# **CHAPTER 1: KHÁI NIỆM CƠ BẢN**

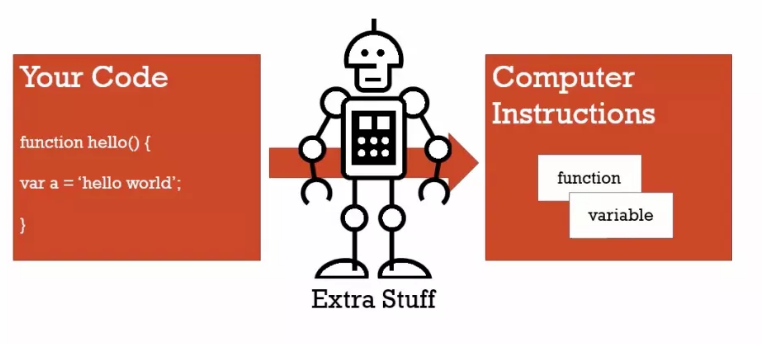
1. **Syntax Parser, Execution Context, Lexical Environment:**
2. **Syntax Parser là gì?**

Javascrip là một “ngôn ngữ lập trình cấp cao” high level programming language, vì thế cú pháp (syntax) của nó khá giống ngôn ngữ tự nhiên.

Ví dụ:



Thực chất Syntax Parser cũng chỉ là một chương trình được các lập trình viên khác viết ra mà thôi. Syntax Parser trong javascript sẽ dịch đoạn code này sang mã máy vì mã máy quá phức tạp và khó, không phải ai cũng học được.



1. **Lexical Environment là gì?**

Lexical Environment là nơi chúng ta viết code của mình (đặt biến, hằng, hàm ở đâu?) Và điều này rất quan trọng.

Như đã nói ở Syntax Parser, code ta viết sẽ được dịch sang mã máy, vì thế nhìn code là có thể biết được biến, hàm chúng được tạo như thế nào trong bộ nhớ máy tính, chúng tương tác với nhau như thế nào.

Ví dụ:



Lexical Environment tức là bạn đặt biến a trong hàm greet() ( Đã trả lời cho câu hỏi: Nơi đặt biến a?)

Một số trang sẽ nói là Lexical Environment là Scope hay Lexical Scope.

Còn đây là định nghĩa gốc chính chủ:

**Lexical Environment  is the place where “the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code” is stored.**

Từ định nghĩa này, bạn sẽ thấy một loạt **từ chuyên môn ( Jargon )** khác như:

**\* association of Identifiers**

**\* lexical structure**

Rồi từ 2 cái này lại đẻ ra:

- environment record

- reference to the outer environment

- Declarative environment record

- Object environment record\

1. **Execution Context là gì?**

Code ta viết sẽ được dịch sang mã máy. Vì thế trong quá trình dịch này sẽ có nhiều thứ xảy ra, một trong số đó là Execution Context.

Ví dụ:



Thì Execution Context sẽ quyết định tạo *biến a* trước hay *biến count* trước. Rồi chương trình sẽ chạy *greet()* trước hay *shakeHands()* trước.

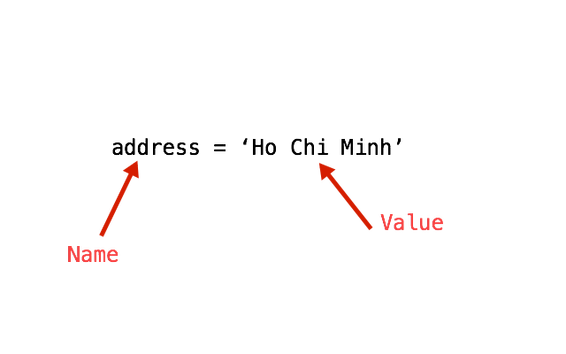
Sẽ có một loạt thuật ngữ chúng ta cần hiểu liên quan đến Execution Context như:

* Global code
* Function code
* Eval Code
* Execution Context Stack
* Creation Stage
* Code Execution Stage

1. **Name-Value Pair và Object:**
2. **Name-Value Pair là gì?**

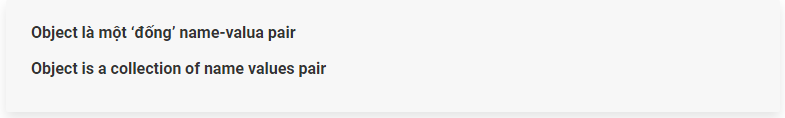
Name-Value là một cặp tên-giá trị, trong trường hợp này là: name là address, value là ‘Ho Chi Minh’

Ví dụ:

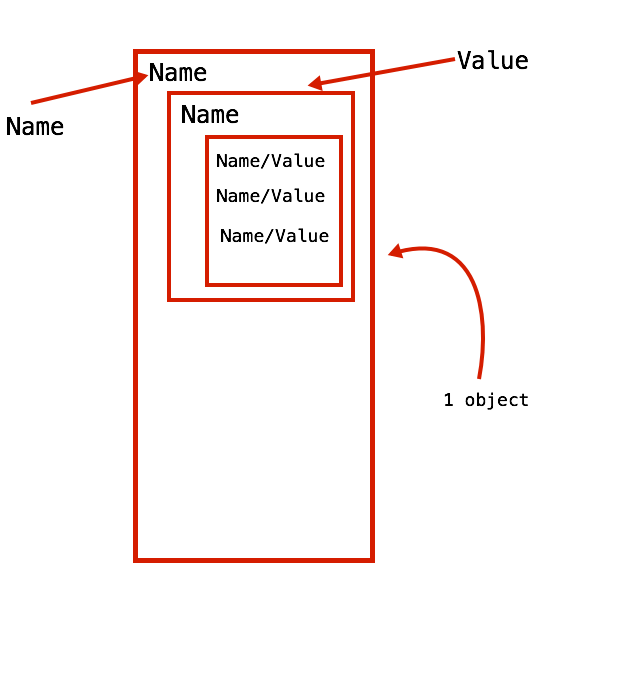


1. **Object trong Javascript là gì?**

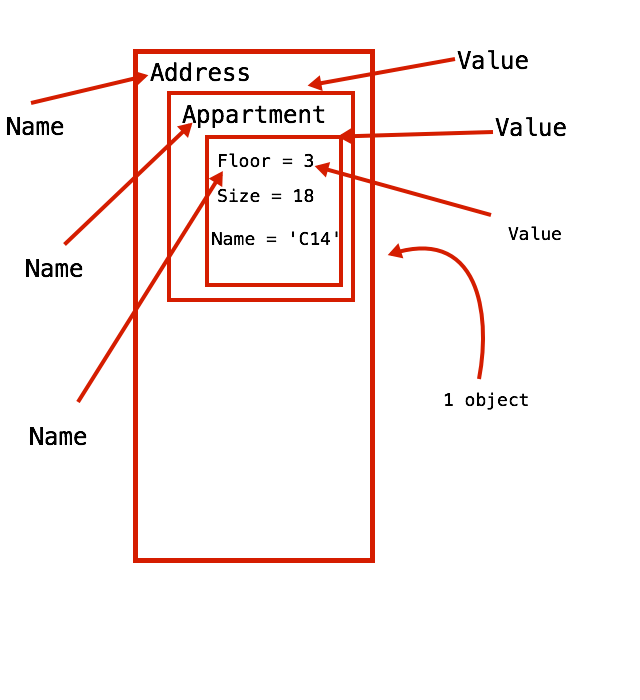
Mấy ngôn ngữ khác, có thể định nghĩa về object rất phức tạp, nhưng trong Javascript định nghĩa object rất đơn giản:



Như hình bên dưới, các name-value pair có thể lồng vào nhau: Value của một name có thể là một ‘đống’ name-value pair khác chứ không phải chỉ có value không.



Để dễ hình dung, ta có thể xem ví dụ một object trong Javascript:

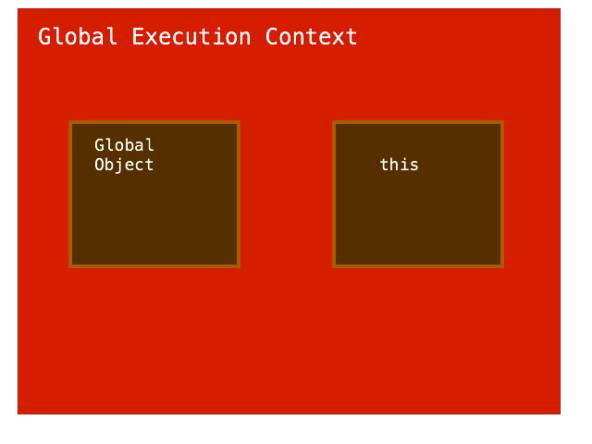


1. **Global Environment và Global Object:**
2. **Global Execution Context là gì:**

Ở trước, chúng ta đã giải thích [Execution Context là gì](https://niviki.com/2017/06/hoc-javascript-1-syntax-parser-execution-context-lexical-environment-la-gi/). Bất cứ khi nào code chạy trong Javascript, nó được chạy bên trong một Execution Context. Sẽ có nhiều Execution Context nhưng cái lớn nhất, ‘leader’ của các Execution Context khác được gọi là Global Execution Context.

1. **Global Execution Context có gì:**

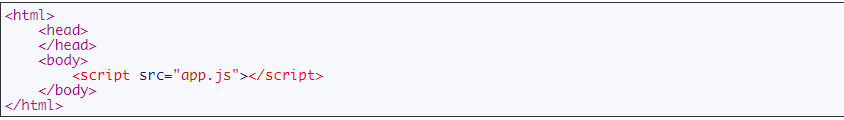
Global Execution Context sẽ tạo cho bạn 2 thứ là **Global Object** và biến là **this**



**Chạy Javascript code**

Bạn chỉ cần tạo một file **index.html** và một file **.js** là được

index.html:

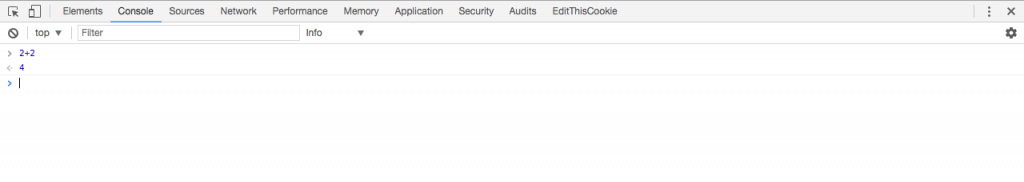


**Chrome DevTool**

Chọn tab Console để xài Command Line API. Tính năng của cái này nhiều lắm mình không nói hết. Nhưng nó có một tính năng giống như Read–eval–print loop của Python hay Ruby.

Tức là gõ code vào cái chạy được luôn, sẽ có 2 trường hợp

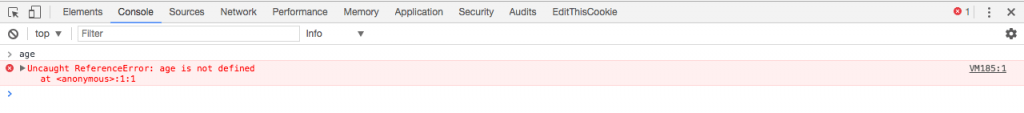
* Bạn có thể chạy các biểu thức toán học, chuỗi, true, false:  
  Ví dụ bên dưới mình gõ 2+2 và nhấn Enter, kết quả sẽ được 4.



Hoặc cộng 2 kí tự:

Chrome DevTool

* Bạn có thể truy cập giá trị biến, hằng, class, hàm, vv trong file đang chạy. Hiện tại trong ví dụ này là file **app.js.**Ví dụ mình muốn truy cập biến **age,**thì mình sẽ gõ **age** vào và nhấn Enter:\

 Thì nó sẽ báo lỗi vì trong file **app.js** chưa có biến nào tên **age**hết.

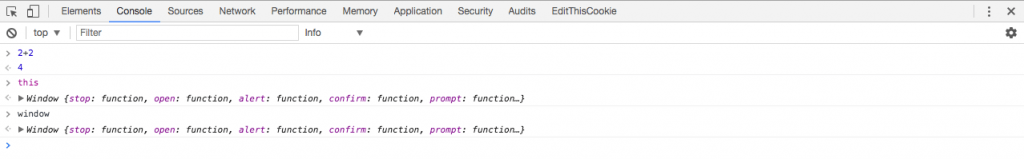
**Global Object**

Global Execution Context sẽ tạo cho bạn 2 thứ là **Global Object** và biến là **this.**Bạn lưu ý là file **app.js**của mình vẫn trống trơn

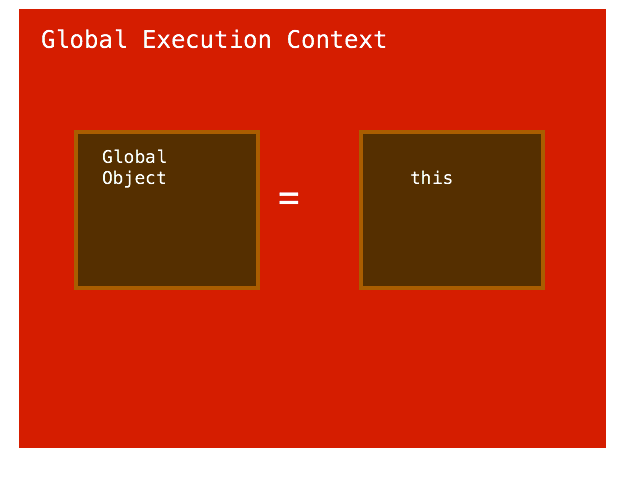
Tiếp theo bạn thử gõ **this** vào và nhấn Enter:



Tiếp theo bạn thử gõ **window** và nhấn Enter:

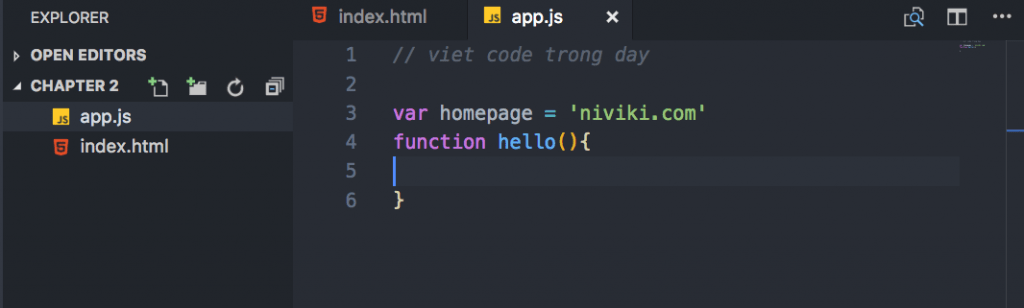


Giải thích: Mặc dù file app.js không có gì nhưng nó vẫn chạy, [Syntax Parser](https://niviki.com/2017/06/hoc-javascript-1-syntax-parser-execution-context-lexical-environment-la-gi/) sẽ chuyển code sang mã máy, do không có code nên nó bỏ qua và không có lỗi. Tiếp theo Global Execution Context sẽ tạo cho bạn 2 thứ là **Global Object**  là window và biến là **this**có giá trị bằng với window luôn.

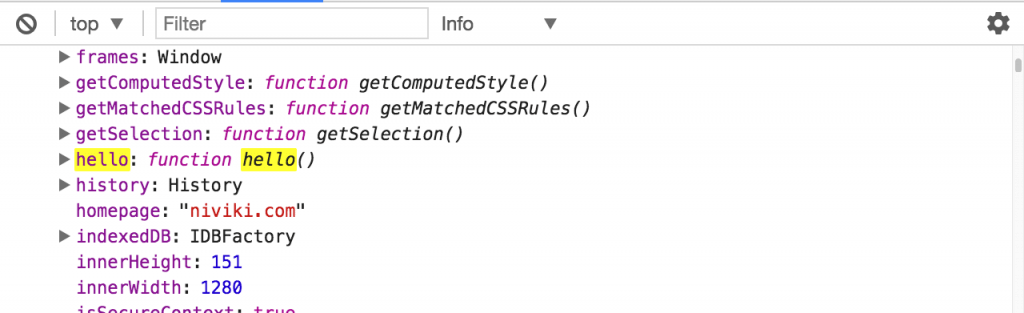


Vì chúng ta đang chạy JS code trên trình duyệt nên global object là window, nhưng nếu chạy ở môi trường khác như NodeJS thì global object sẽ khác. Bạn có thể xem thêm global object của NodeJS [tại đây](https://nodejs.org/api/globals.html#globals_global_objects)

**Code của bạn**



Bạn gõ lại window để lấy global object. Sau đó bạn mở object này ra để xem các [name-value pair](https://niviki.com/2017/06/hoc-javascript-1-syntax-parser-execution-context-lexical-environment-la-gi/) của object window này.



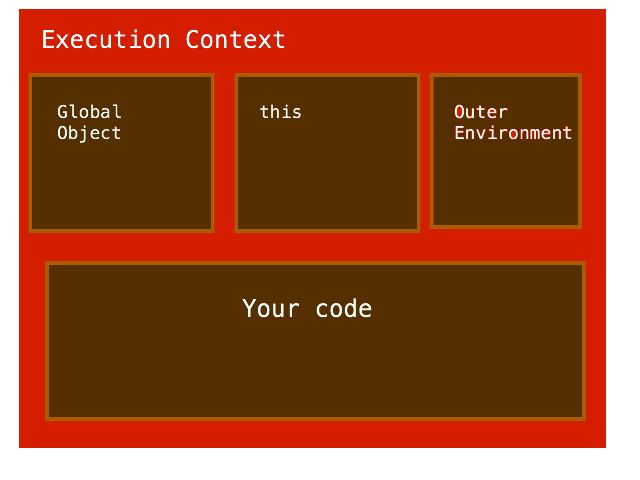
**Global rốt cuộc là gì**

Như ví dụ trên, bạn thấy là cả biến **homepage**và hàm **hello()**đều thuộc global object window.

