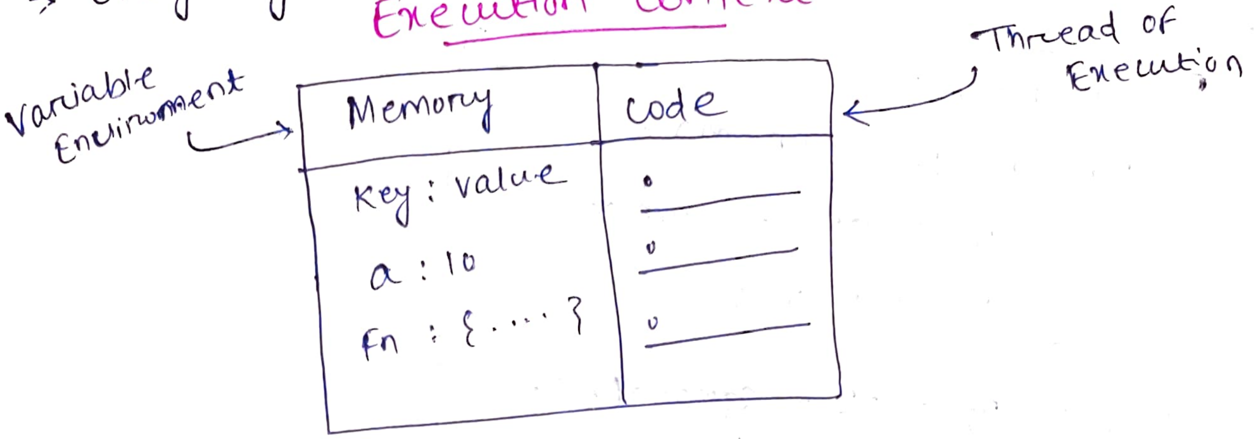
JavaScript

→ 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

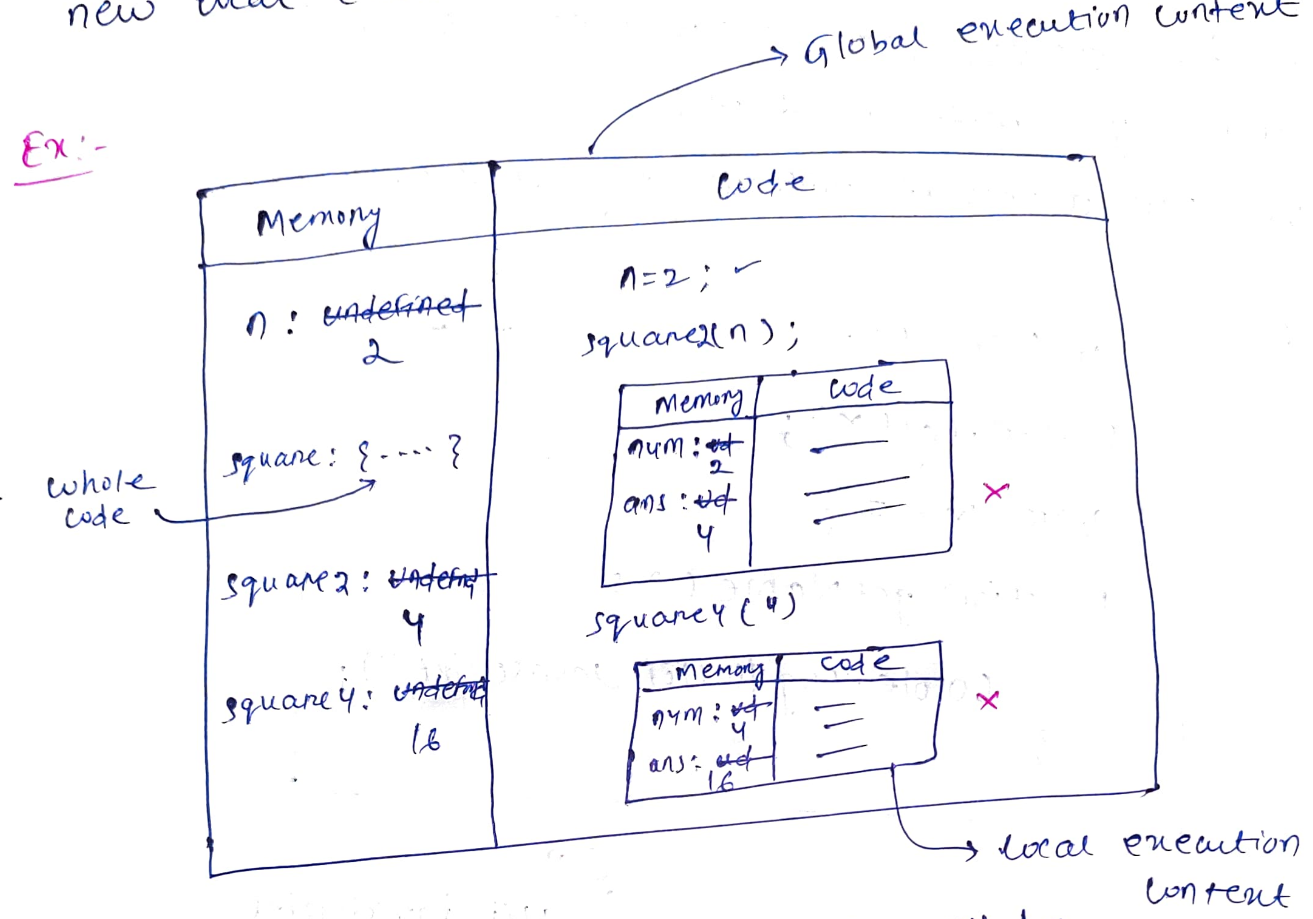
  (a) Memory creation phase - we allocate all the variables with the value undefind. & in case of function it copies the whole Function in the value.

