Akshay

Saini

Namaste

Javascript

Youtube

https://youtube.com/playlist?list=PLlasXeu85E

9cQ32gLCvAvr9vNaUccPVNP

HandBook

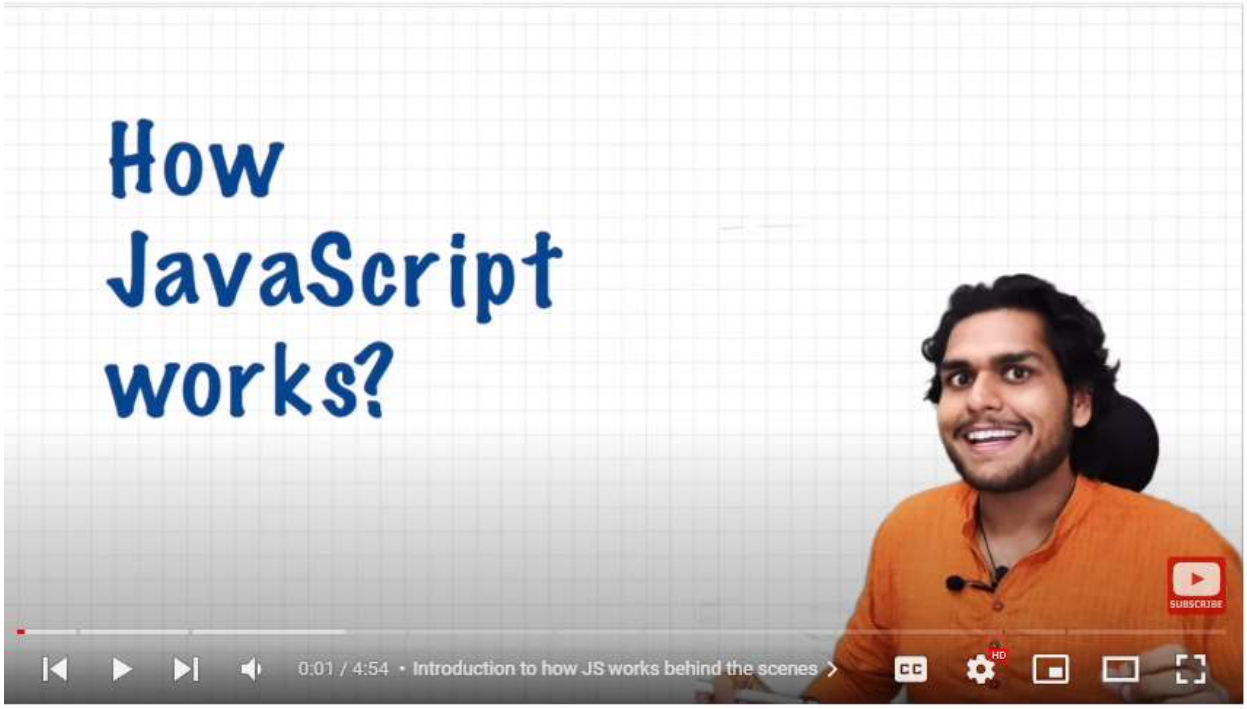
This handbook has been made by Akash Kinkar Pandey.
LinkedIn-https://www.linkedin.com/in/akash-pandey-5985361b8/
Github-https://github.com/akashkinkarpandey
Twitter-https://twitter.com/prokoding

Index

Video Number	Topic	Page Number
1	How JavaScript Works 🔥& Execution Context | Namaste JavaScript Ep.1	5
2	How is JavaScript Code executed? ❤️& Call Stack | Namaste JavaScript Ep. 2	8
3	Hoisting in JavaScript 🔥(variables & functions) | Namaste JavaScript Ep. 3	22
4	How functions work in JS ❤️ & Variable Environment | Namaste JavaScript Ep. 4	35
5	SHORTEST JS Program 🔥window & this keyword | Namaste JavaScript Ep. 5	47
6	undefined vs not defined in JS 🤔 | Namaste JavaScript Ep. 6	51
7	The Scope Chain, 🔥Scope & Lexical Environment | Namaste JavaScript Ep. 7	55
8	let & const in JS 🔥Temporal Dead Zone | | Namaste JavaScript Ep. 8	64

9	BLOCK SCOPE & