Class 9 - Code execution in JS

Functions

1. The function is an abstract body and inside that body particular logic is written and that function expects some values which are known as parameters of the function.
2. At the time of invoking the function we need to pass those values as an argument.
3. code

```
// function accepting parameters
function ServeBeverage(drink, quantity){
    console.log('I want '+ quantity + " " + drink)
}
// calling function by passing arguments
serveBeverage('coffee',4) // print I want 4 coffee
```

4. Ways of defining functions in JS
    a. In JavaScript, we have multiple ways of defining functions.
    b. Traditional way of writing function
    c. We can define functions in a way similar to that used in another programming language for function definition.
    d. code

```
function sayHi(){
    console.log('mrinal says hi')
}
// calling function
sayHi()
```

    e. Function as Expressions(First class citizens)

i. We can write functions inside the variable in JavaScript.
ii. Firsr clss citizen
    1. In programming, the term first-class citizens is used to describe entities that can be passed around and manipulated in various ways within a language. In JavaScript, functions are considered first-class citizens because they can be treated like any other value. This means that functions in JavaScript can be:
    2. Assigned to a variable
    3. Passed as param
    4. Returned from a another function etc

```
// Function as Expressions
let sayHi=function(){
   console.log('Mr. X says hi')
}
// calling function
sayHi()
```

Execution context in javascript

When the JavaScript engine scans a script file, it makes an environment called the Execution Context that handles the entire execution of the code.

During the context runtime, the parser parses the source code and allocates memory for the variables and functions. The source code is then executed.

There are two types of execution contexts: global and function. The global execution context is created when a JavaScript script first starts to run, and it represents the global scope in JavaScript. A function execution context is created whenever a function is called, representing the function's local scope.

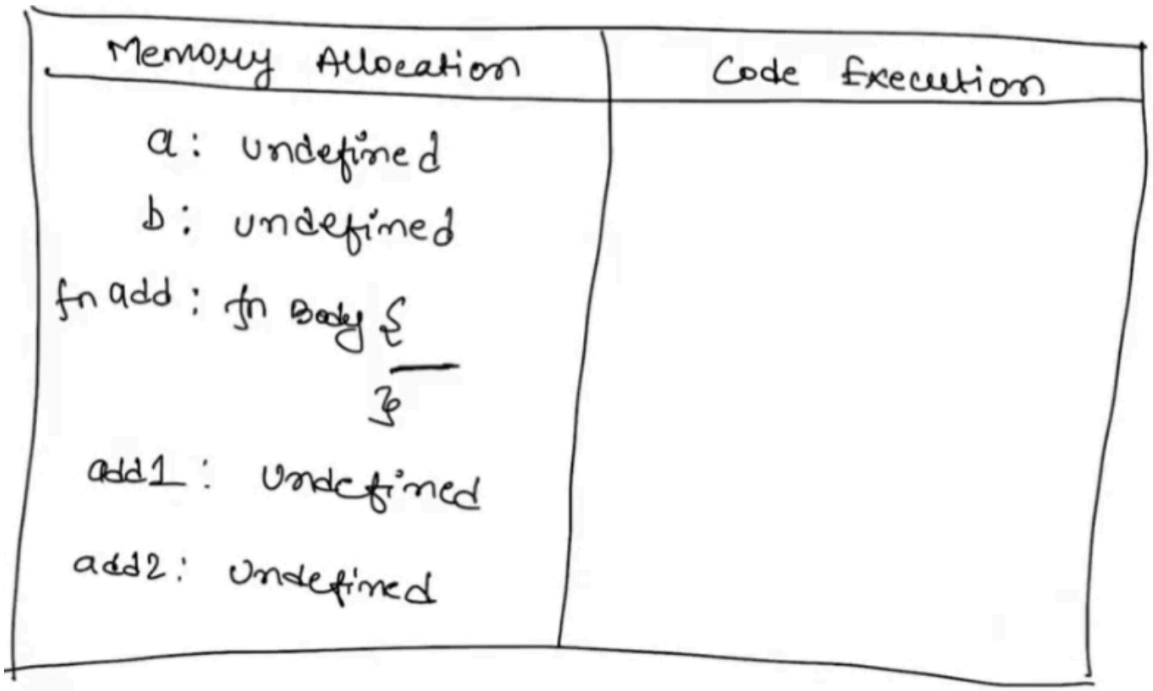
We have this code to Execute-

```
var a = 2
var b = 3
function add(num1, num2){
  var ans = num1+num2
  return ans;
}
let add1 = add(a, b)
let add2 = add(5, 6)
console.log(add1) // prints 5
console.log(add2) // prints 11
```

Javascript code executes in two phases:

Memory Allocation:

1. Every particular variable is assigned with undefined value initially.
2. Initially, no value is assigned to the variable, only their existence in a way is acknowledged. And every function will get its whole body in this phase.

