[NodeJS Developer: câu hỏi phỏng vấn thường gặp](https://www.vietnamworks.com/cau-hoi-phong-van/NodeJS-Developer-ik)

<https://timviec365.net/cau-hoi-phong-van-nodejs-b452.html>

<https://hanhtranglaptrinh.net/mot-so-cau-hoi-phong-van-ve-node-js/>

<https://vn.bitdegree.org/huong-dan/node-js-tutorial/>

# NODE

# 

# How node work

[**https://www.geeksforgeeks.org/how-node-js-works-behind-the-scene/**](https://www.geeksforgeeks.org/how-node-js-works-behind-the-scene/)

# MongoDB

Là document database, thuộc NOSQL các dữ liệu được lưu trữ dưới dạng JSon với các cặp khoá key value thay vì bảng

# NOSQL vs SQL

NoSQL là cơ sơ dữ liệu không có quan hệ còn sql là có quan hệ

đối với sql, thì để tạo ra table phải cần có schema, phải khai ra các field là gì, thuộc loại nào, integer hay text, và phải có các primary key hoặc là foreign key nếu có

Còn nosql, là document database, thuộc NOSQL các dữ liệu được lưu trữ dưới dạng JSon với các cặp khoá key value, không cần schema

Dùng khi nào

Sql dùng khi có cấu trúc dữ liệu rõ ràng liên quan với nhau, khi mà xác định là dữ liệu sẽ có những cái row này column này, các data sẽ có quan hệ với nhau thông qua các foreign key này

Còn nosql là khi không có cấu trúc dữ liệu rõ ràng và có thể thay đổi trong tương lai

# 

# Callback

Callback là một hàm sẽ được thực hiện sau khi một hàm khác đã thực hiện xong - vì thế nó có tên là callback. Dung callback de xu ly cho cac ham bat dong bo

# 

# Promise

Promise là một *cơ chế* trong JavaScript giúp bạn thực thi các tác vụ bất đồng bộ mà không rơi vào *callback hell* hay *pyramid of doom*, là tình trạng các hàm callback lồng vào nhau ở quá nhiều tầng. Các tác vụ bất đồng bộ có thể là gửi AJAX request, gọi hàm bên trong setTimeout, setInterval hoặc requestAnimationFrame, hay thao tác với WebSocket hoặc Worker…

Co 3 trang thai, pending , fulfill, rejected. Cu the hon: trang thai fulfill khi promise vô resolve, trang thai reject la khi promise vo reject, con pending la khi khong vo resolve lan reject



# GraphQL

## What is graphql

GraphQL is a query language for APIs. It gives clients the power to ask for exactly what they need and nothing more. You can get specific fields in the existing data.

It only uses one endpoint instead of many endpoints like REST.

**Graphql vs REST**

|  | **REST** | **Graphql** |
| --- | --- | --- |
| **What is it?** | **REST is a set of rules that defines structured data exchange between a client and a server.** | **GraphQL is a query language, architecture style, and set of tools to create and manipulate APIs.** |
| **Best suited for** | **REST is good for simple data sources where resources are well defined.** | **GraphQL is good for large, complex, and interrelated data sources.** |
| **Data access** | **REST has multiple endpoints in the form of URLs to define resources.** | **GraphQL has a single URL endpoint.** |
| **Data returned** | **REST returns data in a fixed structure defined by the server.** | **GraphQL returns data in a flexible structure defined by the client.** |
| **How data is structured and defined** | **REST data is weakly typed. So the client must decide how to interpret the formatted data when it is returned.** | **GraphQL data is strongly typed. So the client receives data in predetermined and mutually understood formats.** |
| **Error checking** | **With REST, the client must check if the returned data is valid.** | **With GraphQL, invalid requests are typically rejected by schema structure. This results in an autogenerated error message.** |

## 

## Arguments in graphql

In a system like REST, you can only pass a single set of arguments - the query parameters and URL segments in your request. But in GraphQL, every field and nested object can get its own set of arguments, making GraphQL a complete replacement for making multiple API fetches

## Aliases in graphql



Fragments

To reused, avoid repeat fields

## publisher subscriber pattern

# Overloading vs Overriding

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have an IS-A (inheritance) relationship. |
| In case of method overloading, *parameters must be different*. | In case of method overriding, *parameters must be the same*. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| In Java, method overloading can't be performed by changing the return type of the method only. *Return type can be the same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

# Static vs Final

The static keyword can be used in 4 scenarios

* static variables
* static methods
* static blocks of code
* static nested class

**Static variable**

* It is a variable which belongs to the class and not to object (instance).
* Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
* A single copy to be shared by all instances of the class.
* A static variable can be accessed directly by the class name and doesn’t need any object.
* Syntax: Class.variable

**Static method**

* It is a method which belongs to the class and not to the object (instance).
* A static method can access only static data. It can not access non-static data (instance variables) unless it has/creates an instance of the class.
* A static method can call only other static methods and can not call a non-static method from it unless it has/creates an instance of the class.
* A static method can be accessed directly by the class name and doesn’t need any object.
* Syntax: Class.methodName()
* A static method cannot refer to this or super keywords in any way.

**Static class**

* Java also has "static nested classes". A static nested class is just one which doesn't implicitly have a reference to an instance of the outer class.
* Static nested classes can have instance methods and static methods.
* There's no such thing as a top-level static class in Java.

Side note:

**The main method is static since it must be accessible for an application to run before any instantiation takes place.**

| **class StaticVariable {  public static String STATIC = "static"; }  class StaticMethod {  public static void print(String method) {  System.out.println(StaticVariable.STATIC + " " + method);  } }  class StaticBlock {  private static String subject;   static {  System.out.println("Khối static được gọi");  }   static {  subject = "Khối static (static blocks)";  }   StaticBlock () {  System.out.println("hàm main() được gọi");  System.out.println("Subject = " + subject);  } }  class StaticClass {  static class NestedStaticClass {  public static String STATIC\_CLASS = "static class";  }   public void printStaticClass() {  System.out.println(NestedStaticClass.STATIC\_CLASS);  } }  public class StaticAndFinal {   public static void main(String[] args) {  System.out.println(StaticVariable.STATIC + " variable");   StaticMethod.print("method print");   StaticBlock staticBlock = new StaticBlock();   StaticClass staticClass = new StaticClass();  staticClass.printStaticClass();    } }** |
| --- |

The final keyword is used in several different contexts to define an entity which cannot later be changed.

* A final class cannot be subclassed. This is done for reasons of security and efficiency. Accordingly, many of the Java standard library classes are final, for example java.lang.System and java.lang.String. All methods in a final class are implicitly final.
* A final method can't be overridden by subclasses. This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class.
* A final variable can only be initialized once, either via an initializer or an assignment statement.  
  It does not need to be initialized at the point of declaration, this is called a blank final variable, but in this case:
  + A blank final instance variable must be assigned at every constructor of its class.
  + A blank final static variable must be assigned in a static initializer in its class.

