Javascript

Execution Context:

Everything in javascript happens inside execution context.

We can assume this EC to be a big box or a container in which whole javascript is executed.

→ EC has two components in it.

(1) Memory Component (Variable Environment):

It has variables and functions stored as key value pairs.

(2) Code Component: (Thread of execution)

Here, the code is executed one line at a time.

JavaScript is a Synchronous single threaded language

→ JS can execute one command at a time and in a specific order.

∴ It can only go to the next line once the current line has been finished executing.

What happens when you run JavaScript code?

JS is not possible without Execution context.

When a JS program is run, an execution context is created.

EC is created in 2 phases.

① Memory creation phase ② Code Execution phase

Ex:

```
① var n = 2;
② function square(num) {
③     var ans = num * num;
④     return ans;
⑤ }
⑥ var square2 = square(n);
⑦ var square4 = square(4);
```

→ In Memory creation phase, JS will allocate memory to variables and functions.

So as soon as JS encounters line 1, it will allocate memory to n, and stores a special value known as "undefined"

→ Now JS goes to second line and notices a function named square, allocates memory to it. It literally stores the whole function code in the memory space

→ Now for lines 6, 7; JS allocates the memory and stores them as "undefined"

↓

like a place holder

A special keyword in JS

Memory	code
n: undefined square: {. . .} square2: undefined square4: undefined	

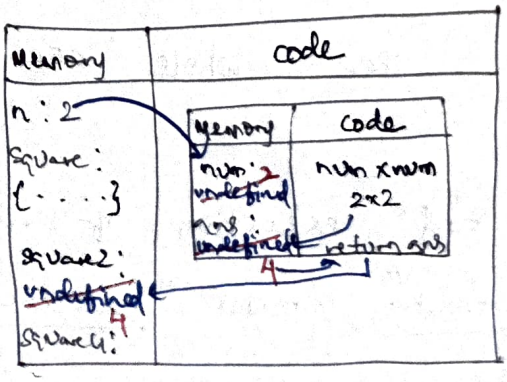
→ In the 2nd phase i.e code execution phase, JS skims through the whole code once again and executes the code now.

→ As soon as JS sees the line ①, it places '2' inside the 'n' [till now the value of n is undefined]

Memory	code
n: ~~undefined~~ 2 square: {. . .} square2: undefined square4: undefined	

→ From line 2 to 5, there is nothing to execute.

→ JS moves to line 6 and founds a function invocation. Whenever a function is invoked a new EC is created.



→ Now, in new EC, in phase 1, mem is allocated to num and ans, undefined val is given to them.

→ When the function is invoked, value of n i.e 2 is passed to num.

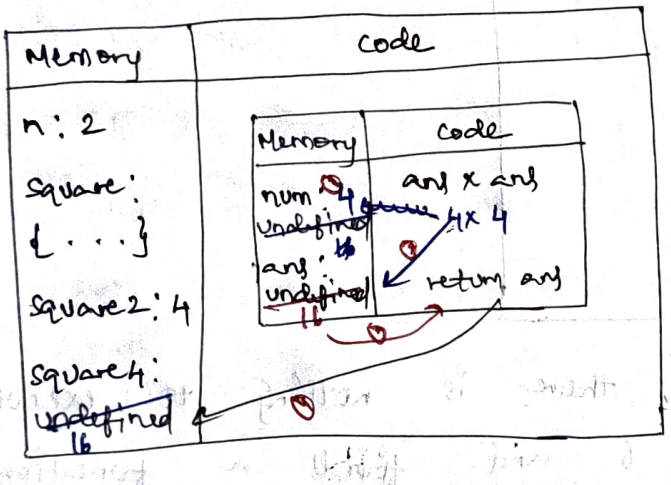
→ Now in code execution phase, num x num is calculated (as 4) and is given to ans.

→ After that 'return ans' is encountered, 'return' tells the function to give back the control to ~~EC~~ program where the function was invoked.

