**Java Core Basic**

**1 Primitive và object data type**

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

**Tính chất**

**Primitive**

**Object**

Nguồn gốc

Các kiểu dữ liệu đc xác định trước.

Các kiểu dữ liệu do người dùng định nghĩa.

Lưu trữ

Được lưu trữ trong ngăn xếp

Biến được lưu trữ trong ngăn xếp, đối tượng gốc đc lưu trữ trong heap

Giá trị mặc định

Kh có giá trị mặc định là null

Giá trị mặc định khi biến tham chiếu là null

Tốc độ

Nhanh

Chậm(do quản lý object phức tạp)

Phương thức

Chỉ chứa dữ liệu không có phương thức.

Chứa dữ liệu và phương thức

VD: Integer b = 5;

System.out.println(b.toString());

Ví dụ

int, float, double, char ,byte, shot, long, boolean.

Array, interface,…

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

• Có thể chuyển đổi giữa 2 kiểu dữ liệu nguyên thuỷ và tham chiếu.

• Từ primitive sang object:

Boxing: gói giá trị từ primitive thành object thủ công

Autoboxing: Java tự chuyển đổi khi cần.

Ví dụ:

int a=0;

//Boxing

Integer b=Integer.valueOf(a);

//AutoBoxing

Integer c=a;

• Từ object sang primitive:

Unboxing: Lấy giá trị ra thủ công

Auto-unboxing: Java tự chuyển đổi khi gán hoặc dùng toán tử

Ví dụ:

Integer x = 10;

// Unboxing

int y1 = x.intValue();

//Auto-unboxing

int y2 = x;

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

• Có thể so sánh 2 kiểu dữ liệu này với nhau nhưng cần primitive và object thuộc wrapper tương ứng(int ⇔ Integer, double⇔ Double). Chúng có thể so sánh nhờ cơ chế autoboxing và auto-unboxing

• Có 3 trường hợp so sánh:

So sánh primitive và object bằng == : Java sẽ tự động unbox từ object thành primitive rồi so sánh địa chỉ.

VD:

int a = 5;

Integer b = 5;

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

So sánh object với object bằng == : nó sẽ so sánh địa chỉ ô nhớ chứ kh so sánh giá trị

VD:

Integer x = new Integer(5);

Integer y = new Integer(5);

System.out.println(x == y); // false (khác địa chỉ)

So sánh bằng equal() : Java sẽ so sánh giá trị vì các wrapper đã được override equal()

VD:

int a = 5;

Integer b = 5;

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

**2 String**

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

• String là 1 class chứ kh phải là kiểu dữ liệu nguyên thuỷ. Có nghĩa là có thể tạo các đối tượng String, gọi các phương thức dựa trên nó và nó có các phương thức và thuộc tính riêng

• String có tính bất biến, nghĩa là khi đối tượng String đc khởi tạo nó sẽ kh thể thay đổi đc nội dung. Các thao tác thay đổi chuỗi thực ra là tạo ra 1 đối tượng String mới trong bộ nhớ(vì vậy ví dụ nếu ta nối 100 chuỗi nó sẽ tạo ra cả 100 chuỗi mới làm giảm hiệu suất nên ta dùng string builder hoặc string buffer để chỉ tạo 1 đối tượng cuối cùng giúp tiết kiệm bộ nhớ và tăng tốc độ xử lý).

VD:

String s = "Hello";

s.concat(" World"); // không thay đổi s, mà tạo object mới "Hello World"

System.out.println(s); // "Hello"

• String pool: Để tối ưu bộ nhớ, java có 1 vùng gọi là String Pool. Khi tạo 1 String literal (ví dụ như String s=”s”), JVM sẽ kiểm tra xem chuỗi đó đã tồn tại trong pool hay chưa, nếu có nó sẽ trả về tham chiếu đến đối tượng đã có, nếu chưa nó sẽ tạo 1 đối tượng mới trong pool. Nhưng nếu ta dùng new thì sẽ tạo object mới trong heap chứ kh dùng pool.

VD:

// Cả s1 và s2 đều tham chiếu đến cùng một đối tượng "Java" trong String Pool String s1 = "Java";

String s2 = "Java";

// So sánh tham chiếu bằng toán tử '=='

System.out.println("s1 == s2? " + (s1 == s2)); // Output: true

// Tạo đối tượng String mới bằng từ khóa new

String s3 = new String("Java");

// So sánh s1 với s3, chúng có nội dung giống nhau nhưng tham chiếu khác nhau

System.out.println("s1 == s3? " + (s1 == s3)); // Output: false

• So sánh chuỗi: Sử dụng toán tử == để so sánh tham chiếu(xem 2 biến có cùng trỏ đến 1 đối tượng trong bộ nhớ kh). Sử dụng phương thức equal để so sánh nội dung của 2 chuỗi.

• Các phương thức hữu ích của String: lớp String cung cấp nhiều phương thức để thao tác với chuỗi, gồm:

length(): trả về độ dài của chuỗi.

charAt(int index): lấy ký tự tại vị trí chỉ định.

substring(int beginIndex, int endIndex): lấy 1 chuỗi con.

toUpperCase(), toLowerCase(): đổi chữ hoa/ thường.

equal(): so sánh nội dung chuỗi.

contains(): kiểm tra xem chuỗi có chứa 1 chuỗi con khác kh.

concat(): nối thêm chuỗi khác vào cuối.

replace(): thay thế ký tự cũ bằng ký tự mới.

**Đặc điểm**

**String**

**String builder**

**String buffer**

Tính biến đổi

Tính bất biến

Có thể thay đổi

Có thể thay đổi

Độ an toàn

An toàn luồng

kh an toàn luồng

An toàn luồng

Hiệu suất

Kém hiệu quả với nhiều thao tác

Hiệu quả nhất

Kém hiệu quả hơn String builder

Trường hợp sử dụng

Khi chuỗi kh thay đổi

Khi cần thao tác trong môi trường đơn luồng

Khi cần thao tác trong môi trường đa luồng

**Có bao nhiêu cách để tạo 1 biến String?**

• Có 3 cách để tạo 1 biến String trong java:

Cách 1: Sử dụng String literal: khi tạo 1 String literal nó sẽ kiểm tra trong String pool, nếu chuỗi đã tồn tại nó trả về tham chiếu đến đối tượng đó, nếu chưa có nó tạo mới đối tượng String trong String pool.

