Javascript Lecture Notes

Dynamic type

Javascript là một ngôn ngữ có kiểu dữ liệu động. Tức một biến có thể có nhiều kiểu dữ liệu khác nhau.

```
let a = 10;
a = 'a';
a = 'Hello a';
a = function () {
  return 'a';
}
a = {
  a: 'b',
}
a = ['a']
```

`let` và `const`

- Let dùng để khai báo biến có thể gán lại giá trị mới cho nó

```
let a = 10;
let b = {};
let c = [];

a = 20;
b = {
  name: 'Hello',
};
let c = 'No more array :)';
```

- Const dùng để khai báo biến và không thể gán lại giá trị mới cho nó

```
const a = 10;
const b = {};
const c = [];

a = 20; // lỗi :(
b = {}; // lỗi luôn :(
c = [a] // lỗi nốt ::((((
```

- Lưu ý, mặc dù const không gán lại giá trị, tuy nhiên những giá trị trong Object và Array vẫn thay đổi được. Vì const chỉ chặn việc **GÁN LẠI biến bằng giá trị khác**. Ba ví dụ sau đều không gán lại biết a và b nên tất cả đều hợp lệ

```
const a = {};
const b = [];

// xiếc javascript
a.b = 10; // không lỗi OvOb
b.push(20); // không lỗi nốt
b[0] = 30; // vẫn không lỗi :v

// a = [] mới lỗi nè
// b = {} cũng sẽ lỗi
```

Scope

- Scope là phạm vi hoạt động của một biến hay một class
- Scope bắt đầu từ dấu `{` và kết thúc khi gặp dấu `}`

```
// bắt đầu scope
{
  let a = 10;
  function abc() {
    return 20;
  }
}
// scope kết thúc
```

- Mọi biến được tạo bằng `let` , `const` và những class được tạo ra từ từ khóa `class` sẽ có thời gian tồn tại từ lúc được khởi tạo tới khi kết thúc scope

```
// bắt đầu scope
{
  let a = 10; // bắt đầu tgian tồn tại của a
  const b = 30; // bắt đầu tgian tồn tại của b
  class ImHere { // bắt đầu tgian tồn tại của class ImHere

  }

  a = 100;
  console.log(a); // in ra 100
  console.log(b); // in ra 30
  const hello = new ImHere(); // ok luôn
}
// scope kết thúc
// kết thúc tgian tồn tại của a, b, ImHere

// Vì dẫd két thúc tgian tồn tại
// a, b, ImHere không thể sử dụng được nữa
a = 100; // lỗi :(
console.log(a); // Đỏ tiếp
console.log(b); // Lại đỏ
const hello = new ImHere(); // Bay màu
```

- Scope bên trong có thể truy cập các biến và hàm của scope bên ngoài nó
- Nhưng scope bên ngoài không thể truy cập các biến và hàm bên trong scope tồn tại bên trong nó

```
{
  let a = 'Xàm';
  const b = 'Nhảm';
  class C {}

  // tạo một scope con
  // bên trong một scope
  {
    // những biến thuộc scope bên ngoài
    // đều có thể dùng được với scope ở bên trong
    a = 'Xàm xí';
    const d = `${b} nhí`;
    const cc = new C();
    console.log(a); // Xàm xí
  } // giết sạch d và cc

  console.log(a); // Xàm xí

  // tuy nhiên các biến thuộc scope con của scope hiện tại
  // sẽ không thể truy cập được từ scope cha :((
  // vì nó chết rồi còn đâu
  console.log(d); // Lỗi do d đã chết mẹ r :((
  console.log(cc); // cc cũng hi sinh theo
}
```

String, String template (String literal)

- String trong Javascript là một kiểu dữ liệu bắt đầu và kết thúc bằng cặp dấu `''` hoặc `""`
- Hai string có thể ghép lại với nhau bằng dấu `+`. Tuy nhiên không thể dùng dấu `-` hay các toán tử khác để làm phép toán với string

```
const str = 'hello';
const anotherStr = "world";

const hello = str + ' ' + anotherStr;
// hello world
console.log(hello);
```

