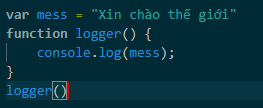
**JAVASCRIPT NÂNG CAO + ES6**

1. **Closure**
   1. *Khái niệm:*

* *Scope (phạm vi):*
* *Các loại phạm vi*

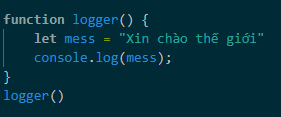
+ Global (toàn cục): bất cứ đâu trong chương trình đều có thể truy cập được biến hay hàm đó

*Vd:*



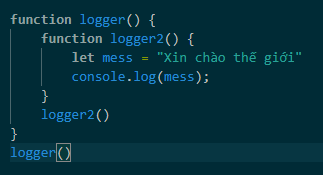
+ Code block( khối mã: let,const): khi khai báo ra biến với từ khóa let hay const thì chỉ truy cập được bên trong khối mã, bên ngoài nó không truy cập được

*Vd:*



+ Local scope (Hàm: var, function): là những biến hay hàm được tạo bên trong function khác thì chỉ sử dụng được cái biến hay hàm bên trong hàm cha

*Vd:*



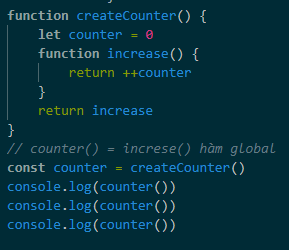
* Khi gọi mỗi hàm luôn có một phạm vi mới được tạo ra
* Các hàm có thể truy cập các biến được khai báo trong phạm vi của nó và bên ngoài nó
* Cách thức một biến được truy cập: Biến sẽ luôn tìm ở phạm vi gần nhất
* Khi nào một biến được xóa khỏi bộ nhớ

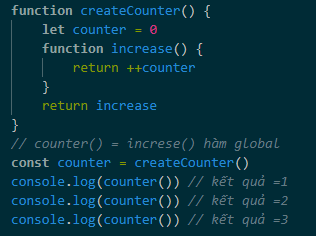
+ Global: các biến sẽ được xóa khi mình tắt task đi. Hạn chế sử dụng biến global để tránh tốn bộ nhớ

+ Biến trong code block & hàm: khi thoát khỏi 1 phạm vi hoặc khi hàm thực thi xong thì nó sẽ tự xóa nó ra khỏi bộ nhớ

+ Biến trong hàm được tham chiếu bởi 1 hàm khác: khi một hàm chạy xong mà biến vẫn được tham chiếu ở một hàm khác thì nó sẽ không được xóa đi

Vd:



* *Closure:*
* Closure là 1 hàm có thể ghi nhớ nơi nó được tạo ra và truy cập được biến ở bên ngoài phạm vi của nó
* 
* Hàm Closures có thể truy cập biến ở 3 phạm vi:
* Biến toàn cục (global)
* Biến được khai báo ở hàm chứa hàm closures (outer function)
* Biến ở bên trong hàm closures

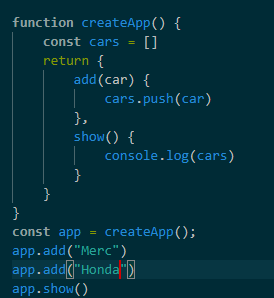
Vd:



* 1. *Ứng dụng của closure*
* Viết code ngắn gọn hơn

|  |  |
| --- | --- |
| VD: | Kết quả |
|  |  |

* Biểu diễn, ứng dụng tính private trong OOP : tạo ra tính an toàn, toàn vẹn cho dữ liệu



* 1. *Lưu ý:*
* Biến được tham chiếu trong closure sẽ không được xóa khỏi bộ nhớ khi hàm cha thực thi xong

