Namaste Javascript

PART 1

Handwritten Notes

Ashraya KK

Chapter-1

__/__/____

HOW JS WORKS & EXECUTION CONTEXT

- "Everything in JS happens inside ExecutionContext"
- Imagine 'execution context' (EC) as a big box where who JS code is executed.

Execution Context

→ has 2 components

1. Memory component (Variable environment)
   → where all variables & fns are stored as key-value pairs
2. Code component (Thread of execution)
   → where code is executed one line at a time

"JS is a Synchronous Single-threaded language"

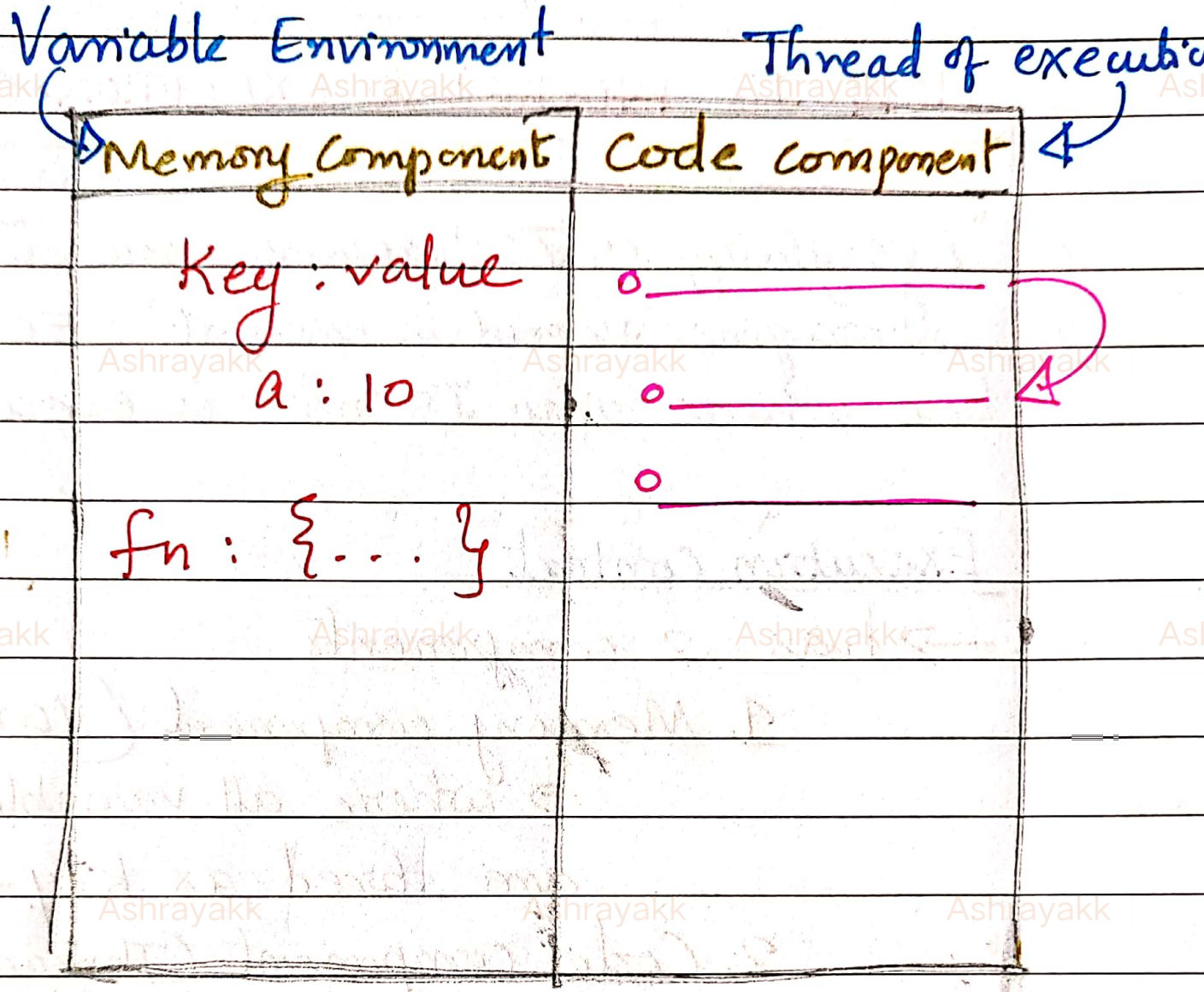
Single-threaded :- JS can only execute one command at a time.

