**1. Cơ sở dữ liệu (database) là gì?**

**- Data (dữ liệu):** Những mô tả về sự vật, sự việc, hiện tượng,… dưới dạng âm thanh, hình ảnh, file… Do đó data quan trọng và cần phải lưu trữ.

VD: bài học, đặc điểm của 1 người,…

**- Database (cơ sở dữ liệu):** Cơ sở dữ liệu là một hệ thống các thông tin có cấu trúc, được lưu trữ trên các thiết bị lưu trữ nhằm thỏa mãn yêu cầu khai thác thông tin đồng thời của nhiều người sử dụng hay nhiều chương trình ứng dụng chạy cùng một lúc với những mục đích khác nhau.

VD: Vở ghi bài học, điêu khắc thông tin lên phiến đá,…

**- RDB (Relational Database – Cơ sở dữ liệu quan hệ):** Cơ sở dữ liệu quan hệ tổ chức dữ liệu theo các bảng và có quan hệ với nhau để giảm thiểu sự dư thừa dữ liệu đồng thời vẫn đảm bảo sự hiệu quả trong lưu trữ và truy xuất dữ liệu, Các thành phần của RDB:

*+ Bảng dữ liệu (table):* Row/record/tuple (hàng), column/field/attribute (cột).

*+ Mối quan hệ (relationship):* 1-1, 1-n, n-n.

*+ Lược đồ thực thể quan hệ (ERD:* Entity relationship diagram).

**2. Phân biệt các câu lệnh DML & DDL? DML, DDL bao gồm các câu lệnh nào?**

**- DDL (Data definition language - ngôn ngữ định nghĩa dữ liệu):**

+ Lệnh CREATE: tạo 1 bảng, 1 view của bảng, hoặc đối tượng khác trong database.

+ Lệnh ALTER: sửa đổi 1 đối tượng database đang tồn tại.

+ Lệnh DROP: xóa toàn bộ 1 bảng, 1 view của bảng hoặc đối tượng khác trong database.

**- DML (Data manipulation language - ngôn ngữ thao tác dữ liệu):**

+ Lệnh SELECT: lấy các bản ghi cụ thể từ 1 hoặc nhiều bảng.

+ Lệnh INSERT: tạo 1 bản ghi

+ Lệnh UPDATE: sửa đổi các bản ghi

+ Lệnh DELETE: xóa các bản ghi.

**- DCL (Data control language)**

**- TCL (Transaction control language)**

**3. Thông dịch, biên dịch khác nhau như thế nào ?**

- Biên dịch: Là quá trình dịch toàn bộ source file sang file .class

- Thông dịch: Là quá trình dịch từng câu lệnh sang mã máy.

**4. JAVA là ngôn ngữ thông dịch hay biên dịch ?**

**-** Java vừa là ngôn ngữ thông dịch, vừa là ngôn ngữ biên dịch

- Java sẽ tiến hành **biên dịch** thông qua compiler nằm trong **JDK** để dịch toàn bộ file .java sang file. class

- Tiếp tục nhờ **JVM** dịch từng dòng trong file .class ra mã máy, nhờ đó máy hiểu và thực thi.

**5. Java Platform là gì?**

- Java Platform (nền tảng Java) là một tập hợp các chương trình giúp phát triển và chạy các chương trình được viết bằng ngôn ngữ lập trình Java.

+ **Java SE** (Standard Edition) – Java core (phiên bản chuẩn & cơ bản của Java).

+ **Java EE** (Enterprise Edition) – phát triển web (phiên bản doanh nghiệp).

+ **Java ME** (Micro Edition) – phát triển ứng dụng mobie & các thiết bị nhỏ.

+ **Java FX** – phát triển các ứng dụng lên quan đến mạng.

**6. Các kiểu dữ liệu trong java, và giá trị mặc định khi khai báo?**

- Có 2 kiểu dữ liệu trong java!

**- Kiểu nguyên thủy:**

+ Kiểu luận lý: Boolean (trả về true hoặc false) (*mặc định là false*)

+ Kiểu số:

. Số nguyên: byte, short, int, long (tương ứng 1,2,4,8 byte) (mặc định: 0)

. Số thực: double (0.0D/d), float (0.0F/f) (tương ứng 4, 8 byte)

+ Kiểu char: 2 byte (\u0000)

**- Kiểu tham chiếu:** mặc định là null

+ String

+ Object

+ …

**7. Sự khác nhau giữa bộ nhớ heap và stack trong java?**

**-------------HEAP-------------**

* Heap memory là bộ nhớ được sử dụng bởi Java Runtime để cấp phát bộ nhớ cho các đối tượng (object) và String.
* Bất kỳ khi nào có một đối tượng được tạo, thì miền giá trị của nó sẽ được tạo lưu ở bộ nhớ Heap.
* Bộ dọn rác (Garbage Collection) chạy trên heap memory để giải phóng bộ nhớ nếu không có đối tượng nào tham chiếu tới.