VD: String s1=”chuỗi 1”;

Cách 2: Sử dụng từ khoá new: Nó sẽ tạo ra 1 đối tượng mới trên heap dù nội dung đó đã tồn tại trong String pool hay chưa

VD: String s2= new String(“Chuỗi 2”);

Cách 3: Sử dụng mảng kí tự: Tạo 1 đối tượng String từ 1 mảng ký tự(char array), Sử dụng khi muốn xây dựng chuỗi từ các ký tự riêng lẻ

VD: char[] charArray={“c”,”h”,”a”,”o”,”b”,”a”,”n”};

String s3=new String(charArray);

**Tìm hiểu về String pool?**

• String pool là 1 vùng nhớ đặc biệt trong Heap của java. Nó dùng để lưu trữ các đối tượng String duy nhất. Khi tạo 1 chuỗi bằng cách sử dụng String iteral, JVM sẽ kiểm tra xem chuỗi đã tồn tại trong String pool hay chưa, Nếu có nó trả về tham chiếu đến đối tượng hiện có thay vì tạo đối tượng mới giúp tiết kiệm bộ nhớ hơn, nếu chưa có nó sẽ tạo mới đối tượng trong String pool.

• Cách hoạt động:

Tạo String iteral : Khi khai báo String s1=”Java”, JVM sẽ tìm kiếm “Java” trong String pool. Vì đây là lần đầu tiên, nó sẽ tạo đối tượng “Java” trong String pool và gán địa chỉ của nó cho s1.

Tham chiếu đến chuỗi đã có: Sau đó, nếu khai báo String s2=”Java”, JVM sẽ lại tìm kiếm trong String pool , sẽ thấy “Java” đã tồn tại và gán địa chỉ của nó cho s2. Vì vậy s1 và s2 sẽ trỏ đến cùng 1 đối tượng.

Tạo đối tượng mới bằng new: Khi dùng String s3 =new String(“Java”), 1 đối tượng mới hoàn toàn sẽ đc tạo ra trên heap, nằm ngoài String pool, mặc dù nội dung của nó cx là “Java” nhưng nó có địa chỉ bộ nhớ khác so với đối tượng “Java” trong String pool.

VD:

String s1 = "Hello";

String s2 = "Hello";

String s3 = new String("Hello");

System.out.println(s1 == s2); // true (cùng trỏ pool)

System.out.println(s1 == s3); // false (s3 là object mới trong heap)

System.out.println(s1.equals(s3)); // true (so sánh nội dung)

• Lợi ích: String pool là 1 ví dụ về việc java tối ưu hoá bộ nhớ. Bằng cách tái sử dụng chuỗi thay vì tạo ra các chuỗi cùng 1 giá trị, String pool giúp tiết kiệm bộ nhớ hơn.

**Làm sao để so sánh hai chuỗi trong java?**

• Sử dụng toán tử ==: sử dụng để so sánh tham chiếu của 2 đối tượng, nó kiểm tra xem 2 biến có trỏ đến cùng 1 đối tượng trong bộ nhớ kh.

String s1 = "Hello";

String s2 = "Hello";

String s3 = new String("Hello");

System.out.println(s1 == s2); // true (cùng String Pool)

System.out.println(s1 == s3); // false (s3 tạo object mới)

• Sử dụng phương thức equal(): equal() đã đc override lại trong lớp String dùng để so sánh nội dung chuỗi.

VD:

String s1 = "Hello";

String s2 = new String("Hello");

System.out.println(s1.equals(s2)); // true (nội dung giống nhau)

**3 Static & final**

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

• Từ khóa static được sử dụng để xác định rằng thành viên đó thuộc về lớp chứ kh phải thuộc về 1 đối tượng cụ thể của lớp đó. Điều này có nghĩa là các thành viên static có thể được truy cập mà kh cần phải khởi tạo đối tượng

• Thuộc tính static đc sd khi cần 1 giá trị chung, đc chia sẻ bởi tất cả các đối tượng của lớp.

VD:

  class Counter {

    static int count = 0; // biến static

    Counter() {

        count++; // mỗi lần tạo object -> tăng count

    }

}

public class Main {

    public static void main(String[] args) {

        new Counter();

        new Counter();

        System.out.println(Counter.count); // 2 (dùng chung cho tất cả object)

    }

}

• Phương thức static đc sd khi logic kh phụ thuộc vào object, chỉ làm việc với với dữ liệu chung hoặc dữ liệu truyền vào

VD:

class MathUtil {

    // phương thức static: không phụ thuộc object nào

    static int square(int x) {

        return x \* x;

    }

}

public class Main {

    public static void main(String[] args) {

        // gọi trực tiếp bằng tên class

        System.out.println(MathUtil.square(5)); // 25

        System.out.println(MathUtil.square(10)); // 100

    }

}

• Cách truy cập thuộc tính và phương thức static:

Sử dụng tên class: Tên lớp.Tên thuộc tính(hoặc tên phương thức());

Sử dụng tên đối tượng: Tên đối tượng.Tên thuộc tính(hoặc tên phương thức());

**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 trong java có nghĩa là kh thể thay đổi đc, nó đc sử dụng để hạn chế các biến, phương thức, lớp. Biến final( khi đc gán giá trị nó sẽ kh thể thay đổi đc), phương thức final(phương thức này sẽ kh thể override lại trong các lớp con), lớp final(class đc đánh dấu là final sẽ kh thể kế thừa đc).

• Khác nhau giữa static và final:

Đặc điểm

static

final

Ý nghĩa

Thuộc về lớp, kh thuộc về đối tượng

Kh thể thay đổi đc

Phạm vi

Chỉ áp dụng cho biến, phương thức và khối mã

Có thể áp dụng cho biến, phương thức và lớp

Thay đổi giá trị

Biến static có thể thay đổi giá trị

Biến final kh thể thay đổi sau khi đc khởi tạo

Bộ nhớ

Chỉ có 1 bản sao duy nhất, dùng chung cho tất cả các object

Tồn tại riêng trong từng object, nhưng giá trị cố định

• Biến khai báo bằng final kh thể chỉnh sửa đc khi đã đc khởi tạo.

**4 OOP**

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

• Có 4 tính chất quan trọng:

Tính đóng gói: là giấu thông tin của đối tượng, kiểm soát quyền truy cập vào các thành phần đó, đảm bảo dữ liệu kh đc thay đổi 1 cách trái phép, chỉ cho phép truy cập thông qua các phương thức công khai(getter, setter).

