# Khai báo biến bằng từ khoá let

## **Giới thiệu nội dung bài viết**

Trong ES5 chúng ta sử dụng từ khoá var để khai báo một biến. Khi sử dụng từ khoá var thì phạm vi hoạt động của biến đó là toàn cục, ở đâu cũng có thể thấy được. Ở phiên bản mới ES6 chúng ta có một cách mới để khai báo biến đó là từ khoá let. Từ khoá let cũng tương tự như từ khoá var nhưng có điều giá trị của nó chỉ tồn tại trong khối lệnh, khi thoát ra ngoài khối lệnh ( trong javascript khối lệnh được ký hiệu là trong dấu ngoặc nhọn { … } ) thì giá trị đó vẫn giữ nguyên.

Ex:

|  |
| --- |
| let x = 10;  if (x == 10) {  let x = 20;  console.log(x); // 20: reference x inside the block  }  console.log(x); // 10: reference at the begining of the script |

## **Javascript let và global(toàn cục)**

Khi chúng ta sử dụng từ khoá var để khai báo biến thì phạm vi của nó sẽ là global (toàn cục). Ví dụ như ta khai báo biến a = 10;

|  |
| --- |
| var a = 10;  console.log(window.a); // 10 |

Khi đó khi ta sử dụng biến window.a ta vẫn thấy và lấy được giá trị a vì biến a là biến toàn cục ở đâu cũng lấy được.

Tuy nhiên nếu như ta sử dụng biến let để khai báo trong trường hợp sau

|  |
| --- |
| let b = 20;  console.log(window.b); // undefined |

Lúc này biến b không được thêm vào đối tượng global như một thuộc tính nên sẽ không thấy được giá trị của b. Chính vì vậy khi ta gọi window.b sẽ nhận được thông báo biến b chưa khai báo.

## **Callback function trong vòng lặp**

Ex:

|  |
| --- |
| for (var i = 0; i < 5; i++) {  setTimeout(function () {  console.log(i);  }, 1000);  } |

Thực tế khi chạy đoạn mã trên anh sẽ nhận được **in ra số 5 trong 5 lần chạy**. Sau khi chạy 5 vòng lặp thì giá trị của i lúc này là bằng 5. Sau đó nó truyền số 5 này vào callback là setTimeout trong 1 giây. Chính vì nó dùng chung 1 biến i nên giá trị của 5 lần đó đều giống nhau cả.

Để giải quyết được vấn đề này thì chúng ta sử dụng let như sau

|  |
| --- |
| for (let i = 0; i < 5; i++) {  setTimeout(function () {  console.log(i);  }, 1000);  } |

## **Khai báo lại giá trị**

Sử dụng từ khoá var cho phép chúng ta khai báo lại giá trị của biến mà không bị báo lỗi

|  |
| --- |
| var counter = 0;  var counter;  console.log(counter); // 0 |

Ở ví dụ trên anh khai báo biến counter = 0, sau đó khai báo lại thì khi chạy chương trình vẫn okie không báo lỗi

Tuy nhiên nếu chúng ta sử dụng let thì sẽ bị báo lỗi như sau

|  |
| --- |
| let counter = 0;  let counter;  console.log(counter); |

Chúng ta sẽ nhận được lỗi là

|  |  |
| --- | --- |
| 1 | Uncaught SyntaxError: Identifier 'counter' has already been declared |

## **Hoisting**

Ex:

|  |
| --- |
| {  console.log(counter); //  let counter = 10;  } |

Như các em thấy anh sử dụng biến counter trong hàm console.log trước khi anh khai báo biến counter = 10. Hoisting có nghĩa là mình sử dụng biến trước khi khai báo nó.

Thực tế khi Javascript chạy và biên dịch nó sẽ di chuyển dòng khai báo counter = 10 lên trước khối lệnh console.log

Khi chạy dòng khối lệnh ở trên ta sẽ nhận được lỗi như sau

|  |
| --- |
| Uncaught ReferenceError: Cannot access 'counter' before initialization  Code language: JavaScript (javascript) |

Chúng ta nhận được lỗi như vậy vì khi chương trình chạy nó sẽ di chuyển dòng lệnh khai let counter lên phía trên khối lệnh. Tuy nhiên nó không gán giá trị 10 vào biến counter, chính vì vậy mà mình sẽ gặp lỗi ReferenceError.

Trên thực tế các anh cũng không bao giờ sử dụng biến trước khi khai báo, mà chúng ta phải tập thói quen khai báo biến rồi mới sử dụng nó. Trong thực tế anh đã gặp rất nhiều trường hợp các bạn dev khai báo biến vô tội vạ , thích thì khai báo biến ở trên, thích thì khai báo ở giữa hoặc cuối file JS nên làm cho việc debug chương trình rất khó khăn.

# Hướng dẫn khai báo hằng số trong ES6

## **Cách khai báo hằng số trong ES6**

Trong ES6 chúng ta sử dụng từ khoá const để khai báo một hằng số. Hằng số là một giá trị không bao giờ thay đổi. Anh lấy ví dụ như số PI là 3.14 đó là một số bất biến.

Cú pháp khai báo hằng số như sau.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | const CONSTANT\_NAME = value; |  |  |  |  |  |  |  |  |  |  |  |  |

Tên biến của hằng số phải là các ký tự viết Hoa. Ví dụ

|  |  |
| --- | --- |
|  | const RATE = 0.1; |

Sự khác nhau giữa let và const là khi sử dụng const chúng ta không thể reassign (thay đổi giá trị của biến) như let thì chúng ta có thể làm được. Anh lấy ví dụ về let và const như sau.

|  |
| --- |
| let a = 10;  a = 20;  a = a + 5;  console.log(a); // 25 |

Khi sử dụng let các em thấy chúng ta có thể thay đổi giá trị của biến a từ 10 sang 20 và 25.

|  |
| --- |
| const RATE = 0.1;  RATE = 0.2 // TypeError |

Khi sử dụng biến const nếu ta thay đổi giá trị từ 0.1 sang 0.2 ta sẽ gặp lỗi Type Error

Khi khai báo hằng chúng ta bắt buộc phải có giá trị cho hằng số. Nếu chúng ta khai bao như sau sẽ báo lỗi

|  |
| --- |
| const RED; // SyntaxError |

## **Object trong ES6**

Trong ES6, Chúng ta sử dụng const để đảm bảo các biến chỉ được phép đọc và không được thay đổi giá trị. Tuy nhiên đối với các thuộc tính của một đối tượng ta có thể thay đổi được.

|  |
| --- |
| const person = { age: 20 };  person.age = 30; // OK  console.log(person.age); // 30 |

Khi khai báo const person thì chúng ta mặc định đối tượng person là không thay đổi. Chính vì vậy nếu ta thay đổi đối tượng person như đoạn mã sau đây thì sẽ bị lỗi.

|  |
| --- |
| person = {age: 40}; // TypeError |

Vì chúng ta không thể gắn person cho một giá trị khác.

## **Array trong ES6**

Ví dụ anh có một mảng màu sắc chứa các màu sau như sau

|  |
| --- |
| const colors = ['red'];  colors.push('green');  console.log(colors); // ["red", "green"]  colors.pop();  colors.pop();  console.log(colors); // []  colors = []; // TypeError |

## **Vòng lặp trong ES6**

Trong ES6 chúng ta có cú pháp mới để chạy vòng lặp duyệt qua các mảng gọi là for of. Anh có ví dụ như sau:

|  |
| --- |
| let scores = [75, 80, 95];  for (let score of scores) {  console.log(score);  } |

Trong trường hợp chúng ta không có ý định modify (thay đổi) giá trị bên trong vòng lặp chúng ta có thể sử dụng từ khoá const trong vòng lặp như sau:

|  |
| --- |
| let scores = [75, 80, 95];  for (const score of scores) {  console.log(score);  } |

Mỗi lần vòng lặp chạy nó sẽ tạo ra một score mới. Ở ví dụ trên chúng ta sẽ nhận được là 75,80,95

Chú ý sẽ có những bạn khai báo const như sau sẽ bị lỗi trong vòng lặp

|  |
| --- |
| for (const i = 0; i < scores.length; i++) { // TypeError  console.log(scores[i]);  } |

Chúng ta sẽ nhận lỗi là Uncaught TypeError: Assignment to constant variable vì biến i đã thay đổi giá trị.

