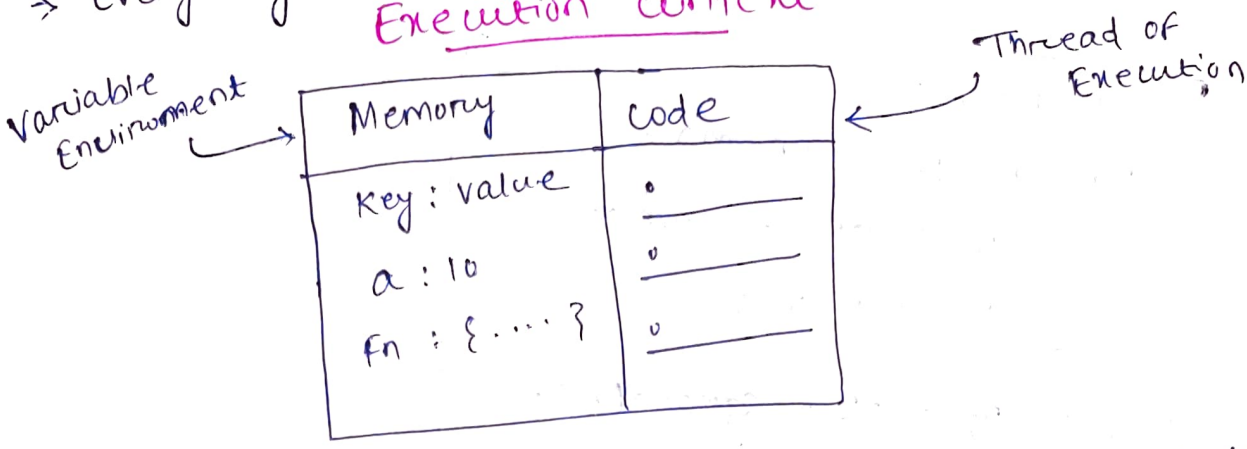
JavaScript

→ Everything in javascript happens inside an "Execution context"

Execution context



- It is like a 'big-box', which has two components in it.

(1) Memory component :-

- It is also know as variable environment.
- This is the place where all the variables & function are stored in (key, value) pairs.

(2) Code component :-

- This is the place where code is executed one line at a time.
- It is also known as thread of execution

→ Javascript is a synchronous single-threaded language.

- That means, JS can execute one command at a time in a specific order.
- when one line is executed completely then after that it goes to second line.

→ What happens when you run javascript code.?

