Decorator**

* **Dấu hiệu:** class **implement cùng interface** và **giữ field cùng interface**, ủy quyền rồi thêm hành vi (log/cache/rate-limit).
* **Bạn sẽ thấy:**
* class CachingProductService implements ProductService {
* private final ProductService target;
* public Product find(id){ /\* cache around \*/ return target.find(id); }
* }
* **Grep:**  
  rg "implements .\*Service" | rg "private final .\*Service"

**Phân biệt Decorator vs Proxy:**

* **Decorator:** bạn **viết tay**, thêm chức năng.
* **Proxy (AOP):** Spring **tạo runtime** (JDK/CGLIB) để chen advice (@Transactional, @Cacheable).

**12) Proxy (AOP/Remote/Lazy)**

* **Dấu hiệu:** @Transactional/@Cacheable/@Retryable; log/stacktrace có $Proxy/CGLIB; Hibernate HibernateProxy.
* **Bạn sẽ thấy:** class com.myapp.UserService$$EnhancerBySpringCGLIB$$...
* **Grep/Runtime:**  
  rg "@Transactional|@Cacheable|@Retryable"  
  Khi log type: logger.info(bean.getClass().getName()) thấy $Proxy/CGLIB.

**13) Composite**

* **Dấu hiệu:** interface Component có operation(), class Composite giữ List<Component> children; thao tác leaf và composite như nhau.
* **Bạn sẽ thấy:** Category chứa List<Category> children.
* **Grep:**  
  rg "List<\\s\*\\w+\\s\*>\\s\*children"

**14) Flyweight**

* **Dấu hiệu:** factory/pool trả về bản dùng chung theo key; đối tượng **bất biến**; Map cache nội bộ.
* **Bạn sẽ thấy:** GlyphFactory.get('A') trả cùng instance.
* **Grep:**  
  rg "Map<.\*,.\*>.\*cache"  
  rg "intern\\(" (String/own intern-like)

**15) Command**

* **Dấu hiệu:** Command interface có execute(); hàng đợi/nhật ký lệnh; hỗ trợ undo/redo hoặc đẩy queue.
* **Bạn sẽ thấy:** queue.add(new CreateOrderCommand(payload))
* **Grep:**  
  rg "interface .\*Command"  
  rg "execute\\(\\)"

**16) Mediator**

* **Dấu hiệu:** lớp trung gian điều phối nhiều component để chúng không tham chiếu chéo nhau; tên XxxMediator hoặc event bus nội bộ.
* **Bạn sẽ thấy:** formMediator.onFieldChanged(...) -> cập nhật các control khác.
* **Grep:**  
  rg "class .\*Mediator"  
  rg "EventBus|Mediator"

**17) Iterator (tự cài)**

* **Dấu hiệu:** lớp implement Iterator<T>/Iterable<T> cho cấu trúc dữ liệu tự viết.
* **Bạn sẽ thấy:** public class MyTree implements Iterable<Node> { public Iterator<Node> iterator(){...} }
* **Grep:**  
  rg "implements Iterable<|implements Iterator<"

## **4.3. Đào sâu vào Signleton, Factory method**

**Singleton**

**1) Mục đích**

Bảo đảm **chỉ có duy nhất 1 thể hiện** của một lớp trong toàn ứng dụng và cung cấp **điểm truy cập toàn cục**. Hữu ích cho tài nguyên dùng chung (registry, config cache, ID generator, metrics hub…).

Lưu ý: “singleton của Spring” (scope mặc định) là **bean do container quản lý**, khác với GoF Singleton (tự quản lý).

**2) Cách triển khai (từ cơ bản đến “chuẩn”)**

**(a) Eager (đơn giản, thread-safe tự nhiên)**

public final class AppConfig {

private static final AppConfig INSTANCE = new AppConfig();

private AppConfig() {}

public static AppConfig getInstance() { return INSTANCE; }

}

• Ưu: đơn giản, an toàn luồng. • Nhược: khởi tạo ngay cả khi không dùng.

**(b) Lazy + synchronized (dễ hiểu, có overhead)**

public final class LazySingleton {

private static LazySingleton instance;

private LazySingleton() {}

public static synchronized LazySingleton getInstance() {

if (instance == null) instance = new LazySingleton();

return instance;

}

}

• Ưu: khởi tạo khi cần. • Nhược: chi phí synchronized mỗi lần gọi.

**(c) Double-Checked Locking + volatile (ph ổ biến)**

public final class DclSingleton {

private static volatile DclSingleton instance;

private DclSingleton() {}

public static DclSingleton getInstance() {

if (instance == null) {

synchronized (DclSingleton.class) {

if (instance == null) instance = new DclSingleton();

}

}

return instance;

}

}

• Ưu: lazy và nhanh sau lần đầu. • Cần volatile để đảm bảo visibility.

**(d) Initialization-on-Demand Holder (khuyến nghị, gọn & sạch)**

public final class HolderSingleton {

private HolderSingleton() {}

private static class Holder { static final HolderSingleton I = new HolderSingleton(); }

public static HolderSingleton getInstance() { return Holder.I; }

}

• Ưu: lazy, thread-safe, không đồng bộ mỗi lần gọi.

**(e) Enum Singleton (chống serialization/reflection tốt nhất)**

public enum EnumSingleton {

INSTANCE;

public void doWork() {}

}

• Ưu: chống phá vỡ bởi serialization & reflection; chuẩn hoá bởi JLS.  
• Nhược: không linh hoạt khi cần kế thừa/lười khởi tạo phức tạp.

**Chống “vỡ” Singleton**

* **Serialization:** thêm readResolve() để trả về instance duy nhất (trừ enum).
* **Reflection:** trong constructor kiểm tra nếu instance đã tồn tại thì ném exception (enum an toàn hơn).
* **ClassLoader:** nhớ rằng mỗi classloader có thể tạo **một singleton riêng** (ứng dụng server phức tạp).

**3) Ưu điểm**

* Đảm bảo tính **duy nhất**, chia sẻ tài nguyên an toàn.
* **Điểm truy cập thống nhất**, có thể lazy để giảm thời gian khởi động.

**4) Hạn chế & phản-mẫu**

* **Global state** → khó test/mocking, tiềm ẩn coupling ẩn.
* Dễ bị lạm dụng cho logic nghiệp vụ có trạng thái → lỗi cạnh tranh (race).
* Thứ tự khởi tạo giữa các singleton có thể gây lỗi khó lần.

**5) Khi nào nên dùng**

* Một thực thể **thực sự** phải đơn bản (ID generator, cache registry, metrics sink).
* Không dùng cho: DAO/Service business (hãy để Spring quản lý scope), kết nối DB (dùng pool/DataSource).

**6) Trong Spring**

* Bean **mặc định là singleton-scope** → bạn hiếm khi cần GoF Singleton.
* Ưu tiên **DI**: inject bean vào nơi cần → test dễ, thay thế mock tiện.

**Factory Method**

**1) Mục đích**

**Uỷ quyền** việc tạo đối tượng cho **subclass** hoặc cho một “điểm mở rộng”, giúp **client không phụ thuộc** vào lớp cụ thể. Mở rộng loại sản phẩm mà **không sửa** mã client (tuân thủ Open/Closed).

Phân biệt:

* **Factory Method (GoF):** lớp Creator có phương thức factoryMethod() để subclass override.
* **Simple Factory (Static Factory):** một lớp có create(type)/of(...) trả về sản phẩm; tiện dụng nhưng không phải GoF.
* **Abstract Factory:** tạo **họ** sản phẩm liên quan (nhiều loại cùng “theme”).

**2) Cấu trúc kinh điển (Creator – Product)**

// Product

interface Message { String content(); }

class SmsMessage implements Message { public String content(){ return "SMS"; } }

class EmailMessage implements Message { public String content(){ return "EMAIL"; } }

// Creator

abstract class Notifier {

// Factory Method

protected abstract Message createMessage();

// Template method bao khung logic chung

public void send() {

Message m = createMessage();

System.out.println("Sending: " + m.content());

}

}

class SmsNotifier extends Notifier { protected Message createMessage(){ return new SmsMessage(); } }

class EmailNotifier extends Notifier { protected Message createMessage(){ return new EmailMessage(); } }

// Client

Notifier n = new SmsNotifier();

n.send();

• **Ý tưởng:** Client dùng Notifier mà không biết/không phụ thuộc SmsMessage/EmailMessage. Mở rộng bằng cách thêm PushNotifier mà không sửa client.

**3) Simple/Static Factory (thực dụng, rất hay dùng)**

class PaymentFactory {

private static final Map<String, Supplier<Payment>> REG = new HashMap<>();

static {

REG.put("paypal", PaypalPayment::new);

REG.put("cod", CashOnDelivery::new);

}

static Payment create(String type) {

Supplier<Payment> s = REG.get(type);

if (s == null) throw new IllegalArgumentException("Unknown: " + type);

return s.get();

}

}

• **Ưu:** gom quy tắc khởi tạo một chỗ; dễ đăng ký/plug-in.  
• **Nhược:** có thể thành “God Factory” nếu nhồi tất cả bằng switch/map.

**4) Ưu điểm**

* **Tách biệt khởi tạo** khỏi sử dụng; client độc lập với lớp cụ thể.
* **Mở rộng** loại sản phẩm bằng subclass/đăng ký mới mà không sửa client.
* Dễ **áp chính sách** khởi tạo (pooling, cache, validation) một nơi.

