# **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 |