Contents

[**1. Java Core Basic** 2](#_Toc207183263)

[**1.1. Primitive and Object data type** 2](#_Toc207183264)

[**1.2. String** 3](#_Toc207183265)

[**1.3. Static & Final** 6](#_Toc207183266)

[**1.4. OOP** 9](#_Toc207183267)

[**1.5. Memory** 14](#_Toc207183268)

[**1.6. Exception handle** 16](#_Toc207183269)

[**2. Java Collections** 18](#_Toc207183270)

[**2.1. Array** 18](#_Toc207183271)

[**2.2. List** 20](#_Toc207183272)

[**2.3. Queue** 22](#_Toc207183273)

[**2.4. Set & Map** 24](#_Toc207183274)

[**3. Java Thread** 28](#_Toc207183275)

[**3.1. Synchronous vs Asynchronous** 28](#_Toc207183276)

[**3.2. Thread** 30](#_Toc207183277)

[**4. Design Patterns** 34](#_Toc207183278)

[**4.1. Tổng quan** 34](#_Toc207183279)

[**4.2. Nhận biết** 38](#_Toc207183280)

[**4.3. Đào sâu vào Signleton, Factory method** 42](#_Toc207183281)

# **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 hướng đối tượng (OOP)**

* **Encapsulation (đóng gói):** ẩn chi tiết dữ liệu, chỉ lộ API cần thiết.
* **Abstraction (trừu tượng):** mô tả *cái gì làm* thay vì *làm thế nào*, thực thi qua abstract class / interface.
* **Inheritance (kế thừa):** tái sử dụng và mở rộng hành vi theo quan hệ *is-a*.
* **Polymorphism (đa hình):** cùng lời gọi nhưng hành vi khác nhau theo kiểu thực thi (ghi đè).

**2) Access modifier trong Java**

|  |  |
| --- | --- |
| **Mức** | **Truy cập được từ đâu** |
| public | Mọi nơi |
| *(default)* = package-private | Trong **cùng package** |
| protected | Cùng package **và** subclass ở package khác |
| private | Chỉ trong **chính lớp** |

Lưu ý: top-level class chỉ có thể là public hoặc *default*; còn thành viên (field/method) dùng đủ 4 mức.

**3) Phân biệt class và instance**

* **Class**: bản thiết kế (khai báo field, method, constructor, thành viên static).
* **Instance (object)**: một thể hiện cụ thể của class, có trạng thái riêng (các field không-static).

**4) Abstract vs Interface**

**Abstract class**

* Không khởi tạo trực tiếp; chứa **abstract method** lẫn **method có thân**, có thể có **state** & **constructor**; cho phép mọi mức truy cập.

**Interface**

* Hợp đồng hành vi: gồm **abstract**, **default**, **static methods** (đều ngầm định public), và hằng số (public static final).
* **Default method** giúp tiến hóa API; được kế thừa như instance method.

**Khi nào dùng abstract class?**

Dùng **abstract class** khi bạn cần một **nền tảng có trạng thái chung + triển khai mặc định đáng kể**:

* Có **field** dùng chung, **constructor**, **protected helper** để giữ invariant.
* Áp dụng **Template Method pattern** (xác định khung xử lý, phần chi tiết để subclass hoàn thiện).
* Muốn kiểm soát **khả năng mở rộng** theo trục “một hệ phân cấp is-a” (Java chỉ cho **một** lớp cha).
* Cần các **access modifier** linh hoạt (private/protected/package) cho members

abstract class PaymentProcessor {

protected final String merchantId; // state chung

protected PaymentProcessor(String merchantId){ this.merchantId = merchantId; }

public final void process(double amount) { // khung xử lý cố định

validate(amount);

String token = authorize(amount);

capture(token);

settle();

}

protected void validate(double amount) { if (amount <= 0) throw new IllegalArgumentException(); }

protected abstract String authorize(double amount); // tuỳ gateway

protected void capture(String token) { /\* default impl \*/ }

protected void settle() { /\* default impl \*/ }

}

**Khi nào dùng interface?**

Dùng **interface** khi cần mô tả **hợp đồng/khả năng** và/hoặc muốn **đa kế thừa kiểu**:

* API công cộng, **plug-in/SPI**, **Strategy**, **Observer**…
* Lớp khác nhau (không chung tổ tiên) vẫn có thể cùng “biết làm” một việc.
* (Java 8+) **default method** để thêm hành vi mặc định nhẹ, **static method** cho helper liên quan giao diện.

