# ES6

## Khai báo biến trong ES6 với let var const:

1.Let:

- Dùng để khai báo biến (Thay thế và khắc phục một số nhược điểm của var)
- Có thể gán giá trị nhìu lần.

2.Const:

- Là hằng số, đại lượng không đổi.
- Không thể gán lại = 1 giá trị khác.
- Nếu hằng số là 1 Object nó không thể gán = 1 object khác tuy nhiên nó có thể set lại giá trị thuộc tính.

### -Phân biệt var, let, const với cơ chế **Hoisting** :

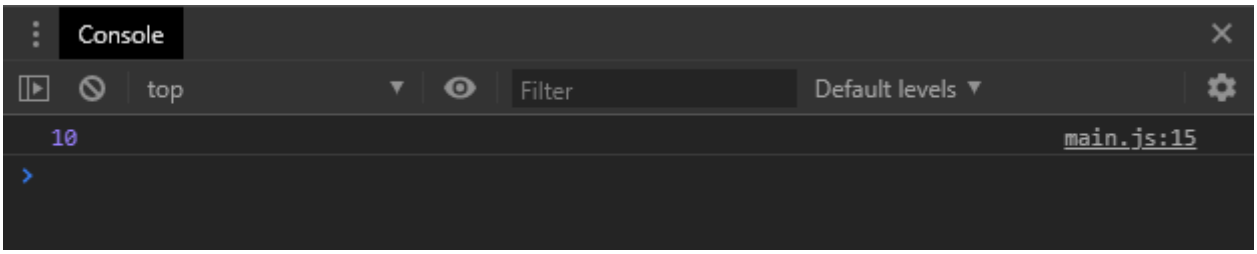
Trong JS cho phép khi khai báo biến kiểu **var** :

Ta có:

```
a= 10;
console.log(a);
var a=20;
```

biến **a** được sử dụng trước khi khai báo và JS cho phép điều đó khi khai báo kiểu **var** .

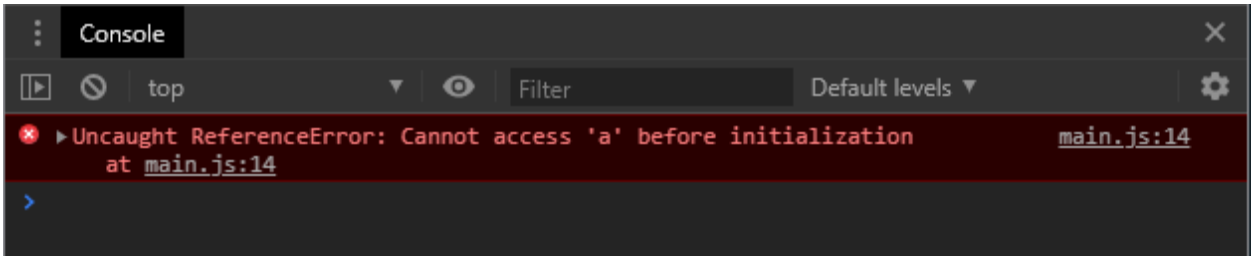
và kết quả:




Nhưng khi khai báo kiểu **let** hoặc **const** :

```
a= 10;
console.log(a);
let a=20;
```

thì kết quả:




Vì biến **a** theo cơ chế của **let** , **const** khi biên dịch qua cơ chế **ES5** thì nó sẽ biên dịch kiểu này :

```
/1/   var a;
/2/   console.log(a);
/3/   a=10;
```

Biến **a** ở dòng 1 dc khai báo chưa có giá trị nên khi thực hiện dòng lệch 2 **console.log(a);** sẽ **lỗi** ,đến dòng 3 thì biến **a** mới dc gán giá trị = **10**

**Vì** những gì khi khai báo biến let hay const trình biên dịch sẽ kéo lên đầu của **Function Scope** .

**Kết luận:** *Mục đích của từ khóa **let, const** là để gàng buộc cơ chế là khai báo biến sau đó mới dc sử dụng .chứ ko phải như **var** có thể sử dụng tùy ý mặt dù chưa khai báo. "**khắc phục cơ chế sử dụng biến khi chưa được khai báo** ."*

## II. Function Scope và Block Scope :

### 1. Function Scope :

- Là phạm vi khai báo biến bên trong một hàm.
- **Biến** bên trong **Scope** sẽ không lấy giá trị ra được từ bên ngoài.
- **Bên ngoài** không sử dụng được **biến** bên trong NHƯNG **bên trong** sử dụng được **biến** bên ngoài **.**

### 2. Block Scope :

- Là phạm vi khai báo **biến** bên trong **{...}** . biến bên trong scope sẽ không lấy dc giá trị từ bên ngoài . **Ngoại trừ Var** nó sẽ không tuân thủ theo quy luật này .

```
if(true){
    var x= 10;
}
console.log(x);
// kết quả : 10
```