- String template hay String literal thì đặc biệt hơn. Nó cũng là một kiểu String, tuy nhiên String template cho phép lập trình viên chèn biến của Javascript vào bên trong nó.
- Cú pháp để khởi tạo một String template ha String literal là dùng cặp `` `String` `` (Nút kế số 1, bên tay trái, nhấn Shift và nhấn nút đó)
- Cách dùng biết trong một string template là dùng cú pháp `${biến}`

```
const name = 'Tai';
const hello = `Hello ${name}`;

// hello Tai;
console.log(hello);
```

Object

- Object là một cấu trúc dữ liệu để **lưu trữ những giá trị có kiểu dữ liệu khác nhau**. Giúp dễ quản lý thông tin thuộc về một đối tượng nào đó.
- Để khởi tạo một mảng, ta dùng cặp ngoặc nhọn `{}`
- Để khai báo những giá trị cần lưu trong mảng, ta dùng cặp `key: value` trong mảng

```
const obj = {
  age: 10,
}
console.log(obj.age); // 10
```

- Để lấy một giá trị từ object ra, ta dùng cú pháp `obj.key` hoặc `obj['key']`

```
const obj = {
  name: 'Long',
  score: 1000000000000,
  increaseScore: function() {
    obj.score += 1;
  }
}

console.log(obj.['name']); // Long
console.log(obj.score); // 1000000000000
console.log(obj.increaseScore());
console.log(obj.['increaseScore']());
console.log(obj.score); // 1000000000002
```

- Để thêm một key cho obj, ngoài khởi tạo từ bạn đầu, ta có thể dùng cú pháp lấy một giá trị từ object ra. Rồi gán một giá trị trực tiếp cho key đó

```
const obj = {};
const keyName = 'isMentor';
obj.age = 10;
obj['name'] = 'Long';
obj[keyName] = true;

console.log(obj.age); // 10
console.log(obj.name); // Long
console.log(obj.isMentor); // true
```

- Lưu ý, phải gán trực tiếp như ví dụ trên. Nếu bạn làm như sau. Thì obj sẽ không được cập nhật giá trị `age = 10`

```
const age = obj.age;
age = 10;
```

- Các key trong object tuyệt đối không được trùng nhau. Nếu xảy ra tình trạng trùng nhau, key trùng được khai báo sau cùng sẽ thay thế key trước đó

```
const obj = {
  a: 1,
  b: 2,
  c: 3,
  a: 4, // trùng nè
}

console.log(obj.a); // in ra 4
```

Array

- Mảng là một cấu trúc dữ liệu để lưu trữ những giá trị có chung kiểu dữ liệu với nhau.
- Array là một Object đặc biệt. Đặc biệt ở chỗ Object này chỉ chứa những cặp key: value, với value có kiểu dữ liệu giống nhau. Và key của Object này gọi là index, được tự động đánh số bắt đầu từ 0
- Vì Array là một Object đặc biệt, nên Javascript cũng cho nó một cú pháp đặc biệt để khởi tạo nó. Đó là muốn khởi tạo một Array, thay vì dùng cặp ngoặc nhọn `{}` thì bạn sẽ dùng cặp ngoặc vuông `[]`

```
const arr1 = [];
const arr2 = [1, 2, 3, 2, 1]; // array of number
const arr3 = ['a', 'ab', 'abc', 'a']; // array of string
const arr4 = [ // array of object
  {
    name: 'a',
  }
  {
    name: 'b',
  }
  {
    name: 'b',
  }
];
```

- Vì key của Array là một số. Nên ta không dùng cú pháp array.0 array.1 được. Đơn giản là vì JAvascript không cho phép điều đó. Nên ta phải dùng cú pháp thứ 2 của obj để lấy lấy value là bằng ngoặc vuông.

```
const arr = [1, 2, 3];
console.log(arr[1]) // 2
console.log(arr[0]) // 1, do index bắt đầu = 0
console.log(arr[2]) // 3
```

- Tương tự để gán giá trị mới cho một index bất kỳ, ta dùng như bên object

```
const arr = [1, 2, 3];
console.log(arr[0]); // 1
arr[0] = 100;
console.log(arr[0]); // 100
```

- Để duyệt qua các phần tử trong mảng, ta có thể dùng một vòng lặp. Đơn giản là dùng vòng lặp for

```
arr = [1, 2, 3]
for (let i = 0; i < arr.length; i++) {
  console.log(a[i]);
}
// in ra:
// 1
// 2
// 3
```

Các hàm cơ bản của Array

- **push(value, value2, value3, ...)**: Đưa một hoặc nhiều giá trị mới vào **cuối** mảng.

```
a = [];
a.push(1, 2, 3, 4, 5);
console.log(a); // [1, 2, 3, 4, 5]
a.push(6);
a.push(7);
a.push(8);
a.push(9);
// 4 dòng trên có thể gom thành a.push(6, 7, 8, 9);
console.log(a); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **pop()**: pop không nhận bất kỳ tham số nào. Nó chỉ đơn thuần sẽ lấy phần tử ở **cuối** mảng, return về, và xóa phần tử đó ra khỏi mảng

```
const a = [1, 2, 3, 4, 5, 6];
const e = a.pop();
console.log(e); // 6
console.log(a); // [1, 2, 3, 4, 5]
a.pop();
console.log(a); // [1, 2, 3, 4]
```

- **unshift(value1, value2, value3, ...)**: Đưa một hoặc nhiều giá trị mới vào **đầu** mảng.

```
a = [4, 5, 6];
a.shift(1, 2, 3);
console.log(a); // [1, 2, 3, 4, 5, 6]
a.shift(0);
console.log(a); // [0, 1, 2, 3, 4, 5, 6]
```

- **shift()**: shift không nhận bất kỳ tham số nào. Nó chỉ đơn thuần sẽ lấy phần tử ở **đầu** mảng, return về, và xóa phần tử đó ra khỏi mảng

```
const a = [1, 2, 3, 4, 5, 6];
const e = a.shift();
console.log(e); // 1
console.log(a); // [2, 3, 4, 5, 6]
a.shift();
console.log(a); // [3, 4, 5, 6]
```

- **indexOf(value)**: Tìm index của một value trong mảng. Trả về index tìm được với value cung cấp. Index này sẽ là **index nhỏ nhất tìm được**. Nếu không tìm thấy, trả về -1

```
const a = [1, 2, 3, 4, 5, 6];
const indexOf3 = a.indexOf(3);
console.log(indexOf3); // 2
```

```
const a = [1, 5, 3, 4, 5, 6];
const indexOf5 = a.indexOf(5);
console.log(indexOf5); // 1
// vì có 2 số 5, 1 ở vị trí 1, và 1 ở vị trí 4
// indexOf return về index nhỏ nhất -> return 1
```

```
const a = [1, 2, 3, 4, 5, 6];
const indexOf7 = a.indexOf(7);
console.log(indexOf7); // -1 vì không tìm thấy
// ----------------------------
```

- **concat(arr1, arr2, arr3, ...)**: Nối mảng arr với các mảng arr1, arr2, arr3, ... Sau đó gán mảng vừa được nối vào một biến và trả về cho người dùng. **Đặc biệt, hàm này sẽ trả về một mảng mới hoàn toàn và không ảnh hưởng tới chuỗi arr ban đầu**

```
const arr = [1, 2, 3];
const newArr = arr.concat([], [4], [5, 6], [7, 8, 9]);

console.log(arr); // vẫn là [1, 2, 3]
console.log(newArr);
// khi này newArr mới là [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Các hàm nâng cao của Array

map(function (element, index, arr) {})

Bây giờ ta có một mảng

```
a = [1, 2, 3, 4, 5]
```

Ta muốn với mỗi phần tử của mảng `arr` nhân đôi lên thì ta làm như thế nào? Đơn giản là dùng một vòng lặp rồi chỉnh sửa các phần tử của `arr` một cách trực tiếp

```
for (let i = 0; i < arr.length; i++) {
  a[i] = a[i] * 2;
}

console.log(a[i]) // [2, 4, 6, 8, 10]
```

Thì việc ta vừa làm gọi là biến đổi `a[i]` cũ sang một `a[i]` mới. Tuy nhiên có một khuyết điểm đó là nếu viết như code trên. Thì lỡ sau này ta muốn thay đổi không phải nhân 2, mà là cộng 10, trừ 1 hay biến mảng trên thành một mảng Component bên React. Khi vậy với mỗi yêu cầu ta phải làm như sau:

```
for (let i = 0; i < arr.length; i++) {
  a[i] = a[i] + 100;
}


for (let i = 0; i < arr.length; i++) {
  a[i] = a[i] - 1;
}


for (let i = 0; i < arr.length; i++) {
  a[i] = (
    <Number
      value={a[i]}
    />
  );
}
```

Dễ thấy rằng, ta chỉ sửa duy nhất ở ngay chỗ phía sau dấu `=` ở phần gán `a[i]`

```
...
a[i] = a[i] * 2; // thay đổi ở đây, chỗ a[i] * 2
```

Chỉ thay đổi chỗ xíu đó thôi, mà ta phải viết lại toàn bộ vòng for. Rồi gán lại. Như vậy cực kỳ mất tgian luôn. Thấy vậy, Javscript mới giúp làm ra một hàm làm sẵn để thu gọn công việc lại. Hàm đó gọi là **map**.

Hàm **map** đơn giản là bên trong nó sẽ y chang như trên. Sẽ có vòng for, sẽ gán `a[i]` bằng một giá trị mới. Tuy nhiên do map **không biết người dev sẽ muốn `a[i]` biến đổi ra sao**, như thế nào. Nên hàm map sẽ **nhận vào một tham số để giúp nó hiểu cách biến đổi `a[i]` sang `a[i]` mới**. Và tham số đó **là một hàm, nhận vào 3 tham số khác là element, index và array** chính nó. Tham số hàm này yêu cầu **return về giá trị `a[i]` mới dựa trên `a[i]` cũ** (tức element) để hàm map biết cách biến đổi `a[i]` cũ thành mới.

Khi biến đổi xong, hàm map sẽ trả về một mảng hoàn toàn mới và không làm thay đổi ảnh hưởng tới mảng cũ.

Bây giờ chúng ta sẽ xem cách hàm map hoạt động để hiểu hơn

```
class Array {
  map(func) {
    // lấy array ra từ this
    const arr = this.array;
    // copy ra một array mới
    const newArr = arr.concat([]);
    // còn phần này y chang luôn nè
    for (let i = 0; i < arr.length; i++) {
      // newArr?
      // đúng rồi, vì copy ra mảng mới có toàn bộ giá trị mảng cũ
      // nên ta biến đổi trên mảng mới sẽ không làm ảnh hưởng mảng cũ
      newArr[i] = func(array[i], i, arr);
      // array[i] = element
      // i = index
      // arr = array (dude!)
      // gọi func để xử lý phần sau dấu =
      // vì chỉ phần sau dấu = thay đổi
      // nên ta truyền vào một hàm func
      // để chỉ cho hàm map cách biến đổi a[i] cũ sang a[i] mới
    }
    // sau khi biến đổi array mới xong dựa trên array cũ
    // map sẽ trả về array được biến đổi cho người dùng sử dụng
    return newArr;
  }
}
```