Synchronous :- means in specific order.

That means, it can only go to the next line once the current line has been finished executing.

EXECUTION CONTEXT

__/__/____



CHAPTER-2

__/__/____

HOW JS CODE IS EXECUTED ?

JS doesn't happen without Execution context
When we run a JS program, an execution context is created.

Example

```
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);
```

'num' is parameter of fn

'n' is argument of fn

STEP-1 When I run above JS program, an EC will be created :- MEMORY ALLOCATION PHASE

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

At first phase (memory phase), JS will allocate memory to all variables & functions In first line, it sees 'n', then it will allocate memory space to 'n'. Then goes to line 2, it sees a function 'Square'. Then it allocates memory space to 'square'.

For 'n' → it stores a spacial value undefined

For 'square' → stores the whole code of function

In Memory allocation Phase :-

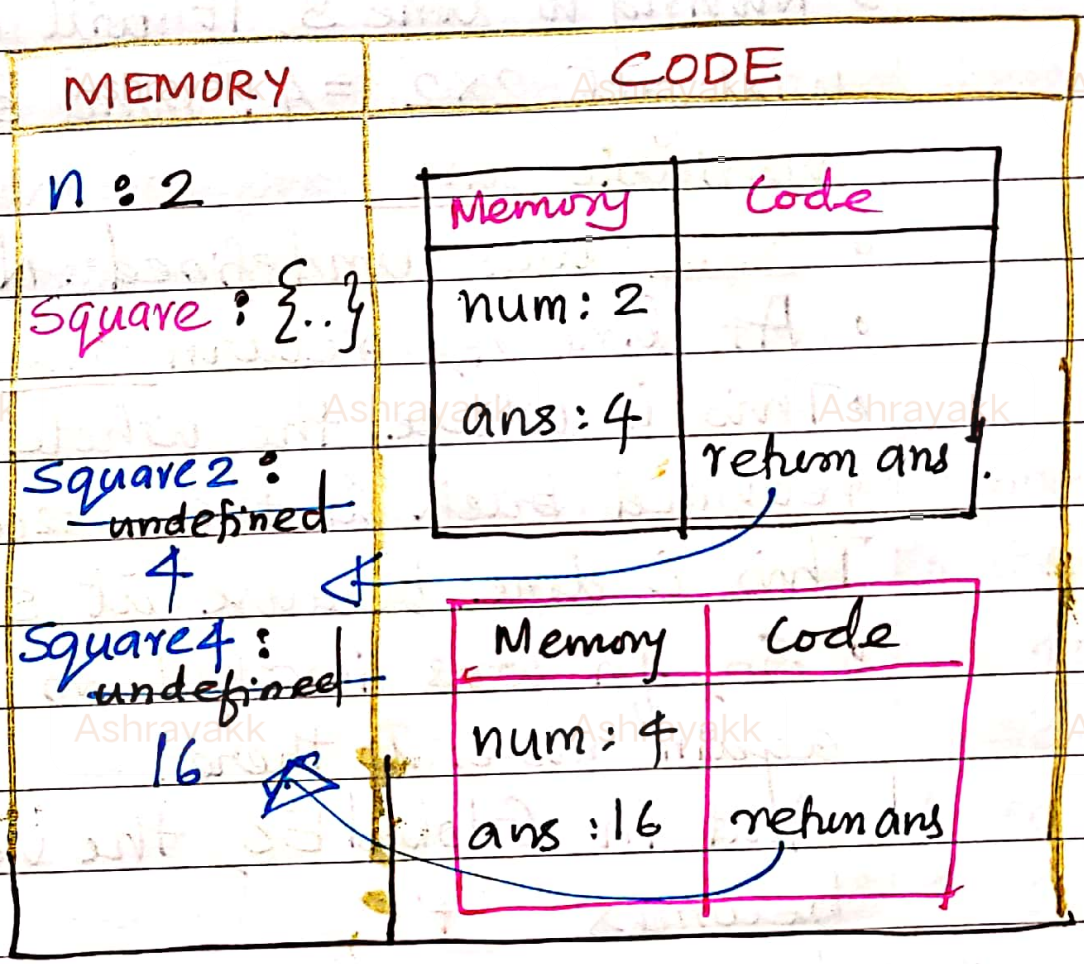
So, for variable, it stores a spacial value called undefined.