**Nếu trùng tên method giữa nhiều nguồn (2 interface hoặc abstract + interface) thì sao?**

**Cả hai abstract,** cùng chữ ký số và cùng kiểu trả về thì implement 1 method là đủ, nó thỏa cả 2.

Cả hai **abstract**, **cùng chữ ký nhưng kiểu trả về *covariant***

Nếu **một kiểu trả về là *subtype* của kiểu kia**. Lớp có thể chọn **kiểu hẹp hơn** để thỏa cả hai.

interface A { Number m(); }

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

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

Cả hai **abstract**, **cùng chữ ký nhưng kiểu trả về *không tương thích***  thì lỗi biên dịch

interface A { Integer m(); }

interface B { String m(); }

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

Hai **default method** cùng chữ ký

**Bắt buộc override** trong lớp để phân giải. Có thể chọn cụ thể một bản

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(); } // chọn bản của A

}

Suy ra với interface–interface, **xung đột default → lớp phải override**; **xung đột abstract → xét “return-type compat”** (covariant thì được, không thì lỗi).

**Lớp cha (abstract/concrete) có method cùng chữ ký, interface có default**

**Class thắng interface**.

* Nếu superclass **concrete**: lớp con **kế thừa** method từ class; default của interface **bị che**.
* Nếu superclass **abstract**: lớp con **phải tự triển khai** (default trong interface **không cứu**).

abstract class Base { abstract void hi(); }

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

class Sub extends Base implements X {

@Override public void hi(){ System.out.println("Sub"); } // buộc phải tự cài

}

**1) Khái niệm & mục tiêu**

**Abstract class**

* Là lớp **không thể khởi tạo trực tiếp**, có thể chứa **trạng thái (fields)**, **constructor**, **method thường** và **abstract method** (không có thân).
* Dùng khi các lớp con **cùng chia sẻ dữ liệu + một phần triển khai** và bạn muốn **định khung (template)** thuật toán nhưng cho phép tùy biến vài bước.

**Interface**

* Là **hợp đồng hành vi**. Chứa **abstract methods** và (Java 8+) **default/static methods**; (Java 9+) có thể có **private methods** dùng nội bộ giữa default/static.
* Dùng khi nhiều lớp **không cùng họ hàng** nhưng **cùng biết làm một việc** (đa kế thừa kiểu), hoặc khi bạn muốn **tách rời** phần triển khai với API.

Quy tắc nhớ nhanh: **Cần state & khung xử lý → abstract class. Cần hợp đồng/mix-in cho nhiều hệ phân cấp → interface.**

**2) So sánh khả năng (bảng ngắn gọn)**

| **Tiêu chí** | **Abstract class** | **Interface** |
| --- | --- | --- |
| Khởi tạo | Không thể new trực tiếp | Không thể new trực tiếp |
| Kế thừa | extends **1** abstract/concrete class | implements **n** interface (đa kế thừa kiểu) |
| Field | Có **instance/static fields** tùy ý | Chỉ **hằng số** public static final |
| Constructor | **Có** constructor (được gọi từ subclass) | **Không** có constructor |
| Method | Abstract + **concrete**; mọi access modifier | Abstract (ngầm public), **default**, **static**, (Java 9+) **private** |
| Access level | public/protected/package/private | Methods abstract/default/static **ngầm public** |
| Dùng cho | Chia sẻ **state + triển khai** | Định nghĩa **hợp đồng/khả năng**; **đa kế thừa** |
| Mẫu thiết kế điển hình | **Template Method**, **Skeleton** | **Strategy**, **SPI/Plugin**, **Observer** |

**5) Thế nào là Overriding và Overloading**

* **Overriding (ghi đè):** subclass định nghĩa lại method **cùng tên & tham số**; kiểu trả về có thể **covariant**; ràng buộc checked exceptions hẹp hơn/ít hơn; quyết định lúc **runtime** (dynamic dispatch). Xảy ra ở class con.
* **Overloading (nạp chồng):** cùng tên nhưng **khác danh sách tham số**; chọn phiên bản lúc **compile-time** theo tham số. Xảy ra trong 1 class.

