Episode - 6

6.3.22

Undefined v/s Not Defined in JS

Javascript Code is executed in a unique way ie using creation of execution context.

In the Memory Allocation Phase/Memory Creation Phase all the variables and functions are allocated memory before a single line of code is execution

And we know, undefined is assigned to the variables which are allocated memory.

Then, in the Code execution Phase, the variables are assigned the respective values (in place of undefined) and code is executed line by line.

If in this code execution phase, there is some variable in the code which has not been allocated memory, then on execution of code, it gives an Error saying the variable is not defined

UNDEFINED: It denotes that variable is allocated memory but not assigned any value as yet.

So undefined is a special keyword in JS that is assigned as a placeholder to the variables which are allocated memory [in the memory execution phase]. Later undefined is replaced by the value that is being assigned to it in the code.

NOT DEFINED:

Not defined is an error that denotes that we are trying to access a variable which has not

been allocated memory. This happens when there is no reference found in the memory for the variable that we are trying to access.

Eg ~~#~~

```
console.log(a);

var a = 7;
console.log(a);
console.log(x);
```

Code Example Depicted Undefined & Not Defined

→ O/P: undefined
7
referenceError: x is not defined

What is happening above is : [Explanation of O/P]

① Memory Creation Phase:

Memory	Code
a: undefined	

JS is a loosely typed/weakly typed language

② Code Execution Phase:

Memory	Code
a: ~~undefined~~ 7	

In 1st line : a is printed as undefined
In 2nd line : a is assigned 7 (in place of undefined)
In 3rd line : a is printed as 7

In 4th line, console.log(x) results in Reference error saying x is not defined
[There is no reference to variable x in the memory because x has not been allocated memory]

"What if I never assign any value to a variable in the code?"

Eg: var a;
console.log(a); → O/P: undefined

Actually, this can be explained through Execution Content Creation:

Memory Execution Phase →
[Memory is allocated to a, and a is assigned undefined]

Memory	Code
a: undefined	

Code Execution Phase →
[The code is executed line by line]

Memory	Code
a: undefined (a remains undefined)	

& 1st line → There is simple declaration of a, and a is not assigned any value, so undefined remains assigned to a.

2nd line: console.log(a) → simply prints undefined

So, if you never assign any value to a variable after declaring it, its value remains undefined throughout, ~~until the time~~

Episode - 7

8-3-22

The Scope Chain, Scope & lexical Environment

Scope [What is scope??]

→ The easiest definition of ~~so~~ scope : "Where inside our code can we access a specific variable"

There are 2 aspects of scope [2 ways to talk about scope]

- What is the scope of a variable?
- Is a variable inside scope of something?

~~Code Eno~~

```
function foo()
{ console.log(a)
}
function bar()
{ var a = 3;
  foo();
}
```

→ // foo is not in lexical scope of bar

VS

→ bar is in lexical scope of foo

```
function foo(a)
{ var b = a * 2;
  function bar(c)
  { console.log(a,b,c);
  }
  bar(b*3);
}
foo(2); // 2, 4, 12
```

Code Example for Understanding Scope

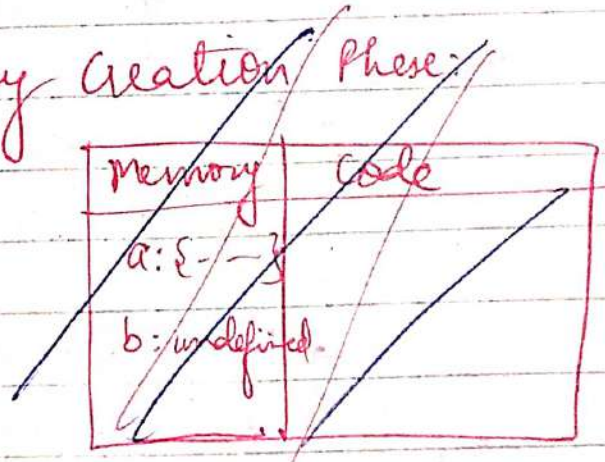
```
function a() {
    console.log(b);
}
var b = 10;
a();
```

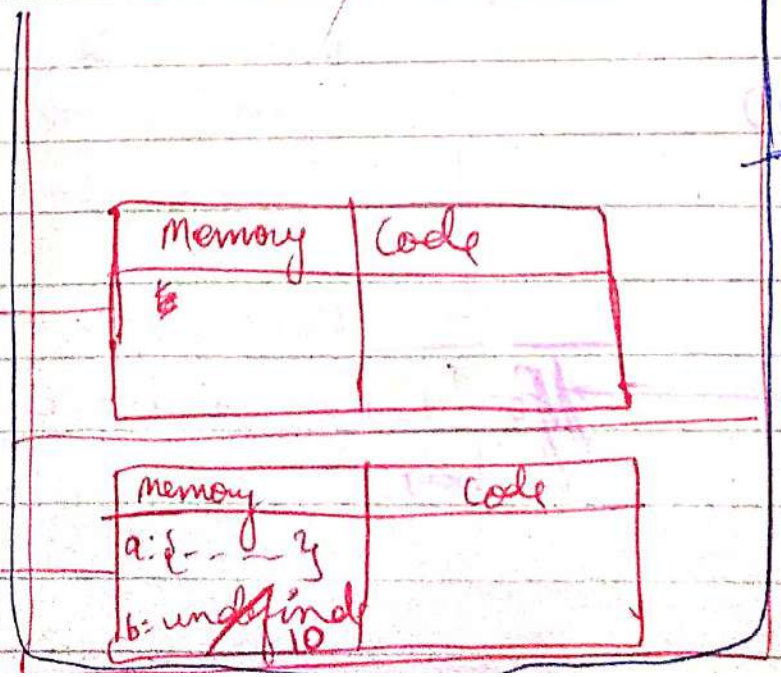
→ O/P : 10

Explanation of O/P : We got O/P = 10, it means we were able to access the global b/s value inside the function a().

Actually, this can be explained using Creation & Execution Context & Call Stack:

→ Memory Creation Phase:





(i) * In the call stack, the Global Execution context is pushed, and in the (i) memory creation phase, a() & b are allocated memory.
* a() points to function body
* b is assigned undefined

(ii) Code Execution phase → then comes function invoketion a(), so a new execution context is created and pushed into stack.

(iii) Inside this new execution context in the memory phase nothing is to be done because there is no variable defined or function defined in a().

```
func a()
{
  console.log(b);
}
```

But in code execution phase, console.log(b) will cause JS engine to look for b in the local scope of a() but it does not find b.
So it goes to the global execution context to find b, and prints b's value as 10.

So we can say that the function a() had access to the global scope, thus it could print b=10

Let us Understand Scope with a little Bigger Example

→ Next page

What will be the O/P? Will it be undefined, or not defined or empty or what?

Ex

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

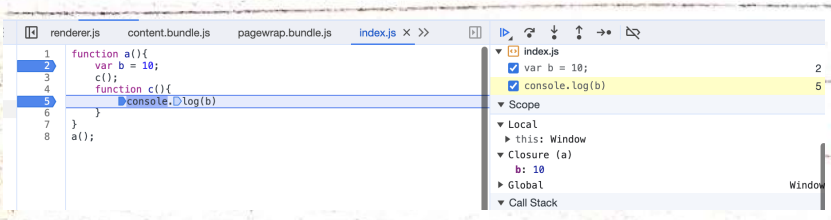
→ O/P : 10

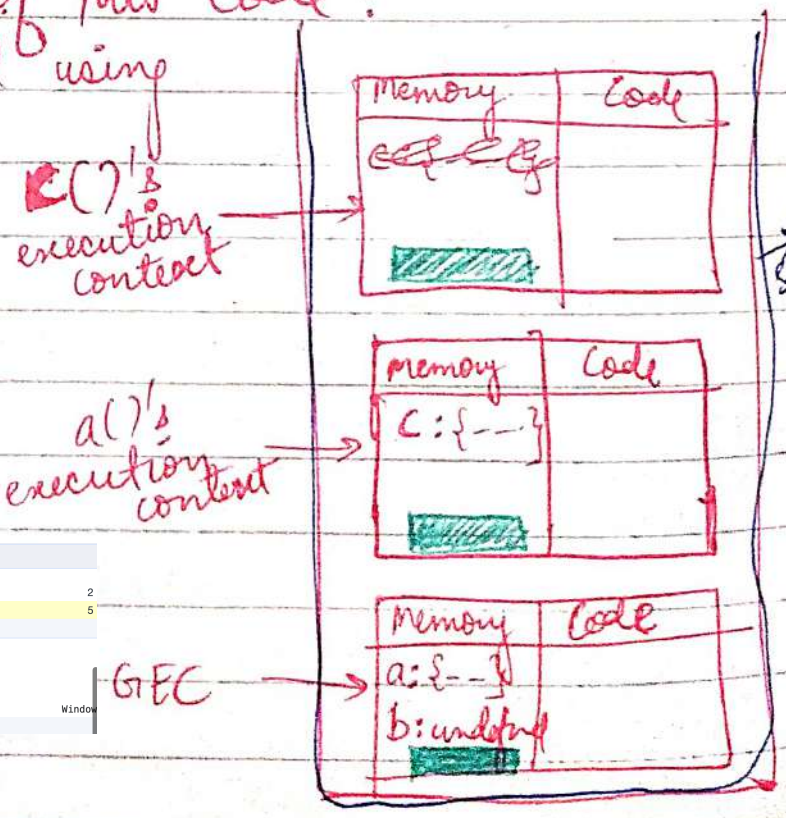
↓

output is again 10

Actually, it means that when we were trying to access global b inside c() which is inside a(), we were able to access it and to print b = 10.

Behind the Scenes of this Code:
(Let us understand using call stack)





We push GEC on the stack.

Memory Creation Phase: a points to function definition, b is assigned undefined

Code Execution Phase: b's values is replaced to 10. The function invocation line → a() (line number 10) cause creation of new execution phase

We push a()'s execution Context on stack

Memory Creation Phase: c() points to function definition of c()

Code Execution Phase: The line ③ is encountered which is a function invocation, so a new execution context is created for c().

We push c()'s execution context on stack:

Memory Creation Phase: Nothing inside c() to which memory should be allocated

Code Execution Phase: console.log(b) is encountered JS engine searches for b inside the local scope of c(), but doesnot find it there.

Then it goes up the scope chain and searches

b in a()'s execution context, but does not find it.

Then it goes up the scope chain and searches b in Global Execution Context, and finds b = 10 so prints 10.

So we can say that function c() had access to the global variable b, ~~and b was in scope of function c(), and~~

"Lexical Environment"

* Whenever an execution context is created, along with it a lexical Environment is also created.
* Lexical Environment of a variable is the local memory along with the lexical environment of its parent.

Meaning of Lexical : It means hierarchical or sequential.

where is it physically placed

If we say c() is lexically inside b(), it means b() contains function c() or we can say c() is present inside b() function

```
function b()
{
    function c()
    {
    }
}
```

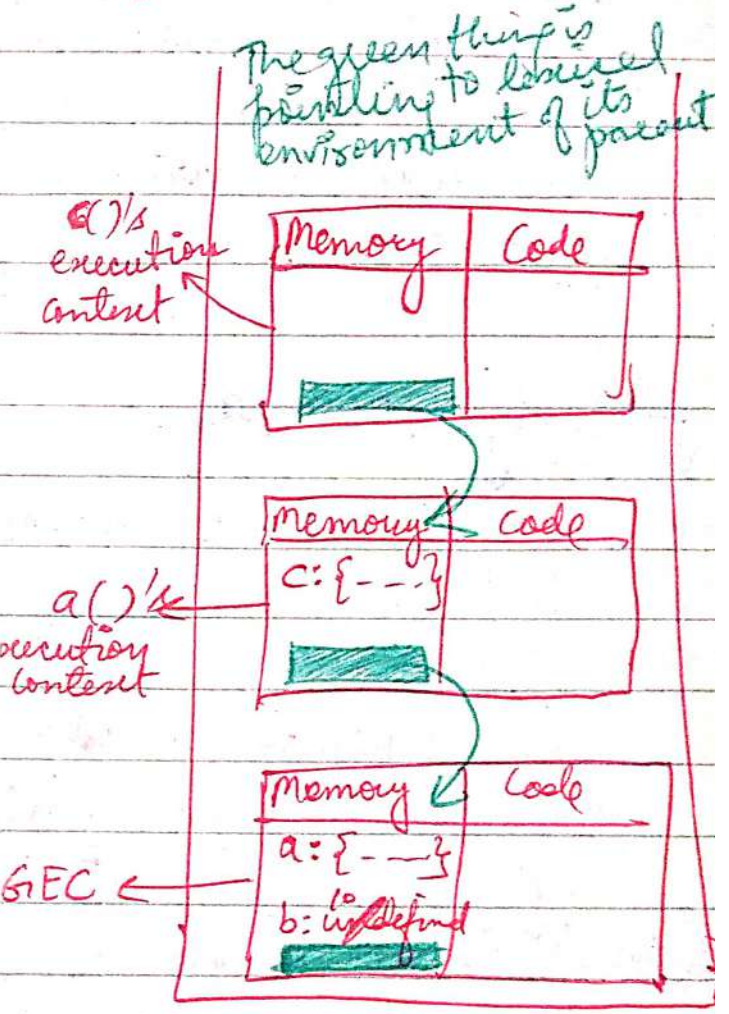
"Lexical Environment in code terms"

So Lexical environment of function c() means & local memory of c() along with lexical environment of c()'s parent ie b().

Let's see this visually using the previous code example, where c() was inside a().

What is Parent?
If x is defined within y then, y is Parent of x

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



When b is not found in c()'s execution context, then JS goes to the lexical env. of c()'s parent ie a()'s execution context

So c() gets access to lexical environment of a() also.

Now lexical environment of a() is the memory component of a() which includes local memory

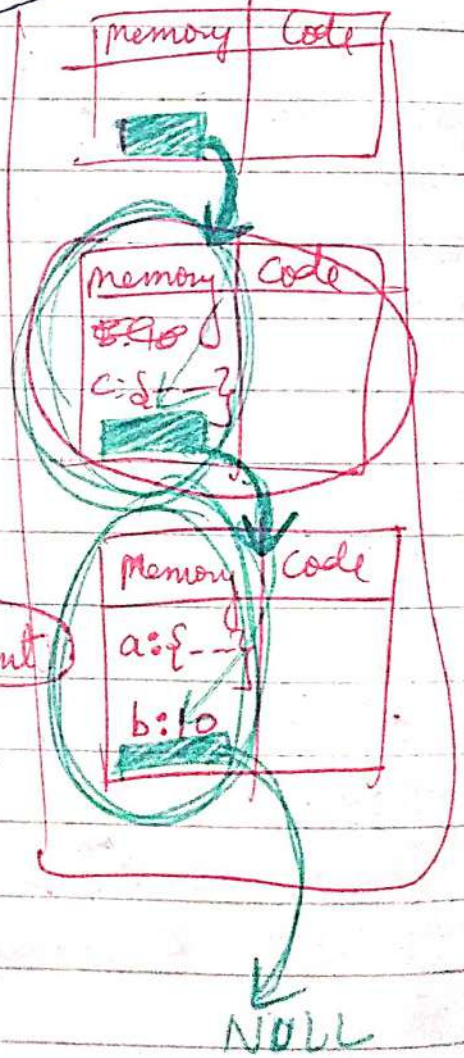
of a() along with the green part (▬) which points to the lexical environment of a's parent i.e. global execution context. So a() has access to global execution context's lexical environment as well as variable b.