For functions, it stores whole function code.

STEP2 CODE EXECUTION PHASE

- Says about how the code is executed after memory allocation.
- Now, JS once again runs through the whole program line by line
- This is where whole calculations & all are done
- In first line, when it encounters n = 2, it actually places 2 inside 'n'. Till now, 'n' was 'undefined'. But in this phase n will become '2'.
- Then it goes to line 2. There is nothing to store. So, it moves to line 6 to see variable 'Square2'. Here comes FUNCTION EVOCATION

Functions are mini programs.
Whenever new function is invoked, a miniprogram is invoked and altogether, new execution context is created.



- In line 6, 'square' fn is invoked. So, a brand new EC is created in phase 2 (code execution.
- So, here in this EC, in memory execution phase, we will allocate memory to 'num' and 'ans'.
- At first, 'undefined' is stored here
- In this EC, for phase 2 (code execution phase)

- The value of 'n' over here, which is '2' is passed to 'num'.
- So, 'num' is the parameter & 'n' is the argument
- Moving to line 3, it will do num x num will get 2x2 = 4. and store in the variable 'ans'
- 'ans' was undefined. Now it become 4.
- At line 4, return ans
- This is where the whole control is again returned back to the execution context. This is done because it saw 'return' keyword
- So, fn was invoked in line 6, and control again moves to there.
- And in Global EC, the value of 'square2' becomes '4'

Now, the whole Local EC created now will be deleted forever. (as soon as we return value).

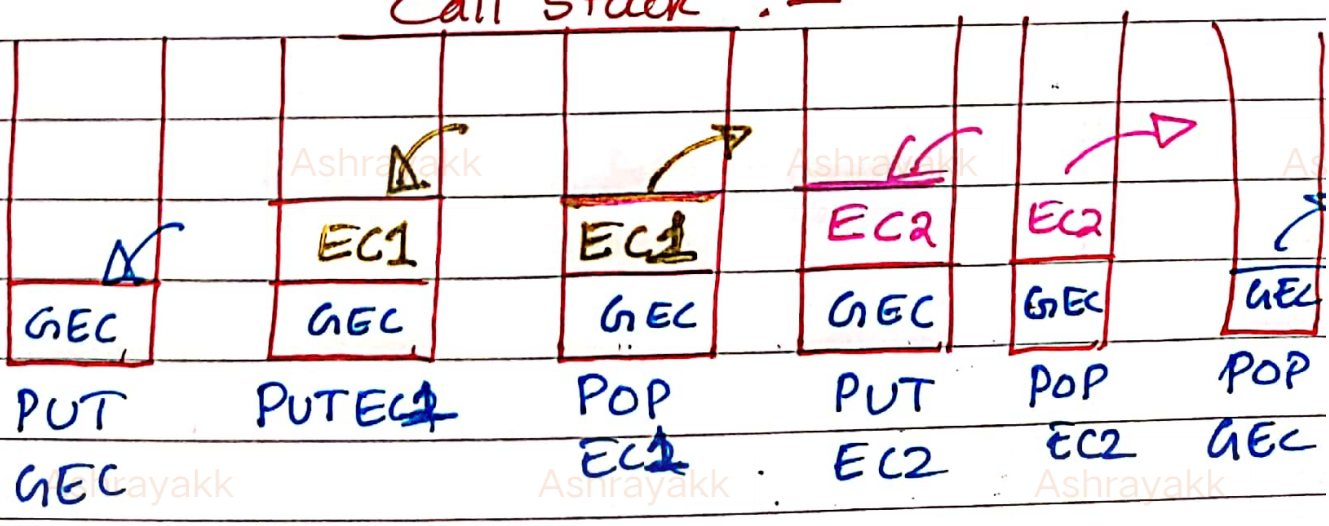
- When we move to line 7, again fn evocation. we get num & ans in that EC which is created. and 'num' directly get value '4'. 'num' & 'ans' will be undefined before but now num is 4. So, code execute. num x num = 4x4 = 16. 'ans' become '16' and return 'ans'. This EC is deleted. 'square4' fn have value 16.

