**Họ và tên: Nguyễn Quốc Tĩnh**

**Tài liệu tổng hợp kiến thức**

**I. Javascript nâng cao**

**1. Closure**

Closures (bao đóng) là một hàm bên trong mà có thể truy cập biến của hàm bên ngoài (chứa nó). Closure có 3 scope, đó là:

* Có thể truy cập đến biến của chính nó (biến được định nghĩa trong dấu ngoặc nhọn của nó);
* Có thể truy cập biến của hàm bên ngoài;
* Có thể truy cập biến toàn cục (global).

Các quy tắc của closures và Side Effects của nó:

* Closures có thể truy cập biến của hàm bên ngoài ngay cả hàm bên ngoài đã trả về
* Closures lưu tham chiếu đến biến của hàm bên ngoài, thay đổi giá trị biến toàn cục lẫn cục bộ
* Closures đôi khi trở nên không như ý: Bởi vì closures có thể truy cập đến các giá trị đã được cập nhật của các biến của hàm bên ngoài, chúng có thể gây ra bugs khi biến của hàm bên ngoài thay đổi với vòng lặp for

Độ ưu tiên các biến trong closure function: Như ta biết thì closure có thể sử dụng biến tại ba vị trí, đó là biến toàn cục, biến hàm cha và biến của chính nó.Giả sử tên các biến ở ba vị trí đó bị trùng nhau độ ưu tiên sẽ được sắp xếp như thế nào? Trường hợp này nó sẽ ưu tiên từ trong ra ngoài như sau:

* Bước 1: Xem biến có nằm trong closure function không? Nếu không thì nhảy qua bước 2, nếu có thì sử dụng.
* Bước 2: Xem biến có nằm trong hàm cha không? Nếu không thì qua bước 3, nếu có thì sử dụng.
* Bước 3: Xem có phải là biến cục bộ không? Nếu có thì sử dụng, nếu không thì nó sẽ khởi tạo biến mới mới.

**2. Context, this, bind**

* Context có thể hiểu đơn giản là ngữ cảnh. Ngữ cảnh cung cấp thông tin về môi trường thực thi đoạn mã. Javascript dùng biến this để mô tả ngữ cảnh. Javascript this rất khó khăn bởi biểu hiện của nó khác nhau tuỳ vào cách gọi hàm cũng như mode thực thi
* Trong javascript, chúng ta dùng từ khóa this để đại diện cho một đối tượng (Object). Đối tượng đó là chủ thế của ngữ cảnh, hoặc là chủ thế của code đang được chạy.
* Bind là một function nằm trong function prototype do đó chỉ có function mới có thể gọi được nó. Chúng ta gọi Bind method dùng để xác định tham số “this” cho một function. Bind() cho phép chúng ta dễ dàng thiết lập một đối tượng cụ thể sẽ bị ràng buộc này khi một chức năng hoặc phương pháp được gọi.

**3. Hoisting**

* Hoisting là cơ chế của JavaScript cho phép các khai báo biến hoặc hàm được dời lên trên đầu phạm vi của chúng trước khi thực thi đoạn code.

3.1. Hoisting đối với biến, ví dụ:

console.log(a);

var a = 'Hello Hoisting'

* Trình biên dịch của Javascript sẽ phân tách phần var a = 'Hello Hoisting' thành 2 phần là khai báo và gán giá trị:
  + Khai báo: var a
  + Gán giá trị a = 'Hello Hoisting'
* Theo Hoisting, Javascipt sẽ chuyển phần khai báo lên trên top. Vì vậy, sẽ chỉ có phần khai báo được chuyển lên trên top, còn phần gán giá trị vẫn giữ nguyên, vậy nên Output sẽ là undefined . Để tránh những trường hợp như vậy xuất hiện về sau thì bạn nên khai báo và khởi tạo cùng một lúc.

