* var - Global scope
  let, const - local scope (script)

* global execution context
  ↓
  created in callstack

```
sum();
sub();
console.log(a);
var a = 10;
function sum() {
  var num1 = 10
  var num2 = 20
  console.log(num1+num2);
}
function sub() {
  var num1 = 10;
  var num2 = 20;
console.log(num2 - num1);
}
```

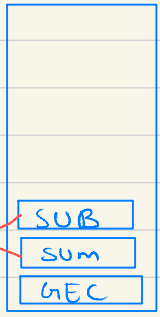
Scan

a: undefined
sum() {
...
}
sub() {
...
}

(MA)
Memory allocation

a = 10

Scan	MA
a = ud	a = 10
b = ud	b = 20



after execution it will be out of GEC

after scanning sum, sub will be stored in Callstack

* Callstack : Brain of JS
  ↓
  Global execution context: where data is stored

```
function sum() {
 var a = 10;
   function sub() {
    console.log(a);
 }
  sub();
}
sum();
```

is in the lexical environment of sum

sum is closure to sub

child cannot access grandparent so inorder to be used by child it needs to be passed to the parent and as child can access parent property it is known as lexical scope

HW: Behavior of pointer vs normal function? (using debugger)

HOF (Higher order Function)

* take function as parameter or returns a function

Callback : Function which can be passed as parameter to a function

* HOF : accepts function as parameter

```
function A(a){ HOF
}
function a(){ : callback
}
```

only when it is passed to a function as a parameter.

```
setTimeout( () => {
    console.log("Hello");
}, 3000);
```

setTimeout : HOF
arrow function : callback

```
setInterval( () => {
    console.log("Refresh");
}, 3000);
```

```
* const arr = ["Shiva", "Anurag", "Yuvrah", "VYOM"];
arr.forEach( (val) => {
console.log(val);
});
```

HOF

#Map , Filter , Reduce

1) `const number = [1,2,3,4,5,6,7,8,9,10];`

`number.map((val) => val * val);`

modify each value of the array

```
{
  return val*val;
}
```

with these brackets we have to write return.

2) Filter

```
const countries = ["Deutschland", "India", "Finland"];

const count = countries.filter((country) => country.includes("land"));

console.log(count);
```

return all the countries which have "land" in them

3) Reduce

```
const numbers = [1, -1, 2, 3];
let a = 0
for(let n of numbers){
    a = a + n;
}
console.log(a);
```

```
numbers.reduce((accumulator, currentValue) => {
return accumulator + currentValue;
}, 0);
```

(accumulator = sum, currentValue = n)

Accumulator value

output : 5

if we do not initialise the accumulator value it will take the first value of array

```
// a=0 , c=1 ⇒ a=1
   a=1 , c=-1 ⇒ a=0
   a=0 , c=2 ⇒ a=2
   a=2 , c=3 ⇒ a=5
```

a=1, c=-1, a=0
a=0, c=2, a=2
a=2, c=3, a=5