**JAVASCRIPT**

# Toán tử ++ và --

Cập nhật tháng 3 năm 2022

Đây là 2 toán tử nghe qua thì rất dễ hiểu, nhưng để hiểu nguyên lý về cách hoạt động của nó chúng ta sẽ phải mất thêm một chút thời gian đó. Để hoàn thành bài học về 2 toán tử này, chúng ta sẽ cùng trải qua một số bài học sau nhé.

Ok, bắt đầu thôi!

## Toán tử ++

Toán tử ++ giúp tăng giá trị của một biến mang giá trị số lên 1. Có 2 cách để sử dụng toán tử ++ là:

1. Dùng làm hậu tố: variable++ (toán tử nằm sau biến)
2. Dùng làm tiền tố: ++variable (toán tử nằm trước biến)

### #1 Sử dụng ++ làm hậu tố

Ở đây, chúng ta sẽ xét ví dụ sử dụng toán tử ++ làm hậu tố trước (vì trong thực tế, chúng ta thường dùng kiểu hậu tố nhiều hơn):

var number = 1;

number++; // dùng làm hậu tố, ++ ở phía sau biến

console.log(number); // 2

number++;

console.log(number); // 3

Sau mỗi khi sử dụng toán tử ++, giá trị của biến number được tăng lên 1. Có vẻ khá dễ dàng để hiểu cách hoạt động của nó phải không?

Tuy nhiên, hãy xem xét thêm ví dụ sau:

var number = 1;

console.log(number++); // 1

console.log(number); // 2

console.log(number++); // 2

console.log(number); // 3

👉 Toán tử ++ khi dùng là hậu tố sẽ **tăng giá trị của biến lên 1** và **trả về giá trị trước khi tăng**.

### #2 Sử dụng ++ làm tiền tố

Ở ví dụ này, chúng ta sử dụng ++ làm tiền tố. Tuy nhiên, kết quả trông sẽ không khác gì khi dùng ++ làm hậu tố:

var number = 1;

++number; // dùng làm tiền tố, ++ ở phía trước biến

console.log(number); // 2

++number;

console.log(number); // 3

Copy

Nhưng khi xem xét kỹ hơn, các bạn sẽ nhìn ra điểm khác:

var number = 1;

console.log(++number); // 2

console.log(number); // 2

console.log(++number); // 3

console.log(number); // 3

Copy

👉 Toán tử ++ khi dùng là tiền tố sẽ **tăng giá trị của biến lên 1** và **trả về giá trị sau khi tăng**.

Toán tử - -

Cách hoạt động tương tự như toán tử ++, điểm khác biệt là thay vì cộng thêm 1, thì toán tử -- sẽ trừ đi 1.

Tổng kết

* x++ tăng giá trị biến lên 1 và trả về giá trị **trước** khi tăng
* ++x tăng giá trị biến lên 1 và trả về giá trị **sau** khi tăng
* x-- giảm giá trị biến xuống 1 và trả về giá trị **trước** khi giảm
* --x giảm giá trị biến xuống 1 và trả về giá trị **sau** khi giảm.

# Truthy và Falsy là gì?

## Truthy - to bool is true

Bất cứ giá trị nào trong Javascript khi chuyển đổi sang kiểu dữ liệu boolean mà có giá trị true thì ta gọi giá trị đó là Truthy.

Các giá trị 1, ['BMW'], { name: 'Miu' } và 'hi' được đề cập trong ví dụ dưới đây là Truthy vì khi chuyển sang Boolean ta nhận được giá trị true.

Ví dụ:

console.log(Boolean(1)) // true

console.log(Boolean(['BMW'])) // true

console.log(Boolean({ name: 'Miu' })) // true

console.log(!!'hi') // true

Copy

!! là gì? Đơn giản thôi. Toán tử ! là toán tử not (phủ định) nên !! là 2 lần phủ định, mà 2 lần phủ định lại trở thành "khẳng định". Trong Javascript thì đây là một "tip" để convert (chuyển đổi) mọi kiểu dữ liệu khác sang Boolean.

Ví dụ:

console.log(!!1) // true

console.log(!!'f8') // true

console.log(!!['Mercedes']) // true

Copy

Thêm !! phía trước các giá trị truthy sẽ luôn trả về true.

## Falsy - to bool is false

Bất cứ giá trị nào trong Javascript khi chuyển đổi sang kiểu dữ liệu boolean mà có giá trị false thì ta gọi giá trị đó là Falsy.

Trong Javascript có 6 giá trị sau được coi là Falsy:

1. false
2. 0 (số không)
3. '' or "" (chuỗi rỗng)
4. null
5. undefined
6. NaN

Ví dụ:

console.log(!!false) // false

console.log(!!0) // false

console.log(!!'') // false

console.log(!!null) // false

console.log(!!undefined) // false

console.log(!!NaN) // false

Copy

Chú ý!

Nội dung đã đề cập phía trên đã đầy đủ khi nói về Truthy và Falsy trong Javascript. Tuy nhiên mình vẫn cần nhấn mạnh lại với các bạn rằng:

Ngoài 6 giá trị đã đề cập tới ở phần Falsy thì toàn bộ các giá trị khác đều là Truthy, kể cả những giá trị sau:

1. '0' (một chuỗi chứa số không)
2. ' ' (một chuỗi chứa dấu cách)
3. 'false' (một chuỗi chứa từ khóa false)
4. [] (một array trống)
5. {} (một object trống)
6. function(){} (một hàm "trống")

Một số người chuyển từ ngôn ngữ khác sang rất có thể sẽ bị nhầm [] (mảng "rỗng") là falsy, bởi vì trong ngôn ngữ họ đã học trước đó [] là falsy.

Với những người hiểu nhầm [] là falsy sẽ gặp trường hợp khó hiểu sau:

Ví dụ:

var cars = [] // Dù là mảng "rỗng" vẫn là truthy

if (!cars) {

// Họ sẽ thắc mắc: "Tại sao lại không lọt vào đây?"

}

Copy

Vì [ ] là truthy nên !cars sẽ trả về false. Câu lệnh if sẽ nhận được kết quả của mệnh đề so sánh là false, vì vậy đoạn mã trong if trên sẽ không được lọt vào.

# Ngoại lệ? - document.all

Trong Javascript (phía trình duyệt) sẽ có sẵn một đối tượng document, và khi bạn thử !!document.all sẽ trả về false. Chẳng lẽ document.all cũng là falsy hay sao?

Bản thân mình cũng thắc mắc điều này nên mình đã search Google "Why document.all is falsy?" và mình đã tìm được câu trả lời [tại đây](https://fullstack.edu.vn/external-url?continue=https%3A%2F%2Fstackoverflow.com%2Fquestions%2F10350142%2Fwhy-is-document-all-falsy).

**Tóm tắt câu trả lời:**

document.all là một ngoại lệ chính thức duy nhất theo đặc tả ECMA (phiên bản 5). Đặc tả này mô tả toàn bộ các object khi chuyển sang boolean sẽ là true. Tuy nhiên, document.all là một ngoại lệ.

**Cụ thể như sau:**

1. document.all chuyển sang boolean sẽ là false
2. document.all khi là toán hạng của toán tử so sánh == hoặc != sẽ là undefined
3. Khi typeof document.all sẽ trả về "undefined"

ECMA là đặc tả chi tiết kỹ thuật mà các ngôn ngữ theo đặc tả này phải tuân theo. Javascript là một ngôn ngữ tuân thủ đặc tả kỹ thuật ECMA.

# Kiểu dữ liệu số (number)

## Kiểu số - Number

Trong Javascript có 2 loại số:

1. Các số thông thường trong Javascript được lưu trữ dưới dạng 64 bit IEEE-754 (còn được gọi là các số phẩy động). Trong hầu hết các trường hợp khi làm việc với Javascript là chúng ta sử dụng loại số này.
2. Số BigInt là loại số sử dụng để biểu thị số nguyên có độ dài tùy ý.

Vì mức độ thông dụng nên ở đây chúng ta chỉ tìm hiểu về loại số thông thường.