# **OOP:**

The four principles of object-oriented programming (abstraction, inheritance, encapsulation, and polymorphism) are features that - if used properly - can help us write more testable, flexible, and maintainable code.

| **Abstraction** | The process of hiding implementation details and exposing only the functionality to the user. In abstraction, we deal with ideas and not events. This means the user will only know “what it does” rather than “how it does”.   * The class should be abstract if the class has one or many abstract method * An abstract class can have constructors, concrete methods, static method, and final method * Abstract class can’t be instantiated directly with the ***new*** operator. It can be:   A b = new B();   * The child class should override all the abstract methods of parent else the child class should be declared with abstract keyword |
| --- | --- |
| **Encapsulation** | The process of wrapping code and data together into a single unit.   * Declare the private variables * Declare getter and setter to get and set variable values |
| **Inheritance** | The process of one class inheriting properties and methods from another class in Java. Inheritance is used when we have **is-a** relationship between objects. Inheritance in Java is implemented using **extends** keyword.   * Single Inheritance * Multilevel Inheritance * Hierarchy Inheritance * Multiple Inheritance * Hybrid Inheritance |
| **Polymorphism** | The ability to perform many things in many ways. The word Polymorphism is from two different Greek words- poly and morphs. “Poly” means many, and “Morphs” means forms. So polymorphism means many forms. The polymorphism can be present in the case of inheritance also. The functions behave differently based on the actual implementation.   * Static and compile polymorphism: * Dynamic and runtime polymorphism |

# Access Modifier:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default. (Not in Typescript)
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

# SOLID:

**S - Single responsibility:** a class should only have one responsibility. Furthermore, it should only have one reason to change.

**O - Open/Close:** Classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application.

**L - Liskov Substitution:** if class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.

**I - Interface Segregation:** Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

**D - Dependency Inversion:** The principle of dependency inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

# Design pattern:

**Singleton:**

Singleton is a creational design pattern that lets you ensure that a class has only one instance and provide a global access point to this instance

**Factory:**

Factory Method is a creational design pattern that defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Builder:**

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

**Prototype:**

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

**Strategy:**

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

**State:**

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

**Observer:**

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing.

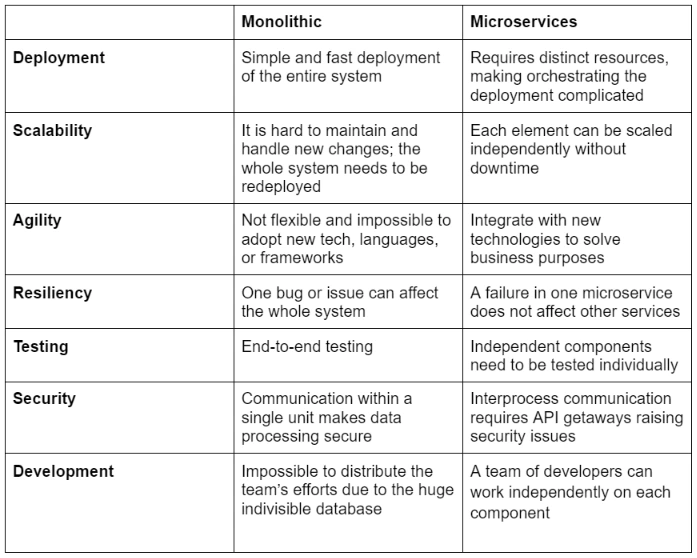
**Command:**

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue the request's execution, and support undoable operations.