# Giá trị mặc định cho tham số

## **Phân biệt tham số và đối số**

Anh có ví dụ là phương thức add có 2 tham số truyền vào là x,y. Khi gọi phương thức add thì trả về kết quả cộng 2 số.

|  |
| --- |
| function add(x, y) {  return x + y;  }  add(100,200); |

Như vậy x và y được gọi là THAM SỐ. Chúng được sử dụng để định nghĩa phương thức add sẽ có 2 giá trị truyền vào.

Còn phương thức gọi hàm add(100,200). Thì 100 và 200 được gọi là ĐỐI SỐ. Đối số là giá trị ta sẽ truyền cho phương thức.

## **Thiết lập giá trị mặc định cho phương thức**

Ví dụ anh có phương thức say(message) với tham số là message. Khi gọi vào phương thức say() thì in cho anh messeag.

|  |
| --- |
| function say(message) {  console.log(message);  }  say(); // undefined |

Khi phương thức say() được gọi chúng ta sẽ nhận được kết quả là Undefined bởi vì chúng ta không truyền đối số cho hàm say()

Giả sử chúng ta mong muốn khi không truyền đối số vào thì chúng ta sẽ đặt giá trị mặc định cho message là 10. Lúc này ta phải chỉnh sửa lại đoạn mã ở trên và thêm phần kiểm tra giá trị biến message như sau

|  |
| --- |
| function say(message) {  message = typeof message !== 'undefined' ? message : 'Hi';  console.log(message);  }  say(); |

Như vậy khi ta gọi hàm say() chúng ta sẽ nhận được giá trị là Hi.

Trong ES6 hỗ trợ chúng ta cách đặt giá trị mặc định cho tham số mà chúng ta không cần phải viết code để kiểm tra với cú pháp như sau.

|  |
| --- |
| function fn(param1=default1,param2=default2,..) {  } |

Chúng ta sử dụng dấu = và giá trị sau tham số để đặt giá trị mặc định. Ví dụ như chúng ta muốn đặt giá trị Hi cho biến message thì chúng ta làm như sau.

|  |
| --- |
| function say(message='Hi') {  console.log(message);  }  say(); // 'Hi'  say(undefined); // 'Hi'  say('Hello'); // 'Hello' |

Khi chúng ta không truyền giá trị thì chúng ta sẽ nhận được chuỗi mặc định là Hi. Còn nếu chúng ta truyền vào đối số là chữ Hello thì sẽ nhận được kết quả Hello

## **Thiết lập giá trị mặc định bằng cách sử dụng tham số khác**

Anh có ví dụ như sau.

|  |
| --- |
| function add(x = 1, y = x, z = x + y) {  return x + y + z;  }  console.log(add()); // 4 |

Khi gọi phương thức add chúng ta có giá trị mặc định của x là 1, giá trị của y là 1 và giá trị của z là 2. Nhưng vậy khi gọi phép toán cộng ta sẽ có 1 + 1 + 2 nên kết quả nhận được sẽ là 4.

## **Thiết lập giá trị mặc định bằng cách sử dụng phương thức**

Anh có ví dụ như sau

|  |
| --- |
| let taxRate = () => 0.1;  let getPrice = function( price, tax = price \* taxRate() ) {  return price + tax;  }  let fullPrice = getPrice(100);  console.log(fullPrice); // 110 |

Tham số tax trong phương thức là bằng price nhân với phương thức taxRate. Như vậy chúng ta có thể khai báo giá trị mặc định trong phương thức bằng cách gọi một phương thức khác.

# Sử dụng Rest Param trong ES6

## **1. Giới thiệu về rest param trong ES6**

ES6 cung cấp cho chúng ta một loại tham số mới truyền trong phương thức được gọi là rest param. Có nghĩa là chúng ta có thể truyền một mảng các tham số vào phương thức. Để truyền được mảng các tham số của một phương thức chúng ta dùng dấu 3 chấm như sau.

|  |
| --- |
| function fn(a,b,...args) {  //...  } |

Như các em thấy phương thức fn có tham số là a,b và … args. Thì …args được gọi là rest param. Chúng ta có thể truyền nhiều giá trị vào.

Ví dụ như chúng ta gọi hàm fn và truyền tham số sau

|  |  |
| --- | --- |
| 1 | fn(1,2,3,'A','B','C'); |

Lúc này tham số ở phương thức fn(a,b,…args) thì a sẽ ứng với số 1, b sẽ ứng với số 2 và mảng [3,’A’,’B’,’C’] sẽ ứng với tham số …args

Nếu như ta chỉ truyền 2 tham số vào phương thức fn như sau

|  |  |
| --- | --- |
| 1 | fn(1,2); |

Thì lúc này mảng …args sẽ là mảng rỗng [].

Chú ý rest param luôn luôn ở vị trí là tham số sau cùng của phương thức. Nếu như ta đặt tham số rest ở giữa sẽ bị lỗi như sau.

|  |
| --- |
| function foo(a,...rest, b) {  // error  }; |

Chúng ta phải đặt lại là foo(a,b,…rest)

## **2. Ví dụ rest parameter trong ES6**

Anh có phương thức sum như sau:

|  |
| --- |
| function sum(...args) {  let total = 0;  for (const a of args) {  total += a;  }  return total;  }  sum(1, 2, 3); |

Khi gọi sum thì kết quả nhận được sẽ là 6.

Vì …agrs là một mảng nên ta cũng có thể sử dụng vòng lặp for để duyệt qua các phần tử hoặc dùng cách phương thức có sẳn của một array như filter

|  |
| --- |
| function sum(...args) {  return args.filter(e => typeof e === 'number')  .reduce((prev, curr)=> prev + curr);  } |

# Sử dụng spread operator trong ES6

## **1. Giới thiệu về toán tử spread**

|  |
| --- |
| const odd = [1,3,5];  const combined = [2,4,6, ...odd];  console.log(combined); |

Các em thấy trong mảng combined chúng ta truyền mảng odd vào và trước nó là 3 dấu chấm (…odd). Khi chương trình chạy thì nó sẽ in ra là 2,4,6,1,3,5.

Như vậy …odd được gọi là toán tử spread. Tham số odd sẽ uppack (giải nén các giá trị trong nó) các giá trị trong mảng odd (1,3,5) và gán vào mảng mới. Chính vì vậy mà ta nhận được kết quả là 2,4,6,1,3,5

## **2. Sự khác nhau toán tử spread và tham số rest**

Sự khác nhau giữa toán tử spread và tham số rest là :

Toán tử spread là cách ta uppack các giá trị trong mảng ra thành các phần tử. Ví dụ như mảng odd có 3 phần tử. Khi gọi combined = [2,4,6, …odd]; lúc này chúng ta có các giá trị 2,4,6,1,3,5

Ngược lại với toán tử spread thì tham số rest nhóm các giá trị đơn lẻ thành 1 biến. Ví dụ dưới đây chúng ta nhóm các giá trị đơn lẻ thành 1 biến trong tham số của phương thức f

|  |
| --- |
| function f(a, b, ...args) {  console.log(args);  }  f(1,2,3,4,5); |

Như các em thấy giá trị 3,4,5 được gán lại cho 1 biến trong mảng là args.

Tham số rest bắt buộc phải ở cuối cùng của phương thức tuy nhiên toán tử spread có thể ở bất kỳ đâu

const odd = [1,3,5];

const combined = [...odd, 2,4,6];

console.log(combined);

# Sử dụng Destructuring trong ES6

## **1. Giới thiệu về Destructuring**

ES6 cung cấp cho chúng ta một chức năng mới được gọi là Destructuring tham số nghĩa là chúng ta có thể gán giá trị thuộc tính của một đối tượng hoặc một mảng vào trong các biến. Nghe có vẻ khó hiểu nên anh sẽ giải thích thông qua ví dụ sau đây. Giả sử ta có phương thức getScore trả về một mảng các giá trị là 70,80,90 như sau.

|  |
| --- |
| function getScores() {  return [70, 80, 90];  } |

Tiếp đến ta khai báo biến scores sẽ gọi hàm getScores và chứa mảng giá trị là 70,80,90 như sau.

|  |
| --- |
| let scores = getScores(); |

Như vậy biến scores sẽ chứa mảng các giá trị 70,80,90. Để lấy các giá trị ra ta sử dụng index . Ví dụ scores[0] tương ứng với giá trị 70.

|  |
| --- |
| let x = scores[0],  y = scores[1],  z = scores[2]; |

Ở ví dụ trên ta gán scores[0] cho biến x , scores[1] cho biến y và scores[2] cho biến z. Lúc này x = 70, y = 80 và z = 90.

Trong ES6 chúng ta hoàn toàn có thể gán các giá trị cho x,y,z là 70,80,90 chỉ cần 1 dòng code thay vì cách làm ở trên như sau

|  |
| --- |
| let [x, y, z] = getScores(); |

Cách khai báo ở trên được gọi là Destructuring chúng ta khai báo biến [x,y,z] nó sẽ tương ứng với từng giá trị mảng trả về của hàm getScores(). Khi hàm thực thi xong các giá trị mảng (70,80,90) sẽ tự động gán vào biến x,y,z theo thứ tự tương ứng. Nếu chúng ta thêm console.log ta sẽ nhận các kết quả sau.

|  |
| --- |
| let [x, y, z] = getScores();  console.log(x); // 70  console.log(y); // 80  console.log(z); // 90 |

Như vậy x sẽ được gán cho vị trí đầu tiên của mảng trong hàm getScores(). Lúc này vị trí đầu tiên là 70. Như vậy y sẽ được gán cho vị trí thứ 2 của mảng trong hàm getScores(). Lúc này vị trí thứ 2 là 80. Như vậy z sẽ được gán cho vị trí thứ 3 của mảng trong hàm getScores(). Lúc này vị trí thứ 3 là 90.

## **2. Các trường hợp ngoại lệ**

Trong trường hợp hàm getScores chỉ trả về mảng có 2 giá trị, thì lúc này biến z sẽ nhận giá trị là undefined như sau

|  |
| --- |
| function getScores() {  return [70, 80];  }  let [x, y, z] = getScores();  console.log(x); // 70  console.log(y); // 80  console.log(z); // undefined |

Trong trường hợp hàm getScores trả về hơn 3 giá trị thì các giá trị bị dư ra sẽ bị loại bỏ đi.

|  |
| --- |
| function getScores() {  return [70, 80, 90, 100];  }  let [x, y, z] = getScores();  console.log(x); // 70  console.log(y); // 80  console.log(z); // 90 |

Trong trường hợp getScores trả về hơn 3 giá trị chúng ta có thế sử dụng tham số Rest để chứa các giá trị bị dư

|  |
| --- |
| let [x, y ,...args] = getScores();  console.log(x); // 70  console.log(y); // 80  console.log(args); // [90, 100] |

# Sử dụng for of trong ES6

## **1. Giới thiệu về For of**

ES6 cung cấp cho chúng ta một chức năng mới để duyệt qua các phần tử như Array, String, Map, Set hoặc các đối tượng thông qua vòng lặp for of.

Cú pháp vòng lặp for of

|  |
| --- |
| for (variable of iterable) {  // statements  } |

Variable được dùng để duyệt qua các phần tử trong mảng. Chúng ta có thể sử dụng từ khoá var, let, const để khai báo trước biến variable. iterable là object chứa dựng các phần tử.

## **2. Duyệt qua mảng**

|  |
| --- |
| let scores = [80, 90, 70];  for (let score of scores) {  score = score + 5;  console.log(score);  } |

Chúng ta sẽ nhận được kết quả là 85,95,75.

Nếu chúng ta muốn giá trị biến score không thay đổi thì sử dụng từ khoá const thay vì let như sau

|  |
| --- |
| let scores = [80, 90, 70];  for (const score of scores) {  console.log(score);  } |

kết quả nhận được là 80,90,70

## **3. Lấy vị trí của các phần tử trong mảng**

Để lấy được vị trí của các phần tử trong mảng chúng ta có thể sử dụng vòng for kết hợp với phương thức entries() của mảng như sau.

|  |
| --- |
| let colors = ['Red', 'Green', 'Blue'];  for (const [index, color] of colors.entries()) {  console.log(`${color} is at index ${index}`);  } |

Chúng ta sẽ nhận được kết quả là

|  |
| --- |
| Red is at index 0  Green is at index 1  Blue is at index 2 |

## **4. Duyệt qua chuỗi**

Chúng ta có thể sử dụng for of để duyệt qua các chuỗi như sau.

|  |
| --- |
| let str = 'abc';  for (let c of str) {  console.log(c);  } |

Kết quả nhận được là a,b,c

## **5. Duyệt qua Map**

Chúng ta sử dụng for of duyệt các phần tử Map như sau.

|  |
| --- |
| let colors = new Map();  colors.set('red', '#ff0000');  colors.set('green', '#00ff00');  colors.set('blue', '#0000ff');  for (let color of colors) {  console.log(color);  } |

Kết quả nhận được là :

|  |
| --- |
| [ 'red', '#ff0000' ]  [ 'green', '#00ff00' ]  [ 'blue', '#0000ff' ] |

## **6. Duyệt qua Set**

Chúng ta có thể sử dụng for of duyệt qua tập hợp Set như sau

|  |
| --- |
| let nums = new Set([1, 2, 3]);  for (let num of nums) {  console.log(num);  } |

## **7. Sự khác nhau for of và for in**

Chúng ta sử dụng for in để duyệt qua các thuộc tính của một đối tượng. Nó không sử dụng để duyệt các tập hợp như Array, Map hoặc Set

Không giống như vòng lặp for in vòng lặp for of được sử dụng để duyệt tập hợp hơn là duyệt các đối tượng.

## **8. Sự khác nhau for of và for**

Anh có ví dụ sau để duyệt qua các phần tử trong mảng.

|  |
| --- |
| let ranks = ['A', 'B', 'C'];  for (let i = 0; i < ranks.length; i++) {  console.log(ranks[i]);  } |

Đoạn code trên không có gì sai, tuy nhiên chúng ta phải theo dõi index (i) tăng lên, nếu mảng lồng mảng thì độ phức tạp sẽ rất cao. Chúng ta sẽ khó khăn trong việc theo dõi index trong mảng.

ES6 cho ra đời for of để giảm đi độ phức tạp tránh lỗi xảy ra khi theo dõi index của mảng. Nếu mảng lồng mảng thì độ phức tạp sẽ ít hơn vì chúng ta chỉ sử dụng giá trị trực tiếp (rank) mà không cần phải theo dõi index trong mảng.

|  |
| --- |
| for(let rank of ranks) {  console.log(rank);  } |

# Sử dụng class trong ES6

## **1. Tạo Class và đối tượng trong JS cách củ**

Trước khi có cách viết Class mới trong ES6, thì để tạo một class trong Javascript ta khai báo theo cách như sau

|  |
| --- |
| function Animal(type) {  this.type = type;  }  Animal.prototype.identify = function() {  console.log(this.type);  }  var cat = new Animal('Cat');  cat.identify(); // Cat |

Chúng ta sử dụng function và prototype để tạo ra các đối tượng và thuộc tính cho một đối tượng.