## Cách khai báo

Cách thông thường khi ta khai báo một số. Ví dụ là: 1000000 (một triệu)

var million = 1000000;

Copy

Cũng là khai báo số 1000000 nhưng có cách viết khác. Bạn có thể thêm chữ e vào sau số 1 và chỉ định số số không phía sau chữ e như sau:

var million = 1e6; // tương tự: 1000000

// hoặc

var billion = 2e9; // tương tự: 2000000000 (hai tỏi à nhầm hai tỉ)

Copy

## Đối tượng Number

Đối tượng Number trong Javascript là đối tượng giúp chúng ta định nghĩa số và làm việc với số. Chúng ta thường sử dụng các phương thức sau của đối tượng Number khi làm việc với số trong Javascript:

| **Phương thức** | **Vai trò** |
| --- | --- |
| Number.isFinite() | Xác định xem giá trị đã cho có phải là số hữu hạn hay không. Trả về boolean |
| Number.isInteger() | Xác định xem giá trị đã cho có phải là số nguyên hay không. Trả về boolean |
| Number.parseFloat() | Chuyển đổi chuỗi đã cho thành một số dấu phẩy động |
| Number.parseInt() | Chuyển đổi chuỗi đã cho thành một số nguyên |
| Number.prototype.toFixed() | Chuyển đổi và trả về chuỗi đại diện cho số đã cho, có số chữ số chính xác sau dấu thập phân |
| Number.prototype.toString() | Chuyển đổi và trả về số đã cho dưới dạng chuỗi |

Ví dụ:

Number.isFinite(2 / 0); // false

Number.isFinite(20 / 5); // true

Number.isFinite(0 / 0); // false

Number.isInteger(999999999); // true

Number.isInteger(0.2); // false

Number.isInteger(Math.PI); // false

Number.parseFloat('10') // 10

Number.parseFloat('10.00') // 10

Number.parseFloat('238,21') // 238

Number.parseFloat('237.22') // 237.22

Number.parseFloat('34 56 78') // 34

Number.parseFloat(' 37 ') // 37

Number.parseFloat('18 is my age') // 18

Number.parseInt('10') // 10

Number.parseInt('10.00') // 10

Number.parseInt('238,21') // 238

Number.parseInt('237.22') // 237

Number.parseInt('34 56 78') // 34

Number.parseInt(' 37 ') // 37

Number.parseInt('18 is my age') // 18

var numberObject = 1234.56789;

numberObject.toFixed(); // '1235'

numberObject.toFixed(1); // '1234.6'

numberObject.toFixed(6); // '1234.567890'

(11).toString(); // '11'

(18).toString(); // '18'

(17.3).toString(); // '17.3'

# Khai báo biến

**var** : Biến được khai báo bằng **var** có phạm vi toàn cục hoặc phạm vi hàm (function scope). Điều đặc biệt về **var** là nó không bị ràng buộc bởi khối lệnh (block scope), nghĩa là biến khai báo bằng **var** có thể truy cập từ bất kỳ đâu trong hàm chứa nó.

function example() {

    var x *=* 10;

*if* (true) {

      var y *=* 20;

      console.log(x); *// 10*

    }

    console.log(y); *// 20*

}

**let** : **let** là từ khóa được giới thiệu trong ECMAScript 6 (ES6) để khai báo biến với phạm vi khối lệnh (block scope). Biến được khai báo bằng **let** chỉ có thể truy cập trong khối lệnh mà nó được khai báo.

function example() {

    let x *=* 10;

*if* (true) {

      let y *=* 20;

      console.log(x); *// 10*

    }

    console.log(y); *// Error: y is not defined*

}

**const** : **const** cũng là từ khóa được giới thiệu trong ES6 để khai báo biến. Biến được khai báo bằng **const** cũng có phạm vi khối lệnh (block scope), nhưng có một điểm khác biệt quan trọng là giá trị của biến **const** không thể thay đổi sau khi được gán giá trị ban đầu. Nó tạo ra một biến hằng (constant).

function example() {

    const x *=* 10;

    x *=* 20; *// Error: Assignment to constant variable*

}

# Chuỗi trong JavaScript

search : backslash in javascript

var fullName *=* 'Vu Duc Tien';

var firstName *=* 'Vu';

var lastName *=* 'Tien';

console.log(fullName.length);

console.log('Toi la: ' *+* firstName *+* ' ' *+* lastName);

console.log(`Toi la: ${firstName} ${lastName}`);

keywords : javascript string method

var myString *=* '    Học JS tai F8!    ';

var myString *=* 'Học JS tai JS JS F8!';

1.length ( độ dài chuỗi )

2.find index ( tìm kiếm )

console.log(myString.indexOf('JS'));

console.log(myString.indexOf('JS', 5));

console.log(myString.lastIndexOf('JS'));

console.log(myString.search('JS'));

3.cut string ( cắt chuỗi )

console.log(myString.slice(4, 6));

console.log(myString.slice(4));

console.log(myString.slice(0));

console.log(myString.slice(*-*3));

**console.log(myString.slice(4, 6));**

slice(4, 6) trích xuất một phần của chuỗi myString từ chỉ mục 4 đến 5 (không bao gồm chỉ mục 6).

Kết quả là một chuỗi chứa các ký tự từ chỉ mục 4 đến 5 của myString.

**console.log(myString.slice(4));**

slice(4) trích xuất một phần của chuỗi myString từ chỉ mục 4 đến hết chuỗi.

Kết quả là một chuỗi chứa các ký tự từ chỉ mục 4 đến cuối chuỗi myString.

**console.log(myString.slice(0));**

slice(0) trích xuất toàn bộ chuỗi myString, bắt đầu từ chỉ mục 0.

Kết quả là một chuỗi chứa toàn bộ nội dung của myString.

**console.log(myString.slice(-3));**

slice(-3) trích xuất một phần của chuỗi myString từ vị trí cuối cùng trở lên, với độ dài là 3.

Kết quả là một chuỗi chứa 3 ký tự cuối cùng của myString.

4.replace ( thay thế )

console.log(myString.replace('JS', 'Javascript'));

console.log(myString.replace(/JS/*g*, 'Javascript'));

5.convert to upper case ( chuyển tất cả thành chữ hoa)

console.log(myString.toLocaleUpperCase());

6.convert to lower case ( chuyển tất cả chữ hoa thành thường )

console.log(myString.toLocaleLowerCase());

7.trim ( bỏ khoảng trắng ở đầu và cuối )

console.log(myString.trim());

8.split ( tạo chuỗi thành mảng qua những điểm chung )

var languages *=* 'Javascript, PHP, Ruby';

console.log(languages.split(', '));

console.log(languages.split(''));

9.get a character by index ( lấy kí tự )

var myString2 *=* 'Vu Duc Tien';

console.log(myString2.charAt(0));  (ko tồn tại trả về rỗng)

console.log(myString2[0]);  (ko tồn tại trả về undefined)

Example :

var coursesStr *=* 'HTML & CSS, JavaScript, ReactJS';

function strToArray(str) {

    var array*=* str.split(',');

*return* array;

}

*// Expected results*

console.log(strToArray(coursesStr))

*// Output: ['HTML & CSS', 'JavaScript', 'ReactJS']*

# Làm việc với Number

NaN : không phải là số, mà là 1 số không hợp lệ

Để ktra NaN : isNaN('');

Key : Javascript number methods

1.to string ( đổi Number thành String )

var age *=* 18;

var PI *=* 3.14;

var number *=* 3000.37492374;

var myString *=* age.toString();

console.log(*typeof* myString);

2.to fixed ( làm tròn số thập phân )

console.log(number.toFixed());

console.log(number.toFixed(2));

Example :

function isNumber(value) {

*if*(*!*isNaN(value) *&&* *typeof* value *===* 'number') {

*return* true;

    } *else* {

*return* false;

    }

}

*// Expected results:*

console.log(isNumber(999)); *// true*

console.log(isNumber('abc')); *// false*

console.log(isNumber('100')); *// false*

# Làm việc với mảng

Key : javascript array method

Mảng trong JavaScript – Array

var language *=* 'Javascript';

var language2 *=* 'PHP';

var language3 *=* 'Ruby';

var languages *=* [

    'Javascript',

    'PHP',

    'Ruby',

    null,

    undefined,

    function (){},

    {},

];

var languages *=* *new* Array (

    'Javascript',

    'PHP',

    'Ruby',

    null,

    undefined,

    function (){},

    {},

);

console.log(Array.isArray(languages));

*// kiểm tra xem có phải Array hay không*

console.log(languages.length);

console.log(languages[0]);

1, toString() ( chuyển mảng thành chuỗi )

console.log(languages.toString());

2, join() ( nối mảng )

console.log(languages.join(', '));

3, pop() ( xóa element cuối mảng và trả về element đã xóa )

console.log(languages.pop());

console.log(languages.pop());

console.log(languages.pop());

4, push() ( thêm phần tử vào cuối mảng )

console.log(languages.push('Dart', 'Java'));

console.log(languages);