Khi dùng **let** :

```
if(true){
    let x= 10;
}
console.log(x);
// kết quả : Uncaught ReferenceError: x is not defined
```

**Kết luận:**

- *khi khai báo **let** **phạm vi hoạt động** của nó nằm trong 1 **scope** ứng với mỗi **scope** thì khai báo **let x** sẽ khác nhau .*
- *còn đối với **var** khi khai báo trùng nó sẽ ảnh hưởng đến bên trong **scope** .*

## III. Arrow Function :

**Arrow Function:** là một cách viết ngắn gọn của ES6. Là **function** được viết rút gọn từ khóa **function** thay bằng dấu mũi tên.

-Ngoài việc viết ngắn gọn **function**

Khi sử dụng Function của ES5:

```
// no-arrow function
var hoTen ="abc";
// nó sẽ hỉu là
// window.hoTen="abc";
let hocVien={
    hoTen:'Nguyễn A',
    lop:'11dhpm',
    layThongTinHocVien: function(){
        function hienThiThongTin(){
            console.log('Họ Tên: '+this.hoTen+' Lớp: '+this.lop);
        }
        hienThiThongTin();
    }
}
hocVien.layThongTinHocVien();
// kq : 'Họ Tên: abc Lớp: undefined'
```

-vì khi sử dụng **function** của **ES5 ngữ cảnh** của con trỏ **this** nó sẽ hỉu là của **window**

**Khi** sử dụng **arrow function ES6 :**

```
//use arrow function
window.hoTen="abc";
let hocVien = {
    hoTen: 'Nguyễn A',
    lop: '11dhpm',
    layThongTinHocVien: function () {
        let hienThiThongTin = () => {
            console.log('Họ Tên: ' + this.hoTen + ' Lớp: ' + this.lop);
        }
        hienThiThongTin();
    }
}
hocVien.layThongTinHocVien();

// kq : 'Họ Tên: Nguyễn A Lớp: 11dhpm'
```

-vì khi sử dụng **arrow function** nó sẽ hiểu ngữ cảnh của con trỏ **this** là của đối tượng **hocVien** mặc dùng t có khai báo biến hoTen với ngữ cảnh con trỏ là **window**.

Câu hỏi: v khi nào dùng **function** và khi nào dùng **arrow function** ?

-Ta sẽ dùng **arrow function** cho cách trường hợp dạng truyền **call back function** (nghĩa là :trong 1 **function** chúng ta cần sử dụng thêm 1 **function** nữa lòng vào trong và khi đó chúng ta muốn sử dụng đúng ngữ nghĩa con trỏ **this** chúng ta dùng **arrow function** ví dụ trong trường hợp trên).

## IV. Rest Params :

**Rest:** Các tham số truyền vào sẽ hợp thành 1 mảng, dùng khi không biết có bao nhiu tham số đầu vào của 1 hàm.

Vì trong JS **không có** khái niệm hàm chồng lên nhau

Ví dụ :

```
function tinhTong(a,b){
    console.log(a+b);
    return a+b;
}
function tinhTong(a,b,c){
    console.log(a+b+c);
    return a+b+c;
```

```
}
tinhTong(1,2);
tinhTong(1,2,3);

// kq : NaN 6
```

Nó sẽ không hàm **tinhTong** ở trên nên khi truyền 2 **param** thì nó trả về **NaN**

Khi sử dụng **...RestParams**

```
function tinhTong(...resParams){
    let tong=0;
    for(let i=0 ;i<resParams.length;i++){
        tong+=resParams[i];
    }
console.log(tong);
}

tinhTong(1,2);
tinhTong(6,7,2,6,3);
tinhTong(3,5,1,6);
// kq :3 24 15
```

**Ví dụ cho phép định nghĩa hàm chồng bằng Rest Param:**

```
let mangHocVien = [
    {
        maHV: 1,
        tenHV: 'Nguyễn Văn C',
    },
    {
        maHV: 2,
        tenHV: 'Nguyễn Văn D',
    },
]

function xuLyMangHocVien(...resParam) {
    if (resParam.length === 2) {
        resParam[1].push(resParam[0]);
        console.log(resParam[1]);
    }else if(resParam.length>2){
```

```
        switch(resParam[2]){
            case 'Delete':{
                let index=resParam[1].findIndex(hv => hv.maHV===resParam[0].maHV)
;
                resParam[1].splice(index,1);
                console.log(resParam[1]);
            }
            case 'Update':{
                let index=resParam[1].findIndex(hv => hv.maHV===resParam[0].maHV)
;
                resParam[1][index].tenHV='Kha đẹp trai';
                console.log(resParam[1]);

            }
        }
    }else{
        console.log(resParam[0]);
    }
}
let hv={
    maHV:3,
    tenHV:'Ngọc Kha',
}
xuLyMangHocVien(mangHocVien); // (2) [{…}, {…}]
xuLyMangHocVien(hv,mangHocVien); // (3) [{…}, {…}, {…}]
xuLyMangHocVien(hv,mangHocVien,'Update'); // (3) [{…}, {…}, {…}]
```

