[latentflip.com/loupe/](http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXIoKSB7CiAgICAgICAgY29uc29sZS5sb2coJ1lvdSBjbGlja2VkIHRoZSBidXR0b24hJyk7ICAgIAogICAgfSwgMjAwMCk7Cn0pOwoKY29uc29sZS5sb2coIkhpISIpOwoKc2V0VGltZW91dChmdW5jdGlvbiB0aW1lb3V0KCkgewogICAgY29uc29sZS5sb2coIkNsaWNrIHRoZSBidXR0b24hIik7Cn0sIDUwMDApOwoKY29uc29sZS5sb2coIldlbGNvbWUgdG8gbG91cGUuIik7!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D) (test JS event loop)

Contents

[**1.** **Prototype** 2](#_Toc73887600)

[**2.** **\_\_proto\_\_** 2](#_Toc73887601)

[**3.** **Reference Vs Value (tham chiếu, tham trị)** 3](#_Toc73887602)

[**4.** **this** 3](#_Toc73887603)

[**5.** **Arrow function** 5](#_Toc73887604)

[**6.** **context and scope** 6](#_Toc73887605)

[**7.** **Curry function** 6](#_Toc73887606)

[**8.** **Closures** 8](#_Toc73887607)

[**9.** **Promise** 8](#_Toc73887608)

[**10.** **Async/await** 9](#_Toc73887609)

[**11.** **Find/ filter (array methods)** 10](#_Toc73887610)

[**12.** **Switch...case** 11](#_Toc73887611)

[**13.** **Object Literal lookups** 13](#_Toc73887612)

1. **Prototype**

Dùng trong **function constructor**

function Dog(name) { // the usual constructor function

this.name = name

}

Dog.prototype = ...

**prototype** is the object that is used to build \_\_proto\_\_ when you create an object with new.

**prototype** is **not available** on the instances themselves (or other objects), but only on the constructor functions.

ứng dụng của **prototype**

nếu muốn add thêm method method/property vào 1 cái global object thì dùng prototype constructor này.



Object Array chưa có myUcase nên có thể add thêm custom method myUcase này vào constructor. Khi các biến khác sử dụng array này thì sẽ có method myUcase.

1. **\_\_proto\_\_**

Dùng trong **instance** của function constructor

function Dog(name) { // the usual constructor function

this.name = name

}

Let a = new Dog(‘Hector’);

a.\_\_proto\_\_ = ....

Add method/property vào function constructor

**\_\_proto\_\_** is available everywhere.

1. **Reference Vs Value (tham chiếu, tham trị)**



Khi gán d = c thì nó chỉ gán địa chỉ/address của c thôi chứ ko gán value của c

Khi ==, === thì nó chỉ so sánh địa chỉ/address thôi chứ ko so sánh value

function add(arr, el) { arr.push(el)} =>Khi truyền input as array thì nó cũng sẽ lấy địa chỉ/address rồi implement chứ ko lấy giá trị của array

function add(arr, el) { arr = [el] } => Nhưng khi gán nó = 1 array khác thì nó sẽ có địa chỉ khác

nếu declare **const arr = [1,2,3]**

thì **vẫn modified được bằng** các phương thức của arr(chứ **ko phải phép gán**). Vì nó ko làm thay đổi địa chỉ/address của const này. Và const obj cũng chỉ lưu địa chỉ/address

this

this (aka "the context") is a special keyword inside each function: giá trị của this chỉ **phụ thuộc** cách function được **gọi/called** (runtime binding)chứ **ko** phải cách nó được định **nghĩa/defined**

1. **this**

theo Mozilla:

*In most cases, the value of this is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called.*

Được định nghĩa khi function được gọi chứ ko phải khi set/define this. (this nó đi theo tiếng gọi của function). Nên ai gọi nó thì nó sẽ đi theo người đó.