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

* **private**: *không* — vì không được thừa kế; cùng chữ ký ở subclass chỉ là method mới.
* **static**: *không* override mà là **hiding** (che khuất); lời gọi tĩnh phân giải theo **kiểu tại biên dịch** (ClassName.m()), không theo runtime.

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

* **Kế thừa**: *có thể nhìn thấy & gọi* như bình thường (nếu không private).
* **Ghi đè**: **không thể**

**8) Phân biệt hai từ khóa this và super**

* **this**: tham chiếu tới **đối tượng hiện tại**; dùng để phân biệt field bị che khuất, gọi **constructor cùng lớp** bằng this(...).
* **super**: tham chiếu đến **phần superclass** của đối tượng hiện tại; gọi **constructor cha** super(...), hoặc gọi bản cài đặt ở cha khi bị override (super.m()).

## **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? Vì sao dùng?**

**Array (mảng)** là cấu trúc dữ liệu lưu các phần tử **cùng kiểu** trong **vùng nhớ liên tiếp**, truy cập bằng **chỉ số (index)** bắt đầu từ 0 (đa số ngôn ngữ).  
Điểm mạnh:

* Truy cập ngẫu nhiên O(1) nhờ địa chỉ phần tử = base + index \* size.
* Tối ưu cache CPU (locality tốt).  
  Điểm yếu:
* Kích thước cố định (trong nhiều ngôn ngữ). Thêm/xoá giữa mảng tốn kém (dịch phần tử).
* Với đa chiều, việc bố trí bộ nhớ/“mảng răng cưa” khác nhau giữa các ngôn ngữ.

**Khi nào dùng?**

* Dữ liệu cố định hoặc biết trước cỡ tối đa.
* Cần truy cập ngẫu nhiên siêu nhanh, tính toán trên chỉ số (DP, hai con trỏ, prefix sum).
* Tối ưu hiệu năng/nhớ: int[], double[] (Java) thường tốt hơn cấu trúc phức tạp.

**2) Mô hình bộ nhớ & locality**

* **Liên tiếp** → duyệt tuần tự nhanh, thao tác khối (memcpy) hiệu quả.
* **Java**: int[] chứa trực tiếp giá trị nguyên thủy; String[] chứa **tham chiếu** đến đối tượng String. Mảng đa chiều là **mảng của mảng** (không đảm bảo liên tiếp hai chiều).
* Header/độ lệch… khác nhau giữa JVM, không nên “đoán cứng”; hiểu nguyên tắc là đủ cho tối ưu.

**3) Độ phức tạp thao tác (Big-O)**

| **Thao tác** | **Thời gian** |
| --- | --- |
| Truy cập theo chỉ số | O(1) |
| Cập nhật theo chỉ số | O(1) |
| Tìm tuyến tính | O(n) |
| Chèn/xoá ở đầu/giữa | O(n) (dịch phần tử) |
| Thêm/xoá ở cuối (mảng tĩnh) | O(1) nếu còn chỗ; ngược lại O(n) để “nới” |

**Dynamic array** (như ArrayList) dùng mảng nền + chiến lược nới size → *amortized* O(1) cho add.

**4) Các biến thể/khái niệm thường gặp**

* **Dynamic array**: tự nới kích thước (Java: ArrayList).
* **Multidimensional**: a[r][c]; trong Java thực chất là “mảng của mảng” (jagged) → có thể mỗi hàng độ dài khác nhau.
* **Sparse data**: khi phần tử “rải thưa”, cân nhắc HashMap/int → value thay vì mảng lớn.
* **Typed arrays (JS)**: Int32Array, Float64Array… tối ưu tính toán số.

