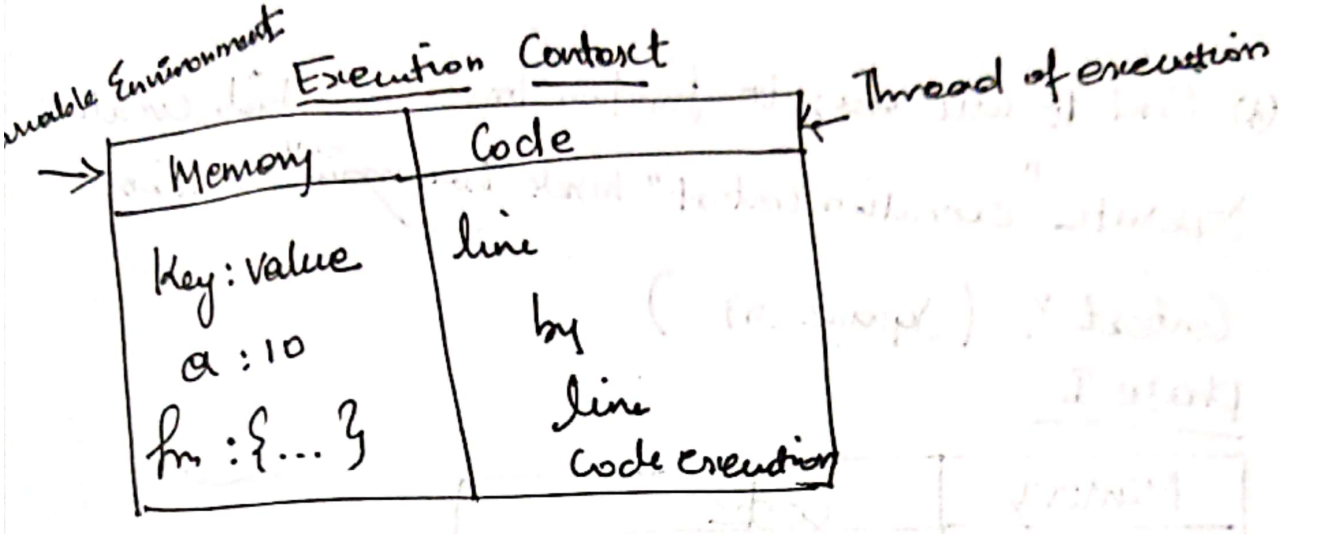
Recap of JS

"Everything in Javascript happens inside an "Execution Context"



(*) Javascript is a Synchronous Single-threaded language

(Synchronous Single threaded)

↳ Code will execute one by one

(*) In the First Phase of Memory Allocation, the Execution Context Store the variable and function both the Special value which is undefined for variable.

(*) For that Function it will Store Whole body inside memory creation

```
Var n=2
function Square (num).
  Var ans = num * n
  return ans;
}
Var Square 2 = Square (n
Var Square 4 = Square
```

(*) In phase two, code execute line by line. Assigning the variable to the given value. Goes to function which is already in Execution Context.

(*) Next it will goes to function invocation which creates seperate "Execution Context" inside the main "Execution Context". (Square(n))

Phase I

Memory	Code
n = 2 Square : {...} Square 2 : undefined Square 4 : undefined	memory | Code num : undefined ans : undefined