3.2. Hoisting đối với hàm: Trình biên dịch Javascript sẽ chuyển phần khai báo lên trên đầu phạm vi giống như cách làm với biến. Ví dụ:

do\_something();

function do\_something() {

    console.log(a);

    var a = 'fly';

}

* Cũng giống như với phần 1, phần bên trong của hàm do\_something cũng được trình biên dịch sử dụng hoisting.
* Nhưng phần khai báo chỉ được chuyển lên trên top của hàm do\_something chứ không phải của chương trình. Vậy nên, Output sẽ là undefined

3.3. Thứ tự ưu tiên trong cơ chế hoisting:

* + Gán biến ưu tiên hơn khai báo hàm
  + Biểu thức hàm ưu tiên hơn gán biến
  + Khai báo hàm ưu tiên hơn khai báo biến

3.4. Đối phó với vấn đề hoisting như thế nào?

Bởi vì cơ chế hoisting khiến chúng ta quản lý code JS khá là khó khăn.Và khó có thể hiểu được thực sự vấn đề đang gì xảy ra nếu bạn quản lý một dự án lớn, phức tạp. Và để đối phó với vấn đề này cũng khá đơn giản.

* Cách 1: Bạn phải hiểu rõ về nó
* Cách 2: Hạn chế hành vi hoisting. Để hạn chế hành vi hoisting trong JavaScript bạn chỉ cần bật chế độ use strict. Hoặc sử dụng từ khóa let hoặc const khi khai báo bất kỳ một biến nào.Mặc dù biến khai báo bằng từ khóa let hoặc const vẫn được đưa lên trên đầu. Nhưng nó không cho phép sử dụng cho đến khi được khai báo.

**4. Prototype**

Prototype là cơ chế mà các object trong javascript kế thừa các tính năng từ một object khác. Tất cả các object trong javascript đều có một prototype, và các object này kế thừa các thuộc tính (properties) cũng như phương thức (methods) từ prototype của mình.

* Tạo ra prototype: Do hàm khởi tạo đối tượng cũng được xem là 1 đối tượng prototype, do đó các đơn giản để tạo ra 1 đối tượng prototype là khai báo một hàm khởi tạo. Ví dụ một object infPerson chứa 1 hàm khởi tạo và 1 thuộc tính (thuộc tính “height”):

//Tạo ra 1 mẫu khởi tạo, cũng là tạo ra 1 prototype object

function infPerson(\_age, \_name, \_address) {

    this.age = \_age;

    this.name = \_name;

    this.address = \_address;

}

//Có thể thêm thuộc tính vào thuộc tính prototype của hàm khởi tạo

infPerson.prototype.height = 0;

//Tạo ra 1 instance của Person

//Có cả 3 thuộc tính của mẫu khởi tạo Person

var person = new infPerson(10, "Khoa", "Da Nang");

for (var att in person) {

    console.log(att);

}

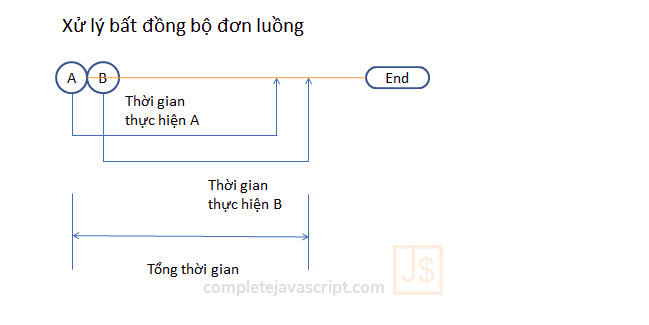
//Xem đối tượng prototype của instance vừa tạo

person.\_\_proto\_\_;

* Sự quan trọng của Prototype:
* Cơ chế kế thừa trong Javascript: Trong JavaScript, việc kế thừa được hiện thực thông qua prototype theo cơ chế prototype-based. Ngắn gọn hơn, để thực hiện kế thừa trong Javascript, bạn cần tạo 1 hàm khởi tạo, sau đó thêm các thuộc tính và phương thức vào thuộc tính prototype của hàm khởi tạo này. Các instance tạo ra bởi hàm khởi tạo này sẽ chứa các thuộc tính và phương thức được định nghĩa ở trên.
* Prototype chain: Prototype rất quan trọng trong việc giúp ta truy cập tới các thuộc tính và phương thức của đối tượng. Đặc tính prototype của đối tượng (hay còn gọi là prototype object) là một “object cha” nơi chứa các thuộc tính và phương thức được kế thừa.