Vậy global tức là những gì không ở bên trong function

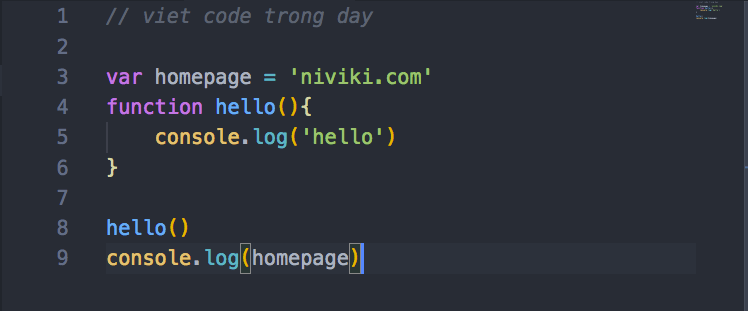
**Kết luận**

Bạn bắt đầu hình dung được code Javascript chạy được nhờ có Execution Context. Nó tạo thêm cho chúng ta những thứ như global object và this. Ngoài ra còn có Outer Environment

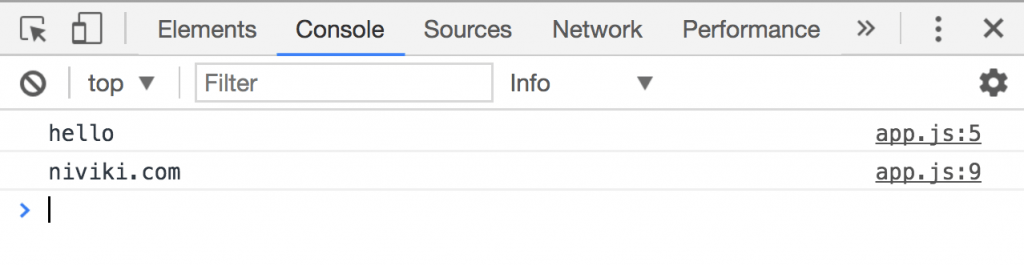


1. **Hoisting:**

Cho đoạn code sau:



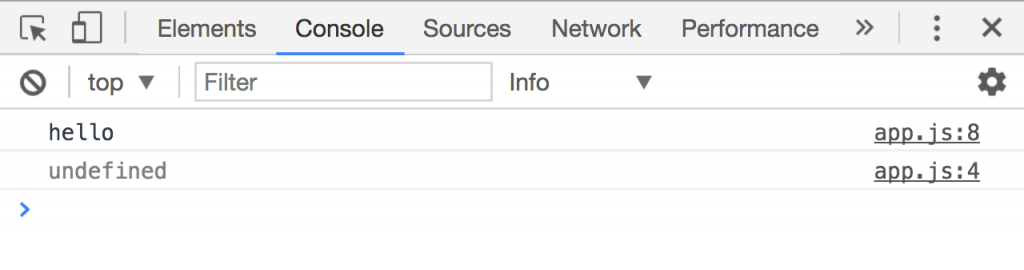
Bạn có thể đoán được kết quả in ra console là gì?



Oke, bây giờ chúng ta thay đổi code một tý:



Lần này bạn đoán kết quả là gì nào?



Ở một số ngôn ngữ khác, nếu làm như vậy sẽ báo lỗi. Nhưng trong Javascript, chúng ta không thấy lỗi gì và nhận được giá trị là **undefined.**Một số tài liệu trên mạng sẽ giải thích hoisting là đưa tất cả khai báo lên trên đầu file, nhưng giải thích như vậy chưa đúng bản chất của vấn đề. Bởi vì code bạn viết được dịch sang mã máy, nên trong quá trình dich này sẽ xảy ra một số công việc bên trong nó.  Chúng ta cùng tìm hiểu thực chất của vấn đề tại sao lại như vậy nhé!

* **Hoisting là gì?**

Với [Execution Context](https://niviki.com/2017/06/hoc-javascript-1-syntax-parser-execution-context-lexical-environment-la-gi/) sẽ có **2 giai đoạn** là Khởi tạo và Thực thi. Trong giai đoạn khởi tạo ta có thể mô phỏng lại những gì Execution Context thực hiện qua hình sau: 

Ngoài việc tạo Global Object, this, Outer Environment như đã biết [ở bài trước](https://niviki.com/2017/06/hoc-javascript-3-global-environment-va-global-object/), thì sẽ có thêm công việc cài đặt ( chuẩn bị ) bộ nhớ cho hàm và biến. Tức là [Syntax Parser](https://niviki.com/2017/06/hoc-javascript-1-syntax-parser-execution-context-lexical-environment-la-gi/) sẽ đọc hết code có trong file và làm công việc này, chứ hoisting không phải là**đưa tất cả code khai báo lên đầu file** như chúng ta vẫn nghĩ.

Quay lại đoạn code này:



Khi chạy, ngay trước khi chạy từng dòng từ dòng 3,  Execution Context **đã tạo bộ nhớ** cho biến **homepage.**Nhưng lúc này nó chưa thực hiện phép gán giá trị, do đó nó không biết giá trị của biến này là gì nên nó phải đặt cho một giá trị mặc định là **undefined** cho **homepage.** Còn với hàm lúc khai báo thì không có phép gán nên phải khởi tạo luôn phần body của hàm là

console.log('hello')  
Sau khi khởi tạo xong thì mới tới giai đoạn là Thực thi chạy từng dòng từ dòng 3 xuống.

**Undefined trong Javascript**

Chúng ta đã biết là kết quả của đoạn code này là undefined:



Và undefined là một keyword trong Javascript. Chúng ta có thể kiểu tra một biến là undefined hay không:



Thậm chí có thể gán một biến bằng với undefined



Tips: Không nên gán giá trị của biến bằng undefined vì sẽ khó debug

1. **Execution Stack là gì?**

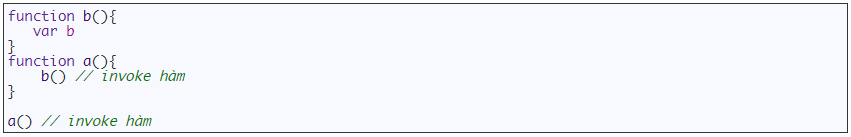
**Invoke hàm**

Quay lại vấn đề để gọi hàm ( chạy hàm ) tiếng Anh là ìn vâu kờ ( invoke )

Oke, sau khi khai báo hàm, để chạy ta dùng 2 dấu ( ) sau tên của hàm đó, ví dụ:

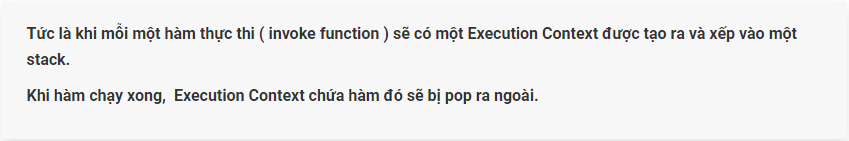


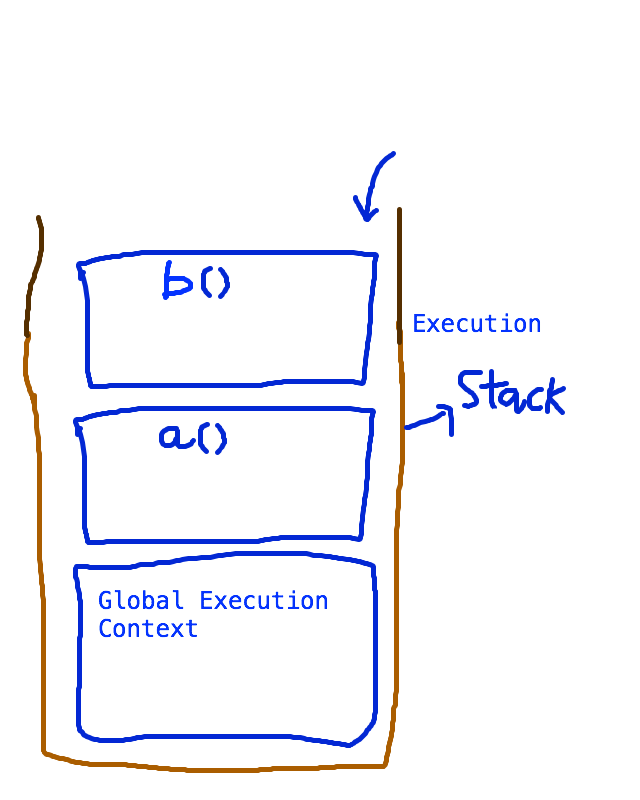
Cho ví dụ sau, hỏi Execution Context được tạo như thế nào:



**Execution Stack**

Khi chạy, trước tiên Global Execution Context sẽ được tạo trước, tiếp theo là các Execution Context khác được tạo và bỏ vào một stack gọi là **Execution Stack**





**Variable Environment**

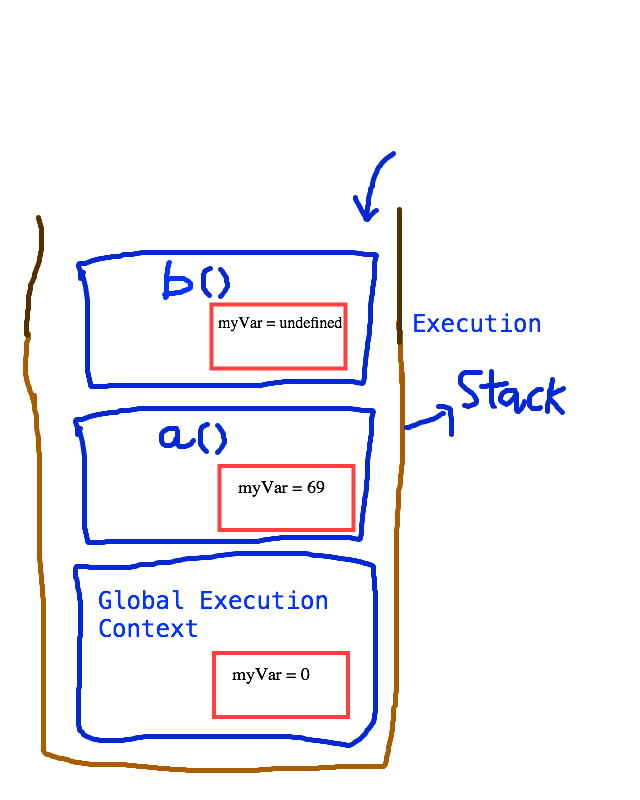
Cũng đoạn code trên, ta thêm một chút:



Khi Execution Context được tạo, nó sẽ chứa luôn các biến bên trong nó.

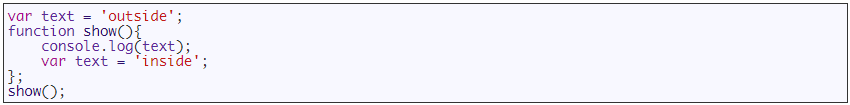
Như ví dụ trên, **myVar**được khởi tạo 3 lần ở 3 Execution Context **khác nhau**, mỗi nơi nó một giá trị ***khác nhau.***

Khi hàm chạy xong, Execution Context bị pop ra hết khỏi **Execution Stack** và chỉ còn 1 Global Execution Context. Vì thế giá trị của *myVar* sau khi chạy hàm **a()** là 0

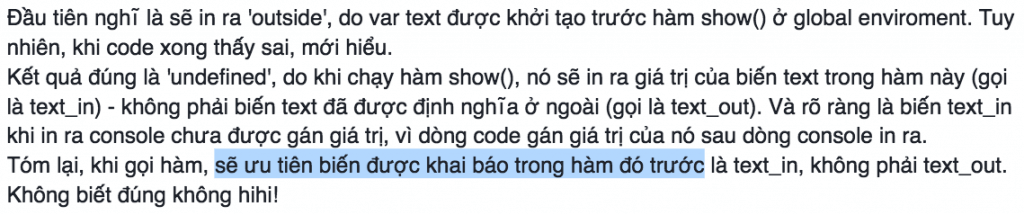


1. **Scope Chain là gì?**
2. **Giới thiệu:**

Ở bài trước [Execution Stack là gì?](https://niviki.com/2017/06/hoc-javascript-5-execution-stack-la-gi/), mình có để đoạn code sau:



Có một bạn vào trả lời như sau:



Chúng ta sẽ tập trung vào đoan code này



Theo bạn, kết quả in ra là gì, giải thích tại sao?

Nếu bạn có theo dõi các bài trước, thì có thể vẽ được Execution Stack như thế này:

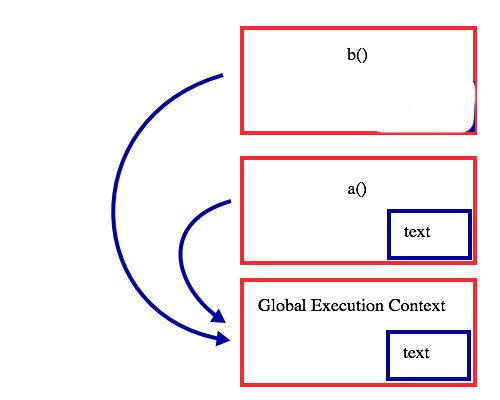


Có thể bạn đang nghĩ vì không có biến text trong hàm b() nên có thể chương trình báo lỗi.

1. **Outer Environment**

Ở bài [Global Environment và Global Object](https://niviki.com/2017/06/hoc-javascript-3-global-environment-va-global-object/), mình có nhắc qua về Outer Environment, vậy nó là gì? Outer Environment của một Execution Context sẽ trỏ tới một Execution Context khác ‘bao’ bên ngoài gần nó nhất ( hay ám chỉ Lexical Environment của Execution Context đó  )

Khó hiểu lúng đúng không? Mình cũng thấy vậy, thiệt ra định nghĩa dài dòng vậy thôi chứ nó đơn giản lắm.



Giống ví dụ trên Outer Environment của hàm cả hàm **a()** và **b()** đều là global

1. **Scope Chain là gì?**

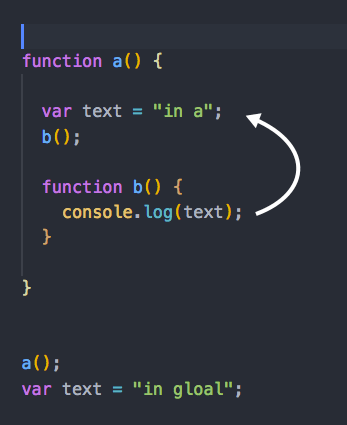
Quay lại đoạn code trên, kết quả sẽ là **undefined**

Vì **trong một Execution Context**, nếu ta truy cập giá trị một biến, mà không tìm thấy biến đó trong Execution Context hiện tại thì nó sẽ tìm ở Outer Environment



Trong **b()**, **text** không có nên nó tìm biến **text** ở Outer Environment là ở Global, mà [do hoisting](https://niviki.com/2017/06/hoc-javascript-4-hieu-ro-khoi-tao-va-hoisting/) nên **text** mới khởi tạo chứ chưa gán giá trị nên kết quả là **undefined**.

Ta chỉnh code lại một tý, thì kết quả là ‘in a’ do Outer Environment của b() đã khác:

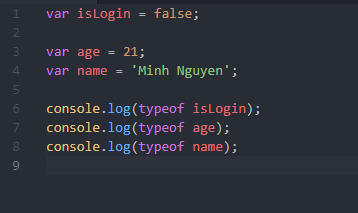
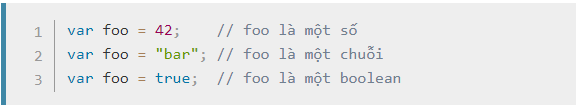


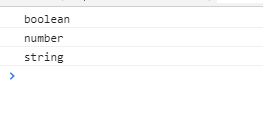
# **CHAPTER 2: TOÁN TỬ**

1. **Dynamic Typing:**

- Tất cả các ngôn ngữ lập trình đều có cấu trúc dữ liệu dựng sẵn, nhưng mỗi ngôn ngữ thường có những kiểu cấu trúc dữ liệu khác nhau. Bài viết này sẽ có gắng liệt kê những kiểu dữ liệu dựng sẵn trong Javascript và những thuộc tính của chúng; chúng có thể được dùng để xây dựng những kiểu cấu trúc dữ liệu khác. Khi có thể, rút ra so sánh với những ngôn ngữ khác.

**- JavaScript** là một ngôn ngữ định kiểu yếu hay động. Điều đó nghĩa là không cần phải khai báo kiểu của các biến trước khi dùng. Kiểu sẽ được xác định tự động trong khi chương trình được thực thi. Điều đó cũng có nghĩa là một biến có thể chứa giá trị của các kiểu dữ liệu khác nhau:



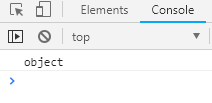
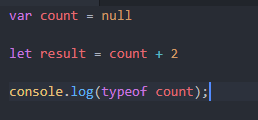


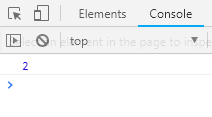
1. **Primitive Type:**

**-** Là kiểu dữ liệu không phải là object và không có phương thức trong đó

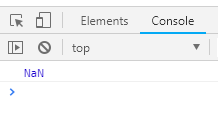
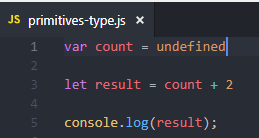
**6 primitives type**

* undefined: biến chưa tồn tại
* null: có tồn tại rồi và chưa có giá trị
* boolean
* string
* number
* symbol (ES6)





Không nên gán 1 biến bằng undefined, vì kết quả sẽ cho ra như vậy

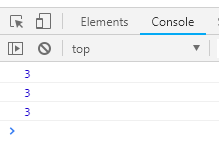
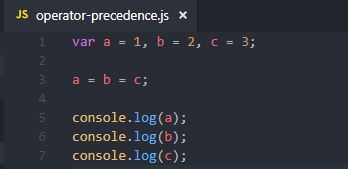


1. **Operator Precedence:**

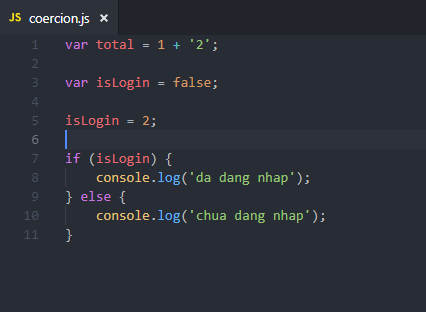
Thứ tự ưu tiên của toán tử

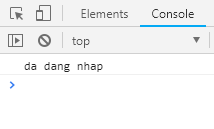
|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator type** | **Associativity** | **Individual operators** |
| 20 | [Grouping](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Grouping) | n/a | ( … ) |
| 19 | [Member Access](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Dot_notation) | left-to-right | … . … |
| [Computed Member Access](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors#Bracket_notation) | left-to-right | … [ … ] |
| [new (with argument list)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) | n/a | new … ( … ) |
| [Function Call](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions) | left-to-right | … ( *…*) |
| 18 | [new (without argument list)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) | right-to-left | new … |
| 17 | [Postfix Increment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment) | n/a | … ++ |
| [Postfix Decrement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Decrement) | … -- |
| 16 | [Logical NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_NOT) | right-to-left | ! … |
| [Bitwise NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_NOT) | ~ … |
| [Unary Plus](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Unary_plus) | + … |
| [Unary Negation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Unary_negation) | - … |
| [Prefix Increment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Increment) | ++ … |
| [Prefix Decrement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Decrement) | -- … |
| [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) | typeof … |
| [void](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void) | void … |
| [delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) | delete … |
| [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) | await … |
| 15 | [Exponentiation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Exponentiation) | right-to-left | … \*\* … |
| 14 | [Multiplication](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Multiplication) | left-to-right | … \* … |
| [Division](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Division) | … / … |
| [Remainder](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Remainder) | … % … |
| 13 | [Addition](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Addition) | left-to-right | … + … |
| [Subtraction](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Subtraction) | … - … |
| 12 | [Bitwise Left Shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | left-to-right | … << … |
| [Bitwise Right Shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | … >> … |
| [Bitwise Unsigned Right Shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | … >>> … |
| 11 | [Less Than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Less_than_operator) | left-to-right | … < … |
| [Less Than Or Equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Less_than__or_equal_operator) | … <= … |
| [Greater Than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Greater_than_operator) | … > … |
| [Greater Than Or Equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Greater_than_or_equal_operator) | … >= … |
| [in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) | … in … |
| [instanceof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof) | … instanceof … |
| 10 | [Equality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Equality) | left-to-right | … == … |
| [Inequality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Inequality) | … != … |
| [Strict Equality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Identity) | … === … |
| [Strict Inequality](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Nonidentity) | … !== … |
| 9 | [Bitwise AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_AND) | left-to-right | … & … |
| 8 | [Bitwise XOR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_XOR) | left-to-right | … ^ … |
| 7 | [Bitwise OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_OR) | left-to-right | … | … |
| 6 | [Logical AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_AND) | left-to-right | … && … |
| 5 | [Logical OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_OR) | left-to-right | … || … |
| 4 | [Conditional](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) | right-to-left | … ? … : … |
| 3 | [Assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators) | right-to-left | … = … |
| … += … |
| … -= … |
| … \*\*= … |
| … \*= … |
| … /= … |
| … %= … |
| … <<= … |
| … >>= … |
| … >>>= … |
| … &= … |
| … ^= … |
| … |= … |
| 2 | [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) | right-to-left | yield … |
| [yield\*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield*) | yield\* … |
| 1 | [Comma / Sequence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator) | left-to-right | … , … |

Ví dụ:



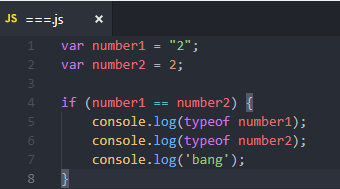
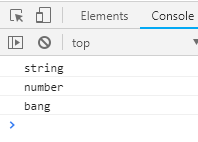
1. **Coercion:**





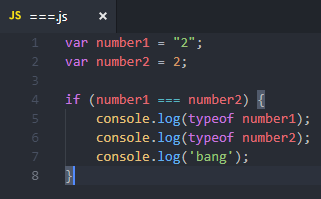
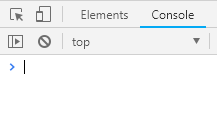
Do **coercion** nên khi gán isLogin = 2 nó hiểu là true

**Tại sao nên dùng so sánh === ?**

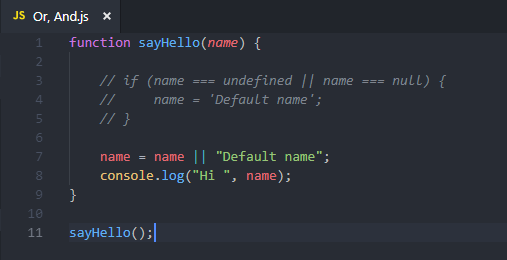
**Coercion** sẽ hiểu và chuyển đổi kiểu dữ liệu của 1 trong 2 thằng để so sánh với nhau.

**Khi dùng ===:**

Nó sẽ so sánh đúng kiểu dữ liệu của 2 thằng

**Toán tử || và &&**

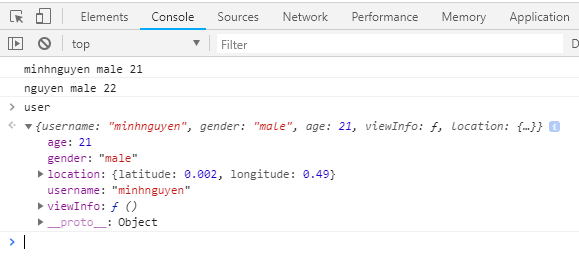
 

# **CHAPTER 3: HÀM CỦNG LÀ OBJECT**

1. **Object:**
2. **Khởi tạo object:**

* **Object Literals**
* **Function**
* **Class (ES6)**





Object có:

- properties (name-value)

- method

- object khác

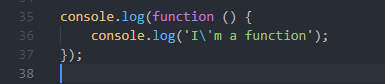
1. **Truy cập thành phần của object:**



1. **First class function languages:**

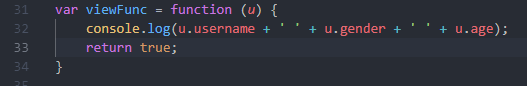
- Coi functions như first-class citizens. Coi function như những kiểu dữ liệu khác như string, number, boolean,… mà những kiểu dữ liệu này có thể là đối số của một hàm khác, kết quả trả về của một hàm khác, hoặc gắn nó vào một biến.

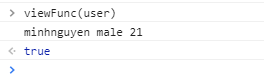
VD1:



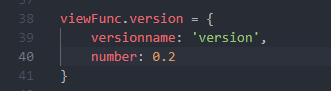


VD2:





VD3: Thêm object vào function

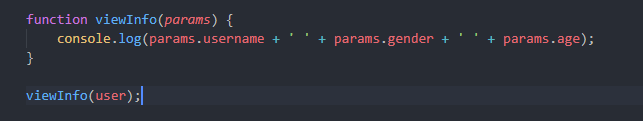


Vì function là object nên ta có thể thêm object khác vào function

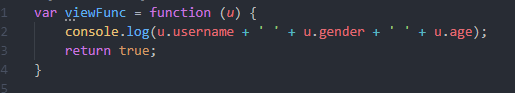


1. **Function Declaration và Function Expression:**

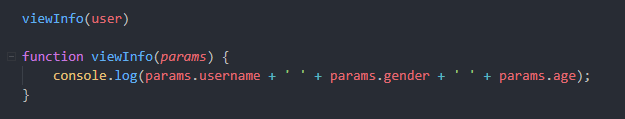
Ví dụ: Function Declaration (statement):



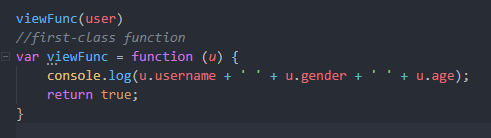
Ví dụ: Function Expression:

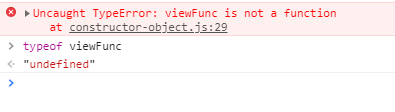


Khác nhau của 2 thằng này ở hoisting



Invoke hàm ở trên function declaration vẫn chạy được



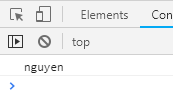
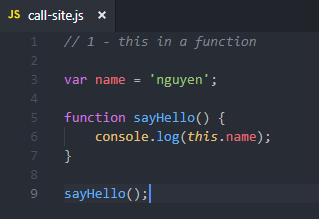


Còn ở function expression khi truy cập biến var viewFunc nó chỉ mới khởi tạo dữ liệu và gán bằng undefined khi gặp dấu = nó mới gán giá trị và lúc này mới invoke và chạy được hàm

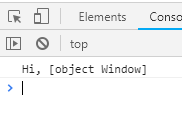
# **CHAPTER 4: THIS TRONG JAVASCRIPT**

1. **This:**
2. **Call-site & Call-stack:**

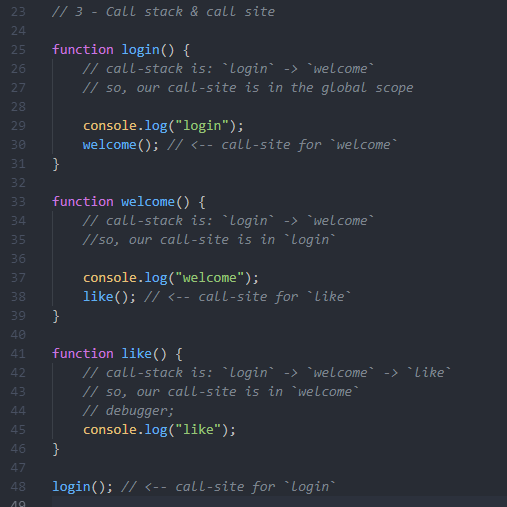
This trong 1 function sẽ trỏ đến global object



This trong function welcome vẫn trỏ đến global



VD: Call-site & Call-stack



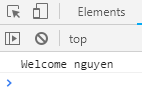
1. **Implicit Binding:**

Ở hình dưới, call-site của welcome() là global nên this.username nó sẽ tìm ở global và không thấy username nên sẽ ra undefined





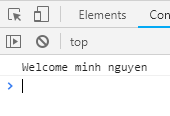
Sửa code lại thành user.welcome(), call-site của welcome() lúc này là user (hay con gọi là context)



1. **Implicit Lost:**

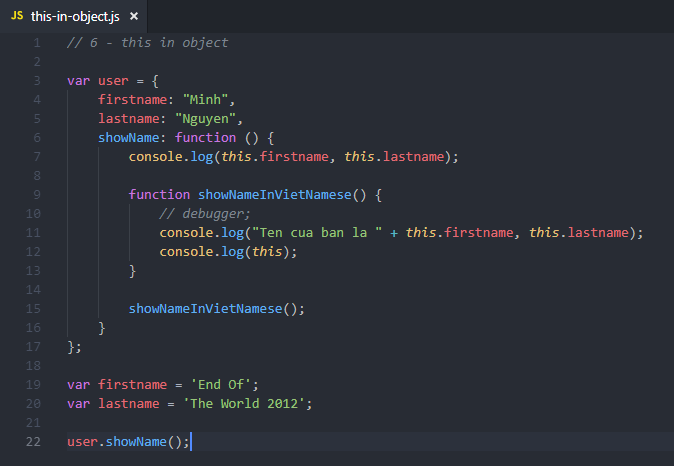


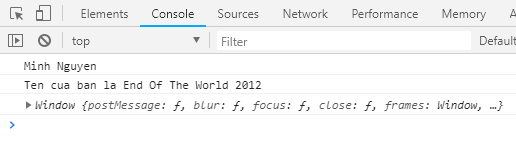
Vì call-site của hi() đang là global nên username nó sẽ lấy ở global object là ‘minh nguyen’



1. **This in object:**

Call-site của function showNameInVietnamese() là global nên nó sẽ tìm firstname và lastname ở global object.





Để nó sử dụng firstname và lastname trong user, ta thêm self = this, khi đó self này chính là user.



1. **Binding:**

Thay vì gắn this cho self ta dùng call(this)



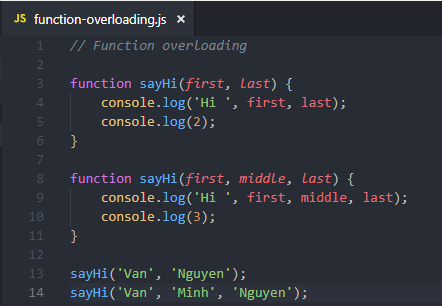
# **CHAPTER 5: HIỂU THÊM VỀ HÀM**

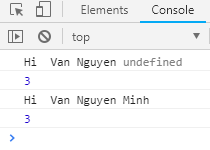
1. **Function:**
2. **Function Arguments:**



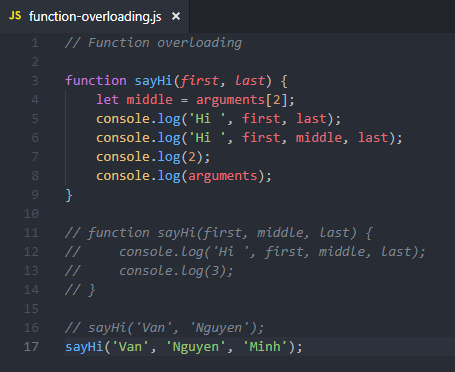
1. **Function overloading:**

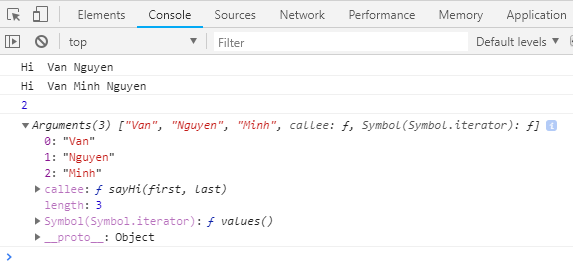
Hình dưới, do cơ chế hoisting nên hàm sayHi có 3 params sẽ chạy



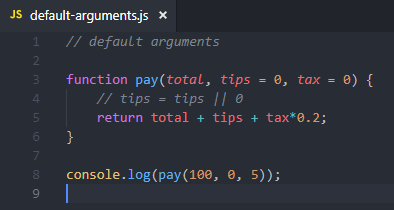


Thường thì không xài function overloading mà xài arguments vì nó flexible hơn :v





1. **Default Arguments:**

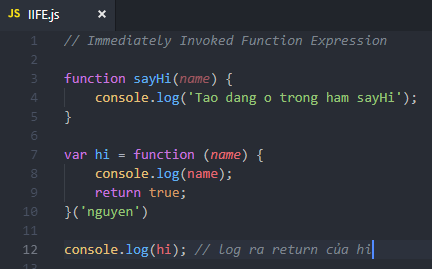


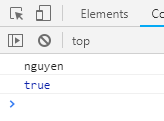
Thay vì dùng toán tử || để kiểm tra thì ta gán arguments mặc định ban đầu luôn.

1. **Immediately Invoke Function Expression (IIFE):**

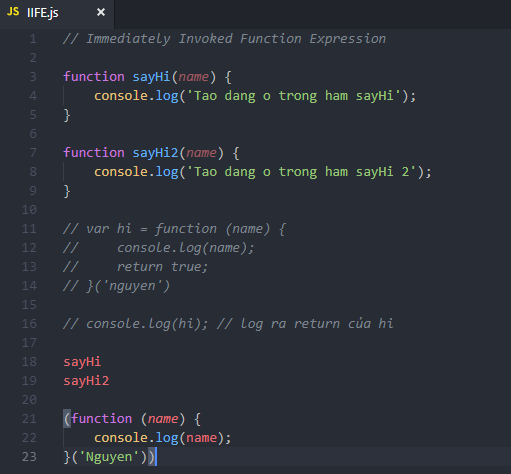
**Là hàm trong javascript mà nó chạy ngay khi được định nghĩa.**

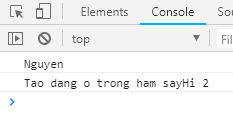
VD:





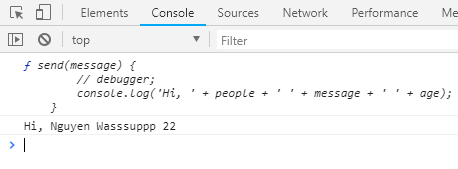
Lưu ý khi dùng IIFE thì nó sẽ chạy trước và sau đó chạy tiếp function ở trên gần nó nhất



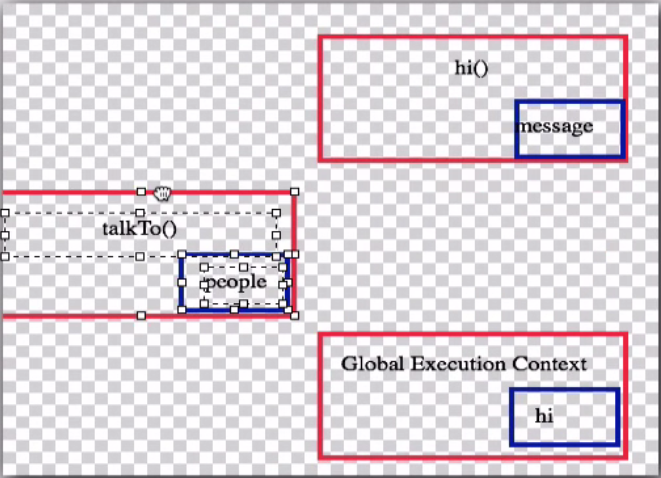


1. **Closure:**

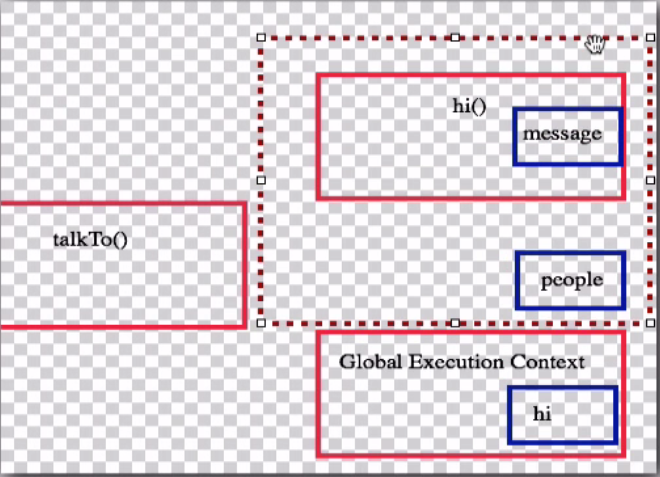


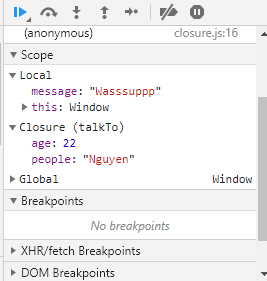
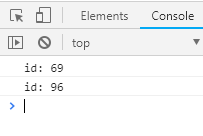


Khi Execution Context của function **Hi()** thực thi thì nó đã remove EXC của function **talkTo()**



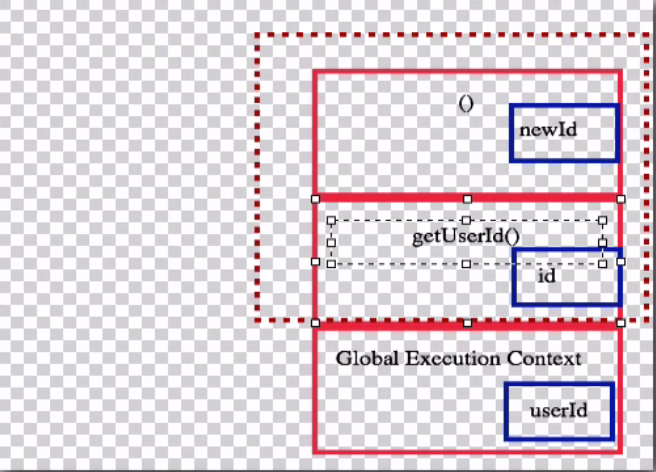
Và để cho function **hi()** có thể lấy được biến **people,** thì nhờ closure nó sẽ bao lấy thằng people và giữ lại