## **2. Tạo Class và đối tượng trong ES6**

Trong ES6 chúng ta tạo class và đối tượng một cách tường minh hơn bằng cách sử dụng từ khoá class để khai báo class, sử dụng từ khoá constructor để khai báo hàm dựng như sau.

|  |
| --- |
| class Animal {  constructor(type) {  this.type = type;  }  identify() {  console.log(this.type);  }  }  let cat = new Animal('Cat');  cat.identify(); |

## **3. Kiểm tra đối tượng trong ES6**

Chúng ta sử dụng instance of để kiểm tra xem đối tượng đó có phải là đối tượng chúng ta mong muốn không, có đúng kiểu dữ liệu ta yêu cầu không.

|  |
| --- |
| let Animal = class {  constructor(type) {  this.type = type;  }  identify() {  console.log(this.type);  }  }  let duck = new Animal('Duck');  console.log(duck instanceof Animal); // true  console.log(duck instanceof Object); // true  console.log(typeof Animal); // function  console.log(typeof Animal.prototype); // function |

## **4. Sử dụng Getter và Setter**

Chúng ta sử dụng từ khoá get và set để thực hiện việc lấy giá trị và set giá trị của đối tượng như sau.

|  |
| --- |
| class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  get fullName() {  return this.firstName + ' ' + this.lastName;  }  set fullName(str) {  let names = str.split(' ');  if (names.length === 2) {  this.firstName = names[0];  this.lastName = names[1];  } else {  throw 'Invalid name format';  }  }  }  let mary = new Person('Mary', 'Doe');  console.log(mary.fullName); // Mary Doe |

# Sử dụng kế thừa trong ES6

## **1. Kế thừa trong JS cách củ**

Trước khi có cách viết kế thừa trong ES6 thì để làm kế thừa trong Javascript ta sử dụng prototype khai báo theo cách như sau

|  |
| --- |
| function Animal(legs) {  this.legs = legs;  }  Animal.prototype.walk = function() {  console.log('walking on ' + this.legs + ' legs');  }  function Bird(legs) {  Animal.call(this, legs);  }  Bird.prototype = Object.create(Animal.prototype);  Bird.prototype.constructor = Animal;  Bird.prototype.fly = function() {  console.log('flying');  }  var pigeon = new Bird(2);  pigeon.walk(); // walking on 2 legs  pigeon.fly(); // flying |

## **2. Kế thừa trong JS trong ES6**

Chúng ta sử dụng từ khoá extends để sử dụng kế thừa. Trong ví dụ dưới đây chúng ta khai báo lớp Animal, Class Bird sẽ kế thừa lại Animal

|  |
| --- |
| class Animal {  constructor(legs) {  this.legs = legs;  }  walk() {  console.log('walking on ' + this.legs + ' legs');  }  }  class Bird extends Animal {  constructor(legs) {  super(legs);  }  fly() {  console.log('flying');  }  }  let bird = new Bird(2);  bird.walk();  bird.fly(); |

## **3. Shadowing method**

Gọi phương thức cha từ phương thức con. Ví dụ chúng ta có lớp Dog kế thừa lớp động vật và ghi đè lại phương thức walk() như sau

|  |
| --- |
| class Dog extends Animal {  constructor() {  super(4);  }  walk() {  console.log(`go walking`);  }  }  let bingo = new Dog();  bingo.walk(); // go walking |

Sau khi chương trình chạy chúng ta sẽ được kết quả là : go walking.

Bây giờ chúng ta muốn từ gọi phương thức walk trong lớp Aninal (lớp cha) thì chúng ta sử dụng từ khoá super như sau. Lúc này nó sẽ gọi hàm walk trong lớp Animal.

|  |
| --- |
| class Dog extends Animal {  constructor() {  super(4);  }  walk() {  super.walk(); // gọi lên phương thức walk của lớp Animal  console.log(`go walking`);  }  }  let bingo = new Dog();  bingo.walk(); |

# Sử dụng Arrow function trong ES6

## **1. Arrow function trong ES6**

Arrow function là một chức năng mới trong ES6 giúp chúng ta có thể viết phương thức một cách ngắn gọn. Anh lấy ví dụ ta có phương thức cộng 2 số viết theo cách truyền thống như sau.

|  |
| --- |
| let add = function(x,y) {  return x + y;  }  console.log(add(10,20)); // 30 |

Khi áp dụng Arrow function chúng ta sẽ viết theo cách mới ngắn gọn hơn như sau.

|  |
| --- |
| let add = (x,y) => x + y;  console.log(add(10,20)); // 30; |

Ta sử dụng ký hiệu mũi tên (=>)này để khai báo chúng ta sử dụng kiểu Arrow function. Thay vì phải viết function(x,y) thì bây giờ ta không cần từ khoá function nữa mà chỉ viết ngắn gọn (x,y). Phần thân của phương thức return x + y sẽ thay bằng x + y.

Trường hợp nếu phần thân của phương thức nằm trong block (khối lệnh) thì chúng ta phải cần thêm từ khoá return như sau

|  |
| --- |
| let add = (x, y) => {  return x + y;  }; |

## **2. Cú pháp Arrow function với tham số nhiều hơn 2**

Arrow function bắt đầu bằng dấu mở ngoặc tròn ( tiếp đến là các tham số và đóng ngoặc ) tiếp sau đó là mũi tên arrow => và phần cuối cùng là các dòng code mình xử lý (expression)

|  |
| --- |
| (p1, p2, ..., pn) => expression; |

Ví dụ như anh có dòng mã sắp xếp thứ tự giảm dần như sau. Nếu ta ko dùng arrow function

|  |
| --- |
| let numbers = [4,2,6];  numbers.sort(function(a,b){  return b - a;  });  console.log(numbers); // [6,4,2] |

Bây giờ chúng ta sẽ viết lại code ở trên bằng arrow function thì các em sẽ thấy nó ngắn gọn hơn như sau

|  |
| --- |
| let numbers = [4,2,6];  numbers.sort((a,b) => b - a);  console.log(numbers); // [6,4,2] |

## **3. Cú pháp Arrow function với 1 tham số**

Trường hợp arrow function chỉ có 1 tham số thì ta khai báo như sau bắt đầu bằng dấu ngoặc tròn ( tiếp đến là tham số và kết thúc bằng dấu đóng ngoặc tron ) sau đó là dấu mũi tên => và cuối cùng là phần code xử lý (statement)

|  |
| --- |
| (p1) => { statements } |
| let names = ['John', 'Mac', 'Peter'];  let lengths = names.map(name => name.length);  console.log(lengths); |

## **4. Cú pháp Arrow function với không tham số**

|  |  |
| --- | --- |
| () => { statements } |  |

Ví dụ như sau

|  |
| --- |
| let logDoc = () => console.log(window.document);  logDoc(); |

## **5. Sự khác nhau giữa Arrow function và function bình thường**

Sự khác nhau về cách dùng từ khoá this. Anh có ví dụ sau đây, chúng ta có lớp Car và phương thức speedUp như sau.

|  |
| --- |
| function Car() {  this.speed = 0;  this.speedUp = function (speed) {  this.speed = speed;  setTimeout(function () {  console.log(this.speed); // undefined  }, 1000);  };  }  let car = new Car();  car.speedUp(50); // undefined |

Khi chương trình chạy đến hàm setTimeout và khi ta console.log thì nhận giá trị speed là undefine. Như vậy this.speed trong hàm console.log của phương thức setTimeout không thể lấy được giá trị tham số speed trong hàm speedUp. Cái này là do phạm vi biến speed chỉ không thể truy cập và thấy được ở hàm console.log.