1. **Context, this, bind**
   1. *Context*

* Context có thể hiểu đơn giản là **ngữ cảnh**. Thường được xác định bằng cách gọi hàm. Ngữ cảnh cung cấp thông tin về môi trường thực thi đoạn mã. Ví dụ lệnh gọi hàm trong ngữ cảnh A sẽ có thể sử dụng được các biến định nghĩa trong A.
  1. *This*
* Trong JavaScript, bạn có thể sử dụng từ khóa this trong ngữ cảnh **toàn cục** và ngữ cảnh **hàm**. Hơn nữa, hành vi của từ khóa this cũng thay đổi giữa chế độ nghiêm ngặt (**strict mode**) và không nghiêm ngặt (**none-strict mode**).
* *This* trong ngữ cảnh toàn cục: Trong ngữ cảnh toàn cục (global context), this tham chiếu đến đối tượng toàn cục

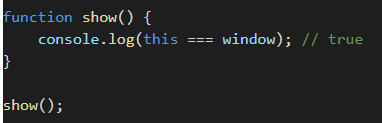
*Vd:*



* *This* trong ngữ cảnh hàm:
* *This* trong lời gọi thông thường:

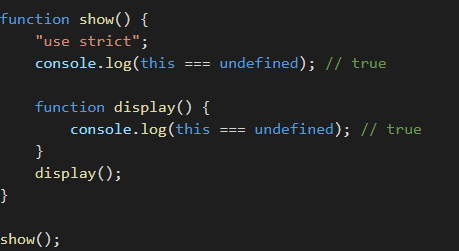
*+* Trong chế độ không nghiêm ngặt (non-strict mode), this tham chiếu đến đối tượng toàn cục khi hàm được gọi

Vd:



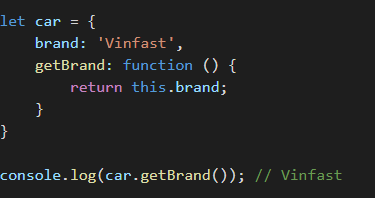
*+* Trong chế độ nghiêm ngặt (strict mode), để hạn chế hành vi không rõ ràng, lời gọi hàm như vậy sẽ được JavaScript đặt giá trị this thành undefined

Vd:



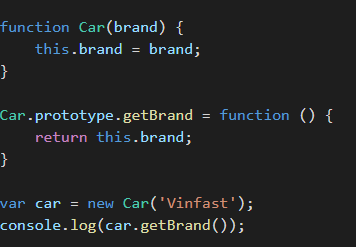
* *This* trong lời gọi phương thức: Khi bạn gọi một phương thức của một đối tượng, JavaScript sẽ đặt this cho đối tượng sở hữu phương thức

Vd:



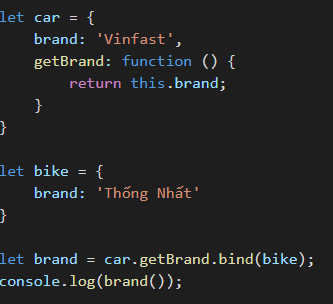
* *This* trong lời gọi hàm khởi tạo: Khi bạn sử dụng từ khóa new để tạo một thể hiện của một đối tượng hàm, bạn sử dụng hàm như một phương thức khởi tạo (constructor)

Vd:



* 1. *Bind*
* Phương thức bind() trả về một hàm mới, khi gọi, this sẽ được chỉ định cụ thể

Vd:



* Phương thức bind() cũng giúp mượn phương thức, mượn hàm (function borring) từ một đối tượng khác mà không cần nhân bản nó.

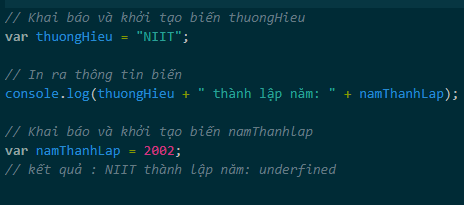
Vd:



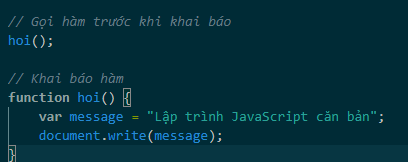
1. **Hosting**
   1. *Khái niệm*

Hoisting là hành vi mặc định của javascript, cho phép các khai báo biến hoặc hàm lên đầu phạm vi của chúng trước khi thực hiện đoạn code

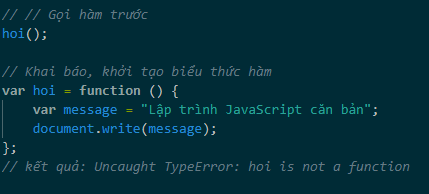
* 1. *Hành vi hoisting đối với biến*
* JavaScript chỉ lưu trữ các khai báo, không lưu trữ các khởi tạo. Bởi vì khi chạy Javascript sẽ thực hiện khai báo một lượt các biến chưa sử dụng và lưu trữ nó



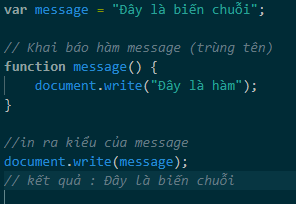
* 1. *Hành vi hoisting đối với hàm*
* Đối với khai báo hàm: Cũng giống như hành vi hoisting đối với biến, việc khai báo sẽ được chuyển lên trên đầu



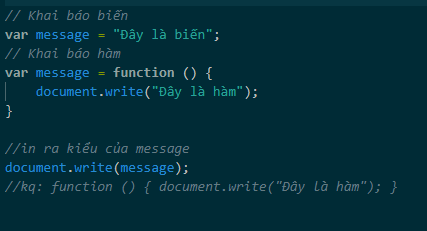
* Đối với biểu thức hàm: Vì biểu thức hàm bản chất là hàm được gán cho một biến. Do đó, nó cũng giống như vừa khai báo và vừa khởi tạo biến.
  + Thế nên, cơ chế hoisting không áp dụng cho biểu thức hàm.



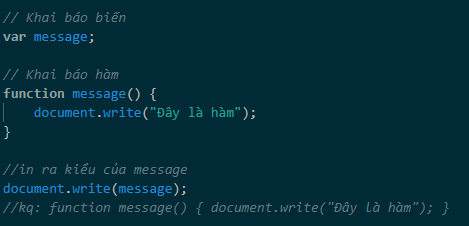
* 1. *So sánh thứ tự ưu tiên trong cơ chế hoisting*
* Gán biến sẽ có độ ưu tiên hơn khai báo hàm



* Biểu thức hàm được ưu tiên hơn gán biến



* Khai báo hàm ưu tiên hơn khai báo biến



* 1. *Biến “let”, “const”, “var” trong hoisting*
* Var: trong javascript, tất cả các biến được khai báo bằng keyword ***var*** sẽ có giá trị ban đầu là ***undefined***. Chúng có giá trị như vậy chính là do hoisting, việc khai báo các biến được đưa vào bộ nhớ và chúng được khởi tạo với giá trị là ***undefined***
* Let, const: biến được khai báo bằng keyword ***let*** hoặc ***const*** khi hoisting ta sẽ thấy chúng không được khởi tạo với giá trị undefined. Thay vào đó, chúng ở trong một trạng thái gọi là **Temporal Dead Zone ( Vùng tạm thời không truy cập được)**, trạng thái này kéo dài từ khi vào scope cho đến khi khai báo xong, trong khoảng này chúng sẽ không thể truy xuất/tham chiếu tới được. Trạng thái TDZ kết thúc sau khi việc khai báo kết thúc.

1. **Prototype**
   1. *Khái niệm*

* 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.

Vd:

|  |  |
| --- | --- |
|  |  |

Như vậy, mỗi object sẽ được chia làm 2 phần như hình vẽ.