**--------------STACK--------------**

* Stack memory bao gồm các giá trị cụ thể của method: các biến local và các tham chiếu tới các đối tượng chứa ở trong heap memory được tham chiếu bởi method.
* Stack memory được tham chiếu theo thứ tự LIFO (Last In First Out – vào cuối cùng thì ra đầu tiên). Tức là lưu trữ kiểu ngăn xếp (stack). Khi có một method được thực thi, một block được tạo ra trong stack memory để chứa các biến nguyên thủy local và các tham chiếu tới các object. Khi method kết thúc, block đó sẽ không còn được sử dụng và được phục vụ cho method tiếp theo.
* Stack memory có kích thước rất nhỏ so với Heap memory.

**8. Trình bày các cách khởi tạo một mảng trong JAVA ?**

**- Khai báo trước khởi tạo sau:**

+ **dataType[ ]** arrayName;

**🡪 arrayName = new dataType[]{element 1, element 2,… element n};**

**- Khai báo và khởi tạo cùng lúc:**

+ **dataType[ ]** arrayName **= new dataType[ ] {**element 1, element 2,… element n**};**

+ dataType[] arrayName = {element 1, element 2,… element n}; (có thể rút gọn ntn)

**9. Phần tử của mảng có thể dùng kiểu dữ liệu nào, và có giá trị mặc định là gì ?**

- Mảng có thể lưu được nhiều kiểu dữ liệu khác nhau, tuy nhiên các phần tử trong mảng phải cùng kiểu dữ liệu (Kể cả kiểu nguyên thủy và tham chiếu).

- Giá trị mặc định:

+ Kiểu tham chiếu: null

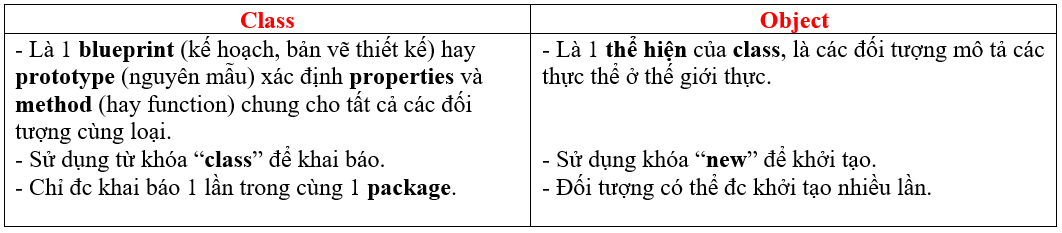
+ Kiểu số nguyên: 0

+ Kiểu số thực: 0.0

**10. OOP là gì ?**

- OOP (Object Oriented Programming): lập trình hướng đối tượng là kỹ thuật lập trình giúp LTV ánh xạ các đối tượng thực tế vào ngôn ngữ lập trình.

**11. Phân biệt class và object?**



**12. Constructor là gì?**

+ Là **phương thức đặc biệt** giúp tạo ra đối tượng.

+ Không có dữ liệu (Value) trả về.

+ Trùng tên với tên **Class**.

+ Trong 1 class có thể có nhiều **constructor** (**overload** => nạp chồng: trùng tên method, khác kiểu dữ liệu hoặc số lượng tham số).

+ 1 **constructor** có thể gọi 1 và chỉ 1 **constructor** khác, nhưng nó phải đc gọi đầu tiên trong body .

+ Nếu như không tạo ra **constructor** nào thì **Java** hỗ trợ tạo **constructor** mặc định.

**13. Cách khai báo constructor và đặc điểm constructor trong JAVA?**

**- Cách khai báo:**

+ Đối với Constructor có tham số: access modifier + tên constructor(tham số)

+ Đối với Constructor không có tham số: access modifier + tên constructor();

**- Đặc điểm:**

+ Là **phương thức đặc biệt** giúp tạo ra đối tượng.

+ Không có dữ liệu (Value) trả về.

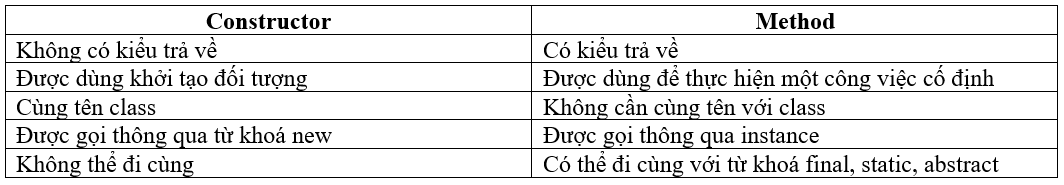
+ Trùng tên với tên **Class**.

+ Trong 1 class có thể có nhiều **constructor** (**overload** => nạp chồng: trùng tên method, khác kiểu dữ liệu hoặc số lượng tham số).

+ 1 **constructor** có thể gọi 1 và chỉ 1 **constructor** khác, nhưng nó phải đc gọi đầu tiên trong body .

+ Nếu như không tạo ra **constructor** nào thì **Java** hỗ trợ tạo **constructor** mặc định.

**14. Phân biệt constructor và method?**



**15. Tính bao đóng là gì ? Làm sao để thu được tính bao đóng trong java ?**

*- Tính bao đóng:* Là 1 quá trình **gói** mã **code** và dữ liệu lại với nhau thành 1 đơn vị duy nhất. Giúp **bảo mật thông tin**, chỉ **show** ra những thứ cần thiết, giúp dữ liệu đc bảo mật và chính xác, kiểm soát dữ liệu (vd: nhập dữ liệu đầu vào).

*- Cách thu được tính bao đóng trong Java:*

+ Tất cả thuộc tính phải để chế độ **private**

+ Có **getter**/**setter** tương ứng.

**16. Tham trị, tham chiếu?**

- Tham trị:

+ Tham trị tức là truyền giá trị

+ Kiểu này dành cho các biến, các tham số khai báo kiểu dữ liệu cơ bản nguyên thủy gồm : byte, short, int, long, float, double, char.

+ Nếu chúng ta gọi một phương thức và truyền một giá trị cho phương thức đó được gọi là truyền giá trị. Việc thay đổi giá trị chỉ có hiệu lực trong phương thức được gọi, không có hiệu lực bên ngoài phương thức. Vì nó chỉ tác động đến bản sao giá trị chứ không phải là địa chỉ nó trong bộ nhớ.

- Tham chiếu:

+ Kiểu này dành cho các biến kiểu dữ liệu tham chiếu.

+ Khi chúng ta gọi một phương thức và truyền một tham chiếu cho phương thức đó được gọi là truyền tham chiếu. Việc thay đổi giá trị của biến tham chiếu bên trong phương thức làm thay đổi giá trị gốc của nó.

**17. Từ khóa static dùng để làm gì ?**

- Từ khóa static giúp tạo ra các thuộc tính và phương thức sử dụng chung cho tất cả các đối tượng cùng loại, thuộc tính và phương thức static thuộc về class chứ không thuộc về đối tượng.

- Đặc điểm:

+ Khối **static** sẽ chạy trước hàm “**main**”, tạo giá trị cho thuộc tính **static**.

+ Có thể áp dụng từ khóa **static** với thuộc tính (biến của **class**), phương thức, khối **static** và **static** **nested class**.

- Mục tiêu:

+ Tiết kiệm vùng nhớ.

+ Không cần “**new**” ra đối tượng nhưng vẫn có thể sử dụng phương thức và thuộc tính **static**.

+ Giúp xây dựng những tiện ích (**util**).

**18. Ràng buộc khi sử dụng static?**

+ Phương thức **static** chỉ gọi được các phương thức và thuộc tính **static**.

+ Phương thức **non-static** có thể gọi được các phương thức và thuộc tính **static**.

+ Phương thức **static** không thể gọi được các phương thức và thuộc tính **non-static**.

+ Không thể sử dụng từ khóa **this** & **super** cho phương thức **static**.

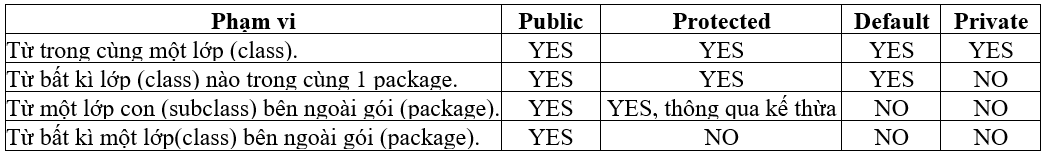
**19. Các loại biến trong JAVA?**

- **Local** (biến cục bộ) – nằm trong phương thức, **constructor**, khối **block** { }.

- **Instance** (biến đối tượng) – là thuộc tính **non-static**: nằm ngoài **contructor**, khối **block** { }.

- **Static** (biến của class) – đc khai báo thông qua từ khóa **static**: nằm ngoài phương thức, **contructor**, khối **block** { }.

**20. Trình bày các loại access modifier, và phạm vi truy cập?**



**21. Kế thừa trong JAVA là gì ?**

- Kế thừa là cơ chế cho phép các lớp con có thể kế thừa các phương thức và thuộc tính của lớp cha.

**- Mục đích:** Tái sử dụng mã nguồn

- **Đặc điểm:**

+ Lớp con không thể kế thừa **constructor** của lớp cha.

=> Gọi **constructor** của lớp cha thông qua “**super**”.

+ Trong **Java** không có **đa kế thừa** (đối với **class**).

=> Vì các lớp cha có thể trùng tên phương thức, chương trình sẽ không hiểu lớp con sẽ kế thừa phương thức nào khi gọi đến.

+ Những tài sản **private** không thể đc kế thừa.

**- Từ khóa:**

+ **super:** là từ khóa hỗ trợ gọi phương thức, thuộc tính và constructor của lớp cha gần nó nhất.

+ **final:** được xây dựng trong rất nhiều ngữ cảnh.

. Nếu đặt ở **biến** => khai báo **HẰNG**.

. Nếu đặt ở **phương thức** => không cho ghi đè (**override**).

. Nếu đặt ở **class** => biến **class** đó thành **vô sinh** (không cho **class** khác kế thừa).

**22. Lớp con kế thừa được những tài sản nào(thuộc tính, phương thức) của lớp cha ?**

- Các thuộc tính của lớp cha.

- Chỉ kế thừa được các METHOD có access modifier lớn hơn hoặc bằng **Protected**

- Không thể kế thừa Constructor (có thể gọi construtor lớp cha thông qua từ khoá super(parameter)).

**23. Lớp Object là gì?**

+ Lớp **Object** là cha của tất cả các **class** trong **Java**.

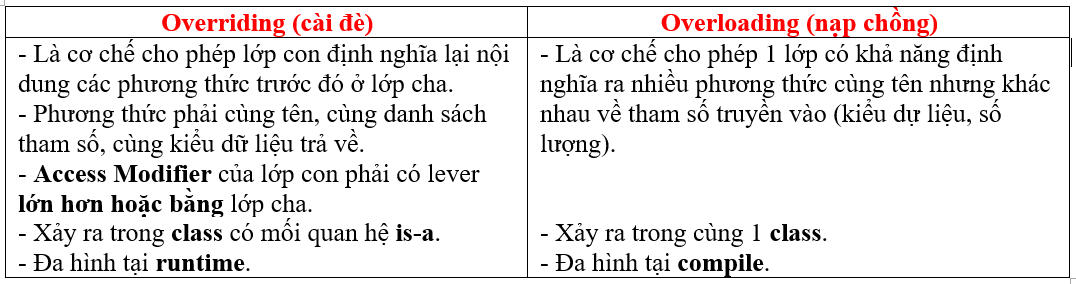
+ **toString( )** – trả về chứa thông tin của đối tượng.

+ **equals( )** – so sánh 2 đối tượng.

**24. Khái niệm đa hình?**

- Tính đa hình là 1 hành động nhưng có thể thực hiện nhiều cách khác nhau thùy thuộc vào ngữ cảnh.

**25. Phân biệt Overloading và Overriding?**



**26. Ép kiểu là gì ? Các loại ép kiểu?**

- **Ép kiểu** – là việc gán giá trị của một biến có kiểu dữ liệu này tới biến khác có kiểu dữ liệu khác**. (**có 2 loại: **tường minh** & **ngầm định**).

+ **Ngầm định(implicit):** thực hiện bởi trình biên dịch, lập trình viên không cần ép kiểu: **con** -> **cha**.

+ **Tường minh(explicit):** lập trình viên phải chủ động ép kiểu (nếu không sẽ bị lỗi **compile**): **cha** -> **con**.

**27. Tính trừu tượng là gì ?**

- **Tính trừu tượng –** chỉ quan tâm đến chức năng, không cần quan tâm đến những xử lí chi tiết.

=> Trong **Java**, tính trừu tượng đc thể hiện qua **abstract class** và **interface**.

**28. Phân biệt abstract class và interface?**

+ Có tạo đc đối tượng không?

=> Cả 2 đều không tạo đc đối tượng.

+ Trường/ thuộc tính?

=> Ở **Interface** đều là hằng số, ở **Abstract Class** thì như Class thông thường.

+ Phương thức (abstract hay non-abstract)?

=> Ở **Interface** đều là abstract, ở **Abstract Class** có thể là abstract hoặc non-abstract.

//trong Java 8, có thể tạo đc phương thức non-abstract trong Interface thông qua default/ static.

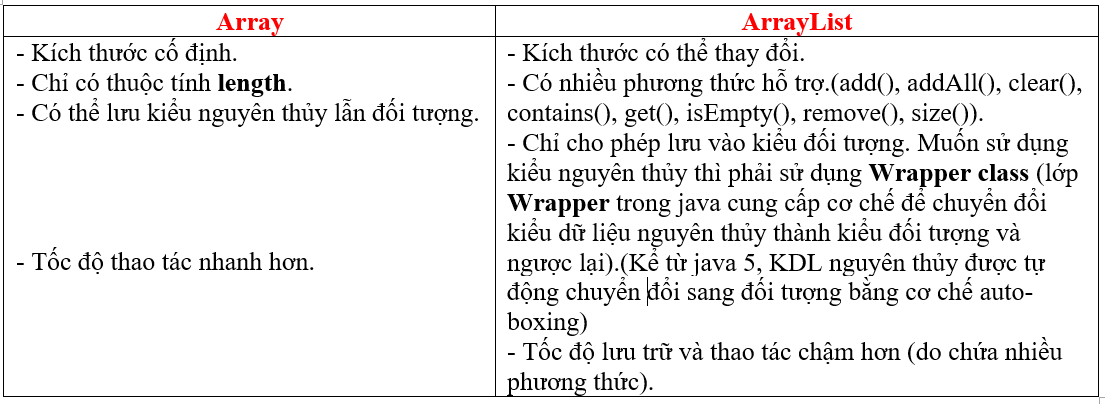
+ Có constructor không?

