Everything in javascript happens inside execution context. U can assume it as a bigbox where whole js code will get executed.

**Execution context – will have 2 components in it –**

1)**memory component-** here all variables and funcs are stored as key value pairs eg: a:10 , func: {…}

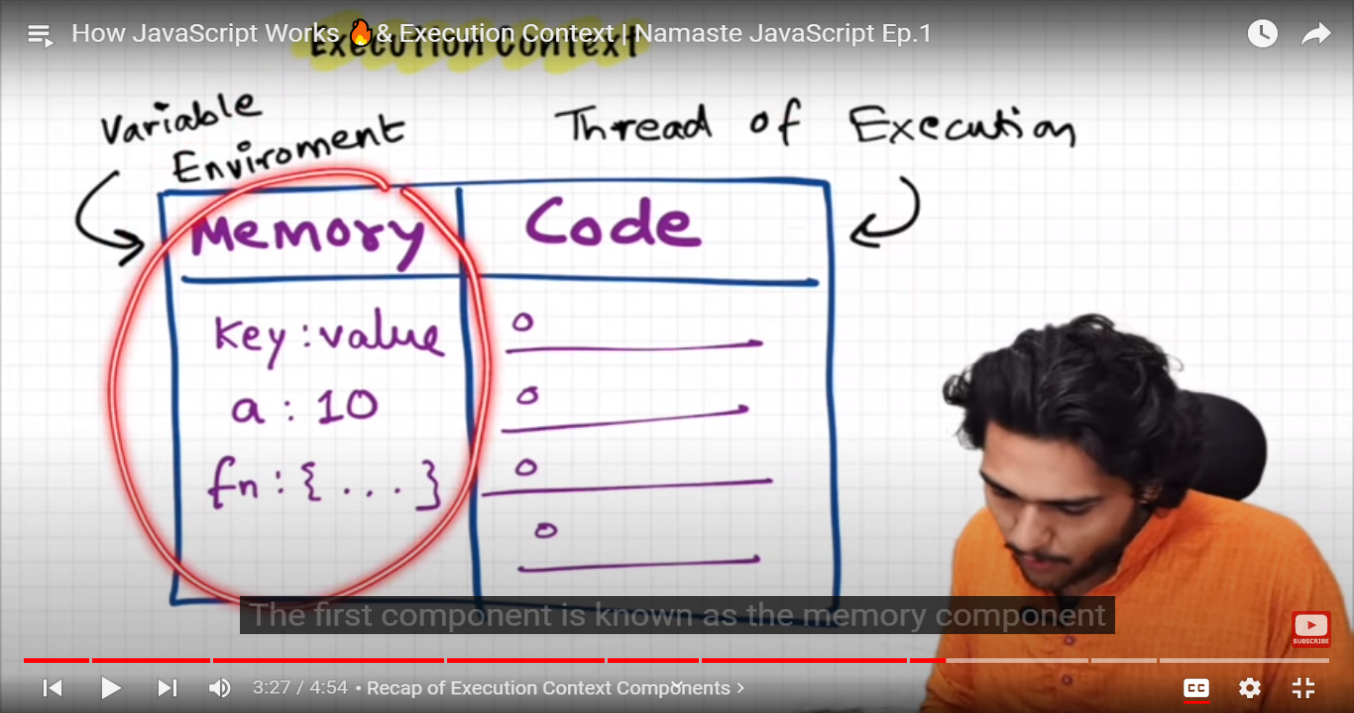
Memory comp also known as **variable environment**.

2)**code component**- it is the place where code will get executed one by one line

It is also known as **thread of execution**. It is like a thread where whole code is executed- one line at a time.

* Javascript is synchronous and single threaded language.

It means js can execute one command at a time and in specific order (one by one)



**Day 2: What is call stack?**

Lets take some example code

        var n=2 ;

        function sqaure(num){

            var ans=num \*num;

            return ans;

        }

        var square2=sqaure(n);

        var sqaure4=sqaure(4)

add this inside script tag

when u run this whole code, a execution context will get created.