Trong Literal Object

const c = {

        name: 'c',

        go: function () {

            console.log(this)

        },

    }

    c.go()

khởi tạo 1 const c và gọi: c.go() thì **this** sẽ đi theo c

 let { go, name } = c

    go()

còn cách trên là dùng destructuring để lấy các properties và methods trong c ra. Cách này giúp gọn code hơn nhưng có vấn đề:

property name vẫn là của c (ok)

method go vẫn trỏ đến c, nhưng rõ ràng khi ta gọi go() thì go() nó được Window object gọi (vì ko có . ở trước nên mặc định là Window) (failed)

=> ko nên dùng destructuring với methods.

 const c = {

        name: 'c',

        numbers: [1, 2, 3, 4, 5],

        onCount: function () {

            this.numbers.forEach(function(n) {

                console.log(this.name + ' - ' + n)

            })

        }

    }

c.onCount()

c.onCount() thì có vấn đề: trong .forEach là 1 callback function.(console.log(this) trước hàm forEach thì this vẫn là c) Theo Mr.Mosh: function thì sẽ trỏ tới Window

khắc phục: dùng arrow function: (n) => console.log(n) hoặc tạo 1 biến trong hàm onCount let \_this = this

Trong Constructor Function

function A() {

        this.name = 'A';

        this.say = function () {

            console.log(this)

        }

    }

    let m = new A();

    m.say()

dùng từ khóa new để tạo 1 instance từ A

m.say() thì say() sẽ trỏ tới m, m là instance của A nên this trỏ tới A.

Trong Class

class B {

        constructor(name, age) {

            this.name = name;

            this.age = age

        }

        onSayHello() {

            console.log(`Hello I am ${this.name} - ${this.age}`)

        }

    }

    let h = new B('hector', 22);

    h.onSayHello()

tạo instance từ B rồi gọi onSayHello() thì this sẽ trỏ tới h, h là instance từ B nên trở đến B

\* Trong Callback

vì Callback là hàm gọi ko có dấu . nên rõ ràng là context sẽ là Window

\* Trong SetTimeout



(context khác với cái hàm gọi setTimeout => this mặc định là window or global)



setTimeout chỉ lấy cái reference của myArray.myMethod thôi. Còn việc gọi là khác(có thể là gọi kiểu callback()) nên this sẽ khác

khắc phục:



Tạo 1 anonymous function trong setTimeout. Lúc này thì myArray.myMethod() được gọi là trước dấu . là myArray nên this sẽ là myArray.

1. **Arrow function**



Vì AF nó sẽ tạo this dựa trên scope mà AF được định nghĩa trong cái scope đó(nó sẽ định nghĩa ở cái scope cha của AF)





1. **context and scope**

The context is (roughly) the **object** that **calls** the function. And the scope is all the **variables** visible to a function **where it is defined**. **context** cares about how it is called, **scope** cares about how it is defined.

1. **Curry function**

var add = function (a){

return function(b){

return function(c){

return a+b+c;

}

}

}

console.log(add(2)(3)(4)); *//output 9*

*// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ES6 \*\*\*\*\*\*\*\*\*\*\*\*\**

const add = a => b => a + b

add(1)(2) *//should return 3*

công dụng:

có thể dùng 1 param mặc định rồi trả về 1 function. Dùng tiếp 1 param để đưa vào function này.

Để tránh việc lặp lại code trong cái return function. Cho 2 arrays

const movies = [

{

"id": 1,

"name": "Matrix"

},

{

"id": 2,

"name": "Star Wars"

},

{

"id": 3,

"name": "The wolf of Wall Street"

}

]

const series = [

{

"id": 4,

"name": "South Park"

},

{

"id": 5,

"name": "The Simpsons"

},

{

"id": 6,

"name": "The Big Bang Theory"

}

]

Muốn lấy id thì phải định nghĩa trong callback 2 lần

movies.map((movie) => movie.id) *//should return [ 1, 2, 3 ]*

series.map((serie) => serie.id) *//should return [ 4, 5, 6 ]*

curry function: (định nghĩa 1 lần)

const get = property => object => object[property];

const getId = get('id');

movies.map(getId); *//should return [ 1, 2, 3 ]*

series.map(getId); *//should return [ 4, 5, 6 ]*

* Function được return thì chỉ nhận duy nhất 1 tham số 1 lần.

1. **Closures**

 a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Currying functions là có sử dụng closures. Vì nó sử dụng biến của function cha để xử lý.

Every closure has three scopes:

* Local Scope (Own scope)
* Outer Functions Scope
* Global Scope

Closure trong loop.





Lúc này console.log(i) = 5 . Vì nó sử dụng biến var và var có thể dùng cho global scope nên bị thay đổi.