**5) Hạn chế**

* Tăng số lớp/độ trừu tượng.
* Nếu dùng sai, factory phình to (switch/map khổng lồ).
* Truyền tham số khởi tạo phức tạp → cân nhắc **Builder** hoặc **Abstract Factory**.

**6) Khi nào nên dùng**

* Cần **plug-in**/chọn lựa implement theo ngữ cảnh (payment, storage driver, serializer).
* Khởi tạo có **chính sách/tiền-điều kiện** cần gom về một mối.
* Kết hợp **Template Method** trong Creator để chia sẻ luồng xử lý.

**7) Trong Java/Spring**

* **JDK**: DocumentBuilderFactory, Calendar#getInstance(), Logger.getLogger() là static factories.
* **JDBC**: DriverManager tải driver qua **Service Provider** rồi tạo Connection.
* **Spring**:
  + @Bean method chính là **factory method** container-managed.
  + FactoryBean<T> (vd LocalContainerEntityManagerFactoryBean) là **mẫu Factory** chuyên dụng.

**So sánh nhanh & quyết định dùng gì**

| **Tình huống** | **Chọn** |
| --- | --- |
| Chỉ được phép tồn tại **1** đối tượng trong toàn hệ thống | **Singleton** (ưu tiên enum/holder; hoặc dùng bean singleton của Spring) |
| Client không nên biết lớp cụ thể; cần **mở rộng loại sản phẩm** linh hoạt | **Factory Method** (hoặc Simple/Static Factory nếu thực dụng) |
| Khởi tạo phức tạp với nhiều tham số tùy chọn | **Builder** (có thể kết hợp Factory để chọn builder phù hợp) |
| Cùng lúc cần tạo **cả họ** đối tượng (liên quan nhau) | **Abstract Factory** |

**Best-practice & Pitfall**

**Singleton**

* Ưu tiên **Holder** hoặc **Enum**.
* Tránh giữ state biến đổi; nếu cần, đồng bộ chặt chẽ hoặc chuyển sang bean Spring + DI.
* Test: cung cấp “backdoor” cấu hình qua DI (singleton Spring), tránh getInstance() hard-coded.

**Factory Method**

* Tránh “God Factory”: dùng **registry Map<String, Supplier<?>>** hoặc **ServiceLoader**.
* Nếu dùng Spring, để container **resolve** implement qua Map<String, Strategy>/List<Strategy> thay vì tự viết factory cứng.
* Kết hợp **Template Method** trong Creator để tái dùng luồng xử lý khung.

**Mẫu code thực dụng (Spring-friendly)**

**1) Singleton theo hướng Spring (không cần GoF Singleton)**

@Service // singleton-scope mặc định

public class IdGenerator {

private final AtomicLong seq = new AtomicLong(0);

public long next() { return seq.incrementAndGet(); }

}

// Inject vào nơi cần: constructor injection → dễ mock trong test.

**2) Factory “đăng ký” không cần switch (plugin-ready)**

public interface Payment { Receipt pay(Order o); }

@Component("paypal") class PaypalPayment implements Payment { /\*...\*/ }

@Component("cod") class CodPayment implements Payment { /\*...\*/ }

@Service

public class PaymentFactory {

private final Map<String, Payment> registry; // Spring auto-inject tất cả bean Payment

public PaymentFactory(Map<String, Payment> registry) { this.registry = registry; }

public Payment of(String method) {

Payment p = registry.get(method);

if (p == null) throw new IllegalArgumentException("Unknown: " + method);

return p;

}

}

// Client: paymentFactory.of(method).pay(order);

• Không cần new, không cần switch, **mở rộng bằng cách thêm bean** mới.

**DI vs Factory: giống & khác**

| **Tiêu chí** | **DI (IoC/Container)** | **Factory (GoF/Simple/Abstract)** |
| --- | --- | --- |
| Mục đích chính | **Tách wiring** (gắn Service A ↔ B ↔ C) khỏi code nghiệp vụ | **Tách khởi tạo** đối tượng khỏi client |
| Ai tạo đối tượng? | **Container** (Spring) khởi tạo & quản lý vòng đời | **Lớp Factory** (do bạn viết) hoặc method @Bean |
| Giảm phụ thuộc bằng | **Tiêm interface** vào constructor; client không new | Client gọi **factory** để lấy “sản phẩm” (interface) |
| Cấu hình/Thay thế | Dùng profile, @Conditional\*, @ConfigurationProperties, Map/List injection | Đổi mapping trong factory (map/switch/registry) |
| Testability | Rất cao (mock bean, test slices) | Khá tốt, nhưng còn phụ thuộc cách viết factory |
| Khi nào dùng | Hầu như mọi service/adapter/repo trong Spring | Khi cần **chính sách khởi tạo**/plug-in/đăng ký sản phẩm |

Thực tế trong Spring: **Container chính là “siêu Factory”**. @Bean là **factory method**; FactoryBean<T> là **mẫu Factory** đặc thù; auto-configuration hoạt động như **Abstract Factory** khổng lồ.

 **“DI và Factory có giảm phụ thuộc bằng cách uỷ quyền cho bên thứ 3?”**  
**Có**, nhưng “bên thứ 3” nên hiểu là **thành phần trung gian**:

* Với **DI**: **IoC Container** chịu trách nhiệm tạo & ghép đối tượng.
* Với **Factory**: **lớp Factory/registry** chịu trách nhiệm tạo sản phẩm.  
  → Client **không phụ thuộc lớp cụ thể**, chỉ nói chuyện qua **interface**.

 **Khác biệt Spring hiện đại vs Core Java trong dùng DP**

* Spring hiện đại: **DP được “dịch” thành tính năng framework** (DI/Proxy/Observer/Chain/FactoryMethod qua @Bean), bạn **khai báo** nhiều hơn **tự cài**.
* Core Java: **tự hiện thực** DP (nhiều boilerplate), tự quản vòng đời & cấu hình.

**A) Spring hiện đại (Spring Boot 3 / Framework 6+)**

* **DI mặc định & constructor injection**: bạn **không viết Singleton thủ công**; bean scope mặc định đã là singleton (container-managed).
* **Factory “ẩn” trong container**:
  + @Bean = **Factory Method**;
  + FactoryBean<T> = **Factory** chuyên dụng;
  + **Auto-Configuration** + @ConditionalOn… = **Abstract Factory** theo điều kiện (class có/không, bean có/không, property…).
* **Strategy “không cần switch”**: inject Map<String, Strategy> / List<Strategy> → thêm chiến lược chỉ việc khai báo bean mới.
* **Proxy/AOP sẵn có**: @Transactional, @Cacheable, @Retryable = **Proxy** + **Decorator** ở runtime.
* **Observer sẵn**: ApplicationEventPublisher / @EventListener = **Observer**.
* **Chain of Responsibility** sẵn\*\*:\*\* filter chain (Servlet/Security), HandlerInterceptor/WebFilter.
* **Cấu hình chuẩn hoá**: @ConfigurationProperties → thay đổi hành vi mà không sửa code (thay vì “Singleton AppConfig” tự viết).
* **AOT/Native** (GraalVM): khuyến khích **constructor injection, class final, tránh reflection tùy tiện** → pattern thiên về **khai báo** hơn là **reflection** tự do.

**B) Core Java (thuần JDK, không Spring)**

* Bạn **tự cài**: **Singleton/Factory/Builder/Strategy/Observer**… bằng code tay (nhiều boilerplate hơn).
* **Khởi tạo & vòng đời** tự quản: dễ rơi vào **Service Locator** (anti-pattern) hoặc static global.
* Dùng **ServiceLoader** (JDK) cho plug-in (gần với Abstract Factory/Strategy động).
* **Config** tự đọc .properties/json/yaml, không có @ConfigurationProperties.
* **AOP/Proxy** nếu cần → tự viết dynamic proxy/CGLIB hoặc dùng thư viện ngoài.

Tóm lại: với Spring hiện đại, **nhiều DP trở thành “tính năng built-in”** (DI, Proxy, Observer…), giúp bạn **dùng qua annotation/cấu hình** thay vì tự lắp ghép pattern bằng tay.

# **5. Database**

## **5.1. Query**

**Thứ tự thực thi logic của một câu truy vấn SQL**

| **Bước** | **Từ khóa** | **Ý nghĩa** |
| --- | --- | --- |
| 1 | **FROM / JOIN** | Xác định bảng nguồn và thực hiện các phép nối dữ liệu. |
| 2 | **WHERE** | Lọc các bản ghi theo điều kiện. |
| 3 | **GROUP BY** | Gom nhóm bản ghi theo cột hoặc biểu thức. |
| 4 | **HAVING** | Lọc các nhóm sau khi gom. |
| 5 | **SELECT** | Xác định cột cần lấy, tính toán, hàm tổng hợp. |
| 6 | **DISTINCT** | Loại bỏ dòng trùng lặp (nếu có). |
| 7 | **ORDER BY** | Sắp xếp kết quả theo một hoặc nhiều cột. |
| 8 | **LIMIT / OFFSET / TOP** | Giới hạn số lượng bản ghi trả về. |

**Điểm cần lưu ý**

* **SELECT không chạy đầu tiên** → vì bạn chỉ có thể chọn/tính toán trên dữ liệu đã được lọc và nhóm từ trước.
* **Alias (tên cột tạm)** định nghĩa trong SELECT chỉ được dùng ở **ORDER BY** hoặc **HAVING**, chứ không dùng được trong WHERE (do WHERE chạy trước SELECT).
* HAVING gần giống WHERE nhưng áp dụng **sau khi GROUP BY**, còn WHERE thì trước.