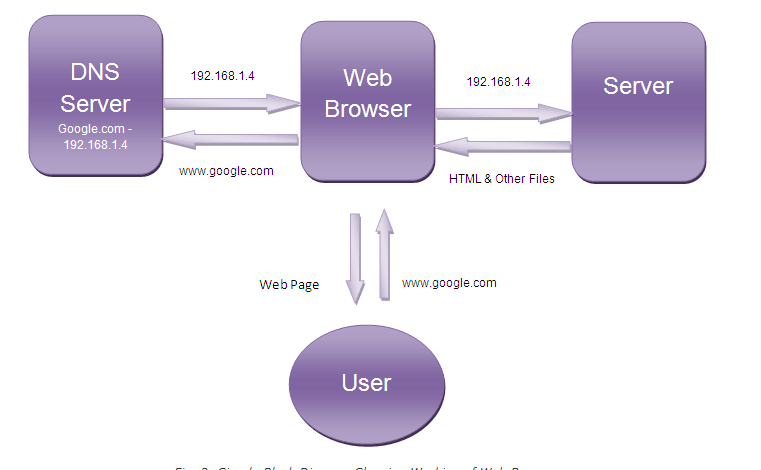
**Iterator:**

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

# Microservice vs Monolith:

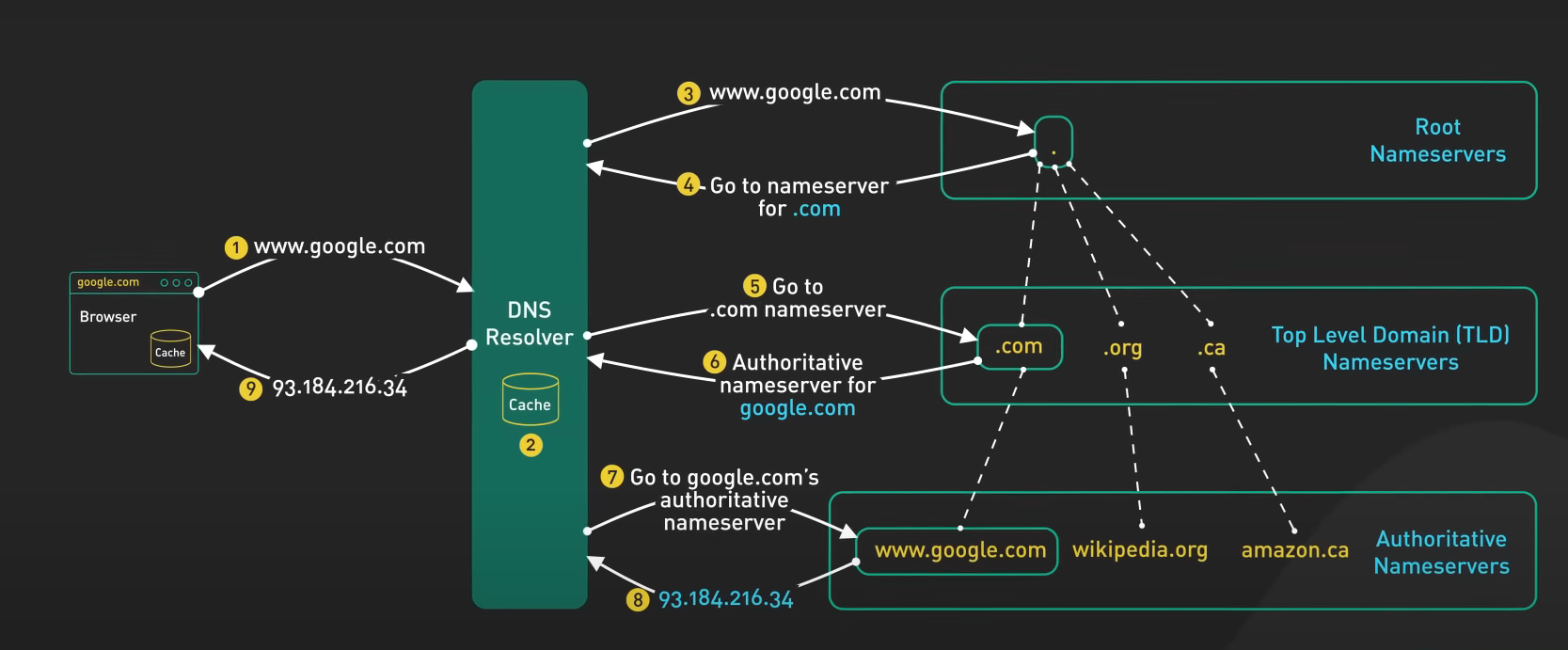


# Web Browser:



## DNS: Domain name system: translate domain names to IP address

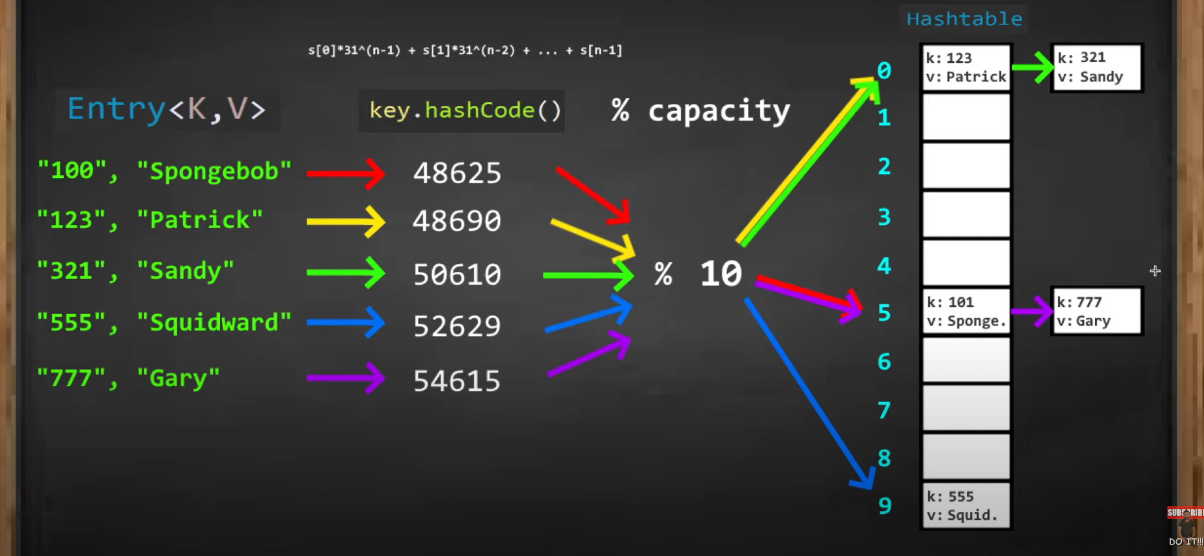
1. Check cache in browser. if not, call DNS resolver
2. Check cache in DNS resolver, if not, call Root name server (Root DNS)
3. Call Root name server
4. Return TLD .com
5. Call TLD (top level domain)
6. Return authoritative nameserver
7. Call Authoritative name server
8. Return IP address for DNS resolver
9. DNS resolver return for browser



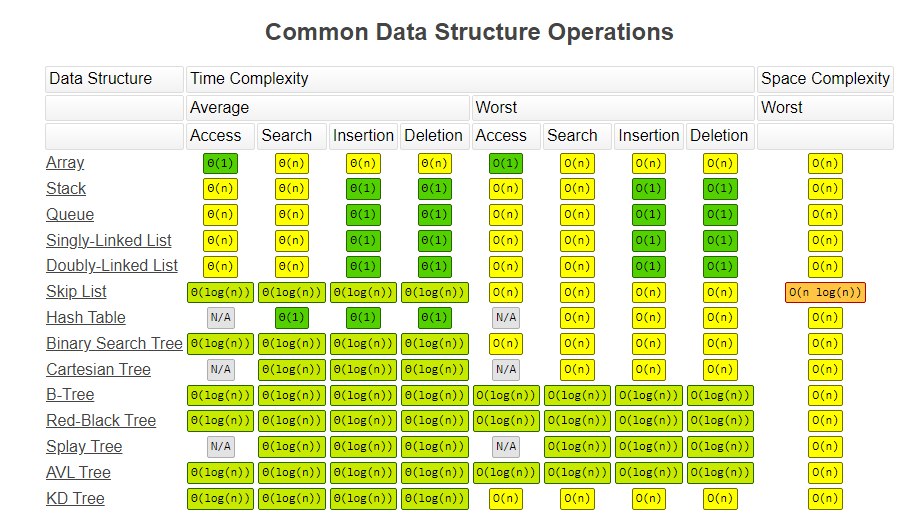
# Data Structure:

## Hashtable:

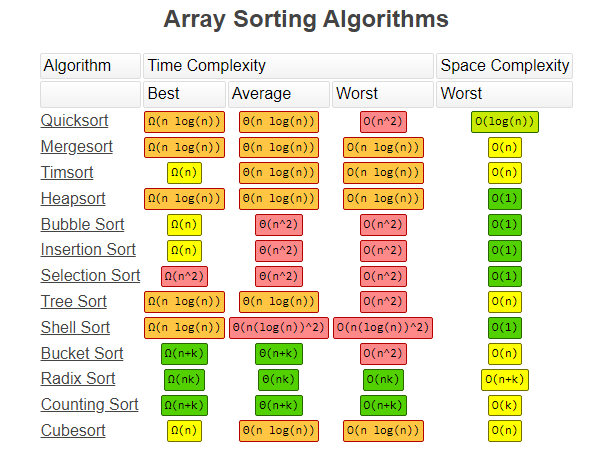
is an array, the key will be hash by modulus the capacity. But got problem with Collision, to solve that we need will store the linked list



## Complexity and Space:



## Sorting with array:



# Kafka:

Apache Kafka is the solution for the distributed and scalable problem.

Producer will update the message to queue so the consumer will consume this message.

In Queue, we have multiple partitions. the message will be distributed to partitions with the partition key.

All the partitions will have the same topic

Each message is a record and has an offset.

Each consumer need to have consumer group Id to know the current offset

# Typescript

## What is Typescript?

A programming language developed by Microsoft

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

## Components of TypeScript?

TypeScript comprises three main components: Language, the TypeScript Compiler, and the TypeScript Language Service.

**Language**: the syntax, keywords, and type annotations.

**The TypeScript Compiler (TSC**): converts the instructions written in TypeScript to its JavaScript equivalent.

**The TypeScript Language Service:** an additional layer of editor-like applications, such as statement completion, signature help, code formatting, and colorization, among other things.