## V. Spread Operator :

**Spread Operator:**toán tử 3 chấm ,dùng để thêm phần tử vào mảng hoặc thêm thuộc tính vào **object**. Ngược với **Rest** nó nhận vào **mảng** và **trả ra từng phần tử**.

```
let mangA = [1, 2, 3, 4];
let mangB = mangA;
mangB.push(5, 6);
mangA.push(7,8);
console.log(mangB); // mình nghĩ : 1, 2, 3, 4
 kq: (8) [1, 2, 3, 4, 5, 6, 7, 8]  !!!!
```

-khi mà gán 1 **mangB** = **mangA** thì theo tính chất của con trỏ trong lập trình thì vùng nhớ của **mangA** sẽ được trỏ cho **mangB**. khi **mangB** thay đổi thì **mangA** cũng thay đổi và tương tự ngược lại với khi thay đổi **mangA**.

Nếu muốn lấy dự liệu từ **mangA** ra xử lý và không muôn dữ liệu thay đổi thì **Spread Operator ES6** sẽ giúp chúng ta làm việc đó .

```
let mangA = [1, 2, 3, 4];
let mangB = [...mangA];
mangB.push(5, 6);
mangA.push(7,8);
console.log(mangB);
console.log(mangA);
/* kq: (6) [1, 2, 3, 4, 5, 6]
       (6) [1, 2, 3, 4, 7, 8]
*/
```

cú pháp [...mangA] tạo ra 1 mảng mới sẽ ôn hết giá trị mangA bỏ vào [] nên khi ta thay đổi mangB mangA ko bị ảnh hưởng .

**Điều này cũng xẩy ra tương tự với object:**

```
 let hs1={
    maHS:1, tenHS:'Nguyen Ngoc A',
 }
 let hs2=hs1;
 hs2.tenHS='Nguyễn Ngọc Kha';
 console.log(hs1);
//  kq : {maHS: 1, tenHS: "Nguyễn Ngọc Kha"}
```

Sử dụng **Spread operation** của **ES6** :

```
let hs1={
    maHS:1, tenHS:'Nguyen Ngoc A',
 }
 let hs2={...hs1};
 hs2.tenHS='Nguyễn Ngọc Kha';
 console.log(hs1);
//  kq: {maHS: 1, tenHS: "Nguyen Ngoc A"}
```

## VI. Default params :

Cho phép **set giá trị mặc định** tham số (**parameters**) của hàm nếu như không có đối số(**argument**) truyền vào.

```
function tinhTong(a = 5, b = 10, c = a + b) {
    console.log(a + b + c);
    return a + b + c;
}
tinhTong(); //nếu không truyền tham số gì thì nó sẽ lấy tham số mặc định của hàm
// kq : 15
tinhTong(10); //thì nó sẽ lấy tham số này thay vào giá trị đầu tiên và cộng tiếp cho tham số thứ 2
// kq : 20
tinhTong(2, 2);//thì nó sẽ lấy tham số 3 mặc định
// kq : 8
tinhTong(2,2,2);
// kq : 6
```

## VI. For in For of:

**For in** duyệt mảng theo chỉ số **index**

```
let arrName = ['Khai', 'Hùng', 'Tiên', 'Mỹ', 'Mọi'];

for(let index in arrName){
    console.log( 'index: '+index+' name: '+arrName[index]);
}
/* kq:
index: 0 name: Khai
index: 1 name: Hùng
...
*/
```

-**for in** sẽ lấy về **vị trí** của phần tử đó.

**for of** sẽ lấy về 1 **đối tưởng** của phần tử đó.

```
let arrName=[{name:'Tùng',age:20},{name:'Nhủ',age:21},
            {name:'Đại',age:22},{name:'Hoa',age:19}]
for (let item of arrName) {
    console.log(item);
```

```
}
/* kq:
   {name: "Tùng", age: 20}
   {name: "Nhủ", age: 21}
   ....
*/
```

## VII. OOP:

**ES5** :

```
// ES5
function HocSinh(mahS, tentS) {
   this.maHS = mahS;
   this.hoTen = tentS;
   this.xuatTenHS = function () {
      console.log(this.maHS, this.hoTen);
   }
}
let hs = new HocSinh(1, 'Nguyen A');
hs.xuatTenHS();
```