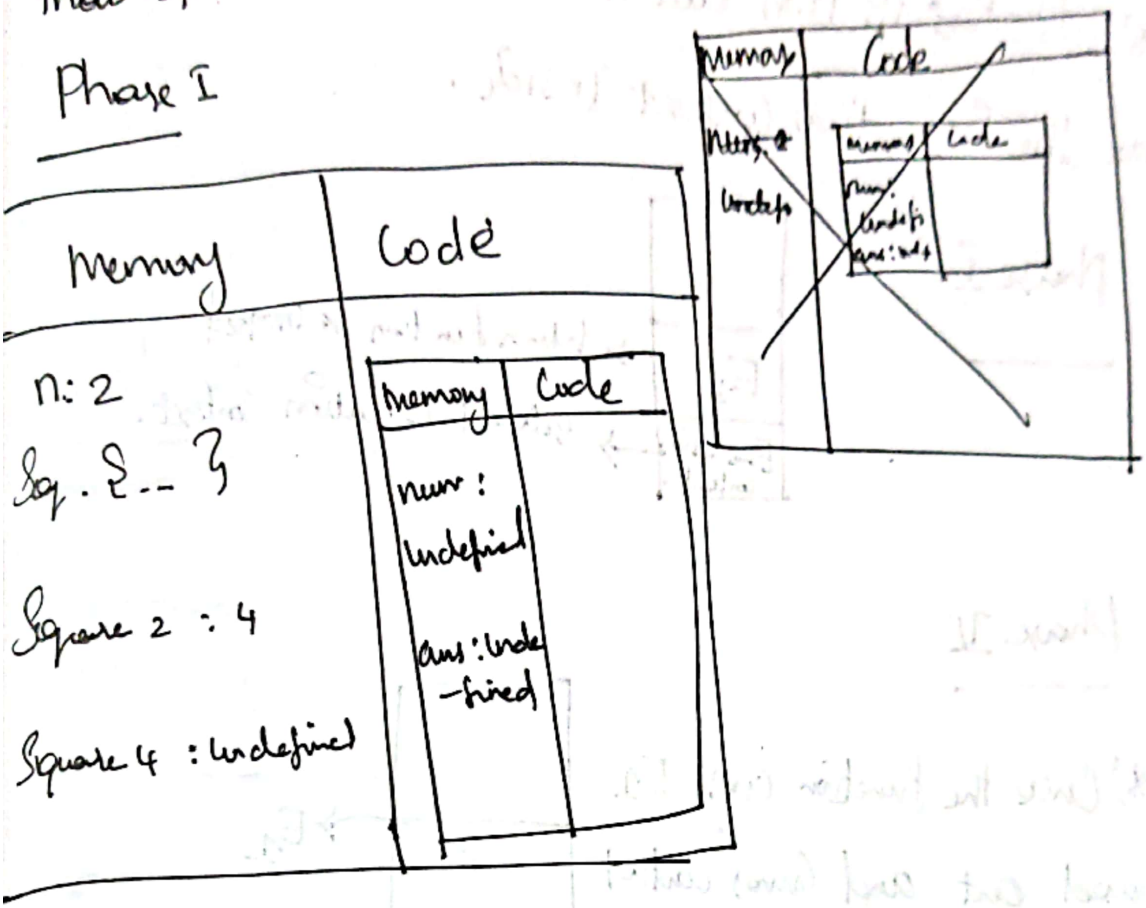
Phase - II

memory	Code
n : 2 Square : {...} Square 2 : 4 Square 4 : undefined	memory | code num : 2 ans : 4

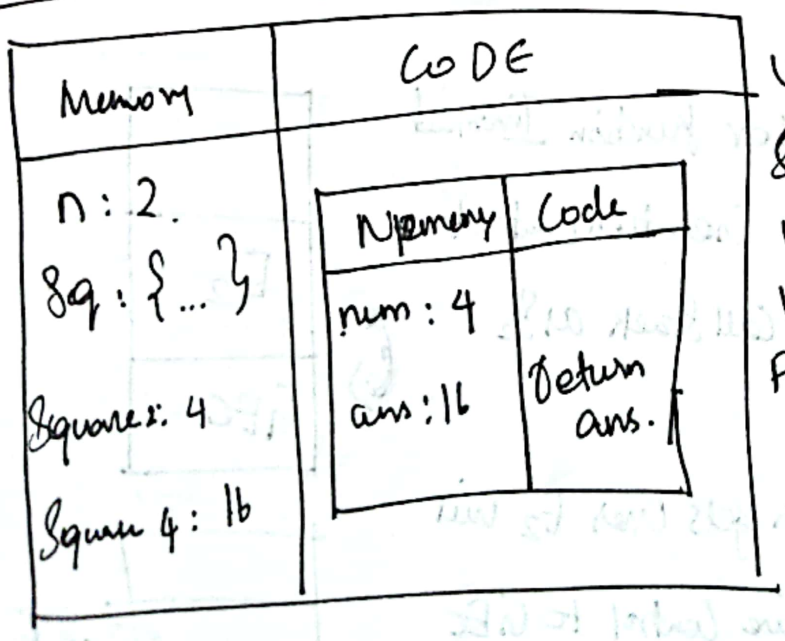
The "Execution Context".

(*) Another Function Invocation is called i.e (Square 4)
It creates Another "Execution Context" with Phase I order that Stores data and execute the Code next.

Phase I



phase II



(*) When return Statement is reached, it will go back to the Value that Stored Function.

(*) After that this "Execution context" be deleted.

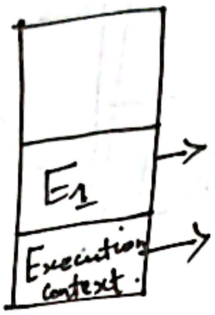
(*) When the Program finished, ...

