**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

1. **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.

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.

**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)

1. **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.

1. **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. |

1. **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.

1. **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 lớp Custom Exception
* Chọn loại ngoại lệ: Checked hay Unchecked
* Ném ngoại lệ tùy chỉnh
* Bắt và xử lý ngoại lệ