**5)Bẫy thường gặp (Java)**

* **ArrayIndexOutOfBoundsException**: luôn kiểm soát biên [0..length-1], viết test “off-by-one”.
* **So sánh sai**: arr1.equals(arr2) → tham chiếu; phải dùng Arrays.equals/deepEquals.
* **Covariance**: String[] là Object[]; gán phần tử sai kiểu ném **ArrayStoreException**.  
  → Tránh trộn kiểu, ưa dùng mảng kiểu nguyên thủy/kiểu cụ thể.
* **Generics + Array**: không thể new List<String>[10]. Dùng List<List<String>> hoặc ép kiểu có kiểm soát.
* **Mảng đa chiều** trong Java **không liên tiếp hai chiều** → tính năng như C/NumPy không áp dụng tương tự.
* **Varargs**: có thể tạo *mảng rác* nếu gọi nhiều lần trong vòng lặp nóng → cân nhắc quá tải hàm không dùng varargs.

**6) Tối ưu hiệu năng**

* **Ưu tiên kiểu nguyên thủy** (int[], double[]) tránh boxing/unboxing.
* **Duyệt tuần tự** thay vì nhảy lung tung → tận dụng cache.
* **Giảm cấp phát**: tái sử dụng buffer, dùng System.arraycopy, Arrays.fill.
* **Biết trước kích thước** → cấp phát một lần, tránh nới nhiều lần.
* **Song song** khi mảng rất lớn & phép tính độc lập: Arrays.parallelSort, IntStream.parallel().
* **Bitset/boolean[]** cho bài toán đánh dấu nhanh (sàng số nguyên tố, visited…).

**7) Chọn Array hay ArrayList (Java)?**

* **Array**: hiệu năng tối đa, kích thước cố định, API mức thấp, thích hợp lõi thuật toán.
* **ArrayList**: linh hoạt kích thước, API phong phú (add/remove, subList, sort…), đổi lại có overhead và *amortized* nới.
* Mẹo: Viết thuật toán bằng int[]/double[]; chỉ chuyển sang List khi cần API/khả năng tương tác.

## **2.2. List**

**1) List là gì? Khi nào dùng?**

**List<E>** là cấu trúc dữ liệu **tuần tự có thứ tự**, cho phép phần tử **trùng lặp**, truy cập theo **chỉ số**.  
Dùng List khi bạn cần:

* Thêm/xóa **linh hoạt** (khác mảng tĩnh).
* Duy trì **thứ tự chèn**.
* API phong phú: add, remove, get, set, sort, subList, replaceAll, removeIf…

List là **interface**; lựa chọn **implementation** quyết định hiệu năng & đặc tính.

**2) Các implementation thường dùng**

| **Kiểu** | **Đặc điểm chính** | **Thời gian (xấp xỉ)** |
| --- | --- | --- |
| **ArrayList** | Dựa trên mảng; truy cập ngẫu nhiên nhanh; nới size theo bội số 1.5x | get/set: O(1); addLast: O(1)\*; chèn/xóa giữa: O(n) |
| **LinkedList** (cũng là Deque) | Danh sách móc nối; tốt cho thao tác 2 đầu | chèn/xóa **đầu/cuối**: O(1); truy cập theo index: O(n) |
| **CopyOnWriteArrayList** | Thread-safe **đọc nhiều, ghi ít** (mỗi lần ghi sẽ copy mảng) | đọc/iterate: rất nhanh & snapshot; ghi: O(n) |
| **Vector/Stack** | Di sản, synchronized toàn bộ; **tránh dùng** | — |

\* amortized

**Gợi ý chọn nhanh**

* Truy cập theo chỉ số, duyệt nhiều → **ArrayList**.
* Cần hàng đợi hai đầu → **LinkedList** (hoặc **ArrayDeque** nếu không cần List).
* Đa luồng đọc nhiều, ít ghi → **CopyOnWriteArrayList**.
* Cần bất biến → List.of(...) / List.copyOf(...).