**So sánh nhanh**

| **Tiêu chí** | **WHERE** | **HAVING** |
| --- | --- | --- |
| Thứ tự xử lý | Trước GROUP BY | Sau GROUP BY |
| Lọc trên | Từng bản ghi (row) | Các nhóm (group) |
| Hàm tổng hợp | ❌ Không dùng trực tiếp | ✅ Dùng được (COUNT, SUM…) |
| Bắt buộc GROUP? | Không cần | Thường dùng với GROUP BY |

** Lọc từng dòng → dùng WHERE**

** Lọc nhóm sau khi tính tổng hợp → dùng HAVING**

**Vì sao có nhiều loại JOIN?**

* Mỗi loại trả lời **một câu hỏi khác nhau** về quan hệ giữa hai tập dữ liệu:
  + “Chỉ lấy phần giao nhau?” (INNER)
  + “Giữ lại mọi dòng bên trái, ghép được thì ghép?” (LEFT)
  + “Giữ mọi dòng cả hai phía?” (FULL)
  + “Tạo mọi cặp kết hợp?” (CROSS)
  + “Lấy những dòng **không có đối ứng**?” (ANTI)
  + “Lấy những dòng **có ít nhất một đối ứng**, nhưng không cần cột bên kia?” (SEMI)
* Về bản chất là các biến thể của **giao/hiệp** (intersection/union) và **phần bù** (difference) trên quan hệ.

**INNER JOIN (giao nhau)**

* **Ý nghĩa**: Chỉ trả về các dòng **khớp** ở cả hai bảng theo điều kiện ON.
* **Khi dùng**: Lấy dữ liệu có liên kết hợp lệ ở cả hai phía (bản ghi “đủ thông tin”).
* **Ví dụ**: Danh sách đơn hàng kèm email khách đã đặt.

**LEFT JOIN (giữ bên trái)**

* **Ý nghĩa**: Giữ **tất cả** dòng từ bảng trái; bảng phải chỉ điền khi khớp, không khớp → NULL.
* **Khi dùng**: Báo cáo “bao phủ” từ phía trái, kể cả khi thiếu đối ứng (dữ liệu mồ côi).
* **Ví dụ**: Liệt kê **mọi khách hàng** và đơn gần đây (nếu có).

**RIGHT JOIN (giữ bên phải)**

* **Ý nghĩa**: Tương tự LEFT nhưng giữ **mọi dòng bảng phải**.
* **Khi dùng**: Hiếm hơn (có thể đảo vị trí bảng để dùng LEFT JOIN).
* **Ví dụ**: Mọi đơn hàng và khách (kể cả đơn “không rõ khách” nếu cho phép).

**FULL OUTER JOIN (giữ cả hai phía)**

* **Ý nghĩa**: Giữ **mọi dòng** từ cả hai bảng, ghép nếu khớp, không khớp thì NULL phía còn lại.
* **Khi dùng**: Đối soát dữ liệu giữa hai nguồn để thấy cả khớp và **lệch** hai phía.
* **Lưu ý MySQL**: **Không hỗ trợ** trực tiếp. Dùng **giả lập** bằng LEFT JOIN … UNION ALL … RIGHT JOIN (lọc trùng).

**CROSS JOIN (tích Descartes)**

* **Ý nghĩa**: Kết hợp **mọi cặp** hàng giữa hai bảng (không điều kiện).
* **Khi dùng**: Sinh lịch/khung dữ liệu, tạo lưới tổ hợp (cẩn thận bùng nổ số dòng).

**SELF JOIN (tự nối)**

* **Ý nghĩa**: JOIN **chính bảng đó** với biệt danh khác.
* **Khi dùng**: Cây phân cấp (category–parent), quan hệ bạn bè, so sánh trong cùng bảng.

Bảng quyết định nhanh (decision table)

| **Câu hỏi của bạn** | **JOIN nên dùng** |
| --- | --- |
| Chỉ lấy bản ghi có đối ứng ở cả hai phía? | **INNER JOIN** |
| Giữ mọi dòng phía A, ghép được thì ghép? | **LEFT JOIN** (A ở bên trái) |
| Giữ mọi dòng phía B, ghép được thì ghép? | **RIGHT JOIN** (hoặc đảo bảng + LEFT) |
| Muốn thấy cả khớp và phần dư hai phía? | **FULL OUTER JOIN** (MySQL: giả lập bằng UNION) |
| Cần mọi tổ hợp giữa hai tập? | **CROSS JOIN** |
| Lấy A có/không có ít nhất một đối ứng ở B? | **SEMI / ANTI** (EXISTS / NOT EXISTS) |
| Dữ liệu phân cấp trong một bảng? | **SELF JOIN** |

**Subquery là gì? Phân loại nhanh**

* **Theo mối liên hệ với truy vấn ngoài**
  + **Non-correlated**: subquery chạy độc lập, không dùng cột bên ngoài.
  + **Correlated**: subquery **tham chiếu cột** của truy vấn ngoài ⇒ chạy lặp lại cho từng dòng (cẩn trọng hiệu năng).
* **Theo kiểu kết quả**
  + **Scalar**: trả về **1 giá trị** (1 hàng × 1 cột).
  + **Row / Tuple**: trả về **1 hàng** (nhiều cột).
  + **Table**: trả về **nhiều hàng/cột** (dùng với IN/EXISTS hoặc như “bảng tạm” trong FROM).

**Khi nào nên dùng subquery?**

* Lọc có/không có đối ứng (semi/anti) với EXISTS/NOT EXISTS.
* So sánh với **tổng hợp** (AVG, SUM…) theo nhóm liên quan.
* Tạo bảng trung gian nhanh (derived table) khi JOIN trực tiếp khó đọc.
* Tuy nhiên: nếu cần **tái sử dụng** nhiều lần → cân nhắc **CTE (WITH)**; nếu cần dữ liệu từ bảng con → **JOIN** thường rõ và nhanh hơn.

**CTE là gì?**

* **CTE (Common Table Expression)** là một tập kết quả tạm thời, có tên (named result set), được định nghĩa bằng mệnh đề WITH ... AS (...) và **chỉ tồn tại trong phạm vi của một câu lệnh SQL ngay sau nó**.
* Có 2 loại:
  1. **CTE thường (non-recursive)**: giống như một subquery/derived table có tên.
  2. **CTE đệ quy (recursive)**: cho phép một CTE tham chiếu chính nó để duyệt/cấp bậc.

**Mục đích sinh ra**

* **Tách** truy vấn phức tạp thành các bước logic dễ đọc.
* **Tái sử dụng** kết quả trung gian trong **cùng** một câu lệnh.
* **Giải bài toán đệ quy** (cây thư mục, sơ đồ tổ chức, BOM…) vốn khó viết bằng JOIN/SELF-JOIN thuần.

**Đặc điểm & Ưu điểm**

1. **Đặt tên và phân tầng bước xử lý**
   * Viết nhiều CTE xếp chồng: WITH step1 AS (...), step2 AS (SELECT ... FROM step1), ....
   * Tài liệu hóa luồng xử lý dữ liệu ngay trong câu lệnh SQL.
2. **Tính cục bộ (scope)**
   * Chỉ sống trong 1 statement → không “bẩn” schema (không cần tạo VIEW/TEMP TABLE).
3. **Đệ quy tự nhiên**
   * Mô tả anchor + step lặp bằng UNION ALL → dễ đọc, dễ kiểm soát độ sâu.
4. **Tối ưu hóa (tùy hệ quản trị)**
   * Thường **được inline** như subquery (SQL Server, MySQL 8…);
   * Một số hệ (PostgreSQL) có thể **materialize** (tạo bảng tạm nội bộ) khi cần, hoặc cho phép cưỡng bức với từ khóa (ví dụ MATERIALIZED) → tránh tính lại khi tham chiếu nhiều lần.

**Cơ chế giúp có các ưu điểm trên:**

* **Name binding & scope**: ràng buộc tên CTE chỉ trong statement kế tiếp.
* **Inline expansion** của optimizer: coi CTE như biểu thức con để đẩy/pull predicate, hợp nhất kế hoạch.
* **Anchor/recursive member**: máy thực thi lặp lại phần recursive trên kết quả tích lũy (fixpoint) cho đến khi không còn dòng mới hoặc đạt ngưỡng.

**Dùng CTE để giải quyết gì trong thực tế?**

* **Cấu trúc phân cấp**: Sơ đồ tổ chức, cây danh mục, thư mục cha–con, BOM (Bill of Materials).
* **Tạo dãy/lich**: Calendar table động, dãy số/dates để join lấp đầy thiếu dữ liệu.
* **Pipeline nhiều bước**: Lọc → chuẩn hóa → tính KPI → xếp hạng → phân trang.
* **Top-N per group**: Kết hợp **CTE + window functions** để chọn bản ghi đứng đầu theo nhóm.
* **Làm sạch và gỡ rối truy vấn**: Thay vì lồng subquery sâu khó đọc.