CALL STACK

To handle the creation, deletion of EC, it manages a stack.

- Bottom of stack has GEC (Global Execution Context) all the time
- Whenever any JS program is run this call stack is populated with GEC
- Whenever a fn is invoked (or a new EC is created, that EC is put inside this stack.
- When EC1 is deleted, it will get popped out of the stack & control moved to GEC
- When a new fn is invoked, again a new EC is created (EC2), it is put inside the stack. Once execution is completed, EC2 is popped out & control back to GEC
- After the whole program is executed, GEC is also popped out of callstack

Call stack :—



__/__/____

"

Call Stack maintains the order of execution of execution contexts"

Other names of Call Stack

1. Execution Context Stack
2. Program Stack
3. Control Stack
4. Runtime Stack
5. Machine Stack.

CHAPTER-3

__/__/____

HOISTING IN JAVASCRIPT

Hoisting is a phenomenon in javascript by which we can access variables and functions evenbefore you have initialised it.
Means, we can access it without any error.

Eg:-

```
1. var x = 7;
2. function getName() {
3.     console.log("Namaste Javascript");
4. }
5. getName();
6. console.log(x);
7. console.log(getName);
```

Output

```
Namaste Javascript
7
f getName() {
        console.log("Namaste Javascript");
}
```

__/__/____

Suppose, code is in the below way :-

```
1. getName()
2. console.log(x);
3. console.log(getName);
4. var x = 7;
5. function getName() {
6.     console.log("Namaste JS");
7. }
```

Then, the output will be :- O/p

Namaste Javascript
undefined
ƒ getName() { console.log("Namaste JS"); }

[Suppose, I didn't ~~put the~~ initialised x and removed line 4 completely. Then, I will get error :- 'x is not defined'.]

This is known as hoisting in javascript

Explanation :-

In above code, in the case of variable 'x' we get undefined. but for function 'getName', we get that fn itself.

Whenever a JS program is executed, it will create a Execution context, which have 2 phases.
In the first phase (M/y creation phase), 'x' is allocated a space having value 'undefine
'getName' is a function. So, whole ~~go~~ code is put (Actual copy of that fn.

So, when I run the above code,

1. At first getName() fn executed.
   So, it will print 'Namaste JS' in console
2. Then comes printing 'x'. So, now value of x is undefined. So, it will get printed in console
3. Then printing the function itself in console

What happen when we write functions in different ways ?

1. Arrow Functions

Suppose, the function is like this :-

```
1. getName()
2. Var getName = () => {
       console.log ("Namaste Javascript"); }
```

Now, the o/p will be error. Says 'getName' is not a function

__/__/____

If getName is an arrow function, it behaves as a variable. So, in my allocation phase, at first 'getName' value will be undefined & in code execution phase, when we invoke it, it says, 'getName' is not a function.

2. Another way of fn declaration :-

```
Var getName = function () { .. }
```

In this case also 'getName' behaves as a variable and store 'undefined' initially.

CHAPTER-4

__/__/____

How functions work in JS & Variable Environment

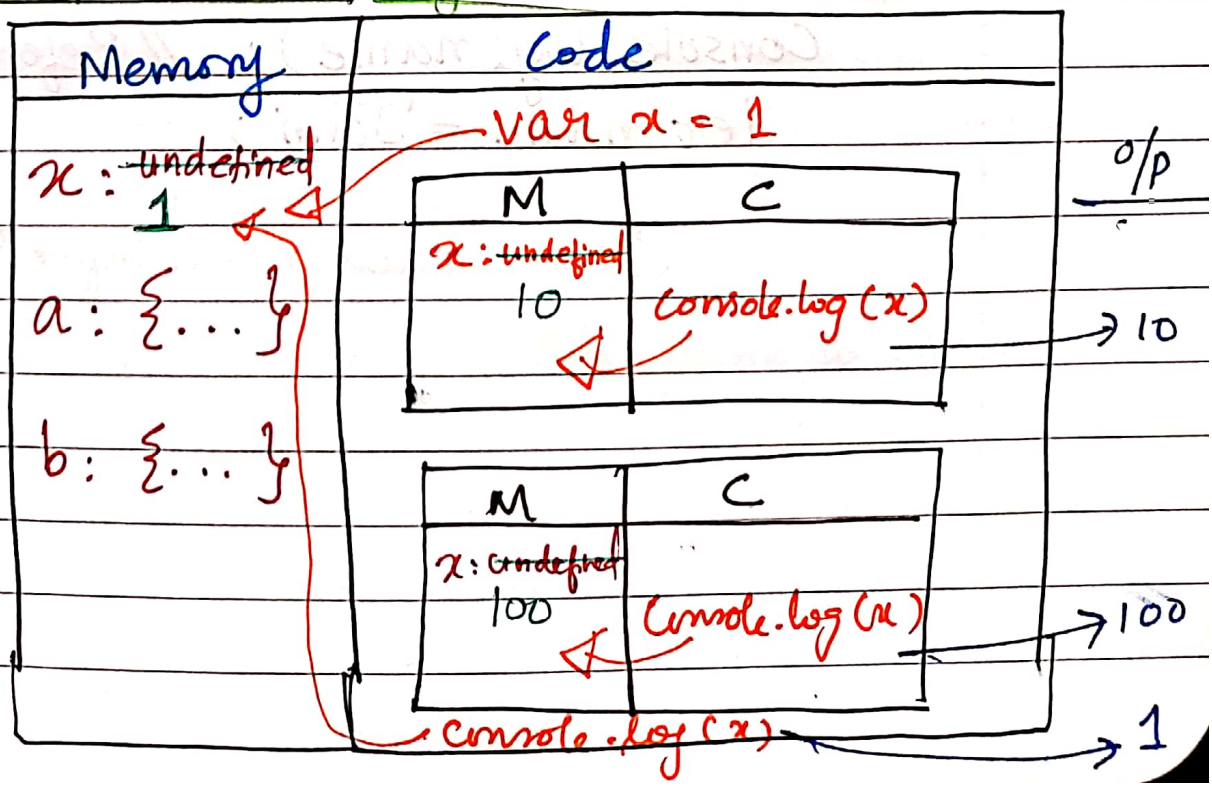
Let us take an example of a JS program :-

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

Output

10
100
1

Explanation Diagram :-



That is how the execution context maintains it's space. Any EC have their own memory space, have virtual environment where they run seperately, independent of each other.

Variable Environment :-

JS Engine scans through the code and creates a new property for each variable in a place called variable environment.

NOTE :-

Variables created with let and const are also hoisted, but they are placed in the temporal dead zone with their initial value set to uninitialized.

So,

```
console.log(name); // Reference Error:
let name = 'John';
```

Cannot access name before initialization.

CHAPTER-5

__/__/____

SHORTEST JS PROGRAM, WINDOW & THIS KEYWORD

A shortest JS program is an empty file. It's because, eventhough it is empty, still JS engine is doing a lot of jobs behind the scenes.

→ When we run an empty file, JS engine creates a global execution context (GEC).

→ It also create something known as window

- Go to console
- write 'window'
- something will comeup (a big object with lot of functions & methods).
- These fns & variables are created by JS engine

→ just like this 'window', JS engine creates 'this' keyword.

At global level, this points to window object

What is Window?

→ window is actually a global object which is created along with the global execution.

→ Along with GEC & window, a this variable is created.

In case of browsers, Global object is known as window

Wherever JS is running, there must be a JS Engine.
Just like in chrome, it is V8.

At Global level, this === window

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

Both are same.

If we don't put anything in front of it assumes that it would be window.a.
ie, it is in the global space

[console.log(a) = console.log(window.a) =
console.log(this.a)]

→ all are same

CHAPTER - 6

__/__/____

UNDEFINED V/S NOT-DEFINED

undefined

→ not equal to empty. It takes memory
→ ~~not~~ It's like a placeholder which is kept for the time being until the variable assign some other value.

JS is a loosly typed language. Means it does not attaches it's variables to any specific data types.

Eg:-

```
var a ;
console.log(a);          // undefined
a=10;
console.log(a);          // 10
a="Hello";
console.log(a);          // Hello
```

O/P

Loosly typed lang are also known as weakly typed language.

Don't do this mistake :-

```
a = undefined;
```

X

Not a good practice to do this. Lead to inconsistency

__/__/____

NOT - DEFINED

	o/p
`console.log(a);`	// undefined
`Var a = 7;`	
`console.log(x);`	// error. x is not defined
`console.log(a);`	// 7.

