Contents

[**1. Java Core Basic** 2](#_Toc207036132)

[**1.1. Primitive and Object data type** 2](#_Toc207036133)

[**1.2. String** 3](#_Toc207036134)

[**1.3. Static & Final** 6](#_Toc207036135)

[**1.4. OOP** 9](#_Toc207036136)

[**1.5. Memory** 13](#_Toc207036137)

[**1.6. Exception handle** 15](#_Toc207036138)

# **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

}

**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).