**debugger**

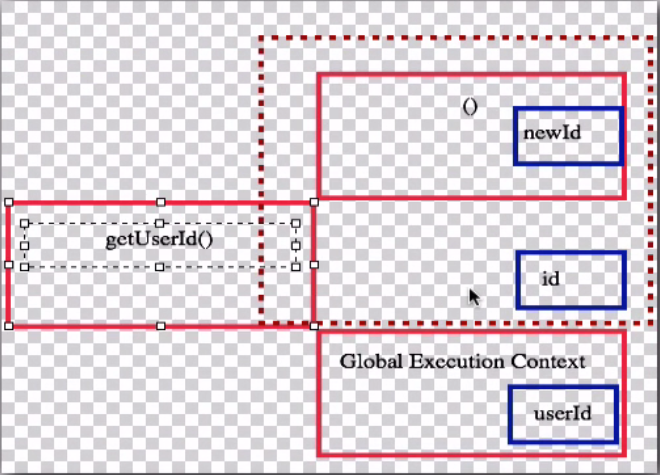
 

1. **Closure - return object:**

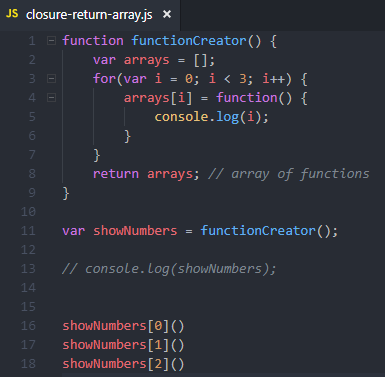
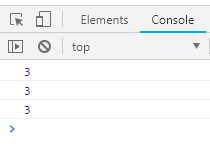
Ban đầu lúc khởi tạo EXC



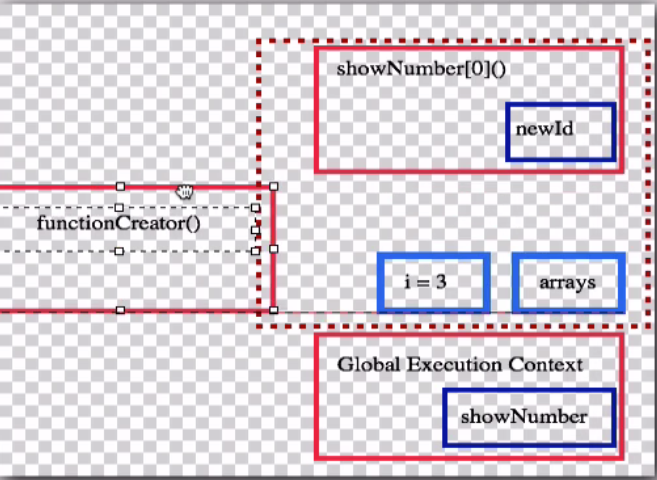
Và khi ra khỏi hàm thì nó nó sẽ remove EXC, nhưng nhờ **closure** nên sẽ giữ lại biến id



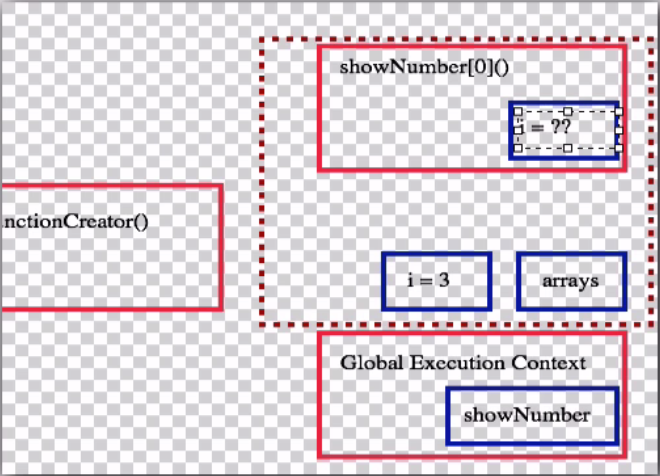
1. **Closure – return mảng:**

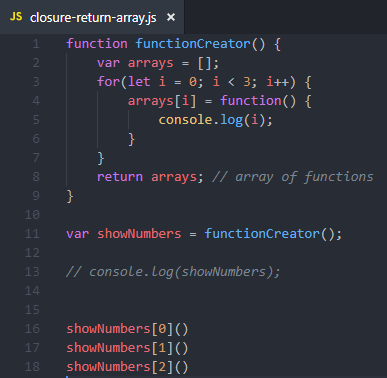
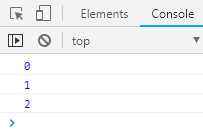
Do EXC nên khi ra khỏi hàm functionCreator() thì EXC của nó bị xoá đi và nhờ closure nên nó sẽ giữ lại biến i lúc này = 3 và arrays có 3 phần từ, mỗi phần tử là 1 function



Và khi invoke lần lượt 3 func thì nó sẽ lần lượt tạo ra EXC mới nhưng do closure nên i lúc này vẫn = 3 và nó sẽ log ra giá trị 3



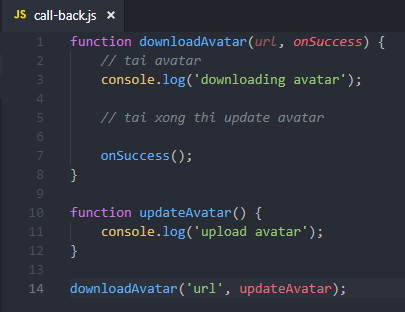
Cách khắc phục ta dùng let vì phạm vi của nó nằm trong block scope chứ không phải EXC

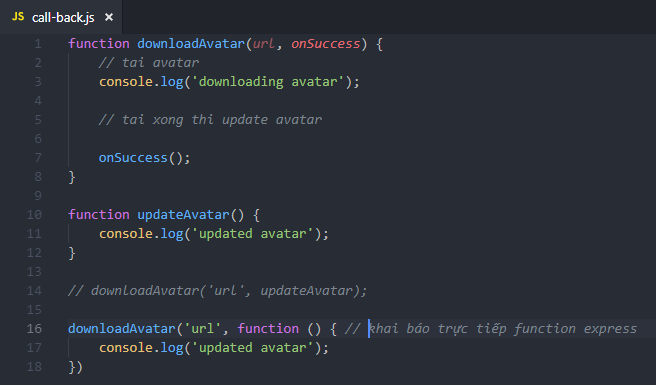
 

1. **Callback:**

Bỏ một function A vô một function B

B chạy xong nhớ chạy A (B xong nhớ gọi lại A giùm)







1. **Bind, call và apply:**

Hàm displayName xài cái this của cái được bind là object nguyen

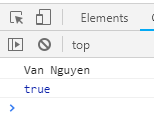


Đối với call thì nó sẽ chạy hàm luôn



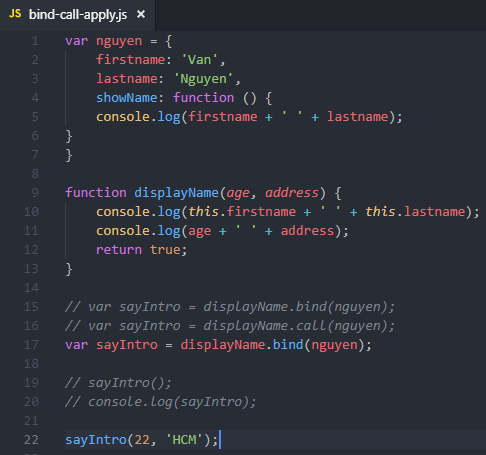
Khi đó sayIntro không phải là một function nửa mà chính là kết quả trả về của displayName



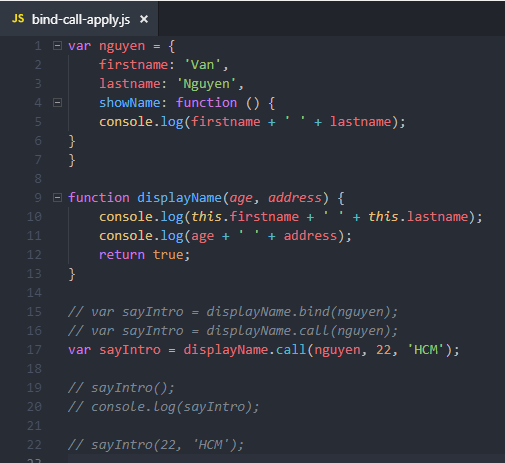


Sự khác nhau giữa bind, call và apply:

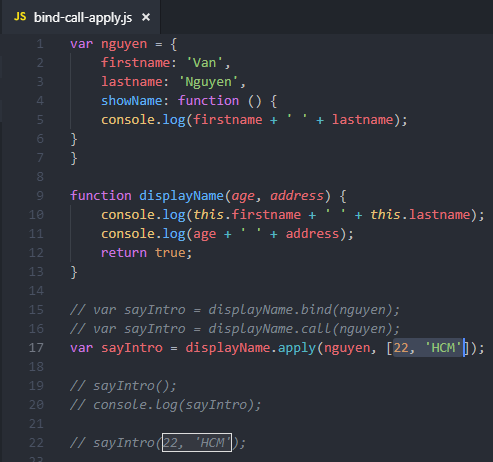
Truyền đối số của bind



Truyền đối số của call

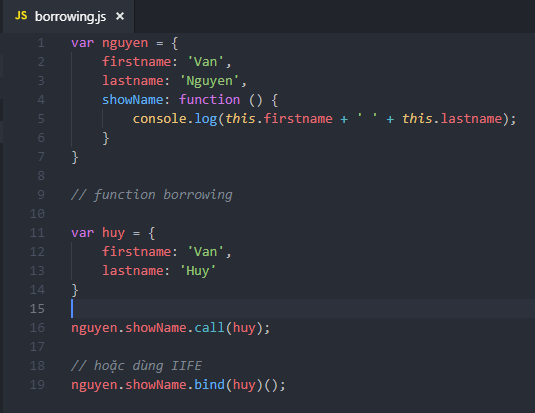


Truyền đối số của apply



1. **Function borrowing:**

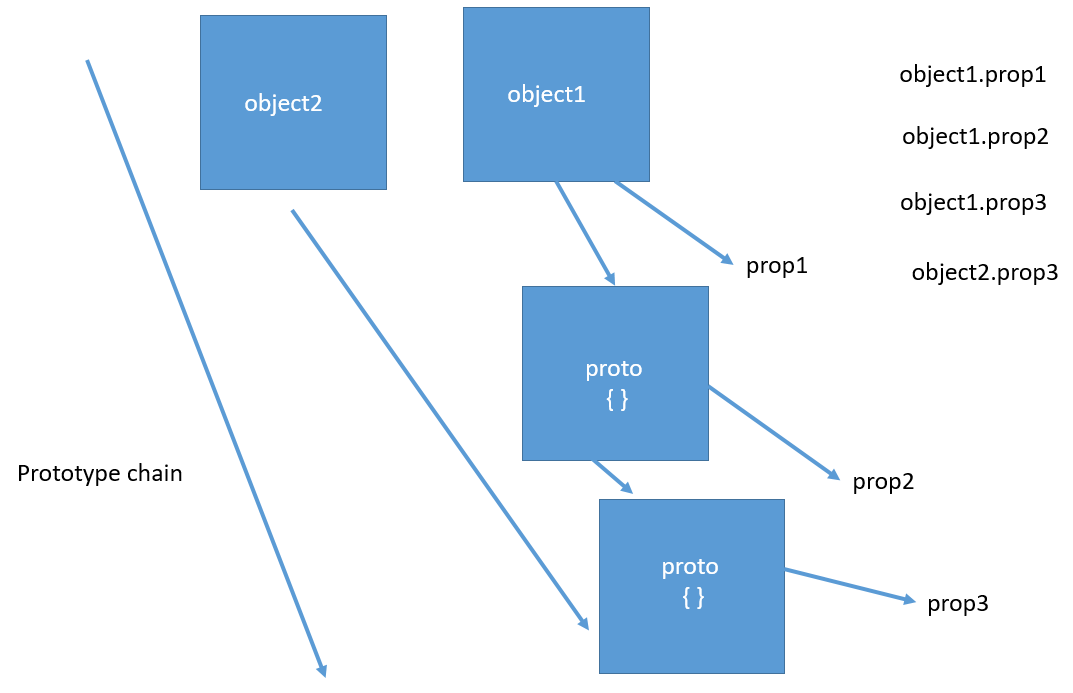
Vì muốn showName của object huy mà chưa có function showName nên ta sẽ mượn của object nguyen, sau đó dùng call hoặc bind.



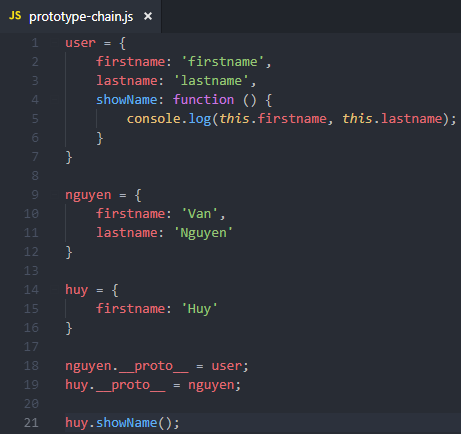
# **CHAPTER 7: HƯỚNG ĐỐI TƯỢNG TRONG JAVASCRIPT**

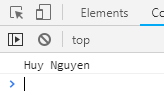
1. **OOP:**
2. **Prototype Inheritance và Prototype chain:**

Mỗi khi object được tạo ra thì có các prop ngoài ra còn có proto là 1 object và củng có prop và proto tiếp theo.  
Theo hình dưới, object1 muốn truy cập tới prop3 thì nó sẽ kiếm trong object1 trước không có thì nó sẽ xuống proto tiếp theo để tìm, nhưng nếu tìm prop4 và tìm hết proto không có sẽ undefined và cái chiều kiếm từ trên xuống dưới gọi là Prototype Chain



**VD:**

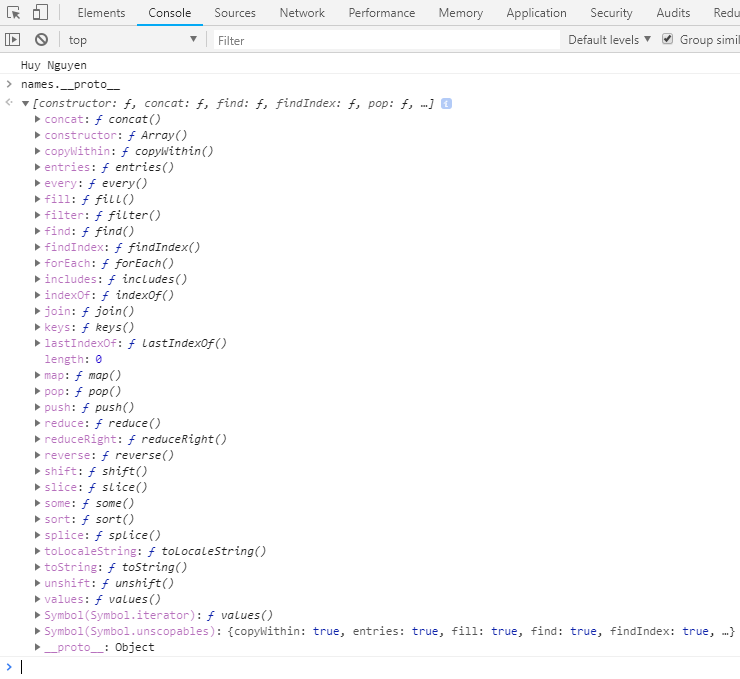




**VD: hiểu rõ hơn về \_\_proto\_\_**



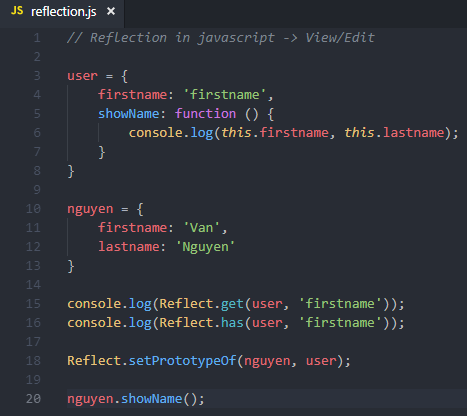
Truy cập vào \_\_proto\_\_ của biến names:

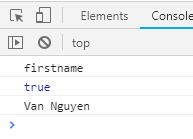


1. **Reflect:**

Reflect có thể view, edit và thay đổi \_\_proto\_\_ của object,…..

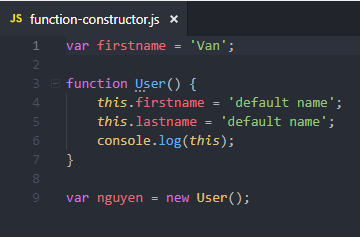
VD: Thay vì trỏ tới \_\_proto\_\_ để thay đổi như ở vd trước thì ta dùng reflect, có thể view các prop của object user,…  
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect>



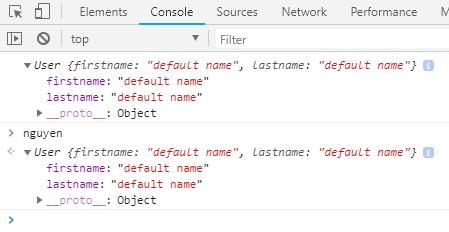


1. **Function constructor và từ khoá new:**

Khi gặp từ khoá new thì function User đã tạo ra một vùng bộ nhớ mới và đổi từ khoá this trong function đó gán với object đó và không liên quan gì đến global nửa



Khi đó this.firstname chính là nó đang thêm thuộc tính firstname vào object this này và nó gắn với User chứ không liên quan tới global nửa.