**ES6** :

```
// ES6
class HocSinh_ {
   maHS;
   tenHS;
   constructor(mahs, tenhs) {
      this.maHS = mahs;
      this.tenHS = tenhs;
   }
   // xuatThongtinHS = () => {
   //    console.log(this.maHS,this.tenHS);
   // }
   xuatThongtinHS(){
      console.log(this.maHS,this.tenHS);
   }

}
let hs1 = new HocSinh_(1, 'nguyen B');
hs1.xuatThongtinHS();
```

## VIII. OOP -extend:

Khởi tạo **class NhanVien** :

```
class NhanVien{
   maNV;
   tenNV;
   constructor(maNV,tenNV){
      this.maNV=maNV;
      this.tenNV=tenNV;
   }
   tinhLuong(){
      return 1000;
   }
}
```

Khởi tạo class **QuanLy** kế thừa từ **nhanVien :**

```
class QuanLy extends NhanVien{
   dsPB=[];
   constructor(maNV,tenNV,dspb){
      super(maNV,tenNV);
      this.dsPB=dspb;
   }
   tinhLuong(){
      return super.tinhLuong()+1000;
   }
}
let ql=new QuanLy(1,'nguyen ngoc kha',[{maPb:5,tenPB:'CNTT'}]);

console.log(ql);
```

## VIV. OOP -import -export:

Nếu **export default,** khi ta import có thể đặt tên biến tùy ý(không có dấu {})

```
 export default QuanLy;
```

```
import QuanLy1 from './QuanLy';
let ql=new QuanLy1(1,'nguyen ngoc kha',[{maPb:5,tenPB:'CNTT'}]);
```

nếu **export** không có **default** thì khi import ta phải đặt tên **biến** giống với tên đã **export** và có dấu {}

## X. CÁC HÀM XỬ LÝ MẢNG:

-cho 1 mảng sản phẩm có tên **mangSP .**

```
let mangSP = [
   { maSP: 1, tenSP: 'Sony Xperia X22', giaTien: 17500000, hangSX: 'SONY' },
   { maSP: 2, tenSP: 'Sony Xperia XZ1', giaTien: 15500000, hangSX: 'SONY' },
   { maSP: 3, tenSP: 'Google Pixel XL', giaTien: 27500000, hangSX: 'GOOGLE' },
   { maSP: 4, tenSP: 'Google Pixel 2', giaTien: 17500000, hangSX: 'GOOGLE' },
   { maSP: 5, tenSP: 'Samsung note 9', giaTien: 15500000, hangSX: 'SAMSUNG' },
   { maSP: 6, tenSP: 'Samsung s10', giaTien: 27500000, hangSX: 'SAMSUNG' },
   { maSP: 7, tenSP: 'Samsung s20 ultra', giaTien: 32500000, hangSX: 'SAMSUNG' },
]
```

**filter**() :

-trả về một **mảng** với tất cả các phần tử thõa điều kiện trong **filter.**

```
let mangDtSony= mangSP.filter(sp=>sp.hangSX==='SONY');
console.log(mangDtSony);
/* kq
   0: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
   1: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
*/
```

```
let mangDtSony = mangSP.filter(sp => sp.giaTien >=20000000);
console.log(mangDtSony);
/* kq
   0: {maSP: 3, tenSP: "Google Pixel XL", giaTien: 27500000, hangSX: "GOOGLE"}
   1: {maSP: 6, tenSP: "Samsung s10", giaTien: 27500000, hangSX: "SAMSUNG"}
   2: {maSP: 7, tenSP: "Samsung s20 ultra", giaTien: 32500000, hangSX: "SAMSUNG"}
*/
```

**Find() :**

-phương thức **find**() trả về kết quả là một **đối tượng** với phần tử vượt qua kiểm tra .

-nếu ko có phần tử nào thỏa điều kiện thì trả về undefine.

-nếu có hơn 2 **object** thỏa điều kiện thì nó sẽ trả về **object đầu.**

-thường dùng cho cách **thuộc tính riêng biệt** như : **maSV** , **maPB** …

Tìm sản phẩm có mã sp= 3

```
let timSP = mangSP.find(sp => sp.maSP === 3);
console.log(timSP);
/* kq
      {maSP: 3, tenSP: "Google Pixel XL", giaTien: 27500000, hangSX: "GOOGLE"}
*/
```

Khi **ko** có **phần tử** nào **thỏa điều kiện**:

```
let timSP = mangSP.find(sp => sp.maSP === 13);
console.log(timSP);
/* kq
     undefined
*/
```

Khi ta cố tình tìm sản phẩm với **hangSP**===”**SONY**” bằng **find()**

```
let timSP = mangSP.find(sp => sp.hangSX === 'SONY');
console.log(timSP);
/* kq
     {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
*/
```

- kết quả trả về ta thấy có 2 phần tử nhưng với **find**() chỉ lấy phần tử đầu tiên.