* Phần đầu tiên là những thuộc tính riêng
* Phần thứ hai là prototype, chứa những phương thức và thuộc tính được kế thừa từ object. Cụ thể trong ví dụ này là object Number.
  1. *Thêm thuộc tính và phương thức vào prototype*
* Thêm thuộc tính vào prototype:

Cú pháp: ObjectName.prototype.property\_name = data;

Vd:

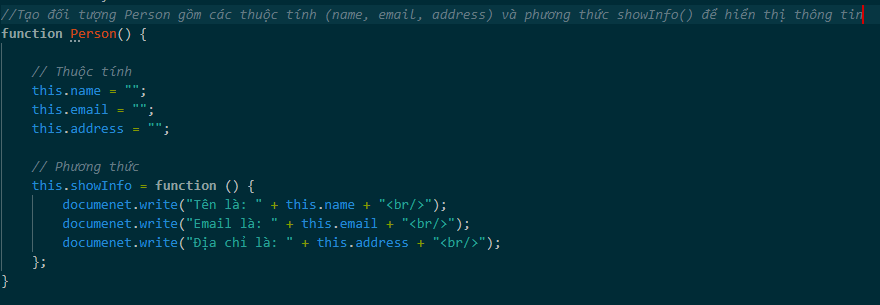
|  |  |
| --- | --- |
|  |  |

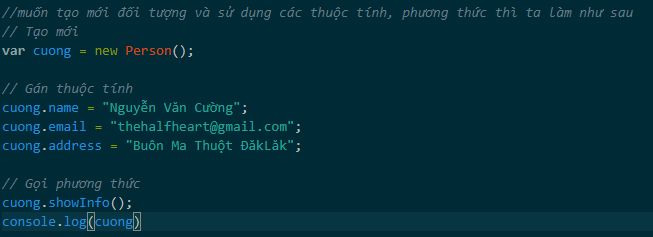
* Thêm phương thức vào prototype:

|  |  |
| --- | --- |
|  |  |

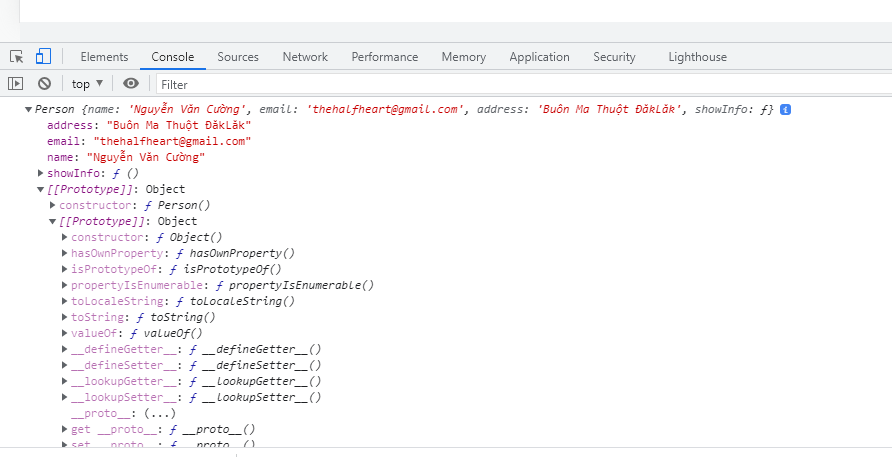
* 1. *Khởi tạo một prototype*

Vd:





Kết quả:



Như vậy, một đối tượng prototype đã được tạo ra

* 1. *Prototype trong class javascript*

Class được thêm vào từ ES6, nó cũng là một đối tượng, nên nó cũng có các thuộc tính và prototype.

Vd:

|  |  |
| --- | --- |
|  |  |

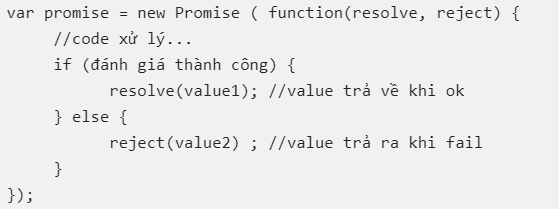
Như vậy, những thuộc tính sẽ được xem là của đối tượng, còn những phương thức thì nó sẽ đưa vào trong prototype

* 1. *Sự quan trọng của prototype trong Javascript*
* Sự kế thừa trong Javascript
* Truy cập vào các thuộc tính của đối tượng prototype chain(quá trình truy xuất đến các thuộc tính giữa các lớp kế thừa với nhau)