## Why should we use Typescript

It adds more quality to your code, cuz tons of small bugs that we can ship into the production just by not having restricted type language.

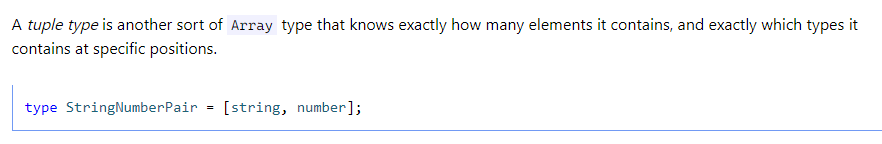
IDE fully understands what you are doing so it keeps reading from all different files. It gives very nice auto competition, make sure you are not messing around.

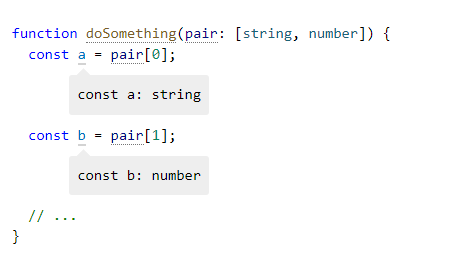
## Type in Typescript

Primitive Type: string, number, boolean

Non primitive: array, any, functions, tuples, object type, union type, type alias, interface

## Tuple





## **Type vs Interface**

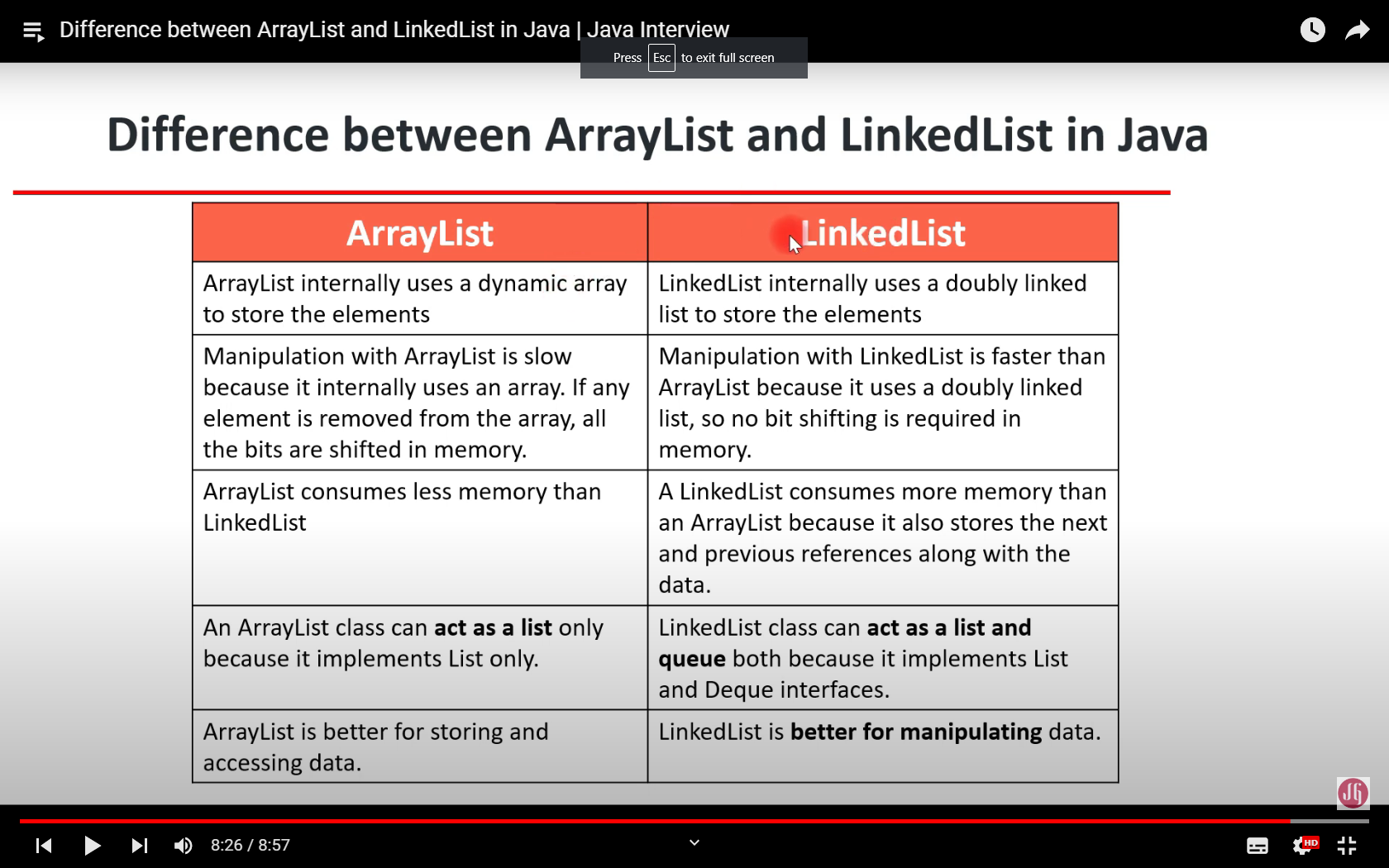
| Type | Interface |
| --- | --- |
| Can not | Declaration merging |
| Extending a type via intersections |  |
| Computed properties | Cannot Computed properties  ⇒ báo error: A computed property name in an interface must refer to an expression whose type is a literal type or a 'unique symbol' type |
| Type có Union | Dont have Union |

# Interview QA:

## HashSet - HashMap - HashTable:

| Hash Set | HashMap | HashTable - ConcurrentHashMap | TreeMap |
| --- | --- | --- | --- |
| * Implement Set interface * Store data as Objects * Internally uses Hashmap * Does not allow duplicate elements * Allow only one null element * Performance O(1), Space is O(n) | * Not thread safe * Implement Map interface * Store data as key-value pairs * Internally uses an array of Entry<K, V> objects * Does not allow duplicate keys, but allow duplicate values * Allow only 1 null key but multiple null values * Performance O(1), Space is O(n) | * Thread safe * Do not allow null value and null key * Slightly slower * Synchronized * Performance O(n), Space is O(n) | * Implement Map interface * Not allow null keys, but can multiple null values * Performance O(log(n)), Space is O(n) * Sorted |
|
|
|
|

## ArrayList - LinkedList:



## Fail-safe and Fail-fast:

Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation. ArrayList, HashMap, HashSet, ...

Whereas, Fail-Safe systems don’t abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible. ConcurrentHashMap, CopyOnWriteArrayList,...

## Streams and Collections:

| **STREAMS** | **COLLECTIONS** |
| --- | --- |
| It doesn’t store data, it operates on the source data structure i.e collection. | It stores/holds all the data that the data structure currently has in a particular data structure like Set, List or Map, |
| They use functional interfaces like lambda which makes it a good fit for programming languages. | Don’t use functional interfaces. |
| Java Streams are consumable i.e; to traverse the stream, it needs to be created every time. | Non-consumable i.e; can be traversable multiple times without creating it again. |
| Java streams support both sequential and parallel processing. | Supports parallel processing and parallel processing can be very helpful in achieving high performance. |
| All the Java stream API interfaces and classes are in j**ava.util.stream** package. | Specific classes for primitive types such as **IntStream**, **LongStream**, and **DoubleStream** are used in collections since primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time. |
| Streams are not modifiable i.e one can’t add or remove elements from streams. | These are modifiable i.e one can easily add to or remove elements from collections. |
| Streams are iterated internally by just mentioning the operations. | Collections are iterated externally using loops. |

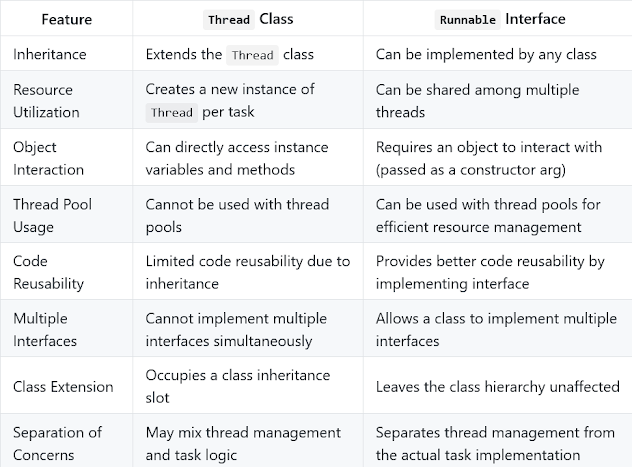
## Final keyword:

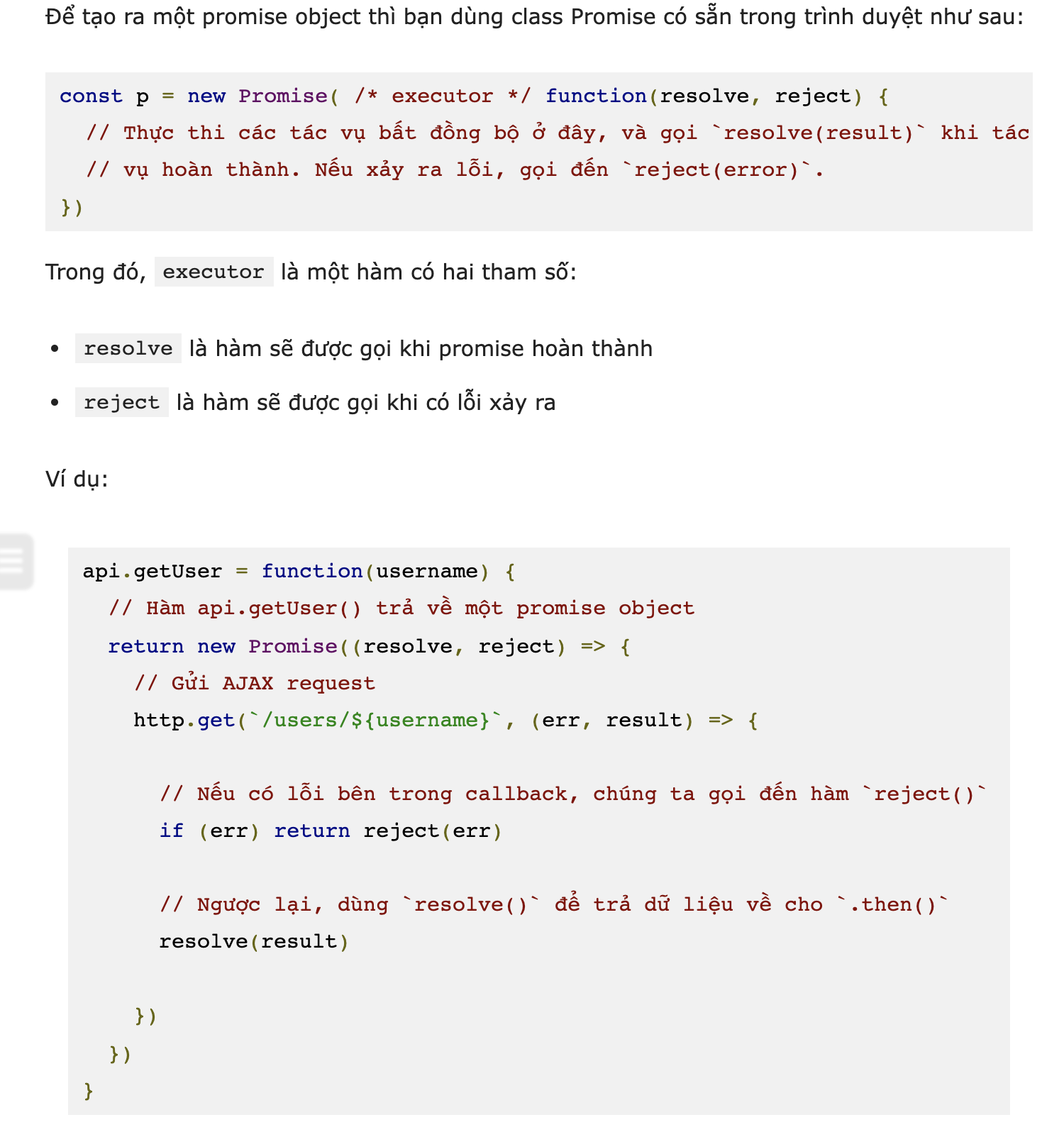
* Can the final function be inherited? - Yes, But cannot overriding this function

## What is static class:

A class can be said to be a static class if all the variables and methods of the class are static and the constructor is private. Making the constructor private will prevent the class from being instantiated. So the only possibility to access is using the Class name only.

## Thread vs Runnable





De lay ket qua cua promise thi .then , neu promise bi loi thi se vao .catch, ngoai ra con co .finally . Then, catch nhan vao function co doi so ung voi cai ma promise resolve, reject.

Neu trong then tra ve promise thi then tiep theo se lay cai resolve cua promise ban dau. Con neu then return khong phai promise (nhu array hoac so) thi then tiep theo lay cai return cua cai then ban dau ( tuc la lay cai array hoac so ma then ban dau return)

**Async- Await**

async/await là một *cơ chế* giúp bạn thực hiện các thao tác bất đồng bộ một cách *tuần tự* hơn. Async/await vẫn sử dụng Promise ở bên dưới nhưng mã nguồn của bạn (theo một cách nào đó) sẽ trong sáng và dễ theo dõi.



neu de async truoc function thi no se trả về promise, cai resolve của promise là cái mà async function return, con reject cua promise thi dung throw, hoac return loi



**Async vs Sync**

Hiểu một cách đơn giản thì đồng bộ hay chính xác là xử lý đồng bộ là code sẽ được chạy tuần tự theo trình tự đã viết sẵn từ trên xuống dưới, code bên dưới chỉ chạy khi code ở bên trên đã chạy xong và trả về kết quả, còn bất đồng bộ là các đoạn code ở phía dưới có thể chạy dù code bên trên chưa thực thi và trả về kết quả .