**5. Bất đồng bộ**

Javascript là ngôn ngữ lập trình bất đồng bộ và chỉ chạy trên một luồng. Bất đồng bộ là nhiều công việc có thể được thực hiện cùng lúc. Và nếu công việc thứ hai kết thúc trước, nó có thể sẽ cho ra kết quả trước cả câu lệnh thứ nhất. Vì thế, đôi khi kết quả của các câu lệnh sẽ không trả về đúng theo đúng thứ tự như trực quan của nó.



Cơ chế giúp bạn thực hiện việc này trong JavaScript có thể là sử dụng Callback, Promise hoặc Async/await.

**5.1. Callback hell**

Call Back là một hàm được truyền vào một hàm khác với tư cách như một tham số của hàm đó. Tuy nhiên, Callback cũng có nhược điểm. Đó là khi chúng ta muốn nhiều hành động bất đồng bộ thực hiện theo đúng thứ tự liên tiếp nhau, chúng ta phải gọi nhiều hàm callback lồng vào nhau nhiều lần, gây ra đoạn code rất khó kiểm soát và không tối ưu. Đây gọi là tình trạng Callback Hell. Ví dụ như muốn in các số từ 1 đến 10, mà mỗi hành động in đều là một hàm bất đồng bộ.

function printNumber(number, callback) {

    setTimeout(

        () =& gt; {

        console.log(number)

        callback()

    },

    Math.floor(Math.random() \* 100) + 1

    )

}

function printAll() {

    printNumber(1, function () {

        printNumber(2, function () {

            printNumber(3, function () {

                printNumber(4, function () {

                    printNumber(5, function () {

                        printNumber(6, function () {

                            printNumber(7, function () {

                                printNumber(8, function () {

                                    printNumber(9, function () {

                                        printNumber(10, function () {

                                        })

                                    })

                                })

                            })

                        })

                    })

                })

            })

        })

    })

}

**5.2. Promise**

* Promise là một đối tượng bao hàm một hàm chứa các đoạn code không đồng bộ. 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.
* Promise sinh ra để xử lý kết quả của một hành động cụ thể, kết quả của mỗi hành động sẽ là thành công hoặc thất bại và Promise sẽ giúp chúng ta giải quyết câu hỏi "Nếu thành công thì làm gì? Nếu thất bại thì làm gì?". Cả hai câu hỏi này ta gọi là một hành động gọi lại (callback action).
* Khi một Promise được khởi tạo thì nó có một trong ba trạng thái sau:
* Fulfilled Hành động xử lý xong và thành công
* Rejected Hành động xử lý xong và thất bại
* Pending Hành động đang chờ xử lý hoặc bị từ chối

Trong đó hai trạng thái Reject và Fulfilled ta gọi là Settled, tức là đã xử lý xong.



* Cách tạo một promise, ví dụ:



* Trong đó: resolve: là một hàm callback xử lý cho hành động thành công. reject là: một hàm callback xử lý cho hành động thất bại.
* Công việc mình muốn thực hiện đó là sau khi mình đọc xong nội dung của file 1 và in ra nội dung của file 1 thì lúc này minh mới được tiến hành đọc nội dung của file 2. Như vậy mình đã xử lý bằng cách truyền các callback vào lần lượt trong từng hàm then và lưu ý thì để hàm then phía sau có thể thực hiện được thì callback trong hàm then trước đó phải trả về một promise.