Tính kế thừa: Cho phép lớp con kế thừa tất cả các phương thức và thuộc tính của lớp cha, điều này giúp tái sử dụng mã nguồn

Tính đa hình: Cho phép 1 phương thức có thể có nhiều cách triển khai khác nhau. Có 2 loại đa hình là overriding(định nghĩa lại phương thức của lớp cha) và overloading(có cùng tên phương thức nhưng khác tham số)

Tính trừu tượng: là khả năng ẩn giấu đi các thông tin chi tiết  phức tạp, kh cần thiết của 1 đối tượng và chỉ tập trung vào những thông tin quan trọng mà người dùng cần

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

• Access modifier trong java có 4 loại: private, default, protected, public

• Đặc điểm:

Private: chỉ cho phép truy cập  bên trong lớp đó.

Default: chỉ cho phép truy cập trong cùng 1 package.

Protected: chỉ cho phép truy cập trong cùng 1 package và có thể truy cập từ package khác nếu đc kế thừa.

Public: Cho phép truy cập từ bất kì đâu trong chương trình.

**Phân biệt class và instance?**

**Đặc điểm**

**Class**

**Instance**

Khái niệm

là 1 khuôn mẫu hoặc bản thiết kế

là thực thể cụ thể của bản thiết kế

Bộ nhớ

Tồn tại trong bộ nhớ khi chương trình chạy, kh chiếm kh gian cho dữ liệu cụ thể của đối tượng

Tồn tại trong bộ nhớ heap, chứa dữ liệu cụ thể cho các thuộc tính của nó

Cách tạo

đc định nghĩa bằng từ khoá class

đc tạo bằng từ khoá new

Ví dụ

class NV{};

NV nv1=new NV();

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

**Đặc điểm**

**Abstract**

**Interface**

Tính kế thừa

1 class chỉ có thể kế thừa duy nhất 1 abstract class

1 lớp có thể triển khai nhiều interface

Thành phần

Có thể chứa phương thức abstract và non-abstract, thuộc tính và các modifier(private, protected, public)

Từ java 8 trở đi có thể có phương thức abstract, default và static(đều có thân)

constructor

có constructor

kh có constructor

Access modifier

Có thể dùng private, protected, public cho field và method

Các field mặc định là public static final(hằng số).

Mục đích

Đc sd để cung cấp 1 lớp cơ sở chung cho các lớp con có nhiều điểm chung và cần tái sử dụng

Đc sd để định nghĩa các hành vi mà các lớp khác nhau có thể tuân theo, bất kể chúng thuộc hệ thống phân cấp nào.

• Trường hợp sử dụng abstract class: Khi muốn tạo khung xương chung cho các class con.

Ví dụ:

abstract class Animal {

    String name;

    int age;

    Animal(String name, int age) {

        this.name = name;

        this.age = age;

    }

    abstract void makeSound();

}

class Dog extends Animal {

    Dog(String name, int age) {

        super(name, age);

    }

    @Override

    void makeSound() {

        System.out.println("gau gau");

    }

}

• Trường hợp sử dụng interface: dùng để định nghĩa hành vi chung mà nhiều lớp khác nhau có thể thực hiện đc.

Ví dụ:

interface Flyable {

    void fly();

}

class Bird implements Flyable {

    @Override

    public void fly() {

        System.out.println("Bird flies with wings");

    }

}

class Airplane implements Flyable {

    @Override

    public void fly() {

        System.out.println("Airplane flies with engine");

    }

}

• Trường hợp 2 interface hoặc 1 abstract và 1 interface có cùng tên phương thức và cùng kiểu trả về: Kh gây xung đột, lớp con chỉ cần triển khai 1 lần

• Trường hợp 2 interface hoặc 1 abstract và 1 interface có cùng tên phương thức và khác kiểu trả về: Lỗi biên dịch do java kh phân biệt đc phương thức nào cần gọi .

**Thế nào là Overriding và Overloading?**

• Overriding là việc 1 lớp con cung cấp 1 cách triển khai cụ thể cho 1 phương thức đã đc định nghĩa trong lớp cha. Điều này cho phép lớp con có hành vi khác so với lớp cha của nó, trong khi vẫn giữ nguyên tên và các tham số của phương thức.

Ví dụ:

class Animal {

    public void makeSound() {

        System.out.println("The animal makes a sound.");

    }

}

class Dog extends Animal {

    @Override

    public void makeSound() {

        System.out.println("The dog barks.");

    }

}

• Overloading là việc 1 lớp có nhiều phương thức cùng tên nhưng khác nhau về tham số. Điều này cho phép thực hiện hành động tương tự trên các loại dữ liệu khác nhau hoặc số lượng tham số khác nhau.

Ví dụ:

class MathOperations {

    // Nạp chồng phương thức add()

    public int add(int a, int b) {

        return a + b;

    }

    public double add(double a, double b) {

        return a + b;

    }

    public int add(int a, int b, int c) {

        return a + b + c;

    }

}

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

• function có access modifier là private hoặc static kh thể overriding đc.

• Vì nếu access modifier là private thì lớp con kh thể nhìn thấy và kế thừa đc phương thức này nên kh thể ghi đè nó.

• Static là phương thức thuộc về lớp kh thuộc về đối tượng. Mà overriding là 1 khái niệm liên quan đến đa hình, nơi các đối tượng con cung cấp cách triển khai cho 1 phương thức của lớp cha nên nó kh liên quan đến khái niệm đa hình dựa trên đối tượng

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

• Một phương thức được khai báo là final có thể kế thừa được, nhưng không thể bị ghi đè (override) bởi các lớp con.

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

**Đặc điểm**

**This**

**Super**

Khái niệm

Từ khoá this để chỉ đối tượng hiện tại, sử dụng this để truy cập các phương thức và thuộc tính của chính đối tượng mà bạn đang làm việc

Từ khoá super dùng để chỉ lớp cha của dối tượng hiện tại, Nó cho phép truy cập các phương thức thuộc tính của lớp cha.

Mục đích sử dụng

Phân biệt biến cục bộ và biến instance: Khi tên của một tham số trùng với tên của một thuộc tính, this giúp phân biệt chúng.

Truy cập phương thức bị ghi đè: Khi một lớp con ghi đè một phương thức của lớp cha, bạn có thể sử dụng super để gọi phương thức gốc của lớp cha.

Constructor

Gọi một constructor khác của cùng một lớp.

Gọi constructor của lớp cha.

**5 Memory**

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

• Cấp phát tĩnh là quá trình phân bổ bộ nhớ cho một biến hoặc một mảng tại thời điểm biên dịch. Kích thước của bộ nhớ cần thiết phải được biết trước và không thể thay đổi trong suốt quá trình thực thi chương trình.

Đặc điểm:

• Thời điểm: Tại thời điểm biên dịch.

• Kích thước: Cố định và không thay đổi.

