Scoping Mechanism In JS

Everything in Javascript happen inside the execution context, Yo can assume the execution context is like a bigbox where whole code is executed, but how the execution context is being created it is created when we run a javascript program all the things happen behind the scenes.

The Execution Context is divided into two parts

1. **Memory Phase or Memory Component** (Variable Component) : so this is the place where all the variable and functions are stored in the form of key and value pairs

memory phase	code execution phase
key : value	code exection done
a: 10	here, one line at a
function : {.....}	time

a : key
10: value

2. **Code Execution Phase** : In this phase the code will get execute but it will execute line by line, this is otherwise known as code of execution

> **Note**
>
> Javascript is a Synchronous Single Threaded Programming Language that mean's everything will happen line by line. If the 1st line execution is done then only it will go to the second line

How a Javascript Code is executed

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

6 var square = square(n)
7 var square4 = square(4)
```

So as we know that everything in the JavaScript happen inside the execution context and it is divided into two parts one is memory component and the other one is code execution phase so what will happen in the 1st phase.

Phase One

1st phase JavaScript will allocate memory to all the variable and function , so when it will hit the first line it will allocate memory to n by reserving a memory space for n and it store a special value which is known as undefined.

and then it move to the second line where it will see that it is a function and it store the whole function into it.

so for variable it will store undefined and for function the whole code will get into it.
and then it will reserve some memory for square2 and square4

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

Phase Two

Now comes the code execution phase, so till now the value of n is undefined but now it will get replaced with 2.

Now it will go to line number 2 and its a function and a function will execute only when it is called so it has nothing to do in line number 2.

now it will move to line number 6 because 2-5 there is nothing to do as if now now comes to the line number 6 now here the function is being getting called (function invocation) function invocation mean function name i.e square along with the round brackets that means function execution starts.

so functions are like heart in JavaScript its just like a mini program and when a mini program is invoked again an execution context is created a function execution context is created which is again divided into two parts memory part and code execution part

Code Execution Part (Phase One)

Memory Phase	**Code Execution Phase (phase one)**
n : 2	memory part | code execution part
square:	num : undefined |
	ans : undefined |

Code Execution Phase (Phase Two)

Memory Phase	**Code Execution Phase (phase two)**
n:2	memory part | code execution part
square:	num: 2 |
	ans : 4 |

Here in phase two of code execution part when the function will invoked the value of n which is 2 is passed to `num` , remember earlier in `num` a special place holder undefined is stored but now it will be replaced with 2 and these things will happen in phase two during code execution part

and then it will come to line number 3 `var ans = num * num` which is 4 now `num` will also be replaced with 4.

and in line number 4 it will return `ans` and return `ans` mean it will return to that place where it is invoked and the function is invoked in line number 6 so now the value of square 2 will be 4 earlier it was undefined but now it will be replaced with 4.

whenever you see a keyword return that mean now its over you just return the control back to the execution context where the function was invoked

Scope Resolution

Scope resolution mean allocating some memory to variables or functions.
But before that we need to have some idea about how many types of scopes are there because every variable will have a different scope that depends upon how the program is being written.

So there are three types of scopes are available in JS.
- Global Scope.
- Function Scope
- Block Scope

Global Scope

Global scope refer to the scope where the variable is gonna be accessible from anywhere in the program , now when I say anywhere that mean if I want to access it inside the function then I can access, if I want to access it inside the loop then also i can access, if i want to access inside the block scope then also i can access.

Example of global scope:

```
let name = "deepak" // Global Scope
function getName(){
        console.log(name)
```

```
}
getName()
// defined name outside the function but accessing it inside the function
```

Function Scope

Function scope refer to the scope where it is accessible only inside the function. if you are creating a variable inside the function we cannot access it everywhere like Global Scope we can access it anywhere inside the function not outside.

```
let name = "deepak" // Global Scope
function getName(){
        let userName = "deepak_kumar"
        console.log(name)
}
console.log(userName) // this is not allowed
getName()
```

Block Scope

Whenever we define a pair of curly braces may be inside if, while-loop. if-else it create something called block. Block is a collection of valid JS Instruction enclosed in a pair of curly braces.

```
if(10>20){
        // this called block of if
}else{
        // this is called block of else.
}
```

Any variable defined inside a block has a block scope.
if we are defining something inside a block we can only access that variable inside the block not outside the block.

So is the scope of the variable defined only by where it is declared.?

No, scope of the variable is decided by how it is declared. There are three ways of declaration of variable which tell us about the scope as well .
. var: it help us to declare function scope and global scope variables
. let: it help us to declare block scope variables.
. const: const also help us to declare block scope variables.

These three variables help us to declare variables and also help us to define the scope of the variables.

var

`var` can only make global and function scoped variables but now a days `var` is not used often so its better to use let or `const,` But with let and `const` we cannot make global scope variable because let and `const` are usually used in declaring block scope variable.

so how is block scope and function scope are different.
if we are defining any variable in the function scope then we can use that variable or access that variable anywhere inside the function not outside the function.

```
var teacher = "Sanket Singh"
function fun(){
        var teacher = "Sarthak Singh"
        console.log('hello',teacher)
}
function gun(){
var student = "Deepak"
console.log('hello',student)
```

```
}
fun()
gun()
```

Notes:

. Scope mean where i can access a specefic variable, function. Scope is directly proptional to the Lexical Environment.
. Whenever a execution context is created a lexical environment is also created.
. Lexical environment is the local memory along with the lexical environment of the parent.

Lexical Scoping

JS Does scope resolution using lexical scoping mechanism. In Lexical Scoping we allocate scope to the variables during compile time. So in JS the allocation of value to the variables are done in Phase 2 and in Phase 1 the scope of the variables is decided. Scope of the variable decided mean where actually the variable is present.

> [Important Notes]
> Lexical Scoping is a fundamental concept in programming that determines the accessibility of the variables and functions based on where it is declared in the source code. In simple terms lexical scope is the scope of a variable or function determined during compile time by its physical location in the code. Lexical scope is static and remain the same through-out the program execution.
>
> Lexical scope determine the accessibility of variables and functions depending on their location in the source code