**Cú pháp nhanh**

**CTE thường:**

WITH sales\_clean AS (

SELECT s.\*, NULLIF(TRIM(customer), '') AS customer\_clean

FROM sales s

WHERE order\_date >= '2025-01-01'

),

ranked AS (

SELECT customer\_clean, product\_id, revenue,

ROW\_NUMBER() OVER (PARTITION BY customer\_clean ORDER BY revenue DESC) AS rn

FROM sales\_clean

)

SELECT \*

FROM ranked

WHERE rn = 1;

**CTE đệ quy (ANSI / PostgreSQL / MySQL 8: dùng WITH RECURSIVE):**

WITH RECURSIVE org AS (

-- Anchor: bắt đầu từ root

SELECT id, manager\_id, name, 0 AS depth, CAST(id AS CHAR(200)) AS path

FROM employees

WHERE manager\_id IS NULL

UNION ALL

-- Recursive: nối con

SELECT e.id, e.manager\_id, e.name, o.depth + 1,

CONCAT(o.path, '>', e.id) AS path

FROM employees e

JOIN org o ON e.manager\_id = o.id

WHERE o.depth < 10 -- chặn độ sâu

)

SELECT \* FROM org;

**SQL Server (không cần từ khóa RECURSIVE, nhưng cấu trúc giống):**

WITH Org AS (

SELECT Id, ManagerId, Name, 0 AS Depth

FROM Employees

WHERE ManagerId IS NULL

UNION ALL

SELECT e.Id, e.ManagerId, e.Name, o.Depth + 1

FROM Employees e

JOIN Org o ON e.ManagerId = o.Id

)

SELECT \* FROM Org

OPTION (MAXRECURSION 100);

**Lỗi sai thường gặp & cách tránh**

1. **Đệ quy không có điều kiện dừng / giới hạn độ sâu**
   * Hậu quả: vòng lặp vô hạn hoặc “nổ” dữ liệu.
   * Cách tránh: thêm điều kiện dừng (ví dụ WHERE depth < N) hoặc OPTION (MAXRECURSION n) (SQL Server).
2. **Dùng UNION thay vì UNION ALL trong phần đệ quy**
   * UNION loại trùng mỗi vòng → rất chậm và có thể làm mất dòng hợp lệ.
   * Dùng UNION ALL rồi tự khử trùng (nếu cần) ở ngoài hoặc bằng kỹ thuật đánh dấu visited.
3. **ORDER BY bên trong CTE**
   * Nhiều hệ **không đảm bảo thứ tự** trong CTE; ORDER BY chỉ hợp lệ khi kết hợp TOP/OFFSET hoặc ở truy vấn ngoài.
   * Đặt ORDER BY ở **câu SELECT cuối**.
4. **Kỳ vọng “tính một lần dùng nhiều nơi” nhưng hệ không materialize**
   * SQL Server/MySQL thường inline → tham chiếu nhiều lần có thể khiến **tính lặp lại**.
   * Nếu **cần tái sử dụng nhiều lần** hoặc debug/perf: cân nhắc **TEMP TABLE** hoặc (PostgreSQL) thêm MATERIALIZED.
5. **Đặt filter sai chỗ**
   * Lọc trong anchor/recursive member khác với lọc ở ngoài → có thể tăng dữ liệu trung gian.
   * **Đẩy điều kiện sớm** (early filter) vào anchor/recursive nếu logic cho phép.
6. **Không chống vòng lặp/cycle** trong đồ thị có chu trình
   * Dẫn đến lặp vô hạn.
   * Cách tránh: theo dõi path/visited (như ví dụ path), hoặc dùng tính năng CYCLE nếu hệ quản trị hỗ trợ.
7. **Lạm dụng CTE cho mọi thứ**
   * Quá nhiều CTE liên tiếp làm truy vấn khó bảo trì; khi một tập trung gian cần **được vận hành nhiều lần/được index tạm**, hãy dùng **#temp table** (SQL Server) / **TEMP TABLE**.
8. **Hiểu nhầm về tối ưu hóa**
   * PostgreSQL trước v12 coi CTE là rào cản tối ưu (optimization fence) → có thể chậm. Từ v12 trở đi có thể inline (trừ khi ép MATERIALIZED). Biết rõ DB của bạn để chọn chiến lược.

**Khi nào nên dùng CTE, khi nào không?**

* **Nên dùng**:
  + Truy vấn dài, nhiều bước → cần “đặt tên từng bước”.
  + Bài toán phân cấp/đồ thị.
  + Top-N per group, phân trang, xử lý tổ hợp + window functions.
* **Không nên** (hoặc cân nhắc):
  + Cần dùng lại kết quả **nhiều lần** trong nhiều câu lệnh → dùng TEMP TABLE/VIEW.
  + Cần index tạm trên kết quả trung gian → TEMP TABLE.
  + Hệ quản trị của bạn **materialize bắt buộc** CTE và gây chậm → thử subquery hoặc join trực tiếp.

**Nó là gì?**

* **ROW\_NUMBER()**: gán **số thứ tự duy nhất** cho từng dòng **trong mỗi nhóm (PARTITION)** theo **ORDER BY**. Không coi trọng “điểm hòa”.
* **RANK()**: gán hạng, **có nhảy số khi hòa** (ví dụ: 1, 2, 2, 4).
* **DENSE\_RANK()**: gán hạng **không nhảy số khi hòa** (ví dụ: 1, 2, 2, 3).

Cả ba đều là **window functions**: dùng cú pháp

<func>() OVER (

[PARTITION BY <cột\_nhóm>, ...]

ORDER BY <tiêu\_chí\_xếp\_hạng>

)

**Mục đích sinh ra**

* Xếp hạng/phân hạng **trong từng nhóm dữ liệu** (theo khách hàng, danh mục, lớp học…).
* Lấy **Top-N mỗi nhóm**, phân trang ổn định, chọn **bản ghi đại diện** (dedup).
* Giữ câu SQL **mạch lạc** thay vì subquery lồng nhau phức tạp.

**Đặc điểm & Ưu điểm**

* **Linh hoạt theo nhóm**: PARTITION BY giúp xếp hạng độc lập theo từng nhóm.
* **Kiểm soát hòa**: chọn **RANK** (nhảy số) hay **DENSE\_RANK** (không nhảy số) tùy yêu cầu nghiệp vụ; **ROW\_NUMBER** để lấy duy nhất 1 dòng.
* **Dễ đọc & kết hợp**: hoạt động tốt cùng **CTE** và **window khác** (SUM/AVG/…).
* **Hiệu năng tốt** cho rất nhiều bài toán Top-N so với JOIN/NOT EXISTS rắc rối (miễn là **ORDER BY trong window** khớp index).

**Cơ chế giúp có ưu điểm mạnh**

* **Windowing**: Engine **phân vùng (partition)** dữ liệu rồi **sắp xếp (sort)** theo ORDER BY; sau đó **quét** từng partition để phát hạng.
* **Thứ tự thực thi logic**: window functions được tính **sau** FROM/WHERE/GROUP BY/HAVING nhưng **trước** ORDER BY cuối; vì thế **không dùng được** trong mệnh đề WHERE (trừ DB hỗ trợ QUALIFY).
* **Xử lý hòa**:
  + ROW\_NUMBER: tăng 1 cho mỗi dòng sau sort (không quan tâm hòa).
  + RANK: chỉ tăng khi giá trị **ORDER BY** thay đổi, số hạng **nhảy** theo số dòng hòa.
  + DENSE\_RANK: tương tự RANK nhưng **không nhảy số**.

**Dùng để giải quyết gì trong thực tế?**

* **Top-N mỗi nhóm**: best-seller theo danh mục, top điểm mỗi lớp.
* **Chọn bản ghi mới nhất** theo khóa tự nhiên (dedup theo email, user\_id…).
* **Phân trang ổn định** (stable pagination) dựa trên thứ tự rõ ràng.
* **Cắt dữ liệu theo hạng**: khuyến mãi chỉ áp cho **rank ≤ 3**, v.v.

**Bảng so sánh nhanh**

| **Hàm** | **Hòa điểm (ties)** | **Ví dụ thứ hạng cho điểm 100, 95, 95, 90** |
| --- | --- | --- |
| ROW\_NUMBER | Không coi hòa, mỗi dòng 1 số | 1, 2, 3, 4 |
| RANK | Có hòa, **nhảy số** | 1, 2, 2, 4 |
| DENSE\_RANK | Có hòa, **không nhảy số** | 1, 2, 2, 3 |

**Lỗi sai thường gặp & cách tránh**

1. **Thiếu ORDER BY trong window** → hạng **không xác định**.  
   → Luôn chỉ rõ ORDER BY; thêm **tie-breaker** (PK) để ổn định.
2. **Dùng sai hàm cho bài toán có hòa**:
   * Cần “Top 3 **bao gồm cả hòa**” → **RANK** (hoặc DENSE\_RANK).
   * Cần “**duy nhất** 1 dòng mỗi nhóm” → **ROW\_NUMBER**.
3. **Dùng window function trong WHERE** (không hợp lệ ở nhiều hệ).  
   → Bọc một lớp SELECT ngoài, hoặc dùng QUALIFY (nếu DB hỗ trợ như BigQuery/Snowflake).
4. **Nhầm lẫn ORDER BY trong window vs ORDER BY ngoài cùng**:
   * ORDER BY **trong window** = tiêu chí xếp hạng.
   * ORDER BY **ngoài** = cách **trình bày** kết quả cuối.
5. **Không kiểm soát NULL** trong ORDER BY → thứ tự bất ngờ.  
   → Chủ động sắp NULL bằng CASE (SQL Server/MySQL) hoặc NULLS FIRST/LAST (PostgreSQL/Oracle).
6. **Hiệu năng kém do sort lớn**:  
   → Tạo **index** khớp PARTITION BY + ORDER BY (ví dụ (category\_id, revenue DESC, product\_id)), hoặc **tiền lọc** dữ liệu trước khi xếp hạng.
7. **Kỳ vọng “khung (frame)” ảnh hưởng hạng**:
   * Với 3 hàm này, **frame không tác động**; chúng phụ thuộc **phân vùng & thứ tự**, không phải ROWS/RANGE.