• Vị trí: Bộ nhớ được cấp phát trong vùng stack (ngăn xếp) hoặc vùng dữ liệu tĩnh của chương trình.

• Tốc độ: Nhanh, vì hệ thống đã biết trước kích thước và vị trí.

• Cấp phát động là quá trình phân bổ bộ nhớ cho một biến hoặc một đối tượng tại thời điểm thực thi (run-time). Kích thước của bộ nhớ có thể thay đổi trong quá trình chạy chương trình.

Đặc điểm:

• Thời điểm: Tại thời điểm thực thi.

• Kích thước: Linh hoạt, có thể thay đổi tùy thuộc vào nhu cầu của chương trình.

• Vị trí: Bộ nhớ được cấp phát trong vùng heap.

• Tốc độ: Chậm hơn cấp phát tĩnh, vì phải tìm kiếm vùng bộ nhớ trống.

**Phân biệt bộ nhớ heap và bộ nhớ stack ?**

• Stack là một vùng bộ nhớ được sử dụng để lưu trữ các giá trị kiểu dữ liệu nguyên thủy và các tham chiếu đến đối tượng. Nó hoạt động theo nguyên tắc LIFO (Last-In, First-Out), tức là phần tử được thêm vào cuối cùng sẽ được lấy ra đầu tiên.

Thời gian sống: Các biến trên stack chỉ tồn tại trong suốt thời gian thực thi của một phương thức. Khi phương thức kết thúc, các biến này sẽ tự động bị hủy.

Tốc độ truy cập: Rất nhanh, vì trình biên dịch biết rõ kích thước và vị trí của các biến này.

Kích thước: Cố định, thường nhỏ hơn heap. Khi stack đầy, sẽ xảy ra lỗi StackOverflowError.

• Heap là một vùng bộ nhớ được sử dụng để lưu trữ tất cả các đối tượng và mảng. Đây là vùng bộ nhớ linh hoạt và lớn hơn nhiều so với stack. Dữ liệu trong heap được quản lý bởi Garbage Collector của Java.

Thời gian sống: Các đối tượng trên heap tồn tại cho đến khi không còn tham chiếu nào trỏ tới chúng nữa. Sau đó, chúng sẽ được Garbage Collector dọn dẹp để giải phóng bộ nhớ.

Tốc độ truy cập: Chậm hơn stack, vì việc tìm kiếm và cấp phát bộ nhớ cần nhiều thao tác hơn.

Kích thước: Linh hoạt, có thể tăng hoặc giảm trong quá trình thực thi. Khi heap đầy, sẽ xảy ra lỗi OutOfMemoryError.

**6 Handle Exception**

**Phân biệt throw và throws?**

**Đặc điểm**

**Throw**

**Throws**

Mục đích

Để ném một ngoại lệ cụ thể, thường là khi một điều kiện lỗi xảy ra trong chương trình.

Báo cho người sử dụng phương thức biết rằng họ cần phải xử lý (bằng try-catch) hoặc tiếp tục khai báo (bằng throws) ngoại lệ đó.

Vị trí

Bên trong thân phương thức.

Trong chữ ký của phương thức.

Số lượng

Chỉ có thể ném một ngoại lệ tại một thời điểm.

Có thể khai báo nhiều ngoại lệ, ngăn cách bằng dấu phẩy.

Cú pháp

throw new TênException();

public void tenPhuongThuc() throws TênException { ... }

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

**Đặc điểm**

**Checked exception**

**Unchecked exception**

Khái niệm

là những ngoại lệ mà trình biên dịch Java bắt buộc bạn phải xử lý. Chúng thường đại diện cho những lỗi có thể xảy ra ngoài tầm kiểm soát của chương trình, nhưng có thể lường trước được.

là những ngoại lệ mà trình biên dịch không bắt buộc bạn phải xử lý. Chúng thường xảy ra do lỗi logic trong lập trình, chẳng hạn như truy cập một chỉ số không hợp lệ của mảng. Chúng ta có thể ngăn chặn những lỗi này bằng cách viết mã cẩn thận hơn.

Thời điểm kiểm tra

Tại thời điểm biên dịch

Tại thời điểm thực thi

Cách xử lý

Bắt buộc phải xử lý chúng bằng khối try-catch hoặc khai báo bằng từ khóa throws trong chữ ký phương thức. Nếu không làm vậy, chương trình sẽ không thể biên dịch.

Không bị bắt buộc phải xử lý chúng. Tuy nhiên, nếu bạn không xử lý, chương trình sẽ bị dừng và báo lỗi.

Ví dụ

IOException (lỗi nhập/xuất file), SQLException (lỗi truy cập cơ sở dữ liệu), FileNotFoundException.

NullPointerException (truy cập đối tượng rỗng), ArrayIndexOutOfBoundsException (truy cập chỉ số ngoài giới hạn), IllegalArgumentException (tham số không hợp lệ).

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

**Đặc điểm**

**try catch**

**try with resource**

Quản lý tài nguyên

Thủ công trong khối try-catch-finally

Tự động

Cú pháp

Cần thêm khối finally để đóng tài nguyên

Tài nguyên được khai báo trong ngoặc đơn () của try

Độ an toàn

Có nguy cơ rò rỉ tài nguyên

An toàn hơn, đảm bảo tài nguyên được đóng

Tính tương thích

Các phiên bản Java cũ

Java 7 trở lên

**Làm thế nào để tạo được 1 custom exception ?**

• Để tạo 1 custom exception thì cần tạo 1 lớp kế thừa từ 1 trong 2 lớp Exception hoặc RuntimeException.

• Tạo constructor để truyền thông báo lỗi

• Có thể thêm field riêng để chứa thông tin chi tiết

**Java Collection**

**1 Array**

**Nêu hiểu biết về kiểu mảng trong java**

• Mảng là 1 cấu trúc dữ liệu để lưu trữ tập hợp các giá trị có cùng kiểu dữ liệu trong 1 biến duy nhất.

• Đặc điểm:

• Đồng nhất kiểu dữ liệu: Tất cả các phần tử trong mảng phải có cùng kiểu dữ liệu, ví dụ: một mảng int chỉ có thể chứa các số nguyên.

• Kích thước cố định: Kích thước của mảng được xác định khi nó được tạo và không thể thay đổi sau đó.

• Chỉ mục(index): Các phần tử của mảng được truy cập thông qua một chỉ mục số nguyên, bắt đầu từ 0.