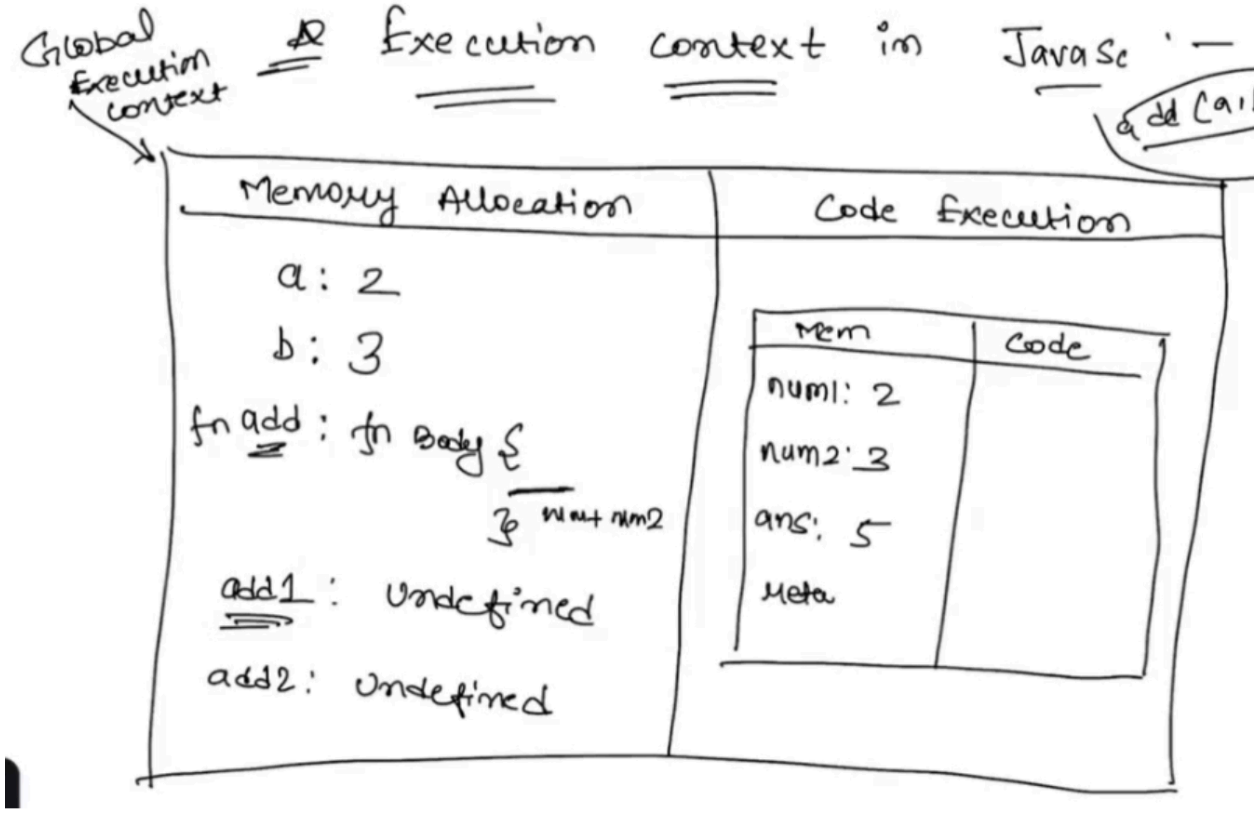


Code Execution

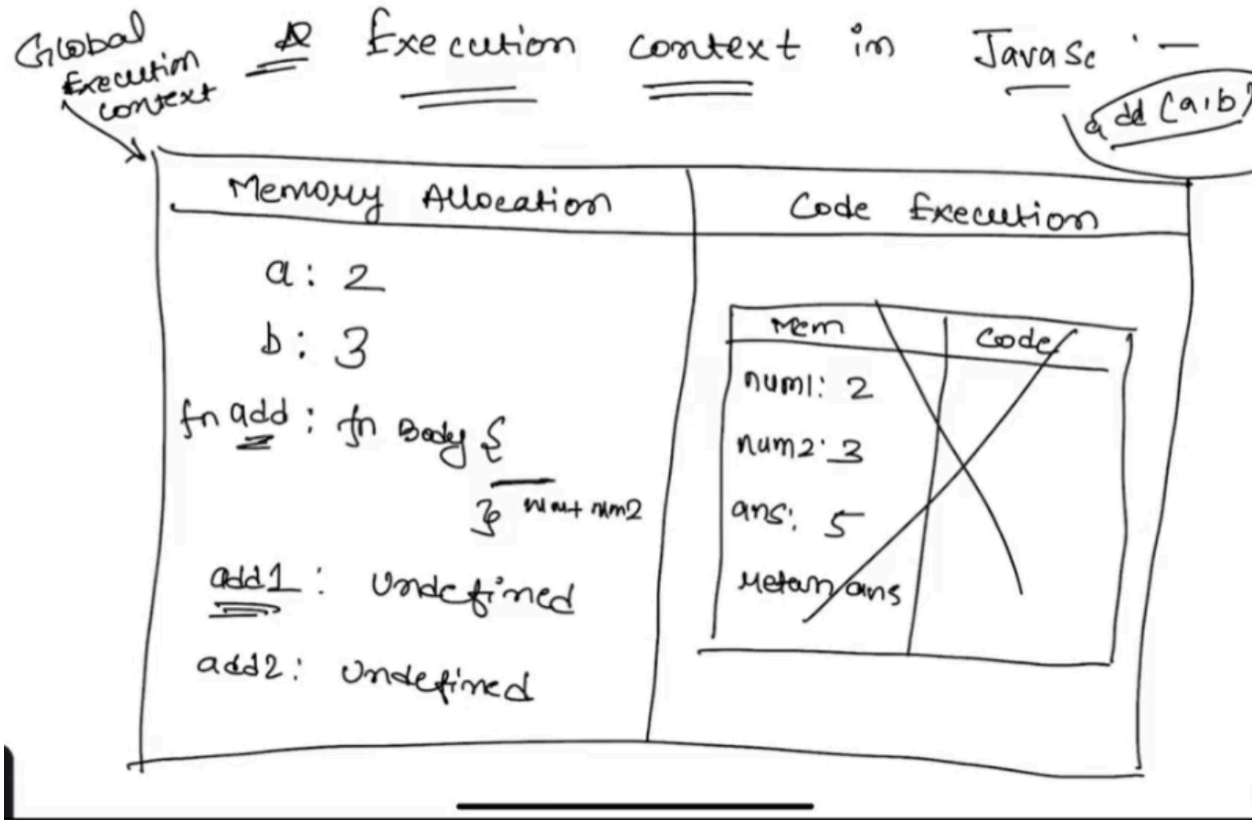
1. Code will get executed in this phase.
2. Now the values of variables are assigned.
3. When a function is called and it starts executing then another execution context is created, memory allocation and its code block is created .
4. For every function calling, basically another execution context will be created for the function.

5. And execution context of the whole program is known as the global execution context, and for a function it is called function's Execution Context
6. Variables declared inside the function are written inside the execution context of that particular function.

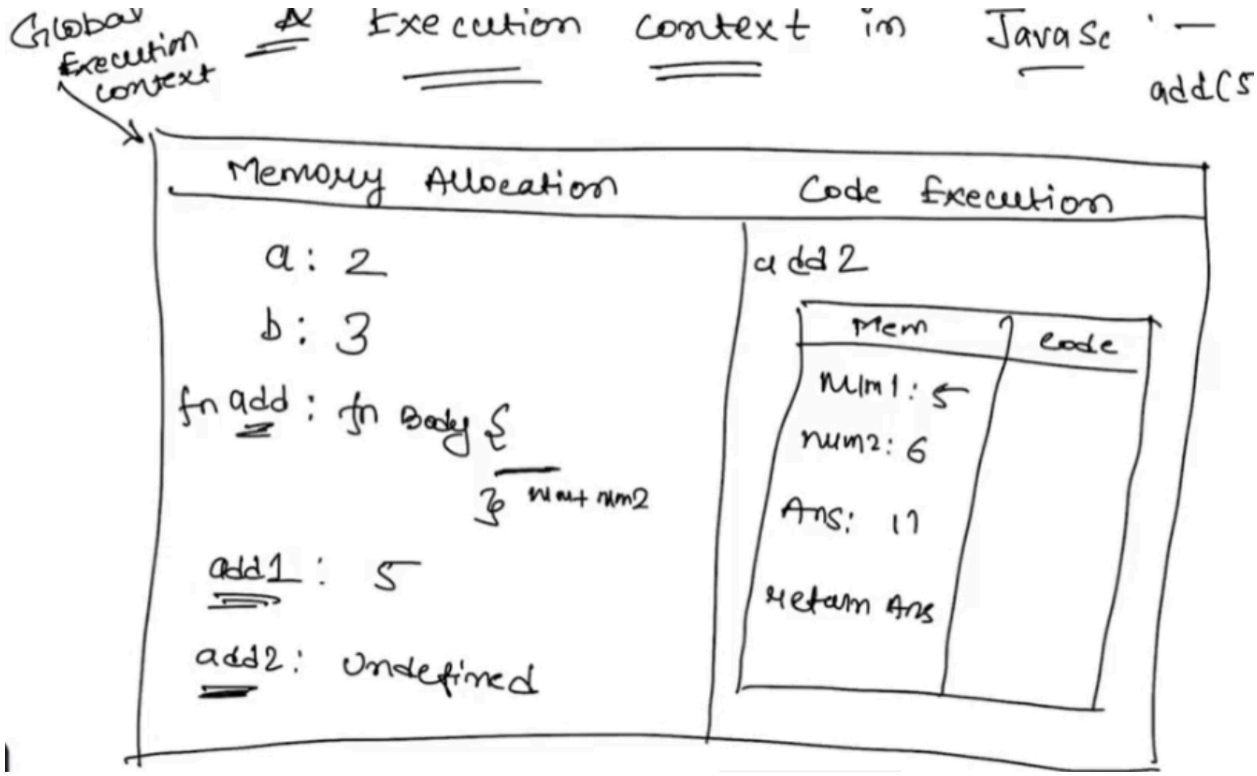
   So in our case, Execution context is created when the function is called and the answer is stored in the add1 variable.



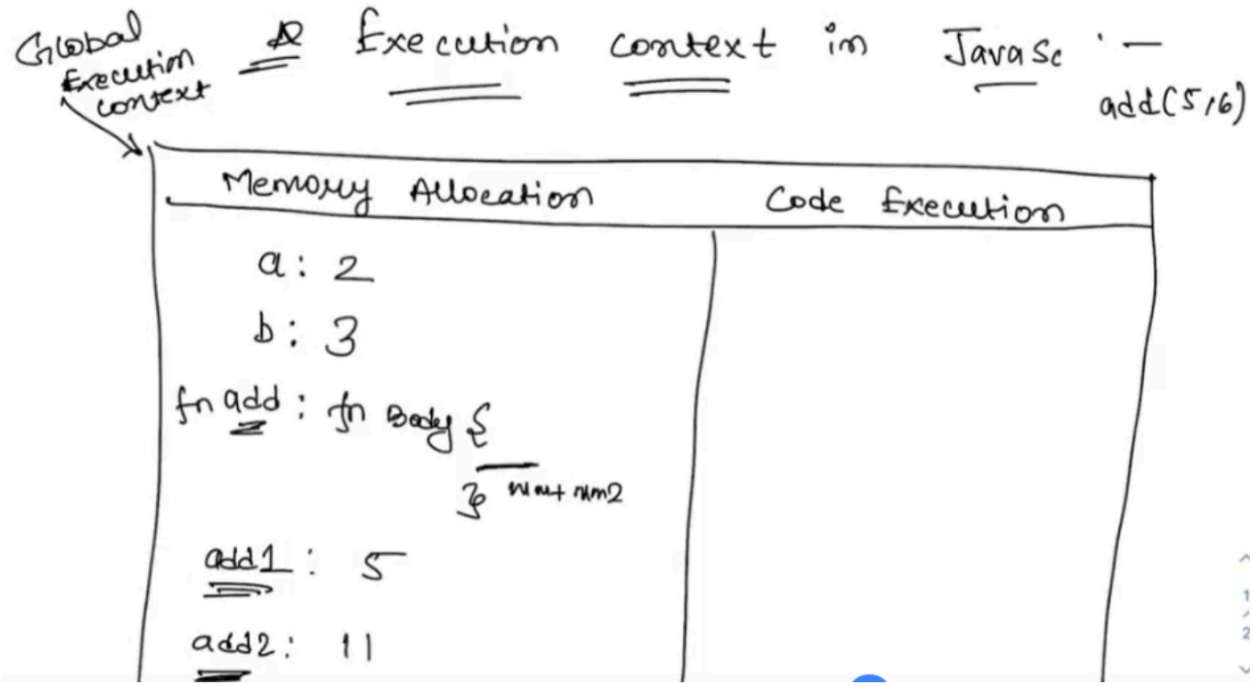
* After completing the function execution, a particular function execution context returns the value and the job of it is done



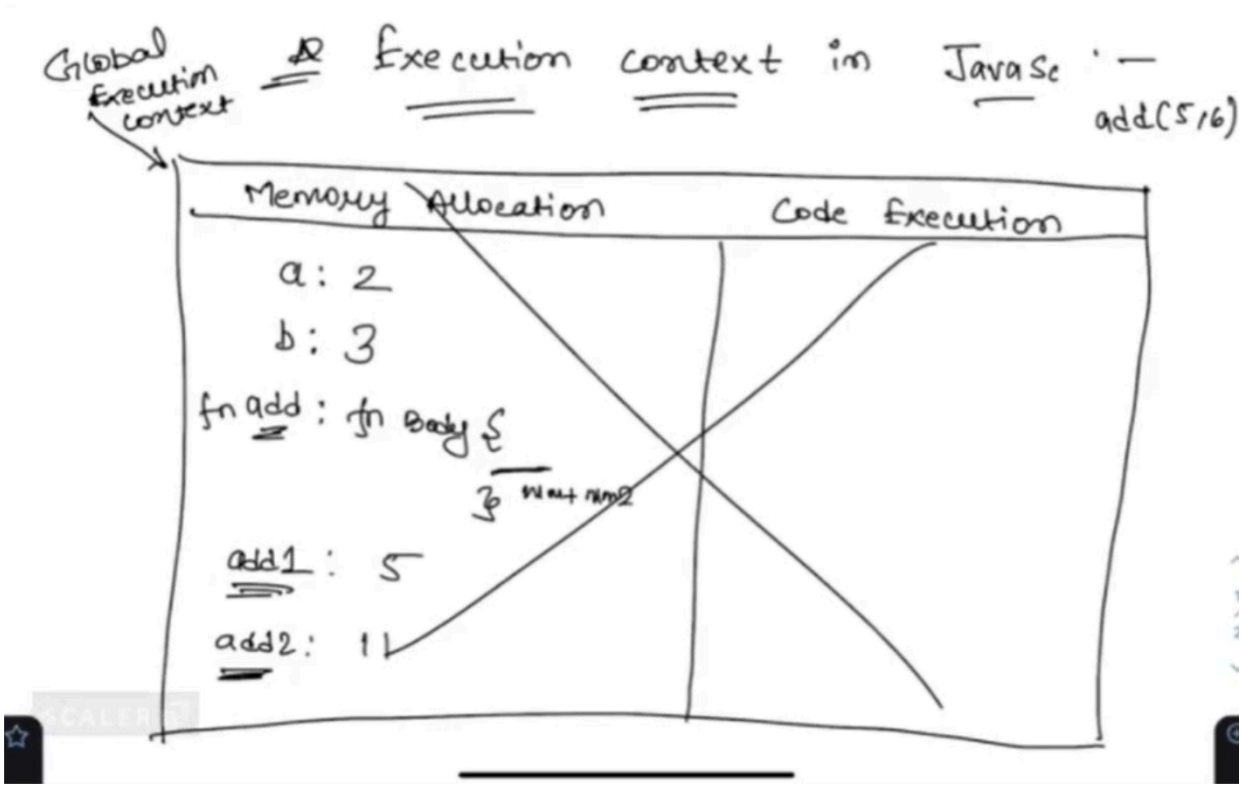
Now again execution context is created when the add() function is called again and the return value of the function is stored in add2.