1. **Bất đồng bộ**

|  |  |  |
| --- | --- | --- |
|  | **Ưu điểm** | **Nhược điểm** |
| **Đồng bộ** | - Code chương trình sẽ chạy tuần tự từ trên xuống dưới. Khi nào lệnh trên hoàn thành thì lệnh dưới mới được chạy  - Dễ kiểm soát quá trình xử lý  - Dễ kiểm soát lỗi phát sinh | **-** Do chạy theo thứ tự nên sẽ có trạng thái chờ, lệnh trên chạy quá lâu sẽ làm ảnh hướng đến các lệnh dưới |
| **Bất đồng bộ** | **-** Code chương trình có thể không theo tuần tự nữa, nhiều lệnh có thể thực hiện cùng lúc. Có khi lệnh dưới cho kết thúc và cho kết quả trước cả lệnh phía trên  - Tối ưu được sức mạnh của hệ thống  - Giúp giảm thời gian chờ, giúp code chạy nhanh hơn | **-** Không phải hệ thống nào cũng dùng bất đồng bộ  -  Khó làm quen xử lý và kiểm soát lỗi phát sinh |

* 1. *Callback hell*
* Callback function là một cách thức phổ biến, dễ hiểu, dễ triễn khai. Với những vấn đề phức tạp, việc xử lý bất đồng bộ bằng callback nếu làm không tốt thì có thể xảy ra tình huống nhiều hàm callback lồng nhau (callback hell)
* Callback hell là trạng thái code lồng nhau nhiều cấp dẫn đến hình thành hình kim tự tháp code, nhìn rất phức tạp, khó hiểu, khó debug lỗi, khó maintain…
* Cách xử lý callback hell:
* Thiết kế ứng dụng theo dạng module
* Định nghĩa hàm trước khi gọi
* Sử dụng Promises (ES2015 – ES6)
* Sử dung Async/Await (ES2017 – ES8)
  1. *Promise*
* Một đối tượng dùng trong những lúc xử lý bất đồng bộ. Một promise đại diện cho 1 tác vụ nào đó chưa hoàn thành ngay được. Ở 1 thời điếm trong tương lai, promise sẽ trả về giá trị khi thành công (resolve) hoặc thất bại (reject).
* Bằng cách dùng **promise** , bạn có thể kết hợp các hàm xử lý để sử dụng kết quả khi xử lý bất đồng bộ hoàn tất. Nhờ vậy mà bạn có thể lập trình bất đồng bộ gần giống với kiểu lập trình đồng bộ – tức là đợi xử lý bất đồng bộ xong mới thực thi các thao tác cần sử dụng kết quả của xử lý đó.
* *Cú pháp:*



* Các trạng thái của Promise:

Mỗi đối tượng promise có các trạng thái như sau:

* **Pending**(đang xử lý) : ): là trạng thái mà promise đã tạo ra nhưng chưa được thực thi.
* **Fulfilled**(resolve – đã hoàn thành) : là trạng thái mà Promise đã được thực thi và return kết quả resolve(). Khi hàm resolve được trả về – promise gọi đến lệnh .then() – để tiếp tục thực hiện logic tiếp theo.
* **Rejected**(đã bị từ chối): Promise đã thực thi và return kết quả reject(). Khi hàm reject được trả về – promise gọi đến lệnh .catch() – để tiếp tục thực hiện logic tiếp theo.
  1. *Asyn await*

**Async / Await** là một tính năng của JavaScript giúp chúng ta làm việc với các hàm bất đồng bộ theo cách thú vị hơn và dễ hiểu hơn. Nó được xây dựng trên Promises và tương thích với tất cả các Promise dựa trên API.

* **Async / Await**  là cách viết gọn hơn để tạo, trả về và resolve Promise.
* Async/Await không thay thế promise mà nó kết hợp với promise để cho ra cú pháp dễ nhìn hơn

***Trong đó:***

* **Async** : khai báo một hàm bất đồng bộ (async function someName(){...}).
  + Tự động biến đổi một hàm thông thường thành một Promise.
  + Khi gọi tới hàm async nó sẽ xử lý mọi thứ và được trả về kết quả trong hàm của nó.
  + Async cho phép sử dụng Await.
* **Await** : tạm dừng việc thực hiện các hàm async. (Var result = await someAsyncCall ())
  + Khi được đặt trước một Promise, nó sẽ đợi cho đến khi Promise kết thúc và trả về kết quả.
  + Await chỉ làm việc với Promises, nó không hoạt động với callbacks.
  + Await chỉ có thể được sử dụng bên trong các function async.