**Khi nào dùng cái nào?**

* **ROW\_NUMBER**: Chọn **duy nhất** 1 bản ghi/nhóm (mới nhất, rẻ nhất…), phân trang ổn định.
* **RANK**: Top-N **có thể vượt N** khi hòa; dùng khi yêu cầu nghiệp vụ muốn **giữ toàn bộ hòa**.
* **DENSE\_RANK**: Cần thứ hạng **liền mạch** (không nhảy số) để báo cáo/trực quan.

**VIEW là gì?**

* **VIEW** là **truy vấn được đặt tên** (stored query) lưu trong schema như một đối tượng DB.
* Khi SELECT từ VIEW, hệ quản trị sẽ **thay thế** VIEW bằng câu SQL gốc (và tối ưu/thi hành). VIEW **không tự lưu dữ liệu** (trừ **materialized/ indexed view** ở một số DB).

**Mục đích sinh ra**

* **Ẩn phức tạp** truy vấn, chuẩn hóa logic nghiệp vụ (1 nơi định nghĩa, nhiều nơi dùng).
* **Ổn định giao diện dữ liệu** (API dữ liệu): đổi cấu trúc bảng bên dưới, VIEW giữ nguyên schema cho ứng dụng.
* **Phân quyền/che giấu dữ liệu** (chỉ cho xem một số cột/hàng).
* **Tái sử dụng** và **tài liệu hóa** các phép tính/joins.

**Đặc điểm & Ưu điểm**

1. **Tái sử dụng & nhất quán**: 1 định nghĩa – dùng ở nhiều câu SQL/ứng dụng.
2. **Độc lập lược đồ**: hạn chế ảnh hưởng khi đổi cấu trúc bảng thật (đổi/đổi tên cột → chỉ cần cập nhật VIEW).
3. **Bảo mật**: cấp quyền trên VIEW thay vì bảng (ẩn cột nhạy cảm; lọc theo điều kiện).
4. **Tổ chức mã SQL**: mã sáng sủa hơn, dễ kiểm thử.

**Lưu ý về hiệu năng:** VIEW **thường không tự làm truy vấn nhanh hơn** — nó chỉ là “mã SQL có tên”. Tốc độ phụ thuộc vào **kế hoạch thực thi** sau khi optimizer “mở” VIEW ra. Muốn tăng tốc đọc, dùng **materialized view** (Oracle/Postgres) hoặc **indexed view** (SQL Server) – có chi phí cập nhật/refresh.

**Cơ chế giúp có ưu điểm**

* **Query rewriting / inlining**: Optimizer **merge** câu SELECT trên VIEW vào SQL ngoài để đẩy điều kiện (predicate pushdown), loại bỏ cột dư, v.v.
* **Dependency metadata**: DB theo dõi phụ thuộc giữa VIEW ↔ bảng → đổi bảng sẽ invalid/ cảnh báo VIEW (SQL Server SCHEMABINDING, Postgres dependency graph).
* **Tùy chọn materialize** (tùy hệ):
  + **SQL Server**: *Indexed View* (yêu cầu WITH SCHEMABINDING, index clustered, tuân thủ nhiều ràng buộc).
  + **PostgreSQL**: MATERIALIZED VIEW + REFRESH [CONCURRENTLY].
  + **Oracle**: *Materialized View* + *Query Rewrite*.
  + **MySQL**: **không có** materialized view native (dùng event/trigger/bảng trung gian giả lập).

**Dùng VIEW để giải quyết gì trong thực tế?**

* **Lớp truy cập dữ liệu ổn định** cho BI/reporting: tổng hợp KPI, chuẩn hóa JOINs phức tạp.
* **Bảo mật**: *column masking* & *row filtering* (ví dụ chỉ thấy đơn hàng của cửa hàng mình).
* **Chuẩn bị dữ liệu** cho công cụ báo cáo (Power BI/Looker/Tableau) không phải viết JOIN dài.
* **Tạo “góc nhìn” đa schema**: hợp nhất nhiều bảng từ nhiều nguồn.

**Những lỗi thường gặp & cách tránh (kèm ví dụ)**

1. **Nghĩ rằng ORDER BY trong VIEW sẽ áp đặt thứ tự khi SELECT**
   * Sai: Thứ tự chỉ được đảm bảo khi **SELECT ngoài cùng** có ORDER BY.
   * *Ví dụ sai*:
   * CREATE VIEW v\_top AS
   * SELECT \* FROM orders ORDER BY created\_at DESC; -- tưởng là luôn “mới nhất trước”
   * SELECT \* FROM v\_top; -- KHÔNG đảm bảo thứ tự
   * *Cách đúng*: SELECT \* FROM v\_top ORDER BY created\_at DESC;
2. **Xếp chồng VIEW lồng nhau (view-on-view)** → kế hoạch thực thi phức tạp, khó tối ưu
   * *Mẹo*: Hạn chế tầng lồng; profile kế hoạch (EXPLAIN). Nếu nặng, cân nhắc viết lại 1 view “phẳng” hoặc dùng bảng tạm/mv cho heavy aggregation.
3. **Dùng SELECT \* trong VIEW** → đổi schema bảng gốc gây lệch cột/đổi thứ tự
   * *Cách đúng*: liệt kê cột rõ ràng để tránh rủi ro.
4. **Cập nhật qua VIEW lọc mà không WITH CHECK OPTION** → ghi dữ liệu “lọt ra ngoài” bộ lọc
   * *Ví dụ sai (Postgres/MySQL/SQL Server)*:
   * CREATE VIEW v\_active AS
   * SELECT \* FROM users WHERE active = 1;
   * INSERT INTO v\_active(id, name, active) VALUES (100, 'A', 0);
   * -- Row vừa chèn KHÔNG xuất hiện trong v\_active (sai mong đợi).
   * *Cách đúng*:
   * CREATE VIEW v\_active AS
   * SELECT \* FROM users WHERE active = 1
   * WITH CHECK OPTION;
   * -- INSERT/UPDATE qua view buộc phải thỏa active=1
5. **Kỳ vọng VIEW có index riêng**
   * Thường **không** (vì view không lưu dữ liệu). Nếu cần index → **indexed view/materialized view** (tùy hệ) hoặc index trên **bảng gốc**.
6. **Cập nhật VIEW không updatable**
   * VIEW chứa DISTINCT, GROUP BY, UNION, aggregate, SELECT list biểu thức phức tạp, hoặc JOIN nhiều bảng (tùy DB) → **không updatable** (trừ khi dùng **INSTEAD OF TRIGGER** ở SQL Server/Oracle).
   * *Ví dụ*:
   * CREATE VIEW v AS
   * SELECT dept\_id, COUNT(\*) AS n
   * FROM employees GROUP BY dept\_id;
   * UPDATE v SET n = 10 WHERE dept\_id = 5; -- thất bại (không updatable)
7. **MySQL VIEW “TEMPTABLE” → không updatable**
   * MySQL có ALGORITHM = MERGE | TEMPTABLE | UNDEFINED; nếu engine chọn TEMPTABLE, view **không updatable**.

**Có thể CRUD trên VIEW không?**

* **Đọc (SELECT)**: luôn được.
* **Insert/Update/Delete**:
  + **Tùy DB & tùy loại VIEW**:
    - **PostgreSQL**: *simple view* thường **tự updatable** (không aggregate/distinct/group/union; 1 bảng cơ sở); có thể dùng **INSTEAD OF trigger** để tùy biến ghi nhiều bảng.
    - **SQL Server**: view có thể update/insert/delete nếu đủ điều kiện; có thể dùng **INSTEAD OF trigger** để cho phép CRUD phức tạp (multi-table). Hỗ trợ WITH SCHEMABINDING và **Indexed View**.
    - **MySQL**: nhiều view **updatable** nếu không có DISTINCT/GROUP BY/UNION…; join-view chỉ updatable với một số ràng buộc, và **không** nếu bị TEMPTABLE. Có WITH CHECK OPTION.
    - **Oracle**: tương tự; hỗ trợ **INSTEAD OF trigger** cho view phức tạp.
* **Materialized/Indexed view**: CRUD **không** ghi trực tiếp vào MV/IV; phải ghi vào bảng gốc (DB sẽ duy trì MV/IV theo cơ chế riêng).