After the completion of function execution it again returns the value that was calculated



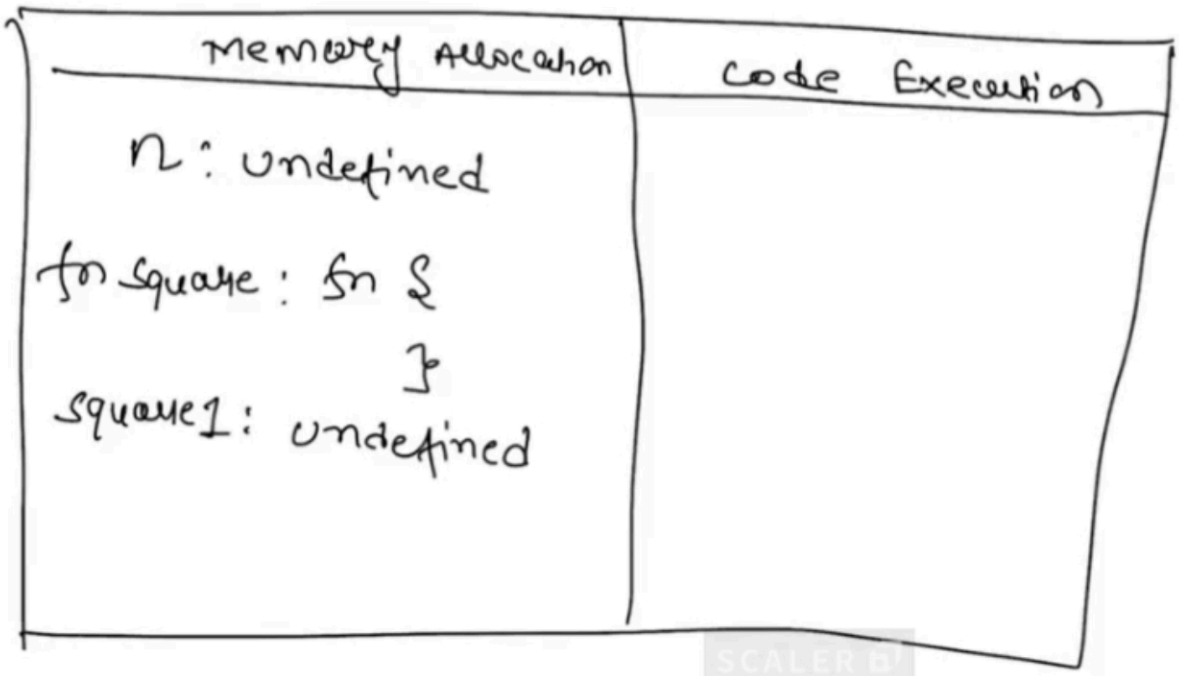
Now the execution of the program is completed and the whole execution context is now over and the final output is returned



Q. Lets try one more

```
var n = 3
function square(num){
   var ans = num*num
   return ans
}
let square1 = square(n)
```

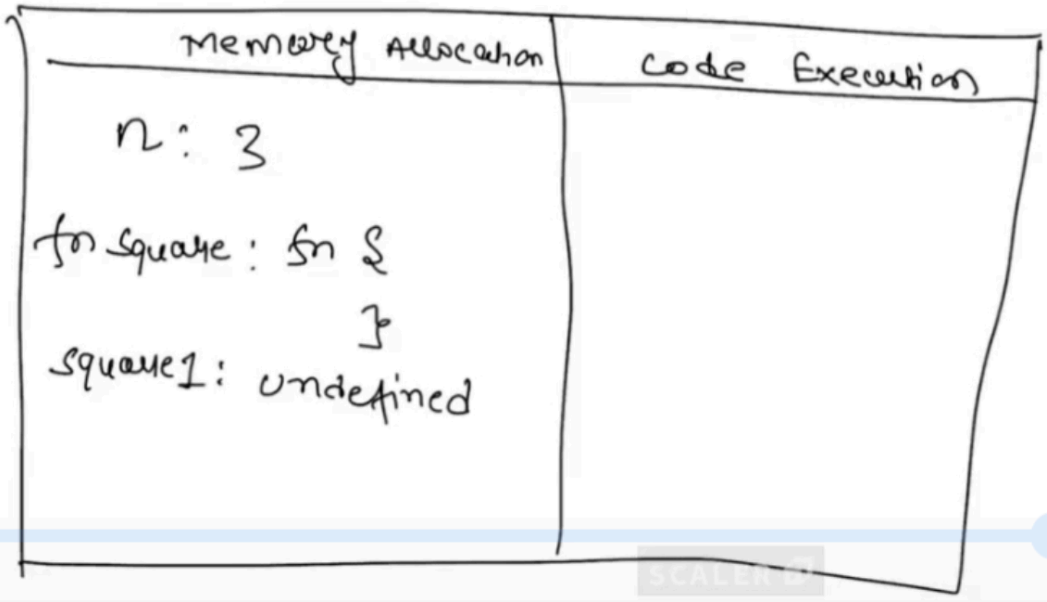
1. Firstly global execution context will be created.