Vd:

|  |  |
| --- | --- |
|  |  |

1. **Class**

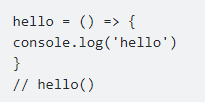
* Lớp để tạo ra các đối tượng (thực chất là một cách thức khác tạo ra các đối tượng Javascript đã biết). Đối tượng trong nó có các thuộc tính, phương thức. Tạo ra đối tượng sử dụng từ khóa class và hàm khởi tạo constructor
* Tính kế thừa của lớp: Giống như nhiều ngôn ngữ lập trình hướng đối tượng, JS cung cấp tạo ra một lớp mới có kế thừa lại thuộc tính, phương thức của một lớp khác. Để tạo ra một lớp con kế từ lớp cha dùng từ khóa extends

**ES6**

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

|  |  |
| --- | --- |
| **let** | **Const** |
| Từ ES6 | Từ ES6 |
| Let + tên biến | Const + tên biến = giá trị |
| Chỉ có phạm vi trong khối khai báo (block scope)  Không thể khai báo lại trong cùng phạm vi | Chỉ khai báo một lần, không thể thay thế giá trị |

* 1. **Arrow function**
* Arrow function là hàm giúp code trở nên ngắn gọn, súc tích hơn
* Arrow Function sử dụng kí tự =>



* Arrow function ta có thể bỏ qua từ khoá return



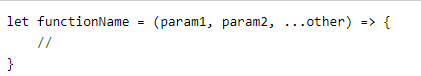
* Sử dụng được arrow function trong trường hợp: map, filter, foreach,... Và code sẽ tối giản hơn so với việc sử dụng function thông thường

|  |  |
| --- | --- |
| **Arrow function** | **Function thông thường** |
|  |  |

* Khác với function thông thường, **arrow function không có bind. V**ì vậy, không định nghĩa lại this. Do đó, this sẽ tương ứng với ngữ cảnh gần nhất của nó
* Bởi nó không định nghĩa this, nên ***arrow function không phù hợp làm method của object***, vì vậy nếu định nghĩa method của object, function vẫn là sự lựa chọn đúng đắn.
  1. **Default Parameter**
* Là giá trị mặc định của tham số khi truyền vào các function
* So sánh giữa ES5 và ES6

|  |  |  |
| --- | --- | --- |
|  | ES5 | ES6 |
| Khái niệm | Trong ES5 để tạo giá trị mặc định thì bạn sử dụng cặp dấu “||” để thiết lập ngay bên trong thân của hàm | Trong ES6 có cách khai báo giá trị mặc định đơn giản hơn rất nhiều - đó là sử dụng phép gán ngay tại vị trí khai báo tham số cho function. |
| Ví dụ |  |  |

* 1. **Rest**
* 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.
* Cú pháp: Để khai báo các tham số còn lại của một function thì bạn đặt 3 dấu chấm “.” trước biến đại diện



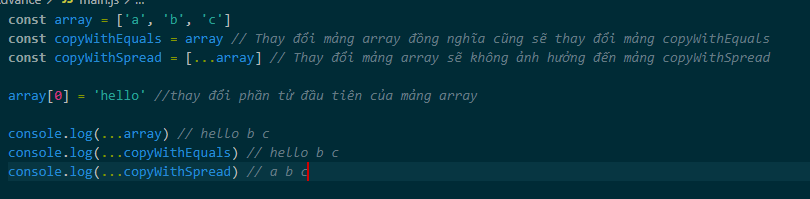
Trong đó, tham số other là một Rest Parameter vì nó có 3 dấu chấm đặt ở trước.