5, shift() ( xóa phần tử ở đầu mảng )

console.log(languages.shift());

6, unshift() ( thêm phần tử mới vào đầu mảng )

console.log(languages.unshift('Dart', 'C++'));

console.log(languages)

7, splicing()

Xóa phần tử từ vị trí đặt con trỏ (vị trí, số p.tử cần xóa, chèn 1 el nào đó nếu số phần tử cần xóa = 0)

languages.spilice(1, 1, 'Dart')

=> thay phần tử ở vị trí thứ 1 *=* 'Dart'

languages.splice(1, 1);

console.log(languages);

8, concat() ( nối 2 mảng )

var languages *=* [

    'Javascript',

    'PHP',

    'Ruby'

];

var languages2 *=* [

    'Java',

    'Swift',

    'Go'

];

console.log(languages.concat(languages2));

console.log(languages2.concat(languages));

9, slicing() ( cắt lấy 1 vài element của mảng )

var languages *=* [

    'Javascript',

    'PHP',

    'Ruby'

];

console.log(languages.slice(1, 2))

console.log(languages.slice(0))

console.log(languages.slice(*-*2, *-*1))

Example :

function joinWithCharacter(array, charactor) {

*return* array.join(charactor);

}

*// Ví dụ khi sử dụng*

var cars *=* ['Honda', 'Mazda', 'Mercedes'];

var result *=* joinWithCharacter(cars, ' - ');

console.log(result); *// Expected: "Honda - Mazda - Mercedes"*

# Object

Khi key là function thì gọi là phương thức

var emailKey *=* 'email';

var myInfo *=* {

    name: 'Vu Duc Tien',

*// 'full-name': 'Vu Tien',*

    age: 18,

    address: 'Ninh Binh, VN',

    [emailKey]: 'thanhxuananhvaem.2k3@hotmail.com',

    getName: function() {

*return* this.name;

*// this trong đây là myInfo*

    }

};

*// myInfo.gmail = 'vuductientienti@gmail.com';*

*// myInfo['my-email'] = 'vu\_duc\_tien@outlook.com.vn'*

console.log(myInfo);

console.log(myInfo.getName());

console.log(myInfo.name);

console.log(myInfo['full-name']);

*delete* myInfo.gmail;

*delete* myInfo.name;

console.log(myInfo);

# Object constructor

Prototype là gì? Sử dụng khi nào?

Nếu coi constructor là 1 bản thiết kế, thì prototype là các dụng cụ

function User(firstName, lastName, avatar) {

    this.firstName *=* firstName;

    this.lastName *=* lastName;

    this.avatar *=* avatar;

    this.getName *=* function() {

*return* `${this.firstName} ${this.lastName}`

    }

}

User.prototype.className *=* 'F8';  *// thêm thuộc tính*

User.prototype.getClassName *=* function() {  *// thêm phương thức*

*return* this.className;

}

*// prototype có thể thêm vào đối tượng được tạo ra*

var User *=* function (firstName, lastName, avatar) {

    this.firstName *=* firstName;

    this.lastName *=* lastName;

    this.avatar *=* avatar;

    this.getName *=* function() {

*return* `${this.firstName} ${this.lastName}`

    }

}

Object :

var author *=* *new* User('Sơn', 'Đặng', 'Avatar');

var user *=* *new* User('Tiến', 'Vũ', 'Avatar');

author.title *=* 'Chia sẻ dạo tại F8';

user.comment *=* 'Tien ne hihi'

console.log(user.className);

console.log(author.className);

console.log(user.getClassName());

console.log(author.constructor);

# Đối tượng Date

Key: javascript date object mozilla

var date *=* Date(); *// là 1 chuỗi*

var date *=* *new* Date(); *// là 1 đối tượng*

var year *=* date.getFullYear();

var month *=* date.getMonth() *+* 1;

var day *=* date.getDate();

console.log(`${year}/${month}/${day}`)

console.log(date);

# Math Object

*-* Math.PI

*-* Math.round()

*-* Math.abs()

*-* Math.ceil()

*-* Math.floor()

*-* Math.random()

*-* Math.min()

*-* Math.max()

console.log(Math.PI)

console.log(Math.round(2.49)) *// làm tròn*

console.log(Math.abs(*-*5))  *// lấy trị tuyệt đối*

console.log(Math.ceil(2.49))   *// làm tròn trên*

console.log(Math.floor(2.9))   *// làm tròn dưới*

console.log(Math.random())  *// trả về số thập phân nhỏ hơn 1*

console.log(Math.floor(Math.random() *\** 10))   *// lấy 1 số ngẫu nhiên từ 0-9*

console.log(Math.min(*-*100, 1, 7, 3, 9, *-*20, *-*111))

console.log(Math.max(*-*100, 1, 7, 3, 9, *-*20, *-*111))

# Câu lệnh rẽ nhánh If – Else

## Vòng lặp

*for* (var i*=*0; i *<=* 1000; i*++*){

    console.log(i,': Hello World');

}

var getRandNumbers *=* function (min, max, length) {

    var arr *=* [length];

*for*(var i*=*0; i *<* length; i*++*) {

        arr.push(Math.random() *\** (max *-* min) *+* min);

*// arr[i] = random;*

        console.log(arr)

    }

*return* arr;

}

getRandNumbers(1, 15, 9);

## Vòng lặp for – in

“để lấy ra các ‘key’ trong object”

var myInfo *=* {

    name: 'Vu Tien',

    age: 20,

    address: 'Ninh Binh, VN'

}

*for* (var key *in* myInfo){

*// console.log(key)*

    console.log(myInfo[key])

}

var languages *=* [

    'javascript',

    'php',

    'ruby'

]

*for*(var key *in* languages) {

*// console.log(key)*

    console.log(languages[key])

}

var subject *=* 'Javascript';

*for* (var key *in* subject) {

*// console.log(key);*

    console.log(subject[key]);

}

Example :

function run(object) {

    var arr *=* [];

*for*(var key *in* object){

        arr.push(`Thuộc tính ${key} có giá trị ${object[key]}`)

    }

*return* arr;

}

*// Expected results:*

console.log(run({ name: 'Nguyen Van A', age: 16 }));

*// Output:*

*// [*

*//     "Thuộc tính name có giá trị Nguyen Van A",*

*//     "Thuộc tính age có giá trị 16"*

*// ]*

## Vòng lặp for – of

Sử dụng trong trường hợp muốn lấy ra phần tử của 1 mảng hoặc lấy ra phần tử của 1 chuỗi

Không dùng trực tiếp được với object

var languages *=* [

    'javascript',

    'php',

    'ruby'

]

var language *=* 'javascript';

*for*(var value *of* language) {

    console.log(value);

}

var myInfo *=* {

    name: 'Vũ Tiến',

    age: 18

}

*// console.log(Object.keys(myInfo));*

*// console.log(Object.values(myInfo));*

*for*(var value *of* Object.values(myInfo)) {

*// console.log(myInfo[value]);*

    console.log(value);

}

## Vòng lặp While

var myArr *=* [

    'Javascipt',

    'Php',

    'Ruby'

]

var i*=*0;

var myArrLength *=* myArr.length;

*while* (i *<=* myArrLength){

    console.log(myArr[i]);

    i*++*;

}

## Vòng lặp do – while

Chạy luôn không kiểm tra điều kiện, bắt đầu kiểm tra điều kiện từ lần lặp thứ 2

var i *=* 0;

var isSuccess *=* false;

*do* {

    i*++*;

    console.log('Nạp thẻ lần', i);

*// Thành công*

*if* (false) {

        isSuccess *=* true;

    }

} *while* (*!*isSuccess *&&* i *<* 3)

## Break và Continue trong vòng lặp

*for*(var i*=*0; i *<* 10; i*++*){

*if* (i*%*2 *!==* 0) {

*continue*;  *// là số lẻ thì bỏ qua, chạy tiếp*

    }

    console.log(i);

*// if (i >= 5) {*

*//     break;   // i >= 5 thì dừng và thoát khỏi vòng lặp*

*// }*

}

## Vòng lặp lồng nhau – Nested loop

var myArray *=* [

    [1, 2],

    [3, 4],

    [5, 6]

];

*for* (var i *=* 0; i *<* myArray.length; i*++*) {

*for* (var j *=* 0; j *<* myArray[i].length; j*++*){

        console.log(myArray[i][j]);

    }

}

var array *=* ['a', 'b', 'c', 'a', 'b', 'c'];

console.log([*...*(*new* Set(array))]);

console.log(array)

# Đệ quy

1, Xác định điểm dừng

2, Logic handle 🡺 Tạo ra điểm dừng

function deQuy(num) {

*if*(num *<* 0){

*// Dừng*

    }

    deQuy();

}

deQuy();

function countDown(num) {

*if*(num *>* 0){

        console.log(num);

*return* countDown(num *-* 1);

    }

*return* num;

}

countDown(10);

function loop(start, end, cb) {

*if* (start *<* end) {

        cb(start);

*return* loop(start *+* 1, end, cb);

    }

}

var array *=* ['Javascipt', 'PHP', 'Ruby'];

loop(0, array.length, function(index) {

    console.log(array[index]);

});

function giaiThua(num) {

*if* (num *>=* 1){

*return* num*\**giaiThua(num*-*1);

    }

*return* 1;

}

*// khi đệ quy chạy tới num = 0 thì vi phạm sẽ không chạy nữa và trả về 1 chính là vị trí cuối cùng của giai thừa*

console.log(giaiThua(3))

function giaiThua(num) {

*if* (num *<=* 0){

*return* 1;

}

*// khi đệ quy chạy tới num = 0 thì vi phạm sẽ không chạy nữa và trả về 1 chính là vị trí cuối cùng của giai thừa*

*return* num*\**giaiThua(num*-*1);

}

console.log(giaiThua(6))

# Callback

Là hàm đươc truyền qua đối số khi gọi hàm khác

1. Là hàm

2. Truyền qua đối số

3. Được gọi lại (trong hàm nhận đối số)

var courses *=* [

    {

        id: 1,

        name: 'JavaScript',

        coin: 300

    },

    {

        id: 2,

        name: 'PHP',

        coin: 0

    },

    {

        id: 3,

        name: 'Java',

        coin: 0

    },

    {

        id: 4,

        name: 'Ruby',

        coin: 500

    },

{

        id: 5,

        name: 'Ruby',

        coin: 500

    },

    {

        id: 6,

        name: 'ReactJS',

        coin: 100

    },

    {

        id: 7,

        name: 'Ruby',

        coin: 150

    }

];

* **array** là mảng cần tìm kiếm.
* **callback** là một hàm được thực thi cho từng phần tử trong mảng.
* **element** là giá trị hiện tại của phần tử trong mảng.
* **index** là chỉ mục của phần tử hiện tại trong mảng.
* **array** là mảng đang được tìm kiếm.

**forEach() :** lặp qua từng phần tử tương tự như vòng for

array.forEach(callback(currentValue, index, array) {

*// Thực hiện các tác vụ với mỗi phần tử*

});

**every() :** lặp qua từng phần tử trong mảng và kiểm tra điều kiện được định nghĩa trong hàm callback. Nếu tất cả các phần tử đều thỏa mãn điều kiện, phương thức trả về true. Nếu có ít nhất một phần tử không thỏa mãn điều kiện, phương thức trả về false.

array.every(callback(element, index, array) {

*// Kiểm tra điều kiện*

*// Trả về true hoặc false*

});

var isFree *=* courses.every(function(course, index) {

*return* course.coin *===* 0;

});

console.log(isFree)

// Output => false

**some() :** dùng để kiểm tra xem có ít nhất một phần tử trong một mảng thỏa mãn một điều kiện nào đó hay không. Nó trả về một giá trị boolean (true hoặc false) dựa trên kết quả kiểm tra, chỉ cần 1 phần tử thỏa mãn điều kiện sẽ trả về true.

array.some(callback(element, index, array) {

*// Kiểm tra điều kiện*

*// Trả về true hoặc false*

});

var isFree *=* courses.some(function(course, index) {

*return* course.coin *===* 0;

});

console.log(isFree)

*// Output => true*

**find() :** dùng để tìm kiếm và trả về phần tử đầu tiên trong một mảng thỏa mãn một điều kiện nào đó. Nếu không tìm thấy phần tử thỏa mãn, phương thức trả về undefined.

array.find(callback(element, index, array) {

*// Kiểm tra điều kiện*

*// Trả về true hoặc false*

});

var course *=* courses.find(function(course, index) {

*return* course.name *===* 'Ruby';

});

console.log(course)

**filter() :** dùng để lọc các phần tử trong một mảng dựa trên một điều kiện nào đó và trả về một mảng mới chứa các phần tử thỏa mãn điều kiện đó.

array.filter(callback(element, index, array) {

*// Kiểm tra điều kiện*

*// Trả về*

});

var listCourses *=* courses.filter(function(course, index) {

*return* course.name *===* 'Ruby';

});

**map() :** chỉnh sửa, thay đổi 1 element trong array.

function coursesHandler(course, index, originArray) {

    console.log(course);

*return* {

        id: course.id,

        name: `Khóa học: ${course.name}`,

        coin: course.coin,

        coinText: `Giá: ${course.coin}`,

        index: index,

        originArray: originArray,  *// originArray là array gốc*

    };

*// return course.name;*

*return* `<h2>${course.name}</h2>`;

}

var newCourses *=* courses.map(coursesHandler);

console.log(newCourses)

console.log(newCourses.join(''))

**reduce() :** sử dụng để thực hiện một phép tích lũy trên các phần tử của một mảng và trả về một giá trị duy nhất.

array.reduce(callback(accumulator, currentValue, index, array), initialValue)

* **array** là mảng ban đầu.
* **callback** là một hàm được thực thi cho từng phần tử trong mảng.
* **accumulator** là giá trị tích lũy, được cập nhật sau mỗi lần thực hiện callback.
* **currentValue** là giá trị hiện tại của phần tử trong mảng.
* **index** là chỉ mục của phần tử hiện tại trong mảng.
* **initialValue** (tuỳ chọn) là giá trị khởi tạo của accumulator. Nếu không được cung cấp, phần tử đầu tiên của mảng sẽ được sử dụng làm giá trị khởi tạo.

var totalCoin *=* courses.reduce(function(accumulator, currentValue){

*return* accumulator *+* currentValue.coin;

}, 0)

console.log(totalCoin)

var i*=*0;

function coinHandler(accumulator, currentValue, currentIndex, originArray){

*// (giá trị khởi tạo, giá trị hiện tại, chỉ mục, chính là tên object(courses))*

    i*++*;

    var total *=* accumulator *+* currentValue.coin;

*// console.log(i)*

    console.table({

        'Lượt chạy' : i,

        'Biến lưu trữ' : accumulator,

        'Giá khóa học' : currentValue.coin,

        'Tích trữ được' : total

    })

*// console.log(currentValue)*

*return* total;

*// return accumulator + currentValue.coin;*

}

var totalCoin *=* courses.reduce(coinHandler, 0)  *// 0 là giá trị khởi tạo, tùy thuộc vào nhu cầu*

console.log(totalCoin)

# Biến lưu trữ

var totalCoin *=* 0;

*// Lặp qua các phần tử*

*for*(var course *of* courses) {

*// Thực hiện việc lưu trữ*

    totalCoin *+=* course.coin;

}

console.log(totalCoin)

var totalCoin *=* courses.reduce(function(total, course){

*return* total *+* course.coin;

}, 0)

console.log(totalCoin)

var sports *=* [

    {

        name: 'Bơi lội',

        gold: 11

    },

    {

        name: 'Boxing',

        gold: 3

    },

    {

        name: 'Đạp xe',

        gold: 4

    },

    {

        name: 'Đấu kiếm',

        gold: 5

    },

]

function getTotalGold(array) {

*return* array.reduce(function(sum, sport){

*return* sum *+* sport.gold

    }, 0)

}

console.log(getTotalGold(sports))

const sports *=* [

    {

        name: 'Bóng rổ',

        like: 6

    },

    {

        name: 'Bơi lội',

        like: 5

    },

    {

        name: 'Bóng đá',

        like: 10

    },

]

*// var favour = sports.filter(function(yt){*

*//     return yt.like > 5;*

*// })*

*// console.log(favour)*

function getMostFavoriteSport(array) {

*return* array.filter(function(favor){

*if*(favor.like *>* 5) {

*return* favor.name;

        }

    })

}

*// Kỳ vọng*

console.log(getMostFavoriteSport(sports))

*// Output: [{ name: 'Bóng rổ, like: 6 }, { name: 'Bóng đá, like: 10 }]*

# InitialFunction (tạo hàm)

**Array.prototype** là một đối tượng trong JavaScript, nó đại diện cho nguyên mẫu (prototype) của tất cả các mảng trong JavaScript.