A variable that has not been declared at all is 'not defined'.

A variable that has been declared but not assigned a value is 'undefined'.

CHAPTER-7

__/__/____

THE SCOPE CHAIN, SCOPE & LEXICAL ENVIRONMENT

SCOPE in JS is directly related to lexical environment.

Lexical env in JS is a data structure that stores the variables & functions that are defined in the current scope & all of the outer space.

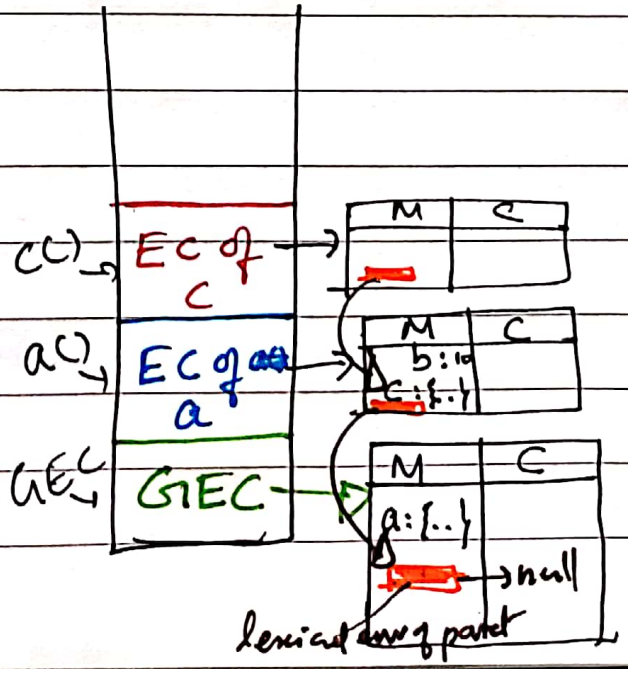
SCOPE means where you can access a specific variable or a function in ~~our~~ the code.

```
eg:- function a() {
        var b = 10;
     }
     a();
     console.log(b); // error : b is not defined
```

Scope of 'b' means where I can access this variable 'b'.

Example with callstack

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



Wherever an execution context is created, a lexical environment is also created.

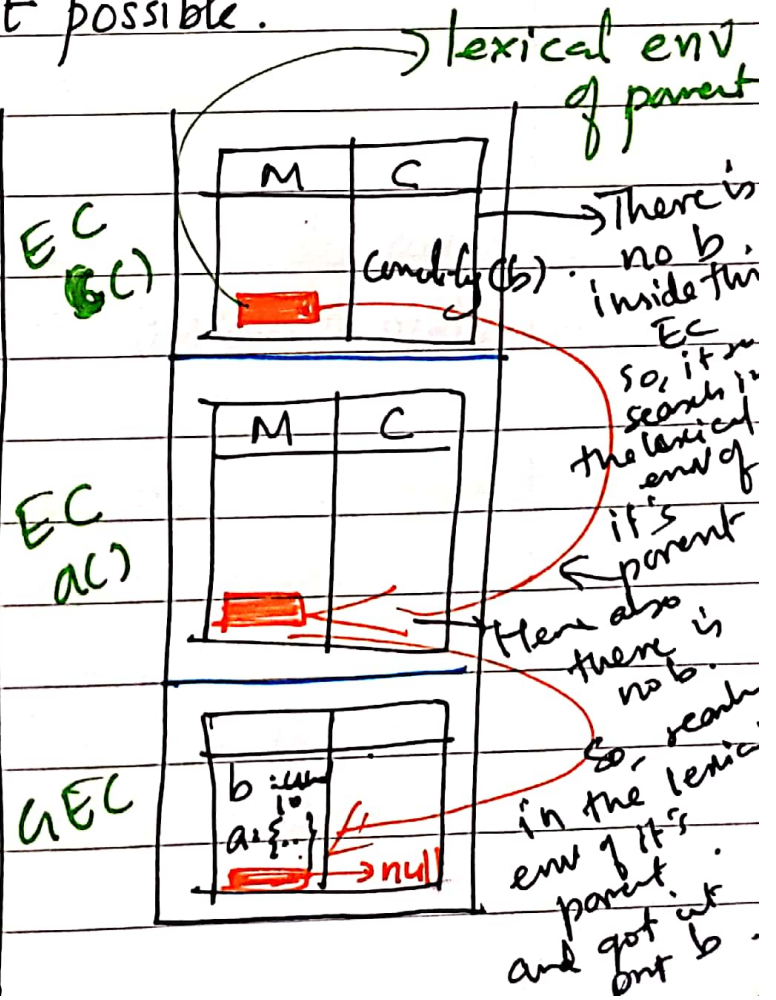
Lexical env is the local memory along with the lexical env of it's parent.