Các em có thể đọc kỹ hơn ở bài viết phạm vi của biến gồm global, local, lexical scope etc. Cái ví dụ ở trên là thuộc vào phạm vi lexical scope.

Chúng ta sẽ sửa lại code ở trên bằng cách khai báo thêm biến let self = this trước khi thực hiện gọi hàm setTimeout như sau

|  |
| --- |
| function Car() {  this.speed = 0;  this.speedUp = function (speed) {  this.speed = speed;  let self = this;  setTimeout(function () {  console.log(self.speed);  }, 1000);  };  }  let car = new Car();  car.speedUp(50); // 50; |

Không giống như function bình thường nếu chúng ta sử dụng arrow method cho đoạn code trên thì ta không cần lo lắng về phạm vi (scope) hoặc showdows . Đoạn code mới được viết như sau

|  |
| --- |
| function Car() {  this.speed = 0;  this.speedUp = function (speed) {  this.speed = speed;  setTimeout(  () => console.log(this.speed),  1000);  };  }  let car = new Car();  car.speedUp(50); // 50; |

# Khi nào không nên dùng Arrow function trong ES6

## **1. Bắt và xử lý sự kiện**

Anh có ví dụ sau đây, người dùng sẽ điền giá trị vào input sau đó anh lấy giá trị đó hiển thị ra ở một thẻ div như sau

Anh tạo một thẻ input để lấy tên người dùng.

|  |
| --- |
| <input type="text" name="username" id="username" placeholder="Enter a username"> |

Anh tạo một thẻ div để hiển thị tên người dùng lấy được từ input

|  |
| --- |
| <div id="greeting"></div> |

Anh viết đoạn mã javascript để lấy tên người dùng và gán vào thẻ div như sau

|  |
| --- |
| const greeting = document.querySelector('#greeting');  const username = document.querySelector('#username');  username.addEventListener('keyup', () => {  greeting.textContent = 'Hello ' + this.value;  }); |

Tuy nhiên khi chương trình chạy anh sẽ nhận được kết quả là ‘Hello’ undefined. Có nghĩa là this.value lấy không được giá trị trong hàm bắt sự kiện addEventListener. Như đã nói trong phần mở đầu arrow function không sở hữu phạm vi biến this của input, chính vì vậy mặc dùng ta ghi this.value thì ta ngầm định this này chính là của input nhưng thật sư không phải.

Để sửa lại đoạn code trên có thể bắt được sự kiện từ input thì ta phải dùng cách viết bình thường của javascript không dùng arrow function như sau.

|  |
| --- |
| username.addEventListener('keyup', function () {  input.textContent = 'Hello ' + this.value;  }); |

## **2. Các phương thức trong đối tượng**

Anh ví dụ có đối tượng Counter như sau.

|  |
| --- |
| const counter = {  count: 0,  next: () => ++this.count,  current: () => this.count  }; |

Đối tượng counter có 2 phương thức là next và current. Phương thức next để lấy giá trị tiếp theo và current là lấy giá trị hiện tại. Anh sẽ gọi phương thức next của đối tượng counter như sau.

|  |
| --- |
| console.log(counter.next()); |

Như vậy kết quả anh muốn muốn sẽ là 1. Tuy nhiên khi chương trình chạy anh sẽ nhận được kết quả là NaN. Nguyên nhân khi anh sử dụng this.count thì khi đó biến this này đang hiểu là đối tượng window chứ không phải là đối tượng counter. Để tránh biến this tham chiếu tới phạm vi toàn cục (global) thì ta sửa lại đoạn code trên như sau.

|  |
| --- |
| const counter = {  count: 0,  next() {  return ++this.count;  },  current() {  return this.count;  }  }; |

Chúng ta bỏ khai báo arrow function đi và viết theo cách viết bình thường.

## **3. Các phương thức trong Class**

Tương tụ như trường hợp phương thức trong đối tượng thì phương thức trong class chúng ta cũng không dùng arrow function.

|  |
| --- |
| class Counter {  constructor() {  this.count = 0;  }  next = () => {  return ++this.count;  }  current = () => {  return this.count;  }  } |

Chúng ta không khai báo theo kiểu arrow function trong Class mà khai báo theo cách sau đây.

|  |
| --- |
| class Counter {  constructor() {  this.count = 0;  }  next() {  return ++this.count;  }  current() {  return this.count;  }  } |

## **4. Phương thức prototype**

Chúng ta lấy ví dụ về prototype cho đối tượng counter như sau

|  |
| --- |
| function Counter() {  this.count = 0;  }  Counter.prototype.next = () => {  return this.count;  };  Counter.prototype.current = () => {  return ++this.next;  } |

Thì giá trị this.next và this.current đang tham chiếu tới đối tượng cục bộ (global) chứ không phải đối tượng counter. Chúng ta sẽ modify đoạn code trên lại như sau để this có thể tham chiếu đến counter.

|  |
| --- |
| function Counter() {  this.count = 0;  }  Counter.prototype.next = function () {  return this.count;  };  Counter.prototype.current = function () {  return ++this.next;  } |

## **5. Sử dụng tham số là đối tượng arguments**

Anh có ví dụ sau

|  |
| --- |
| const concat = (separator) => {  let args = Array.prototype.slice.call(arguments, 1);  return args.join(separator);  } |

Thay vì dùng theo arrow function thì ta nên thay đổi lại như sau

|  |
| --- |
| function concat(separator) {  let args = Array.prototype.slice.call(arguments, 1);  return args.join(separator);  } |

# Sử dụng map trong ES6

## **1. Giới thiệu về Map**

ES6 cung cấp một dạng tập hợp mới gọi là Map. Map sẽ lưu các phần tử theo KEY và VALUE. Cú pháp tạo Map trong ES6 như sau

|  |
| --- |
| let map = new Map([iterable]); |

Ví dụ chúng ta tạo một map trong ES6 như sau.

|  |
| --- |
| let students = new Map(); |

## **2. Các phương thức có trong Map**

Một số phương thức hữu ích của map là :

clear() : xoá các phần tử trong map.

delete(key) : xoá phần tử xát định, dựa trên giá trị key.

entries() : trả về một tập hợp

get(key) : lấy phần tử xát định, dựa vào giá trị key.

has(key) : kiểm tra xem giá trị key có tồn tại hay không. Nếu có giá trị key trong tập hợp sẽ trả về giá trị true.

keys() : trả về một iterator (tập hợp) tất cả các KEY có trong map.

set(key,value) : thêm giá trị cho key trong tập hợp.

values() : trả về một iterator chứa tất cả các giá trị có map.

## **3. Ví dụ các phương thức trong Map**

Tạo đối tượng Map. Giả sử như anh có danh sách các users (người dùng như sau)

|  |
| --- |
| let john = {name: 'John Doe'},  lily = {name: 'Lily Bush'},  peter = {name: 'Peter Drucker'}; |

Bây giờ anh muốn tạo quyền cho mỗi users bằng cách dùng map như sau.

|  |
| --- |
| let userRoles = new Map(); |

Tiếp đến ta set quyền cho user như sau.

|  |
| --- |
| userRoles.set(john, 'admin'); |

Chúng ta sử dụng phương thức set để thiết lập john tương ứng với role là admin. Tiếp đến chúng ta có thể set quyền cho lily và peter như sau.

|  |
| --- |
| userRoles.set(lily, 'editor')  .set(peter, 'subscriber'); |

Ngoài cách làm ở trên chúng ta có thể truyền một iterable khi khởi tạo đối tượng map như sau.

|  |
| --- |
| let userRoles = new Map([  [john, 'admin'],  [lily, 'editor'],  [peter, 'subscriber']  ]); |

Bây giờ nếu chúng ta muốn xem role của john là gì thì sử dụng phương thức get như sau.

|  |
| --- |
| userRoles.get(john); // admin |

Nếu chúng ta truyền vào KEY mà không có trong Map thì sẽ nhận được giá trị undefined

|  |
| --- |
| let foo = {name: 'Foo'};  userRoles.get(foo); //undefined |

Kiểm tra xem key có tồn tại hay không bằng phương thức has như sau.

|  |
| --- |
| userRoles.has(foo); // false  userRoles.has(lily); // true |

Để lấy kích thướt của map thì ta sử dụng phương thức size như sau

|  |
| --- |
| console.log(userRoles.size); // 3 |

Để duyệt qua các phần tử ta sử dụng phương thức keys. Ví dụ như anh muốn duyệt qua các user như sau.

|  |
| --- |
| for (let user of userRoles.keys()) {  console.log(user.name);  }  // John Doe  // Lily Bush  // Peter Drucker |

Để duyệt qua các role có trong map anh thực hiện như sau

|  |
| --- |
| for (let role of userRoles.values()) {  console.log(role);  }  // admin  // editor  // subscriber |

Để duyệt qua các phần tử gồm có key và value trong map thì anh sử dụng phương thức entries() như sau.

|  |
| --- |
| for (let elem of userRoles.entries()) {  console.log(`${elem[0].name}: ${elem[1]}`);  }  // John Doe: admin  // Lily Bush: editor  // Peter Drucker: subscriber |

Chúng ta có thể áp dụng destruction để viết code như sau nhưng vẫn cho ra kết quả như trên

|  |
| --- |
| for (let [user,role] of userRoles.entries()) {  console.log(`${user.name}: ${role}`);  } |

Chúng ta cũng có thể dùng vòng lặp forEach để duyệt qua các phần tử trong map như sau.

|  |
| --- |
| userRoles.forEach((role, user) =>  console.log(`${user.name}: ${role}`)  ); |

Chuyển đổi (convert) các KEY hoặc VALUE trong map thành mảng (Array) bằng cách sử dụng toán tử spread như sau.

|  |
| --- |
| var keys = [...userRoles.keys()];  console.log(keys);  kết quả  [ { name: 'John Doe' },  { name: 'Lily Bush' },  { name: 'Peter Drucker' } ] |

Để convert các Value qua mảng ta làm như sau

|  |
| --- |
| var roles = [...userRoles.values()];  console.log(roles);  [ 'admin', 'editor', 'subscriber' ] |

Để xoá một phần tử trong map ta sử dụng phương thức delete như sau.

|  |
| --- |
| userRoles.delete(john); |

Để xoá hết các phần tử ta sử dụng phương thức clear như sau.

|  |
| --- |
| userRoles.clear(); |
| Sử dụng set trong ES6**1. Giới thiệu về Set** ES6 cung cấp cho chúng ta một tập hợp mới gọi là SET. Tập hợp này lưu trữ các giá trị không trùng lặp. Cú pháp để tạo Set như sau   |  | | --- | | let setObject = new Set(); |  **2. Các phương thức phổ biến của Set** add(value) : thêm phần tử vào tập hợp.  clear() : xoá hết các phần tử trong set.  delete(value) : xoá một phần tử trong tập hợp.  entries() : trả về một Iterator chứa các giá trị của một tập hợp.  has(value) : kiểm tra xem giá trị có tồn tại trong tập hợp chưa. Trả về true nếu tồn tại.  ## 3. Ví dụ Set  Giả sử anh có một Set gồm các phần tử là ký tự như sau.   |  |  | | --- | --- | | 1 | let chars = new Set(['a', 'a', 'b', 'c', 'c']); |   Vì Set chỉ chứa các giá trị không trùng lặp. Do vậy khi anh console.log thì chỉ có giá trị a và c.   |  | | --- | | console.log(chars);  Kết quả là :  Set { 'a', 'b', 'c' } |   Để lấy kích thướt của Set anh sẽ dùng hàm size như sau   |  | | --- | | let size = chars.size;  console.log(size);// 3 |   Để thêm một phần tử vào thì anh sử dụng hàm add như sau   |  | | --- | | chars.add('d');  console.log(chars);  Kết quả là :  Set { 'a', 'b', 'c', 'd' } |   Chúng ta có thể add nhiều phần tử liên tiếp như sau.   |  | | --- | | chars.add('e')  .add('f');  Kết quả là :  Set { 'a', 'b', 'c', 'd','e','f' } |   Kiểm tra giá trị đã tồn tại trong Set chưa ta dùng phương thức has như sau.   |  | | --- | | let exist = chars.has('a');  console.log(exist);// true |   Để xoá phần tử ra khỏi tập hợp Set ta dùng hàm delete như sau.   |  | | --- | | chars.delete('f');  console.log(chars); // Set {"a", "b", "c", "d", "e"} |   Để xoá hết các phần tử ta dùng hàm clear như sau   |  | | --- | | chars.clear();  console.log(chars); // Set{} |   Để duyệt qua các phần tử ta dùng vòng lặp for of như sau.  Giả sử ta có tập hợp roles như sau.   |  | | --- | | let roles = new Set();  roles.add('admin')  .add('editor')  .add('subscriber'); |   Ta sử dụng vòng lặp for of để duyệt qua các phần tử như sau.   |  | | --- | | for (let role of roles) {  console.log(role);  }  Kết quả nhận được là :  admin  editor  subscriber |   Set cũng hỗ trợ các phương thức như keys(), values(), entries() như Map để duyệt qua các phần tử.   |  | | --- | | for (let [key, value] of roles.entries()) {  console.log(key === value);  }  kết quả nhận được :  true  true  true | |

# Sử dụng mảng trong ES6

## **1. Giới thiệu về phương thức Array.of**

Trong ES5 phiên bản củ, khi chúng ta khai báo một mảng với độ dài của mảng, thì javascript sẽ tạo ra một mảng có kích thước tương ứng với độ dài truyền vào nhưng chỉ chứ 1 phần tử

|  |
| --- |
| let numbers = new Array(2);  console.log(numbers.length); // 2  console.log(numbers[0]); // undefined |

Ở ví dụ trên ta khai báo mảng có 2 phần tử nhưng khi lấy vị trí đầu tiên ta nhận được giá trị undefined. Cái này gây ra khó hiểu vì khi mình khai báo mảng có 2 phần tử Array(2) nhưng khi lấy phần tử đầu tiên lại không có.

Trường hợp thứ 2, khi ta truyền tham số vào là chuỗi chứ không phải number. Thì javascript sẽ tạo ra một mảng có phần tử đầu tiên là kích thướt của mảng như sau.

|  |
| --- |
| numbers = new Array("2");  console.log(numbers.length); // 1  console.log(numbers[0]); // "2" |

Để cho việc khai báo mảng một cách rõ ràng hơn thì trong ES6 cung cấp cho chúng ta phương thức mới, đó là array.of với cú pháp như sau :

|  |
| --- |
| Array.of(element0[, element1[, ...[, elementN]]]) |

Array.of cũng tương tự như Array tuy nhiên khi tạo mảng bằng Array.of thì luôn luôn có 1 giá trị trong mảng. Ví dụ như sử dụng Array.of như sau :

|  |
| --- |
| let numbers = Array.of(3);  console.log(numbers.length); // 1  console.log(numbers[0]); // 3 |

Trong ví dụ trên ta truyền số lượng mảng là 3 thì phương thức array.of sẽ tạo một array có 1 phần tử

|  |
| --- |
| let chars = Array.of('A', 'B', 'C');  console.log(chars.length); // 3  console.log(chars); // ['A','B','C'] |

Trong ví dụ trên ta tạo ra mảng có 3 phần tử A,B,C thì kích thướt của mảng sẽ là 3.