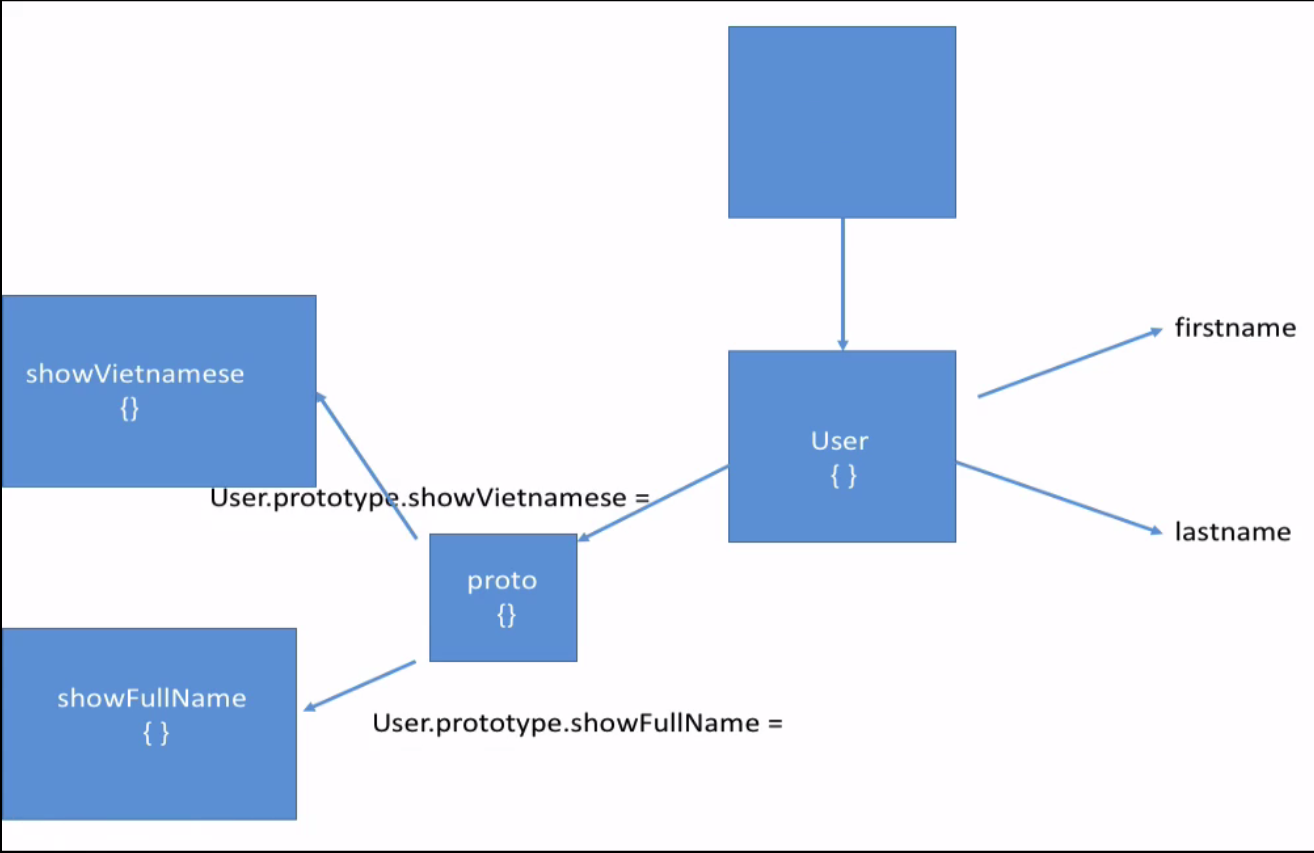


Function constructor có tham số, dùng từ khoá new để khởi tạo object



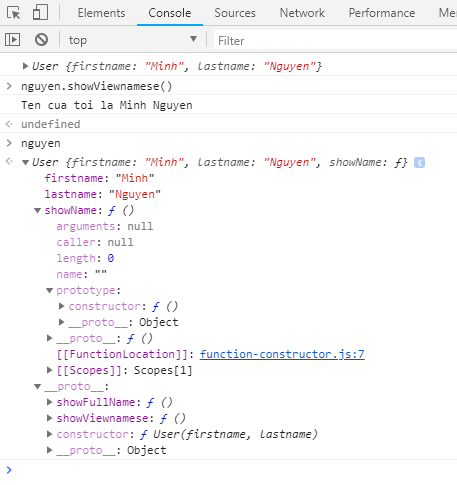


1. **Sử dụng .prototype:**

****

nguyen





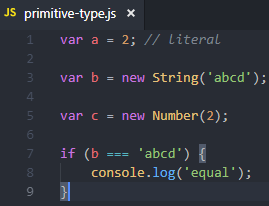
Như sơ đồ trên, object **nguyen** được tạo bởi function constructor của **User** trong đó có sẵn thuộc tính firstname và lastname, sau khi tạo xong thì nó tự động có proto có function constructor đó, sau này khi ta thêm tính năng thì nó sẽ thêm vào proto bằng từ khoá .prototype (khác với \_\_proto\_\_ )

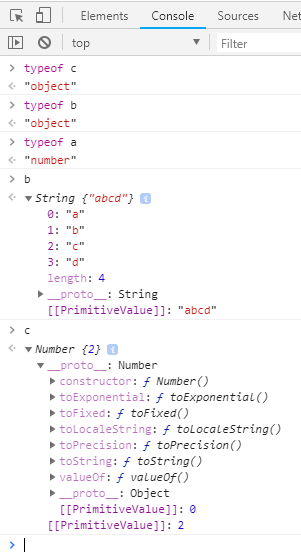
- Ưu điểm:

Khi ta khai báo một function trong object, thực chất function củng là object nên nó sẽ tốn bộ nhớ để lưu trữ, nhưng khi dùng .prototype để tạo phương thức từ bên ngoài khi cần thì mới xài và chỉ cần trỏ tới proto thì có thể gọi được thì nó sẽ ít tốn bộ nhớ hơn. Vì lúc khởi tạo nó không cần phải tốn bộ nhớ để lưu trữ thêm 1 function.

1. **Tạo primitive type từ function constructor:**

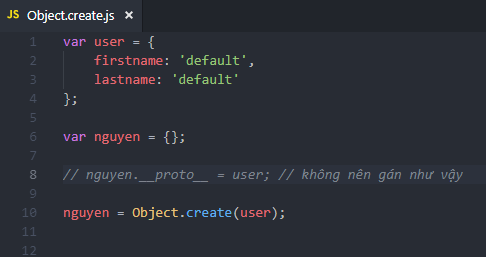
Vì b được tạo theo primitive type nên không thể so sánh theo kiểu string được

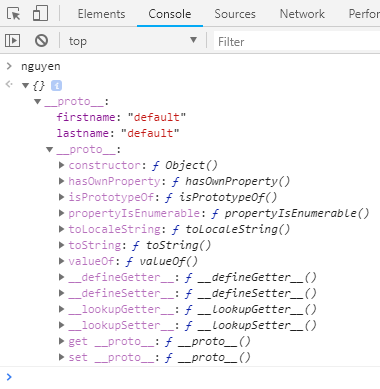




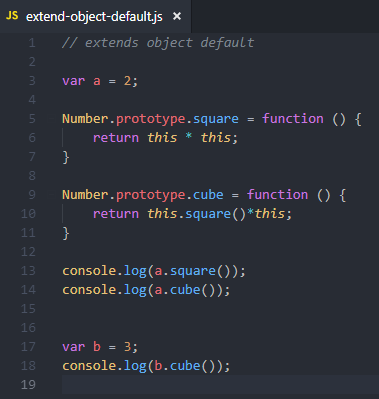
1. **Tạo đối tượng với Object.create:**

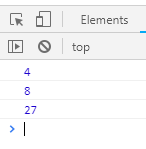
Không nên gán trực tiếp \_\_proto\_\_ mà phải sử dụng Object.create mặc dù nó cho kết quả như nhau





1. **Extend Object Default:**

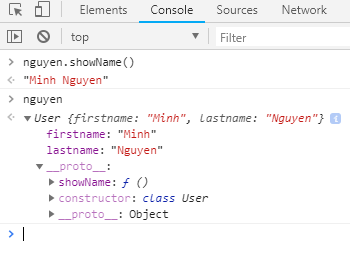




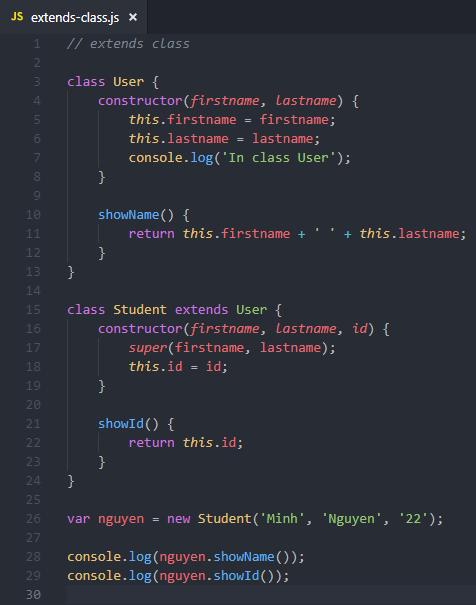
1. **Tạo object với class trong ES6:**

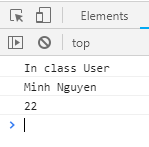
Ta củng có thể sử dụng prototype để thêm function cho class giống như function constructor, thực chất class trong javascript củng là một object





1. **Extends class:**

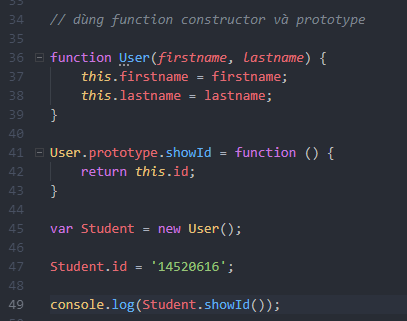




Thay vì extends như trên ta có thể dùng Function constructor và prototype để tạo object Student. Làm theo kiểu class như trên thì người ta gọi là **Syntactic Sugar**

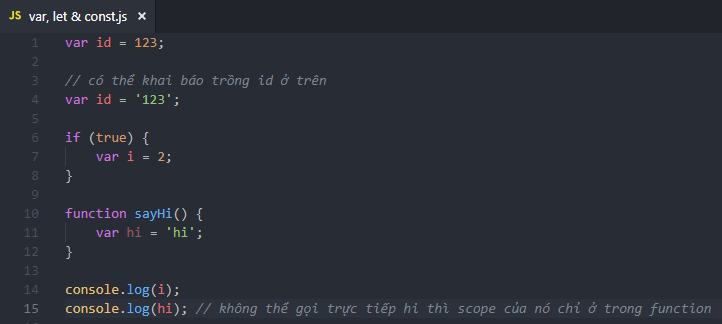
**- Syntactic sugar** is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

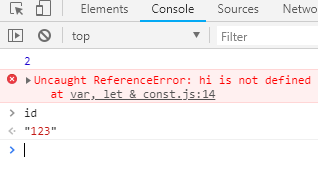
- **Syntactic Sugar** là cú pháp trong một ngôn ngữ lập trình được thiết kế để làm cho mọi thứ dễ đọc hay thể hiện hơn. Nó khiến ngôn ngữ "ngọt ngào hơn" cho con người sử dụng: mọi thứ có thể được thể hiện rõ ràng hơn, chính xác hơn, hay theo một cách khác mà một số người khác thích.



# **CHAPTER 7: ES6**

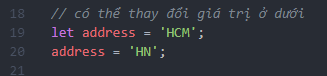
1. **ES6:**
2. **Var, let và const:**

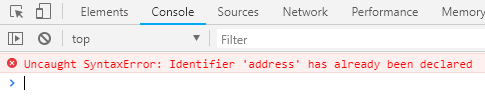




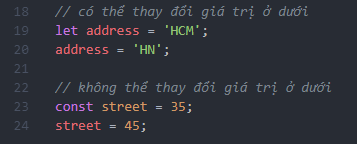
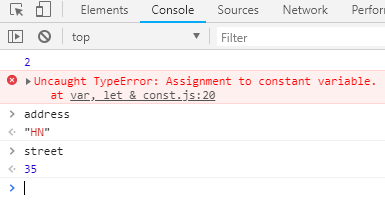
Ta có thể khai báo trồng biến var, khi var nằm trong function thì scope của nó chỉ ở function đó nên bên ngoài global ko gọi trực tiếp được

Còn với let thì không thể khai báo trồng được, tương tự const củng vậy

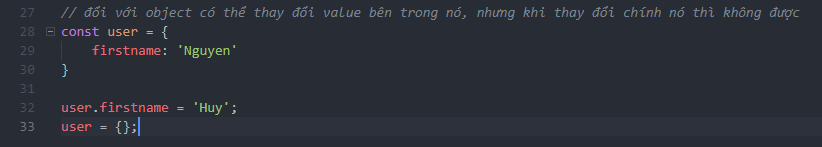
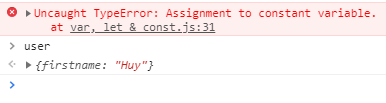




Nhưng với let ta có thể thay đổi giá trị ở dưới, còn const thì không được

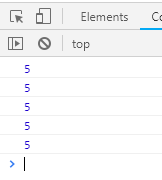
 

Đối với const object ta có thể thay đổi value bên trong nó, nhưng khi thay đổi cả object thì không được

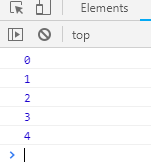
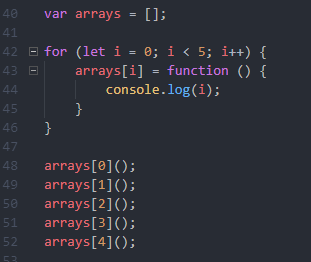
 

1. **Khi nào thì sử dụng var, let & const:**

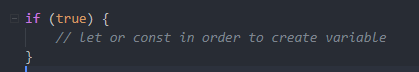
Dùng var ở vòng lặp for thì nó sẽ lấy giá trị cuối cùng của biến i



Khắc phục ta dùng let

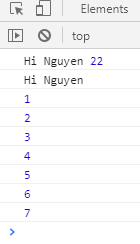


Còn trong block if thì nên dùng let hoặc const để khai báo biến



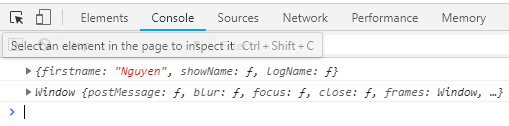
1. **Arrow function:**

Khi chỉ có 1 params và body chỉ có 1 dòng thì có thể bỏ () và {}

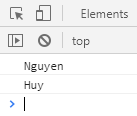


* **This trong arrow function:**

This trong function expression ám chỉ object cha là user, còn this trong arrow function thì cùng bậc với object cha là global object



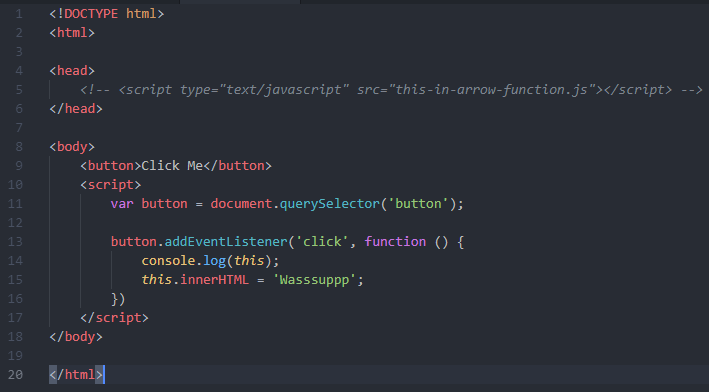
**VD:**

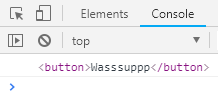


**VD:** Trong HTML

Khi dùng function expression

This lúc này đang ám chỉ là object cha là button, nên khi bấm vào `Click Me` nó sẽ đổi text thành wasssuppp

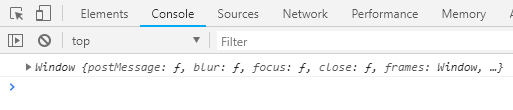


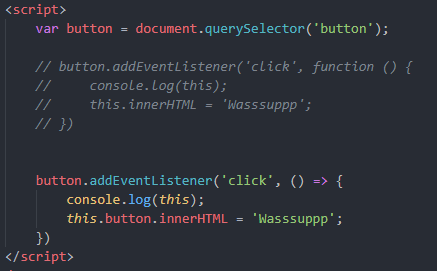
Khi dùng arrow function

Bấm `Click Me` nó sẽ không đổi text của button vì this hiện tại đang là global nằm cùng bậc với button.addEventListener(…)





Sửa lại thành this.button thì text mới đổi được

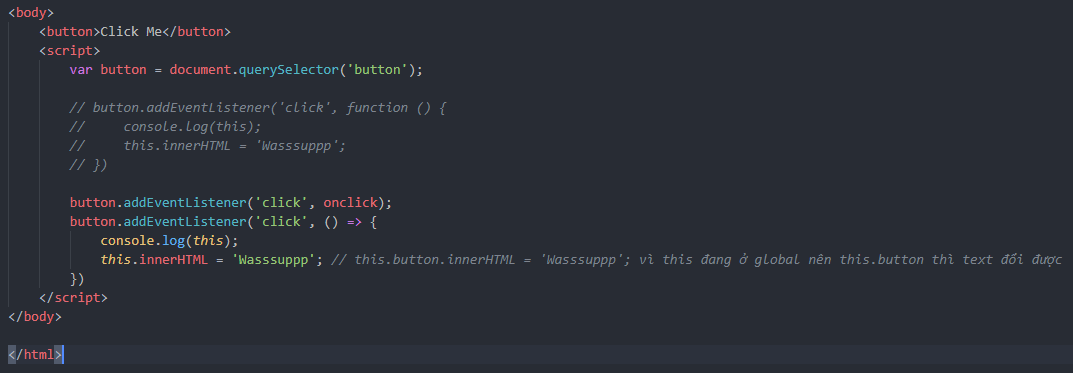


* Một số trường hợp nên tránh dùng arrow function:
* Trong object:



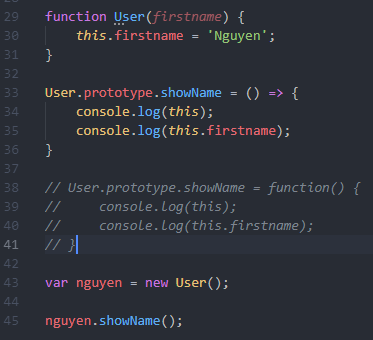
* Trong HTML:

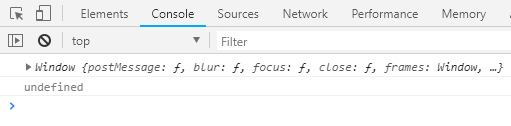
Trường hợp trong hàm callback ta có sử dụng lại object cha không, ở đây là button nếu có (chẳng hạn như thay đổi giá trị button) thì nên xài function thường, còn không thì dùng arrow function cho gọn.



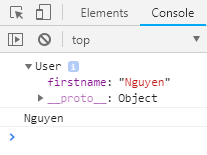
* Trong prototype:

Nếu dùng arrow function thì this đang ở global, nên nó sẽ không lấy được firstname của object User



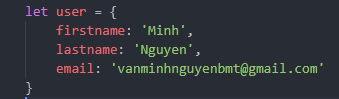


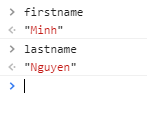
Nếu dùng function thường thì ok  

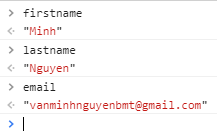
1. **Destructuring & Spread Operator:**
2. **Destructuring với object:**

Cho object user



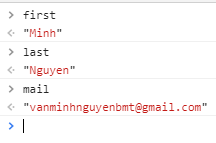
Cách thông thường để lấy giá trị trong object là 

Với destructuring:



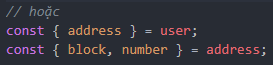
Đặt lại tên biến trong destructuring:





Object lồng trong object:

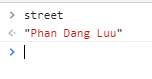




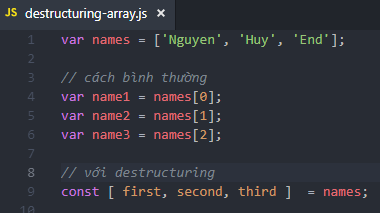
Giá trị mặc định trong destructuring:



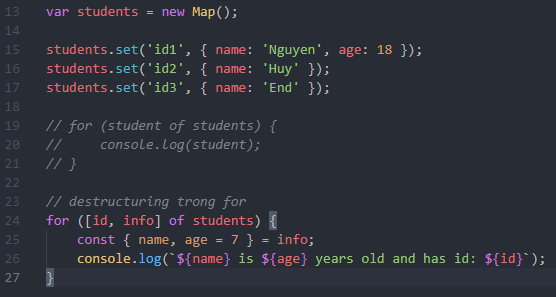
Thêm street vào object address và gán giá trị ban đầu là ….