Code :-

```
var n = 2;
function square(num) {
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

- First the global execution content is created in two phase i.e

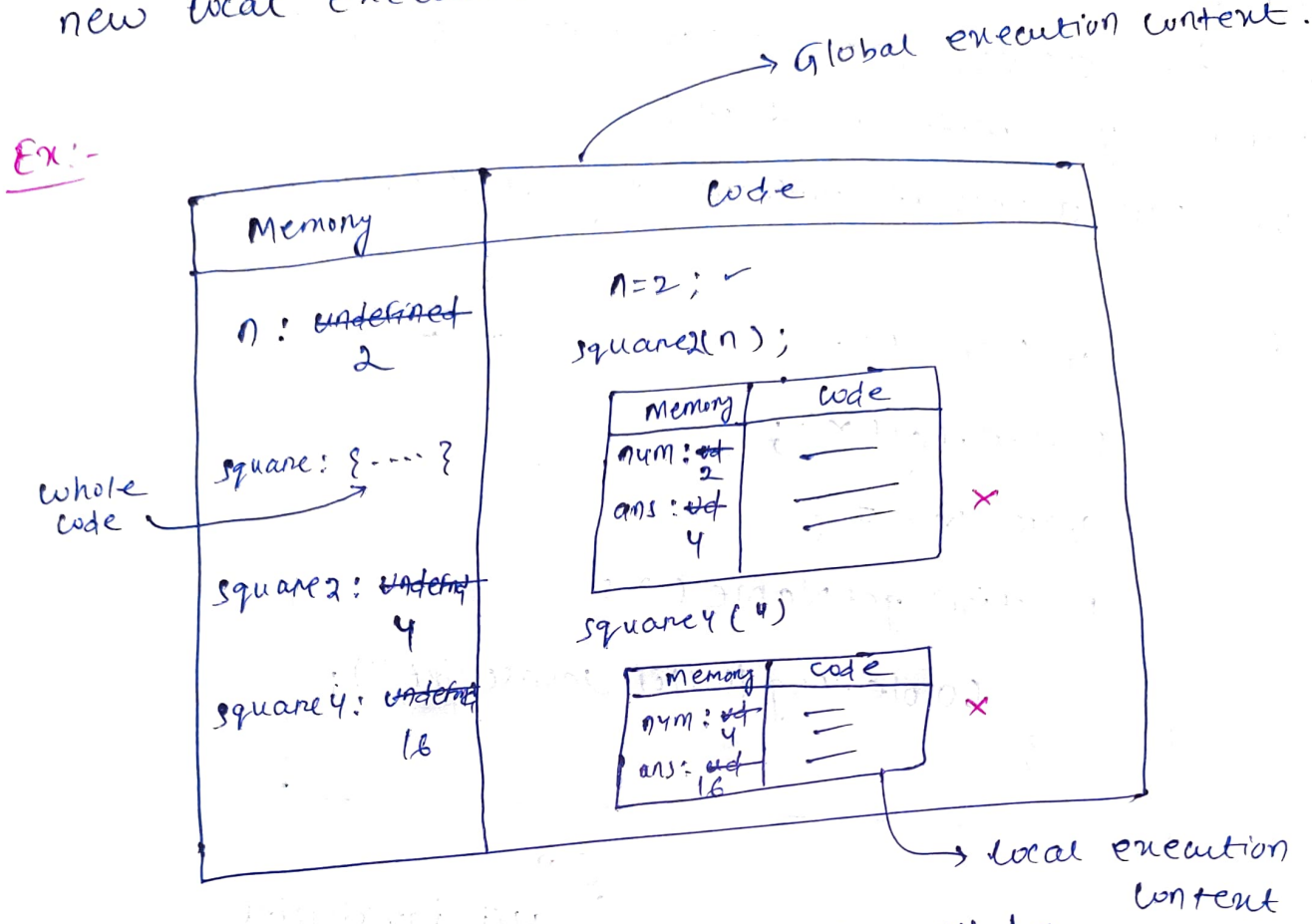
(a) Memory creation phase - we allocate all the variables with the value undefind. & in case of function it copies the whole function in the value.

(b) Code execution phase -

- Now the variable value 'undefined' is replaced by actual initialized value.
- When we encounter function call, then again we create local execution content, then again -
    - it will create memory
    - & goes to code execution
- After this it delete the local execution content.

- Every time it encounter function call, it will create new local execution content.

Ex:-



→ Whenever a function is called, it will be stored in call stack.

→ In javascript, call stack maintains the order of execution of 'execution content'.

→ call stack is also known as

- Execution content stack
- program stack
- control stack
- Runtime stack
- Machine stack (All are same).

Hoisting :-

→ Hoisting is a phenomena in javascript by which we can access variables & functions even before you initialized it.

→ we can access it without any errors.

Ex:-

```
getName();
console.log(x);
var x = 7;
function getName() {
    console.log("Hii javascript");
}
```

O/P

Hii javascript
Undefind

→ if we print the function name

```
console.log(getName);
function getName() {
    console.log("Talwin");
}
console.log(getName);
```

→ Doubt

O/P

f getName() {
    console.log("Talwin");
}
2 times

- Because in execution context, it will store the whole function as value.

→ When we write function in terms of arrow or any other & before initializing we call the function, it will give error.

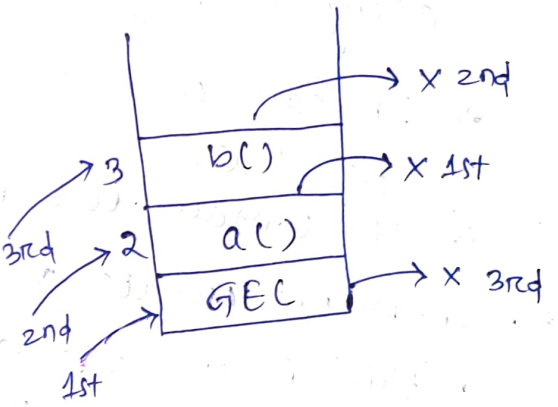
How function works?