## **2. Giới thiệu về phương thức Array.from**

Trong phiên bản ES5 để duyệt qua các phần tử trong mảng ta làm như sau:

|  |
| --- |
| function arrayFromArgs() {  var results = [];  for (var i = 0; i < arguments.length; i++) {  results.push(arguments[i]);  }  return results;  }  var fruits = arrayFromArgs('Apple', 'Orange', 'Banana');  console.log(fruits);  Kết quả  [ 'Apple', 'Orange', 'Banana' ] |

Trong phiên bản ES6 thì chúng ta sẽ sử dụng chức năng mới là Array.from để duyệt các phần tử như sau

|  |
| --- |
| function arrayFromArgs() {  return Array.from(arguments);  }  console.log(arrayFromArgs(1, 'A'));  Kết quả  [ 1, 'A' ] |

Chúng ta có thể sử dụng Array.from với hàm mapping như sau

|  |
| --- |
| function addOne() {  return Array.from(arguments, x => x + 1);  }  console.log(addOne(1, 2, 3));  Kết quả  [ 2, 3, 4 ] |

Sử dụng Array.from với This.

|  |
| --- |
| let doubler = {  factor: 2,  double(x) {  return x \* this.factor;  }  }  let scores = [5, 6, 7];  let newScores = Array.from(scores, doubler.double, doubler);  console.log(newScores);  Kết quả  [ 10, 12, 14 ] |

## **3. Giới thiệu về phương thức Array.find**

Trong ES5 phiên bản cũ chúng ta muốn tìm kiếm giá trị trong mảng thì sử dụng hàm indexOf() và lastIndexOf() để tìm vị trí của các phần tử cần tìm trong mảng. Trong phiên bản mới ES6 thì hỗ trợ cho chúng ta phương thức find.

Cú pháp như sau :

|  |
| --- |
| find(callback(element[, index[, array]])[, thisArg]) |

Ví dụ như ta có một mảng khách hàng có tên khách hàng và thẻ tín dụng như sau.

|  |
| --- |
| let customers = [{  name: 'ABC Inc',  credit: 100  }, {  name: 'ACME Corp',  credit: 200  }, {  name: 'IoT AG',  credit: 300  }]; |

Để tìm Khách hàng có thẻ tín dụng lớn hơn 100 ta làm như sau.

|  |
| --- |
| console.log(customers.find(c => c.credit > 100));  Kết quả nhận được là  { name: 'ACME Corp', credit: 200 } |

# Sử dụng Object assign trong ES6

## **1. Giới thiệu về phương thức Object assign**

Chúng ta sử dụng phương thước Object.assign để copy hoặc merge tất cả thuộc tính của một đối tượng này qua một đối tượng khác. Cú pháp như sau.

|  |  |
| --- | --- |
| 1 | Object.assign(target, ...sources) |

Sources : đối tượng sẽ ta sẽ copy tất cả các thuộc tính Target : đối tượng mới nhận được các giá trị copy từ source.

Anh có ví dụ như sau.

|  |
| --- |
| let widget = {  color: 'red'  };  let clonedWidget = Object.assign({}, widget);  console.log(clonedWidget);  Kết quả  { color: 'red' } |

Như vậy ta copy các thuộc tính của đối tượng widget sang đối tượng clonedWidget. Đối tượng cloneWidget sẽ có thuộc tính color bằng red giống như đối tượng widget.

## **2. Merge 2 đối tượng**

Chúng ta có thể dùng object.assign() để trộn 2 đối tượng lại với nhau như sau

|  |
| --- |
| let box = {  height: 10,  width: 20  };  let style = {  color: 'Red',  borderStyle: 'solid'  };  let styleBox = Object.assign({}, box, style);  console.log(styleBox);  Kết quả  {  height: 10,  width: 20,  color: 'Red',  borderStyle: 'solid'  } |

Trường hợp nếu merge 2 đối tượng có cùng thuộc tính. Thì giá trị của thuộc tính được lấy bởi đối tượng sau cùng như sau.

|  |
| --- |
| let box = {  height: 10,  width: 20,  color: 'Red'  };  let style = {  color: 'Blue',  borderStyle: 'solid'  };  let styleBox = Object.assign({}, box, style);  console.log(styleBox);  Kết quả  {  height: 10,  width: 20,  color: 'Blue',  borderStyle: 'solid'  } |

Như ta thấy 2 đối tượng box và style đều có giá trị color. Tuy nhiên khi merge 2 đối tượng thì giá trị color được lấy theo đối tượng sau cùng. Trong trường hợp này chúng ta sẽ lấy giá trị là color = Blue chứ không phải lấy giá trị color = Red.

# Sử dụng Symbol trong ES6

## **1. Giới thiệu về symbol**

ES6 cung cấp cho chúng ta 1 kiểu dữ liệu nguyên thuỷ mới gọi là Symbol. Nó không giống như kiểu integer, string, boolean và các kiểu khác. Cú pháp tạo Symbol như sau.

|  |
| --- |
| let s = Symbol('foo'); |

Symbol mỗi lần được tạo ra sẽ nhận được giá trị là duy nhất và khác nhau. Ví dụ khi chúng ta so sánh

|  |
| --- |
| console.log(Symbol() === Symbol()); // false |

Chúng ta sẽ thấy hai giá trị Symbol sinh ra là khác nhau.

Kiểu dữ liệu Symbol cho phép chúng ta thêm các giá trị mô tả cho Symbol. Ví dụ anh thêm mô tả là first name cho biến firstName như sau.

|  |
| --- |
| let firstName = Symbol('first name'),  lastName = Symbol('last name'); |

Chúng ta có thể lấy giá trị môt tả của Symbol bằng phương thức toString().

|  |
| --- |
| console.log(firstName); // Symbol(first name)  console.log(lastName); // Symbol(last name) |

Hàm console.log sẽ gọi phưuong thức toString() của Symbol.

Symbol là kiểu dữ liệu nguyên thuỷ nên chúng ta không sử dụng toán tử new để khai báo Symbol.

|  |
| --- |
| let s = new Symbol(); // error |

## **2. Mục đích sử dụng symbol**

Chúng ta sử dụng Symbol khi chúng ta muốn có một giá trị duy nhất trong hệ thống.

Ví dụ trong đoạn code mình có các kiểu String hoặc Integer mà chứa giá trị duy nhất thì ta nên thay qua bằng cách dùng Symbol. Anh ví dụ như mình có biến status lưu các trạng thái là open, in progress, closed, completed etc. Thì ta sử dụng Symbol như sau

|  |
| --- |
| let statuses = {  OPEN: Symbol('Open'),  IN\_PROGRESS: Symbol('In progress'),  COMPLETED: Symbol('Completed'),  HOLD: Symbol('On hold'),  CANCELED: Symbol('Canceled')  };  // complete a task  task.setStatus(statuses.COMPLETED); |

Chúng ta sử dụng Symbol khi muốn thực hiện các tính toán thuộc tính của một đối tượng.

|  |
| --- |
| let status = Symbol('status');  let task = {  [status]: statuses.OPEN,  description: 'Learn ES6 Symbol'  };  console.log(task); |

# Sử dụng Generator trong ES6

## **1. Giới thiệu về Generator**

Các dòng code trong phương thức javascript được thực thi từ trên xuống dưới và nó không thể dừng lại giữ chừng. Anh lấy ví dụ về phương thức foo() như sau

|  |
| --- |
| function foo() {  console.log('I');  console.log('cannot');  console.log('pause');  } |

Khi chương trình chạy anh sẽ nhận được kết quả là I cannot pause.