**VIEW vs CTE vs (Temp Table / Materialized View)**

**Hiểu đúng câu “cùng tạo bảng tạm”**:

* **VIEW** **không** tạo bảng tạm; nó là **định nghĩa truy vấn**.
* **CTE** **không** tạo đối tượng schema; là **kết quả tạm thời trong phạm vi 1 câu lệnh**.
* **Bảng tạm** (temporary table) **có dữ liệu vật lý** tạm thời (session/transaction).
* **Materialized view**/**Indexed view** **lưu dữ liệu vật lý** và có thể index/refresh.

| **Tiêu chí** | **VIEW** | **CTE** | **Temp Table** | **Materialized/Indexed View** |
| --- | --- | --- | --- | --- |
| Tồn tại | Lâu dài (schema) | Trong **1 statement** | Trong **session/tx** | Lâu dài (schema) |
| Lưu dữ liệu | Không | Không | **Có** | **Có** |
| Tối ưu | Optimizer inline/merge | Inline như subquery | Kế hoạch tách bước, có thể index | Đọc **nhanh**; phải **refresh/duy trì** |
| Tái sử dụng | Cao (nhiều câu lệnh) | Không (chỉ 1 câu) | Trong phiên | Cao |
| CRUD trực tiếp | Có thể (điều kiện) | Không | Có | Không (ghi vào bảng gốc) |
| Dùng khi | Ẩn phức tạp, security, API dữ liệu | Tách bước logic 1 truy vấn | Cần tạm lưu, index, xử lý nhiều bước/nhắc lại | Tăng tốc đọc báo cáo/aggregation tốn kém |

**Khi nào dùng cái nào?**

* **VIEW**: chuẩn hóa & bảo mật lớp đọc; ổn định schema cho app/BI.
* **CTE**: làm “bước trung gian” cho **một** truy vấn dài; cả đệ quy.
* **Temp table**: pipeline nhiều bước có tái sử dụng trong **phiên**, cần **index tạm** để tối ưu.
* **Materialized / Indexed view**: workload đọc nặng/aggregation tốn kém, chấp nhận chi phí cập nhật/refresh.

**Stored Procedure là gì?**

* **Stored Procedure** (thủ tục lưu trữ) là **khối lệnh SQL có tên** được biên dịch/lưu trên DBMS, gọi bằng CALL (MySQL/PostgreSQL) hoặc EXEC (SQL Server).
* Có **tham số IN/OUT/INOUT**, có thể trả **nhiều result set**, và **có side-effects** (INSERT/UPDATE/DELETE, giao dịch…).

**Mục đích sinh ra**

* Gom nghiệp vụ **gần dữ liệu**: giảm round-trip mạng, tái sử dụng logic phức tạp.
* **Bảo mật/kiểm soát truy cập**: cấp EXECUTE thay vì cho phép DML trực tiếp.
* Tối ưu hiệu năng ổn định với kế hoạch thực thi (plan cache).

**Đặc điểm & Ưu điểm**

* **Nhiều kết quả**: trả về 0..n result set + OUT params.
* **Quyền & kiểm soát**: dễ audit, hạn quyền (row/column level qua view + proc).
* **Giao dịch**: mở/commit/rollback trong thân procedure (tùy DB).
* **Tách lớp nghiệp vụ chung** giữa nhiều app/trình báo cáo.

**Cơ chế “mạnh”**

* **Compilation/Plan cache**: lần đầu biên dịch, các lần sau tái dùng kế hoạch (SQL Server mạnh nhất khoản này; các DB khác cũng tối ưu tương tự).
* **Parameterization**: tối ưu theo kiểu dữ liệu/giá trị tham số.
* **Batching**: gom nhiều lệnh trong một call → giảm round-trip.

**Dùng procedure để giải quyết gì?**

* **Quy trình nghiệp vụ nhiều bước**: tạo đơn hàng → chèn item → tính tổng → cập nhật tồn kho → ghi log → commit.
* **ETL nhẹ/tiền xử lý báo cáo**: tổng hợp theo kỳ, snapshot dữ liệu.
* **Nâng bảo mật**: chỉ cho EXECUTE trên proc, ẩn bảng gốc/chi tiết join.
* **Thao tác hàng loạt**: upsert/bulk operations có kiểm soát.

**Lỗi thường gặp & cách tránh (kèm ví dụ)**

1. **Không SET NOCOUNT ON (SQL Server)** → JDBC hiểu nhầm rowcount là result set → OUT param khó đọc.
2. CREATE PROCEDURE dbo.p\_demo AS
3. BEGIN
4. -- NÊN:
5. SET NOCOUNT ON;
6. -- ... DML/SELECT thật sự cần trả về
7. END
8. **Gọi sai API JDBC**: dùng executeQuery() cho proc không trả ResultSet → lỗi “No results returned”.  
   → Dùng execute() (sau đó duyệt getResultSet()/getMoreResults()).
9. **Nhầm thứ tự tham số / kiểu dữ liệu** → SQLException.  
   → Khai báo rõ IN/OUT và dùng registerOutParameter() đúng Types.
10. **Không tiêu thụ hết các result set** → kết nối “bận”, lỗi ở lệnh tiếp theo.  
    → Vòng while (hasResults) { ... hasResults = cs.getMoreResults(); }.
11. **Thay đổi “shape” result set** mà không sửa code Java mapping.  
    → Cố định tên cột/alias; tài liệu hóa hợp đồng (contract).
12. **Đưa quá nhiều nghiệp vụ vào DB** → khóa chặt vendor, khó CI/CD test.  
    → Chỉ đặt vào DB phần nào *thực sự có lợi* (bảo mật, batch, sát dữ liệu).
13. **MySQL function/trigger cập nhật bảng** (bị hạn chế) dùng sai chỗ.  
    → Cập nhật dữ liệu → dùng **Procedure**, không phải **Function**.

**So sánh nhanh: Procedure vs Function**

| **Tiêu chí** | **Procedure** | **Function (UDF)** |
| --- | --- | --- |
| Cách gọi | CALL/EXEC p(...) | SELECT f(...); hoặc trong biểu thức/WHERE |
| Trả về | 0..n result set + OUT param | **1 giá trị** (hoặc 1 record type tùy DB) |
| Side-effects (DML) | **Có** (điển hình) | **Hạn chế mạnh** (SQL Server/PG: không DML trong UDF thuần; MySQL: bị ràng buộc) |
| Dùng trong SELECT | **Không** | **Có** (nhúng vào biểu thức) |
| Giao dịch nội bộ | Thường **có thể** BEGIN/COMMIT/ROLLBACK | Thường **không** |
| Tối ưu & Index | Tối ưu như batch | UDF có thể làm truy vấn **không SARGable** (SQL Server) nếu dùng sai |

Ghi chú theo hệ:  
• **PostgreSQL**: từ v11 có CREATE PROCEDURE + CALL. Trước đó hay dùng FUNCTION (VOLATILE/STABLE/IMMUTABLE).  
• **SQL Server**: CREATE PROCEDURE, CREATE FUNCTION (UDF) **không** được DML/transaction trong scalar UDF (trừ TVF).  
• **MySQL**: CREATE PROCEDURE/FUNCTION; function bị hạn chế hơn về side-effects.

## **5.2. Index**

**1) Index là gì?**

Cấu trúc dữ liệu (thường **B-tree**) giúp **tìm/đọc nhanh** (seek/range scan) theo giá trị khóa, thay vì quét toàn bảng (full scan).

* **Đọc nhanh hơn** nhưng **ghi tốn kém hơn** (mỗi INSERT/UPDATE/DELETE phải cập nhật index), tốn RAM/đĩa.

**2) Phân biệt Clustered Index và Non-clustered Index**

**Khái niệm**

* **Clustered index**
  + Quy định **thứ tự vật lý (hoặc logic gần-vật-lý)** của bản ghi trong bảng.
  + **SQL Server**: lá của B-tree **chính là dữ liệu hàng**; mỗi bảng **tối đa 1** clustered index (nếu không có → bảng dạng heap).
  + **MySQL/InnoDB**: **PRIMARY KEY là clustered index**; dữ liệu được tổ chức theo PK; các secondary index **lưu PK làm con trỏ**.
  + **PostgreSQL**: không có “clustered index” duy trì tự động. Có lệnh CLUSTER để **sắp xếp lại** bảng theo một index, nhưng **không tự duy trì** khi dữ liệu thay đổi (bảng vẫn là heap).
* **Non-clustered index**
  + Cây B-tree **riêng** chỉ chứa **khóa + con trỏ** về dữ liệu gốc.
  + **SQL Server**: con trỏ là **(clustered key)** nếu bảng có clustered index, hoặc **RID** nếu là heap.
  + **MySQL/InnoDB**: node lá chứa **khóa của secondary index + PK**; từ PK quay về bản ghi.
  + **PostgreSQL**: index riêng biệt; truy cập dữ liệu qua **Index Scan**/**Bitmap Index Scan**.

**So sánh hiệu năng (và vì sao)**

* **Tốc độ tra cứu (lookup) theo khóa index:**
  + **Clustered**: tìm là **đến ngay dữ liệu** (SQL Server) hoặc chỉ cần **1 bước** theo PK (InnoDB) → **ít I/O hơn**.
  + **Non-clustered**: thường phải **thêm 1 bước “lookup”** để lấy đủ cột từ bảng (Key Lookup/RID Lookup) → nếu trả **nhiều dòng**, tổng chi phí có thể cao.
* **Truy vấn theo khoảng (range by key) / ORDER BY theo key:**
  + **Clustered** rất mạnh vì **dữ liệu đã được “xếp” sẵn** → đọc tuần tự, giảm random I/O, thường **khỏi cần sort**.
  + **Non-clustered**: vẫn mạnh nếu **covering** hoặc **ORDER BY** khớp index; nếu không, có thể phải **sort thêm** hoặc **lookup hàng loạt**.
* **Ghi (insert/update/delete):**
  + Mọi index đều **tăng chi phí ghi**.
  + **Clustered** chịu **page split/fragmentation** nặng nếu khóa **ngẫu nhiên** (ví dụ GUID) → giảm throughput.
  + **InnoDB**: vì mọi secondary index **lưu PK**, nếu PK **dài** (UUID chuỗi) → **phình to** mọi index → chậm và tốn đĩa.
  + **SQL Server**: cập nhật **clustered key** sẽ **cascade** đến **mọi non-clustered index** (do chúng lưu clustered key làm con trỏ).
* **Full scan / Bulk insert:**
  + **Heap** (không clustered) đôi khi nhanh hơn cho **full scan** hoặc **bulk insert**, nhưng phải đánh đổi khả năng **range scan/ORDER BY**.