1. **Destructuring với array:**

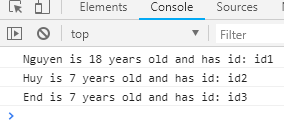


Destructuring trong for kết hợp với default value destructuring



Với cách bình thường:

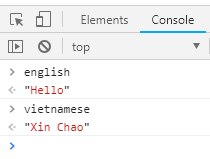




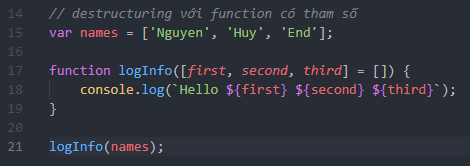
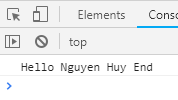
1. **Destructuring Function:**

VD: Function không có tham số

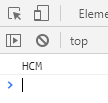




VD: Function có tham số là array

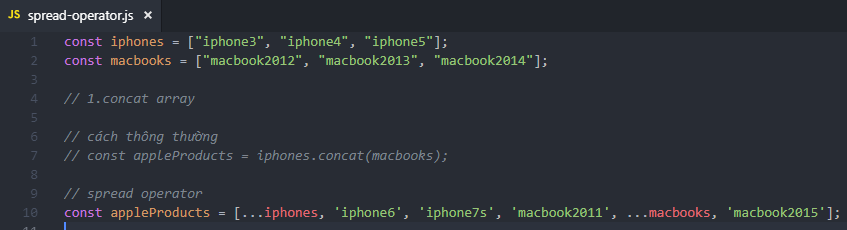
VD: Function có tham số là object

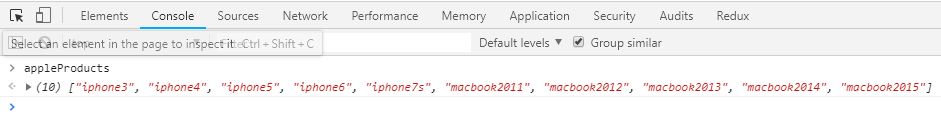
 

1. **Spread Operator:**

* Nối mảng: Trường hợp dùng concat khi ta muốn thêm giá trị vào đầu hoặc cuối mảng thì phải push trước rồi mới concat làm thế nó không flexible.

Nhưng với operator thì mọi chuyện trở nên đở lằng nhằng hơn.



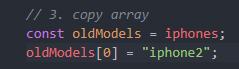
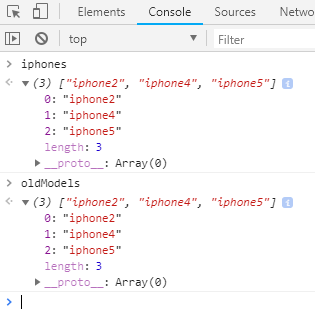


* Spread a string:

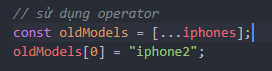
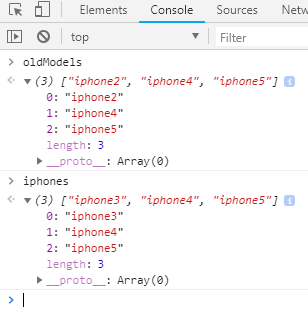
 

* Coppy array:

Trường hợp gán bằng như này nó sẽ vừa gán giá trị và cùng dùng chung địa chỉ ô nhớ luôn, nên khi oldModels thay đổi giá trị thì iphones củng bị thay đổi.

Sử dụng operator

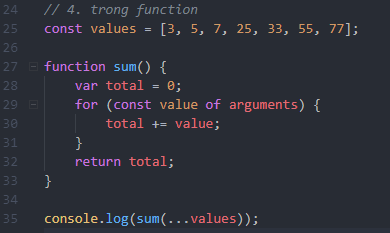
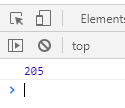
 

Lúc này chỉ có giá trị của oldModels thay đổi, iphones vẫn giữ nguyên.

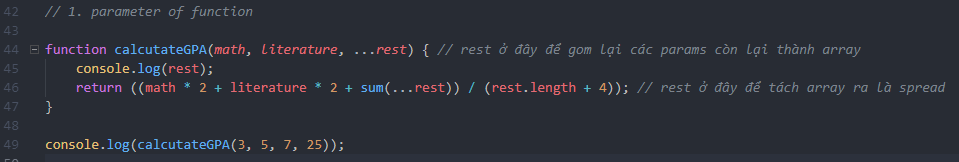
1. **Spread operator và rest operator khác nhau thế nào:**

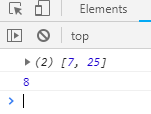
Spread dùng để tách array hoặc chuỗi string ra còn rest gom nhóm những cái đó lại thành array

* Dùng spread operator trong function

Đối với function có tham số thì …rest ở param để gom các params lại thành array (Rest Operator), còn rest ở dưới để tách array ra (Spread Operator)





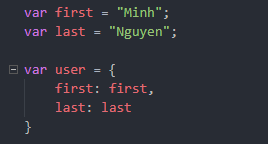
* Dùng …rest trong destructuring



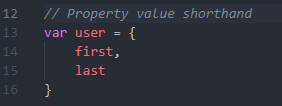
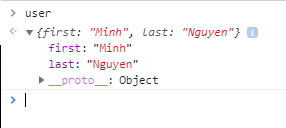


1. **Object Literals Upgrade:**

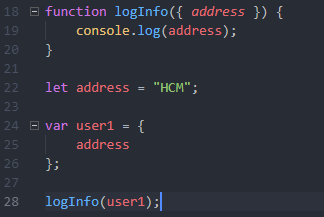
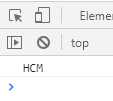
* **Property value shorthand:**



Sử dụng property value shorthand

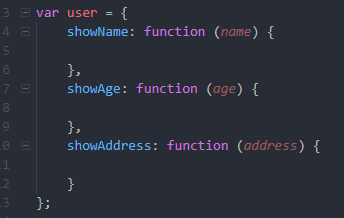
 

Sử dụng trong param function là object:

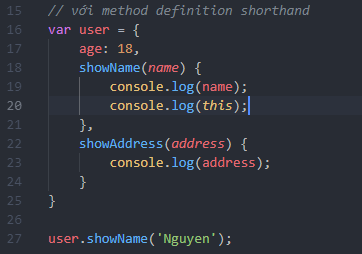
 

* **Method definition shorthand:**

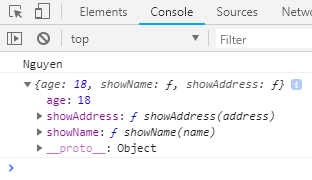
Cách cũ:



Với Method definition shorthand:

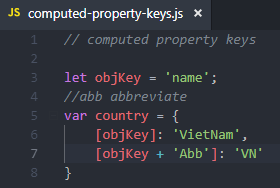
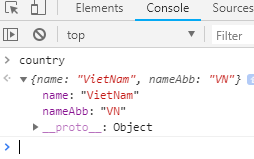


Và this trong function showName ám chỉ object user chứ không phải global như trong arrow function

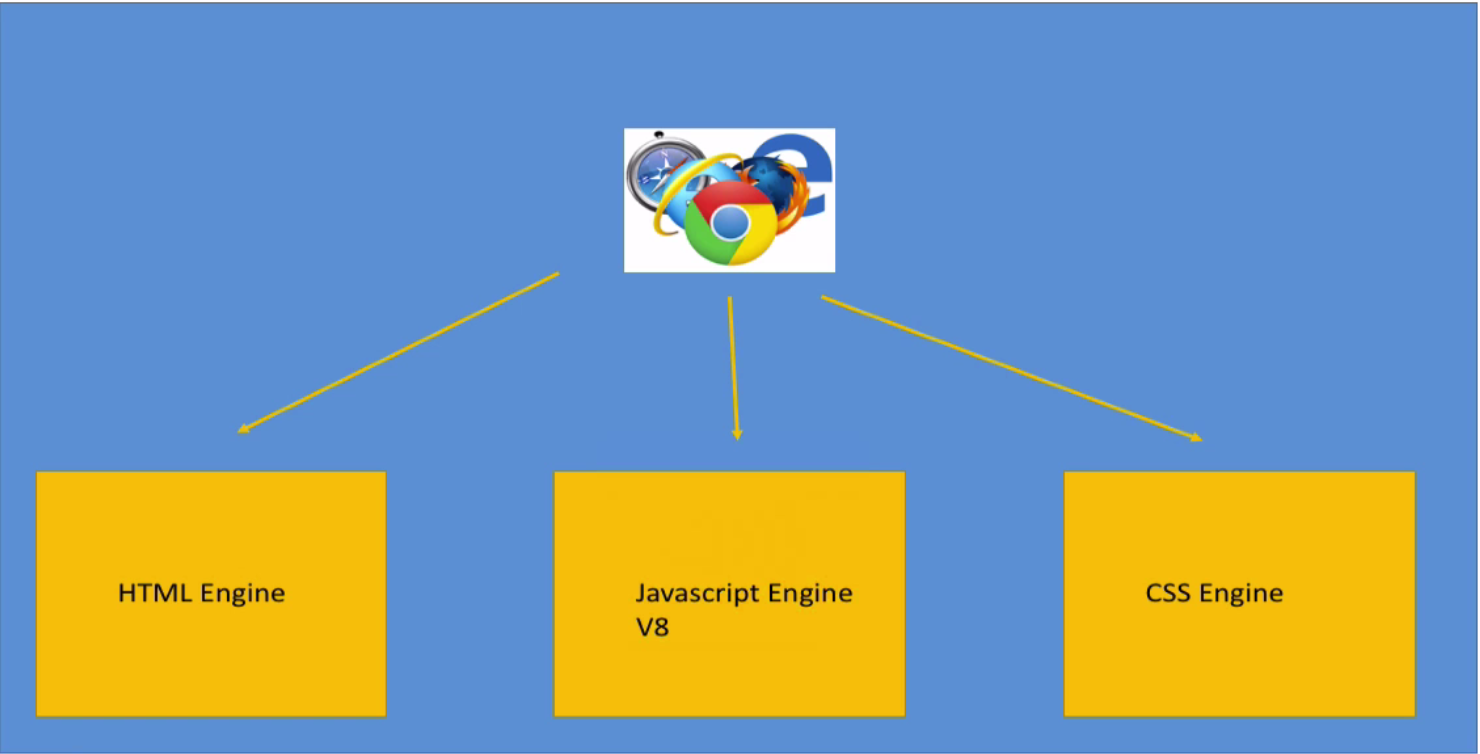


* **Computed property keys:**

Có thể dùng biến làm key cho object, có thể cộng chuỗi,….

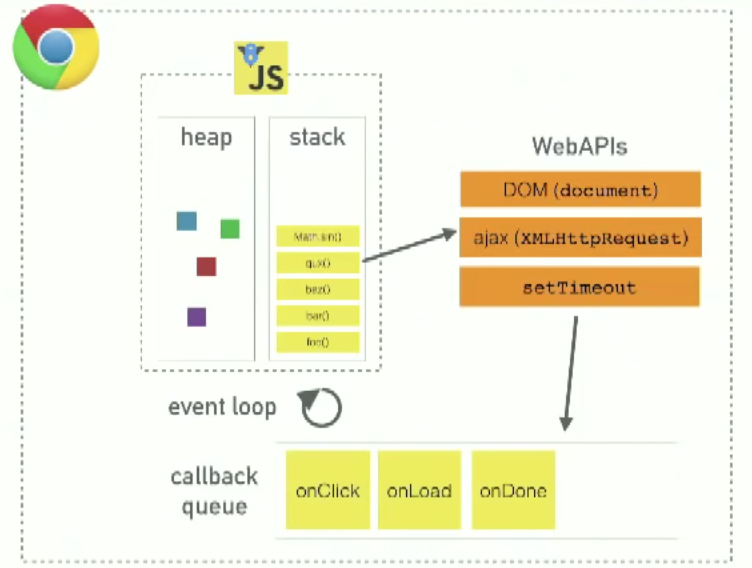
1. **Concurrency model and Event Loop:**
2. **Javascript Engine là gì ? V8 là sao ?**



Browser khi hoạt động thì có 3 engine chủ yếu là:

* HTML Engine: Dịch file HTML ra mã máy
* CSS Engine: Dịch file CSS ra mã máy
* Javascript Engine: Dịch file js ra mã máy, engine phổ biến là V8 được sử dụng trong google chrome là mã nguồn mở được viết chủ yếu = C++

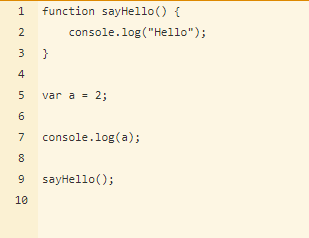
1. **Synchronous và Asynchronous, callback queue và event loop:**



Trong javascript engine V8 có heap và stack, heap là nơi cấp phát và giải phóng bộ nhớ, stack thì gần giống như execution stack. Ngoài ra còn có các thành phần như WebAPIs của browser và event loop và callback queue

* Synchronous:

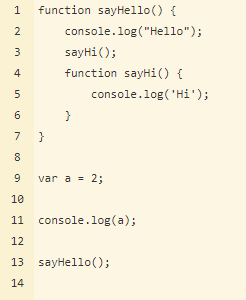
Website test: <http://latentflip.com/loupe>



console.log(a) sẽ chạy thêm vào call stack và chạy trước, sau khi chạy xong thì nó sẽ xoá ra khỏi stack và chạy hàm sayHello() và thực thi nội dung trong hàm và chạy như vậy gọi là Synchronous

VD khác:



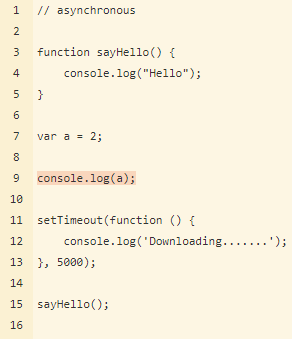
 



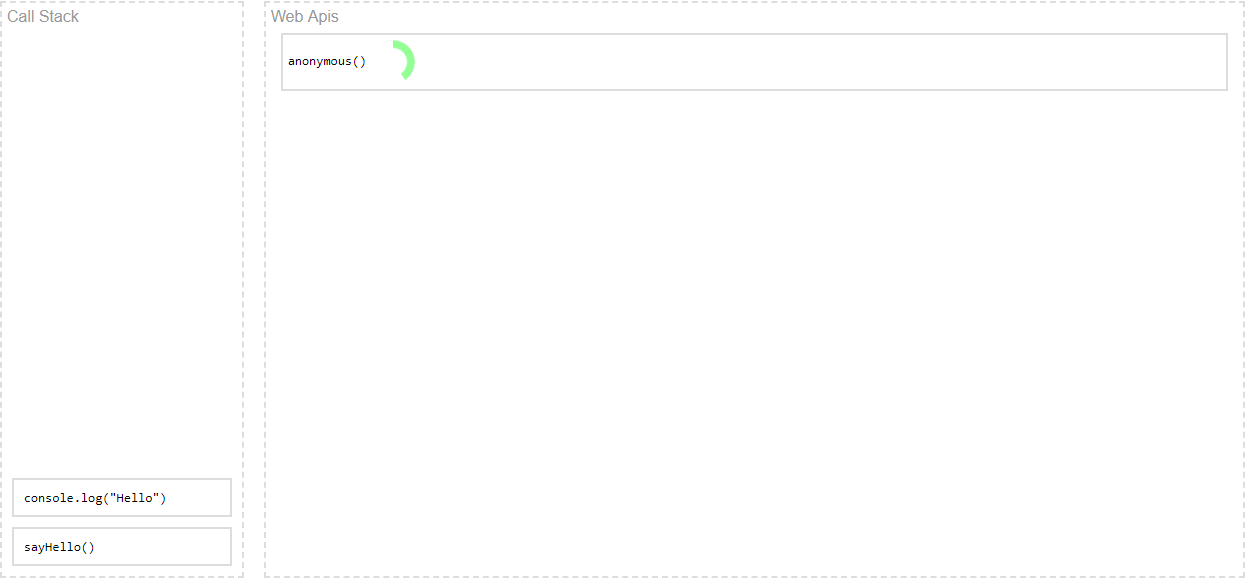
* Asynchronous:

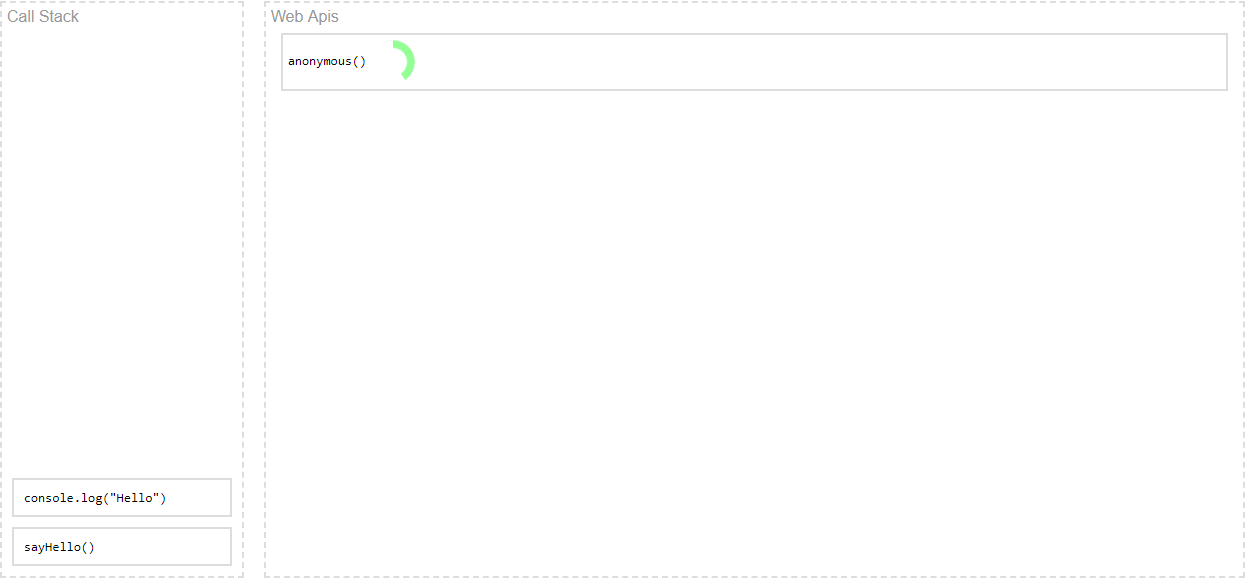
Ta có 1 ví dụ về hàm setTimeout để xem tiến trình chạy bất đồng bộ như thế nào.