This execution context will be created in 2 phases –

1. Memory Creation phase
2. Code execution phase

In first phase , js will allocate mem to all variables and functions

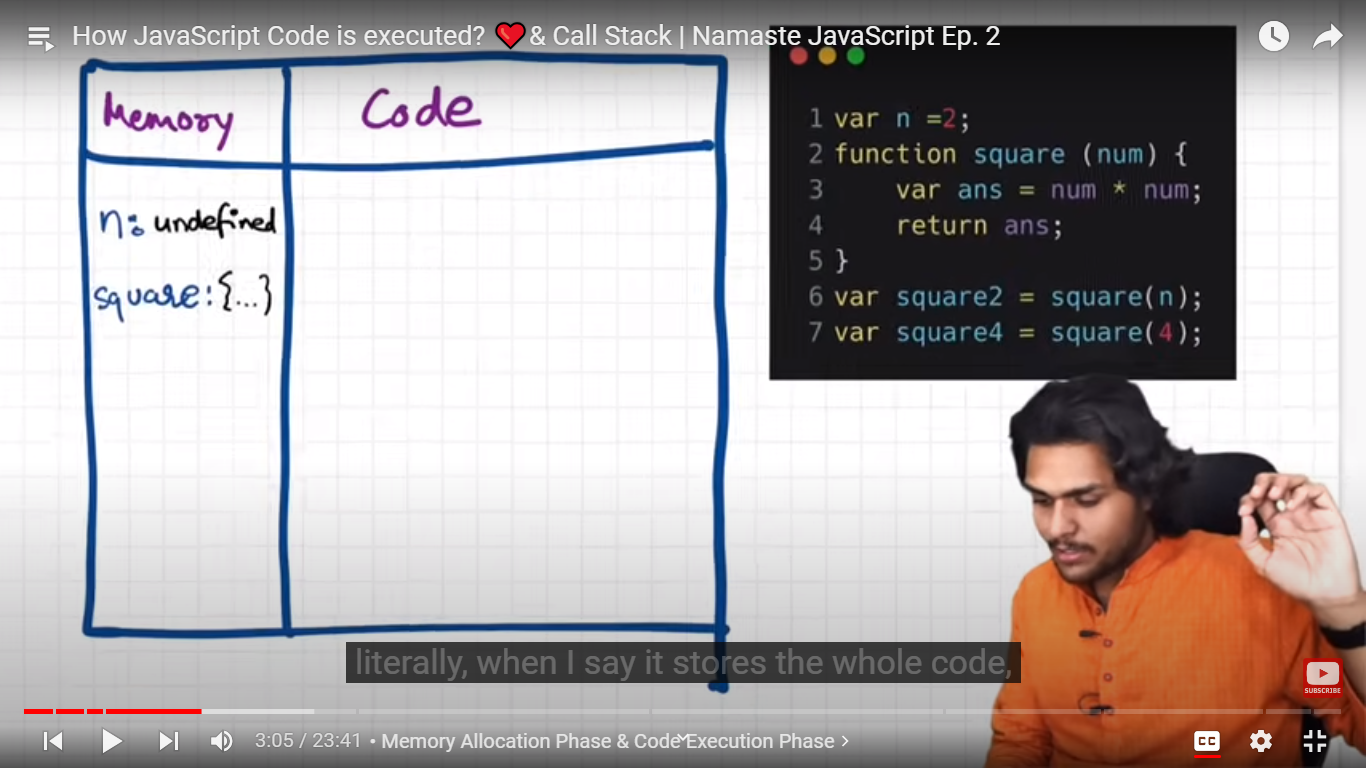
**Process:**

At line one, it will allocate memory to n in memory comp – eg: n:

At line two ,it allocate memory to function square in mem comp – eg: square:

It stores its values as undefined in case of variables .

In case of functions , it stores the whole code as its value



It also store memory for sqaure2 and square 3, as they are variables, their value will be undefined

Code execution phase: 2nd phase- after mem allocation, the 2nd phase will start

Here code actually gets execute, js will run whole code once again and now it executes code.

Var n=2 , 1st line , now value 2 will be get placed in place of undefined at identifier/holder n

Form line 2 to 5 – there is nthng to execute .

Then line 6- we invoke a function.

A function is a like a mini pgm

Whenever a func gets called, a new execution context will get created



Again there will be 2 phases inside this nested execution context.

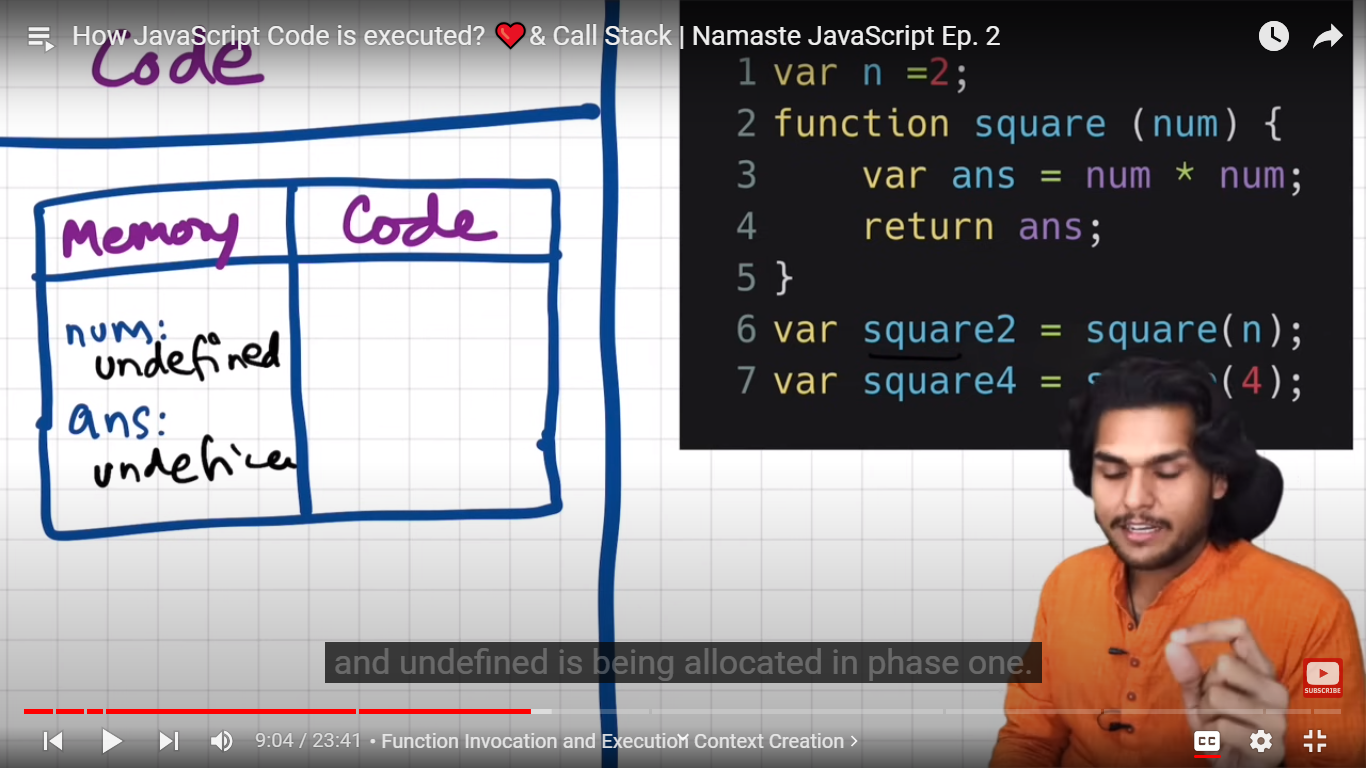
**Phase1: mem allocation to vars(parameter and variables) and funcs( which are inside square func only).**

Allocation mem to num and ans – we get its values as undefined.=> only allocation of mem done, no assignment of values.

**Nested Memory comp:**

**Num:undefined**

**Ans:undefined**



**Phase2: code execution phase**

Whenever we called the func here square(n) here the n value – whichc is 2 will get passed to the function

Num is known as parameter of function

N is known as argument .

* The parameter which we pass to func while calling it is known as argument
* The parameter of func in function definition is known as a parameter.

Now

Nested mem comp:

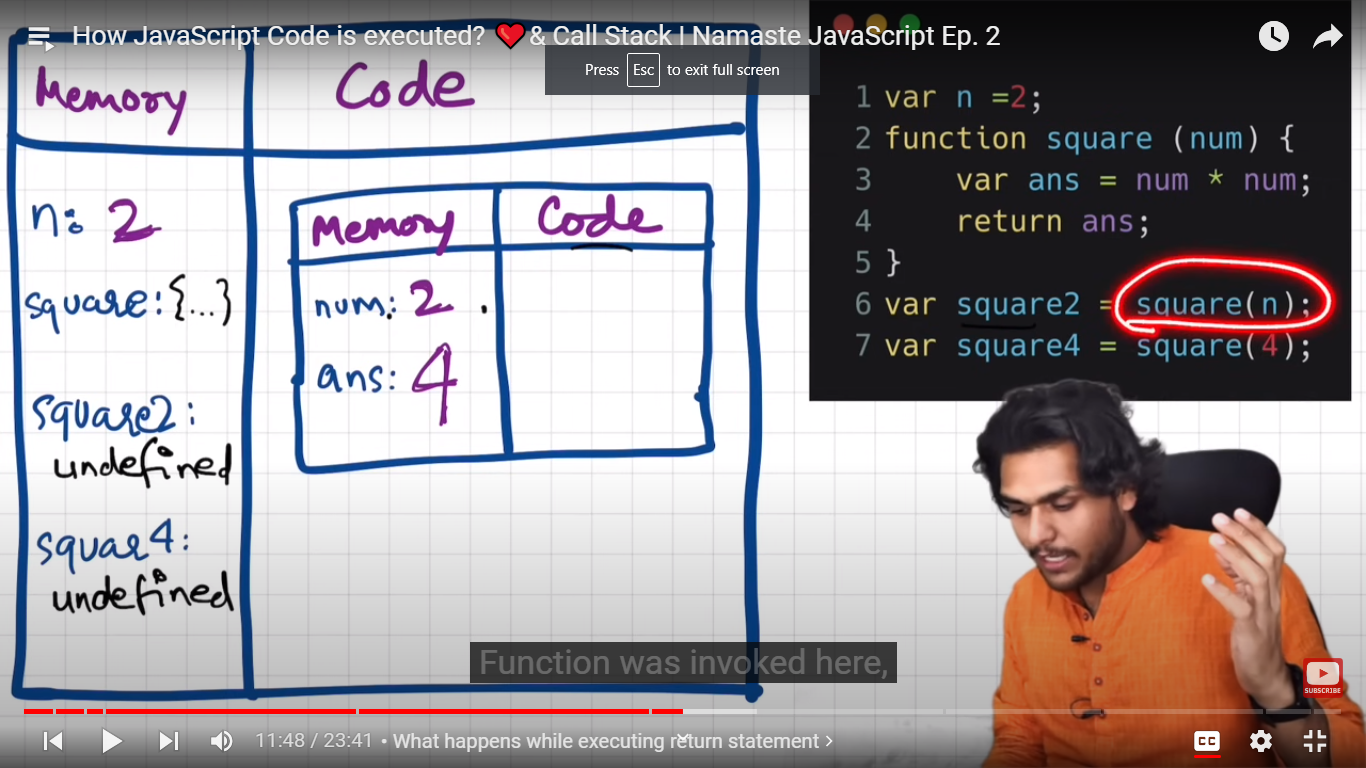
num:2

next line ans=num\*num

it will calculate value (num\*num ) in code comp and then replace undefined value of ans with the calculated value