**Kết luận thực chiến:**

* Chọn **clustered key** nhỏ, tăng dần (IDENTITY/SEQUENCE, snowflake…), **ổn định** (ít/không cập nhật).
* Dùng **non-clustered/secondary index** có chiến lược: **phục vụ predicate, JOIN, ORDER BY/GROUP BY** cụ thể; cố gắng **covering** để tránh lookup.

**3) Có giới hạn số lượng index trên 1 bảng không?**

* **Có giới hạn kỹ thuật theo hệ** (con số khác nhau, thường rất cao: ví dụ **SQL Server: 1 clustered + ~999 non-clustered**). MySQL/PostgreSQL cũng có giới hạn cao (hàng chục–hàng trăm), hiếm khi chạm trần trong thực tế.
* **Giới hạn thực tế** là **chi phí**:
  + Mỗi index làm **tăng thời gian ghi**, **tăng kích thước** table/index, **tăng thời gian lock/maintain** (rebuild, vacuum…).
  + Nhiều index còn làm **optimizer** khó chọn kế hoạch tối ưu.
* **Quy tắc ngón tay cái**: giữ số index **ít nhưng “đúng mục tiêu”** (thường 5–10 là đủ cho 1 bảng OLTP lớn), đo đạc trước/sau.

**4) Đánh index dựa trên cơ sở nào? (nguyên tắc thiết kế)**

1. **Dựa trên truy vấn thật (workload)**
   * Cột trong WHERE, JOIN ON, ORDER BY, GROUP BY, DISTINCT.
   * Mẫu lọc **chính tả**: =/IN/BETWEEN/> < (SARGable).
   * Tránh viết predicate **không SARGable** (hàm lên cột, tính toán bên trái).
     + ❌ WHERE DATE(created\_at) = '2025-08-28'  
       ✅ WHERE created\_at >= '2025-08-28' AND created\_at < '2025-08-29'
2. **Độ chọn lọc (selectivity)**
   * Chỉ số tốt nhất khi cột có **độ phân biệt cao**.
   * Với **index tổng hợp (multi-column)**: **thứ tự cột** quan trọng. Cột **lọc mạnh nhất** nên đứng **trước**.
3. **Covering index (bao phủ đủ cột)**
   * Thêm cột vào index để **truy vấn chỉ đọc từ index** (không phải quay về bảng).
   * **SQL Server**: INCLUDE (colA, colB); **PostgreSQL**: INCLUDE (v11+); **MySQL**: không có INCLUDE, muốn cover thì thêm vào **danh sách khóa** (lưu ý giới hạn kích thước).
4. **Partial/Filtered index**
   * **PostgreSQL**: CREATE INDEX ... WHERE status = 'ACTIVE' (rất hiệu quả).
   * **SQL Server**: **Filtered Index** tương tự.
   * **MySQL**: không hỗ trợ partial index (chỉ có **prefix** index cho chuỗi).
5. **Hướng ORDER BY**
   * Postgres có thể **quét đảo chiều** trên 1 index cho ASC/DESC.
   * MySQL 8 hỗ trợ **DESC index**; trước đây bị hạn chế.
   * SQL Server dùng 1 index cho cả ASC/DESC (quét ngược).
6. **Kích thước khóa & kiểu dữ liệu**
   * Khóa **ngắn** → B-tree **cao thấp hơn** → ít I/O hơn.
   * InnoDB: **PK ngắn** càng quan trọng (do mọi secondary index chứa PK).
7. **OLTP vs OLAP**
   * OLTP: ưu tiên **seek** và **ghi nhanh** → ít index, đúng điểm đau.
   * OLAP: cân nhắc **materialized view**, **columnstore** (SQL Server), hoặc index đặc thù.

**5) Làm sao biết truy vấn đã dùng index?**

**Dùng kế hoạch thực thi (execution plan)**

* **MySQL**
  + EXPLAIN SELECT ... → cột **type** (ref, range, index, ALL...), **key**, **key\_len**, **rows**, **Extra**.
  + Extra có **“Using index”** = **covering** (chỉ đọc index). **“Using where; Using index”** là tốt; **“ALL”** = full scan.
  + Muốn số liệu thực tế: EXPLAIN ANALYZE SELECT ... (8.0+) để xem **actual rows**.
* **PostgreSQL**
  + EXPLAIN (ANALYZE, BUFFERS) SELECT ... → sẽ thấy **Index Scan**, **Index Only Scan**, **Bitmap Index Scan** (kèm **Heap/Bitmap Heap Scan**).
  + **Index Only Scan** nhanh nhất khi **visibility map** đủ tốt (ít phải recheck heap).
* **SQL Server**
  + Bật **Actual Execution Plan** (SSMS) hoặc SET STATISTICS IO, TIME ON.
  + Tên toán tử: **Index Seek/Scan**, **Key Lookup (RID Lookup)**. Có **Key Lookup** nghĩa là non-clustered không cover đủ cột (có thể tối ưu bằng INCLUDE).

**Dùng thống kê hệ thống**

* **SQL Server**: sys.dm\_db\_index\_usage\_stats, sys.dm\_db\_missing\_index\_\*.
* **PostgreSQL**: pg\_stat\_user\_indexes, pg\_stat\_statements.
* **MySQL**: performance\_schema + **sys schema**; xem digest/latency theo truy vấn.

## **5.3. Partition**

**Partition là gì?**

* **Partitioning** chia **một bảng lớn** thành **nhiều phân vùng nhỏ** (partitions) dựa trên **khóa phân vùng** (partition key) nhưng **vẫn được truy cập như 1 bảng logic duy nhất**.
* Mỗi partition là một **đơn vị lưu trữ & bảo trì độc lập** (có thể thêm/xóa/truncate/tái lập chỉ mục riêng).

**Mục đích sinh ra**

* **Tăng tốc truy vấn** bằng cách **đọc ít dữ liệu hơn** (partition pruning).
* **Bảo trì/ETL linh hoạt**: nạp/xóa dữ liệu theo “lô thời gian” (sliding window), TRUNCATE PARTITION nhanh.
* **Tăng khả năng song song** (scan nhiều partition), giảm contention.
* **Cải thiện tính sẵn sàng** (đảo/di chuyển partition).

**Ưu điểm chính cho truy vấn (hiệu năng)**

1. **Partition pruning (loại trừ phân vùng)**: chỉ đọc các partition liên quan điều kiện WHERE (thường theo *thời gian*).
2. **Partition-wise operations**: một số hệ hỗ trợ **join/aggregate theo từng partition** rồi hợp nhất (giảm I/O).
3. **Ít dữ liệu phải sort/scan** nếu ORDER BY/GROUP BY gắn với khóa phân vùng.

**Hiệu năng = Index + Partition phối hợp.** Partition **không thay thế** index. Nó giúp **cắt bớt dữ liệu cần xét**, còn index quyết định **cách tìm trong phần còn lại**.

**Cơ chế tạo ra ưu điểm**

* **Metadata mapping**: Mỗi partition có **biên (bounds)** (RANGE/LIST/HASH). Optimizer dùng điều kiện truy vấn để **chọn subset partition**.
* **Local index**: đa số hệ tạo index **riêng từng partition** → seek/scan chỉ trong partition tương ứng.
* **Parallelism**: mỗi partition là đơn vị **chia nhỏ công việc** cho các worker.

**Khi nào nên partition?**

* Bảng **rất lớn** (từ vài chục GB / hàng trăm triệu bản ghi trở lên) và **truy vấn/ETL mang tính thời gian** (30/90/365 ngày gần nhất, theo tháng/quý).
* Cần **xóa/lưu trữ (archive)** dữ liệu cũ nhanh (theo tháng/quý/năm).
* Cần **bulk load**/swap dữ liệu theo lô (nhập 1 tháng rồi SWITCH/EXCHANGE PARTITION).
* Dữ liệu **nghiêng theo thời gian** (most recent hot), cần tách hot/cold.

**Khi không nên**

* Bảng nhỏ/vừa (vài triệu bản ghi trở xuống) → chi phí quản trị > lợi ích.
* Truy vấn không gắn khóa phân vùng (không pruning được).
* Lạm dụng tạo **hàng nghìn partition cực nhỏ** → overhead lớn.

**Các kiểu partition (cốt lõi)**

* **RANGE** (phổ biến nhất): theo mốc thời gian/số (tháng/quý/năm).
* **LIST**: theo danh sách giá trị rời rạc (vùng, quốc gia…).
* **HASH**: phân tán đồng đều khi không có quy luật rõ ràng (tránh hotspot).
* (MySQL) **SUBPARTITION**: RANGE + HASH (ít dùng, chỉ khi thật cần).

**Ảnh hưởng tới Index (liên hệ Clustered/Non-clustered)**

* **Index thường là “local” theo partition** (mỗi partition có B-tree riêng).
* **SQL Server**: nên **align** (index partitioned cùng function/scheme) để SWITCH nhanh; non-aligned index cản trở switch.
* **PostgreSQL**: **mọi ràng buộc unique/PK bắt buộc chứa partition key** (để đảm bảo duy nhất xuyên các partition).
* **MySQL/InnoDB**: secondary index **per-partition**; PK ngắn vẫn quan trọng (ảnh hưởng kích thước mọi index).