Khi khởi tạo một mảng mới, JavaScript sẽ tạo một đối tượng mới từ **Array.prototype** và gắn kết các phương thức và thuộc tính của **Array.prototype** cho mảng đó. Điều này cho phép các mảng chia sẻ cùng một tập hợp các phương thức và thuộc tính tiện ích, mà ta có thể truy cập thông qua **Array.prototype.**

Array Methods

forEach() : Duyệt qua từng phần tử của mảng

Array.prototype.myForEach *=* function(cb){

*for*(var index *in* this) {

*// Kiểm tra xem có phải key nằm trong Object không, nếu không phải thì không lặp qua*

*if*(this.hasOwnProperty(index)){

            cb(this[index], index, this)

        }

    }

}

courses.myForEach(function(course, index, array){

    console.log(index, course);

})

every() : Duyệt qua...=> tất cả phần tử phải thoả mãn điều kiện

=> true/false

*// Every cách 1*

Array.prototype.myEvery *=* function(cb) {

    var outputLength *=* 0;

*for*(var index *in* this){

*if*(this.hasOwnProperty(index)){

*if*(cb(this[index], index, this)){

                outputLength *+=* 1;

            }

        }

    }

*if* (outputLength *<* this.length) {

*return* false;

    } *else* {

*return* true;

    }

}

*// Every cách 2*

Array.prototype.myEvery *=* function(cb) {

    output *=* true;

*for*(var index *in* this) {

*if*(this.hasOwnProperty(index)){

            var result *=* cb(this[index], index, this)

*if*(*!*result) {

                output *=* false;

*break*;

            }

        }

    }

*return* output;

}

var checkCoin *=* courses.myEvery(function(course, index){

*return* course.coin *>=* 0;

})

console.log(checkCoin)

some() : Duyệt qua...=> 1(vài) phần tử thoả mãn điều kiện => true/false

Array.prototype.mySome *=* function(cb) {

*for*(var index *in* this){

*if*(this.hasOwnProperty(index)){

            var result *=* cb(this[index], index, this)

*if*(result){

*return* true;

            }

        }

    }

*return* false;

}

var isEmpty *=* courses.mySome(function(course){

*return* course.name *===* 'Ruby';

})

console.log(isEmpty)

find() : Duyệt qua...=> Tìm các phần tử thoả điều kiện   
=> Chỉ 1 phần tử đầu tiên thoả mãn điều kiện

Array.prototype.myFind *=* function(callback) {

    var arrayLength *=* this.length;

*for*(var i*=*0; i *<* arrayLength; i*++*) {

        var result *=* callback(this[i], i)

*// Nếu đúng thì trả về phần tử đó*

*if*(result){

*return* this[i];

        }

    }

}

var course *=* courses. myFind(function(course, index) {

*return* course.name *===* 'Ruby';

});

console.log(course)

filter() : Duyệt qua...=> Tìm các phần tử thoả điều kiện   
=> Tất cả phần tử thoả mãn điều kiện

Array.prototype.myFilter *=* function(cb) {

    var output *=* [];

*for*(var index *in* this) {

*if*(this.hasOwnProperty(index)){

            var result *=* cb(this[index], index, this)

*if*(result) {

                output.push(this[index])

            }

        }

    }

*return* output;

}

var price *=* courses.myFilter(function(course, index, array){

*// console.log(course, index, array);*

*return* course.coin *>* 400;

})

console.log(price)

map() : Duyệt qua...=> Thay đổi các phần tử/giá trị trong mảng => Mảng mới

Array.prototype.myMap *=* function(cb) {

    var output *=* [], arrayLength *=* this.length;

*for* (let  i *=* 0; i *<* arrayLength; i*++*) {

        var result *=* cb(this[i], i);

        output.push(result);

    }

*return* output;

}

function courseHandler(course) {

    var money *=* course.coin;

*if*(course.coin *===* 0) {

        money *=* 'Miễn phí';

    } *else* {

        money *=* money *+* `$`;

    }

*return* {

        id: course.id,

        name: `Khóa học: ${course.name}`,

        coin: course.coin,

        textCoin: `Thành tiền: ${money}`

    }

}

var newCourses *=* courses. myMap(courseHandler);

console.log(newCourses);

reduce() : Duyệt qua...=> Thay đổi các phần tử/giá trị trong mảng => Mảng mới

Array.prototype.myReduce *=* function (callback, result) {

    let i *=* 0;

*// Trường hợp không truyền initialValue*

*if* (arguments.length *<* 2) {

      i *=* 1; *// Gán currentValue là phần tử thứ 2 của mảng*

      result *=* this[0]; *// Biến tích trữ = phần tử đầu tiên của mảng*

    }

*// Khi không truyền initialValue => Sẽ lặp từ phần tử thứ 2*

*// Trường hợp khi truyền initialValue*

*for* (; i *<* this.length; i*++*) {

      result *=* callback(result, this[i], i, this)

*// result = total: biến tích trữ,*

*this[i]: currentValue, index, Array gốc*

    }

*return* result;

}

*// myReduce function*

function getTotalCoin(courses) {

var totalCoin *=* courses. myReduce(function (total, price) {

*return* total *+* price.coin;

}, 0);

*return* totalCoin;

}

console.log('totalCoin:', getTotalCoin(courses));

**arguments** : thuộc tính có sẵn, trả về số lượng các đối số truyền vào hàm.

Lưu ý rằng **arguments** là một đối tượng có cấu trúc giống mảng nhưng không phải là mảng thực sự. Để thao tác với **arguments** như một mảng, ta có thể chuyển nó thành một mảng thực sự bằng cách sử dụng **Array.from(arguments)** hoặc **Array.prototype.slice.call(arguments).**

# HTML DOM

Là viết tắt của Document Object Model

Gồm 3 thành phần:

1. Element : là thẻ tag trong file HTML
2. Attribute : là các thuộc tính trong thẻ tag
3. Text : là phần tử của DOM

HTML DOM : là quy chuẩn mô tả hình của các thành phần trong tài liệu html được đưa ra bởi W3C.

DOM API :

* Là bộ API nằm trong Web API có mặt trên những môi trường hỗ trợ trình duyệt web – như trên trình duyệt
* DOM API cung cấp bởi các đối tượng và phương thức hỗ trợ truy xuất, chỉnh sửa các đối tượng / thành phần trong DOM

var listItemNodes *=* document.querySelectorAll('.box-1 li')

*// select tất cả các thẻ li thuộc box-1*

var boxNode *=* document.querySelector('.box-1');

*// Công việc 1: sử dụng tới boxNode (trường hợp chỉ cần dùng tới .box-1)*

*// ....*

console.log({

    element: boxNode

})

*// Công việc 2: sử dụng tới các li là con của `.box-1`*

console.log(boxNode.querySelectorAll('li'));

console.log(boxNode.getElementsByTagName('li'));

console.log(boxNode.querySelector('p'));

## 1 số phương thức Get

1. getElementById
2. getElementsByClassName
3. getElementsByTagName
4. querySelector
5. querySelectorAll
6. HTML collection
7. document.write

Chỉ 1 và 4 mới select được trực tiếp Element

var headings *=* document.querySelectorAll('.heading')

*for*(var i*=*0; i *<* headings.length; i*++*) {

    console.log(headings[i])

}

console.log(document.forms['form-1'])

## DOM Attribute

*// seter*

var headingElement *=* document.querySelector('h1')

headingElement.title *=* 'heading'

headingElement.id *=* 'id\_heading'

headingElement.className *=* 'class\_heading'

var aElement *=* document.querySelector('a')

aElement.href *=* 'tienne.info'

console.log(aElement)

headingElement.setAttribute('class', 'heading')

## Get value of Attribute

(lấy giá trị của Attribute)

console.log(headingElement.getAttribute('class'))

headingElement.title *=* 'title-test'

console.log(headingElement.getAttribute('title'))

## innerText, textContent (lấy text)

* textContent : lấy ra nội dung thực trong DOM (nguyên bản textNode, vd: khoảng trắng, xuống dòng…)
* innerText trả lại nội dung như những gì nhìn thấy trên trình duyệt
* Cả 2 đều không lấy ra thẻ tag, nhưng text content lấy ra tất cả nội dung trong thẻ tag

var headingElement *=* document.querySelector('.heading')

headingElement.innerText *=* 'New heading'  *// Thay đổi text*

console.log(headingElement.innerText)

console.log(headingElement.textContent)