Vd:

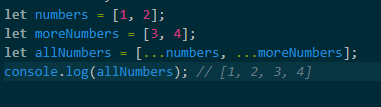
|  |  |
| --- | --- |
|  |  |

* **Nhược điểm:** Khi truyền với số lượng tham số không xác định thì rất khó nâng cấp và bảo trì
  1. **Spread**
* Được sử dụng trong các câu lệnh, biểu thức hoặc khi gọi hàm.
* Được sử dụng trong các câu lệnh, biểu thức hoặc khi gọi hàm.
* Giải nén các phần tử của các đối tượng có thể lặp như mảng, object và map vào một danh sách
* Sự hữu dụng của spread operator, đó là chúng sẽ tạo ra một tham chiếu mới, sau đó sao chép giá trị của tham chiếu cũ vào tham chiếu mới này. Khi đó, mọi thao tác làm thay đổi tham chiếu cũ sẽ không ảnh hưởng đến mảng được sao chép, điều mà sẽ xảy ra nếu chúng ta sao chép mảng sử dụng phương thức gán “=”

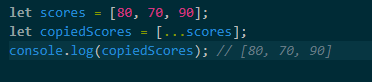
**Vd:**



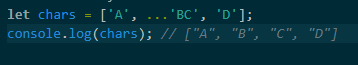
* Sử dụng Spread với:
* **Array**

+ Nối mảng:  


+ Coppy mảng:

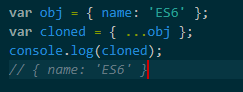


* **String**

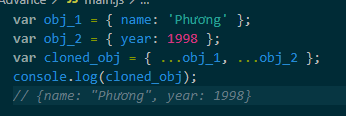


* **Object**

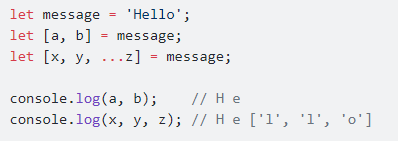
+ Coppy mảng:



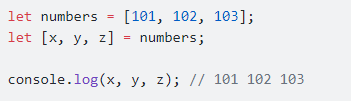
+ Gộp mảng:



* 1. **Destructuring Assignment**
* Là một cú pháp cho phép tách dữ liệu được lưu trữ bên trong Objects hoặc Arrays và gán chúng cho các biến riêng biệt.
* Sử dụng mảng để tách các phần tử thành các biến,  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**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator) nếu cần thiết
* **Strings:**



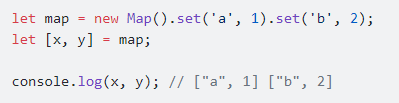
* **Arrays:**



* **Sets:**

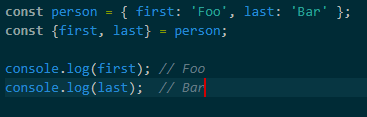


* **Maps:**



* Sử dụng Object để gán property value của một object cho các biến tương ứng:

**Vd:**



* 1. **Template Literals**
* Template Literals hay còn gọi là Template Strings là một cú pháp mới để khai báo String trong Javascript được giới thiệu trong ES2015/ES6.
* Nó cho phép chúng ta sử dụng [multi-line string](http://stackoverflow.com/questions/805107/creating-multiline-strings-in-javascript), sử dụng biến, biểu thức, hàm bên trong string mà không phải thông qua phép cộng string
* So sánh:

|  |  |  |
| --- | --- | --- |
|  | **Trước ES6** | **ES6** |
| **String** | - Sử dụng một dấu nháy đơn **''** hoặc dấu nháy kép **" "** với các chuỗi  Vd: | - Chỉ cẩn sử dụng dấu nháy đơn **``**   **Vd:** |
| **Multiline Strings** | **Vd:** | **Vd:** |
| Biểu thức nội suy | **-** Sử dụng toán tử + để nối các biến và biểu thức trong một chuỗi  Vd: | **-**  Sử dụng cú pháp $ {…} để gán các biến và biểu thức bên trong một chuỗi  Vd: |
| Tagged Templates | **-** Sử dụng một hàm để truyền các đối số  Vd: | **-** Được viết giống như một định nghĩa hàm  Vd: |