**Axios**

Axios là một HTTP client được viết dựa trên Promises, dùng để hỗ trợ cho việc xây dựng các ứng dụng API, ho tro Promise API

* Make [XMLHttpRequests](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) from the browser
* Make [http](http://nodejs.org/api/http.html) requests from node.js

**Event-Driven**

**Non-Blocking IO**

**Blocking IO**

**Stream**

**ContentType**

**Thread Pool**

**DOM**

.Viet tat cua Document object model co 3 thanh phan element (a tag), attribute (href), va text. Dung javascript co the truy cap DOM de thanh doi cac thanh phan. Mọi thành phần đều được xem là một node. Mỗi khi trang web được load lại, trinh duyet tao ra dom cua trang do.

**VIRTUAL DOM**

**Window**

# Modules ( os, path,fs , http)

Module là các đoạn code được đóng gói lại với nhau và được giữ Private, chỉ các hàm và biến được định nghĩa bên trong Module là có thể truy cập và thao tác với nhau. Còn khi nào cần sử dụng Module từ bên ngoài thì chúng ta sẽ chìa các API là các biến, các hàm, hoặc cả 2 biến và hàm ra bên ngoài bằng cách sử dụng đối tượng exports hoặc module.exports.

**ES6 va CommonJS**

**ES6** la tap hop ky thuat nang cao cua Javascript , co them nhung tinh nang moi nhu arrowfunction, destructuring, let , const, class, for of ….

**CommonJs**

La core cua nodejs, no ho tro require/ exports, module.exports de tuong tac giua cac file.

Con ben es6 dung import vs export,

Muon xai es6 ben node thi phai dung type module trong package json.

**Destructuring**

Destructuring là một cú pháp cho phép bạn gán các thuộc tính của một Object hoặc một Array. Điều này có thể làm giảm đáng kể các dòng mã cần thiết để thao tác dữ liệu trong các cấu trúc này. Có hai loại Destructuring: Destructuring Objects và Destructuring Arrays.

**[a, b] = [1, 2]**

# Closure

Function con có thể lấy giá trị của function ở ngoài. Ví dụ:

function debug(){

var huy = 2

return function(){

var qh = 5

console.log(huy+qh)

}

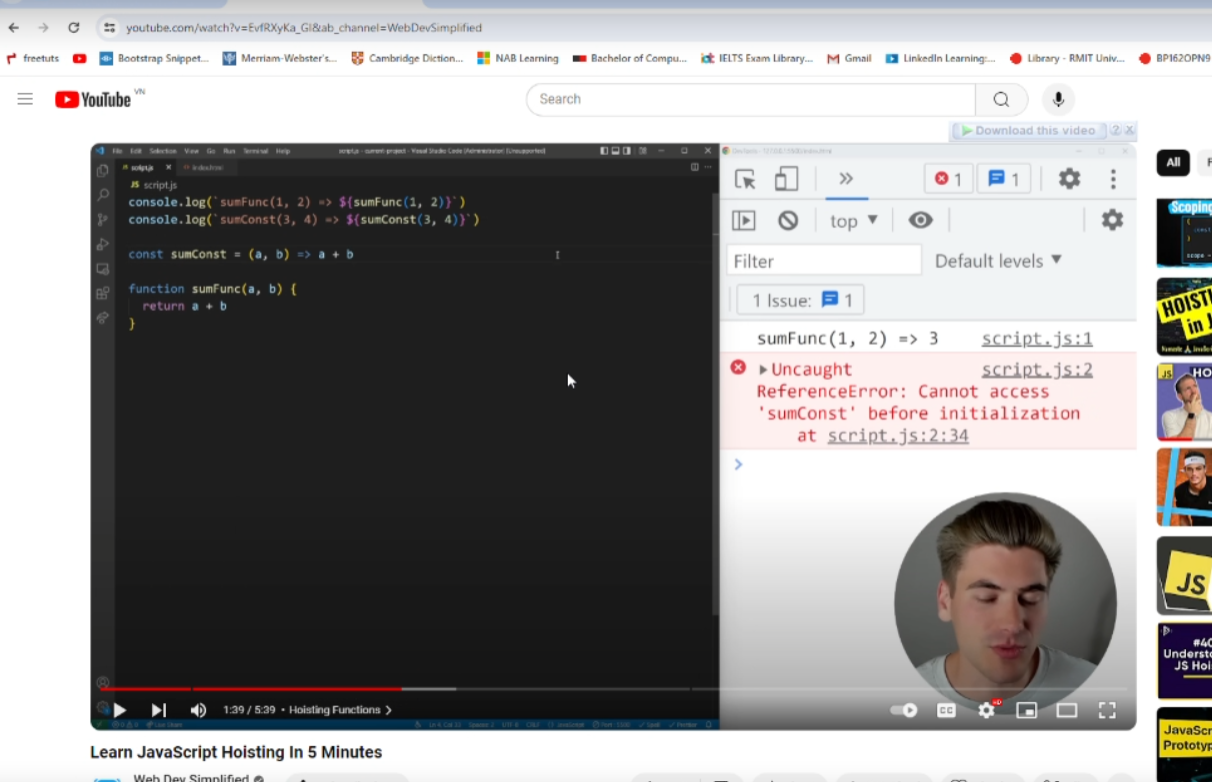
}

debug()() // đáp án là 7

Definition ,: La mot ham co the truy cap duoc bien ben ngoai pham vi cua no, co the ghi nho duoc noi no tao ra, the hien tinh encapsulation trong oop (tai vi chi khi goi thang cha, moi co the truy cap va chinh sua thang con ben trong).

# Hoisting

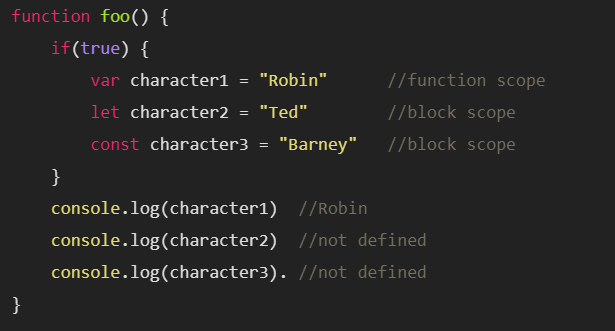
Hoisting is a concept or behavior in JavaScript where the declaration of a function, variable, or class goes to the top of the scope they were defined in.

****

**Function: function bth thì sẽ có hoisting => ra kết quả của function đó luôn, còn const thì ko => nó sẽ ra lỗi**

**var** variables are hoisted but with a default value of **undefined**. **let** and **const** variables, and classes are hoisted but inaccessible as they do not have a default initialization.