=> **Interface** không có, **Abstract Class** có.

+ Đa kế thừa?

=> **Interface** hỗ trợ đa kế thừa, **Abstract Class** không hỗ trợ đa kế thừa.

**29. So sánh Array và ArrayList?**



**30. So sánh ArrayList và LinkedList?**



**31. So sánh List, Set và Queue ?**

**-----LIST-----**

- **List** – là cấu trúc dữ liệu tuyến tính, trong đó các phần tử đc sắp xếp theo 1 thứ tự xác định, cho phép các phần tử đc trùng lặp nhau.

- Các class triển khai của List: LinkedList, ArrayList,…

**-----SET-----**

**- Set** – là cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử chỉ xuất hiện duy nhất 1 lần (tương tự như tập hợp trong toán học).

- Các class triển khai của Set:

+ **HashSet** – các phần tử đc lưu trữ dưới dạng bảng băm, không duy trì thứ tự chèn, không lưu phần tử trùng lặp.

+ **LinkedHashSet** – các phần tử đc lưu trữ dưới dạng bảng băm với cấu trúc dữ liệu dạng danh sách liên kết, duy trì thứ tự chèn.

+ **Tree Set** – sử dụng 1 tree cho lưu giữ, mặc định các phần tử đc sắp xếp tăng dần(nhỏ- >lớn).

**-----QUEUE-----**

**- Queue -** cấu trúc dạng danh sách. Thêm và lấy phần tử theo qui tắc FIFO (First In First Out).

- Các class triển khai của Queue: LinkedList, ArrayDeque, PriorityQueue.

**32. Set là gì, các lớp triển khai của Set?**

**- Set** – là cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử chỉ xuất hiện duy nhất 1 lần (tương tự như tập hợp trong toán học).

- Các class triển khai của Set:

+ **HashSet** – các phần tử đc lưu trữ dưới dạng bảng băm, không duy trì thứ tự chèn, không lưu phần tử trùng lặp.

+ **Linked HashSet** – các phần tử đc lưu trữ dưới dạng bảng băm với cấu trúc dữ liệu dạng danh sách liên kết, duy trì thứ tự chèn.

+ **Tree Set** – sử dụng 1 tree cho lưu giữ, mặc định các phần tử đc sắp xếp tăng dần(nhỏ- >lớn).

**33. Generic là gì?**

- **Generic** – là cơ chế cho phép truyền dữ liệu vào như là tham số (**tham số hóa** kiểu dữ liệu), cho phép sử dụng **method**, **class**, **interface** với nhiều kiểu dữ liệu khác nhau.

**34. Ưu điểm và hạn chế khi dùng generic ?**

**- Ưu:**

+Dùng để viết những thuật toán tổng quát: rút gọn mã nguồn

+ Kiểu đối tượng an toàn: Chỉ lưu một kiểu đối tượng duy nhất. Không cho phép lưu trữ 2 đối tượng có kiểu khác nhau

+ Hạn chế ép kiểu

+ Kiểm tra lúc biên dịch: giúp hạn chế bị lỗi lúc Runtime.

**- Nhược:**

+ Không đc sử dụng cho kiểu nguyên thủy.

+ Không sử dụng đc từ khóa static (vì chưa xác định đc kiểu dữ liệu nên chưa tạo đc bộ nhớ).

+Không thể ép kiểu để kiểm tra (ví dụ: không ép đc Person -> Student).

+ Không khởi tạo đc instance (ví dụ: <E> e = new <E>();).

+ Không thể overloading (Ví dụ: phương thức List<String> và List<Integer> thì nó chỉ hiểu là List mà thôi).

+ Không thể khởi tạo mảng <E>.

**35. Stack là gì, các phương thức của stack ?**

- **Stack** là cấu trúc dữ liệu dạng danh sách. Thêm và lấy phần tử theo qui tắc FILO (First In Last Out)

- **Các phương thức thao tác**:

+ Thêm : push, add

+ Lấy ra và xoá : pop (lấy ra phần tử trên cùng. Trả về phần tử đó)

+ Lấy ra không xoá: peek (lấy ra phần tử trên cùng nhưng không xoá).

+ Tìm kiếm: search

+ Chứa đối tượng: constains

**36. Queue là gì, các class triển khai của queue ?**

- **Queue** – là cấu trúc dữ liệu dạng danh sách, thêm và xóa phần tử theo cơ chế FIFO.