Đầu tiên console.log(a) sẽ được đưa vào call stack và chạy đầu tiên, nhưng khi đến setTimeout nó chưa vội log ra liền mà sẽ bỏ vào trong Web apis và có 1 khoảng thời gian chờ và sau đó đưa vào call back queue và sau đó mới đưa lên call stack để gọi nhưng trong khi nó đang làm vậy thì thằng sayHello() đã được đẩy vào call stack và chạy rồi. Vì vậy, tiến trình đó được gọi là asynchronous hay còn gọi là bất đồng bộ









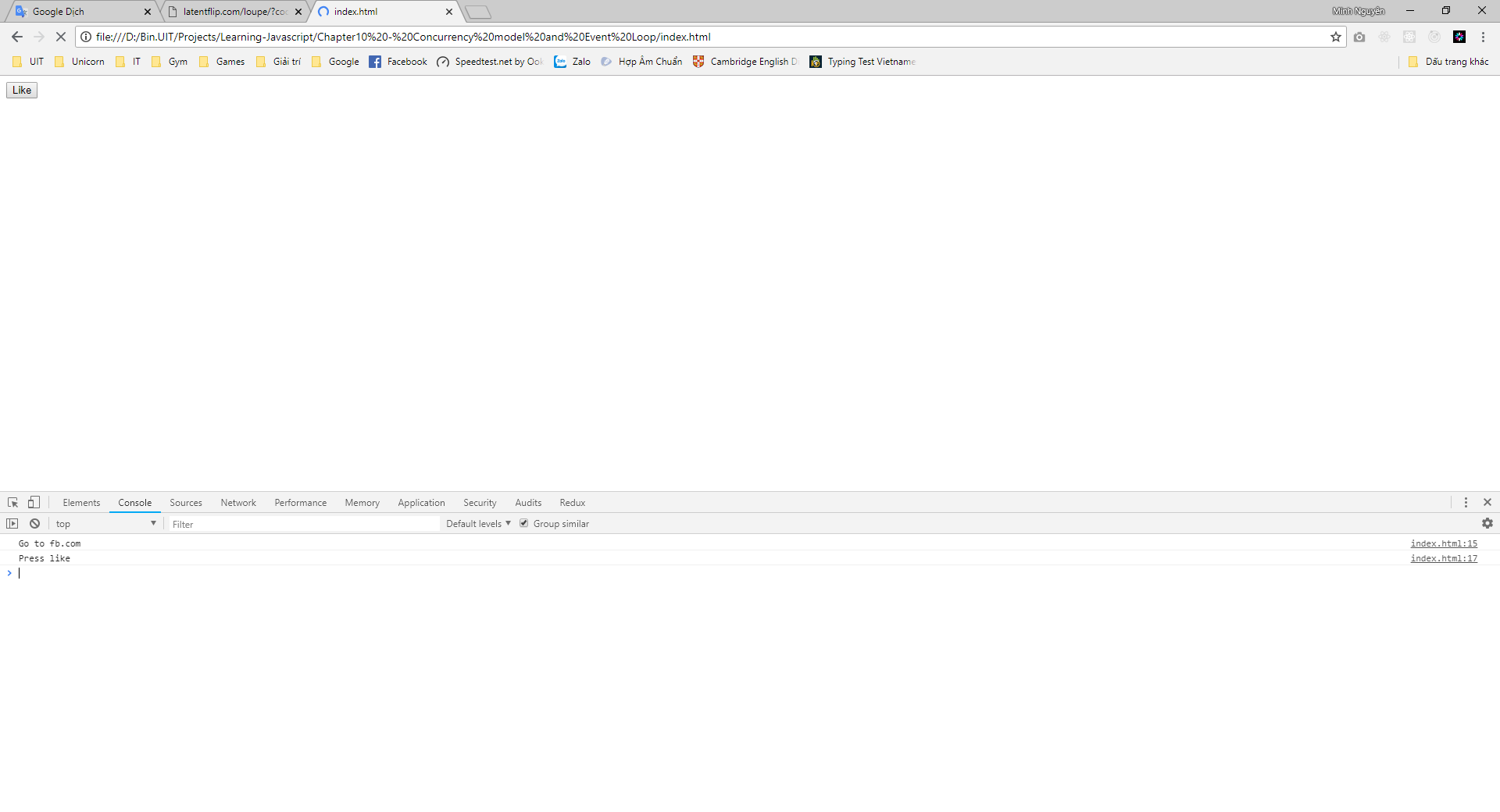


1. **Blocking và non-blocking:**

Vì chạy synchronous nên phải đợi while chạy xong thì mới chạy tiếp ở dưới được, và khi while chạy quá lâu mà ta phải đợi nó chạy xong mới làm tiếp được ở đây là thực hiện click Like. Vì vậy gây ra tình trạng đứng máy hay bị blocking, đây là điểm bất lợi của synchronous

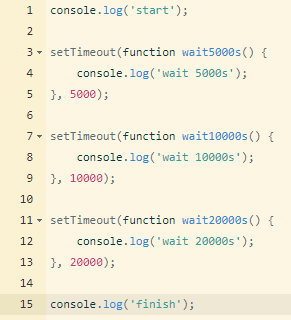
Để khắc phục tình trạng trên thì những công việc mà tốn nhiều thời gian ta đưa vào asynchronous chạy bất đồng bộ hay còn gọi là non-blocking

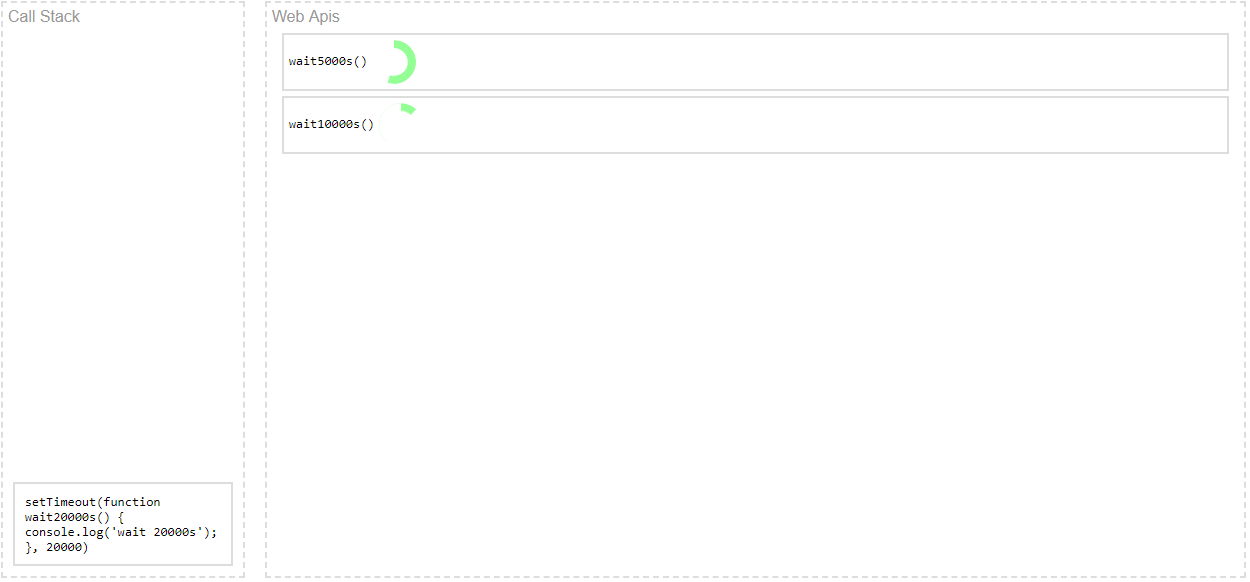


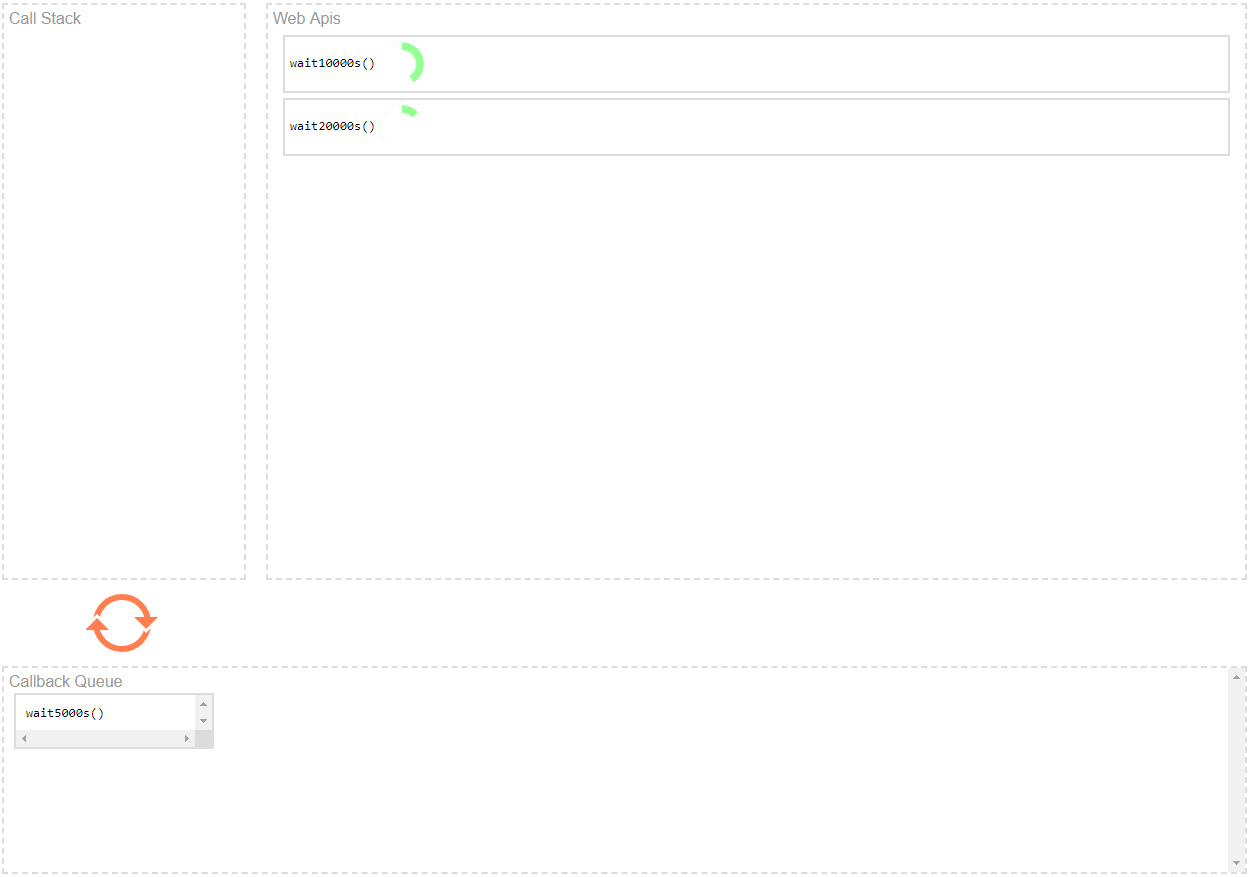


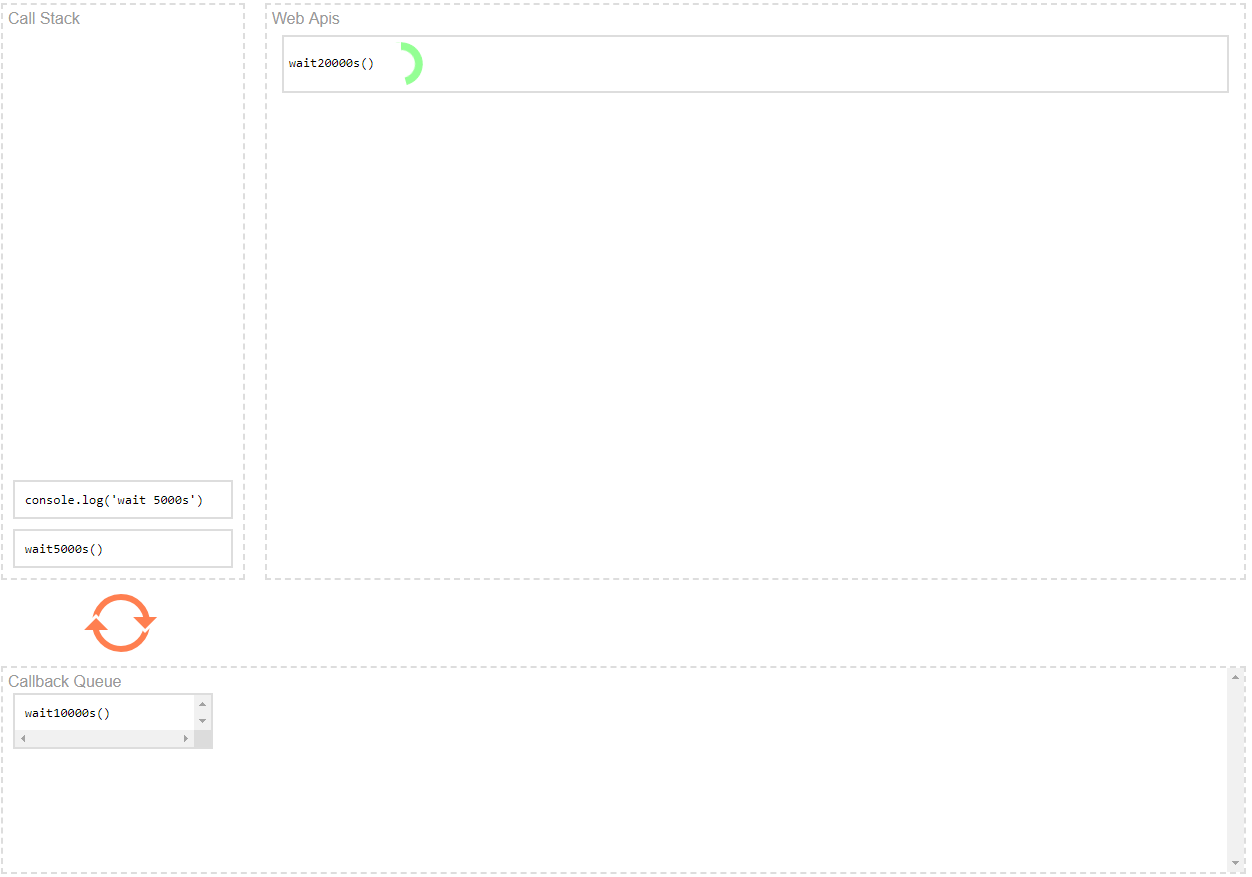
Ví dụ: Non-blocking

Ở đây thằng nào được đưa vào call stack trước thì nó sẽ được đẩy qua Web apis để chạy ngầm, nhưng khi qua Web apis thì thằng nào thực hiện xong trước sẽ được đẩy xuống Callback queue và sau đó được đẩy lên tiếp call stack để chạy. Khi thực hiện tiến trình như vậy thì gọi là non-blocking hay asynchronous.





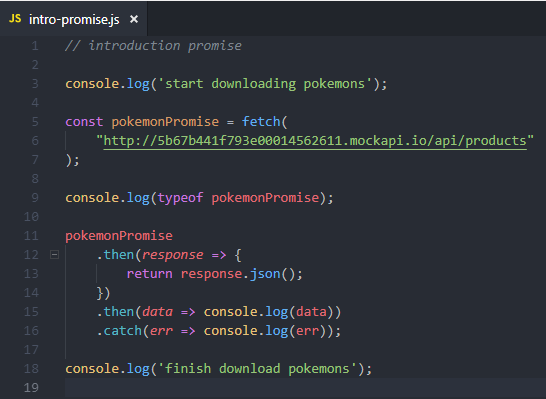


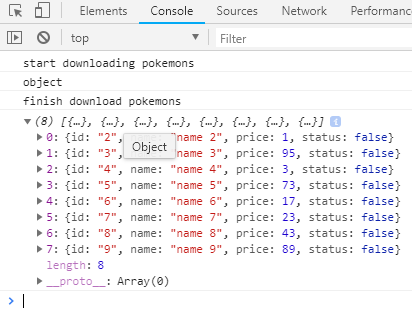


1. **Promise:**

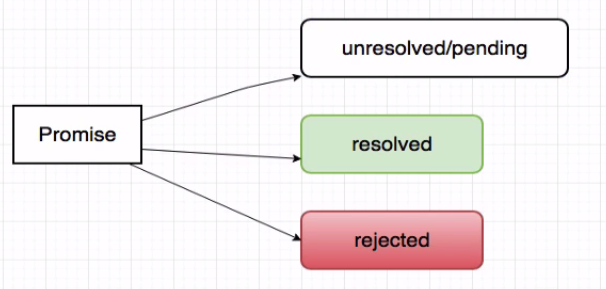
Là một object đại diện cho sự thành công hoặc thất bại của một tác vụ bất đồng bộ

1. **Basic:**

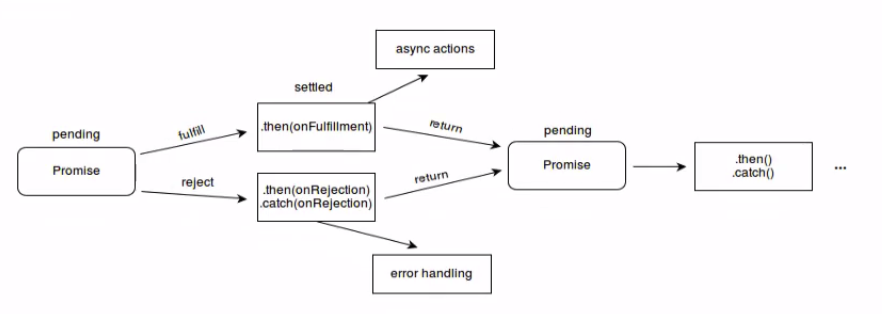


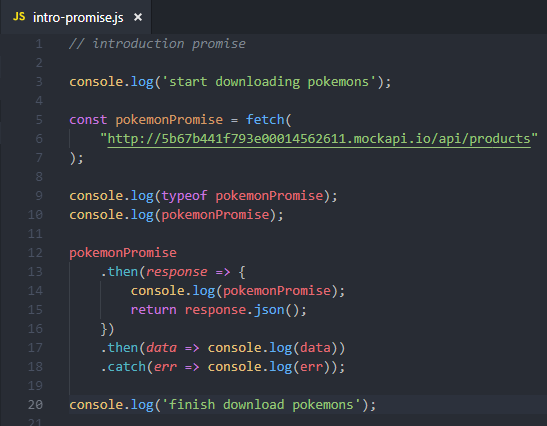


1. **3 States of Promise:**

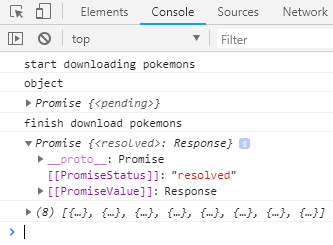


**/ fulfill**

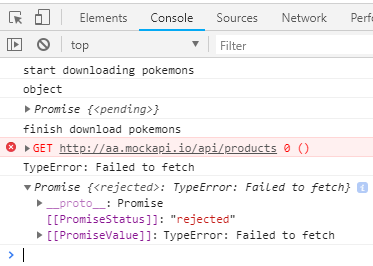




Ban đầu khi chưa vào .then() thì PromiseStatus là “pending”. Nhưng khi fetch dữ liệu thành công thì PromiseStatus đổi thành “resolved”