# Let, var , const

****

**Var**

* Được khởi tạo với giá trị undefined. (hoisting)
* Phạm vi hàm (function scope) hoặc phạm vi toàn cục (global scope)
* Có thể gán lại (Re-assignable) và có thể khai báo lại (Re-declarable)
* Không thuộc vùng chết tạm thời (Temporal Dead Zone - TDZ)

**Let**

* Được khởi tạo ở đầu của scope ( hoisting) nhưng không có giá trị khởi tạo, nên khi console.log trước khi gán giá trị thì sẽ bị error
* Phạm vi khối (Block scope)
* Có thể gán lại nhưng không thể khai báo lại
* Có thể khai báo lại được nếu khác scope
* Phụ thuộc vào vùng chết tạm thời (TDZ)

**Const**

* Block scope and hoisting
* Nếu console.log trc khi khai báo, ra ra lỗi error   
  // ReferenceError: Cannot access 'name' before initialization
* Không được sửa giá trị của const (với biến primitives) - string, number, boolean, null, undefined.
* Vẫn được sửa giá trị của const (với biến tham chiếu) - object, array, function

# === vs ==

| **==** | **equal to** | **x == 8** | **false** |  |
| --- | --- | --- | --- | --- |
| **x == 5** | **true** |  |
| **x == "5"** | **true** |  |

| **===** | **equal value and equal type** | **x === 5** | **true** |  |
| --- | --- | --- | --- | --- |
| **x === "5"** | **false** |  |

**Mo rong**

**var a = [];**

**var b = [];**

**var c = a;**

**So sanh dia chi vi array luu dia chi**

**console.log(a == b); // false**

**console.log(a === b); // false**

**console.log(a == c); // true**

**console.log(a === c); // true**

**STREAM**

**Co 4 loai stream**

1. **Readable**
2. **Writable**
3. **Complex : R + W**
4. **Transform: change data**

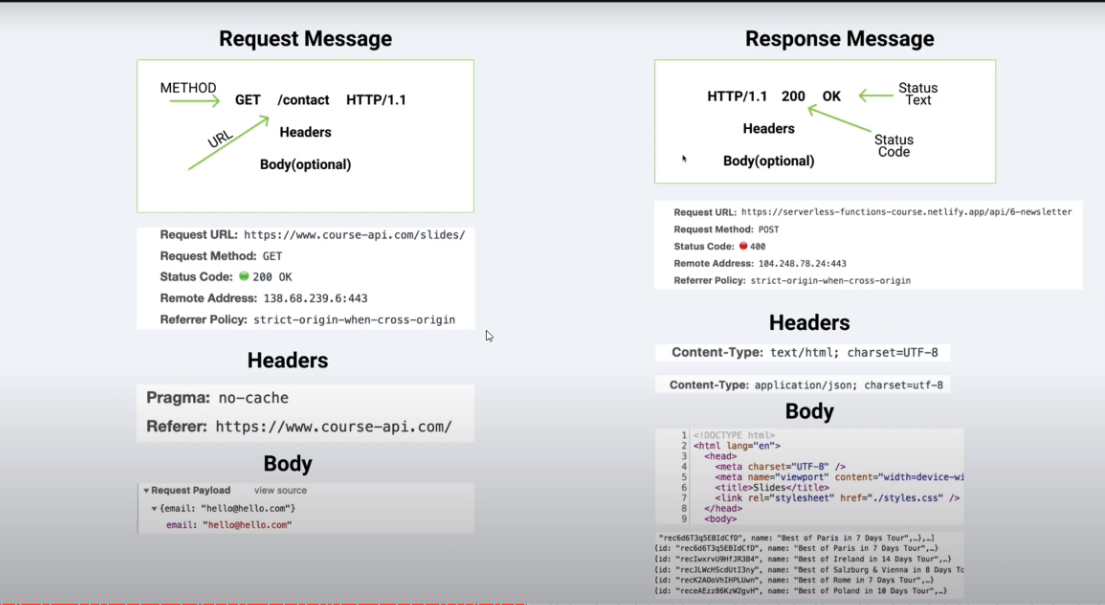
**HOW DOES IT WORK**

****

**String -> JSON -> JSON.parse();**

**JSON -> string -> JSON.stringify();**

**Local storage la luu duoi dang string nen muon dung local thi phai stringigify**

****

****

**ENV**

**Deploy bien env tren heruku**

**EVENT LOOP**

[**https://www.youtube.com/watch?v=IGkz8Zu9rRk**](https://www.youtube.com/watch?v=IGkz8Zu9rRk)

**Body-parser**

Dùng để parse json từ req.body. Ví dụ:

import bodyParser from "body-parser";

app.use( .json());

app.get("/", (req, res)=>{

if(req.body.name){

res.send(req.body.name)

}else{

res.send("ga")

}

})

**Express**

Expressjs là một framework được xây dựng trên nền tảng của Nodejs. Nó cung cấp các tính năng mạnh mẽ để phát triển web hoặc mobile. Expressjs hỗ trợ các method HTTP và midleware tạo ra API vô cùng mạnh mẽ và dễ sử dụng.

**Route**

**express.router()**

**giúp** chia route dễ dàng hơn, quản lý code của mình dễ hơn, project lớn thì sẽ cần nhiều dòng lệnh route cho mỗi đối tượng, file index.js sẽ cực kì lớn và khó kiểm soát, vì vậy cần sử dụng router để bóc tách thành các file nhỏ, mỗi file sẽ ứng với mỗi route riêng

**Jwt**

**JWT** la mot tieu chuan giup tao ra chuoi ma hoa ma no co chua du lieu de giup ban co the trao doi thong tin giua nhung he thong khac nhau mot cach an toan

JWT gom 3 phan header, payload va signature

Phần **header** sẽ chứa kiểu dữ liệu , và thuật toán sử dụng để mã hóa ra chuỗi JWT.

Phần **payload** sẽ chứa các thông tin mình muốn đặt trong chuỗi **Token** như **username** , **userId** , **author**.

Phan signature này sẽ được tạo ra bằng cách mã hóa phần **header** , **payload** kèm theo một chuỗi **secret (khóa bí mật)**

Dung **base64UrlEncoder :** thuật toán mã hóa **header** và **payload**

**Cach hoat dong cua jwt**

User dang nhap usernam password gui len server , server se tao ra jwt token va gui lai cho client

Client se luu jwt token nay vao localstorage

Sau do moi lan muon gui len sever de lay du lieu thi phai luon dinh kem jwt token (bearer) nay de server check lai coi jwt token nay da dung hay chua

Neu hop le tra ve du lieu , con khong tra ve fail

**Middleware**

ExpressJS khi hoạt động, về cơ bản sẽ là một loạt các hàm Middleware được thực hiện liên tiếp nhau. Sau khi đã thiết lập, các request từ phía người dùng khi gửi lên ExpressJS sẽ thực hiện lần lượt qua các hàm Middleware cho đến khi trả về response cho người dùng. Các hàm này sẽ được quyền truy cập đến các đối tượng đại diện cho Request - req, Response - res, hàm Middleware tiếp theo - next, và đối tượng lỗi - err nếu cần thiết.