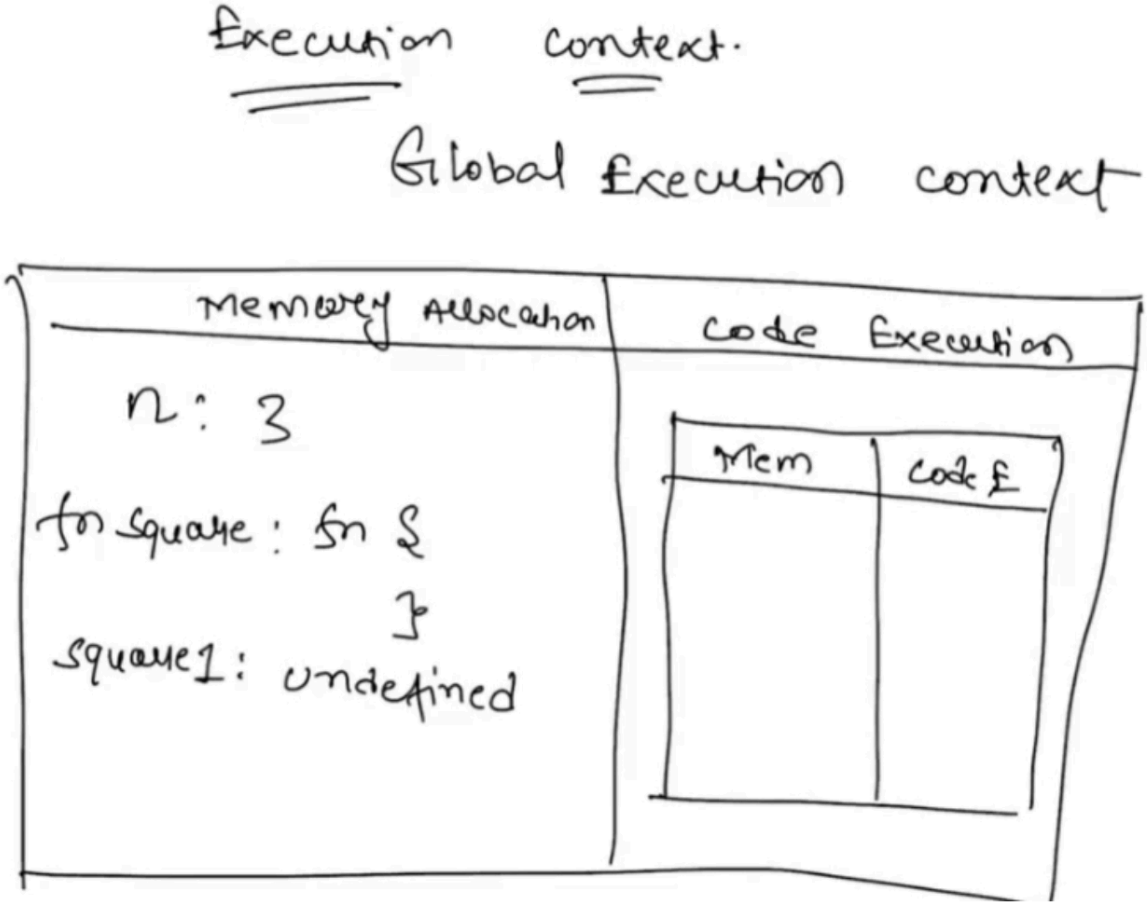
2.
3. Then the value to the n will be assigned.

Execution context.

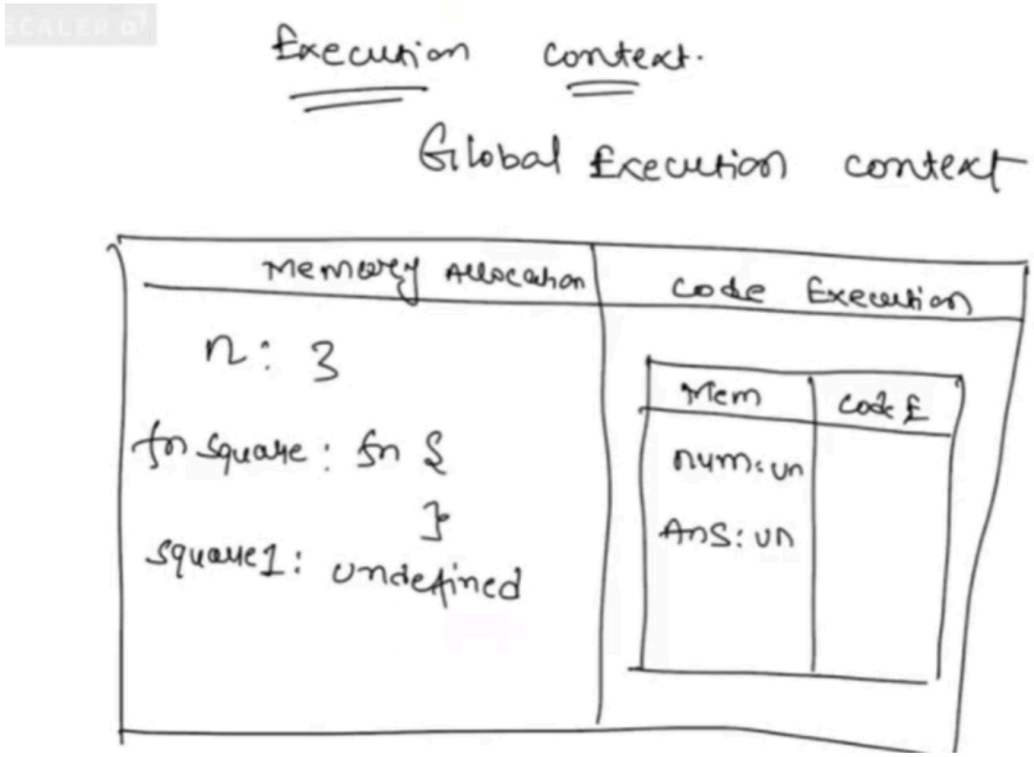
Global Execution context



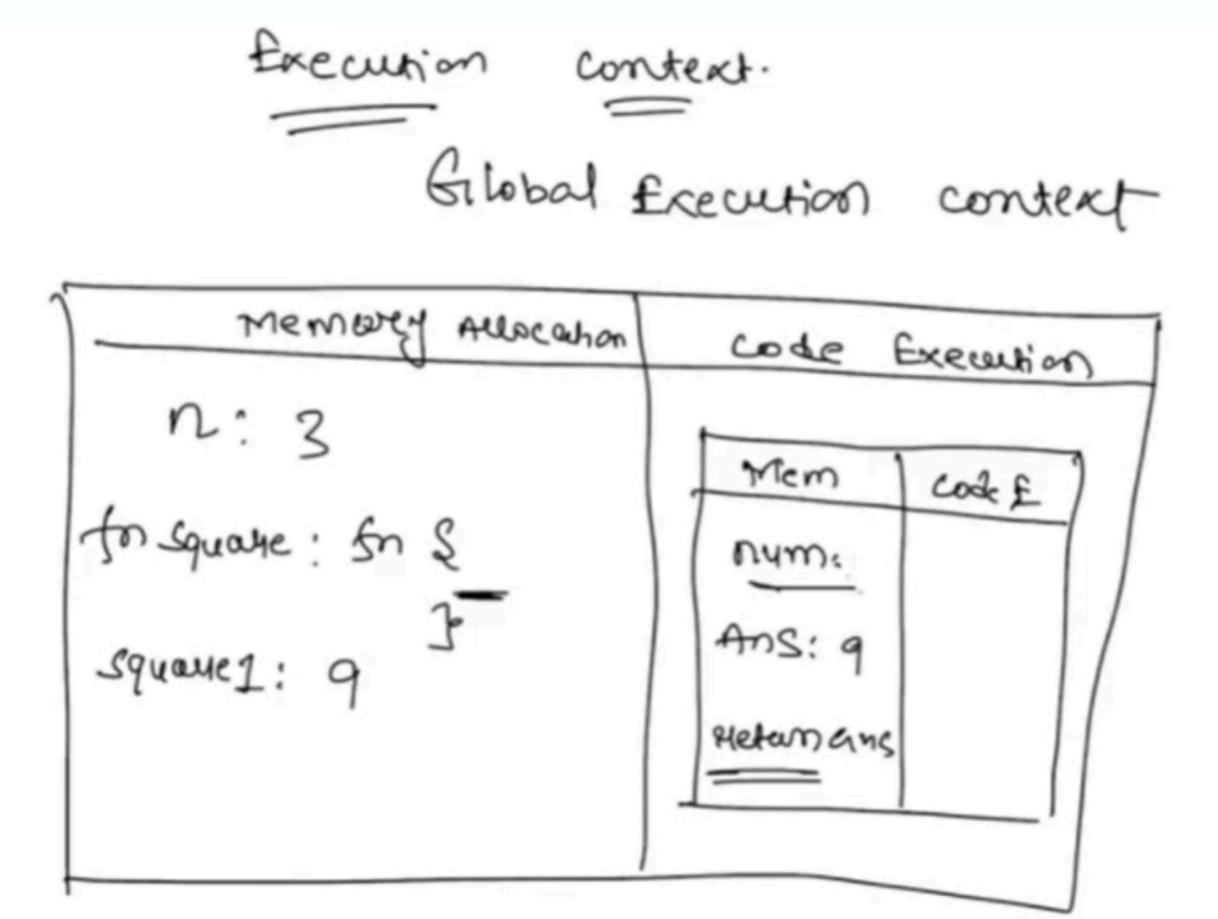
4.
5. Now execution context is created for the square method as it is called.



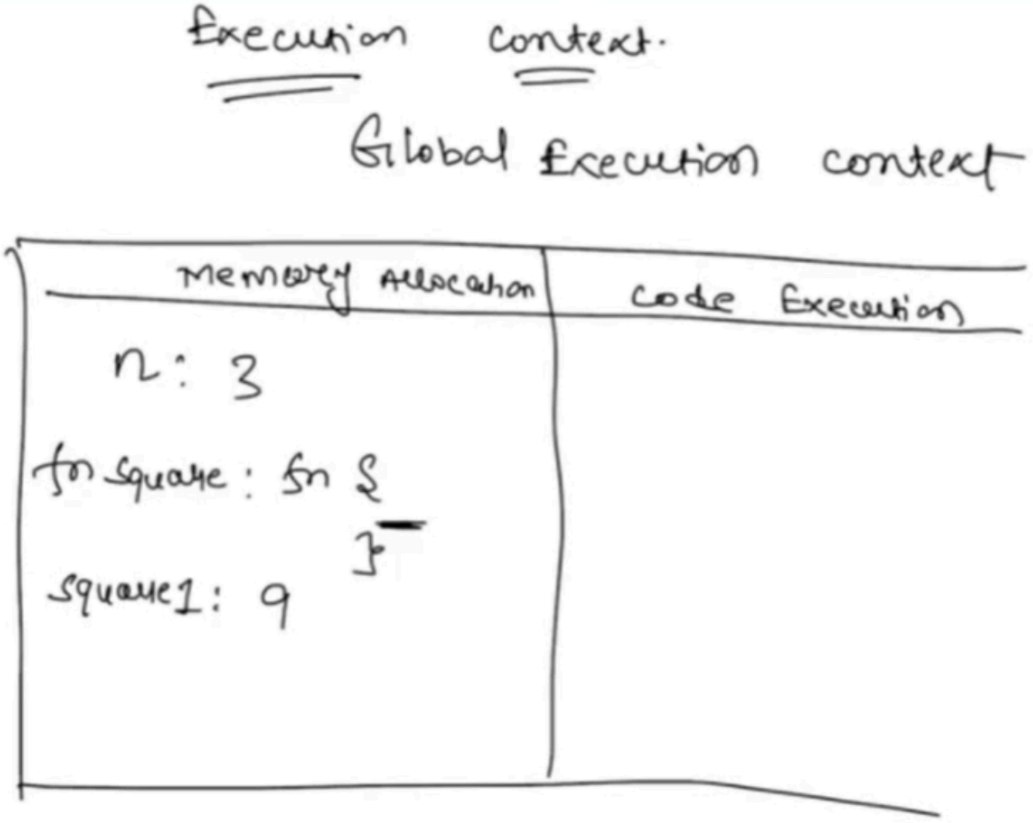
6.
7. Variables declared inside the function are initialized by undefined values in the function execution context.



8.
9. Now the value will be assigned to the variables of the function execution context.