• Khai báo và khởi tạo mảng: có 3 cách phổ biến

• Khai báo và khởi tạo với kích thước

VD: int[] arr=new int[5];

• Khai báo và khởi tạo trực tiếp

VD:int[] arr={“a”,”b”,”c”};

• Object array

VD: String[] names = new String[]{"Alice", "Bob", "Charlie"};

• Mảng đa chiều: thực chất là mảng của mảng

VD:

int[][] matrix = {

    {1, 2, 3},

    {4, 5, 6},

    {7, 8, 9}

};

System.out.println(matrix[1][2]); // 6

• Vòng lặp với mảng:sử dụng vòng lặp for và for-each

• Vòng lặp for:

VD: for (int i = 0; i < numbers.length; i++) {

    System.out.println(numbers[i]);

}

• Vòng lặp for-each:

VD:for (String name : names) {

    System.out.println(name);

}

**Dùng kiểu mảng mang lại ưu , nhược điểm gì?**

• Ưu điểm:

• Truy cập phần tử nhanh nhờ chỉ số (O(1)).

• Lưu trữ dữ liệu cùng kiểu gọn gàng, dễ quản lý

• Nhược điểm:

• Kích thước cố định: gây ra lãng phí bộ nhớ nếu bạn cấp phát quá nhiều không gian, hoặc thiếu không gian nếu bạn cần lưu trữ thêm phần tử.

• Kh linh hoạt: thêm hoặc xóa phần tử ở giữa tốn kém vì cần dịch chuyển thủ công các phần tử để duy trì thứ tự.

**2 List Interface**

**Nêu ra các đặc điểm List Interface?**

• List là 1 interface trong java collections framework, nó đc sd để định nghĩa các phần tử có thứ tự và nó cho phép phần tử trùng lặp

• Đặc điểm:

• Có thứ tự: phần tử được lưu trữ theo thứ tự chèn vào, có thể truy cập phần tử theo chỉ số

• Cho phép phần tử trùng lặp: có thể thêm nhiều phần tử có cùng 1 giá trị vào list

• Có chỉ số: có thể truy cập trực tiếp get(index), set(index, value).

• Cung cấp nhiều phương thức đặc trưng:

add(int index, E element) → chèn phần tử vào vị trí chỉ định.  
 remove(int index) → xóa phần tử theo vị trí.

get(int index) → lấy phần tử theo chỉ số.  
 set(int index, E element) → cập nhật phần tử tại chỉ số.

indexOf(Object o) → trả về chỉ số xuất hiện đầu tiên.

lastIndexOf(Object o) → trả về chỉ số xuất hiện cuối cùng.  
 subList(int from, int to) → lấy danh sách con.

• Triển khai phổ biến:

ArrayList → mảng động, truy cập nhanh, thêm/xóa chậm.  
 LinkedList → danh sách liên kết, thêm/xóa nhanh, truy cập chậm.

Vector (cũ, synchronized).

Stack (kế thừa từ Vector).

**Kể ra các class triển khai từ List Interface?**

• ArrayList

• LinkedList

• Vector

• Stack

**Phân biệt rõ trường hợp sử dụng của từng class đó?**

• ArrayList(tự động tăng gấp đôi capacity khi đầy → đây là lý do nó gọi là mảng động.)

• cấu trúc: mảng động

• Ưu điểm:

• Truy cập phần tử nhanh

• Duyệt mảng bằng vòng lặp nhanh

• Nhược điểm:thêm và xóa phần tử ở giữa chậm vì phải dịch chuyển các phần tử ở phía sau

• Dùng khi:

• Khi cần truy cập nhiều bằng chỉ số

• Danh sách có số lượng thay đổi vừa phải, không thường xuyên thêm/xóa giữa danh sách.

• LinkedList

• Cấu trúc: danh sách liên kết kép

• Ưu điểm:

• Thêm/xoá ở đầu và cuối nhanh

• Thêm/xóa ở giữa cũng không cần dời phần tử, chỉ cần thay đổi con trỏ liên kết.

• Nhược điểm:

• Truy cập ngẫu nhiên chậm O(n) vì phải duyệt từ đầu.

• Dùng khi:

• Khi thao tác thêm/xóa nhiều (đặc biệt ở đầu/cuối).

• Khi không cần truy cập nhiều bằng chỉ số.

• Vector

• Cấu trúc: giống ArrayList nhưng synchronized(an toàn luồng)

• Ưu điểm: Hữu ích khi làm việc trong môi trường đa luồng.

• Nhược điểm: Đồng bộ hóa làm giảm hiệu năng.

• Dùng khi: cần một danh sách an toàn luồng (thread-safe). Tất cả các phương thức của nó đều được đồng bộ hóa (synchronized), đảm bảo rằng chỉ có một luồng có thể truy cập danh sách tại một thời điểm

• Stack

• là subclass của Vector

• Tuân thủ theo nguyên tắc LIFO(Last In First Out)

• Có thêm phương thức:

• push() → thêm phần tử.

• pop() → lấy và xóa phần tử cuối.

• peek() → xem phần tử cuối mà không xóa.

• Dùng khi: Khi cần cấu trúc dữ liệu ngăn xếp: ví dụ undo/redo, duyệt cây/đồ thị (DFS), quản lý lời gọi hàm (call stack).

**3 Queue Interface**

**Nêu các đặc điểm của Queue Interface, Dequeue Interface?**

• Queue: là 1 interface trong java collections framework, đại diện cho một danh sách các phần tử được sắp xếp theo thứ tự FIFO (First-In, First-Out), nó giống như một hàng người xếp hàng chờ đợi, ai đến trước thì được phục vụ trước.

Đặc điểm chính:

• FIFO (First-In, First-Out): vào trước ra trước

• Không cho phép truy cập ngẫu nhiên bằng chỉ số.

• Phương thức đặc trưng (có 2 nhóm, khác nhau ở cách xử lý khi queue rỗng/đầy):

add(e) / offer(e) → thêm phần tử.

remove() / poll() → xóa và trả về phần tử đầu.

element() / peek() → xem phần tử đầu mà không xóa.

Sự khác nhau:

add(), remove(), element() ném exception nếu lỗi.

offer(), poll(), peek() trả về giá trị đặc biệt (false hoặc null).

• Triển khai phổ biến:

LinkedList (vừa là List vừa là Queue).

PriorityQueue (ưu tiên theo thứ tự tự nhiên hoặc Comparator).

ArrayDeque (một dạng deque nhưng có thể dùng như queue).

• Dequeue: là một interface mở rộng từ Queue, cung cấp các chức năng để làm việc với cả hai đầu của hàng đợi. Tên gọi Double-EndedQueue thể hiện rõ điều này. Bạn có thể thêm và xóa các phần tử từ cả đầu và cuối của Deque.