- **Các phương thức thao tác**:

+ Thêm phần tử add(đầy thì trả về ngoại lệ)/offer(trả về false),

+ Lấy ra xem element(rỗng thì ném ra ngoại lệ)/peek(trả về null),

+ Lấy ra xem phần tử đầu xem và xóa remove(rỗng thì ném lỗi)/poll(trả về null),

+ isEmpty, size,…

- **Các class triển khai của Queue**:

+ **LinkedList** (đã học).

+ **PriorityQueue** – là 1 dạng **Queue** mà trong đó, các phần tử sẽ đc sắp xếp.

+ **ArrayDeque** – là 1 dạng **Deque** (**Queue** 2 chiều), implement **Deque** và **Queue**.

**37. Phương thức cơ bản của queue ?**

- **Các phương thức thao tác**:

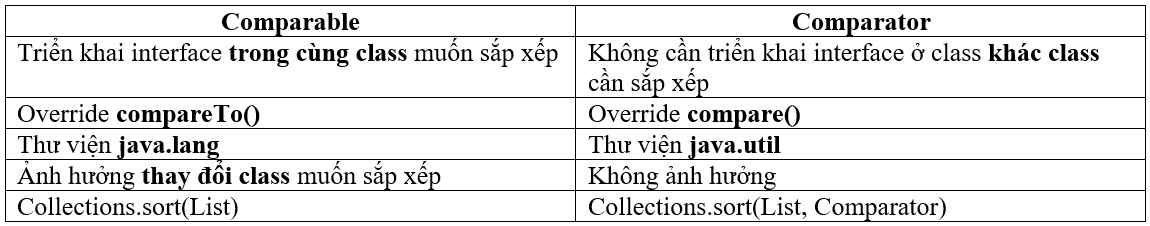
+ Thêm phần tử add(đầy thì trả về ngoại lệ)/offer(trả về false),

+ Lấy ra xem element(rỗng thì ném ra ngoại lệ)/peek(trả về null),

+ Lấy ra xem phần tử đầu xem và xóa remove(rỗng thì ném lỗi)/poll(trả về null),

+ isEmpty, size,…

**38. So sánh Comparable và Comparator, khi nào dùng cái nào ?**



- Khi bài toán đã cho cần sắp xếp theo nhiều trường hợp khác nhau thì dùng comparator.

- Khi bài toán đã cho chỉ yêu cầu 1 trường hợp sắp xếp.

**39. Map là gì, các class triển khai?**

**- Map là** cấu trúc dữ liệu theo dạng key – value

- Các cặp key – value còn gọi là **entry**

- Lưu trữ **key không trùng** và **value có thể trùng**

**- Map** thuộc **Java Collection Framework**

**- Các class triển khai:**

**+ HashMap:** Cho phép 1 key có giá trị null, Không duy trì thứ tự chèn

**+ LinkedHashMap:** Tương tự HashMap nhưng duy trì thứ tự chèn vào

**+ TreeMap:** Tương tự HashMap nhưng sắp xếp theo thứ tự tăng dần (đối với KDL nguyên thuỷ), Không cho phép key null

**40. Phương thức equals() và hashcode() trong Java là gì?**

- **equal( )** – dùng để kiểm tra 2 đối tượng bất kỳ có bằng nhau hay không?

+ Toán tử ‘= =’ được dùng để so sánh địa chỉ 2 đối tượng và giá trị của các biến nguyên thủy. Còn **equals()** được dùng để định nghĩa thế nào là 2 đối tượng trùng nhau, **equals()** chỉ áp dụng cho kiểu đối tượng, không áp dụng cho kiểu nguyên thủy.

+ Vì mặc định, **equals()** được triển khai trong **Object class** sẽ so sánh địa chỉ **object** thay vì so sánh giá trị các thuộc tính trong **object**. Hai **đối tượng** này là duy nhất và được cấp phát vùng nhớ riêng trong **heap**.

+ Muốn **equals()** so sánh giá trị các thuộc tính trong **object** thay vì so sánh địa chỉ thì phải **override** lại phương thức **equals().**

- **hashCode( )** – **Mã băm**, là giá trị định danh cho 1 đối tượng, **hashCode** trả về của đối tượng được tạo ra sẽ là **mã băm** của địa chỉ để **gom nhóm** các đối tượng, giúp tăng tốc độ tìm kiếm của chương trình lên từ **30-50** lần.

+ Nếu hai đối tượng bằng nhau thì chúng phải có cùng **hashCode()**.

+ Nếu hai đối tượng có cùng **hashCode()** thì chúng có thể bằng nhau hoặc là không.

**41. Cây nhị phân là gì?**

**- Cây nhị phân là** một cấu trúc dữ liệu trong đó **mỗi nút** có thể có **nhiều nhất 2 nhánh con** được chia 2 nhánh là left-subtree và right-subtree. **Các nút anh em** nhau gọi là **Sibling**. Các nút không có node con gọi là **Leaf node**. **Nút trên cùng của cây nhị phân được gọi là nút gốc** hoặc **nút cha** và **các nút dưới từ nút gốc** được gọi là **nút con. (Giá trị được sắp xếp left < center < right đối với BST).**

**42. Trình bày các cách duyệt cây nhị phân?**

+ **Inorder** (trung thứ tự) – cây con trái -> node cha -> cây con phải.