  (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 ~~[illegible]~~ 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 staack
- 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 error.

Ex:-

```
getName();
cosole.log(x);

var x = 7;

function getName(){

        cosole.log("Hii javascript");
   }
```

O/P

```
Hii javascript
Undefind
```

→ if we print the function name

```
cosole.log(getName);

function getName(){
        cosole.log("Tahir");
   }
cosole.log(getName);
```

→ Doubt

O/P

```
f getName() {
        console.log("Tahir");
  }
        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.

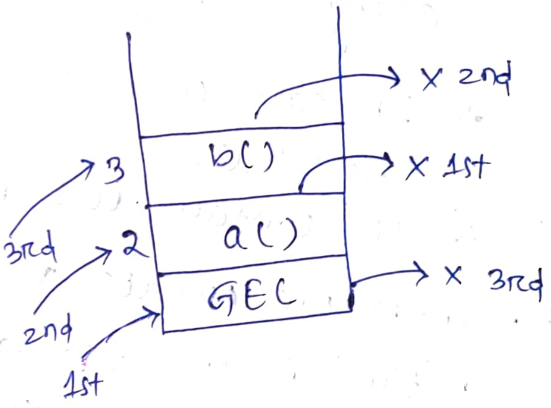
How function works?

```
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 context, 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~~ content & also create window object which is created by javascript 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 browser's 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;
}
console.log(widow.a);
console.log(a);
console.log(this.a);
```

O/P:-

```
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 = "tahin";
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 :-

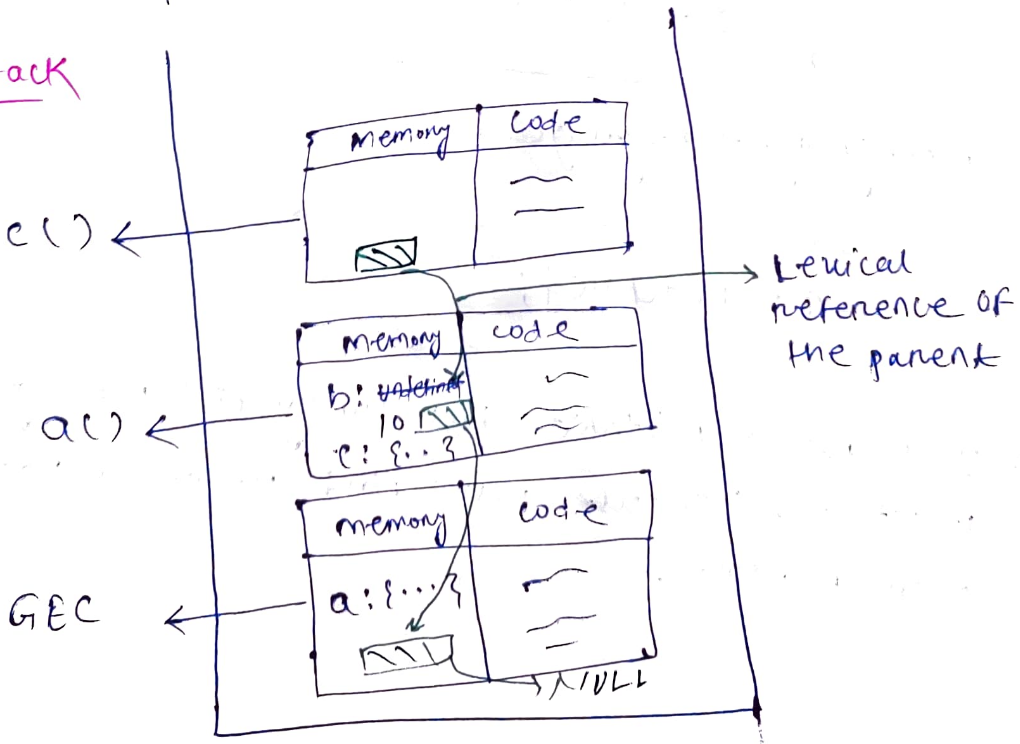
→ Lexical means 'Hierarchy' / 'order'.

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 content & 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 content 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 betn 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.

```
~~var~~ 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