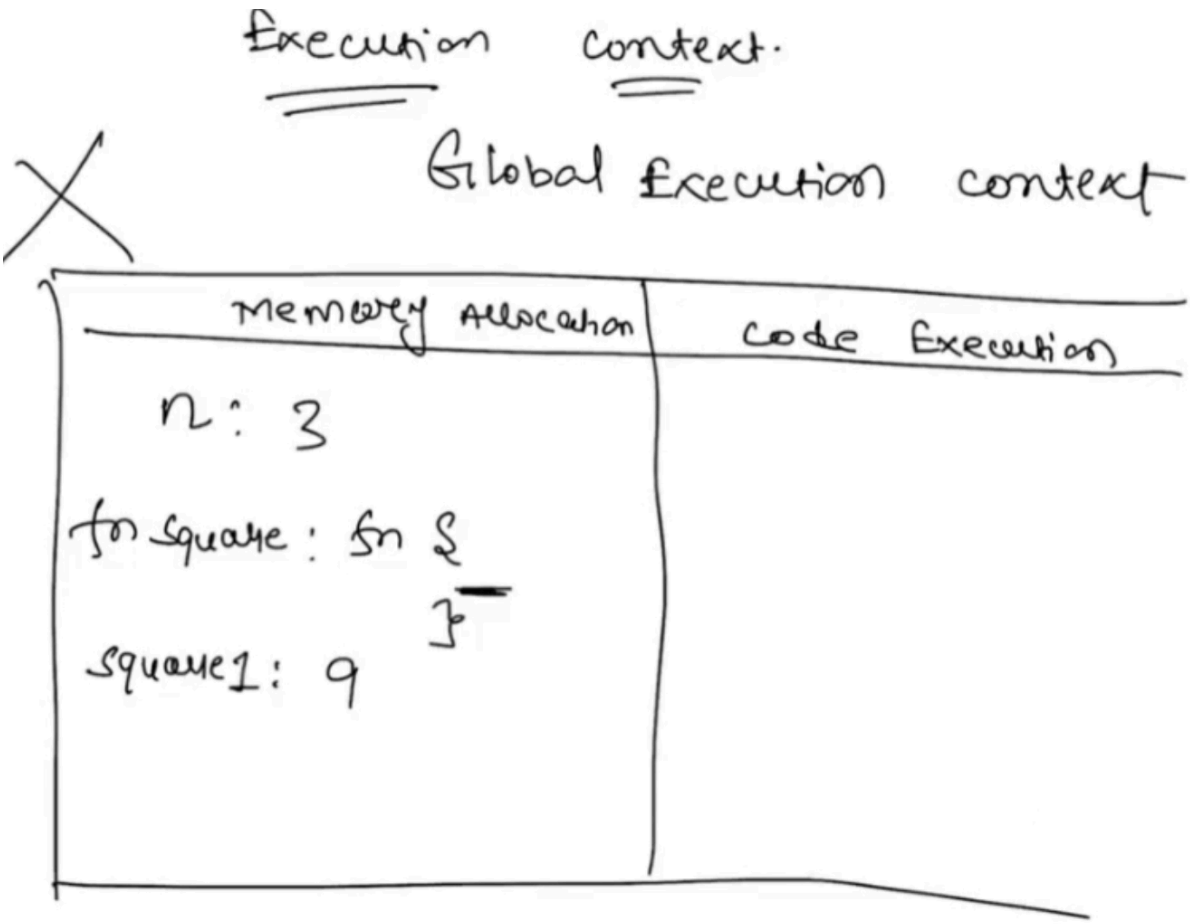


10.
11. After that function execution context returns value to the global execution context and the function execution context is not required further.



12.

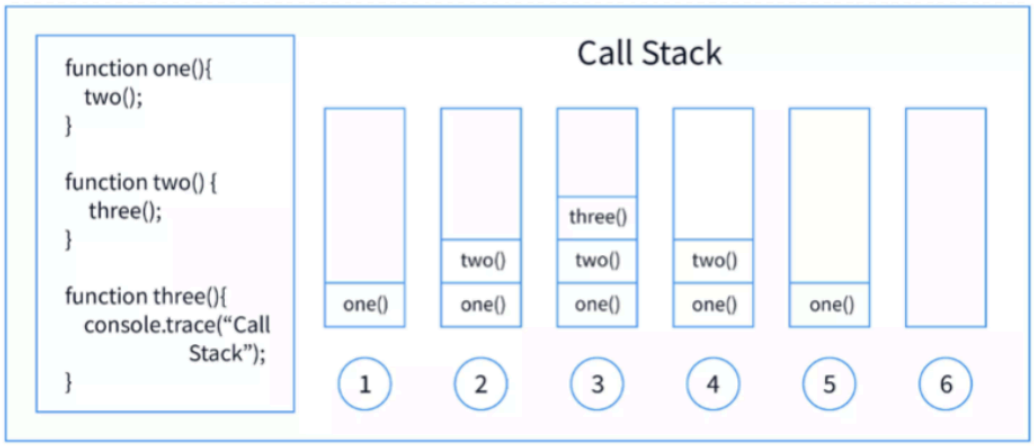
13. Now the GEC completes and is returned



14.

CALL stack

1. JS handles different execution context beautifully using call stack
2. When the program starts , there is a global execution stack that is put on the stack, followed by other function calls



3.
4. When the code completion happens, the GEC is also popped out and the call stack is empty
5. It is your call stack that maintains the order of execution

Hoisting

1. What do you think will be the output

```
var msg = 'hello'

function greet(){
   console.log(msg, " Happy New Year")
}

console.log(msg)
greet()
```

2. Now lets see the interesting part. What will be the output

```
console.log(msg) // undefined
greet()
```

```
var msg = 'hello'

function greet(){
   console.log(msg, " Happy New Year") // undefined, Happy new year
}
```

3. Basically we are trying to access the variable and function before they are defined or assigned a value
4. What if we comment the variable line and check

```
console.log(msg)
greet()

// var msg = 'hello'

function greet(){
   console.log(msg, " Happy New Year")
}
```

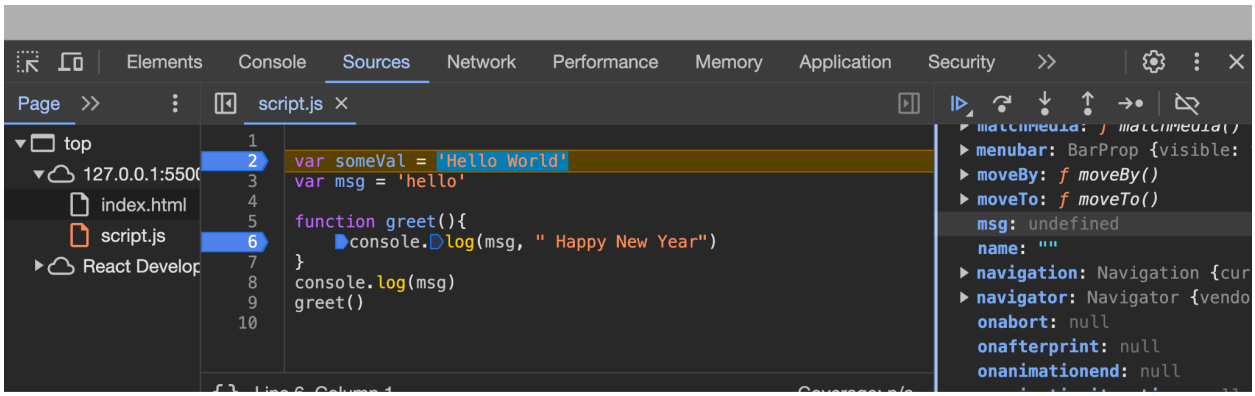
5. Now there is a reference error. If you notice, earlier it was undefined , now it not defined
6. What if we console the function itself before it is defined