Khi đã hiểu rồi thì bây giờ tập áp dụng thôi

```
arr = [1, 2, 3, 4, 5];
// không xài index với array nên bỏ luôn
const newArray = arr.map(function (element) {
  // chỉ cho map cách biến đổi a[i] cũ sang a[i] mới
  return element * 2;
});
console.log(arr); // [1, 2, 3, 4, 5]
console.log(newArray); // [2, 4, 6, 8, 10]
```

```
arr = [1, 2, 3, 4, 5];
// không xài index với array nên bỏ luôn
const newArray = arr.map(function (element) {
  // chỉ cho map cách biến đổi a[i] cũ sang a[i] mới
  return element + 1;
});
console.log(arr); // [1, 2, 3, 4, 5]
console.log(newArray); // [2, 3, 4, 5, 6]
```

```
arr = [1, 2, 3, 4, 5];
// không xài index với array nên bỏ luôn
const newArray = arr.map(function (element) {
  // chỉ cho map cách biến đổi a[i] cũ sang a[i] mới
  return (
    <Number
      value={a[i]}
    />
  );
});
console.log(arr); // [1, 2, 3, 4, 5]
console.log(newArray); // [C1, C2, C3, ...] Với C1 = component 1, ...
```

filter(function (element, index, arr) {})

Giống với **map**, **filter** cũng giúp rút gọn đoạn code lại. Tuy nhiên đoạn code rút gọi lần này là để **xóa, lọc** đi một hoặc nhiều phần tử có trong mảng.

```
a = [1, 2, 1, 3, 1, 4, 1, 5];
```

Bây giờ ta muốn xóa các số 1 đi. Khi này ta có thể dùng một mảng mới (`newArr`) là một mảng rỗng. Khi duyệt qua từng phần tử trong `a` . Nếu `a[i]` bằng `1` thì ta bỏ qua, còn nếu không phải thi ta đơn giản đưa phần từ đó vào mảng `newArr` thông qua hàm `push` .

```
const newArr = []
for (let i = 0; i < a.length; i++) {
  if (a[i] !== 1) newArr.push(a[i]);
}

console.log(a); // [1, 2, 1, 3, 1, 4, 1, 5]
console.log(newArr); // [2, 3, 4, 5]
```

Vậy bây giờ ta muốn xóa các phần tử có giá trị là 2?

```
const newArr = []
for (let i = 0; i < a.length; i++) {
  if (a[i] !== 2) newArr.push(a[i]);
}

console.log(a); // [1, 2, 1, 3, 1, 4, 1, 5]
console.log(newArr); // [1, 1, 3, 1, 4, 1, 5], mất số 2
```

Rồi các phần tử lớn hơn 3, nhỏ hơn 4? Chả lẽ mồi lần vậy là phải viết lại vòng for, tạo biến mới để push vào? Lại mất thời gian.

Thế là các lập trình viên Javascript mới đẻ ra hàm filter để giải quyết vấn đề này. Hàm **filter** giống hàm **map**, sẽ nhận vào một **hàm để xét điều kiện push vào newArr**. Hàm truyền vào filter có

những giá trị giống hệt với map. Đó là **function (element, index, arr)**. Và function này sẽ thay vì thay thế phần sau dấu = như map. **Nó sẽ thay thế điều kiện trong câu lệnh if.**

Đào sâu vào filter nào

```
class Array {
  filter(func) {
    const arr = this.array;
    const newArr = [];
    for (let i = 0; i < arr.length; i++) {
      // vì mỗi khi muốn xóa cái gì đó
      // ta chỉ sửa duy nhấn ở chỗ trong if
      // nên để thuận tiên ta thay chỗ trong if = function
      // function này sẽ giúp filter làm sao để loại bỏ
      // những phần tử muốn xóa đi
      if (func(arr[i], i, arr)) newArr.push(arr[i]);
      // như vậy code chỉ cần 1 vòng for, 1 biến newArr và 1 function truyền vào
      // không cần phải mất thời gian viết lại nữa :3
    }
    return newArr;
  }
}
```

Hiểu được rồi thì lại tập áp dung nào

```
const arr = [1, 2, 1, 4, 1, 6, 7, 8, 9, 10];
const newArray = arr.filter(function (element) {
  return element === 1; // xóa số 1
});

console.log(arr); // [1, 2, 1, 4, 1, 6, 7, 8, 9, 10]
console.log(newArray); // [2, 4, 6, 7, 8, 9, 10]


const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const newArray = arr.filter(function (element) {
  return element > 5; // xóa số > 5
});

console.log(arr); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log(newArray); // [1, 2, 3, 4, 5]
```