'Lexical' term means 'in hierarchy' or 'in sequence'.

In above eg, we can say, function c() is lexically sitting inside function a().
So, Lexical parent of 'c' is 'a'.
Any variables inside 'a' can be accessed by 'c' but viseversa is not possible.

SCOPE CHAIN :-

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



__/__/____

SCOPE CHAIN

→ Chain of all the lexical environment and the parent references

CHAPTER-8

__/__/____

TEMPORAL DEAD ZONE & LET AND CONST IN JS

- What is a Temporal Dead Zone?
- Are let & const declarations hoisted in JS?
- Diff b/w Syntax error, Reference error and type error.

let & const declarations are HOISTED!!!

These are in the temporal dead zone for a time

Example:

```
console.log(a);   // RefError: Can't access a before initialization
console.log(b);   // undefined
let a = 10;
var b = 100;
```

When we use let, we can only access 'a' after we initialize it (assigns a value in it)

But 'a' is allocated a memory ~~before it is~~ like 'b'; but there is difference.

'a' is stored in a different memory space than Global space (where 'b' is stored).

So, we cannot access 'a' before we put some value in it.

Temporal Dead Zone :-

It is the time since when this 'let' variable was hoisted and till it is initialized some value. That time in b/w is known as the temporal dead zone.

So, whenever we try to access a variable inside temporal dead zone, it gives you a reference error

Eg :-

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

Output

```
// RefError : x is not defined
// RefError : Cannot access 'a' before initialization
// undefined
```

Window.b ⟶ 100

window.a ⟶ undefined

Cannot be accessed because it is in different m/y space.

So, 'let' is more strict than 'var'.

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

Syntax Error :- Identifier 'a' has been declared already.

When there is this error, JS won't execute a single line of code.

'const' is even more strict than 'let'.

'const' → if we are using it, we should initialize it in the same line.

```
Eg:- const b;
     b = 1000;
```
// SyntaxError : Missing initializer in const declaration.

Syntax expects value wherever variable is ~~initialized~~ declared with const.

Syntax Error :- means code doesn't even run
It's says invalid js code. whole code is rejected.

But in let, we can do this :-

```
Eg:- let b;
     b = 10;
```
is possible.

Type Error :-

→ comes when a value is not of expected type.
→ also when attempted to alter a constant value

```
const b = 1000;
b = 10;
```
// Typeerror :- Assignment to constant variable.

This happens because variable 'b' is a const type.

To avoid Temporal dead zone, always put declaration and initialization at the top.
ie, we are shrinking Temp window zone to zero.

CHAPTER-9

__/__/__

BLOCKSCOPE AND SHADOWING IN JS

BLOCK

→ A block is defined by curly braces { }

→ Block is also known as Compound statement

→ A block is used to combine multiple javascript statement into one group.

Eg:-
```
{
    var a = 10;
    console.log(a);
}
```

→ We group it together so that we can use multiple statements in a place where JS expects one statement.

Eg:- writing this `if (true)` only gives error :-

//SyntaxError: Unexpected end of input.

Because it expects one statement. Ex:-

if (true) abc ; } → removes end

But if we want to write multiple statements we can only do it by grouping into one. This can be done using curly braces

__/__/____

BLOCKSCOPE

→ What all variables and functions we can access inside a block is called blockscope.

Example :-

```
{
    var a = 10;          → inside global scope
    let b = 20;          ]
    const c = 30;        ] → inside blockscope
    console.log(a);      // 10
    console.log(b);      // 20
    console.log(c);      // 30
}
console.log(a);          // 10
console.log(b);          // ERROR :
console.log(c);          RefError : b is not defined
```

Output :-

SHADOWING

```
var a = 100;
{
    var a = 10;
    console.log(a);      // 10
}
console.log(a);          // 10
```

O/P

Var a = 10, shadows var a = 100

__/__/____

Var a = 100 was shadowed and var a = 10 also modified var a = 100 and reunited 100 as 10.
This is because both are pointing to same memory location.

```
let b = 100;                    | o/p
{
  let b = 20;
  console.log(b);               // 20
}
console.log(b);                 // 100.
```

This 'b' has block scope.