Context" also get deleted.

Call Stacks

(*) JS has its own Call Stack which used to store the Execution Context Inside.

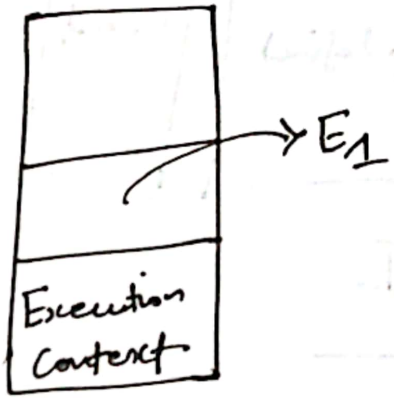
Phase I



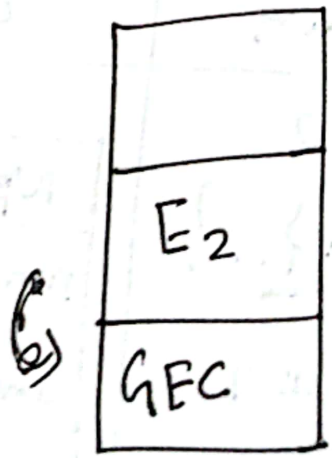
When function is Invoked Global Execution Context.

Phase II

(*) Once the function over, E1 moved out and Gives Control to Global Execution Context



(ii) When another function Invoked It will Create new Execution Context and moved to Call Stack as E_2



(iii) When function gets over E_2 will move out and Give Control to GEC

(*) This is how Javascript manage the execution context creation, deletion and Everything is controlled by Js.

(*) After the execution of whole program, the Call Stack will be empty as there is "no execution context".

Call Stack has many names :-

(i) Execution Context Stack
(ii) Program Stack.
(iii) Control Stack.
(iv) Runtime Stack.
(v) Machine Stack.

③ Hoisting

(*) The variable and function are stored in memory creation before execution of the code.

undefined

~~Where we Initialize value~~
after the call function
it shows undefined

(*) When execution creation created it stores value for variable and functions.

not-defined

(*) When we fail to initialize the value for the particular variable any where in the program it shows not-defined.

```
getname();
Var getname2 = function(){
}

var getName=()=>{
...
}
```

→ undefined

Both function Because the getname are considered as variable

(*) Call Stack execution in editor (not important)

```
var a;
Console.log(a);
if(a === undefined){
 Console.log("a is undefined");
}
else{
   Console.log("a is not undefined");
}
```

Output: → undefined
→ a is undefined

(*) Javascript is a loosely typed language i.e it doesn't attach any variables to the specific datatype

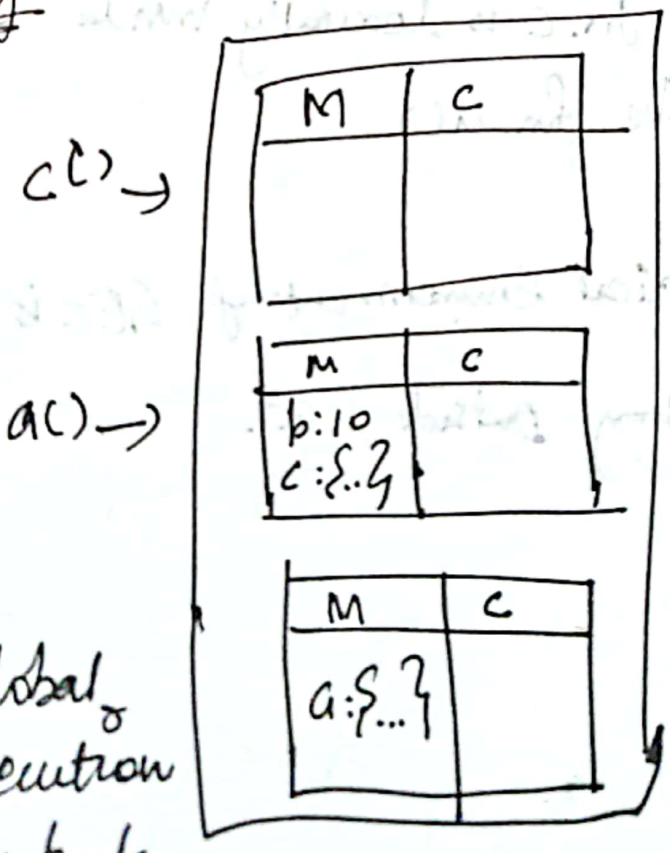
(*) also known as weekly typed language (Learner)

undefined → Its like a placeholder which kept inside the variable and in the whole block of code, that variable doesn't assign any variable.