```
var x = 1;
a();
b();
console.log(x);
function a() {
    var x = 10;
    console.log(x);
}
function b() {
    var x = 100;
    console.log(x);
}
```

O/P

10
100
1

Call stack



→ Because all the x variable here have different execution content, they are not overlapping with each other.

Window & this Keyword :-

→ The shortest js code is the empty js file, ~~when~~ Because we run the empty file, it still create the global execution ~~file~~ context & also create window object which is created by javasenipt engine into the globalspace. And we can access all it's functionality anywhere in js program.

→ It also create 'this' keyword & it's pointing to window object.

Window :-

→ It is a global object which is created along with global execution content.

→ In case of browsers it is called as window.

→ It contains lots of predefind functionality.

* When we create execution content, a 'this' is created along with it, even for the functional execution content.
  . At global level, this points to global object.

* Anything which is not inside the function is global space.

Ex:-

```
var a = 10;
function b() {
    var x = 10;
}
cosole.log(widow.a);
console.log(a);
console.log(this.a);
```

OIP:-

```
10
10
10
```

→ The global variables & functions get attached to the global object i.e 'window'

- That's why we able to print (widow.a as 10) & also (this.a) because 'this' is pointing to ~~widow~~ window object.

Undefind Vs not defind :-

→ Before executing a single line of code, js allocates it's variables 'undefind', which is a special keyword.

→ (Undefind != empty), it is taking it's space until the value initialized is replaced.

→ It is a placeholder.

(*) . javascript is loosely typed language, because there is no datatypes for variable. A variable can store anything like boolean, integer, decimal value, string etc.

Ex:-

```
var a = 10;
a = "tarun";
a = 10.67;
```

- Also weekly typed language.

(*)

```
a = undefined;
```

→ It's not a mistake, but surely it's not a good practice, because 'undefind' keyword has their own purpose.

Scope & Lexical Environment :-

Scope :-

→ Where we can access specific function or variable.

Lexical environment :-

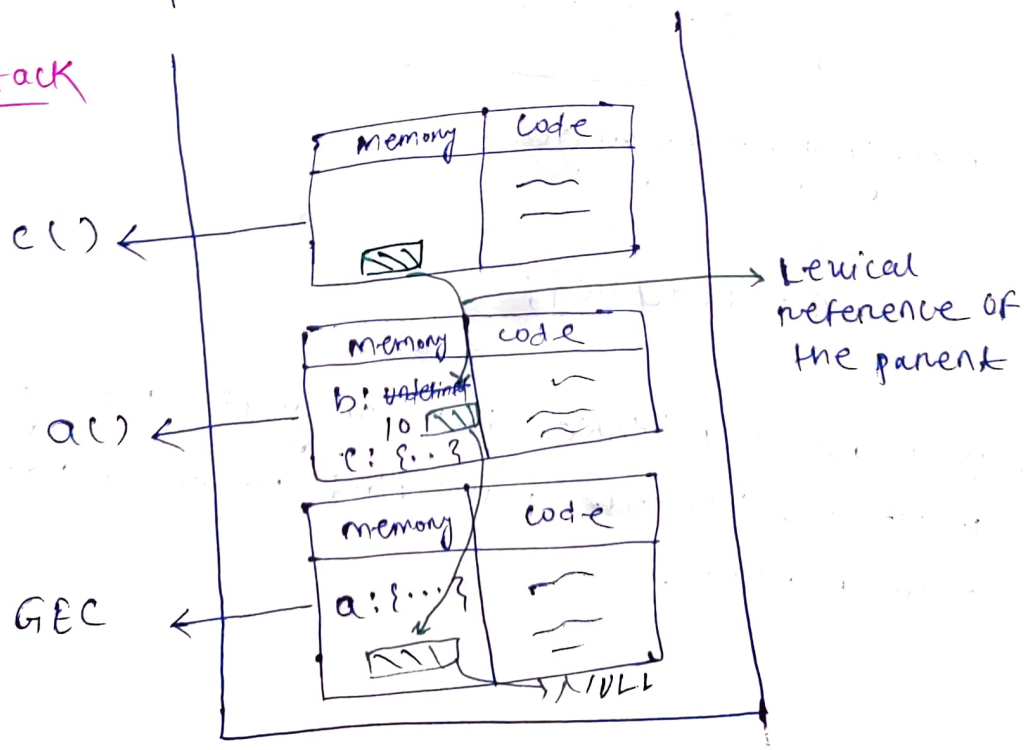
→ Lexical means 'Hierarchy' / 'order'.

Ex:-

```
function a() {
    var b = 10;
    c();
    function c() {
        console.log(b);
    }
}
a();
```

- Here c() is present inside lexical parent a().
- And a() is present inside lexical parent of global scope.

Call stack



→ if we want to access some variable inside Local ~~con~~ execution context & it is not present, Then it will look at their 'Lexical parent'/ 'Lexical environment of their parent'.

→ The way of finding the variables in their Lexical parent environment is "Scope chaining."

* Lexical environment is created when an execution context is created.
  it equals with (local memory + reference to lexical environment of parent.)

* The whole chain of lexical environment is Scope chain.

Let & Const :-

→ let keyword was introduced in ES6 (2015).
→ variable defined with let can't be redeclared.
→ must be declared before use.

• In case of let, the 'let' is hoisted but not in global space, but in some different location which is not accessible until it is initialized or defined.

Temporal dead zone :-

• It is the time since when let variable was hoisted & till it is initialized some value, the time bet^n that is known as temporal dead zone.

→ In case of let & const, they are not attached to window object, they stored in separated memory.

→ We can't redeclare let & const.