The global execution context's lexical environment includes its own local memory along with the lexical env. of its parent. The lexical environment of Global Execution Context points to NULL

Do this after that

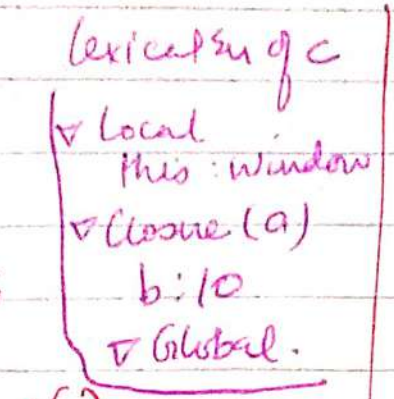
"What if we had never declared b anywhere inside our code?"

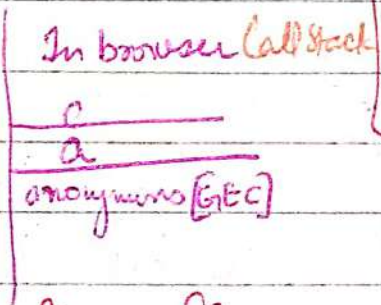
Then, when we would have gone to the lexical environment of (Global Execution Context's parent) (which points to NULL). So when we encounter NULL, it means that the (b is not defined) [because we could not find b in any lexical environment].

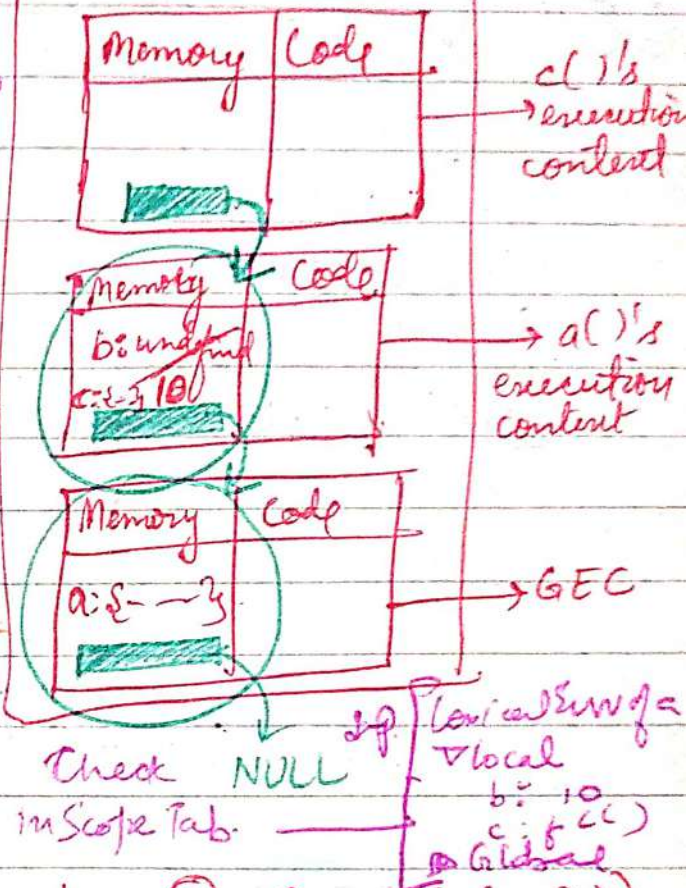


"How Scope & lexical Environment work in Javascript"

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







* When JS encounters line number ⑦ (console.log(b)), it finds b in the local scope of function c() but does not find b there. So JS engine goes to the green part ( ) which has the reference to lexical environment of c()'s parent.
* So we go to lexical env. of a() [which includes the local memory of a() along with lexical env. of a()'s parent. Now JS engine finds b inside local memory of a() and prints b = 10
finds b, and control goes back to c() and prints b = 10

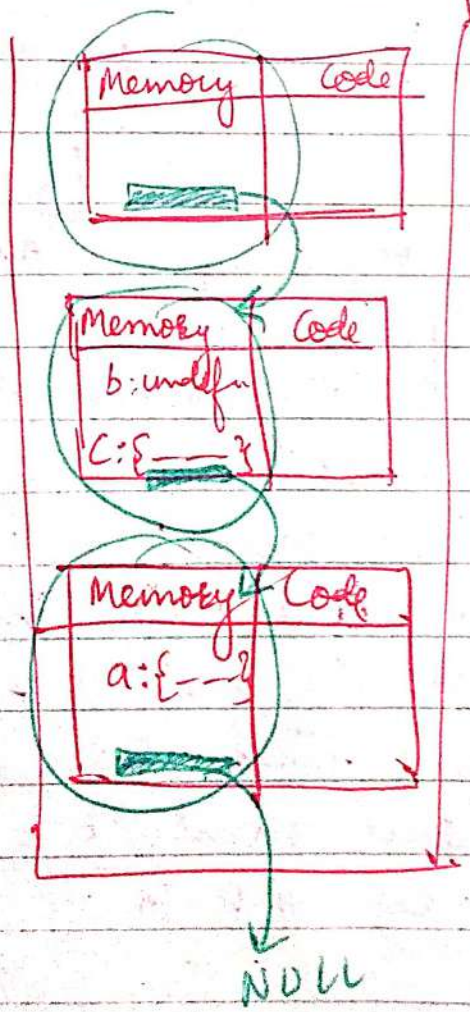
"The Scope Chain in JavaScript"

Def: Scope chain is the chain of all the lexical environments and reference to parent's lexical environment.

If the ~~local mem~~ JS does not find anything in the local memory, it goes on the next level of the scope chain i.e. it goes to the lexical environment of the parent, and so on till it finds the required ~~var~~ thing.

If the thing is not found even in the global Execution Context, then control goes to the lexical env of GEC's parent [which actually has NULL]. So it means, the variable was never declared in the entire code :)

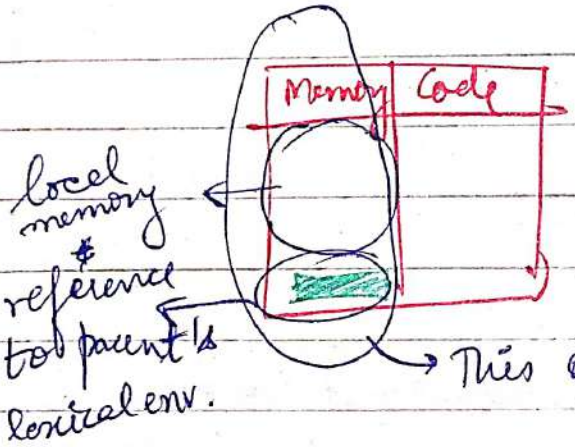
[So if the scope chain gets exhausted, & it means the thing we we trying to access is not present in the code



"A Quick Revision of Scope & Lexical Environment & the Scope Chain"

* **Scope**: Where a variable can be accessed inside our code.

* **Lexical Environment**: → It is created whenever an Execution Context is created.
  → It includes (local memory + reference to lexical env. of its parent)



→ If we say c() is lexically inside a(), it means c's parent is a, and c() is sitting inside a().

→ Lexical Parent → Very Imp

* **Scope Chain**: The entire chain of lexical environments & is known as the scope chain
  → The scope chain decides whether a variable or function is present in the scope or not
  → If the scope chain is exhausted & the variable is not found, it means the variable has never been declared in the code.

[Also see the browser demo of & Lexical environment & scope chain in Inspect]

Episode - 8

10.3.22

"let & const in JS Temporal Dead zone"

The first core concept:

* Are let and const ~~the~~ declerations hoisted in JS?

Ans: let & const declerations are hoisted in JS, but they are hoisted very differently than var declerations.

Code Example to diff. b/w let & var hoisting:

Eg:
```
console.log(b);
let a = 10; // let decleration & initialisation
var b = 100; // var decleration & initialisation
```

↳ o/p: undefined

This happens because of hoisting of var variables, so we can access them even before they are declered & initialised.

In the memory execution phase, b is allocated memory and assigned undefined.

In code execution phase, b = undefined is printed

"But what happens with let a ??"

a is a "let" variable, so it has some different hoisting mechanism behind the

scenes.

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

Here → O/P: a cannot be accessed before being initialised, "Reference Error"

→ This happens because we are trying to access a (let variable) even before initialising it.

→ And we cannot access let variables before they are initialised. This is because before initialisation the let variables are in the Temporal Dead Zone.

→ As far as hoisting is concerned, let variable "a" is also hoisted [ie. it is also allocated memory even before the code execution starts & and a is assigned undefined] → But a will be ~~allocated~~ stored in a different memory called Script. a is not stored in the global object (in which b is stored).

→ ~~And we cannot~~ So, we can say let & const variables are also hoisted [ie. allocated memory and assigned undefined], but they are stored in a separate memory space than the var variables

And we cannot access these let & const variables from this memory space unless they are initialised. If we try to access them before initalisation, we get "Reference Error: Cannot access before initialisation"

var	let	const
Hoisting		
→ Hoisted in JS so allocated memory & assigned undefined even before execution starts.	→ Hoisted in JS (same like var)	→ Hoisted in JS (same like var)
Storage		
→ Stored in the global object called window	→ Stored in a separate memory space called Script	→ (same as let)
Accessing Them		
→ Can be accessed before initialising, O/P: undefined	→ Cannot be accessed before initialising them. O/P: ReferenceError: "Cannot access before initialising"	→ Cannot be accessed before initialising them. O/P: ReferenceError: "Cannot access before initial

** V Imp

functions first

Both function declarations & variable declarations are hoisted. But a ~~subtle~~ subtle detail (that can show up in code with multiple "duplicate" declarations) is that functions are hoisted first & then variables

Consider:

Example 1

```
1  foo();  // 1
2
3  var foo;
4  function foo()
5  { console.log(1);
6  }
7  foo = function(){
8    console.log(2);
9  }
```

If you put a debugger next to line 1., u will notice, that in memory creation phase, there was an entry of foo created in Global with value of

```
function foo()
{ console.log(1)
}
```

Example 2

```
foo();

function foo()
{ console.log(1);
}

var foo;

~~function foo~~
foo = function(){
    console.log(2)
};
```

Same in this example as well.

for example 1, var foo was duplicate (& thus ignored) declaration, even though it came before the function foo()... declaration, because function declarations are hoisted before normal variables.

While duplicate/multiple var declarations are effectively ignored, subsequent function declarations do override the previous ones

```
foo(); // 3

function foo() {
    console.log(1);
}

var foo = function() {
    console.log(2);
}

function foo() {
    console.log(3);
}
```

```
console.log(foo);

var a = 10;
var a = 20
var a = 30

var foo = 3;
console.log(foo); // 3
var foo = 6;
var foo = 10;
console.log(foo); // 10
```

While this all may sound like nothing more than interesting academic trivia, it highlights the fact that duplicate definitions in the same scope are a really bad idea & will often lead to confusing results

"Correct Way to access let & const variables"

Eg
```
let a=10;
console.log(a);
var b=100;
```
→ O/P: 10

This is because we are accessing the let variable "a" after initialising it.

Behind the scene:

[Memory Creation Phase]

In Scope (In Browser)

Script: a: undefined

Global: b: undefined

[Code Execution Phase]

In Scope (In Browser)

Script: a: 10

Global: b: 100

So now we can access "a" from script because a has been initialised. So, it prints 10 in the console.

```
let a;
console.log(a); // Uncaught SyntaxError: Identifier 'a' has already been declared
var a;
```

```
var a;
console.log(a)
let a;
```

"Temporal Dead Zone in JS"

→ Temporal Dead zone is the time difference between the hoisting of let variable and the initialisation of let variable.

→ So before the initialisation happens, the let & const are said to be in Temporal Dead Zone.

→ It is the phase between hoisting and initialisation.

→ When let & const variables are in Temporal Dead Zone, they cannot be accessed.

→ If we try to access them during Temporal Dead Zone, it gives a Reference Error saying cannot access it before initialization.

Eg ~~① console.log(a); ② let a = 10;~~

① console.log(a);
⋮
④ let a = 10;
⑤ var b = 100;

Everything before line number ④ is in the Temporal Dead Zone.

→ The initialisation of a heppens in line ④

* Here, console.log(a) is line number ① which is before line number ④, so

we are accessing a ~~and~~ when it is in the Temporal Dead zone. So we get the ReferenceError.

"Reference Error Explained in Depth"

① If we try to access a variable that has never been defined in the code:
→ ReferenceError: x is not defined

Eg: console.log(x); // we are trying to access x which has never been defined ~~be~~ anywhere in the code
let a = 10;
let b = 100;
↳ O/P: ReferenceError: x is not defined

Memory allocation phase:

Global:

Memory	Code
~~global~~ b: undefined	

Script

Memory	Code
~~Script~~ a: undefined	

Code Execution Phase:

Global object:

Memory	Code
b: 100	

Script:

Memory	Code
a: 10	

When the line console.log(x) is encountered, JS finds/searches but cannot find x there, so it says

& there is no reference of x (or x is not in the memory), so x is not defined

(2) When we try to access let & const variable while it is in Temporal Dead Zone

→ ReferenceError: cannot access a before initialisation

↓

Eg: As seen before

Relation of Global Object (Window) and Variables var, let & const

[Global Object & var]

(1) → Whenever we create a var variable in the JS code, it gets attached to the global object i.e. Window.

Eg:
```
let a = 10
var b = 100;
console.log(window.b);
```
→ o/p: 100

Now in the Memory Creation phase, along with the Global Execution Content a Global Object → Window is also created And all the var variables are attached to this Window Object.

[global object and let & const]

(2) → But this is not the case with let & const variables:

```
let a = 10
console.log(window.a);
```
→ O/P: undefined

Explanation of O/P: Actually, the let & const variables are not attached to the global window object. They are present in some seperate storage. So if we try to access them in global object, we get undefined as the output because a does not exist on the window object <<Doubt>>

Sol: Whenever we refer to a key of an object that does not exist, we get undefined

(3) "this" at global level and let & const 4 var

→ Same case with "this". [this is same as window, at the global level]

```
console.log(this.b)  ⟶  O/P: 100
console.log(this.a)  ⟶  O/P: undefined
```
(Because a is not attached to "this")

Conclusions about let & const:

* We can't access let & const variables using window object or this keyword at global level

* We see let & const are a bit more strict than var

when we try to access 'a' on the 'this' object and the key 'a' does not exist, it returns undefined

Duplicate Declaration / Redeclaration of let & const variables in JS

* let & const are stricter than var.
This can be shown by the following fact:

"If we try to redeclare let or const variables, a Syntax Error is popped up and JS does not even run the code! It stops upfront and displays the error."

* <Redeclaration of let> → Syntax Error

Eg

```
console.log("Hello World");
console.log("Hi There");

let a = 10;
let a = 100; // Redeclaration of let variable a
```

O/P: Identifier

O/P: SyntaxError: Identifier 'a' has already been declared.

This is because we are trying to redeclare let variable in the same scope.

Eg

```
let a = 10;
function c()
{ let a = 100;
}
```

→ This does not give Syntax Error because we are redeclaring let in diff scope.

* <Redeclaration of var> ⟶ No error

Eg:
```
console.log("Hello World");
console.log("Hi There");

var b = 10;
var b = 100; //Redeclaration of var variables
console.log(b);
```

O/P: Hello World
Hi There
100

Now, no error is raised because we can redeclare var variables.

* <Redeclaration of const> ⟶ gives Syntax Error

Eg:
```
console.log("Hello World");
const a = 10;
const a = 100;
```

O/P:→ SyntaxError: Identifier 'a' has been already declared

This raises a syntax error because we are redeclaring a constant variable. The syntax of JSD does not allow this

* <Initialisation of Const later in the code> ⟶ gives Type Error

Eg:
```
console.log("Hello World");
const a = 6;
a = 100;
```

O/P: "TypeError: Assignment to constant variable"

This raises a typeerror because we are trying to reassign a value to const type variable

Important Diff. Between let & const

let :

```
let a;
a=10;
```
→ This is totally acceptable

↓

In this case, no error occurs because let is less strict.
It allows initialization later in the program.

const :

```
const b;
b=10;
```
→ This raises a syntax error

* const is very strict, it does not allow simple declaration of b without initialising it.
* It raises "Syntax Error" saying Missing Initializer in const declaration.

↙

This means it expected that syntax should be followed (it will be initialised along with declaration) i.e. const b = 1000; but actually, the syntax was violated.

Diff. b/w Syntax Error vs TypeError vs Reference Error

Syntax Error

① → When we don't initialise the const after declaring it, this is a missing syntax. So Syntax Error arises

```
const b;
b=10;
```

② → When we try to redeclare a let variable in the same scope again Syntax Error arises, saying "it has already been declared"

```
let a=19;
let a=20;
```

↓ This is a problem with the syntax of code

TypeError

→ It appears when we try to assign a value to a constant variable

Because we are assigning a value to const type

```
const b=10;
b=1000;
```

Reference Error

→ When JS engine searches for a specific variable inside the memory space, but we cannot access it or it does not exist in the memory, then Reference Error occurs.

Eg ①

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

↓

Here we are trying to access [a] while it is in the Temporal Dead Zone.

o/p: Reference Error: Cannot access 'a' before initialization

Eg ②

```
console.log(x)
```

↓

We are trying to access x, that does not exist in the memory because it has never been declared.

o/p: Refer Error: Not Defined

What to prefer – const, let or var ?

→ Always prefer using const because it is the most strict out of all 3

Whenever you want to put a value that won't change later & won't be needed to be redefined, & use const

→ If const can't be used in a situation, then try to use let (as it is stricter than var, so prefer let over var)

Let will have a temporal dead zone (as opposed to var), so we will not run into unexpected errors like undefined etc.

→ Use var as less as possible, keep it aside. It should be used very rarely

[Because var is the least strict & can lead us into unexpected errors]

Conclusion

* Prefer using const the most then let and use var very rarely.

const > let > var

How to avoid Temporal Dead zone?

→ Try writing all declaration as well as initialization lines at the very top of the code to avoid any kind of error caused by accessing let & const while they are in Temporal Dead zone.

→ So, actually, we are shrinking the Temporal Dead zone window to size = 0. This avoids any errors caused due to Temporal Dead zone.

Interview Question - Hoisting of let & const

→ If the interviewer asks whether let & const are hoisted explain:

↳ Yes, they are hoisted, and are allocated memory in a separate memory space (other than global object)

↳ The phase b/w allocation of memory & initialisation is called Temporal Dead Zone

↳ We can't access let & const variables while they are in the Temporal Dead Zone. This causes ReferenceError.

Episode - 9

16.3.22 | What is a block? How is it diff. from Scope?

→ A block is defined as a set of stetements enclosed within { }.

```
{
//---
//---
}
```
→ This is what is a block.

→ We use a block when we want to combine multiple JS stetements into one group.

Eg

```
{
var a=10;
console.log(a);
}
```
→ This is a block

→ Block is also known as a Compound Statement

But why will we need ~~compound~~ to combine multiple JS stetements?

→ Sometimes we need to combine stetements into one group to make a block, so that we can use multiple stetements in a place where JS expects only one statement. Let's understand with an exam

Eg `if (true)`] → O/P: 'SyntaxError: Unexpected end of input'

Why so? Because 'if' expects a single statement after the closing brace. But we usually

→ But we usually have to write a number of statements inside if. This is the situation where we use block **

** Syntax

```
if (true)
{
  let a = 10;
  console.log(a);
}
```

→ This entire block helps us place multiple statements in a place where just one statement could be placed

→ `if (true) console.log(true);` → This is the normal syntax of if, it needs just one statement.

But not always is the case that we place just one statement in 'if'.

What is Block Scope?

→ What all variables & functions we can access inside this block is the block scope.

```
{
}
```

Let us understand how hoisting works for const, var, let inside a {block}
And how ~~block scope~~ they behave behind the scenes:

```
① {
②     var a = 10;
③     let b = 20;
④     const c = 30;
⑤ }
```

Behind the scenes, when we open Inspect and put debugger on line ②, and open scope, we see that a block scope is created, a global object is created.

Block ▼
- b: undefined
- c: undefined

→ b & c are placed in block scope

Global ▼
- a: undefined → a is placed in global

* let and const variables are hoisted & allocated memory in a separate memory space (Block Scope), and are assigned undefined.

* var variable is also hoisted & allocated memory in the global ~~execution context~~ scope or global object.

→ This is the reason why let & const are block scoped. They are stored in a separate memory space reserved for the block. And once we are out of the block, we can't access the let & const variables.

~~* Once we~~

Eg:

```
{
  var a = 10;
  let b = 20;
  const c = 30;
}
```

→ Suppose we reach here in execution of code. Now, the block scope made for the block will be destroyed/deleted and so will be let & const variables b and c. [See Scope in Inspect to see that Block Scope vanishes] put debugger ~~to~~ outside '}'

scope

This was behind the scenes, now let us prove that let & const are block scoped using a code example:

```
{: ① { var a = 10;
   ② let b = 20;
   ③ const c = 30;
   ④ console.log(a);
   ⑤ console.log(b);
   ⑥ console.log(c);
   ⑦ }
   ⑧ console.log(a);
   ⑨ console.log(b);
   ⑩ console.log(c);
```

O/P: 10
20
30
10
ReferenceError: b is not defined (and now the program is stopped)

Explanation of O/P: So the 3 console.log() inside the block print 10, 20, 30 for a, b, c respectively.
Then control reaches end of block, and we reach line no. 8, and print 10 (for a)

But at next line ie line no. 9, we are trying to access let variable b which is not in scope any more. b was in block scope which has been deleted now, so b is not in the memory. This pops up a reference error message saying b is not defined

Vaishnavi

and code execution comes to a halt.