+ **Postorder** (hậu thứ tự) – cây con trái -> cây con phải -> node cha.

+ **Preorder** (tiền thứ tự) – node cha -> cây con trái -> cây con phải.

+ **Breath-first** – duyệt lần lượt theo từng **level(từ root xuống tầng thấp hơn theo chiều từ trái sang phải)**.

**43. Ngoại lệ (Exception là gì**) ?

- **Exception** – là 1 **sự kiện bất thường**, nó làm phá vỡ **flow** (luồng thực thi) của chương trình và có thể làm chết chương trình.

**44. Phân loại Exception?**

- **Checked Exception** – xảy ra tại thời điểm **compile** (có thể gọi là **compile time exceptions**), bắt buột phải xử lí (**handle**).

+ **FileNotFoundException** (một file được chỉ định với đường dẫn không tồn tại, hoặc không có quyền truy cập vào file đó).

+ **IOException** (lỗi không đọc được file).

+ **ClassNotFoundException** (nếu không thể chuyển kiểu object này sang kiểu object khác hoặc không tìm thấy class muốn tham chiếu tới).

+ **DatabaseException**.

+ **NoSuchFieldException**.

- **Unchecked Exception** – xảy ra lúc **runtime**, không chắc chắn xảy ra, không bắt buộc phải **handle** tại thời điểm **compile**.

+ **NullPointerException** (được ném ra khi chương trình cố gắng tham chiếu tới một đối tượng nhưng nó không có vị trí nào trên bộ nhớ, tức là có giá trị null).

+ **ArrayIndexOutOfBoundsException** (truy nhập mảng với chỉ số không hợp lệ).

+ **ArithmeticException** (xảy ra khi chia cho số 0).

+ **NumberFormatException** (xảy ra khi cố gắng để chuyển đổi một chuỗi thành một số, xuất hiện khi khởi tạo biến bằng Integer.parseInt(scanner.nextLine()) hoặc Double.parseDouble(scanner.nextLine())).

+ **InputMismatchException** (xảy ra khi cố gắng để chuyển đổi một chuỗi thành một số, xuất hiện khi khởi tạo biến bằng scanner. .nextInt() hoặc scanner. .nextDouble ()).

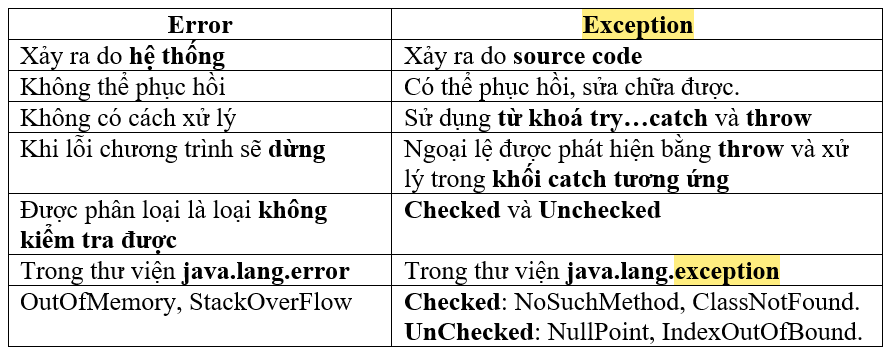
- **Error** – lỗi do môi trường thực thi (**JVM**), không thể **handle** + chương trình sẽ chết.

+ **StackOverflowError** (tràn vùng nhớ stack, stack đạt đến giới hạn tối đa).

+ **VirtualMachineError** (được ném ra khi máy ảo Java gặp lỗi nội bộ hoặc giới hạn tài nguyên, điều này khiến máy không thể hoạt động. Đây là một cơ chế tự bảo vệ được JVM sử dụng để ngăn toàn bộ ứng dụng bị treo).

+ **OutOfMemoryError** (khi hết bộ nhớ trong heap, chủ yếu khi bạn cố gắng tạo một object và không có đủ không gian trên heap để phân bổ đối tượng đó).

**45. Phân biệt Error và Exception?**



**46. Có bao nhiêu cách để xử lý ngoại lệ (handle)?**

- **throw/throws** – ném ngoại lệ cho phương thức khác xử lí.

- **try – catch**: xử lí ngay tại nơi xảy ra **exception**.

**47. Một số lưu ý khi dùng try-catch?**

+ Tại một thời điểm chỉ xảy ra một ngoại lệ và tại một thời điểm chỉ có một khối **catch** được thực thi. Khi **exception** đã bị bắt ở một catch thì các catch tiếp theo sẽ không được bắt.