Đặc điểm chính:

• Có thể dùng như Queue (FIFO) hoặc như Stack (LIFO).

• Phương thức đặc trưng (cho cả hai đầu):

• Thêm vào đầu: addFirst(), offerFirst()

• Thêm vào cuối: addLast(), offerLast()

• Lấy từ đầu: removeFirst(), pollFirst()

• Lấy từ cuối: removeLast(), pollLast()

• Xem phần tử: peekFirst(), peekLast()

• Tính linh hoạt: Deque là một lựa chọn tốt khi bạn cần một cấu trúc dữ liệu linh hoạt, có thể hoạt động như một ngăn xếp (sử dụng push() và pop()) hoặc một hàng đợi.

• Triển khai phổ biến:

• ArrayDeque (mảng vòng, nhanh hơn Stack và LinkedList).

• LinkedList (danh sách liên kết kép).

**Kể ra các class triển khai từ Queue Interface, Dequeue Interface , phân biệt trường hợp sử dụng tương ứng?**

• Các lớp triển khai từ queue:

• LinkedList: Thường dùng làm Queue khi cần thao tác thêm/xóa nhanh chóng và không làm việc trong môi trường đa luồng. Ví dụ: hàng đợi xử lý tác vụ trong một ứng dụng đơn luồng.

• PriorityQueue: Dùng khi cần xử lý phần tử theo độ ưu tiên thay vì thứ tự thêm vào (ví dụ: xử lý sự kiện, hàng đợi bệnh nhân ưu tiên nặng hơn ra trước).

• Các lớp triển khai từ dequeue:

• LinkedList: Thường dùng làm Deque khi cần thao tác ở cả hai đầu và muốn có các phương thức tiện lợi.

• ArrayDeque: Lựa chọn tốt nhất khi bạn cần một Queue hoặc Stack trong môi trường đơn luồng. Ví dụ: dùng làm Stack để duyệt cây, hoặc làm Queue để lưu trữ các yêu cầu.

**4 Set Interface**

**Nêu ra các đặc điểm Set Interface?**

• Set là 1 interface trong java collections framework, đại diện cho tập hợp các phần tử, set kh cho phép các phần tử trùng lặp và kh đảm bảo thứ tự các phần tử

• Đặc điểm:

• Kh cho phép các phần tử trùng lặp: nếu phần tử đã tồn tại, thao tác này sẽ bị loại bỏ và phương thức add sẽ trả về false

• kh có thứ tự: Các phần tử trong 1 Set không được sắp xếp theo một thứ tự cụ thể. Bạn không thể truy cập các phần tử bằng chỉ số (index) như trong List và thứ tự các phần tử khi lặp qua không được đảm bảo.

• Thao tác dựa trên giá trị : Các thao tác của Set như thêm, xóa, hay kiểm tra sự tồn tại của một phần tử đều dựa trên giá trị của nó, không phải vị trí.

• Các phương thức quan trọng:

• false nếu phần tử đã tồn tại.

• remove(Object o): Xóa một phần tử khỏi Set.

• contains(Object o): Kiểm tra xem Set có chứa phần tử đã cho hay không.

• size(): Trả về số phần tử hiện có trong Set.

• Các lớp triển khai phổ biến:

• HashSet

• LinkedHashSet

• TreeSet

**Kể ra các class triển khai từ Set Interface?**

• HashSet

• Dựa trên HashMap.

• Không đảm bảo thứ tự phần tử.

• Thao tác trung bình O(1)(thêm, xóa, kiểm tra), rất nhanh.

• ​​Dùng khi chỉ cần tập hợp duy nhất, không quan tâm thứ tự.

• LinkedHashSet

• Kế thừa từ HashSet.

• Duy trì thứ tự chèn phần tử.

• Nhanh gần như HashSet, nhưng tốn thêm bộ nhớ để lưu thứ tự.

• Dùng khi muốn unique + giữ nguyên thứ tự thêm vào.

• TreeSet

• Cài đặt dựa trên TreeMap (Red-Black Tree)

• Duy trì thứ tự sắp xếp tự nhiên (hoặc theo Comparator).

• Thao tác chậm hơn HashSet (O(log n)).

• Dùng khi muốn unique + sắp xếp tự động.

• EnumSet

• CopyOnWriteArraySet

• ConcurrentSkipListSet

**Phân biệt rõ trường hợp sử dụng của từng class đó?**

• HashSet:

• Đặc điểm:

• Cấu trúc dữ liệu dựa trên hash table

• kh duy trì thứ tự phần tử

• kh cho phép phần tử trùng lặp

• cho phép null

• Hiệu năng:Thêm, xóa, tìm kiếm:O(1) trung bình

• Khi nào dùng:

• Khi kh quan tâm thứ tự phần tử

• Cần thao tác tìm kiếm nhanh

• VD:

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

set.add("A");

set.add("B");

• LinkedHashSet:

• Đặc điểm:

• Kế thừa từ HashSet nhưng duy trì thứ tự chèn

• Cho phép null

• Hiệu năng: Thêm, xóa , tìm kiếm:O(1) trung bình(nhưng chậm hơn hashSet do phải duy trì danh sách liên kết)

• Khi nào dùng: Khi bạn muốn duy trì thứ tự chèn nhưng vẫn cần hiệu năng tìm kiếm tốt

• VD:

Set<String> set = new LinkedHashSet<>();

set.add("A");

set.add("B"); // iteration order: A, B

• TreeSet:

• Đặc điểm:

• Cấu trúc dữ liệu: Red-Black Tree(cây cân bằng)

• Sắp xếp tự nhiên hoặc theo comparator

• kh cho phép null

• Hiệu năng: Thêm, xóa, tìm kiếm: O(log n)

• Khi nào dùng:

• Khi bạn muốn duy trì phần tử theo thứ tự sắp xếp.

• Thích hợp cho các phép toán range queries (subSet, headSet, tailSet).

• VD:

Set<Integer> set = new TreeSet<>();

set.add(3);

set.add(1); // iteration order: 1, 3

• EnumSet:

• Đặc điểm:

• Chỉ dùng cho enum types

• Rất nhanh, hiệu quả bộ nhớ, dùng bit vector.

• Kh cho phép null

• Hiệu năng: O(1) với bộ enum nhỏ

• Khi nào dùng:

• Khi muốn tạo tập hợp từ enum constants

• Khi cần hiệu năng cao và tiết kiệm bộ nhớ

• VD:

enum Day { MON, TUE, WED }

Set<Day> set = EnumSet.of(Day.MON, Day.WED);

• CopyOnWriteArraySet:

• Đặc điểm:

• Thread-safe (synchronized) dựa trên CopyOnWriteArrayList.