## 

## innerHTML

innerHTML giúp chúng ta thêm được elementNode, textNode, attributeNode…

Ghi đè vào các thẻ html có trong class

var boxElement *=* document.querySelector('.box')

boxElement.innerHTML *=* '<h1 title="heading">New heading</h1>'

console.log(boxElement.innerHTML)

console.log(document.querySelector('h1').innerText)

## outerHTML

Ghi đè lên thẻ chính chứa class

console.log(boxElement.outerHTML)

*// (lấy từ chính vi trí querySelector)*

console.log([boxElement])

## DOM CSS

var boxElement *=* document.querySelector('.box')

boxElement.style.width *=* '100px'

boxElement.style.height *=* '200px'

boxElement.style.backgroundColor *=* 'green'

Object.assign(boxElement.style, {

    width: '200px',

    height: '200px',

    backgroundColor: 'green'

})

console.log(boxElement.style.backgroundColor)

## classList property

+) add : thêm class

+) contains : kiểm tra xem class có tồn tại hay không

+) remove : xóa class

+) toggle : khi thực hiện code thì kiểm tra class có tồn tại hay không, nếu có thì xóa và nếu không có thì thêm vào

var boxElement *=* document.querySelector('.box')

console.log(boxElement.classList)

console.log(boxElement.classList[0])

console.log(boxElement.classList.length)

console.log(boxElement.classList.value)

boxElement.classList.add('red')

console.log(boxElement.classList.contains('red'))

setInterval(() => {

*for*(var i*=*0; i *<* 3; i*++*){

        boxElement.classList.toggle('red');

        boxElement.classList.toggle('green');

    }

}, 1000)

## DOM EVENT

1. Attribute event
2. Assign event usign the element node

var h1Element *=* document.querySelector('h1')

console.log(h1Element)

h1Element.onclick *=* function() {

    console.log(Math.random() *\** 100)

}

1. Input / select
2. Key up / down

onchange : khi value bị thay đổi thì lấy ra giá trị của nó, onchange sẽ hoạt động khi bỏ focus ra ngoài

oninput : khi value bị thay đổi tới đâu thì lấy giá trị ra tới đó

var inputValue;

var inputElement *=* document.querySelector('input[type="text"]')

*// onchange : khi value bị thay đổi lấy ra giá trị của nó, onchange sẽ hoạt động khi bỏ focus ra ngoài*

*// oninput : khi value bị thay đổi tới đâu thì lấy giá trị ra tới đó*

inputElement.oninput *=* function(e) {

    inputValue *=* e.target.value

}

var inputElement *=* document.querySelector('input[type="checkbox"]')

inputElement.onchange *=* function(e) {

    console.log(e.target.checked)

}

var inputElement *=* document.querySelector('select')

inputElement.onchange *=* function(e) {

    console.log(e.target.value)

}

keydown : khi nhấc lên mới thực hiện nên bị chậm 1 bước

keyup : khi nhấc lên sẽ thực hiện luôn

var inputElement *=* document.querySelector('input[type="text"]')

inputElement.onkeydown *=* function(e) {  *// onkeyup, onkeypress*

*switch*(e.which) {

*case* 27:

            console.log('EXIT')

*break*;

*case* 13:

            console.log('SEND CHAT');

*break*;

    }

}

1. preventDefault ( ngăn chặn hành vi mặc định )
2. stopPropagation ( loại bỏ sự kiện nổi bọt )

document.anchors ( trả về thẻ có name )

document.links ( trả về tất cả các thẻ a )

var aElements *=* document.links

*for* (var i*=*0; i *<* aElements.length; i*++*){

    aElements[i].onclick *=* function(e) {

*// nếu chuỗi bắt đầu bằng 'https://f8.edu.vn' thì cho chuyển trang*

*// nếu chuỗi không bắt đầu bằng 'https://f8.edu.vn' thì bỏ đi hành vi*

*mặc định của thẻ a*

*if*(*!*e.target.href.startsWith('https://f8.edu.vn')) {

            e.preventDefault();

        }

    }

}

var ulElement *=* document.querySelector('ul');

*// ngăn chặn hành vi khi chúng ta onmousedown vào thẻ ul, nếu không có thì khi click vào ul sẽ bị mất*

ulElement.onmousedown *=* function(e) {

    e.preventDefault();

}

ulElement.onclick *=* (e) => {

    console.log(e.target)

}

document.querySelector('div').onclick *=* (e) => {

   console.log('DIV')

}

document.querySelector('button').onclick *=* (e) => {

    e.stopPropagation(); *// ngăn chặn nổi bọt thẻ cha thẻ ông*

    console.log('Click me!')

}

*// 1. Event listenner (lắng nghe các sựu kiện xảy ra trong DOM)*

*// Xử lý nhiều việc khi 1 event xảy ra*

*// Lắng nghe / hủy bỏ lắng nghe*

var btn *=* document.getElementById('btn');

btn.onclick *=* (e) => {

*// việc 1*

    console.log('Việc 1')

*// việc 2*

    console.log('Việc 2')

*// việc 3*

    alert('Việc 3')

}

setTimeout(function() {

    btn.onclick *=* () => {}

}, 3000)

btn.addEventListener('click', () =>{

    console.log('Event 1')

})

btn.addEventListener('click', () =>{

    console.log('Event 2')

})

btn.addEventListener('click', () =>{

    console.log('Event 3')

})

function viec1() {  *// mỗi function là 1 listener*

    console.log('Việc 1')

}

function viec2() {  *// mỗi function là 1 listener*

    console.log('Việc 2')

}

btn.addEventListener('click', viec1)

btn.addEventListener('click', viec2)

setTimeout(function () {

    btn.removeEventListener('click', viec1)

*// hủy bỏ sự kiện của 1 listener nào đó*

}, 3000)

# Json, Fetch, Postman

## Json

* Là 1 định dạng dữ liệu chuỗi, KHÔNG phải là kiểu dữ liệu
* Viết tắt của : Javascript Object Nation
* JSON : number, boolearn, null, array, object
* Mã hóa (encode) / Giải mã (decode)
* Stringify (convert sang bộ mã khác) : Javascript type 🡺 JSON
* Parse (trả về kiểu dữ liệu ban đầu) : JSON 🡺 Javascript type

var json *=* '["Javascript", "PHP"]'

var json *=* '{"name":"Vu Duc Tien", "age":18}'

var a *=* '1';

console.log(JSON.parse(json))

console.log(JSON.stringify(1))

## Promise

* Sync : đồng bộ
* Async : bất đồng bộ
* Pain : nỗi đau

Sync (đồng bộ) : viết trước chạy trước (chạy theo luồng)

Async (bất đồng bộ) : setTimeout(), setInterval(), fetch, XMLHttpRequest, file reading (đọc file), request animation frame.

Pain (nỗi đau) gồm :

Callback hell : khi sử dụng callback; callback lồng callback; khi callback ngoài chạy xong thì callback trong mới được thực thi.

setTimeout(function(){

        console.log(1) *// viec 1*

        setTimeout(function(){

            console.log(2) *// viec 2*

            setTimeout(function(){

                console.log(3) *// viec 3*

                setTimeout(function(){

                    console.log(4) *// viec 4*

                    setTimeout(function(){

                        console.log(5) *// viec 5*

                    }, 1000)

                }, 1000)

            }, 1000)

        }, 1000)

    }, 1000)

Pyramid of doom : khi viết code tổng quan

1. Em có nắm rõ khái niệm Promise không ?

Có, Promise là 1 khái niệm sinh ra để giúp chúng ta xử lý các thao tác bất đồng bộ, trước khi có Promise chúng ta thường sử dụng callback, sẽ xảy ra vấn đề là callback hell nó sẽ bị sâu vào rất khó nhìn code bị rối khó hiểu cho nên Promise được sinh ra từ phiên bản Javasript mới hơn trong phiên bản ES6 và chúng ta sử dụng nó để khắc phục tình trạng callback hell để giúp chúng ta viết code không bị sâu vào dễ đọc dễ hiểu hơn.

Để tạo ra promise chúng ta sẽ sử dụng từ khóa 'new' với 'Promise' và trong constructor của nó chúng ta sẽ truyền vào 1 'Executor function()' khi Executor thực thi trong 'Executor function()' sẽ nhận được 2 tham số dạng hàm 'resolve()' và 'reject()'.

reslove() chúng ta sẽ gọi nó khi thao tác logic xử lý thành công.

reject() chúng ta sẽ gọi nó khi thao tác logic xử lý thất bại.

Và khi chúng ta sử dụng Promise(), đối tượng promise sẽ được tạo ra, chúng ta sẽ sử dụng qua những phương thức là 'then()' hoặc 'catch()'.