+ Tất cả các khối **catch** phải được sắp xếp từ cụ thể nhất đến chung nhất (**từ exception con đến exception cha**), tức là phải khai báo khối lệnh catch để xử lý lỗi **NullPointerException**, **ArithmeticException**, … trước khi khai báo catch để xử lý lỗi **Exception**.

+ Khối lệnh **finally** luôn được thực thi dù chương trình có xảy ra ngoại lệ hay không (ngay cả sử dụng lệnh **return**).(99.99% là thực thi còn 0.01% không thực thi là do một thread bị ngừng; sử dụng phương thức System.exit(); do một exception phát sinh trong finally block )

+ Đối với mỗi khối **try**, có thể không có hoặc nhiều khối **catch**, nhưng chỉ có một khối **finally**.

+ Khối **finally** sẽ không được thực thi nếu chương trình bị thoát bằng cách gọi **System.exit()** hoặc xảy ra một lỗi (**Error**) không thể tránh khiến chương trình bị chết.

+ Khối try bắt buộc phải có

+ Có thể không có catch nhưng phải có finally

**48. Serialization là gì ? lưu ý khí sử dụng Serialization?**

- **Serialization** – là cơ chế cho phép chuyển đổi từ đối tượng (**object**) sang **Byte Stream**, ngược lại **Deserialization** là cơ chế chuyển từ **Byte Stream** sang **Object**.

+ Trong **Java**, để 1 **Object** có thể **Serialization** thì class đó phải **implement** **Interface** **Serializable**.

+ Khi lớp cha đã **implement** **Serializable** thì lớp con không cần **implement** lại **Serializable**.

+ Từ khóa **transient/static** giúp cho thuộc tính không bị **Serializable**.

**49. SOLID là gì? Phát biểu của từng nguyên lý?**

- **SOLID** – là 1 trong những nguyên tắc (chỉ dẫn) để giúp chúng ta xây dựng đc các ứng dụng **OOP** hiệu quả, giúp lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain (bảo trì).

- **S** – **Single responsibility principle** – Nguyên lý Trách nhiệm Duy nhất: Mỗi lớp chỉ nên đảm nhiệm 1 nhiệm vụ duy nhất.

+ Mục đích: Dễ quản lí mã nguồn, Các lớp tập trung vào nhiệm vụ của mình, Giảm tính phụ thuộc giữa các thành phần, Có thể phát triển đồng thời các lớp độc lập với nhau, Dễ dàng mở rộng, Dễ dàng bảo trì,…

- **O** – **Open closed principle** – Nguyên lý đóng mở: Có thể thoải mái mở rộng 1 **class**, nhưng không đc sửa lỗi bên trong **class** đó.

+ Mục đích: Dễ mở rộng, thay đổi,…

- **L** – **Liskov substitution principle** – Nguyên lý thay thế Liskov: Trong 1 chương trình, các **object** của **class con** có thể thay thế **class cha** mà không làm thay đổi tính đúng đắn của chương trình.

+ Mục đích: Tránh sai sót khi mở rộng thiết kế.

- **I** – **Interface segregation principle** – Nguyên lý phân tách Interface thay vì dùng 1 **Interface** lớn, nên tách thành nhiều **Interface** nhỏ với nhiều mục đích cụ thể.

+ Mục đích: tránh dư thừa mã nguồn không cần thiết, dễ quản lí hơn

- **D** – **Dependency inversion principle** – Nguyên lý đảo ngược phụ thuộc:

+ Các **module** cấp cao không nên phụ thuộc vào các **module** cấp thấp, cả 2 nên phụ thuộc vào **abstraction**.

+ **Interface** (**abstraction**) không nên phụ thuộc vào chi tiết, mà chi tiết nên phụ thuộc vào trừu tượng.

**50. Design pattern là gì? Chia làm bao nhiêu loại. Tổng số là bao nhiêu?**

- Là 1 giải pháp tổng thể cho các vấn đề chung trong thiết kế phần mềm. DP không phải là Ngôn ngữ lập trình, được sử dụng phổ biến trong OOP.

**- Lợi ích:**

+ Giảm thời gian phát triển phần mềm

+ Hạn chế rủi ro

+ Khi làm việc nhóm -> có tiếng nói chung

+ Tái sử dụng mã nguồn

+ Giúp code dễ đọc

**- Tổng số có 23 loại DP chia làm 3 nhóm:**

+ **Creational Design Pattern (khởi tạo)** – có 5DP, giải quyết các vần đề liên quan đến khởi tạo đối tượng: Singleton, factory,…

+ **Structoral Design Pattern (cấu trúc)** – có 7DP, thiết kế các thành phần của lớp và đối tượng: Proxy, Façade,…

+ **Behavioral Design Pattern (hành vi/ tương tác)** – có 11DP, giải quyết các vấn đề trong giao tiếp giữa các đối tượng: Observer, Strategy,…