-cho 1 mảng sản phẩm có tên **mangSP .**

```
let mangSP = [
   { maSP: 1, tenSP: 'Sony Xperia X22', giaTien: 17500000, hangSX: 'SONY' },
   { maSP: 2, tenSP: 'Sony Xperia XZ1', giaTien: 15500000, hangSX: 'SONY' },
   { maSP: 3, tenSP: 'Google Pixel XL', giaTien: 27500000, hangSX: 'GOOGLE' },
   { maSP: 4, tenSP: 'Google Pixel 2', giaTien: 17500000, hangSX: 'GOOGLE' },
   { maSP: 5, tenSP: 'Samsung note 9', giaTien: 15500000, hangSX: 'SAMSUNG' },
   { maSP: 6, tenSP: 'Samsung s10', giaTien: 27500000, hangSX: 'SAMSUNG' },
   { maSP: 7, tenSP: 'Samsung s20 ultra', giaTien: 32500000, hangSX: 'SAMSUNG' },
]
```

**Findindex():**

- **findIndex()** cũng giống như **find()** dùng để **tìm** trên các **thuộc tính đặt trưng** như : **maSV**, **maSP …**

-Phương thức **findindex**() trả về **kq là chỉ số của phần tử** ứng với vị trí của phần tử trong mảng.

-khác với **find()** nếu **ko** có **phần tử** nào sẽ trả về **undefine** , **NHƯNG** đối với **findIndex**() thì trả về **kq : -1** .

-Trong **trường Hợp** có **2** hoặc **nhìu** kết quả thỏa **điều kiện** trong **findIndex**() thì nó sẽ trả về **kết quả** xuất hiện **đầu tiên.**

Khi tìm kiếm sp có **maSP ===7** nó trả về **index** của sp đó:

```
let index = mangSP.findIndex(sp => sp.maSP === 7);
console.log('index : '+index);

/* kq: index : 6 */
```

Khi có **2 hoặc nhìu** kết quả thỏa điều kiện thì **findIndex**() sẽ trả về **index** của **phần từ đầu tiên** :

```
let index = mangSP.findIndex(sp => sp.hangSX === 'SONY');
console.log('index : '+index);
/* kq: index : 0 */
```

Khi không có **kết quả** thỏa điều kiện nó sẽ trả về **vị trí = -1**:

```
let index = mangSP.findIndex(sp => sp.hangSX === 100);
console.log('index : '+index);
/* kq: index : -1 */
```

TÍCH HỢP DÙNG ĐỂ TÌM VÀ XÓA 1 PHẦN TỪ :

```
let index = mangSP.findIndex(sp => sp.maSP === 1);
if(index!==-1){
   mangSP.splice(index,1);
   console.log('succeed!');
}else{
   console.log(index);
}
```

-Nếu như hàm **findIndex**() trả về **index !== -1** tức là có **tồn tại** phần tử đang tìm thì ta sẽ **xóa** phần tử thứ **index** đó bằng hàm **splice**({**vị trí xóa**},{**số lượng pt cần xóa**});

Ngược lại thì trả về kết quả **không tồn tại** phần tử đang tìm **{kq : -1}** .

-cho 1 mảng sản phẩm có tên **mangSP .**

```
let mangSP = [
   { maSP: 1, tenSP: 'Sony Xperia X22', giaTien: 17500000, hangSX: 'SONY' },
   { maSP: 2, tenSP: 'Sony Xperia XZ1', giaTien: 15500000, hangSX: 'SONY' },
   { maSP: 3, tenSP: 'Google Pixel XL', giaTien: 27500000, hangSX: 'GOOGLE' },
   { maSP: 4, tenSP: 'Google Pixel 2', giaTien: 17500000, hangSX: 'GOOGLE' },
   { maSP: 5, tenSP: 'Samsung note 9', giaTien: 15500000, hangSX: 'SAMSUNG' },
   { maSP: 6, tenSP: 'Samsung s10', giaTien: 27500000, hangSX: 'SAMSUNG' },
   { maSP: 7, tenSP: 'Samsung s20 ultra', giaTien: 32500000, hangSX: 'SAMSUNG' },
]
```

**ForEach**():

-là **phương thức thực thi một hàm 1 lần cho mỗi phần tử** .nếu mảng có **7 phần tử** thì thực thi hàm đó **7 lần**.