• Mỗi lần thay đổi tạo một bản sao mới.

• Hiệu năng:

• Thêm, xóa: O(n) (do copy toàn bộ mảng).

• Đọc nhanh, thread-safe.

• Khi nào dùng: Trong đa luồng, khi thường xuyên đọc và hiếm khi ghi.

• VD:

Set<String> set = new CopyOnWriteArraySet<>();

• ConcurrentSkipListSet:

• Đặc điểm:

• Thread-safe, sắp xếp tự nhiên hoặc theo Comparator.

• Dựa trên ConcurrentSkipListMap.

• Hiệu năng: Thêm, xóa, tìm kiếm: O(log n), thread-safe.

• Khi nào dùng:

• Khi cần tập hợp sắp xếp trong môi trường đa luồng.

• Khi cần phép toán range queries trong multi-threading.

• VD:

Set<Integer> set = new ConcurrentSkipListSet<>();

**1 Map Interface**

Map là một interface trong Java Collections Framework, được sử dụng để lưu trữ các cặp khóa-giá trị (key-value). Không giống như List hay Set, Map không phải là một tập hợp các phần tử riêng lẻ, mà là một ánh xạ từ các khóa duy nhất đến các giá trị tương ứng.

**Nêu ra các đặc điểm Map Interface?**

• Key-Value Pair: Map lưu trữ dữ liệu dưới dạng cặp (key, value).

• Key duy nhất: Mỗi key trong Map là duy nhất, value có thể trùng.

• Value có thể null (tùy class), key có thể null (tùy class).

• Các thao tác cơ bản: put(key,value), get(key), remove(key), containsKey(), containsValue(), size().

**Kể ra các class triển khai từ Map Interface?**

• HashMap

• LinkedHashMap

• TreeMap

• Hashtable

• ConcurrentHashMap

• EnumMap

**Phân biệt rõ trường hợp sử dụng của từng class đó?**

• HashMap:

• Kh duy trì thứ tự key

• Cho phép 1 null key và nhiều null value

• Thêm, xóa, tìm kiếm nhanh

• LinkedHashMap:

• Kế thừa HashMap nhưng duy trì thứ tự chèn.

• Thường dùng để làm LRU cache (với accessOrder=true).

• TreeMap:

• Key được sắp xếp tự nhiên hoặc theo Comparator.

• Không cho phép null key.

• Thích hợp range queries (subMap, headMap, tailMap).

• Hashtable:

• Thread-safe (synchronized).

• Không cho phép null key/value.

• ConcurrentHashMap

• Thread-safe, hiệu năng cao hơn Hashtable (dùng phân đoạn lock).

• Không cho phép null key/value.

• Dùng trong môi trường multi-threading.

• EnumMap

• Key là enum, **sắp xếp theo thứ tự enum**.

• Rất nhanh, tiết kiệm bộ nhớ.

• Thường dùng khi key là tập hợp cố định các enum constants.

**Java Thread**

**1 Synchronous vs Asynchronous**

**Phân biệt synchronous vs asynchronous**

**Đặc điểm**

**Synchronous (Đồng bộ)**

**Asynchronous (Bất đồng bộ)**

Thứ tự thực thi

Tuần tự, từng cái một

Song song, không theo thứ tự

Luồng

Một luồng duy nhất

Có thể sử dụng nhiều luồng hoặc một luồng

Hiệu suất

Kém khi có tác vụ chờ đợi

Tốt hơn khi có tác vụ chờ đợi

Độ phức tạp

Dễ viết mã và gỡ lỗi

Phức tạp hơn

**Phân biệt trường hợp sử dụng, ưu nhược điểm của async và sync?**

• Synchronous:

• Trường hợp sử dụng:

• Khi các tác vụ phụ thuộc vào kết quả nhau.

• Khi cần thứ tự thực hiện chính xác.

• Khi tác vụ nhanh và không muốn thêm độ phức tạp.

• Ưu điểm:

• Dễ đọc, dễ debug.

• Thứ tự thực hiện rõ ràng.

• Dễ xử lý lỗi (try/catch).

• Nhược điểm:

• Nếu tác vụ tốn thời gian (I/O, mạng) → block luồng, làm chậm.

• Không tận dụng được song song hoặc đa luồng.

• Asynchronous:

• Trường hợp sử dụng:

• Khi tác vụ tốn thời gian (gọi API, đọc/ghi file lớn, xử lý I/O, database, mạng).

• Khi không cần kết quả ngay lập tức, có thể xử lý kết quả sau.

• Trong UI hoặc server đa luồng, tránh block luồng chính.

• Ưu điểm:

• Gọi API mạng trong ứng dụng web/mobile.

• Tải dữ liệu background.

• Task song song trên server để tăng throughput.

• Nhược điểm:

• Khó debug, khó theo dõi thứ tự thực hiện.

• Quản lý lỗi phức tạp hơn (callback, Future, Promise).

• Nếu sử dụng quá mức, code có thể phức tạp, khó bảo trì.

**Tìm hiểu từ khóa : synchronized trong java**

• Khái niệm:

• synchronized là một từ khóa trong Java để đồng bộ hóa truy cập vào tài nguyên chung giữa các thread.

• Nó giúp chỉ cho phép 1 thread truy cập vào đoạn mã hoặc đối tượng cùng lúc, tránh race condition.

• Cách thức hoạt động:

• Khi một luồng gọi một phương thức hoặc truy cập vào một khối mã được đánh dấu là synchronized, nó sẽ tự động lấy một khóa (lock) trên đối tượng hoặc lớp đó.

• Đối với phương thức instance (non-static): Khóa được lấy là trên đối tượng hiện tại (this).

• Đối với phương thức static: Khóa được lấy là trên đối tượng Class của lớp đó.

• Đối với khối lệnh: Khóa được lấy trên đối tượng được chỉ định trong ngoặc đơn.

• Khi luồng đã có khóa, không có luồng nào khác có thể truy cập vào khối mã hoặc phương thức được đồng bộ hóa đó cho đến khi luồng đầu tiên hoàn thành và nhả khóa.

• Ưu và nhược điểm:

• Ưu điểm:

• An toàn luồng (Thread-safe): Đảm bảo dữ liệu nhất quán và ngăn ngừa race condition.

• Đơn giản: Dễ sử dụng và tích hợp vào mã nguồn.