**5.3. Async await**

* Promise đã giải quyết khá tốt những vấn đề của callback. Tuy nhiên, dùng promise đôi khi ta vẫn thấy hơi khó chịu vì phải truyền callback vào hàm then và catch. Code cũng sẽ hơi dư thừa và khó debug, vì toàn bộ các hàm then chỉ được tính là 1 câu lệnh nên không debug riêng từng dòng được. Và khi ES7 ra đời, có một tính năng đó là async/await đã gải quyết được vấn đề này.
* Cách sử dụng asyn/await:
  + Để sử dụng hàm async, ta cần khai báo từ khóa async ngay trước từ khóa định nghĩa hàm. Tức là, với hàm định nghĩa với từ khóa function ta phải khai báo ngay trước function, với hàm mũi tên (arrow function) ta phải khai báo trước tập tham số đầu vào.
  + Kết quả trả ra của hàm async luôn là một Promise dù bạn có gọi await - có xử lý bất đồng bộ hay không. Promise này sẽ ở trạng thái thành công với kết quả được trả ra với từ khóa return của hàm async, hoặc trạng thái thất bại với kết quả được đẩy qua từ khóa throw trong hàm async. Như vậy,chúng ta có thể thấy bản chất của hàm async chính là Promise.
  + Với Promise, ta có thể xử lý ngoại lệ với catch khá đơn giản. Tuy nhiên cũng không dễ dàng theo dõi và dễ đọc. Nhưng với hàm async, việc này cực kì đơn giản bằng từ khóa try catch hệt như các thao tác đồng bộ.
  + Ví dụ:



**6. Class**

6.1. Định nghĩa: Class trong javascript là một kỹ thuật giúp ta tạo ra các lớp mẫu chương trình để thể hiện cho các đối tượng. Qua đó, ta có thể thêm các hành động và thuộc tính của đối tượng vào trong lớp thông qua hai khái niệm thuộc tính và phương thức.

6.2. Cú pháp để tạo ra một class trong JavaSript:

class TenClass {

    // Hàm khởi tạo

    constructor() { ... }

    // Các phương thức

    phuongThuc1() { ... }

    phuongThuc2() { ... }

    phuongThuc3() { ... }

    }

}

* Constructor (hàm khởi tạo) là một phương thức đặc biệt để tạo và khởi tạo một đối tượng từ một class nào đó. Nó sẽ tự động được gọi khi bạn tạo ra một đối tượng từ class. Chỉ có thể có một phương thức đặc biệt với tên là “constructor” trong một lớp.
* Khi tạo một phương thức thì bạn không cần sử dụng từ khóa function.

6.3. Phương thức getter và setter

* Các phương thức getter được sử dụng để truy cập các thuộc tính của một đối tượng. Để tạo phương thức getter trong JavaScript, chúng ta sử dụng từ khóa get. Có bao nhiêu thuộc tính thì sẽ có bấy nhiêu hàm getter trong class. Khi chúng ta sử dụng, chúng ta có thể truy cập giá trị như là một thuộc tính.

// Các phương thức getter

get getHoTen() {

    return this.hoTen;

}

get getGioiTinh() {

    return this.gioiTinh;

}

get getNamSinh() {

    return this.hoTen;

}

get getMSSV() {

    return this.MSSV;

}

// Khởi tạo đối tượng

var sinhVien1 = new SinhVien("Ngô Minh Trung", "Nam", "B1704863", "1999");

// Truy cập giới tính thông qua phương thức getter

console.log(sinhVien1.getGioiTinh);

* Phương thức setter cho phép bạn thay đổi giá trị của thuộc tính của đối tượng. Để tạo ra phương thức setter ta sử dụng từ khóa set. Phương thức setter cho phép bạn thay đổi giá trị của thuộc tính của đối tượng. Để tạo ra phương thức setter ta sử dụng từ khóa set

set setHoTen(hoTen) {

    this.hoTen = hoTen;

}

set setGioiTinh(gioiTinh) {

    this.gioiTinh = gioiTinh;

}

set setNamSinh(namSinh) {

    this.namSinh = namSinh;

}

set setMSSV(MSSV) {

    this.MSSV = MSSV;

}

// Khởi tạo đối tượng

var sinhVien1 = new SinhVien("Ngô Minh Trung", "Nam", "B1704863", "1999");

// Thiết lập lại giới tính

sinhVien1.setGioiTinh = "Nữ";