Và khi có vấn đề lỗi xảy ra nó sẽ vào .catch và PromiseStatus đổi thành “rejected”

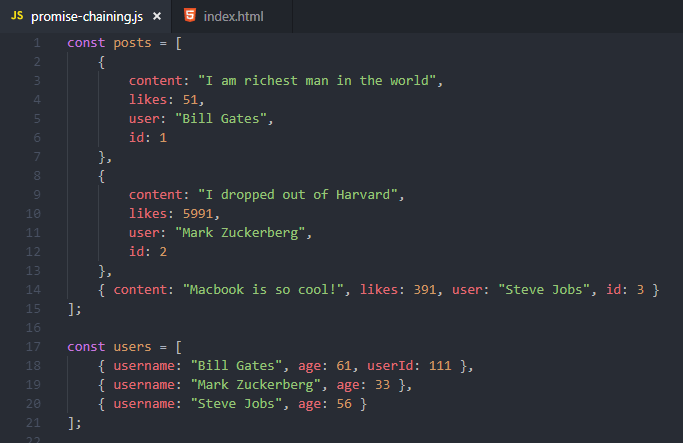


1. **Tự tạo promise:**



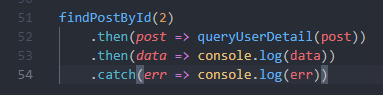
1. **Promise Chaining:**

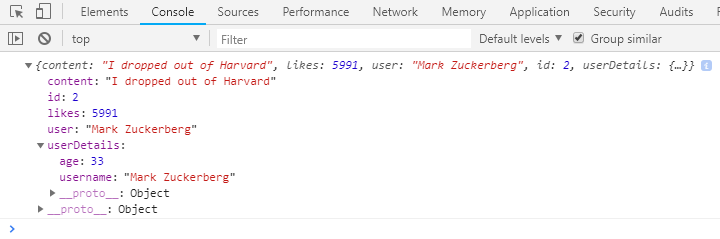
Có 2 object:



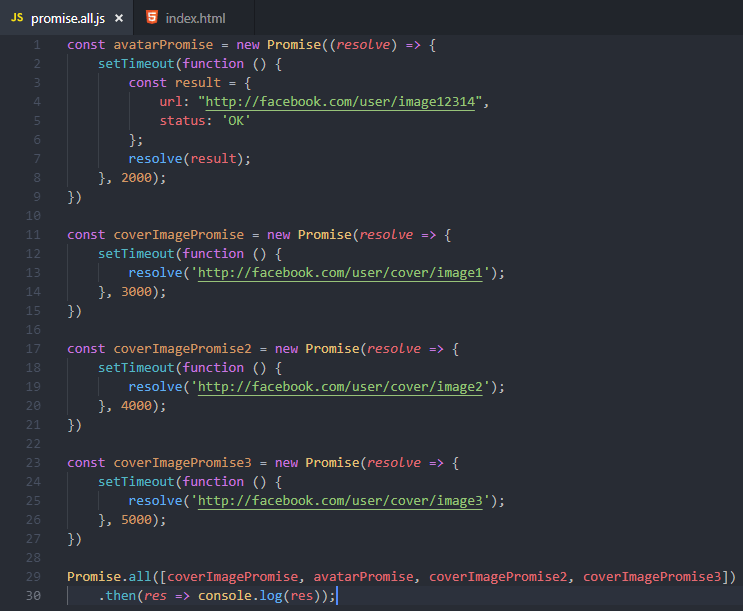
2 Custom promise:





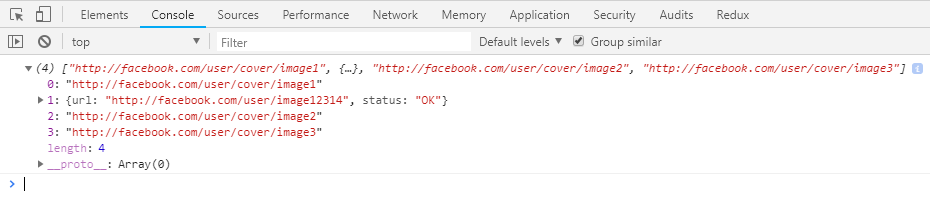


1. **Promise.all:**

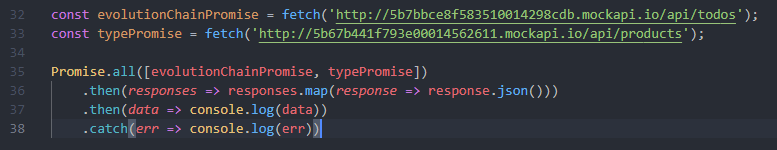


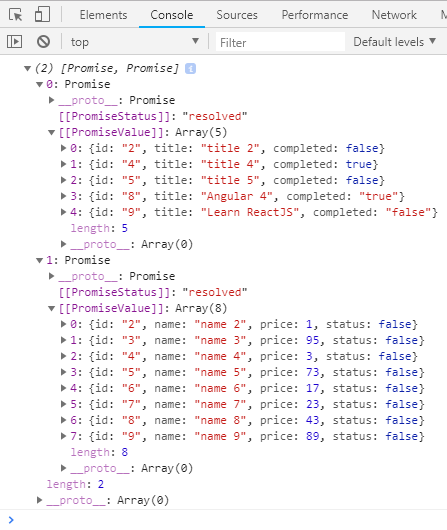
Đoạn code như hình trên, thì promise.all sẽ chạy hết tất cả promise và nó sẽ trả về response khi đã thực hiện xong promise có thời gian dài nhất.

Và kết quả là nó trả về theo thứ tự promise chúng ta đặt ở array.



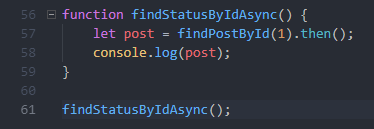
VD: Call API



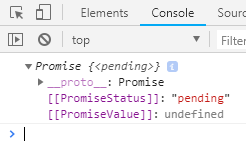


1. **Async await:**
2. const posts = [
3. {
4. content: "I am richest man in the world",
5. likes: 51,
6. user: "Bill Gates",
7. id: 1
8. },
9. {
10. content: "I dropped out of Harvard",
11. likes: 5991,
12. user: "Mark Zuckerberg",
13. id: 2
14. },
15. { content: "Macbook is so cool!", likes: 391, user: "Steve Jobs", id: 3 }
16. ];
17. const users = [
18. { username: "Bill Gates", age: 61, userId: 111 },
19. { username: "Mark Zuckerberg", age: 33 },
20. { username: "Steve Jobs", age: 56 }
21. ];
22. function findPostById(*id*) {
23. const post = posts.find(*post* => post.id === id);
24. return new Promise((*resolve*, *reject*) => {
25. setTimeout(function () {
26. if (post) {
27. resolve(post);
28. } else {
29. reject('can not find post with id: ' + id);
30. }
31. }, 2000);
32. });
33. }
34. function queryUserDetail(*post*) {
35. const user = users.find(*user* => user.username === post.user);
36. return new Promise((*resolve*, *reject*) => {
37. setTimeout(function () {
38. if (user) {
39. post.userDetails = user;
40. resolve(post);
41. } else {
42. reject('can not find user of posts: ' + post);
43. }
44. }, 2000);
45. });
46. }
47. *// findPostById(2)*
48. *// .then(post => queryUserDetail(post))*
49. *// .then(data => console.log(data))*
50. *// .catch(err => console.log(err))*

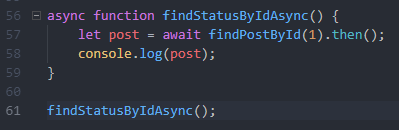
Ta có function findStatusByIdAsync khi chưa gắn async await:



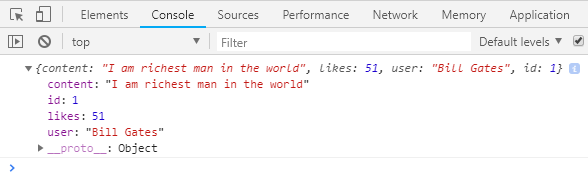
Kết quả, PromiseStatus vẫn là pending và giá trị trả về vẫn chưa lấy được vì khi hàm findPostById chưa chạy xong thì nó đã chạy xuống dưới log ra.



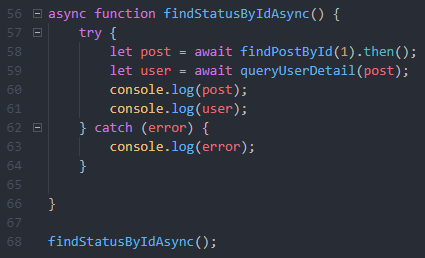
Để khắc phục thì ES7 sinh ra async, await thay vì thao tác với .then() nhiều lần thì dùng async await ngắn gọn hơn.

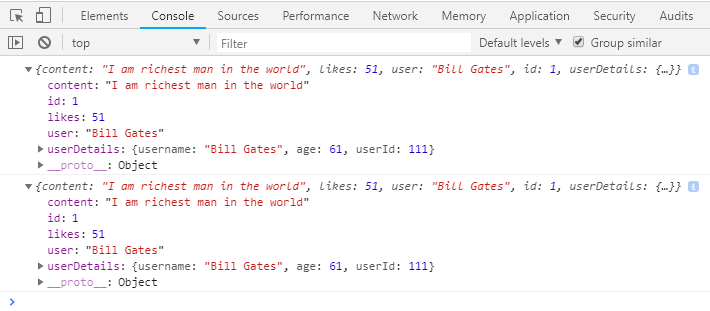


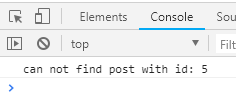
Kết quả, là nó sẽ đợi findPostById lấy dữ liệu xong mới chạy xuống và log ra.



Sử dụng nhiều await thay vì promise chaining:



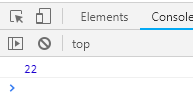
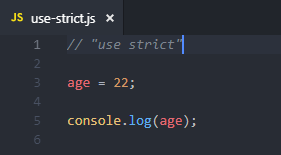


Trường hợp lỗi

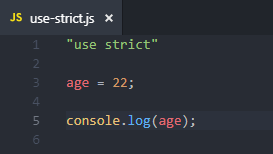
1. **Lưu ý:**
2. **Use strict:**

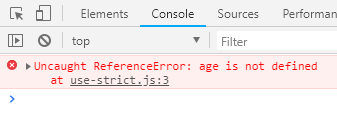
**VD1: khai báo biến without var**

Không sử dụng “use strict”

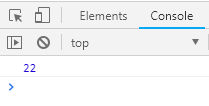
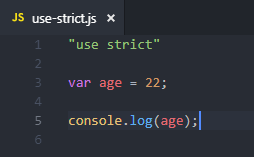


Khi sử dụng “use strict”, nó sẽ không cho phép khai báo biến khi chưa khai báo kiểu cho biến

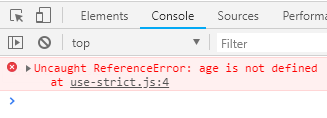
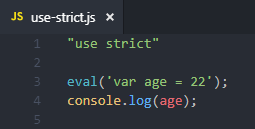




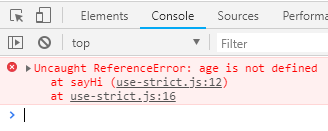
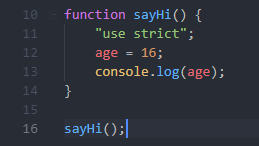
Để khắc phục lỗi này ta phải khai báo var, let hoặc const cho biến age



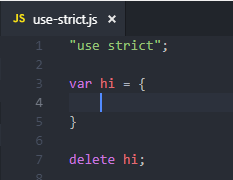
**VD2: sử dụng eval trong strict mode**

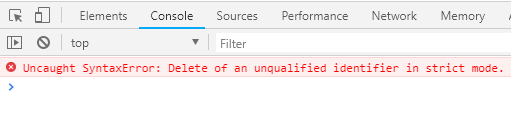


**VD3: Sử dụng trong function**



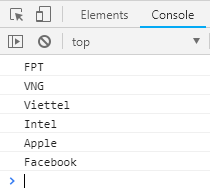
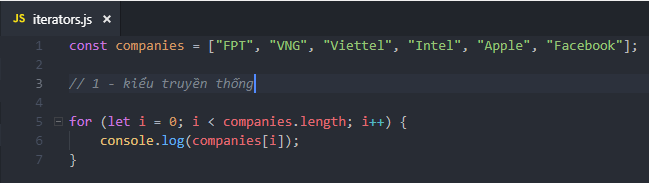
**VD4: Sử dụng delete trong strict mode:**



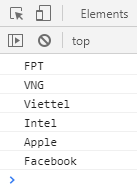
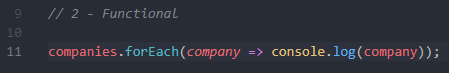


1. **Vòng lặp trong javascript:**

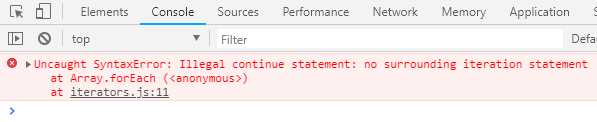
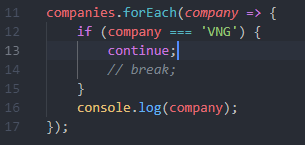
* **Kiểu truyền thống:**



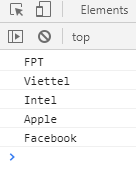
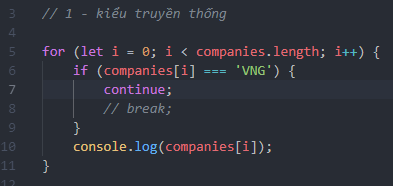
* **Functional:**



Nhưng với cách này thì không thể sử dụng **break** hay **continues** được

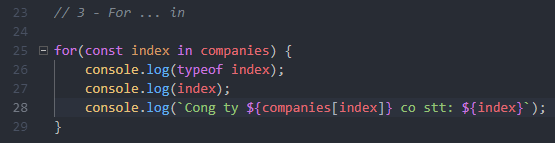


Với kiểu truyền thống thì xài được:



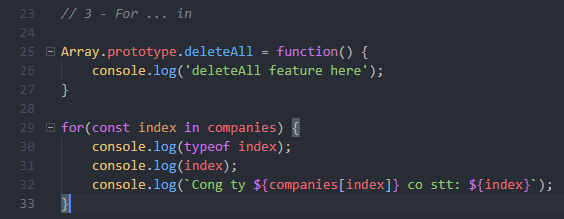
* **For … in**

Index ở đây là chỉ số phần tử của mảng (kiểu string)

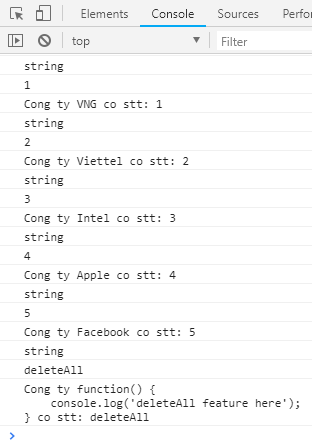




Nhưng khuyết điểm của for … in là khi ta xài thư viện có thêm tính năng cho array:

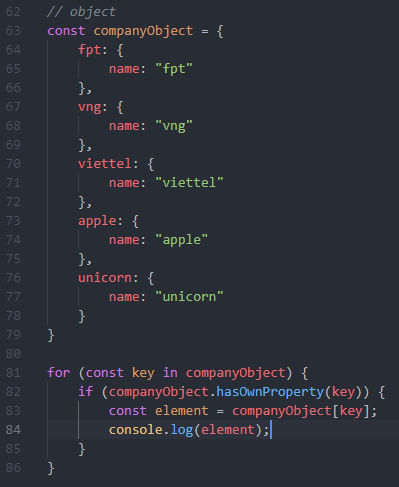
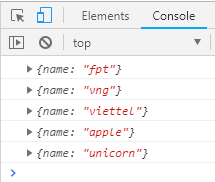


Lúc này kết quả sẽ bị ảnh hưởng và nó in ra luôn trong prototype

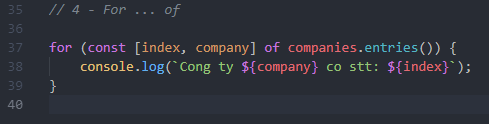


\* Sử dụng với object:

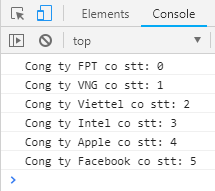
key ở đây chỉ lấy ra key của object, sau đó dựa vào key này mới lấy ra value được.

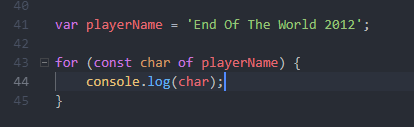
* **For … of**



\* Sử dụng companies.entries() để lấy ra được chỉ số phần tử trong for … of

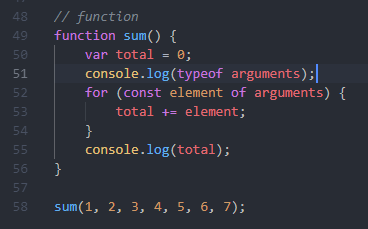
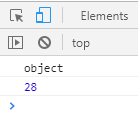


\* Sử dụng với string:

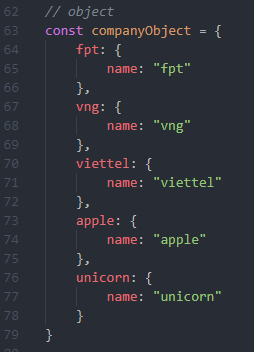




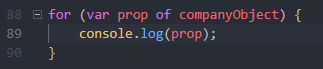
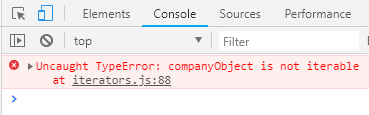
\* Sử dụng với arguments của function

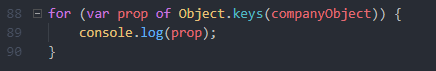
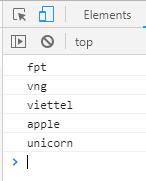
\* Sử dụng trong object:



- Sử dụng trực tiếp với object như này sẽ báo lỗi

- Khắc phục thì ta dùng Object.keys(companyObj)

 Ở đây nó trả về key của companyObject chứ không phải value.

Trả về value:

