Execution context

# 

# **Hoisting**

1. In JS, before the code is executed, the variables get initialized to undefined.

2. Arrow functions enact as variables and get "undefined" during the memory creation phase while functions actually get run.

3. Hoisting: Mechanism in JS where the variable declarations are moved to the top of the scope before execution. Therefore it is possible to call a function before initializing it.

4. Whenever a JS program is run, a global execution block is created, which comprises of

2: Memory creation and Code execution.

Golden Rules:

1. Variable declarations are scanned and are made undefined

2. Function declarations are scanned and are made available

Undefined means variable has been declared but value is not assigned

Not defined means variable is not declared

findings

let and const variables and functions are also hoisted , hoisting just means that they are present in the memory before code execution , but the point here is that let and const are set to uninitialised during memory creation phase and var variables are initialised with undefined but var let const all are hoisted keep in mind

IMPORTANT

Variables defined with let and const are hoisted to the top of the block, but not initialized.

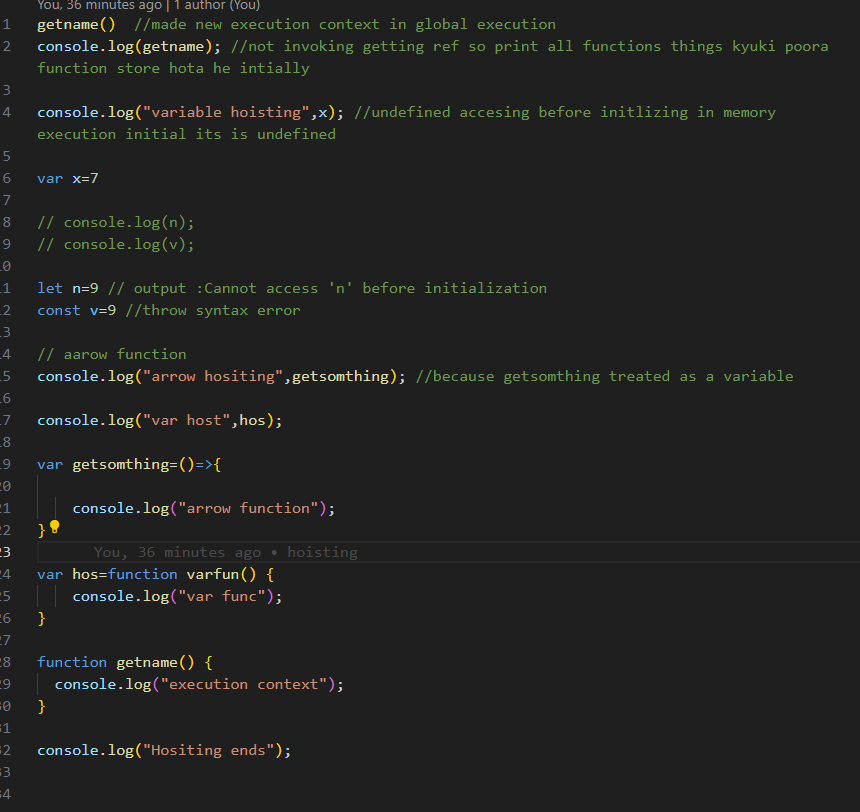
Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

* Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

* Using a const variable before it is declared, is a syntax error, so the code will simply not run.

code



let & const in JS 🔥Temporal Dead Zone

1. let and const are hoisted but its memory is allocated at other place than window which cannot be accessed before initialisation.

2. Temporal Dead Zone exists until variable is declared and assigned a value.

3. window.variable OR this.variable will not give value of variable defined using let or const.

4. We cannot redeclare the same variable with let/const(even with using var the second time).

5. const variable declaration and initialisation must be done on the same line.

6. There are three types of error: [1] referenceError {given where variable does not have memory allocation} [2] typeError {given when we change type that is not supposed to be changed} [3] syntaxError {when proper syntax(way of writing a statement) is not used}.

7. Use const wherever possible followed by let, Use var as little as possible(only if you have to). It helps avoid error.

8. Initialising variables at the top is good idea, helps shrinks TDZ to zero.

/////////////////////////////////////////////////////////////////////////////////////////////

**How functions work in JS ❤️ & Variable Environment**

**1. We learnt how functions work in JS.**

**2. At first a global execution context is created, which consists of Memory and code and has 2 phases: Memory allocation phase and code execution phase.**

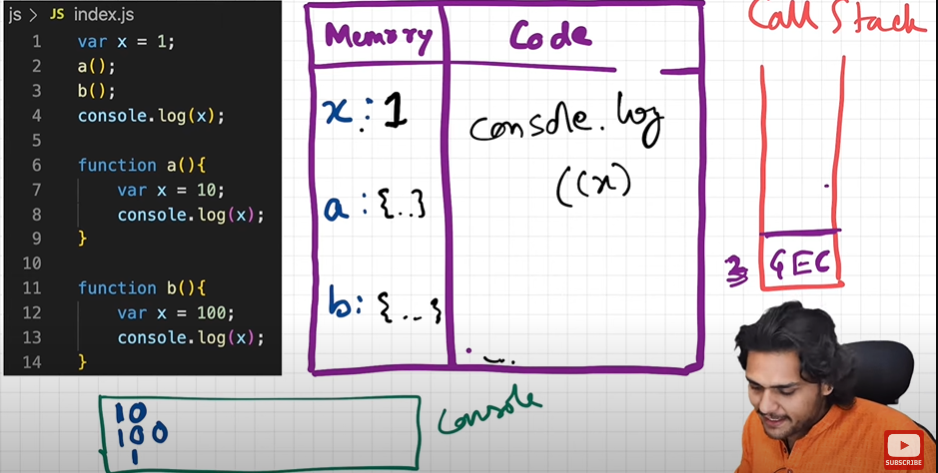
**3. In the first phase, the variables are assigned "undefined" while functions have their own code.**

**4. Whenever there is a function declaration in the code, a separate local execution context gets created having its own phases and is pushed into the call stack.**

**5. Once the function ends, the ExecutionContext is removed from the call stack.**

**6. When the program ends, even the global EC is pulled out of the call stack.**

**When log (x ) its find in local memory of particular execution context first then gloabal**

****

**------------------------------------------------------------------------**

**Shortest js program (empty js file)**

• window object is created by the JS engines of the respective browsers when global execution context is created.

• whenever an execution context is created a "this" variable is also created.

• at the global level "this" points to the global object( window object in case of browsers).

• anything that is not inside a function is the "global space".

• whenever we create any variables or functions in the "global space", they get attached to the global object( window object in case of browsers).

so to access the variables/function defined in the global space ,

we can use any of the below:

console.log(window.a);

console.log(a);

console.log(this.a) //at the global space level, where this points to the window object

------------------------------------------------------------------------------------------------------------------

undefined and not defined

1.Undefined is a Special Placeholder which is used to reserve memory for the variables in the memory creation phase. Even before a single line of code is executed JS engine assigns undefined to the variables.

2.Not Defined means if we try to console or access any variable which is not declared in the code then we get Not Defined error.

3. JS is a loosely typed language or weakly typed language means it does not attaches its variables to specific data types like in C++ and java.

4.Remember undefined !== not defined.

The Scope Chain, 🔥Scope & Lexical Environment

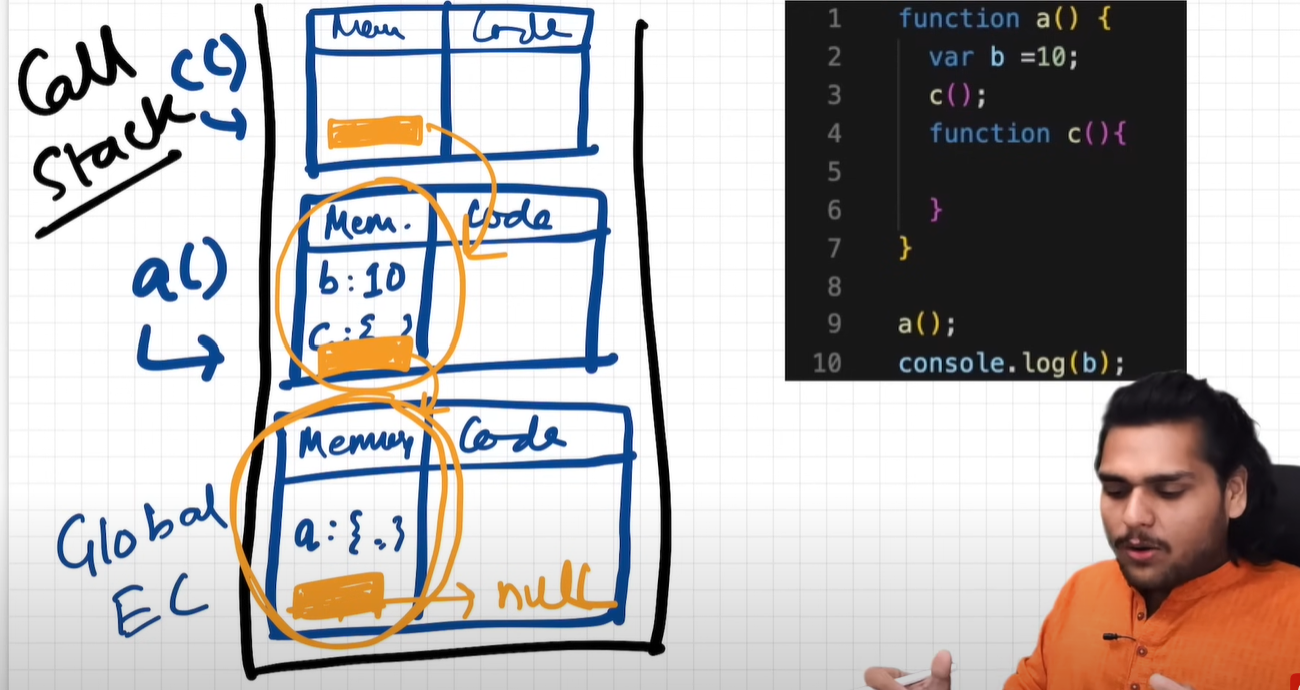
1. Scope of a variable is directly dependent on the lexical environment.

2. Whenever an execution context is created, a lexical environment is created. Lexical environment is the local memory along with the lexical environment of its parent. Lexical as a term means in hierarchy or in sequence.

3. Having the reference of parent's lexical environment means, the child or the local function can access all the variables and functions defined in the memory space of its lexical parent.

4. The JS engine first searches for a variable in the current local memory space, if its not found here it searches for the variable in the lexical environment of its parent, and if its still not found, then it searches that variable in the subsequent lexical environments, and the sequence goes on until the variable is found in some lexical environment or the lexical environment becomes NULL.

5. The mechanism of searching variables in the subsequent lexical environments is known as Scope Chain. If a variable is not found anywhere, then we say that the variable is not present in the scope chain.



//////////////////////////////////////////////////

Q) What is block in JavaScript?

> multiple js statements formed in a group enclosed in brackets and it forms a block

Q) What is need of a block/Grouping?

Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if, else, loop, function etc.

> JavaScript sometimes expect to run a single statement to run, but we need to run commands with multiple statements which is only possible by block

eg. on 4:14

write a simple function:

// even empty script is perfectly valid js script, what about empty brackets!!

{

var a = 10;

let b = 20;

const c =30;

}

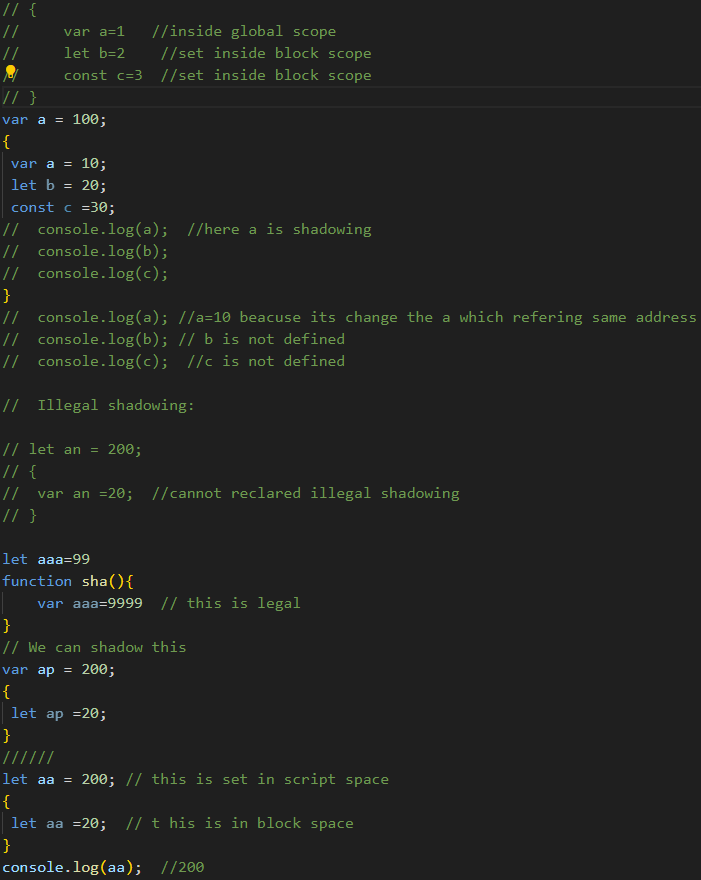
When a js script get hoisted (a Global Execution Context) gets created 'var' listed towards 'Global environment' and other variables 'let' and 'const' declarations go to the 'Block environment'

This become especially important when deciding the scope of a particular variable, since b and c are located in 'Block environment' and for a as we know exists in 'Global environment' any statement out of the "Block" can access 'a' ie. ' Variable in Global environment' and other are not!

so when we understand the extent of Global and local environment variables and their 'Scopes' == Environment that forms the lexical hierarchy of 'Scopes' and 'Scopes' have Levels like 'Scope inside scope'

Shadowing :- Providing same name to the variable as of those variable which are present in outer scope.

So in block " var a = 10;" influences the value within the block hence console.log(a); >> 10 and outside of the block 'Variable in Global environment' influences value of a hence console.log(a); >> 100



as 'var' declaration goes to 'Global environment' and sets in Memory context, it cannot be set using 'Block environment' value Hence: Uncaught SyntaxError: Identifier 'a' has already been declared

-------------------------------------------------------------------------------------------------------

**Closures**

**Function bundled with its lexical environment is known as a closure.**

**Whenever function is returned, even if its vanished in execution context but still it remembers the reference it was pointing to. Its not just that function alone it returns but the entire closure and that's where it becomes interesting !**

**A closure gives access to all the variables of it's parent function even after the that parent function has returned or executed. The function keeps a refernce to it's outer scope which preserves the scope chain throughout the time**

**­­**

**Uses of closures**

**-module design pattern**

**-currying**

**- functions like once**

**-maintaining state in async world**

**-memoize**

**-settimeouts**

**-iterators and may more**