-hàm nhận **tham số đầu vào** là từng **phần tử** của mảng và **vị trí**. **forEach**( ( **item**, **index** ) => {} )

-**forEach** duyệt tự động theo **chiều dài** của **mảng** và **mỗi lần duyệt** nó sẽ trả về **1 đối tượng** . Nói nôm na nó cũng giống như **filter()** nhưng khác ở chỗ **filter()** chúng ta cần hứng 1 giá trị gì đó.

-**filter()** thì trả về mảng mới ,còn **forEach**() không trả về gì cả.

Ví dụ duyệt mảng có tên **mangSP** ở trên :

```
mangSP.forEach((sp,index) => {
   console.log(sp);
})

/* kq
   {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
   {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
   {maSP: 3, tenSP: "Google Pixel XL", giaTien: 27500000, hangSX: "GOOGLE"}
   {maSP: 4, tenSP: "Google Pixel 2", giaTien: 17500000, hangSX: "GOOGLE"}
   {maSP: 5, tenSP: "Samsung note 9", giaTien: 15500000, hangSX: "SAMSUNG"}
   {maSP: 6, tenSP: "Samsung s10", giaTien: 27500000, hangSX: "SAMSUNG"}
   {maSP: 7, tenSP: "Samsung s20 ultra", giaTien: 32500000, hangSX: "SAMSUNG"}
*/
```

Nhưng khi ta cố tình **return** 1 giá trị nào đó thì kết quả:

```
let mang = mangSP.forEach((sp, index) => {
   console.log(sp);
   return sp;
})
console.log(mang);
// kq : undefined
```

**Kết luận** : **mục đích** của **forEach**() được tạo ra chỉ **dùng** để **duyệt mảng** ,**tạo nội dung** hoặc làm gì đó chứ nó **ko trả về gì cả**.

**Map() :**

-hàm **map()** tương tự hàm **forEach**() NHƯNG KHÁC ở chỗ hàm **map**() có giá trị **return** là **1 mảng mới** được tạo ra từ các **đối tượng** được **return** trong **callback function** .

Sử dụng lại mảng **mangSP** ở trên . khi ta dùng **map**() muốn trả về 1 mảng có **giaTien** < 20tr .

```
let mangmoi =mangSP.map((sp,index) =>{
   if(sp.giaTien<20000000){
      return sp;
   }
})
 console.log(mangmoi);

/* kq
(7) [{…}, {…}, undefined, {…}, {…}, undefined, undefined]
   0: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
   1: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
   2: undefined
   3: {maSP: 4, tenSP: "Google Pixel 2", giaTien: 17500000, hangSX: "GOOGLE"}
   4: {maSP: 5, tenSP: "Samsung note 9", giaTien: 15500000, hangSX: "SAMSUNG"}
   5: undefined
   6: undefined
*/
```

Tương tự như **forEach**() **map**() vẫn chạy 7 lần khi thõa điều kiện nó sẽ trả phần từ đó và boot nó vào trong **mangmoi** và ko thõa thì nó vẫn trả về giá trịnh "**undefined** " vào **mangmoi** . nên **kq** chúng ta có là gồm **7 phần tử** nhưng **thõa yêu cầu** bài toán thì chỉ **có 4 phần tử**.

➔ để **khắc phục** và làm **đúng yêu cầu** bài toán thì ta chỉ cần dùng **filter**() với điều kiện trên . kq trả về sẽ đúng như mong đợi.

```
let mangmoi =mangSP.filter((sp,index) =>{
   if(sp.giaTien<20000000){
      return sp;
   }
})
console.log(mangmoi);
/* kq
(4) [{…}, {…}, {…}, {…}]
   0: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
   1: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
   2: {maSP: 4, tenSP: "Google Pixel 2", giaTien: 17500000, hangSX: "GOOGLE"}
   3: {maSP: 5, tenSP: "Samsung note 9", giaTien: 15500000, hangSX: "SAMSUNG"}
*/
```

**Kết Luận** : hàm **map**() được dùng khi chúng ta muốn **render** nội dung (giống như **forEach**() ) NHƯNG chúng ta **muốn** tạo ra **1 mảng** nội dung mới .thì ta sẽ dùng hàm **map()** để tạo ra một nội dung mới từ mảng nội dung cũ. * Còn **forEach**() chúng ta không nắm bắt được giá trị cuối cùng trả về. **công việc** của nó tương tự như **map**() nhưng nó không bắt được giá trị cuối cùng trả về. (hàm **map**() đa số được dùng nhìu để tạo nội dung trong **React** , có 1 số hoặc thậm chí ngta có thể dùng map() để thay thế cho **forEach**() )

**Reduce() :**

-Hàm **reduce**() thực thi **n lần** so với **n phần tử** của **mảng** nhằm tạo ra **1 giá trị mới** (có thể là 1 **biến** ,1 **mảng** , 1 **object** …. Tùy theo xử lý **return** trong **hàm**).