**Có giới hạn bao nhiêu partition?**

* **Giới hạn kỹ thuật**: rất cao (hàng nghìn đến chục nghìn tùy hệ).
* **Giới hạn thực tế**: càng nhiều partition → overhead plan/metadata I/O càng lớn.
  + Thực chiến OLTP: **vài chục đến vài trăm** lành mạnh; OLAP có thể nhiều hơn nhưng vẫn nên nhóm theo tháng/quý thay vì ngày/giờ trừ khi bắt buộc.

**Đánh partition dựa trên cơ sở nào?**

1. **Workload thật**: điều kiện lọc phổ biến (thường theo **thời gian**).
2. **Kích thước mỗi partition**: chọn **đủ lớn** để giảm overhead nhưng **đủ nhỏ** để bảo trì nhanh (tháng/quý thường hợp lý).
3. **Độ cân bằng**: tránh **skew** (mất cân). Nếu dữ liệu đổ toàn vào “tháng hiện tại” → cân nhắc **subpartition/hash** cho hot partition.
4. **Chốt “sliding window”**: chiến lược tạo/đóng/bỏ partition định kỳ (automation).
5. **Ràng buộc & Index**:
   * PG: **PK/UNIQUE phải chứa partition key**.
   * SQL Server: **aligned index** để SWITCH.
   * MySQL: đảm bảo predicate **SARGable** trên partition key để pruning.

**Làm sao biết truy vấn có dùng partition?**

* **MySQL**: EXPLAIN SELECT ... → cột **partitions** hiển thị partition được quét; EXPLAIN ANALYZE cho “actual rows”.
* **PostgreSQL**: EXPLAIN (ANALYZE, BUFFERS) → thấy **Append/Bitmap Append** chỉ chứa **các partition liên quan**; plan note **“Partition pruning”** (phiên bản mới).
* **SQL Server**: xem **Actual Execution Plan** → Partitioned Index Scan/Seek, Predicate có **Range** trên PartitionID; SET STATISTICS IO ON cũng giúp so I/O.

## **5.4. Stored Procedure**

**1) Stored Procedure (SP) là gì? Sinh ra để làm gì?**

**Stored Procedure** là **đoạn chương trình SQL được biên dịch/lưu sẵn trong CSDL** (MySQL/SQL Server/PostgreSQL/Oracle…), có thể **gọi bằng tên** kèm tham số để thực thi các thao tác đọc/ghi dữ liệu, kiểm tra ràng buộc, giao dịch (transaction)…

**Mục đích chính**

* **Đưa logic gần dữ liệu** để giảm số lần round-trip giữa ứng dụng ↔ DB, tăng tốc.
* **Tập trung hóa nghiệp vụ**, tái sử dụng, chuẩn hóa đầu vào/đầu ra.
* **Bảo mật & phân quyền**: cấp quyền **EXECUTE** thay vì quyền trực tiếp trên bảng.
* **Giao dịch nhất quán**: gom nhiều câu lệnh vào **1 transaction**.

**2) Đặc điểm & ưu điểm – cơ chế tạo “sức mạnh”**

* **Được DB quản lý**: tên, tham số, quyền, logging, dependency.
* **(Thường) được tối ưu & cache kế hoạch thực thi**: giúp **ổn định hiệu năng** cho các lời gọi lặp lại (SQL Server/Oracle nổi bật; PostgreSQL/MySQL tối ưu khác nhau theo phiên bản).
* **Giảm lưu lượng mạng**: thay vì gửi nhiều câu lệnh rời rạc, chỉ **CALL** 1 thủ tục.
* **Bao đóng nghiệp vụ + giao dịch**: BEGIN/COMMIT/ROLLBACK tại một nơi.
* **Kiểm soát truy cập**: không cần cấp quyền SELECT/INSERT… trên bảng gốc.
* **Phù hợp xử lý theo lô (batch/ETL)**: chạy theo lịch, ít phụ thuộc app.

**Cơ chế cốt lõi**: chạy **gần dữ liệu** + (ở nhiều hệ) **plan-cache** + **set-based operations** + **transaction atomicity** ⇒ giảm chi phí NET/CPU, tăng thông lượng.

**3) Dùng SP để giải quyết gì trong thực tế?**

* **Nghiệp vụ phức tạp đa bảng**: tạo đơn hàng, xuất kho, ghi nhận thanh toán.
* **ETL/Báo cáo định kỳ**: tổng hợp số liệu ngày/tuần/tháng.
* **API hướng DB**: app chỉ gọi EXEC sp\_xxx → DB chịu trách nhiệm chuẩn hóa.
* **Bảo mật**: che bảng nhạy cảm, chỉ expose SP với kiểm tra quyền/đầu vào.
* **Kiểm soát đồng thời**: khóa/kiểm tra tồn tại, phát hiện xung đột.
* **Audit/Logging**: ghi lịch sử thay đổi (history tables) một cách tập trung.

**4) Lỗi sai thường gặp (và cách tránh) – kèm ví dụ “dùng sai”**

**(a) Dồn quá nhiều nghiệp vụ vào DB (overuse)**

* **Hệ quả**: SP quá lớn/khó test/khó CI/CD, phụ thuộc chặt vào vendor SQL.
* **Khắc phục**: để **quy tắc miền (domain rule) thay đổi thường xuyên** ở tầng app; SP lo **điều phối dữ liệu**, tính toán nặng, transaction xuyên bảng.

**(b) Vòng lặp từng dòng (RBAR – *Row By Agonizing Row*)**

-- SAI (giả định SQL Server): cập nhật từng dòng bằng CURSOR

DECLARE cur CURSOR FOR SELECT Id FROM OrderItems WHERE OrderId = @OrderId;

OPEN cur;

FETCH NEXT FROM cur INTO @Id;

WHILE @@FETCH\_STATUS = 0

BEGIN

UPDATE OrderItems SET Amount = Qty \* Price WHERE Id = @Id; -- chạy lặp lại

FETCH NEXT FROM cur INTO @Id;

END

CLOSE cur; DEALLOCATE cur;

**Đúng – set-based**:

UPDATE oi

SET Amount = oi.Qty \* oi.Price

FROM OrderItems oi

WHERE oi.OrderId = @OrderId;

**(c) Thiếu transaction hoặc xử lý lỗi**

-- SAI: nếu INSERT 1 thành công, INSERT 2 thất bại => dữ liệu dở dang

INSERT INTO Orders(...);

INSERT INTO OrderItems(...); -- lỗi giữa chừng -> mất nhất quán

**Đúng (SQL Server)**:

BEGIN TRY

BEGIN TRAN;

INSERT ... -- 1

INSERT ... -- 2

COMMIT TRAN;

END TRY

BEGIN CATCH

IF XACT\_STATE() <> 0 ROLLBACK TRAN;

THROW;

END CATCH

**Đúng (MySQL)**

START TRANSACTION;

-- INSERT ...

-- INSERT ...

COMMIT; -- hoặc ROLLBACK khi lỗi qua HANDLER

**(d) SQL injection trong dynamic SQL**

-- SAI: nối chuỗi trực tiếp

SET @sql = 'SELECT \* FROM Users WHERE name = ''' + @name + '''';

PREPARE s FROM @sql; EXECUTE s;

**Đúng**: **tham số hóa**

SET @sql = 'SELECT \* FROM Users WHERE name = ?';

PREPARE s FROM @sql; EXECUTE s USING @name;

**(e) Không SARGable → không dùng được index**

-- SAI: hàm bọc cột

WHERE DATE(created\_at) = '2025-08-01' -- làm mất index

-- Đúng:

WHERE created\_at >= '2025-08-01' AND created\_at < '2025-08-02'

**(f) Parameter sniffing (SQL Server)**

* Dùng 1 plan cho mọi giá trị tham số → có thể **lệch kế hoạch**.
* **Khắc phục**: OPTION (RECOMPILE) khi phù hợp, hoặc OPTIMIZE FOR UNKNOWN,/local variable.

**(g) Lặt vặt nhưng hay gặp (SQL Server)**

* Quên SET NOCOUNT ON; → trả về nhiều thông điệp “x rows affected”, làm hỏng client xử lý đa kết quả.
* SELECT \* trong SP public → thay đổi schema làm **vỡ** client.

**6) Hiệu năng & maintainability – best practices**

* **Thiết kế tham số hóa** (không nối chuỗi SQL), validate đầu vào ngay trong SP.
* **Ưu tiên thao tác theo tập (set-based)**, tránh cursor/vòng lặp từng dòng.
* **SARGable** để tận dụng index; cân nhắc **covering index** cho truy vấn trọng yếu.
* **Quản lý transaction ngắn gọn**, tránh giữ khóa lâu; chọn **isolation level** phù hợp.
* **Theo dõi plan** & thống kê (stats); với SQL Server, xử lý **parameter sniffing** đúng chỗ.
* **Phiên bản hóa SP** (tên hoặc migration), có **unit test/integration test** tối thiểu.
* Tránh SELECT \*; trả về **schema rõ ràng** để client ổn định.
* Ghi SET NOCOUNT ON (SQL Server) trong SP trả về nhiều bước.

**7) Khi nào nên / không nên dùng SP?**

* **Nên**: nghiệp vụ **gắn chặt dữ liệu**, cần **transaction xuyên bảng**, batch/ETL, tối ưu hiệu năng nơi nút thắt là **round-trip** & **băng thông**.
* **Không nên**: logic **thay đổi liên tục**, cần **test/CI/CD** mạnh ở tầng app, hoặc khi muốn **đa CSDL**/đa vendor (tránh khóa chặt vào dialect).