'.then()' và '.catch()' đều nhận 'callback function()' và sẽ được thực thi vào 'then()' nếu promise của chúng ta đựơc 'resolve()' hoặc thực thi vào 'catch()' nếu promise của chúng ta 'reject()'

Promise có 3 trạng thái:

- Pending: trạng thái chờ thành công hay thất bại

- Fulfilled: thành công

- Rejected: thất bại

# IIFE, SCOPE, CLOURSE

## IIFE

IIFE là viết tắt của "Immediately Invoked Function Expression" (Biểu thức Hàm Thực Thi Ngay Lập Tức). Đây là một cú pháp trong JavaScript cho phép bạn tạo và thực thi một hàm một cách tức thì ngay sau khi nó được khai báo.

(function() {

*// Code của hàm*

})();

(function(parameter1, parameter2) {

*// Code của hàm*

})(value1, value2);

(function() {

    var message *=* "Hello from IIFE";

    console.log(message); *// Output: "Hello from IIFE"*

})();

*// Biến "message" không tồn tại ở đây và sẽ gây lỗi*

console.log(message); *// Output: ReferenceError: message is not defined*

## Scope – phạm vi

- Các loại phạm vi

- Global - Toàn cầu

- Code block - Khối mã: let, const

- Local scope - Hàm: var, function

- Khi gọi mỗi hàm luôn có 1 phạm vi mới được tạo

- Các hàm có thể truy cập các biến được khai báo trong phạm vi của nó và bên ngoài nó

- Cách thức 1 biến được truy cập

- Khi nào 1 biến bị xóa khỏi bộ nhớ?

- Biến toàn cầu

- Biến trong code block & trong hàm

- Biến trong hàm được tham chiếu bởi 1 hàm

## Closure

Là 1 hàm có thể ghi nhớ nơi nó được tạo và truy cập được biến ở bên ngoài phạm vi vủa nó

# Ứng dụng

- Viết code ngắn gọn hơn

- Biểu diễn, ứng dụng tính Private trong OOP

# Tóm tắt

# Lưu ý

- Biến tham chiếu (refer) trong cloruse sẽ không được xóa khỏi bộ nhớ khi hàm cha chưa thực thi xong

- Các khái niệm Javascript nâng cao rất dễ gây nhầm

# Hosting, Strict mode, Data types

## Hoisting

- Đưa lên đầu

## Strict mode

Báo lỗi hoặc ngăn chặn khi sử dụng những đoạn mã không an toàn hay dễ gây nhầm lẫn

Cách sử dụng:

- Thêm ‘use strict’ vào đầu file.js

- Thêm ‘use strict’ vào ngay đầu thẻ mở <script>

- Thêm ‘use strict’ vào đầu phạm vi hàm

* Báo lỗi khi gán lại giá trị cho thuộc tính có writable: false
* Khai báo hàm trong code block thì hàm sẽ thuộc phạm vi code block
* Không đặt tên biến tên hàm bằng từ khóa ‘nhạy cảm’ của ngôn ngữ

Công dụng

1. Tránh quên từ khóa khai báo biến
2. Tránh dùng tên biễn lẫn với logic
3. Sử dụng bộ nhớ hiệu quả vì tránh sử dụng biến global

'use strict';

const student *=* {}

Object.defineProperty(student, 'fullName', {

    value: 'Vu Duc Tien',

    writable: true,

*// Mặc định writable se là false nhưng chúng ta có thể thay đổi thành true*

*// Khi false thì ko thể thay đổi giá trị fullName và sẽ báo lỗi*

})

student.fullName *=* 'Nguyen Van B'

console.log(student)

function sum(a, a) {

*return* a *+* a

}

console.log(sum(6, 9))  *// output: 18*

*// Khi khai báo biến trùng tên thì biến sau sẽ ghi đè*

*// Trong trường hợp này nếu sử dụng 'use strict' thì sẽ báo lỗi*

## Primitive Types & Reference Types

Với các Javascript Engine hiện đại ngày nay thì việc sử dụng bộ nhớ đã hiệu quả hơn. Khi một đối tượng không còn được tham chiếu tới bởi 1 biến thì Javascript Engine sẽ xóa bỏ vùng nhớ của object đó để tối ưu.

1. Value types (Primitive Types)

* String
* Number
* Boolean
* BigInt
* Symbol
* Undefined
* Null

1. Reference types (Non-primitive)

* Object
* Array
* Function

# This, Bind, Call, Apply

## Từ khóa “this”

Từ khóa this trong Javascript đề cập đến đối tượng mà nó thuộc về.

Đặc tính

1. Trong 1 phương thức this tham chiếu tới đối tượng truy cập phương thức (đối tượng trước dấu .)
2. Đứng ngoài phương thức, this tham chiếu tới đối tượng global

Lưu ý

* This trong hàm tạo đại diện cho đối tượng sẽ được tạo
* This trong 1 hàm là undefined khi ở strict mode
* Các phương thức bind(), call(), aplly() có thể tham chiếu this ở 1 đối tượng khác

Các bạn nghĩ đơn giản như này, phân biệt qua cách hàm/method được gọi. Được gọi qua object.method() thì this là object. Được gọi theo tenHam() thì this là window (strict mode là undefined). Mỗi hàm luôn có context riêng, hàm trong method cũng có context riêng của nó (trừ arrow function).

Trong bài này mình chỉ "đả thông" về tư duy hướng đối tượng thôi nhé. Với Javascript để thể hiện đối tượng chúng ta sẽ dùng Function constructor hoặc Class để tạo đối tượng mới "chuẩn" nhé. Nhưng sẽ học sau, cuối cùng mình cũng sẽ nói thôi hihi 😍

*// const iPhone7 = {*

*//     // Thuộc tính - Property*

*//     name: 'iPhone 7',*

*//     color: 'Pink',*

*//     weight: 300,*

*//     // Phương thức - Method*

*//     takePhoto(){*

*//         console.log(this)*

*//     },*

*//     objChild: {*

*//         methodChild() {*

*//             name: 'Child Object',*

*//             console.log(this)*

*//         }*

*//     }*

*// }*

*// console.log(iPhone7.takePhoto());*

*// // Khi dùng 'this' thì nó chính là iPhone 7 hay nói các khác khi iPhone7 . tới takePhoto() thì 'this' trong đấy chính là iphone7*

*// iPhone7.objChild.methodChild()*

*// // Khi này this chính là objChild vì nó nằm ngay trước Method 'this'*

*// Hàm tạo*

function Car(name, color, weigth) {

    this.name *=* name;

    this.color *=* color;

    this.weigth *=* weigth;

    this.run *=* function() {

        console.log('Running...', this)

    }

*// Khi 1 thuộc tính là 1 hàm và thuộc 1 đối tượng thì sẽ gọi là phương thức*

}

Car.prototype.run1 *=* function() {{

*// console.log(this)*

*// Context*

*// function test() {*

*//     console.log(this)*

*//     // this nằm trong hàm chính là window*

*// }*

*// test()*

    const test1 *=* () => {

        console.log(this)

    }

    test1()

}}

*// Lưu ý: arrown function không có this nên nó sẽ lấy this ở bên ngoài ngay gần nó tức là lấy context*

const mercerdes450 *=* *new* Car('Mercerdes s450', 'Black', 1200)

console.log(mercerdes450.run1())

*// const button = document.querySelector('button')*

*// // console.log(button)*

*// button.onclick = function() {*

*//     console.log(this)*

*//     console.dir(this.innerText)*

*// }*

## Bind()

Trong JavaScript, phương thức bind() là một phương thức của các hàm, được sử dụng để ràng buộc (bind) ngữ cảnh (context) của hàm đó với một đối tượng cụ thể. Khi ta sử dụng bind(), ta tạo ra một phiên bản mới của hàm có ngữ cảnh cố định, đảm bảo rằng khi hàm được gọi, this sẽ luôn trỏ tới đối tượng được chỉ định.