-Cú pháp: **reduce**( ( {**giá trị đầu ra** } , { **item** } , { **index** } ) =>{} ,{**giá trị ban đầu cho giá trị output** } ) .

- Hàm này gồm 2 tham số :

-Một là nhóm **callback** : ( {**giá trị đầu ra** } , { **item** } , { **index** } ) .

-Hai là giá trị ban đầu cho **output**(tham số ban đầu của {**gia trị đầu ra**} của nhóm **callback** );

Bài toán ta có 1 mảng sp ,yêu cầu tính tổng tiền trong các sp trong mảng ( **ko dùng for** ):

```
let mangSP = [
   { maSP: 1, tenSP: 'Sony Xperia X22', giaTien: 17500000, hangSX: 'SONY' },
   { maSP: 2, tenSP: 'Sony Xperia XZ1', giaTien: 15500000, hangSX: 'SONY' },
   { maSP: 3, tenSP: 'Google Pixel XL', giaTien: 27500000, hangSX: 'GOOGLE' },
   { maSP: 4, tenSP: 'Google Pixel 2', giaTien: 17500000, hangSX: 'GOOGLE' },
   { maSP: 5, tenSP: 'Samsung note 9', giaTien: 15500000, hangSX: 'SAMSUNG' },
   { maSP: 6, tenSP: 'Samsung s10', giaTien: 27500000, hangSX: 'SAMSUNG' },
   { maSP: 7, tenSP: 'Samsung s20 ultra', giaTien: 32500000, hangSX: 'SAMSUNG' },
]
```

Duyệt mảng tính tổng tiền sp (ứng dụng **reduce**() để tính ra 1 kết quả mới từ **mangSP** ):

```
let tongTien = mangSP.reduce((TT, sp, index) => {
   return TT += sp.giaTien;
}, 0);
console.log(tongTien);
// kq : 153500000
```

Đầu tiên: tham số **TT** sẽ mang giá trị là **0**

Sau đó: nó sẽ tính **TT** và gán ngược lại giá trị output **TT ban đầu** , sau đó boot lại biến **tongTien**

Lần chạy thứ 2 nó sẽ lấy giá sản phẩm tính tổng tiền rồi return lại **TT**

Nó lập đi lập lại đến hết mảng .

Và cuối cùng ta sẽ có 1 con số là **tongTien** .

Tuy nhiên **reduce**() không chỉ để tính tổng tiền . nó còn có thể **tạo ra 1 giá trị mới** như là 1 **mảng** :

```
let mangDTSony = mangSP.reduce((mangSony,sp,index)=>{
   if(sp.hangSX==='SONY'){
      mangSony.push(sp);
   }
   return mangSony;
},[]);
console.log(mangDTSony);
/*
(2) [{…}, {…}]
   0: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
   1: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
*/
```

Đầu tiên **mangSony** sẽ được gán bằng một mảng rỗng **[]** với tham số thứ 2 của **reduce()**.

Sau đó : nó sẽ thực thi lệch if trong hàm nếu thõa điều kiện thì nó sẽ **push** vào **mangSony**

Tiếp đó là trả về mảng **mangSony** đó.

VÀ TƯƠNG TỰ VỚI 1 OBJECT.

**Lưu ý :**

**-** ta cũng có thể dùng hàm **filter**() để thực hiện việc trả về 1 mảng tương tự như trên tuy nhiên đối với hàm **filter**() thì nó chỉ xử lý được các **API** với 1 **cấp** thôi .

-có thể sau này ta có những **object** mà ta muôn **push** thêm 1 **thuộc tính** nào đó hoặc 1 cái **menu nhìu cấp** thì ta chỉ có thể duyệt bằng hàm **reduce**() thì lúc đó ta có thể thấy nó **rõ ràng tường minh hơn** .

-ngoài ra **reduce**() còn có thêm 1 hàm với chức năng tương tự  chính là **reduceRight**() .NHƯNG **reduceRight**() sẽ duyệt theo 1 **chiều ngược lại**.

**Reverse() :**

-Hàm **reverse**() là hàm trả về 1 mảng đảo ngược mảng ban đầu .

```
let mangsp_reverse=mangSP.reverse();
console.log(mangsp_reverse);

/*
(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
   0: {maSP: 7, tenSP: "Samsung s20 ultra", giaTien: 32500000, hangSX: "SAMSUNG"}
   1: {maSP: 6, tenSP: "Samsung s10", giaTien: 27500000, hangSX: "SAMSUNG"}
   2: {maSP: 5, tenSP: "Samsung note 9", giaTien: 15500000, hangSX: "SAMSUNG"}
   3: {maSP: 4, tenSP: "Google Pixel 2", giaTien: 17500000, hangSX: "GOOGLE"}
   4: {maSP: 3, tenSP: "Google Pixel XL", giaTien: 27500000, hangSX: "GOOGLE"}
   5: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
   6: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
*/
```

**Sort() :**

-Hàm **Sort()** dùng để sắp xếp **mảng** theo thứ tự **tăng dần** hoặc **giảm dần** .

- Có thể ứng dụng **sắp xếp** các **mảng đối tượng** theo giá trị của **thuộc tính**.

Cú pháp : sort( ( {item thứ 2} ,{item thứ nhất} ) => {} )

Sắp xếp **tăng dần** theo **TenSP** ( **theo chuỗi** ):

```
let mangSPtheoTen = mangSP.sort((sp_TiepTheo, sp) => {
   let tenSPTiepTheo=sp_TiepTheo.tenSP.toLowerCase();
   let tenSP=sp.tenSP.toLowerCase();
   if(tenSPTiepTheo>tenSP){
      return 1; //giữ nguyên
   }
   if(tenSPTiepTheo<tenSP){
      return -1; //đảo vị trí
   }
   return 1;
});
console.log(mangSPtheoTen);
```

kết quả :

```
0: {maSP: 4, tenSP: "Google Pixel 2", giaTien: 17500000, hangSX: "GOOGLE"}
1: {maSP: 3, tenSP: "Google Pixel XL", giaTien: 27500000, hangSX: "GOOGLE"}
2: {maSP: 5, tenSP: "Samsung note 9", giaTien: 15500000, hangSX: "SAMSUNG"}
3: {maSP: 6, tenSP: "Samsung s10", giaTien: 27500000, hangSX: "SAMSUNG"}
4: {maSP: 7, tenSP: "Samsung s20 ultra", giaTien: 32500000, hangSX: "SAMSUNG"}
5: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
6: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
```

**Giải thích :**

Đầu tiên **sort**() sẽ lấy **phần tử 2** và **phần tử 1** ra so sánh

Lần 2 chạy sẽ lấy **phần tử 3** và **phần tử 2** tương tự cho những lần khác .

-Cách hoạt động:

Ta sẽ tạo ra 1 biến tạm để lưu trữ **tên của nó** là **tenSP** cần so sánh đặt biệt phải có **.toLowerCase()** để biến đổi nó thành chữ thường nó sẽ chuyễn thành mã ASCII để so sánh việc biến bền **lowercase**() sẽ giúp số ASCII cần so sánh và tiện cho việc sắp xếp vị trí theo thứ tự ký tự trong mã ASCII

Sau khi quy ra mã số : nó sẽ thực hiện dòng **if** nếu mã **tenSPTiepTheo** > **tenSP** thì sẽ **return 1** tức là **giữ nguyên**. Nếu bé hơn thì **return -1** để **đảo vị trí . nếu ko có** trong 2 trường hợp **if** trên thì **return 1** (**giữ nguyên**).

***Công thức này có thể ghi nhớ để tiện cho việc sắp xếp mảng theo chuỗi.***

Sắp xếp **tăng dần** theo **giá** ( **theo Số** ):

```
let mangSPTheoGia=mangSP.sort((sp_tieptheo,sp) => {
   return sp_tieptheo.giaTien-sp.giaTien;
});
console.log(mangSPTheoGia);
```

kết quả :

```
0: {maSP: 2, tenSP: "Sony Xperia XZ1", giaTien: 15500000, hangSX: "SONY"}
1: {maSP: 5, tenSP: "Samsung note 9", giaTien: 15500000, hangSX: "SAMSUNG"}
2: {maSP: 1, tenSP: "Sony Xperia X22", giaTien: 17500000, hangSX: "SONY"}
3: {maSP: 4, tenSP: "Google Pixel 2", giaTien: 17500000, hangSX: "GOOGLE"}
4: {maSP: 3, tenSP: "Google Pixel XL", giaTien: 27500000, hangSX: "GOOGLE"}
5: {maSP: 6, tenSP: "Samsung s10", giaTien: 27500000, hangSX: "SAMSUNG"}
6: {maSP: 7, tenSP: "Samsung s20 ultra", giaTien: 32500000, hangSX: "SAMSUNG"}
```

cách hoạt động:

ta sẽ cho giá tiền của **sp_TiepTheo** (tức là phần tử 2) trừ đi giá tiền của **sp** ( tức là phần tử 1)

lần chạy tiếp theo sẽ là phần tử 3 -phần từ 2 theo giá tiền.

kết quả trả về nếu **Âm** (bé hơn 0) thì sẽ đảo vị trí.

Nếu **Dương** (lớn hơn 0) thì sẽ giữ nguyên.

***Công thức này có thể ghi nhớ để tiện cho việc sắp xếp mảng theo Số.***