Javascript nâng cao ES6, ES7...

Những điều cần biết

1. Scope - Pham vị của biến

- Là phạm vi sử dụng của 1 biến tùy thuộc vào vị trí được khai báo
- Bất kỳ biến nào cũng đều sẽ có scope của riêng nó
- 1 Biến có thể là:
    - const: hằng số không thể thay đổi
    - var, let: Biến có thể thay đổi
    - function: Function cũng là biến
- const, let giới hạn bởi dấu {} bao ngoài nó hoặc là toàn cục
- Var giới hạn bởi dấu {} bao ngoài nhưng ko bao gồm for, white, if

2. const - let - var

`const` - khai báo hằng số

- Const không thể thay đổi trong quá trình thực thi.
- Const phải có giá trị khi khai báo
- Const không thể khai báo lại
- Scope nằm trong cặp ngoặc `{ }` gần nhất
- Khi const giữ giá trị object. Object có thể thay đổi giá trị bên trong nó nhưng không thể thay đổi bằng 1 object khác (Không khuyến khích). vd:

```
const obj = {
    a: 1,
    b: 2
}

obj.a = 10 // No Error
obj = {...} // Error - Const không thể thay đổi bằng nguyên 1 obj khác
```

`var` - khai báo biến (kiểu cũ)

- Var khai báo 1 biến có giá trị bất kỳ, hoặc không cần giá trị khi khai báo

- Var có thể khai báo lại
- Có thể thay đổi giá trị trong quá trình thực thi
- Scope nằm trong cặp ngoặc function hoặc class, nếu var đc khai báo trong cặp `{ }` của if hoặc `( )` của for vd:

```
if (true) {
    var a = 1
}

console.log(a) // No Error

for (var i = 0; i < 1; i++) { }

console.log(i) // No Error
```

`let` - khai báo biến (nên sử dụng)

- Let khai báo 1 biến có giá trị bất kỳ, hoặc không cần giá trị khi khai báo
- Let có thể khai báo lại
- Có thể thay đổi giá trị trong quá trình thực thi
- Scope nằm trong cặp ngoặc `{ }` hoặc `( )` gần nhất

Arrow function

```
const funcA = () => { return 1 }
const funcB = () => 1 // return trực tiếp
```

- Là cách khai báo function kiểu mới, ngắn gọn hơn
- Không thể dùng từ khóa `this` bên trong arrow function, khi sử dụng sẽ tự hiểu `this` của function bao ngoài+
- Nếu không có cặp `{ }` thì giá trị sẽ được `return` trực tiếp
- Không thể dùng biến `arguments` mặc định của function. Có thể dùng `Rest syntax (ES6)` để thay thế

Destructuring

- Destructuring là một cú pháp cho phép bạn gán các thuộc tính của một Object hoặc một Array. Điều này có thể làm giảm đáng kể các dòng mã cần thiết để thao tác dữ liệu trong các cấu trúc này. Có hai loại Destructuring: Destructuring Objects và Destructuring Arrays.

```
// Object
let obj = {
    a: 1,
    b: 2
}
```

```
let a = obj.a
let b = obj.b

let { a, b, c } = obj // let a = obj.a, b = obj.b
// c = undefined

// Array
let arr = [1,2,3,4,5,6,7]

let [a, b, c, d, , , g] = arr
```

- Có thể dùng destructuring khi nhận giá trị trả về từ 1 function là Object hoặc Array

```
// Object
function funcA() {
    return {
        a: 1,
        b: 2,
        c: () => {}
    }
}

let { a, b, c } = funcA()
// c là một function

// Array
function funcB() {
    return [1,2,3,4,5,6]
}

const [a, b, c, d, , , g] = funcB()
```

- Hoặc có thể từ tham số truyền vào 1 function

```
// Object
function funcA({ a, b, c }) {
    // c là một function
    ....
}

funcA({
        a: 1,
        b: 2,
        c: () => {}
    })

// Array
function funcB([a,b,c,d]) {
```

```
    ....
}

funcB([1,2,3,4,5,6,7])
```

Spread (Phân giải 1 array, object ra)

- Là cách ngắn gọn để thao tác với mảng, object một cách hữu dụng như thêm phần tử vào mảng, kết hợp mảng (object), truyền tham số cho function

```
Math.max(1,3,5) // output: 5
Math.max([1,3,5]) // output: NaN

// Spread
Math.max(...[1,3,5]) // output: 5
// ==> Math.max(1,3,5)
```

```
--> Khi chúng ta truyền nhiều tham số vào hàm max (không  giới hạn), thì hàm max sẽ for qua tất cả các giá trị và tìm số lớn nhất

--> Nhưng khi chúng ta có 1 array và muốn truyền vào hàm max, chúng ta không thể truyền nguyên 1 array mà phải dùng Spread để truyền vào, spread sẽ tự động gắn các item của array vào từng tham số của hàm
```

- Ngoài chức năng như mình đã kể ở trên, spread operator còn có rất nhiều các chức năng hữu dụng khác giúp code của chúng ta ngắn gọn và dễ nhìn hơn rất nhiều, có thể kể đến như :
    - Sao chép một mảng

```
const arr1 = [1, 2, 3, 4, 5]

const arr2 = [...arr1] // Tạo ra 1 bản copy từ arr1
```

    - Tách hoặc kết hợp một hay nhiều mảng

```
const arr1 = [1, 2, 3, 4, 5]
const arr2 = [7, 8, 9, 10, 11]

const arr3 = [...arr1, 6, ...arr2] // Tạo ra 1 bản copy từ arr1, arr2
```

    - Sử dụng mảng như danh sách các argument

```
// Spread
Math.max(...[1,3,5]) // output: 5
// ==> Math.max(1,3,5)
```

- Thêm một item vào một list

```
const arr1 = [1, 2, 3, 4, 5]

const arr2 = [...arr1, 6]
```

- Thao tác với state trong React

```
const state = { a: 1, b: 2, c: 3 }

const { a, ...another } = state
```

- Kết hợp các objects

```
const state1 = { a: 1, b: 2, c: 3 }
const state2 = { a: 4, d: 5, e: 6 }

const state3 = {...state1, ...state2}
/*
    state3 = {
        a: 4,
        b: 2,
        c: 3,
        d: 5,
        e: 6
    }
*/
```

- Chuyển NodeList thành một array

```
const list = [...document.getElementByClassName('.abc')]
```

- Biến 1 string thành array (split)

```
[..."😋😛😜🤪😝"]]
// Array [ "😋😛😜🤪😝" ]
[..."🙂🙃😉😊😇🥰😍🤩!"]
// Array(9) [ "🙂", "🙃", "😉", "😊", "😇", "🥰", "😍", "🤩", "!" ]
```

Rest (gôm 1 array, object vào)

```
// Rest
const funcA = (a, b, ...params) => {
    console.log(a) // 1
    console.log(b) // 2
    console.log(params) // [3, 4, 5]
}

const arr = [1,2,3,4,5]
// Spread
funcA(...arr)
```

- Nhìn chung Rest cũng giống Spread, nhưng nó là quá trình ngược lại, gôm những giá trị không được khai báo vào 1 biến. Được sử dụng trong tham số trong function
- Rest sử dụng được cho cả arrow function và function
- Rest parameter phải được đặt cuối của tham số

Closure

- **Closure** (tạm dịch: bao đóng) cho phép lập trình viên Javascript viết mã tốt hơn.
- Trước khi làm quen với closure, bạn nên hiểu rõ về scope của 1 biến: const, var, let
- Closure là một hàm được tạo ra từ bên trong một hàm khác, hàm bên trong có thể sử dụng các biến của hàm bao ngoài nó. Có thể trả về hàm bên trong đó ra ngoài trong trường hợp cần thiết

```
function a() {
    var name = "I'm a Copy";
    function b() { // Closure
        console.log(name);
    }
}

function counter(){
    let count = 1;
    return () => {
        return count++;
    };
}
```

- Hàm bên trong không chỉ truy cập được các biến bên ngoài, mà còn sử dụng được các biến khi sử dụng truyền vào
- Closure có thể truy cập biến bên ngoài, ngay cả khi hàm bên ngoài đã được trả về

- Closure lưu tham chiếu đến biến của hàm bên ngoài:

```
    function celebrityID () {
        var celebrityID = 999;
        // Ta đang trả về một object với các hàm bên trong.
        // Tất cả các hàm bên trong có thể truy cập đến biến của hàm ngoài
(celebrityID).
        return {
            getID: function ()  {
                // Hàm này sẽ trả về celebrityID đã được cập nhật.
                // Nó sẽ trả về giá trị hiện tại của celebrityID, sau khi setID
thay đổi nó.
                return celebrityID;
            },
            setID: function (theNewID)  {
                // Hàm này sẽ thay đổi biến của hàm ngoài khi gọi.
                celebrityID = theNewID;
            }
        }

    }
```

- Trong chế độ `strict mode` con trỏ `this` khi sử dụng sẽ là `undefined`. Đối với chế độ bình thường `this` chính là `window`
- Closure giúp quản lý code hiệu quả và ngắn gọn hơn, tăng khả năng sử dụng lại code

Currying

- Currying là 1 function trả về 1 function, trong function đó lại trả tiếp 1 function khác
- Có thể kết hợp cả closure và curring trong cùng 1 function
- Ví dụ ta có 1 hàm discount giãm 10% giá trị

```
    const discount = () => {
        let price = 1000
        let percent = 10 / 100;
        return () => {
            price -= percent * price
            return price
        }
    }

    const dis = discount()
    console.log(dis()) // - 10% = 900
    console.log(dis()) // - 10% = 810
```

- Ví dụ ta muốn tạo ra nhiều loại giảm giá với các loại khác nhau

```
const product = () => {
    let price = 1000
    return (percent) => () => {
        price -= percent * price
        return price
    }
}

let discount = product()

const dis10 = discount(10 / 100)
const dis20 = discount(20 / 100)
console.log(dis10()) // - 10% = 900
console.log(dis20()) // -20% = 720
```

Template string

```
const name = 'Đặng Thuyền Vương'
const str = `Hello ${name}!` // Hello Đặng Thuyền Vương
```

Optional chaining (?.)

- Kiểm tra giá trị của 1 biến trước khi sử dụng
- Thường được dùng cho object, array, function
- Không thể dùng `?.` cho phép gán
- Mặc định giá trị khi attribute đó không tồn tại là `undefined`

```
const obj = { a: 1 }

console.log(obj?.a) // 1
console.log(obj?.b?.c?.d) // undefined
console.log(obj?.b?.()) // undefined
console.log(obj?.a?.()) // Error: a không phải là function

const arr = [1,2]

console.log(arr?.[0]) // 1
console.log(arr?.[10]?.a?.b) // undefined
```

Shallow copying và Deep copying

- Shallow copying: (Copy cạn)
    - Tạo 1 bản copy của object hoặc array

- Các reference trong object vẫn được giữ nguyên. Các con trỏ vùng nhớ vẫn đang trỏ cùng 1 địa chỉ vùng nhớ
- Khi thay đổi 1 giá trị trên bản dữ liệu, giá trị của biến kia vẫn được thay đổi theo
- Thường sử dụng cho setState của React

```
const obj = {
    a: {
        a1: 1,
        a2: 2,
        a3: 3
    }
}

let obj2 = {...obj}

obj2.a.a1 = 100
// obj.a.a1 = 100
```

- Deep copying: (Copy xâu)
  - Tạo ra 1 bản copy hoàn toàn mới
  - Các object sẽ không còn reference với nhau. Con trỏ vùng nhớ đang trỏ trên 2 vùng nhớ khác nhau
  - Khi thay đổi giá trị trên 1 bản dữ liệu, giá trị của biến kia không thay đổi theo
  - Thường sử dụng cho việc copy hoàn toàn 1 object

```
const obj = {
    a: {
        a1: 1,
        a2: 2,
        a3: 3
    }
}

let obj2 = JSON.parse(JSON.stringify(obj))

obj2.a.a1 = 100
// obj.a.a1 = 1
```

Promise, async/await

- Promise là 1 object đặc biệt, 1 object Promise sẽ luôn luôn có `then` và `catch`
- Then xẩy ra khi gọi hàm `resolve`

- catch xẩy ra khi gọi hàm `reject`
- Dùng Promise để khử `callback hell` trong trường hợp cần thiết
- Giá trị nhận được trong `then/catch` sẽ tùy thuộc vào giá trị khi ta `resolve/reject`

```
const delay = (callback, timmer = 1000) => {
    setTimeout(callback, timmer)
}

delay(() => {
    console.log(1)
    delay(() => {
        console.log(2)
        delay(() => {
            console.log(3)
        })
    })
})

// Promise

const delay = (timmer = 1000) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve()
        }, timmer)
    })
}

delay()
.then(() => {
    console.log(1)
    return delay()
})
.then(() => {
    console.log(2)
    return delay()
})
.then(() => {
    console.log(3)
})
.catch(() => {

})


delay()
.then(() => {
    console.log(1)
    return 1111111111
})
```

```
    .then((res) => {
        console.log(res)
        return 22222222
    })
    .then((res) => {
        console.log(res)
    })
    .catch(() => {

    })


    const fetchApi = () => {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve({ data: 234234234 })
                // reject({ data: 234234234 })
            }, timmer)
        })
    }

    fetchApi()
    .then((res) => {
        console.log(res) // { data: 234234234 }
    })
    .catch((error) => {
        console.log(error) // { data: 234234234 }
    })
```

- `async/await` là cú pháp để run Promise giống như code bình thường
- Code trong function có sử dụng `await` bắt buộc phải khai báo function `async`
- Khi function được khai báo `async` mặc định function đó là function `return` về `Promise`

```
    async function run(){}

    const run = async () => {
        console.log(1)
        await delay()
        console.log(2)
        await delay()
        console.log(3)
        throw {error: '.....'}
    }

    run()
    .then(res => {
        console.log('end')
    }).catch(err => console.log(err))
```

```
    try{
        await run()
    }catch(err){
        console.log('error', err)
    }
```

IIFE (Immediately Invoked Function Expression)

```
- Một hàm được khơi tạo không tên và thực thi ngay tại thời điểm khởi tạo

- Được dùng giới hạn scope của 1 biến trong trường hợp quá nhiều biến trùng tên

- Thường sử dụng khi muốn gọi hàm async/await nhưng lại không có function bao
ngoài

- Khi gọi async/await trong useEffect React
```

```
    (() => {

    })()

    ((number) => {

    })(100)

    (async () => {
        await ....
    })()
```

Thao tác xử lý logic với các bài toán thông dụng

Thao tác với Array

- Các hàm thao tác với Array:
  - find
  - findIndex
  - filter
  - map
  - forEach
  - push
  - pop
  - shift
  - unshift

    - some
    - every
    - sort
    - concat
    - reduce
- Các thao tác:
    - Thêm cuối, lây cuối, thêm đầu, lấy đầu
    - filter những item theo điều kiện nào đó
    - Tìm object với điều kiện
    - Tìm index của object với điều kiện
    - Kiểm tra tất cả item trong mảng có đúng điều kiện
    - Check xem có bất kỳ 1 item nào có giá trị như mong muốn
    - Tính tổng trên mảng

Thao tác với Object

- Các thao tác với Object
    - Từ `object` sang `string`
    - Từ `string` sang `object`
    - Shallow copy
    - Deep copy
    - Shallow copy và update 1 attribute
    - Deep copy và update 1 attribute

Bài tập

1. Làm chức năng countdown cho đồng hồ đếm ngược, có 2 đồng hồ trên cùng 1 giao diện, sử dụng closure
   --> Xem demo và bài giải ở Baitap/ngay1/ngay1_closure.html

2. Làm chức năng tăng giảm số lượng sản phẩm, có 2 box, mỗi box có nút tăng/giảm số lượng. Box 2 có thêm các nút tăng 10, tăng 20, giãm 10, giãm 20
   --> Xem demo và bài giải ở Baitap/ngay1/ngay1_currying.html

3. Làm theo đề bài ở file Baitap/ngay1/ngay1_array.html

4. Làm theo đề bài ở file Baitap/ngay1/ngay1_destructuring.html

5. Làm theo đề bài ở file Baitap/ngay1/ngay1_object.html