6.4. Phương thức tĩnh (Static method)

* Từ khóa static định nghĩa một phương thức static cho một lớp, còn được gọi là phương thức tĩnh.Các phương thức tĩnh được gọi mà không cần khởi tạo lớp của chúng và không thể được gọi thông qua một đối tượng của lớp.
* Ví dụ:

class Point {

    constructor(x, y) {

        this.x = x;

        this.y = y;

    }

    static distance(a, b) {

        const dx = a.x - b.x;

        const dy = a.y - b.y;

        return Math.hypot(dx, dy);

    }

}

const p1 = new Point(5, 5);

const p2 = new Point(10, 10);

p1.distance; //undefined

p2.distance; //undefined

console.log(Point.distance(p1, p2)); // 7.0710678118654755

Chúng ta không thể gọi hàm distance thông qua p1 hoặc p2. Mà phải gọi trực tiếp từ class của nó đó chính là Point.

6.5. Hoisting đối với class: Không giống như các hàm và các khai báo JavaScript khác, các khai báo class (lớp) không được hoisting.

6.6. Kế thừa class:

* JavaScript cũng hỗ trợ tính kế thừa (inheritance) như các ngôn ngữ khác, để tạo một lớp kế thừa, bạn sử dụng từ khóa extends. Kế thừa cho chúng ta khả năng sử dụng lại code đã viết. Sử dụng lại các thuộc tính và phương thức của một lớp có sẵn khi tạo một lớp mới giúp chúng ta tiết kiệm thời gian làm việc. Một lớp được kế thừa từ lớp cha sẽ có tất cả tác phương thức từ lớp cha.
* Ví dụ: Ta có một lớp ConNguoi và một lớp SinhVien được kế thừa từ lớp ConNguoi

class ConNguoi {

    constructor(hoTen, namSinh) {

        this.hoTen = hoTen || "unknown";

        this.namSinh = namSinh || "unknown";

    }

    getHoTen() {

        return "Tên " + this.hoTen + ", sinh năm " + this.namSinh;

    }

}

class SinhVien extends ConNguoi {

    constructor(maSSV, hoTen, namSinh) {

        super(hoTen, namSinh);

        this.maSSV = maSSV || "unknown";

    }

    getThongTin() {

        return this.getHoTen() + ", mã sinh viên " + this.maSSV;

    }

}

var sinhVien1 = new SinhVien("B1704863", "Ngô Minh Trung", "1999");

console.log(sinhVien1.getThongTin());

**II. ES6**

**1. Block-scoped: let, const**

* Block Scoped là phạm vi trong một khối, nghĩa là chỉ hoạt động trong phạm vi được khai báo bời cặp {}.

1.1. Dùng từ khóa let để khai báo biến trong ES6

Cú pháp: let var\_name = var\_value;

let cho phép chúng ta cập nhật giá trị của biến chứ không cho phép chúng ta tái khái báo lại biến đó. Giống với var, let cũng có tính hoisting tuy nhiên lại khác nhau ở chỗ thay vì var được khởi tạo với giá trị là undefined thì let sẽ không có bất kỳ giá trị khởi tạo nào. Điều này dẫn đến việc nếu chúng ta sử dụng biến let trước khi khai báo thì sẽ gặp lỗi Reference Error.

1.2. Dùng từ khóa const để khai báo biến trong ES6

* Từ khóa const là viết tắt của từ constant . Nó cũng giống như let, nhưng tuy nhiên là const không thể sắp xếp lại giá trị.
* Tương tự với let cũng có scope là block scoped, và hoisting thì chúng ta có thêm 1 kiểu khai báo biến nữa là const. Trong biến const nếu trường hợp kiểu của biến là primitive (bao gồm string, number, boolean, null, và undefined) thì chúng ta sẽ không thể tái khai báo hay cập nhật giá trị mới để thay thế cho giá trị trước đó của biến.