|  |  |
| --- | --- |
| Mem comp | Code comp |
| num: 2 | Num\*num |
| Ans= 4 |  |

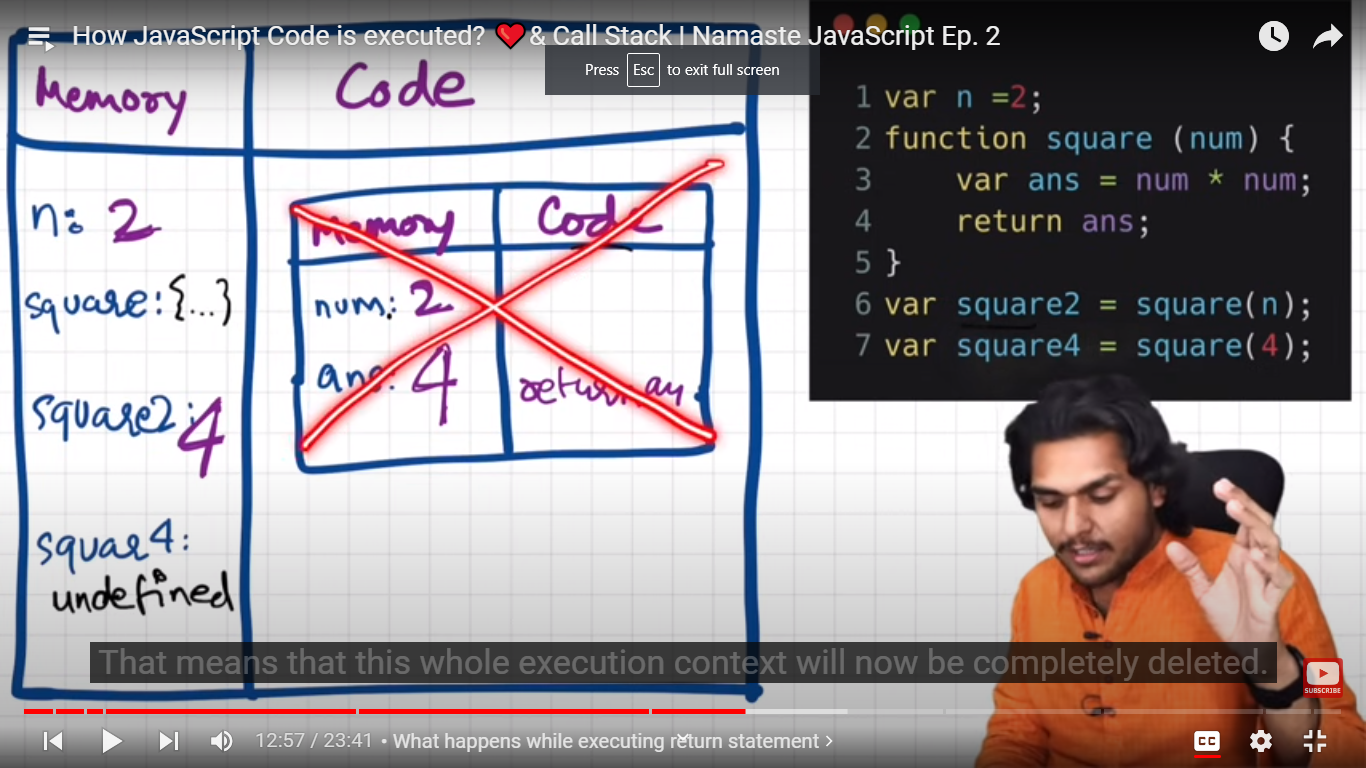
Now return ans line – return stmt will return control back to execution context where the func was invoked.



Func was invoked here at sqaure2

Now value 4(ans) will get replaced with undefined value of sqaure2().

When the func is executed totally and returned, then its execution context instance will get deleted.



Now line 7- var suare4=square(4)

Again a new execution context created . same process

1st phase:

Num:undefined

Ans:undefined

2nd phase:

num:4

ans= 16

then this execution context deleted

square4: 16

now js done with whole work, whole pgm execution . now whole global execution context get deleted.