Three closures have been created by the loop, but each one shares the same single lexical environment

1. **Promise**

A Promise is in one of these states:

* pending: initial state, neither fulfilled nor rejected.
* fulfilled: meaning that the operation was completed successfully.
* rejected: meaning that the operation failed.

Khi fulfilled xong thì gọi .then(value => ...) để xử lý tiếp. Cái việc nó fulfilled như thế nào thì ko cần quan tâm

Rejected cũng vậy, ko quan tâm rejected như thế nào dùng .catch(err => ...) để show err

let myFirstPromise = new Promise((resolve, reject) => {

setTimeout( function() {

resolve("Success!") // Yay! Everything went well!

}, 250)

})

myFirstPromise.then((successMessage) => {

console.log("Yay! " + successMessage)

});

Khởi tạo 1 promise. Promise nhận 1 callback. Callback này nhận 2 input resolve, reject.

Trong callback thực thi

* resolve() thì .then().
* reject() thì .catch()

1. **Async/await**

async function f3() {

var y = await 20;

console.log(y); // 20

}

f3();

function nào có chứa await thì function đó sẽ phải có async.

Async function trả về promise. Nên khi gọi hàm này thì dùng **then**/**await** để nó run đúng thứ tự. Còn nếu hàm này ở cuối thì khỏi **then**/**await.**

async function f4() {

try{var z = await Promise.reject(30); }

catch(e) {console.error(e); // 30}

}

f4();

khi dùng async/await thì dùng thêm try...catch để bắt lỗi.

Top level await (dùng await mà ko cần async)

You can use the await keyword on its own (outside of an async function) within a [JavaScript module](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules). This means modules, with child modules that use await, wait for the child module to execute before they themselves run. All while not blocking other child modules from loading.

Downside of async/await

Khi await thì cái sau phải đợi cái trước

function timeoutPromise(interval) {

return new Promise((resolve, reject) => {

setTimeout(function(){

resolve("done");

}, interval);

});

};

async function timeTest() {

await timeoutPromise(3000);

await timeoutPromise(3000);

await timeoutPromise(3000);

}

* tốn 9s
* trick:

async function timeTest() {

const timeoutPromise1 = timeoutPromise(3000);

const timeoutPromise2 = timeoutPromise(3000);

const timeoutPromise3 = timeoutPromise(3000);

await timeoutPromise1;

await timeoutPromise2;

await timeoutPromise3;

}

Here we store the **three Promise objects** in variables, which has the effect of setting off their associated processes all running simultaneously. because the promises all started processing at essentially(bản chất) the same time, the promises will all fulfill at the same time. => tốn 3s

1. **Find/ filter (array methods)**

2 methods này sẽ trả về item trong array đó nếu thỏa điều kiện.

Vì vậy, 2 methods này sẽ trỏ đến item trong array đó. => nếu update những item này thì sẽ vô tình update lun các item trong array gốc.

Solution: clone/cloneDeep array gốc trước khi dùng find/filter.

**Clone:** [...arr] thì chỉ copy array sang vùng nhớ khác. Còn các object item trong arr vẫn trỏ chung 1 địa chỉ. Cơ bản là vẫn xài các method của array và ko ảnh hưởng đến arr gốc. Nên, để mutate 1 object item trong [...arr] thì

clone cObj = {...arr[0]} ra,

reassign cái cObj.id = 123,

gán lại arr[0] = cObj.



**cloneDeep:** cả arr thì nó mới thành 1 cái array mới hoàn toàn.









1. **Switch...case**



Dùng để check input. Nếu input === với case nào thì call case đó.

Có thể đặt biến trong case scope



Pros: nên dùng khi có “fall through” là phù hợp nhất.



Cons: switch thường được viết trong 1 hàm, phải có breaks, và phải có default, phải return giá trị và

1. **Object Literal lookups**



Có thể dùng Object để thay thế switch... case.

Switch...case và Object có thể dùng với function



(trong hình là Object nhưng switch vẫn có thể làm được)

Pros: dùng trong trường hợp 1 dòng hoặc ít line, ko cần phải return giá trị, có thể viết thằng kiểu Object ko cần bao trong 1 hàm

Cons: Object khi có “fall through”(có nhiều case áp dụng 1 kết quả) thì Object phải duplicate hàm, ko thể đặt biến được( phải tạo ra 1 function và nhớ call function đó)