Scope :- (depends on learcal Environment)

(*) It means where you can access a specific variable (or) function in the code.

eg :-




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

lexical Environment

(*) ~~lexical Environment~~ :

(*) Whenever the Execution Context is created, a lexical Environment also created.

(*) lexical environment is the local memory along with the lexical environment of its parent.

lexical => Hianechy, Sequence of.

Eg:- function a() { } → a() lexically inside the Global Execution Content.

var b=10;

c();

function c() { } → fn. c is lexically inside the fn. a().

}

(*) In that order, the lexical Environment of GE

Null, because there is nothing outside GEC.

Scope

Scope chain :-

(*) Nothing But the Chain of lexical environment and Parent references.

let and Const

(*) let and Const declarations are Hoisted

```
let a = 10;
console.log(a);
var b = 100;
```

(*) Before Execution the memory of a is stored in seperate memory location and b is stored in Global EC because of (let & var) difference.

Temporal dead zone :- (TDZ)

Refers to the time between when a variable is declared using let or const and when it is initialized.

```
let a;
a = 10;
```

→ Time Between these two is called Temporal Dead Zone.

window.b => Show output

window.a => undefined => Block let & const.

(*) Because is "a" is not present in window object it is "Global Execution Context".

(*) Var can allow multiple declaration with the same reference.

Eg:- var b=10;
var b=100;

=> This works fine But when we use "let" instead

Eg:- let a=10;
let a=100;

=> Shows Syntax Error Cause let won't allow same reference in the Scope.

=> Const => Stores in seperate memory location like "let"

But "Const" shows mutalize Immediate af declaration

Const a = 100; => But if we do like "let"

Const a; { Shows Syntax error which shows missing Initialized.
a = 100;

Const b = 100; (Type error) { Shows error as assignment to constant variable.
b = 100;

Type error

=> In above code, Const variable trying to assign new value to b which result In Type error.. Because

=> We cannot modify or reassign value to "Const" variable that is why type error. (Type of b = Const) => ~~Error~~ trying to Change the Type error.

Reference error

Eg:- console.log(a) Shows "Reference error" cannot access 'a' before Initialization.
let a = 1900;

In Short

Const => use when you do reassign another value

let => let is best option as it is used Separate memory And has added feature than Const.

Var => Old, use very Consciously Cause Create problems la

⑧ Closures

(*) Closures is the Combination of a function bundled together to the lexical environment.

other words

(*) function along with the lexical environment is called Closures.

(or)

(*) a Closures gives access to the outer function scope from the Inner function scope.

Eg:

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

var z = x();
console.log(z);
z();
```

Closure.

Uses Of Clos

(*) Modul

(*). Curr

(*) Fun

(*) M

(*) M

(*)

(*)

⑨.

Funct

(*)

fun

e

(*) Module Design Pattern

(*) Currying

(*) Functions like Once

(*) Memoize

(*) Maintaining State in async world.

(*) SetTimeouts.

(*) Iterators

(*) and many more....

⑦.

Function Statement :- (or) Function Declaration

(*) The way of declaring the function is called function Statement.

(or)

(*) When you define a function using function ke

And it gets hoisted.

eg:-

```
fun. greet() {
    Console.log("Hello");
}

greet();
```

Works when yo

Call it before the declaration

(*) When a function is assigned to a variable. Unlike function statements, function expressions are not hoisted

```
greet();
const greet = fn() {
    console.log("Hello");
};
```

It won't work because of its not hoisted

Anonymous Function

(*) Functions without name is called Anonymous fn.

```
Function() {
};
```

Anonymous function.

(*) Anonymous Functions are used when the ~~functions~~ functions are used as values.

```
var b = fn.() {
    ...
};
b();
```

Here calling the function without the name.

(*) If we give name to the function in an function Expression is called named Function Expressions.

eg

```
Var b = function xyz() {
    console.log("b called");
}
a();
b();
xyz(); → Shows error Because fn. xyz is
         Stored as variable b
```

Parameter :-

(Pass)

(*) When we use value (or) variable in the function brackets of the function is called parameter

Parameter

eg:

```
function School (name, age){
    Var Stu1 = name, age ;
    Stu1 =
```

Arguments :-

(*) It is the actual value that you pass to the function when you call it. eg :- School(Jaswanth,

First Class Functions :- (or) First class citizens

(*) The ability to use functions as values.

Eg:- passing functions in function,
return function inside function,
Assigning to the variables.

Arrow functions :- [ES 06]