So we can say as soon as we come out of block, let & const variables declared inside the block cannot be accessed from outside the block, as they were confined to the "Block Scope" memory space which has now been deleted.

1.5.22 **Shadowing in JS**

If we have a same named variable as we have inside the block as we have inside the block and we try to access the variable from outside, inner variable shadows the outer variable.

Shadowing [In case of var]

```
① var a = 100;
② {
③   var a = 10; //block
④   let b = 20;
⑤   const c = 30;
⑥   console.log(a);
⑦   console.log(b);
⑧   console.log(c);
⑨ }
⑩ console.log(a);
```

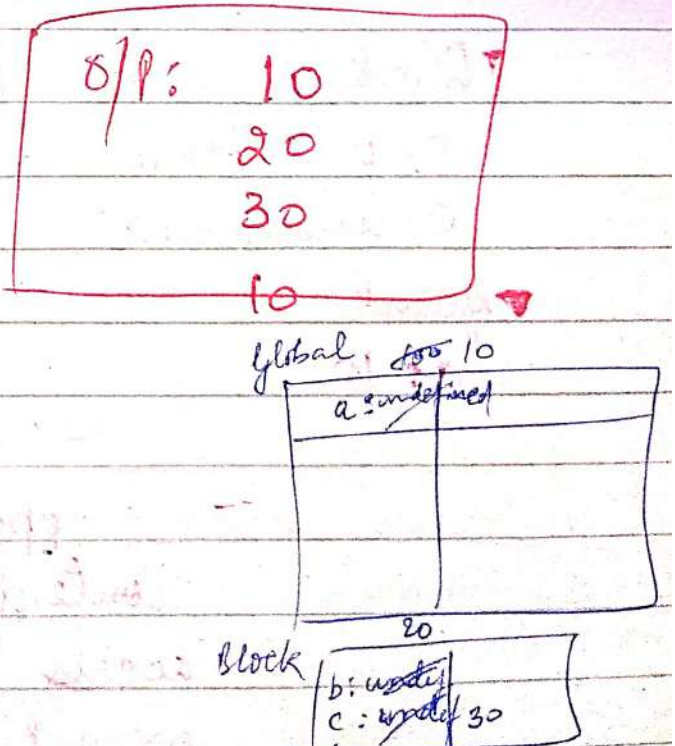
o/p: 10 20 30 10

Explanation:

Here, when we print a from inside the block, the var a inside the block shadowed that

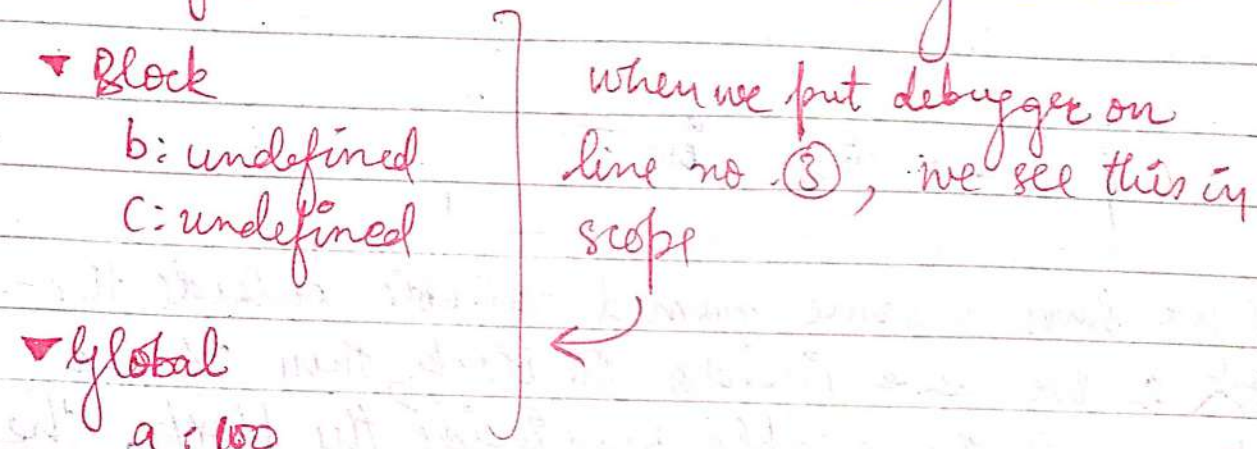
If we try to redeclare a 'var' variable, we don't create a new variable taking separate space in the memory. Rather, both of them point to the same memory location

```
var a = 10; console.log(a);
var a = 5;  (Path) o/p: 5
```

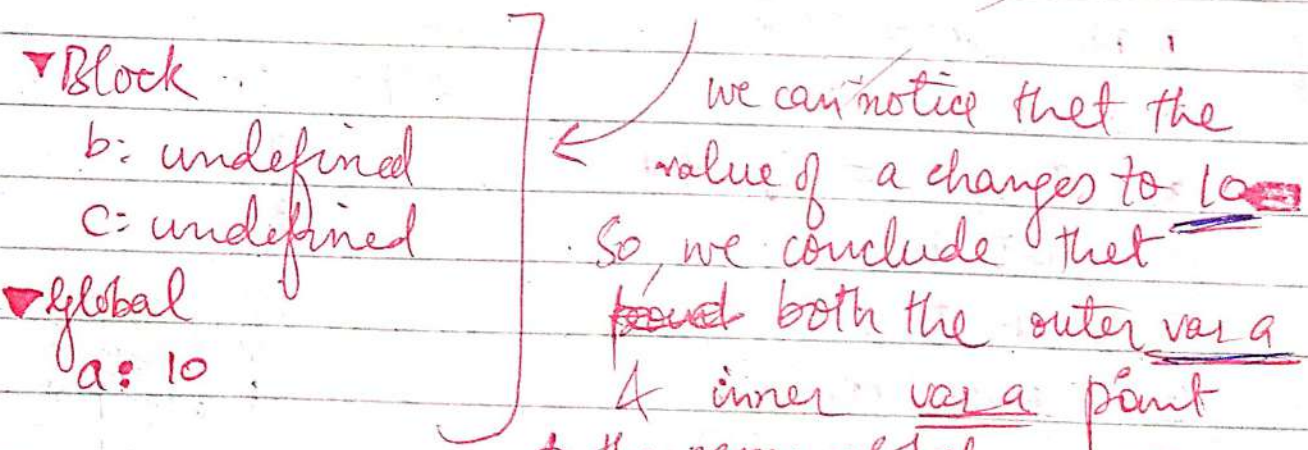


outside the block. So 10 is printed instead of 100

And even if we console.log(a) outside the block, the value 10 is printed instead of 100. Because in the background, var a = 10 is actually modifying the var a = 100 because they both actually point to the same memory location.



Then when we run line no ③ :



So we conclude that both the outer var a & inner var a point to the same global memory space in the Global Execution context. So, now, even if we access "a" from outside the block, we get a = 10 printed.