**3) Tính bất biến, đồng bộ & bản sao**

**Bất biến / không sửa đổi**

* List.of(e1, e2, ...) (Java 9+): **unmodifiable**; **không cho null**.
* List.copyOf(coll): unmodifiable **copy** (nếu đầu vào đã unmodifiable và cùng kiểu, có thể trả lại chính nó).
* Collections.unmodifiableList(list): **view** không sửa đổi **bọc** quanh list gốc (gốc đổi → view đổi).

“Unmodifiable ≠ Immutable”: phần tử bên trong vẫn có thể thay đổi trạng thái nếu là đối tượng mutable.

**4) Độ phức tạp thao tác**

| **Thao tác** | **ArrayList** | **LinkedList** |
| --- | --- | --- |
| get(i), set(i, v) | O(1) | O(n) |
| add(v) cuối | O(1)\* | O(1) |
| add(i, v) giữa | O(n) | O(n)\*\* |
| remove(i) giữa | O(n) | O(n)\*\* |
| removeFirst/Last (qua Deque) | — | O(1) |

\* amortized; \*\* tìm nút O(n), xóa/chèn bản thân nút O(1).

## **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 & Map**

**Map**

* Cấu trúc **key → value**, **key duy nhất**.
* Truy cập theo key, không theo chỉ số.
* Các biến thể chính:
  + **HashMap**: trung bình O(1), **không đảm bảo thứ tự**.
  + **LinkedHashMap**: như HashMap nhưng **giữ thứ tự** chèn (hoặc theo **truy cập**), dùng làm **LRU cache**.
  + **TreeMap**: **có thứ tự** theo Comparator/Comparable, O(log n).
  + **ConcurrentHashMap**: thread-safe, **không cho null**.
  + Đặc biệt: EnumMap (khóa là enum, rất nhanh), WeakHashMap (khóa yếu – tự dọn khi GC), IdentityHashMap (so sánh theo ==).

**Set**

* Tập hợp **không trùng lặp**.
* Các biến thể chính:
  + **HashSet**: trung bình O(1), không thứ tự.
  + **LinkedHashSet**: giữ **thứ tự chèn**.
  + **TreeSet**: có **thứ tự** (Comparator/Comparable), O(log n).
  + Đặc biệt: EnumSet (gọn nhẹ, nhanh), CopyOnWriteArraySet (đọc nhiều, ghi ít).

**2) Độ phức tạp (xấp xỉ)**

| **Cấu trúc** | **Thao tác chính** | **Trung bình** | **Ghi chú** |
| --- | --- | --- | --- |
| HashMap/HashSet | get/put/contains/remove | O(1) | đụng độ nặng có thể O(log n) (tree bin) |
| LinkedHashMap/LinkedHashSet | như Hash\* | O(1) | thêm chi phí liên kết thứ tự |
| TreeMap/TreeSet | put/get/contains/remove | O(log n) | có thứ tự, hỗ trợ range query |
| ConcurrentHashMap | get/put | ~O(1) | thread-safe, cấm null |

**3) Bẫy kinh điển: equals/hashCode & Comparator**

* **Khóa/Phần tử (key/element) trong Hash\*** phải có equals & hashCode **nhất quán**.
  + Nếu tự định nghĩa lớp làm **key**/**Set element**, **bắt buộc** override cả hai.
  + **Không** thay đổi **trạng thái ảnh hưởng equals/hashCode** sau khi đưa vào Map/Set (key **mutable** là mùi lỗi!).
* Với TreeMap/TreeSet:
  + Dùng Comparator hoặc Comparable. Comparator nên **nhất quán với equals** để tránh hành vi bất ngờ (ví dụ coi 2 phần tử “bằng” trong set → mất bớt phần tử).