**2. Arrow function**

* Arrow function - còn được gọi là "fat arrow", là cú pháp được mượn từ CoffeeScript (một ngôn ngữ chuyển tiếp), cú pháp này là cách ngắn gọn hơn dùng để viết function. Ở đây sử dụng kí tự =>, trông giống như một mũi tên "béo". Arrow function là một hàm vô danh và nó thay đổi cách this bind đến function. Arrow function làm code của ta trông ngắn gọn hơn, giúp đơn giản hóa function scoping cũng như từ khóa this.

2.1. Cách dùng arrow function

* Trong trường hợp nhiều tham số:

// (param1, param2, paramN) => expression

// ES5

var multiply = function(x, y) {

    return x \* y;

};

// ES6

var multiply = (x, y) => { return x \* y };

Ví dụ trên cho cùng một kết quả, tuy nhiên cú pháp với arrow function tốn ít dòng mã hơn. Trong trường hợp chỉ có một biểu thức thì không cần tới dấu ngoặc nhọn

* Trong trường hợp 1 tham số: Dấu ngoặc đơn là optional khi chỉ có một tham số

//ES5

var phraseSplitterEs5 = function phraseSplitter(phrase) {

    return phrase.split(' ');

};

//ES6

var phraseSplitterEs6 = phrase => phrase.split(" ");

console.log(phraseSplitterEs6("ES6 Awesomeness"));  // ["ES6", "Awesomeness"]

* Trong trường hợp không có tham số: Dấu ngoặc đơn là bắt buộc khi không có tham số.

//ES5

var docLogEs5 = function docLog() {

    console.log(document);

};

//ES6

var docLogEs6 = () => { console.log(document); }

docLogEs6();

2.2. Khi nào thì nên sử dụng arrow function?: Một usecase phổ biến áp dung arrow function là thao tác mảng, thông thường là khi dùng map hoặc reduce.

2.3. Lưu ý trong việc sử dụng arrow function:

Cú pháp arrrow function là chức năng khá hữu ích trong ECMAScript, tuy nhiên ngoài những ưu điểm thì nó cũng có những hạn chế như việc nó khiến code của bạn tuy ngắn nhưng lại khó hiểu.

* Từ khóa this hoạt động khác hoàn toàn trong arrow function. Các phương thức call (), apply (), và bind () sẽ không thay đổi giá trị của this trong arrow function. (Trong thực tế, giá trị của this bên trong một hàm đơn giản là không thể thay đổi - nó sẽ luôn là giá trị tương tự như khi hàm được gọi) Nếu bạn cần bind đến một giá trị khác, cần sử dụng function expression như bình thường.
* Arrow function không thể được sử dụng làm constructor như function. Nếu khởi tạo new với arrow function, nó sẽ throw ra lỗi. Arrow function không có thuộc tính prototype hay internal methods. Trong trường hợp đó nên dùng cú pháp tạo class của ES6.
* Arrow function được thiết kế không thể sử dụng như là generator. Thay vào đó ta có thể sử dụng từ khóa generators trong ES6. Tốt nhất nên cân đối giữa việc dùng function và dùng arrow function như sau:
  + Sử dụng function trong global scope trong Object.prototype properties
  + Sử dụng class cho object constructors.
  + Sử dụng => ở những chố còn lại

**3. Default Parameter**

* Default Parameter là giá trị mặc định của tham số khi truyền vào các function. Đối với Javascript thì có nhiều bạn chưa biết chức năng này mặc dù trong ES5 đã cung cấp săn cho chúng ta, tuy nhiên người ta cảm thấy cách tạo giá trị mặc định trong ES5 vẫn không hay nên họ đã bổ sung một cách khác mới hơn và đơn giản hơn rất nhiều trong ES6.
* Trong ES6 có cách khai báo giá trị mặc định đơn giản hơn rất nhiều, cách này cũng tương tự như khai báo trong PHP đó là sử dụng phép gán ngay tại vị trí khai báo tham số cho function.
* Ví dụ:

function sayHello(domain = 'freetuts.net')

{

    return domain;

}

// Sử dụng

console.log("KHÔNG truyền tham số: " + sayHello());

console.log("CÓ truyền tham số: " + sayHello('facebook.com'));