Shadowing in let & const :

[In case of let]

```
① let b = 100;
② { //Block starts here

③ var a = 10;
④ let b = 20;
⑤ const c = 30;
⑥ console.log(a);
⑦ console.log(b); //accessing b inside the block
⑧ console.log(c);
⑨ }
⑩ console.log(b); //accessing b outside the block
```

O/P: 10
20
30
100

Explanation of Code & O/P:

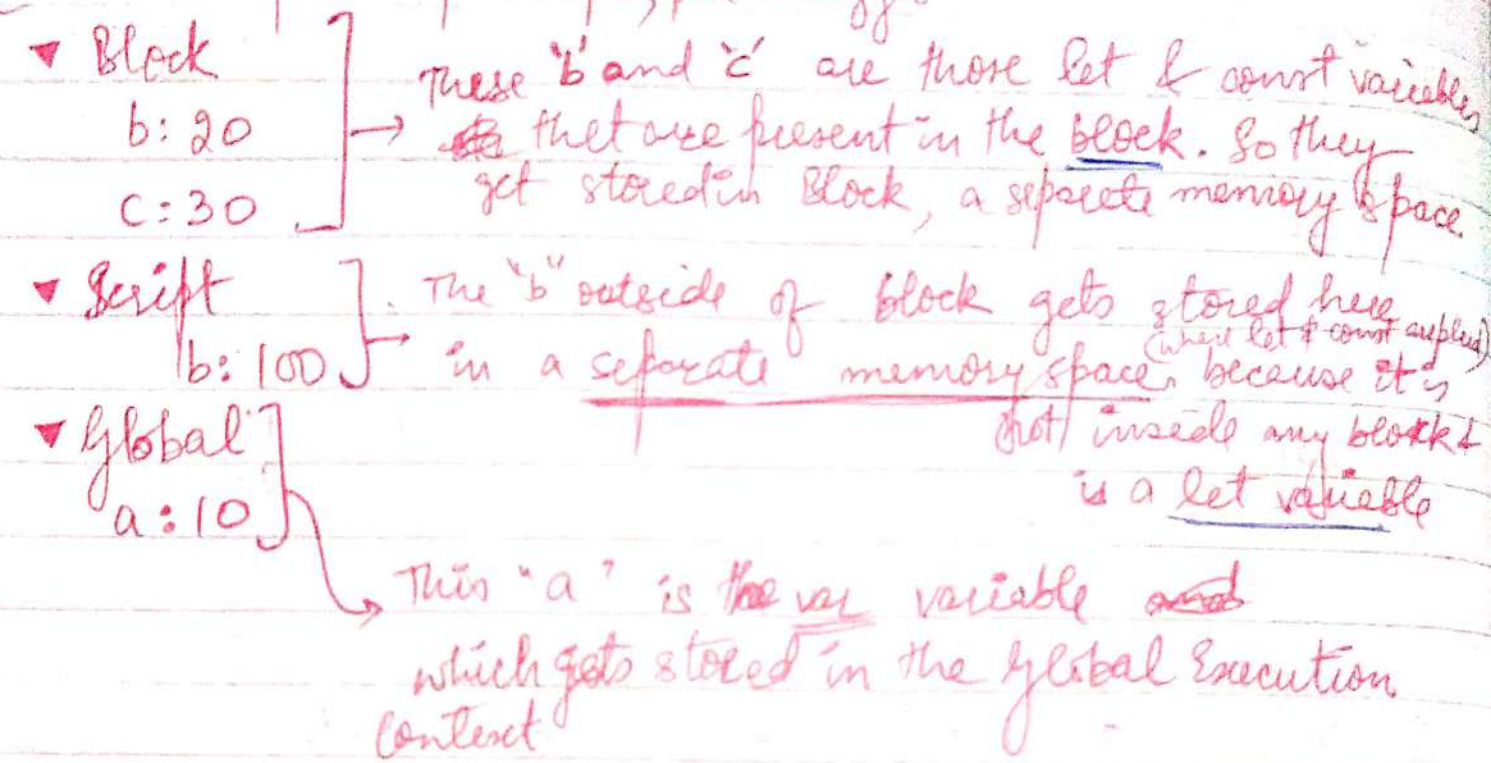
10 → Printed from lines 6, 7, 8 inside the block.
20
30
100 → Printed from line 10 outside the block

Here, when we print b from inside the block, that "let b = 20" shadows "let b = 100"
(Since we are referring to the 'b' which is a ~~let variable~~ & is block-scoped.

But when we print b from outside the block, "let b = 100" is what is the value that gets printed.

This is different from the case of Shadowing of var.

[Whenever inspect & open scope, put debugger



So we can see that here, we have 2 different 'b's, one is in Block scope, the other in Script (a separate memory space where let & const are stored.

* So when we print b from inside the block, the b that is in block scope is printed Hence: 20. Once we come out of the block scope, the block scope gets deleted and so does the 'b' inside it.

* When we print b from outside the block, the b from the Script is printed, hence 100

[In case of const] (same happens in case of const as happened in case of let)

```
const c = 100;
{
    var a = 10;
    let b = 20;
    const c = 30;
    console.log(a);
    console.log(b);
    console.log(c);
}
console.log(c);
```

O/P: 10
20
30
100

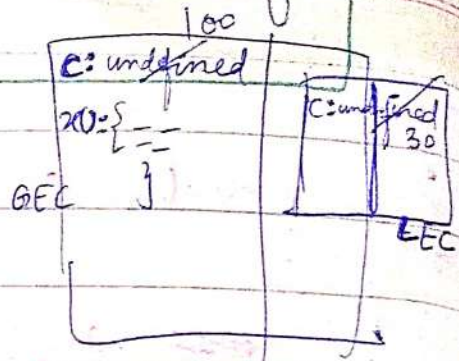
* In the memory, 2 diff c's are stored (one in Block, one in Script)
* When we print c from inside the block, the c that is in block scope gets printed. Hence: 30
* When " " , the c inside script gets printed
Hence: 100

You Don't Know JS

Global variables are automatically also properties of the global object (window in browsers, etc), so it is possible to refer a global variable indirectly as a property reference of the global object eg. window.a.

This technique gives access to a global variable that would otherwise be inaccessible due to it being shadowed. However, non-global shadowed variables cannot be accessed

Shadowing works the same way in case of functions:




```
var c = 100;
function x()
{
    var c = 30;
    console.log(c);
}
x(); //calling function x()
console.log(c);
```

O/P: 30
100

Here, also when we print c from inside the function, it prints the c = 30.

Because when we print c from outside the function, again c = 100 gets printed because var c is pointing to the memory location (c is stored in global execution context)

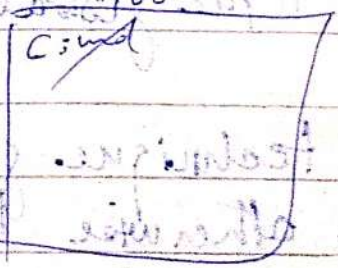
Now, from outside x(); if we print c, it gives 100.

Because the local execution context is deleted once we come out of the function x()

▼ Global
c: ~~100~~ 30 ← The value 30 is overwritten inplace of 100 inside the GEC

Because c is a var variable. Not let or const

```
var c = 100
if (true)
{
    var c = 30;
    console.log(c)
}
console.log(c)
```



ILLEGAL SHADOWING

If we try to shadow a let variable using a var variable inside the block, this is an illegal type of shadowing.

Case1: (Gives Error)

```
let a = 20;
{
  var a = 30;
}
```

o/p: SyntaxError: Identifier 'a' has already been declared.