ES6 cung cấp một loại phương thức kiểu mới giúp chúng ta có thể dừng lại (pause) bất kỳ đâu trong phương thức. Nó được gọi là generator, anh có ví dụ về generator như sau

|  |
| --- |
| function\* generate() {  console.log('invoked 1st time');  yield 1;  console.log('invoked 2nd time');  yield 2;  } |

Đầu tiên để khai báo 1 phương thức là generator anh sẽ sử dụng đấu \* ở phía sau chữ function sau đó là tên phương thức function\* generate. Sau khi in ra console.log đầu tiên thì anh sử dụng 1 từ khoá thứ 2 đó là yield cái này sẽ trả về giá trị tại điểm đang dừng.

Như vậy khi tạo một phương thức generator chúng ta cần 2 từ khoá đó là \* và yield. Chúng ta gọi phương thức generator như sau.

|  |
| --- |
| let gen = generate(); |

Chúng ta sẽ console.log biến gen xem nó là gì ?

|  |
| --- |
| console.log(gen); |

Kết quả ta nhận được sẽ là Object [Generator] {} . Như vậy biến gen của ta làm một đối tượng Generator. Để lấy giá trị dừng đầu tiên ta sử dụng phương thức next().

|  |
| --- |
| let result = gen.next();  console.log(result);  Kết quả là  invoked 1st time  { value: 1, done: false } |

Như vậy ta lấy được kết quả mà phương thức generate() dừng tại vị trí đầu tiên. Nếu muốn lấy tiếp giá trị dừng tiếp theo chúng ta sử dụng tiếp phương thức next().

|  |
| --- |
| result = gen.next();  console.log(result);  Kết quả là  invoked 2nd time  { value: 2, done: false } |

Nếu chúng ta next thêm 1 lần nữa thì sẽ nhận được kết quả undefined vì phương thức generate() chỉ dừng lại có 2 lần (yield)

|  |
| --- |
| result = gen.next();  console.log(result);  Kết quả là  { value: undefined, done: true } |

## **2. Duyệt qua Generator**

Generator là một đối tượng iterable nên chúng ta có thể duyệt qua nó bằng vòng lặp for of như sau.

|  |
| --- |
| for (const g of gen) {  console.log(g);  }  kết quả là  invoked 1st time  1  invoked 2nd time  2 |

# Sử dụng Promise trong ES6

## **1. Giới thiệu về promise**

Trong Javascript thì promise là một đối tượng (object) được trả về từ phương thức nhưng giá trị này bạn hy vọng sẽ nhận được trong tương lai chứ không phải lúc gọi hàm sẽ nhận được ngay. Bởi vì giá trị promise sẽ được trả về trong tương lai nên nó rất thích hợp với các tính năng không đồng bộ hoặc gọi cách khác là bất đồng bộ (asynchronous).

Đồng bộ (synchronous) có nghĩa là làm tuần tự theo một quy trình làm bước A xong mới được làm bước B và C. Còn không đồng bộ (asynchronous) thì không quan tâm thứ tự có thể làm A và B cùng lúc. Cái nào xong trước thì mình sẽ nhận giá trị trước, mình có thể làm song song nhiều việc cùng 1 lúc.

Ví dụ đơn giản về Promise như, Anh muốn hoàn thành khoá học javascript trong tháng tới. Như vậy trong tương lai tháng tới anh không biết chắc là mình sẽ có đủ thời gian và nổ lực để hoàn thành khoá học hay không. (Promise) Đó dạng giống như một hy vọng là anh có thể hoàn thành được khoá học.

Promise có 3 trạng thái

Pending : anh không biết là anh có thể hoàn thành khoá học Javascript trong tháng tới không

Fullfilll : anh đã hoàn thành khoá học javascript trong tháng tới

Rejected : anh không học javascript nữa và bỏ luôn ý định hoàn thành trong tháng tới

Một promise sẽ bắt đầu bằng trạng thái Pending để nói rằng Promise đó chưa xong. Cuối cùng của một promise sẽ ở trạng thái Fullfill (hoàn thành) hoặc Rejected (bỏ qua không làm).

## **2. Tạo một promise**

Để tạo 1 promise chúng ta sử dụng new một đối tượng Promise trong ES6 như sau.

|  |
| --- |
| let completed = true;  let learnJS = new Promise(function (resolve, reject) {  if (completed) {  resolve("I have completed learning JS.");  } else {  reject("I haven't completed learning JS yet.");  }  }); |

Bước 1 : chúng ta tạo Promise bằng toán tử new new Promise. Trong hàm dựng (constructor) của Promise chúng ta định nghĩa 1 function có 2 tham số là resolve và reject như sau Promise(function (resolve, reject).

Bước 2 : khi mình chạy chương trình nếu mình gọi hàm resolve nghĩa là Promised đã hoàn thành và trả về kết quả (fullfil)

Bước 3 : khi mình chạy chương trình nếu mình gọi hàm reject nghĩa là Project không thành công (reject)

## **3. Ví dụ promise**

Anh có ví dụ như sau. Anh khai báo biến completed là true, tiếp đến anh gọi hàm setTime chờ 3s

|  |
| --- |
| let completed = true;  let learnJS = new Promise(function (resolve, reject) {  setTimeout(() => {  if (completed) {  resolve("I have completed learning JS.");  } else {  reject("I haven't completed learning JS yet.");  }  }, 3 \* 1000);  }); |

Khi chương trình chạy anh sẽ thấy promise ở trạng thái pending .

Sau khi chờ 3s thì anh sẽ nhận được kết quả là status của Promise là resolved và giá trị nhận được là “I have completed learning JS.”

Bây giờ nếu anh muốn thấy kết quả reject thì anh thay đổi giá trị complete thành false như sau.

|  |
| --- |
| let completed = false;  let learnJS = new Promise(function (resolve, reject) {  setTimeout(() => {  if (completed) {  resolve("I have completed learning JS.");  } else {  reject("I haven't completed learning JS yet.");  }  }, 3 \* 1000);  }); |

Khi chương trình chạy anh sẽ thấy promise ở trạng thái pending .

Sau khi chờ 3s thì anh sẽ nhận được kết quả là status của Promise là resolved và giá trị nhận được là “I haven’t completed learning JS yet.”

Như vậy từ khoá resolve sẽ chuyển một Promise về trạng thái fullfil và reject sẽ chuyển promise về trạng thái rejected

## **4. Lấy giá trị từ promise**

Promise là một đối tượng được trả về cho nên để lấy được giá trị có trong Promise chúng ta sử dụng phương thức then() với cú pháp như sau.

|  |
| --- |
| promiseObject.then(onFulfilled, onRejected);promiseObject.then(onFulfilled, onRejected); |

Anh có ví dụ như sau. Anh sẽ tạo ra một Promise

|  |
| --- |
| function makePromise(completed) {  return new Promise(function (resolve, reject) {  setTimeout(() => {  if (completed) {  resolve("I have completed learning JS.");  } else {  reject("I haven't completed learning JS yet.");  }  }, 3 \* 1000);  });  } |

Anh sẽ lấy kết quả của Promise trả về bằng phương thức then()

|  |
| --- |
| let learnJS = makePromise(true);  learnJS.then(  success => console.log(success),  reason => console.log(reason)  ); |

Trong trường hợp khi Promise bị reject chúng ta có thể dùng phương thức catch để bắt lại đối tượng Promise như sau

|  |
| --- |
| learnJS.catch(  reason => console.log(reason)  ); |

Thỉnh thoảng nếu chúng ta muốn thực hiện một đoạn code cho dù Promise thành công hay không thành công thì ta sử dụng finally như sau.

|  |
| --- |
| function createApp() {  // ...  }  learnJS  .then(success => console.log(success))  .catch(reason => console.log(reason))  .finally(() => createApp()); |