**4) Ứng dụng của Map trong bài toán thường gặp**

**Tạo, cơ bản, mặc định**

Map<String, Integer> ages = new HashMap<>();

ages.put("An", 20);

ages.putIfAbsent("Bình", 21); // không ghi đè nếu đã có

int ageAn = ages.getOrDefault("An", -1);

boolean hasChi = ages.containsKey("Chi");

ages.remove("An"); // xóa theo key

ages.remove("Bình", 21); // xóa nếu (key,value) khớp

**Duyệt hiệu quả**

for (var e : ages.entrySet()) { // nhanh nhất

System.out.println(e.getKey() + " -> " + e.getValue());

}

**Nhóm dữ liệu (multimap) với computeIfAbsent**

Map<String, List<Integer>> scores = new HashMap<>();

void addScore(String name, int s) {

scores.computeIfAbsent(name, k -> new ArrayList<>()).add(s);

}

**Đếm tần suất / gộp dữ liệu**

Map<String,Integer> freq = new HashMap<>();

for (String w : List.of("a","b","a")) {

freq.merge(w, 1, Integer::sum); // nếu có thì cộng dồn

}

**Cập nhật có điều kiện**

ages.replace("An", 20, 21); // chỉ thay nếu đang là 20

ages.compute("An", (k,v) -> v == null ? 1 : v + 1);

**Map bất biến & view**

Map<String,Integer> imm = Map.of("a",1, "b",2); // unmodifiable

Map<String,Integer> cp = Map.copyOf(imm); // unmodifiable copy

// keySet/values/entrySet là VIEW back-by-map:

Set<String> keys = ages.keySet();

keys.remove("An"); // xóa luôn khỏi ages

**5) Ứng dụng của Set trong bài toán thường**

**Cơ bản**

Set<String> s = new HashSet<>();

s.add("A"); s.add("B"); s.add("A"); // chỉ còn A,B

s.remove("A");

boolean hasB = s.contains("B");

**Set bất biến**

Set<String> imm = Set.of("A","B","C"); // unmodifiable, không cho null

**Toán tử tập hợp (union, intersection, difference)**

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

Set<Integer> b = new HashSet<>(List.of(3,4));

Set<Integer> union = new HashSet<>(a); union.addAll(b); // {1,2,3,4}

Set<Integer> inter = new HashSet<>(a); inter.retainAll(b); // {3}

Set<Integer> diff = new HashSet<>(a); diff.removeAll(b); // {1,2}

**Duy trì thứ tự / sắp xếp**

Set<String> ordered = new LinkedHashSet<>(List.of("B","A","C")); // giữ thứ tự chèn

Set<String> sorted = new TreeSet<>(Comparator.comparingInt(String::length)

.thenComparing(Comparator.naturalOrder()));

sorted.addAll(List.of("aa","b","ccc"));

**Khử trùng lặp & giữ thứ tự chèn**

List<Integer> input = List.of(3,1,2,3,1,4);

List<Integer> dedup = new ArrayList<>(new LinkedHashSet<>(input)); // [3,1,2,4]

**Chọn EnumSet khi phần tử là enum**

enum Perm { READ, WRITE, EXEC }

EnumSet<Perm> p = EnumSet.of(Perm.READ, Perm.WRITE);

p.add(Perm.EXEC); p.remove(Perm.WRITE);

**6) Null, thứ tự, và hành vi khác biệt**

| **Cấu trúc** | **Null key** | **Null value** | **Thứ tự** |
| --- | --- | --- | --- |
| HashMap | **1** null key | nhiều null value | không |
| LinkedHashMap | như HashMap | như HashMap | **thứ tự chèn** / **truy cập** |
| TreeMap | **không** (với natural order) | cho phép | **có thứ tự** |
| ConcurrentHashMap | **không** | **không** | không |
| HashSet | n/a | n/a | không |
| LinkedHashSet | n/a | n/a | **thứ tự chèn** |
| TreeSet | n/a | n/a | **có thứ tự** |

Set dựa trên Map bên dưới (ví dụ HashSet dựa HashMap), nên quy tắc null/thứ tự tương tự loại Map tương ứng.

# **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.