• Nhược điểm:

• Hiệu suất kém: Việc đồng bộ hóa có thể làm giảm hiệu suất, vì nó buộc các luồng phải chờ đợi lẫn nhau.

• Tiềm ẩn Deadlock: Nếu các luồng yêu cầu khóa theo một thứ tự không đúng, chúng có thể bị kẹt trong tình trạng chờ đợi lẫn nhau vô hạn (deadlock).

**2 Thread**

**Process là gì ?**

• Process (tiến trình) là một chương trình đang chạy trong bộ nhớ (RAM). Mỗi process có:

• Mã lệnh (code): những gì bạn viết trong file .java, .exe, v.v.

• Vùng nhớ riêng (address space): stack, heap, data segment.

• Các tài nguyên hệ điều hành cấp phát: file descriptors, kết nối mạng, thread,…

• Trạng thái (state): running, waiting, ready, terminated.

**Thread là gì ?**

• Định nghĩa:

• Thread (luồng) là đơn vị nhỏ nhất của sự thực thi bên trong một process.

• Một process có thể có nhiều thread cùng chạy song song, và chúng chia sẻ tài nguyên của process đó (bộ nhớ heap, file, kết nối mạng...).

• Đặc điểm:

• Nhẹ hơn process → tạo thread ít tốn tài nguyên hơn tạo process.

• Các thread trong cùng 1 process chia sẻ cùng bộ nhớ heap, nhưng có stack riêng.

• Dễ dàng giao tiếp và trao đổi dữ liệu (vì chung heap).

• Nhưng dễ xảy ra race condition nếu không đồng bộ hóa (sử dụng synchronized, Lock, …).

**Có bao nhiêu cách để tạo 1 thread trong java ? Khác biệt giữa việc sử dụng các cách đó gì ?**

• Có 5 cách:

• Kế thừa Thread

• Implement Runnable

• Lambda

• Sử dụng Callable+Future

• Sử dụng Thread Pool(Executor Framework)

• Khác nhau:

**Cách tạo thread**

**Ưu điểm**

**Nhược điểm**

**Khi dùng**

Extend Thread

Đơn giản, dễ hiểu

Không thể kế thừa class khác

Khi cần demo nhanh

Implement Runnable

Linh hoạt, tách logic

Không trả về kết quả

Khi chỉ cần chạy tác vụ song song

Lambda

Ngắn gọn, dễ viết

Thực chất vẫn là Runnable

Dùng trong code hiện đại

Sử dụng Callable+Future

Có trả về kết quả, xử lý exception

Code phức tạp hơn Runnable

Khi cần lấy kết quả từ thread

Sử dụng Thread Pool

Quản lý pool hiệu quả, tiết kiệm tài nguyên

Hơi phức tạp khi mới học

Ứng dụng thực tế, nhiều task song song

**Thế nào là multi thread ? Sử dụng multi thread mang lại ưu nhược điểm gì ?**

• Khái niệm:

• Multi-thread là việc một chương trình (process) có thể tạo và chạy nhiều luồng (thread) cùng lúc.

• Mỗi thread là một "luồng thực thi" độc lập, nhưng cùng chia sẻ tài nguyên (bộ nhớ, biến toàn cục, file...) của process cha.

• Ưu, nhược điểm:

**Ưu điểm**

**Nhược điểm**

Tận dụng tài nguyên CPU đa nhân → tăng hiệu suất.

Ví dụ: máy có 8 core thì có thể chạy 8 thread song song.

Lập trình phức tạp hơn:

• Phải xử lý race condition, deadlock, synchronization.

Cải thiện tốc độ xử lý:

• Một thread đọc dữ liệu từ DB,

• Thread khác vẫn có thể render UI hoặc xử lý logic → không bị chặn.

Khó debug, test:

• Vì lỗi không xảy ra ổn định → có khi chạy đúng, có khi sai.

Cải thiện trải nghiệm người dùng (đặc biệt trong GUI hoặc server):

Ví dụ: Trong app, một thread tải file, thread khác vẫn cho user bấm nút.

Tốn tài nguyên (context switching):

• Mỗi thread cần stack riêng và CPU phải chuyển qua lại (context switch).

• Nếu tạo quá nhiều thread → CPU tốn thời gian quản lý hơn là làm việc.

Phù hợp với tác vụ song song:

• Xử lý nhiều request web, tính toán phân tán, xử lý big data, game engine…

Khó bảo trì:

• Code nhiều thread dễ rối, khó mở rộng.

• Dùng khi:

• Khi có nhiều tác vụ độc lập, có thể chạy song song.

• Khi có tác vụ I/O blocking (chờ đọc file, DB, network) → thread khác vẫn làm việc.

• Khi cần xử lý real-time, concurrent (ví dụ chat app, game, server).

**Làm thế nào để biết được 1 thread, multi thread đã hoàn thành hay chưa?**

• Dùng Thread.isAlive(): Trả về true nếu thread đang chạy, false nếu đã xong.

• Dùng join(): join() cho phép thread khác chờ thread này hoàn thành.

• Dùng Future (nếu dùng ExecutorService):Khi submit task qua thread pool, Java trả về Future.Có thể gọi future.isDone() để check, hoặc future.get() để block chờ kết quả.

• Dùng CompletableFuture: Cho phép chạy async và callback khi xong, không cần join() hay isAlive().

**Có giới hạn việc tạo ra bao nhiêu thread trong 1 ứng dụng java hay không?**

• Có giới hạn nhưng nó kh phải là con số cố định do được đặt ra bởi java mà phụ thuộc vào các yếu tố khác:

• Tài nguyên hệ điều hành:

• Mỗi thread cần một stack memory (mặc định ~1MB trên JVM).

• Nếu bạn tạo quá nhiều thread → hết RAM → OutOfMemoryError: unable to create new native thread.

• OS cũng giới hạn số lượng thread cho mỗi process (ví dụ Linux có ulimit -u để giới hạn số process/thread).

• Cấu hình JVM:

JVM cấp phát stack size cho mỗi thread bằng cờ -Xss.

Ví dụ: Nếu RAM = 1GB, stack mỗi thread = 1MB → tối đa lý thuyết ~1000 thread.

Nếu giảm -Xss256k thì số thread tối đa có thể tăng lên (~4000).

• CPU core:

• Nếu bạn tạo nhiều thread hơn số core CPU thì chúng phải chia sẻ CPU (context switching), làm giảm hiệu năng.

• Ví dụ: CPU 4 core, tạo 1000 thread tính toán nặng → hiệu năng không tăng mà còn giảm.

**Assignment Java Core**

**1 Assignment**

**DesignPattern**

**1 Creational Pattern**