```
let a = 10;
let b = 100;
```

Not possible

const :-

→ Must be declared and assigned in single line

→ We can't re-assign it's value later.

Errors :-

Reference Error

- when javascript engine try to find a variable on memory and can't access it i.e reference error.

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

Type Error

```
const a = 100;
a = 13;
```

- This is known as type error, because we are re-assigning the value in const type, which is not possible.

Syntax Error

```
const a;
```

- This known as syntax error, because it should be initialized when it was declared, i.e the must for const. variable

→ Prefer to use let & const in day to day life.

→ The best way to avoid temporal dead zone by initialize all the let & const at top of the program.

Block :-

→ Also known as compound statement.

→ It is used to group multiple Javascript statements.

→ When a single statement is executed & the condition satisfies or function is called with valid parameters / eg argument, at that time these block of codes is executed.

→ It help us to devide our code into multiple part, for multiple purpose.

Block Scope :-

→ Let & const are block scoped, which means whichever block they exist, they alive only when that block is executing, After that they are dead, no memory space outside that is consumed by them.

→ But var is stored in global scope, wherever it will be declared, ~~the~~ it is valid for every scope, when it's value change at any place, that change is affected everywhere.

Lexical Block Scope :-

Akshay Sir

Closures :-

- A function bundled together with it's lexical environment is known closure
- Otherwise function along with it's lexical scope bundled together forms a closure.

Ex:-

```
function x() {
    var a = 7;
    function y() {
        console.log(a);
    }
    y();
}
x();
```

Lexical scope concept

Ex:-2

```
function aa() {
    let j = 12;
    function bb() {
        console.log(j);
    }
    return bb;
};
var x = aa();
x();
```

→ Now Here x will contain the whole bb() function.

→ print 12, Even it is returned to outside, but it will remember, where it came from.

→ bb() exist inside aa(), After returning to x, bb() will remember all of its lexical environment ?:e closure.

Settimeout :-

```
function x() {
    let i = 10;
    setTimeout( function() {
        console.log(i);
    3, 1000);
    console.log("Namastey");
}
x();
```

O/P

Namastey
10

→ Now setTimeout takes the function store it somewhere and put a timer on it.

→ During that js runs/executes all the codes, then execute setTimeout.

Ex:-

```
function x() {
    for ( var i = 1 ; i <= 5 ; i++ ) {
        setTimeout ( function() {
            console.log (i)
        }, i * 1000);
    }
    console.log ("Hello");
}
x();
```

O/P

Hello
6
6
6
6
6

→ Because javascript doesn't wait for anybody, first it will print 'Hello', After executing the for loop.

→ And again because i is referencing to memory location, at the time setTimeout executes ~~at that~~ i becomes 6, that's why it is printing 5 times 6.

Correct it so that it will print 1, 2, 3, 4, 5

→ we can use let, instead of var, whenever it pass 'i' in function it is different block Scope.

→ Without changing the var print 1, 2, 3, 4, 5

- It left us with one option passing var value into a function, Then calling the setTimeout function.

Sol?

```
function x() {
    for (var i = 1; i<=5; i++) {
        function close(x) {
            setTimeout( function() {
                console.log(x);
            }, i * 1000);
        }
        close(i);
    }
};
x();
```

* Pros & Cons of closure ?

→ whatever you tell to the interviewer, the answers, you should aware of those things.

Callback Hell :—

→ It is a term used to describe a situation in javascript where deeply nested callbacks make the code difficult to read & maintain.

Ex:- Shopping Apps.

Add to cart → Proceed to payment → Order summary → update wallet.

```
cart = [ "shoes", "pants", "kurta" ]

api.createOrder( cart, function() {

    api.proceedToPayment( function() {

        api.showSummary( function() {

            api.updateWallet()

        })

    })

})
```

* Callbacks are super powerful of handling async operation in javascript, async programming exist because callback exist.