**4. Rest**

* Rest Parameters dịch theo tiếng Anh chuyên ngành công nghệ thông tin có nghĩa là tham số còn lại, điều này có nghĩa là bạn có thể khai báo một hàm với số lượng tham số không xác định, đây là một tính năng mới khiến Javascript ngày càng trở nên mạnh mẽ hơn.
* Rest parameter trong ES6 giúp chúng ta có thể định nghĩa một hàm với số lượng tham số có thể thay đổi tuỳ ý. Hay nói theo cách khác khi chúng ta không biết chắc chắn số lượng tham số cần có của một hàm chúng ta có thể sử dụng rest parameter.
* Để khai báo các tham số còn lại của một function thì bạn đặt 3 dấu chấm . trước biến đại diện. Cú pháp:

let functionName = (param1, param2, ...other) => {

    //

}

Trong đó tham số other là một Rest Parameter vì nó có 3 dấu chấm đặt ở trước.

* Ví dụ:

// Khai báo hàm

let domainList = (main, sub, ...other) =>

{

    console.log("Main: " + main);

    console.log("Sub: " + sub);

    console.log("Other");

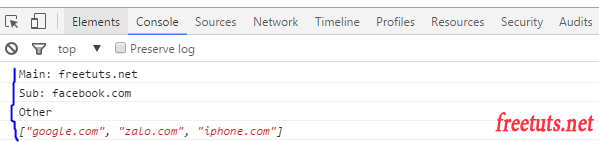
    console.log(other);

}

// Gọi hàm

domainList('freetuts.net', 'facebook.com', 'google.com', 'zalo.com', 'iphone.com');

Kết quả:



Hàm này có 3 tham số truyền vào là main, sub và Rest Parameter other.

**5. Spread**

* Trong Javascript, spread operator là nói đến cách sử dụng ký hiệu dấu ba chấm ...
* Khi làm việc với array, mà ví dụ cụ thể ở đây là copy một array, thông thường sẽ sử dụng cú pháp .slice. Tuy nhiên với ES6, ta có thể sử dụng cú pháp spread operator để duplicate một array. Sử dụng Spread Operator với Array để nối mảng và copy mảng. Ví dụ:

const smiles = ['🙂', '🙂', '🙂'];

// ES5

const cloneSmiles = smiles.slice();

// ES6

const cloneSlimesES6 = [...smiles];

**6. Destructuring Assignment**

* Destructuring Assignments đơn giản chỉ là cách tách các phần tử của Array hoặc Object thành nhiều biến chỉ bằng một đoạn code duy nhất.
* Array Destructuring: Khi sử dụng Array Destructuring, trên thực tế một iterator sẽ được sử dụng để tách các giá trị cần thiết ra khỏi destructuring source (hay giá trị ở RHS của phép gán). Do đó, chúng ta có thể sử dụng Array Destructuring cho các iterable values (không phải chỉ riêng các arrays), cụ thể sẽ là strings, arrays, sets, maps, function arguments, DOM elements. Chúng ta cũng có thể sử dụng Array Destructuring với các toán tử như Spread nếu cần thiết.
* Object Destructuring: Object Destructuring hay còn gọi với một tên khác là Object Property Assignment Pattern cho phép chúng ta gán property value của một object cho các biến tương ứng.

**7. Template Literals**

* Temaplate Literals là một cú pháp mới dùng để khai báo biến, được thêm vào trong phiên bản ES6. Khi sử dụng cú pháp này các bạn có thể:
  + Khai báo biến nhiều dòng một cách hết sức đơn giản. (Nếu bạn sử dụng cách truyền thống thì sẽ không thể nào khai báo được một biến mà có sử dụng ký tự xuống dòng (xem).
  + Binding biến vào trong template một cách dễ dàng.
* Và để khai báo Template Literals thì các bạn cũng sử dụng cú pháp như trước. Nhưng chỉ có một điều duy nhất là khai thay vì khai báo bằng dấu ' hoặc " thì bạn đổi bằng dấu `:

var variableName = `content`;

* Và nếu như bạn muốn binding một biến khác vào trong nội dung của Template Literals thì bạn sử dụng cú pháp sau:

${variable}