→ So return keyword fetches value '4' from local memory and gives it to squared variable in the memory phase of GEC.

→ When a function finished execution, the newly formed EC will be deleted.

→ Now in line 7, the function is again invoked and a brand new EC is created.



→ ~~EC~~ As soon as function is executed, the newly formed EC is deleted.

→ After line 7 is finished, the whole GEC is also deleted.

→ To manage this creation and execution of all EC, javascript uses a "stack" known as "callstack".

call stack

When ever a JS program is run a GEC is populated inside call stack.

→ After that for line 6, a new E1 is created.

→ Whenever the function is executed, E1 is popped out of it and then E2 enters.

→ After E2 also executed its function, E2 is moved out of stack.

→ After whole program is executed, GEC also popped out of the stack and call stack gets empty.

call stack maintains the order of execution of execution contexts.

call stack is also known as

→ Execution context stack
→ Program stack
→ Control stack
→ Runtime stack
→ Machine stack

Hoisting in JavaScript:

Hoisting is a phenomenon in javaScript by which we can access variables and functions without any error even before initializing them.

Ex:

```
Var n = 10;
function getName(){
    console.log("Namaste JS");
}
getName();
console.log(n);
```

o/p:
Namaste JS
10

Ex:

```
getName();
console.log(n);
Var n = 10;
function getName(){
    console.log("Namaste JS");
}
```

o/p: Namaste JS
undefined

14/08/24

```
console.log(getName);
```

↳ prints the function itself

If we try to access a variable before initializing we get o/p as "undefined", but incase of functions when we try to ~~print~~ ~~that~~ access the function,

```
console.log(a);
```
⟶ o/p: undefined

```
console.log(getName);
```
⟶ prints the function itself

```
var a = 7;
function getName(){
    console.log("Namaste JS");
}
```

This is because,

Even ~~before~~ the whole code is run,

in the memory execution phase of GEC, the variable "x" and the function "getName" are allocated memory and stored as undefined and function code respectively.

not defined vs undefined:

Undefined is a special keyword that is given as a placeholder value to variables in memory execution phase.

```
getName();
console.log(x);
console.log(getName);
function getName(){
    console.log("Namaste JS");
}
```

→ This line gives reference error. x is not defined.
i.e we are trying to access a variable that is no where defined in the program.

In case of an arrow function:

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

var getName = () =>{
    console.log("Namaste JS");
}
```

→ TypeError: getName is not a function

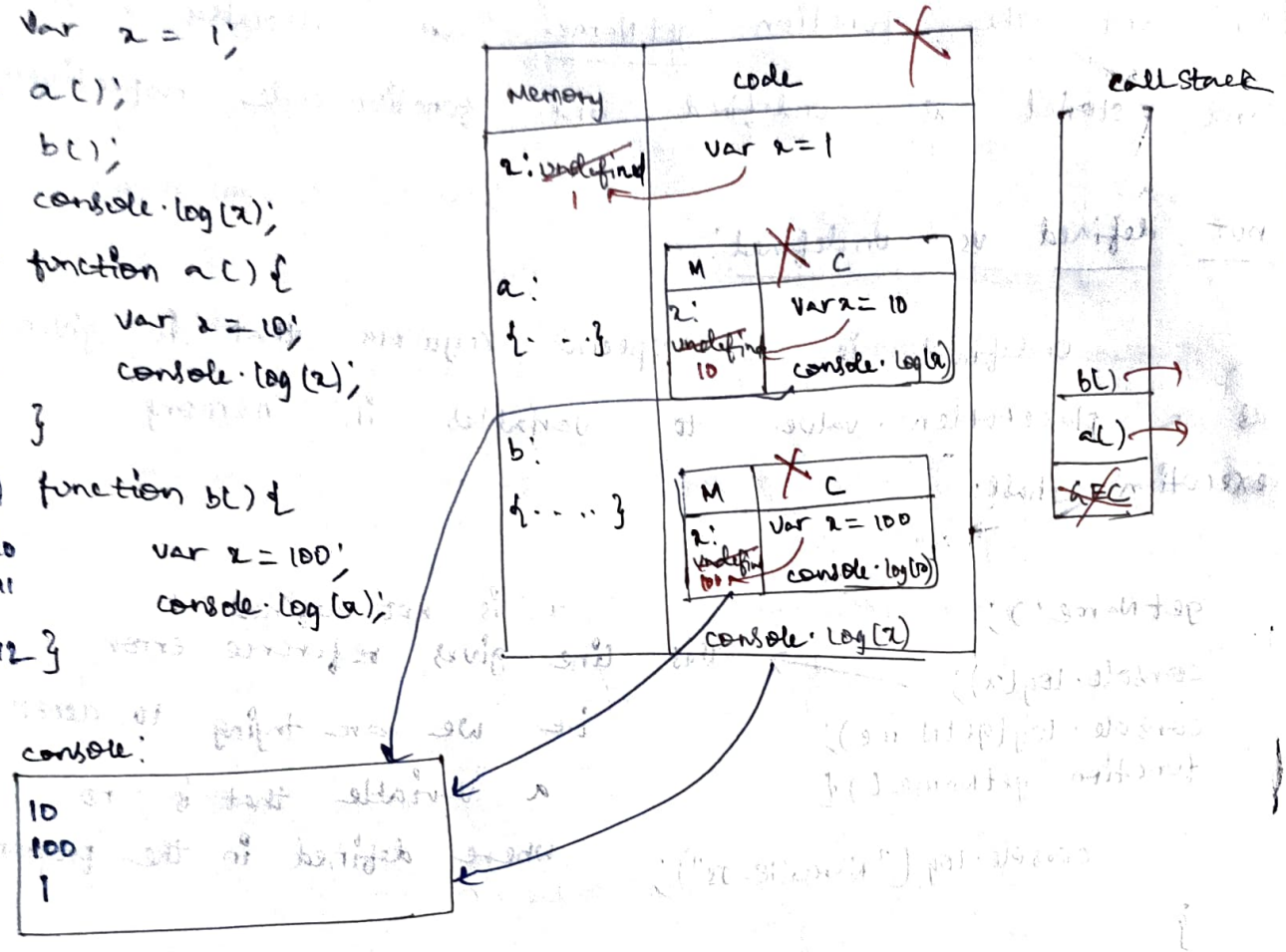
In case of an arrow function, getName is treated as just another variable and "undefined" is given to it, hence the above error.

only in case of general function declaration

i.e `function getName(){ ... }` → Here getName will be treated as a function

How functions work in JS:

```
1  var x = 1;
2  a();
3  b();
4  console.log(x);
5  function a() {
6      var x = 10;
7      console.log(x);
8  }
9  function b() {
10     var x = 100;
11     console.log(x);
12 }
```



→ When the program is run a GEC is created

→ At line 1, x is given as undefined before and now it changes to 1 after executing line 1.

→ Now at line 2, a() EC context is created and it is pushed onto stack.

→ In a()'s execution context, x is undefined initially and when control moves to line 6, x is changed to 100.

→ Then control goes to line 7, from the local memory of a's EC, x is taken as 10 and it logged to console on this line.

→ Now a's EC is popped off the stack.

→ The control goes back to line 2.

→ whenever JS goes to line 3, a new EC is created for b() and it goes into stack.

→ Here also x is given undefined and when line 10 is executed, x is changed to 100.

→ At line 11, x value is taken from local memory of b's EC and is given to console.

→ Now b's EC is also popped off the stack.

→ The control goes to line 3, and nothing to do there.

→ So at line 4, x value is taken from local memory of GEC and it given to the console as 1.

→ Now the GEC also popped off the stack.

Shortest JavaScript program:

The shortest JS program is an empty js file.

Eventhough the file is empty, a global EC is created whenever the file is run.

Along with this GEC, a global object "window" is created.

Also, a "this" keyword is also created.

Global object in case of browsers is known as "window"

At the global level

this === window

∴ this points to global object i.e window

Undefined vs not defined:

```
console.log(a);  ⟶ undefined
var a = 10;
console.log(x);  ⟶ x is not defined
console.log(a);
```

In case of undefined, a will be allocated memory when GEC is created.

But in case of x, memory will not be allocated as it not defined anywhere in the program.

JS is a loosly typed language: (Dynamically Typed)

```
var a;
console.log(a);  ⟶ undefined
a = 10;
console.log(a)   ⟶ 10
a = "Hello World";
console.log(a);  ⟶ HelloWorld
```

In JS, variables are not bound to a specific data type and their type can change during the execution of a program.

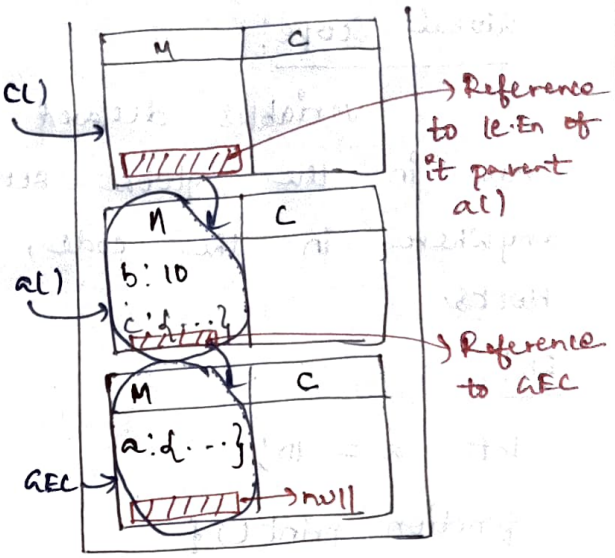
Lexical Environment:

A lexical environment is essentially a structure that holds variable bindings (variable names mapped to their values) and the reference to its parent environment, forming a chain known as scope chain.

Everytime a function is invoked, or a block of code is executed, a new lexical environment is created.

Ex!

```
1 function a(){
2     var b = 10;
3     c();
4     function c(){
5         console.log(b);
6     }
7 }
8 a();
9 console.log(b);
```



Lexical Env = local Memory + Lexical Env of the parent

↓

in a hierarchy / in a sequence

In the above example, we can say that c function is lexically sitting inside a function.

i.e c is physically present inside a.

Where the code is present

→ When JS encounters line 5, it tries to find b in local memory of c. As there is no b available inside c() JS engine goes to the reference of c which is its parent a().

→ b is found in a() and 10 is printed.

Scope:

Scope refers to a region or context where a variable or function is accessible.

Global scope:

variables declared outside of any function or block are in the global scope. These variables are accessible anywhere in the code, including inside functions and blocks.

Ex:

```
let a = 10;
function print() {
    console.log(a); → accessible here
}
print();
console.log(a);
```

Function scope:

variables declared inside a function are in function scope and are accessible within that function and not outside of it.

Ex:

```
function print() {
    let a= 10;
    console.log(a); → accessible here
}
print();
console.log(a); → error: a is not defined
```

Block Scope:

→ Introduced in ES6 with let & const
→ Variables are confined to the block in which they are declared (like inside {} in loops, conditionals)

Ex:

```
if (true) {
    let a = 10;
    console.log(a); → accessible here
}
console.log(a); → error: not defined
```

Lexical Scope:

The scope of a variable is determined by its location in the code (i.e where it is written).

JS uses lexical scoping, meaning that a functions' scope is defined by its location within the nesting of function blocks at the time of writing, not during execution.

Ex:

```
function a(){
    let x = 5;
    function b(){
        console.log(x); → accessible due to lexical scoping
    }
    b();
}
a();
```

Scope chain:

When trying to access a variable, JS first looks in the local scope (function or block). If the value is not found there, it moves up to the next outer scope, continuing this process until it either finds the variable or reaches the global scope.