This 'b' is inside script space.

```
const c = 300;                  | o/p
{
  const c = 80;
  console.log(c);               // 30
}
console.log(c);                 // 300
```

Shadowing is not only the concept of block. It behaves the same way in functions also.

Eg :-

```
const c = 100;
function x() {
    const c = 30;
    console.log(c);
}
x();
console.log(c);
```

O/p

~~100~~
~~30~~
~~30~~

// 30
// 100

Illegal Shadowing :-

```
let a = 20;
{
    var a = 20;
}
```

O/p

Error :-
SyntaxError :-
Identifier 'a' has already been declared

Here, we are trying to shadow a 'let' variable inside ~~'so~~ global scope with a 'var' variable inside block scope

We cannot shadow a 'let' using 'var'.

But can shadow 'var' using 'let'

```
var a = 20;
{ let a = 20; }
```

→ we can shadow like this

CHAPTER-10

__/__/____

CLOSURES IN JS

A closure is the combination of a function bundled together (enclosed) with references to it's surrounding state (the lexical environment)

function + Lexical env. = Closure

↓ bind together

Eg:-

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

O/P

7

In JS, we can pass a function as parameter inside another fn.

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

__/__/____

Closure eg:-

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

Output

```
// f y(){ console.log(a); }
// 7
```

Uses of closures:

1. Module Design Pattern
2. Currying
3. Functions like once
4. memoize
5. maintaining state in async world
6. set Timeouts
7. Iterators
8. Helps in data hiding & encapsulation

All above are possible because of closures.

CHAPTER-11

__/__/____

setTimeout + closures Interview Questions

Interview questions related to closures :-

①

```
function x() {
    var i = 1;
    setTimeout(function() {
        console.log(i);
    }, 3000);
    console.log("Namaste JS");
}
x();
```

Output

Namaste JS

1 ⟩ after 3s.

First, 'Namaste JS' is printed and then after 3sec, '1' is printed.

JS won't wait at 'setTimeout'.

What happens here is :-

`function() { console.log(i) }`

This function inside setTimeout will form a closure. So, this fn remembers reference to 'i' and forms a closure. Means, wherever this fn goes, it takes reference to 'i' along with it.

It will takes this callback fn and stores it into some place and attach the timer to it.

And then, javascript proceed to next line. and prints "Namaste JS".
Once the timer is expired (3000ms is done), it takes out the fn, puts it again to callstack and runs it.
This is how setTimeout works.

(2) Print this in console :-
'1' after 1sec, '2' after '2sec' ...., '5' after '5sec'

Wrong Ans :-

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

O/P

6 1sec
6 after 2s
6 3s
6 4s
6 5s

It is getting printed in correct order, but it's printing '6'. Not 1, 2, 3, 4, 5.

NB :- Even if a function is taken out from it's original scope, it still remembers the lexical environment & get access to those variables

There in above eg, what happens is that :-
When this 'setTimeout' takes this function and stored it somewhere else, attached with a timer, so the function remembers the reference to 'i' and not the value of 'i'.
All these 5 times, it will points to the same reference of i., because the environment for all these functions are same.
When timer expires, it's too late. By that time the value of i will be reaching to 6.
So, i value become 6 and as it points to the reference of i, after each second, '6' will get printed.
This is only because, 'i' is pointing to the same spot in memory.

How to fix this ?

Use 'let' instead of 'var'.

```
for (let i=1 ; i<=5; i++).
```

This will give desired output. Why ?

Because **let has block-scope**

__/__/____

Means, for each and every loop iteration, 'i' is a new variable all together.

So, each time setTimeout function is runned, 'i' has new value in it.

Each time setTimeout method is called, function ~~called a~~ forms a closure with a new variable in it.

By using 'var' and without using 'let', how can we solve this? _____.

Closures will help.

Ans:)

This problem occurs because, the copy of 'i' refers to same memory location. So, we have to give a new copy of 'i' everytime. So, for that we can form a closure

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

O/p

1
2
3
4
5