This is an illegal shadowing because while shadowing the let variable, var variable is crossing its boundaries. The var variable is interfering with the scope of the let variable. In the same scope, let variable cannot be redeclared (so, it raises Syntax Error)

Case2 (Acceptable)

```
let a = 20;
function x()
{
  var a = 30;
  console.log(a);
}
x();
```

→ This does not cause any error. Because var is function-scoped. So var 'a' is stored in the local execution context & gets deleted as we come out of the function.

o/p: 30

Here, var is trying to shadow let variable, but in doing so, it is not crossing its boundaries. var a is not interfering with the scope of let a variable

Acceptable Shadowing

* If we try to shadow let with let → both stored in diff memory spaces → Here no interference in terms of scope.
* Shadowing const with const → Here
* Shadowing var with var → Here
* Shadowing var with let → No interference as let is in block var is in global
* Shadowing var with const → No interference, as const in block var is in global

Illegal Shadowing :

* Shadowing let with var
* Shadowing const with var

[Here var interferes with the scope of let & const, let & const can't be redeclared, so it gives syntax Error]

Lexical Scope : Block scope also follows lexical scope.

```
const a = 20;
{
  const a = 100;
  {
    const a = 200;
  }
}
```

[Here we can see how the various const variables are lexically present inside others.]

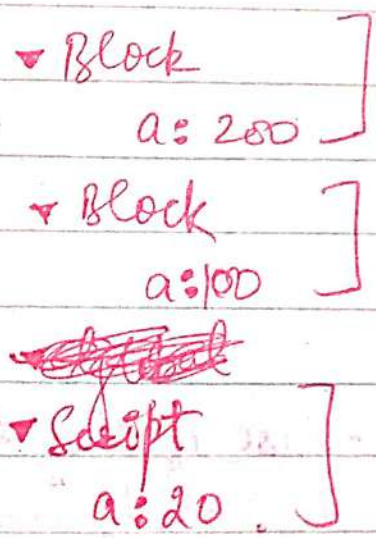
If we try to access a in the innermost block, it finds for the nearest a

Block

```
{
    const a = 20;
    {
        const a = 100;
        console.log(a); // here also nearest a's value is used
        {
            const a = 200;
            console.log(a); // innermost block me access karne ki koshish, so, nearest a ki value is used.
        }
    }
    console.log(a);
    // the nearest a ki value is 20
}
```

O/P: 100
200
20

Scope



We can see 3 different scopes so the 3 a's are stored in 3 different scopes.

If in the program, 'a' is not found in the block scope, then we go up the scope chain and find its value in Script.

So, also, the lexical Scope Chain pattern is followed in the case of Blocks.

Scope

Is there any diff. in scope rules in case of Normal Functions v/s Arrow Functions

* The scope working & rules remains same in case of both normal functions and arrow functions. We need not consider the two types of functions differently for understanding the scope rules.

Episode – 10

Closures in JS

Closure: Function bundled together with its lexical environment forms a Closure.

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

O/P: 7

We know why the O/P is 7. Because of the lexical scope. When x() is called, and inside it y() is called, we find a inside the local memory of function y(). But if it is not found, there, we go up the scope chain and search for a in the lexical parent scope (i.e. global scope) and we find it.

*** V-Imp

Closure: Function + Lexical Environment

What is going on in background! :)

Put debugger on line no. (6) and open the code. We find this under scope:

- ▼ Local
  - this: Window
- ▼ Closure (x)
  - a: 7
- ▼ Global
  - :
  - :

inside y(), it forms a closure of the function y() binded together with its lexical parent (and its variables: a)

The function y() is binded together with the variables of x()

- Functions are ♥ of JS

→ We can assign a function to a variable

Eg:

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

→ We can pass functions as arguments

Eg:

```
function x()
{
    var a = 7;

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

→ We can even return functions in JS:

```
function x()
{
    var a = 7;
    function y()
    {
        console.log(a);
    }
return y; // we are returning a function from
          a function.
}
x()(); let z = x();
       console.log(z);
```

O/P:

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

→ The entire function y() gets printed

Question

JS is synchronous. Once we reach the line
let z = x()
execution is deleted from call stack after this x()'s execution, it no longer remains in the memory. So what will happen to the variables inside x(.)?

We have returned y() outside x() and now y() no longer resides in x(). It now is assigned to z. So how will y() behave outside x()? ie how will y() behave outside of its scope?

When
~~When~~ ~~z = x()~~

```
① function x()
② {
③   var a = 7;
④   function y()
⑤   {
⑥     console.log(a);
⑦   }
⑧   return y;
⑨ }
⑩ var z = x();
⑪ console.log(z);
   //---thousands of lines of code
⑫ z(); //invoking z() actually executes the function y() that is stored in z
```

Now, here, at a later point in the code, z() is called. z() actually stores the function y() in it. When z() is executed, what will it print??

The O/P:

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

→ From line no. (11)

7 → from z();

It prints the value of **a = 7**.

How is this possible that even after x() is removed from the memory and all its variables along with it, how is y() still able to remember a & its value.

* This is possible because once a function is returned, it still remembers its lexical scope (where it was actually present).

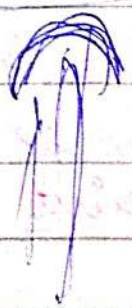
* The functions when get returned, remember where they were actually present.

* Therefore, even though x() no longer exists after line (10), still when z() is invoked, it gives output of a = 7, Because z() has function y() in it, and y() remembers its lexical scope. It remembers that there was a variable a and that y() as it has a strong binding with it.

* So, we can say that when we return a function, it's not just the function that is returned. It is the closure that is returned ie (function + lexical scope) is returned.

V. Imp

12:34



Closures come with a lot of gotchas & corner cases
(They are very frequently asked in interviews as O/P questions)

* Let's see some of those gotchas:

(1)
```
function x()
{
  var a=7;
  function y()
  { console.log(a);
  }
  a=100; //the value of a is changed
  return y;
}

var z=x();
//console.log(z);
z(); //this invokes z() and z() gets executed
```

O/P: 100

Here the question is that if the value of a is changed after the definition of y(), then what will be printed upon execution of z()?
Will it print 7 or 100?

It will print 100 because when y is returned from x(), it remembers its lexical scope.
In y()'s lexical scope, we have var a i.e.
y() remembers the reference to the variable a (not the value =7 of a). The value 7 does not persist, the reference to a persists.

So when z() gets executed & console.log(a) is encountered, then we find a in the local scope of z(), but we can't get it. Then we go to its lexical parent ie x() and find a there whose value is 100 now. So 100 is printed.

So this is why 100 is printed. [Because of closure]

② function z(){
var b = 900;
function x()
{ var a = 7;
function y()
{
console.log(a, b);
}
a = 100;
return y;
}
x()();
}
z();

If we are trying to access a function's parent scope & its parent's parent's scope, then what will be saved in the closure?

The function y() forms a closure with its parent as well as parent's parent's scope.
So y() remembers a reference to variable a as well as b.

Upon inspecting and putting a debugger on (console.log(a,b)) and opening Scope:

▼ Local
this: Window
▼ Closure(x) → we see that y forms a closure with x & z
a: 7
▼ Closure(z)
b: 900

Common uses of Closures:

↳ In Module Design Pattern
↳ currying (it is possible just because of Closures)
↳ In functions like once(), closures are used.

Usually, functions can be called any number of times, from anywhere.
But due to closures, it is possible for functions to remember something (that they have to run only once)

↳ Memoize() fun also uses closure
↳ Maintaining state in async world in JS
↳ setTimeout() → In this we use closure
↳ Iterators
and many more.