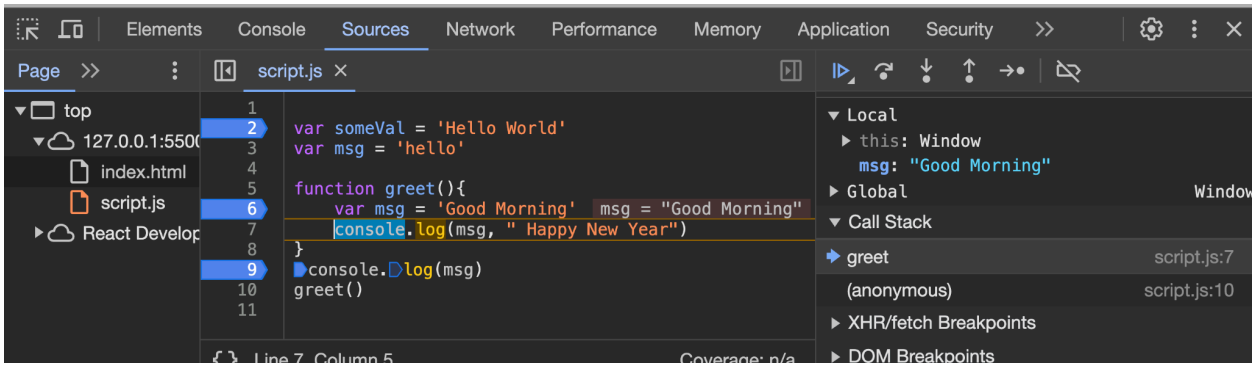
```
console.log(msg)

console.log(greet)

var msg = 'hello'

function greet(){
   console.log(msg, " Happy New Year")
}
```

7. This feature or behavior in which we are able to access variables and functions before they are defined is called as HOISTING in JS
8. But we know how this is happening at a very intuitive level. In the memory creation phase, variables and functions are already assigned with memory
9. Variables get undefined as placeholder value and functions get their function body
10. So in the code execution phase, even if we use these before they are defined, we see this behavior
11. Add a debugger before the code execution started
12. 

13. Check for variables and functions and we will see that they already have some value before the code execution started
14. 

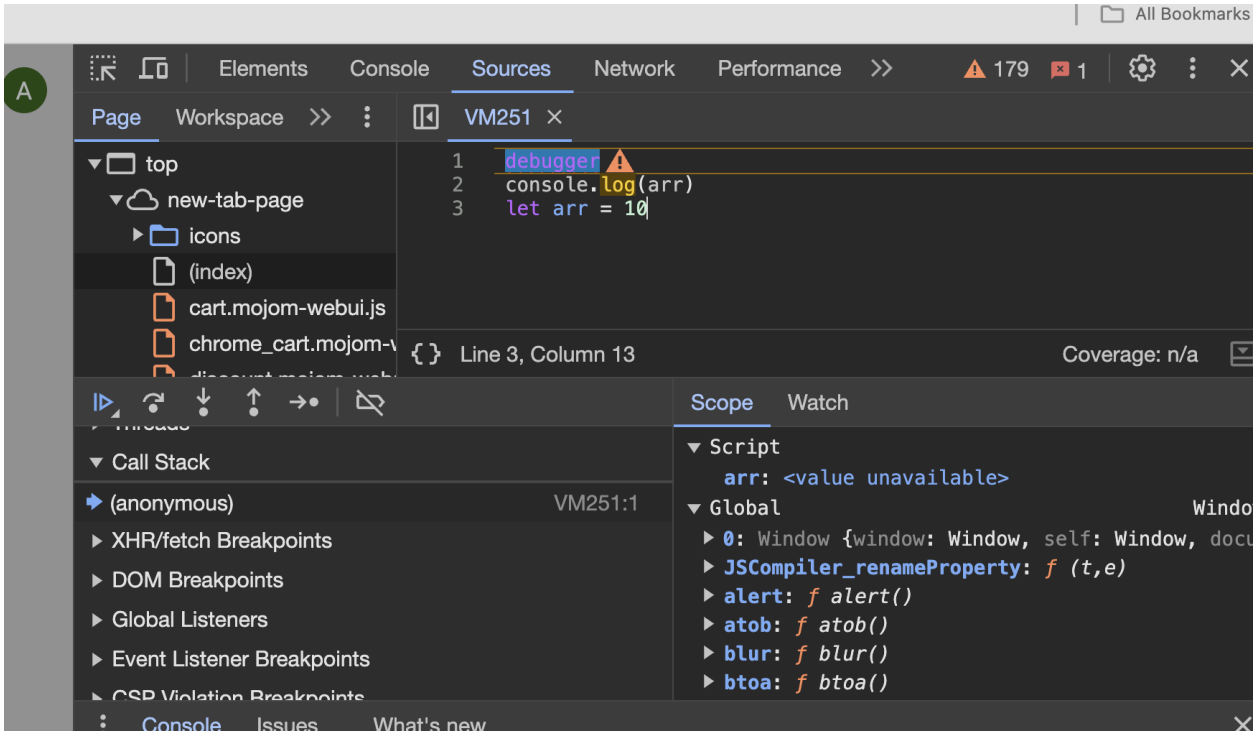
Behavior with let and const . check using the node index.js command

1. console.log(a)

```
printName()
let a = 4

function printName(){
   console.log('my name is Mr. X)
}
let printAge = function(){
   console.log(24)
}
printAge()
```

2. Lets see the behavior in dev tool with a simple example



3. If you notice , the value of arr is inside something called as Script scope (wll cover scopes later)
4. The point is that even before the line #3 is reached, arr is there somewhere in the memory but the value is not available to be used.
5. So hoisting does happen for let and const as well (memory allocation is done in the memory allocation phase), it is just taht these values are in temporal dead zone until declaration line is reached and not available for use.