Khi truyền đối tượng vào trong bind() thì nó sẽ ràng buộc đối tượng đó.

this.firstName *=* 'Minh'

this.lastName *=* 'Thu'

const teacher *=* {

    firstName: 'Minh',

    lastName: 'Thảo',

    getFullName() {

*return* `${this.firstName} ${this.lastName}`

    }

}

*// Case 1*

console.log(teacher.getFullName())  *// Minh Thảo*

*// Khi chúng ta gọi 1 phương thức thông qua 1 đối tượng thì từ khóa 'this' sẽ trỏ về đúng đối tượng mà chúng ta gọi phương thức đó*

*// Case 2*

const getTeacherName *=* teacher.getFullName

*// chưa có toán tử gọi hàm nên lúc này nó chưa gọi hàm mà nó chỉ truy cập vào bên trong hàm*

*// và nó gán function 'getFullName' sang 1 hàm hoặc 1 biến khác*

*// trong Javascript thì function bản chất nó là 1 object (là dạng tham chiếu)*

*// khi gán 1 giá trị tham chiếu cho 1 biến khác thì nó chỉ sao chép vùng nhớ sáng biến kia*

*// console.log(getTeacherName === teacher.getFullName)*

console.log(getTeacherName())   *// Minh Thu*

*// method chính là function*

*// khi gọi hàm không thông qua 1 đối tượng (không có dấu . ở trước) thì 'this' trỏ ra đối tượng global và chọc ra phạm vi window nên ra 'Minh Thu'*

*// Case 3*

const getTeacherName1 *=* teacher.getFullName.bind(teacher)

console.log(getTeacherName1())  *// Minh Thảo*

Phương thức bind() nhận n đối số, nhưng duy nhất đối số đầu tiên chính là object ràng buộc, khi truyền thẳng đối số vào bind() thì sẽ ưu tiên hơn và tất cả các đối số truyền bên ngoài không có tác dụng .

const student *=* {

    firstName: 'Vũ',

    lastName: 'Tiến',

}

*// const getTeacherName2 = teacher.getFullName.bind(student)*

*// console.log(getTeacherName2())  // Vũ Tiến*

const getTeacherName3 *=* teacher.getFullName.bind(student, 'Đức', 'Tiến')

console.log(getTeacherName3('Tiến', 'DZ'))  *// Đức Tiến  , Vũ Tiến*

Lưu ý: nếu đối số không cần thay đổi thì truyền thẳng vào bind() nếu không thì không nên truyền thẳng.

* Phương thức bind() sẽ trả về 1 hàm mới
* Có thể nhận các đối số như hàm ban đầu

const student *=* {

    firstName: 'Vũ',

    lastName: 'Tiến',

}

*// const getTeacherName2 = teacher.getFullName.bind(student)*

*// console.log(getTeacherName2())  // Vũ Tiến*

const getTeacherName3 *=* teacher.getFullName.bind(student)

console.log(getTeacherName3('Tiến', 'DZ'))  *// Tiến DZ  ,   Vũ Tiến*

*// ============== Thực hành với DOM ==========*

const teacher *=* {

    firstName: 'Minh',

    lastName: 'Thảo',

    getFullName() {

        console.log(`${this.firstName} ${this.lastName}`)

    }

}

*// teacher.getFullName()*

const button *=* document.querySelector('button')

*// button.onclick = function () {*

*//     teacher.getFullName()*

*// }*

button.onclick *=* teacher.getFullName.bind(teacher)

*// Bản chất teacher.getFullName nó là 1 function nên gán thẳng nó = button.onclick nhưng console.log lại ra undefined*

*// vì khi chúng ta click thì 'this' ở trong onclick nó lại trỏ về 'button' nên không ra cái gi*

*// Để lấy được giá trị muốn lấy ta phải dùng hàm bind() và ràng buộc obj 'teacher'*

*// ============== Thực hành với DOM 2 ==========*

const $ *=* document.querySelector

const $$ *=* document.querySelectorAll

console.log(document.querySelector('#heading').innerText)

*// <h1 id="heading">Hello bind() method</h1>*

*// Trong method 'selector' đang sử dụng từ khóa 'this', nó sẽ trỏ về object gọi querySelector là 'document' truyền đối số nên sẽ chạy 1 cách đúng đắn*

console.log($('#heading'))

*// Khi chưa có bind() thì lỗi, chúng ta gán cho nó $ thì nó trỏ về object gọi querySelector là 'document' nhưng nó là window*

*// Nó bị mất Context, Context sẽ trỏ ra đối tượng global, Context 'this' bên trong sẽ trỏ về window*

## Call()

- Fn.call() giúp gọi hàm và bind this tới đối tượng khác, mặc định this là window object

- Fn.call() không trả ra hàm mới, nó gọi luôn hàm sau khi bind this (Fn.bind() thì chỉ bind this nhưng không gọi hàm)

- Fn.call() dùng để mượn hàm - function borrowing

- Fn.call() có thể dùng để kế thừa properties & method từ 1 Constructor khác

'use strict'

this.firstName *=* 'Vũ'

this.lastName *=* 'Tiến'

function showFullName() {

    console.log(`${this.firstName} ${this.lastName}`)

}

showFullName.call(this)

function Animal(name, weigth) {

    this.name *=* name

    this.weigth *=* weigth

}

function Chicken(name, weigth, legs) {

    Animal.call(this, name, weigth)

    this.legs *=* legs

}

const sonDang *=* *new* Chicken('Sơn Đặng', 66, 2)

console.log(sonDang)

call(this,…) this ở đây là Chicken, lúc này sẽ truyền thêm đối số legs vào Animal nên obj sonDang có đủ tên, cân nặng và chân

function logger() {

*// console.log(...arguments)*

    Array.prototype.forEach.call(arguments, item => {

        console.log(item)

    })

*// gọi method call() ở forEach, forEach là 1 method trong đó đang dùng 'this' để lấy mảng*

*// dùng call() bind 'this' là arguments thế nên từ khóa 'this' sẽ lấy arguments làm mảng và nó lặp qua*

    const cars *=* ['BMW', 'Porsche']

    cars.forEach(car => {

        console.log(car)

    })

*// trong forEach sẽ có this, và nó trỏ ra cars để loop các phần tử trong cars*

}

logger(1,2,3,4,5)

function logger() {

*// console.log(Array.prototype.slice.call(arguments))*

    const arr *=* Array.prototype.slice.call(arguments)

    console.log(arr)

    arr.forEach(item => console.log(item))

    const arr *=* Array.from(arguments)

    const arr *=* [*...*arguments]

}

logger(1,2,3,4,5)

## Apply()

Phương thức này cho phép gọi 1 hàm với this (bind) và truyền đối số cho hàm gốc dưới dạng mảng.

const teacher *=* {

    firstName: 'Minh',

    lastName: 'Thu',

}

function greet(greeting, message) {

*return* `${greeting} ${this.firstName} ${this.lastName}. ${message}`

}

let result *=* greet.apply(teacher, ['Em chào cô', 'Cô dạy môn gì thế ạ? (Đã xem cô livestream 1 tiếng)'])

console.log(result)

*// So sánh với call method*

result *=* greet.call(teacher, 'Em chào cô', 'Cô dạy môn gì thế ạ? (Đã xem cô livestream 1 tiếng)')

console.log(result)

Mượn hàm

const teacher *=* {

    firsName: 'Minh',

    lastName: 'Thảo',

    isOnline: false,

    goOnline() {

        this.isOnline *=* true;

        console.log(`${this.firsName} ${this.lastName} is Online`)

    },

    goOffline() {

        this.isOffline *=* false;

        console.log(`${this.firsName} ${this.lastName} is Offline`)

    }

}

const me *=* {

    firsName: 'Vũ',

    lastName: 'Tiến',

    isOnline: false,

}

console.log('teacher: ', teacher.isOnline)

teacher.goOnline()

console.log('teacher: ', teacher.isOnline)

console.log('--------------------')

console.log('me: ', me.isOnline)

teacher.goOnline.apply(me)

console.log('me: ', me.isOnline)

## So sánh bind(), call(), apply()

**\*) Giống nhau**

* Cú pháp truy cập

function fn() {

*// bind*

    fn.bind()

*// call*

    fn.call()

*// apply*

    fn.apply()

}

* Là các method được kế thừ từ Function.prototype

function fn() {

    fn.bind *===* Function.prototype.bind

    fn.call *===* Function.prototype.call

    fn.apply *===* Function.prototype.apply

}

**\*) Khác nhau**

1. bind method

* Trả ra hàm mới với `this` tham chiếu tới `thisArg`
* Không thực hiện gọi hàm
* Nếu bind() được kèm `arg1, arg2,…` thì tham số này sẽ được ưu tiên hơn

2. call method

* Thực hiện bind `this` với `thisArg` và thực hiện gọi hàm luôn
* Nhận các đối số từ hàm gốc ‘arg1, arg2,…’

3. apply method

* Thực hiện bind `this` với `thisArg` và thực hiện gọi hàm luôn
* Nhận các đối số cho hàm gốc bằng đối số thứ 2 dưới dạng mảng [arg1, arg2,….]