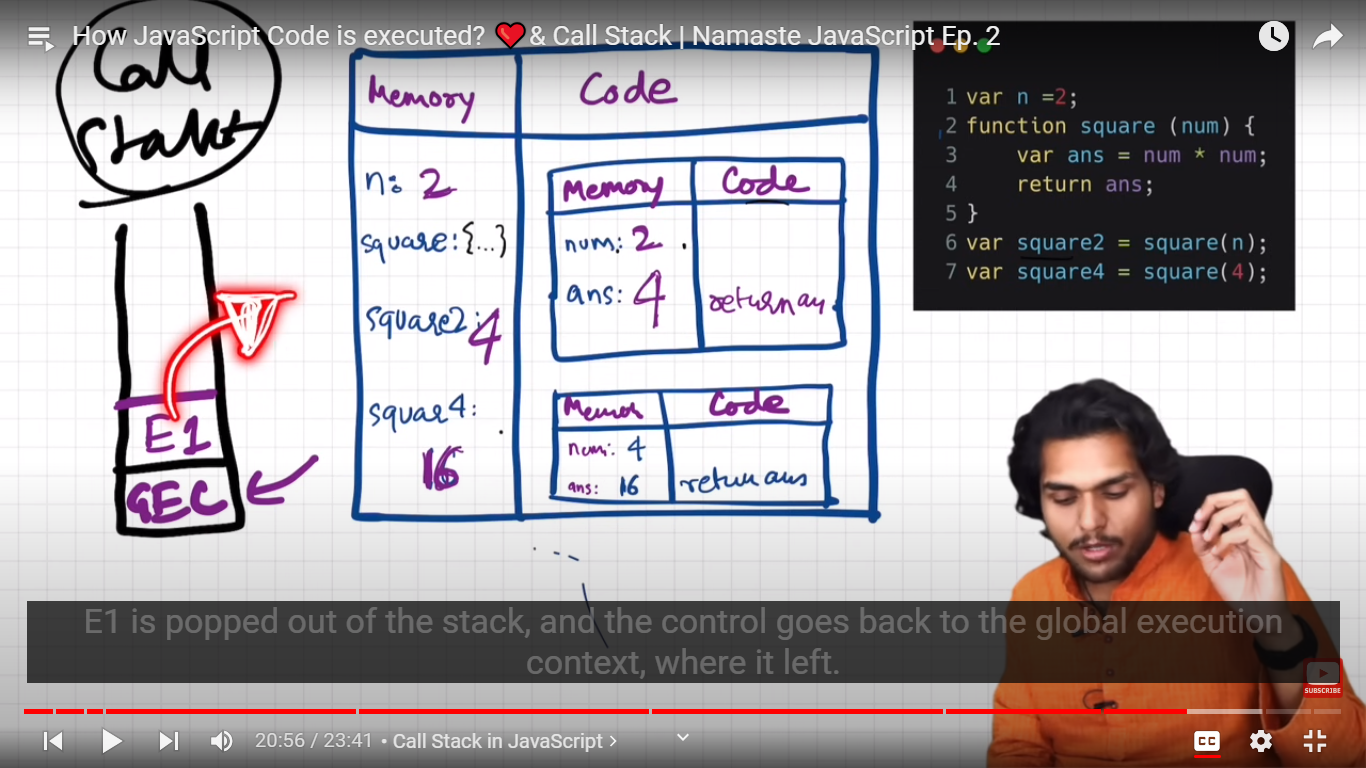
Suppose we have more nested functions- then need to create ,delete and manage many execution contexts.

So to manage it, js engine maintains a callstack

**Call stack:**

Is like a stack, in bottom we have global execution context,this whole execution context will get pushed into stack.

Whenever a new func gets invoked, a new execution context gets created on top of it.



So call stack will only manage all execution contexts.

**Callstack maintains the order of execution of execution contexts.**

Callstack also known as

1)execution context stack

2)pgm stack

3) cntrl stack.

4) runtime stack

5) machine stack.

**Day 3:**

* **Hositing in Javascript:**

**Let**  us create a variable and a function . and call that func and print that variable

        var x=7;

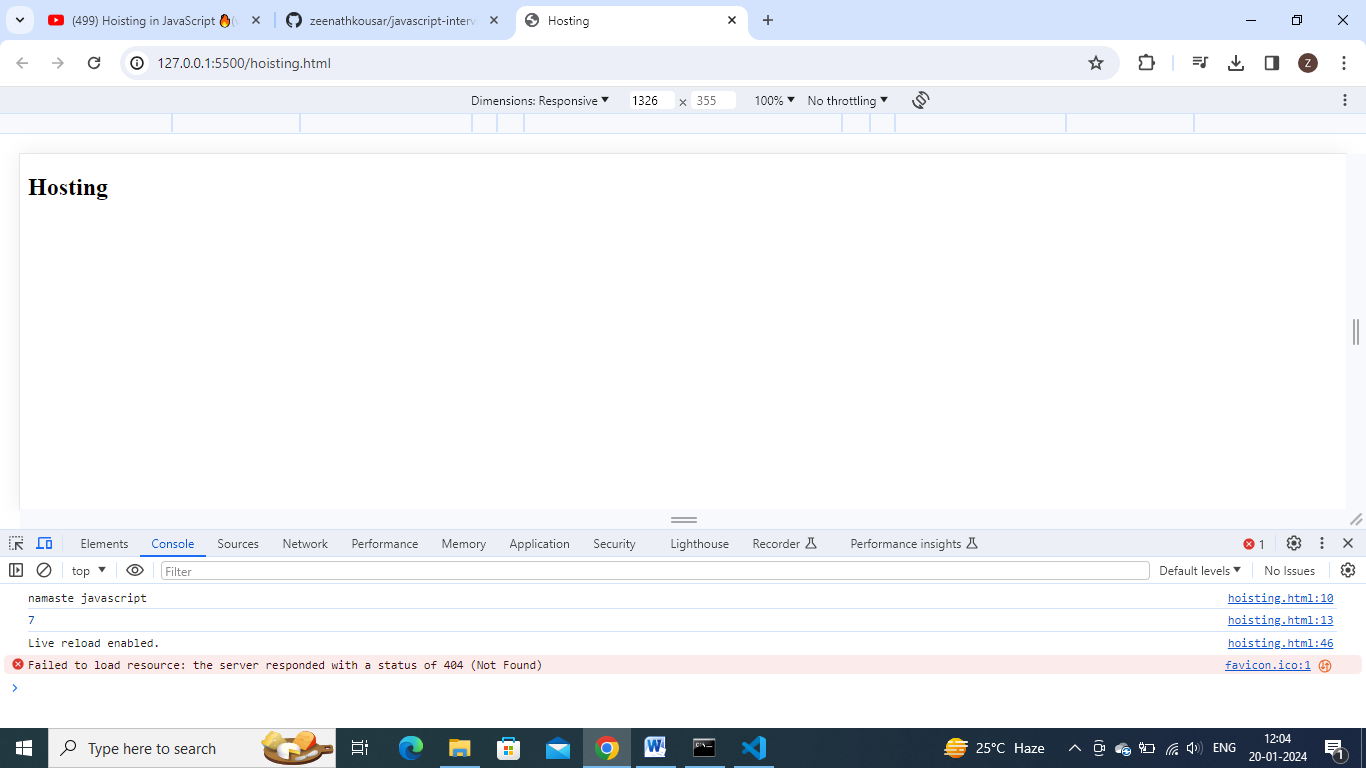
        function getName(){

            console.log('namaste javascript')

        }

        getName();

        console.log(x)



Now copy and paste that printing var and calling func stmts on to the top.

    <script>

         getName();

        console.log(x)

        var x=7;

        function getName(){

            console.log('namaste javascript')

        }

    </script>



**In remaining pgmg languages – it gives err – u cannot access variable witout/before initialization and we cannot call a func without/ before defining it.**

**In javascript.**

**Somehow func is getting called , but variable got undefined.**

    <script>

         getName();

        console.log(x)

        // var x=7;

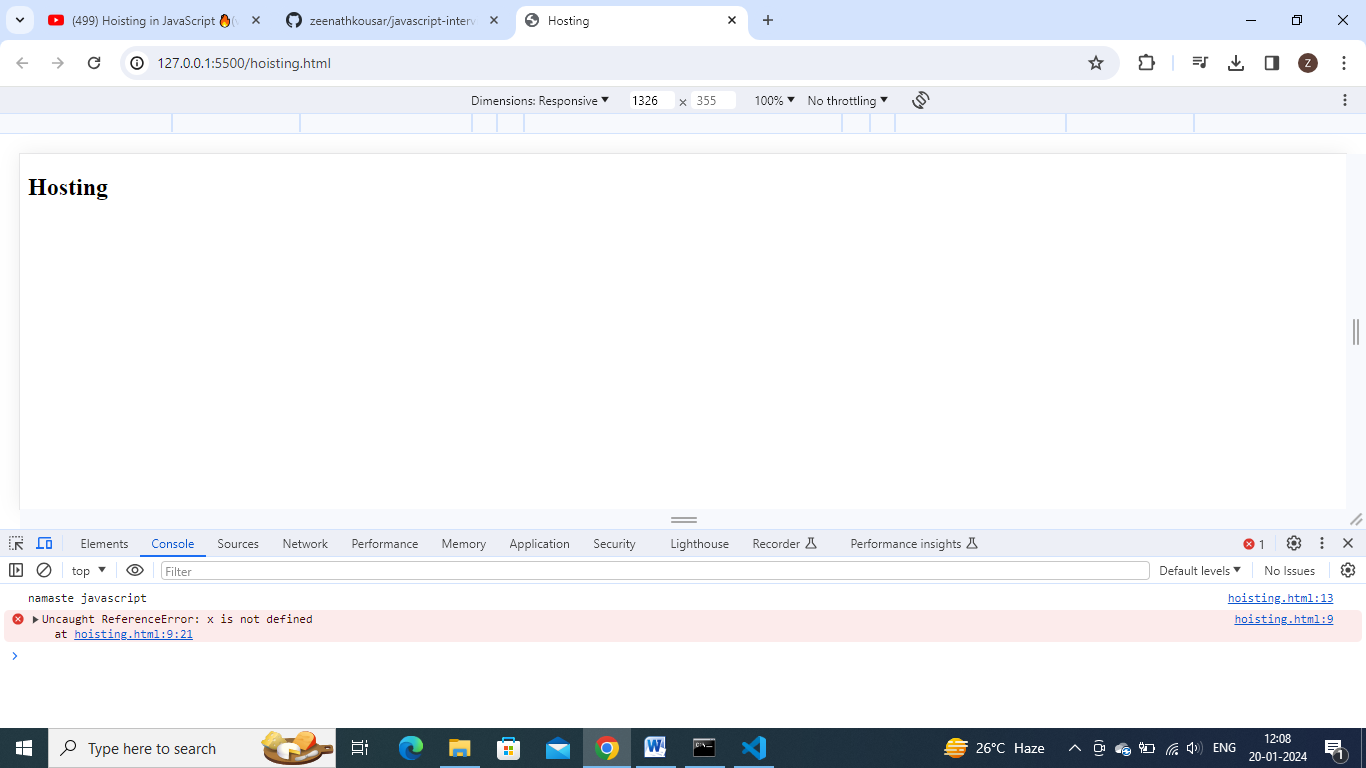
        function getName(){

            console.log('namaste javascript')

        }

    </script>

Remove var x=7 line , u get error now



U get reference err – x is not defined

**Hoisting : is a phenomenon in js, In which u can access these variables and funcs even before initializing it. U can access it without any err.**

        var x=7;

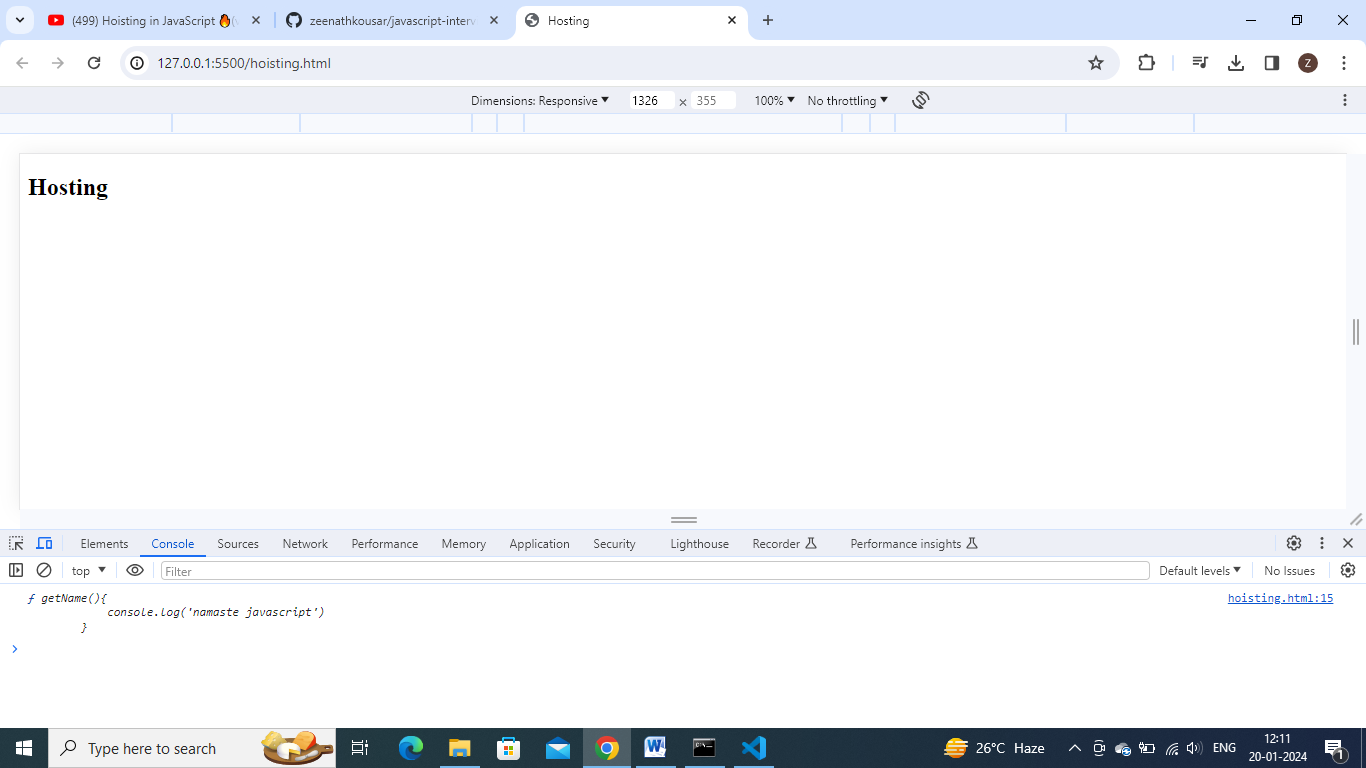
        function getName(){

            console.log('namaste javascript')

        }

        console.log(getName)

console.log(getName) – will print func code as output.



        console.log(getName)

        var x=7;

        function getName(){

            console.log('namaste javascript')

        }

**Moving console.log(getName) to upside , also we got same func code as output.**

So in case of variable x => console.log(x) before initialixing is getting undefined

But console.log(func) before initializing func, we get func code.

Bcoz run this code in terms of memory execution and code execution.

        var x=7;

        function getName(){

            console.log('namaste javascript')

        }

        getName();

        console.log(x)

        console.log(getName)

**reason: so funcs , before running/ before execution (2nd phase) only, funcs will get initialized with func code, but vars initialize with undefined-**

run and check memory wise this code .:

phase 1:

mem comp

x:undefined

getName: {….}

**reason: so funcs , before running/ before execution (2nd phase) only, funcs will get initialized with func code, but vars initialize with undefined-**

phase2:

x:7

        getName();

        console.log(x)

        console.log(getName)

        var x=7;

        function getName(){

            console.log('namaste javascript')

        }

Notdefined vs undefined:

A variable got memory but not initialized with a value => undefined

A variable did not got memory => undeclared

        getName();

        console.log(x)

        console.log(getName)

        // var x=7;

        function getName(){

            console.log('namaste javascript')

        }

In this code at console.log(x) line, x is not present in memory . so got undeclared error.

**Arrow funcs:**

        getName();

        console.log(x)

        console.log(getName)

        var x=7;

        // function getName(){

        //     console.log('namaste javascript')

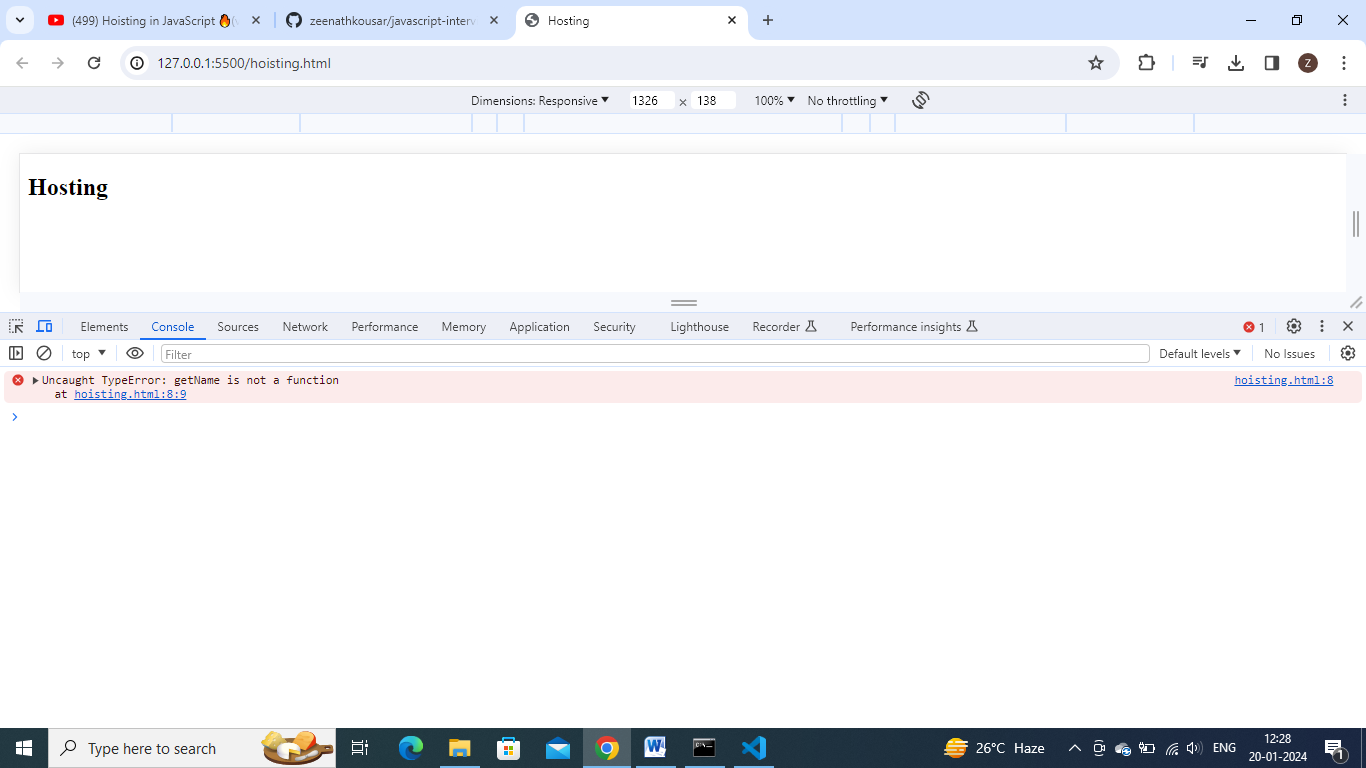
        // }

        var getName=()=>{

            console.log('namaste hello')

        }

**As arrow funcs – means assigned func value to a variable=> err**



**Arrow func behaves like a variable.**

        //another way to define func

        var func2= function getName (){

        }

**In this way also,It work as a variable**

**Day 4: how functions work in js and variable environment:**