Một hàm Middleware sau khi hoạt động xong, nếu chưa phải là cuối cùng trong chuỗi các hàm cần thực hiện, sẽ cần gọi lệnh next() để chuyển sang hàm tiếp theo, bằng không xử lý sẽ bị treo tại hàm đó.

Các chức năng mà middleware có thể thực hiện trong ExpressJS sẽ bao gồm :

* Thực hiện bất cứ đoạn code nào
* Thay đổi các đối tượng request và response
* Kết thúc một quá trình request-response
* Gọi hàm middleware tiếp theo trong stack

Trong Express, có 5 kiểu middleware có thể sử dụng :

* Application-level middleware (middleware cấp ứng dụng)
* Router-level middleware (middlware cấp điều hướng - router)
* Error-handling middleware (middleware xử lý lỗi)
* Built-in middleware (middleware sẵn có)
* Third-party middleware (middleware của bên thứ ba)

**Cors**

La protocal cho phep script chay tren browser client tuong tac voi cac resource cua orgin khac

Khac origin la khi ma khac ve giao thuc (http, https), domain , khac ve port

Vi du

Vi du co app client frontend ( React) ma muon gui request den cho server( node express). Thi luc goi no se khac domain nen chac chan se khong goi duoc. Khi do phai dung cors

Cors phai bo vo noi chua resource( server), no su dung cors de cho phep ai do co the lay duoc cai resoure cua no

**Mongoose**

La mot thu vien de co the su dung mongodb mot cach de dang hon

**App.use**

Để setup middleware cho application

app.use(path, middleware)

Path: đường dẫn mà sẽ apply middleware

middleware : callback, function hoặc array of middleware

**Mongo**

**Env**

Như mình đã đề cập đến ở phần trên thì proccess.env thường được dùng để lưu trữ các biến trong từng môi trường làm việc.

Ví dụ như lúc trên local bạn sử dụng port 3000 để chạy dự án, nhưng khi deloy trên thực tế thì bạn muốn dùng port 4000 chẳng hạn. Lúc này bạn chỉ cần thay đổi giá trị của port trong biến proccess.env. Biến process.env còn được *dùng để lưu trữ các thông tin cần giữ bí mật trong quá trình phát triển ứng dụng như username, password của database,....*

*Trong dự án thực tế, các đoạn mã nodejs cần phải chạy trên các môi trường khác nhau như (Dev, Test, Product), mỗi môi trường có các cấu hình khác nhau. Bởi vậy, nodejs cung cấp cho chúng ta cách thức để tùy chỉnh các biến giúp chạy trên các môi trường khác nhau đó.*

**Router**

**Cookie HTML**

**express.json**

Là built-in middleware của express. khi dùng các phương thức post hay put thì hay kèm theo body. middleware này giúp đọc được cái body được gửi đi đó dưới format Json. và nếu muốn lấy dữ liệu thì dùng req.body

**localStorage, sessionStorage**

**OOp trong javascript**

**Tính đa hình** là khả năng một đối tượng có thể thực hiện một tác vụ theo nhiều cách khác nhau. Ngta thuong dung overriding de the hien tinh da hinh.

**Tinh dong goi** là để data hiding(che giấu thông tin), được thể hiện bằng private, tất cả các thông tin này sẽ không truy cập trực tiếp từ bên ngoài, chỉ có thể truy cập qua các phương thức public như là get hoặc set.

**Tinh truu truong** là mở rộng của tính đóng gói, người dùng sẽ không cần quan tâm đến việc bên trong code như thês nào mà chỉ quan tâm đến cách sử dụng ra sao

**abstract class**

có method and attribute như class bth, nhưng mà có thêm các abstract method, abstract method sẽ không cần phải specify the body, không thể initialize an instance, cung cấp cái khung, tức là khi nhìn vào thì sẽ mường tượng ra được là có những thông tin, method như này thôi, chứ không biết nó được implementation như thế nào.

Vì không thể tạo ra instance, để dùng được thì phải tạo ra một class mà kế thừa cái abstract class. Cái class khi kế thừa bắt buôc phải implement những cái abstract method mà abstract class có.

Quan hệ is a

final khong the overide nen abstract method khong co final

**Interface**

tất cả các method trong interface đều được coi là abstract method và public, không cần ghi keyword abstract trước method

trong interface khi declare các attribute thì nó đều là public by default ,và là final và static ( tức là có giá trị khởi tạo)

ví dụ int age = 15, nếu viết int age; sẽ báo lỗi

lớp có thể implement nhiều interface

interface có thể extend multiple interface nhưng k extend được class

quan hệ has a

Overloading thêm mở rộng hành vi của phương thức, tính đa hình compile time

**Nodemon**

Chay lai web server khi co thay doi trong code

**Local - global npm install**

local packages are installed in the directory where you run npm install <package-name>, and they are put in the node\_modules folder under this directory

global packages are all put in a single place in your system (exactly where depends on your setup), regardless of where you run npm install -g <package-name>

A package should be installed globally when it provides an executable command that you run from the shell (CLI), and it's reused across projects.

Vi du nhu:

npm

create-react-app

vue-cli

grunt-cli

mocha

react-native-cli

gatsby-cli

forever

Nodemon

Neu khong install nodemon global, thi ghi command line la nodemon <project nao do> thi se khong dung duoc, neu muon dung thi project do phai co nodemon trong dependency ( tuc la cai npm install local)

**Npm**

Node package management dung de quan ly cac package cua node

**devDependecy- dependency package.json**

devDependency la chi for development phase, luc ma deploy len sever thi no se khong install devDependency

**--save trong npm install**

--save y o day la phien ban cu cua node, neu khong co --save, package.json se khong co module do

**Public**

Public dung de chua static asset, static asset la nhung file ma server khong can phai thay doi, vi nhu hinh anh , file style, file javascript.

**datastructure**

[**https://www.youtube.com/watch?v=YYta5uWViJc**](https://www.youtube.com/watch?v=YYta5uWViJc)

**Pv react**

[**https://www.youtube.com/watch?v=zbg-SERx6Ig**](https://www.youtube.com/watch?v=zbg-SERx6Ig)

**Cau hoi co the ra :**

1. NodeJs vs ExpressJs khac gi nhau ?
2. Nodejs la gi ?

La moi truong giup chay dc Javascript ben ngoai browser.

1. Node vs Browser

https://codelearn.io/sharing/moi-truong-node-js-browser-va-khac-biet#:~:text=S%E1%BB%B1%20kh%C3%A1c%20bi%E1%BB%87t%20gi%E1%BB%AFa%20hai,th%E1%BB%B1c%20hi%E1%BB%87n%20%E1%BB%9F%20ph%C3%ADa%20client.&text=%E1%BB%9E%20Browser%2C%20b%E1%BA%A1n%20c%C3%B3%20th%E1%BB%83,%2C%20sessionStorage%2C..).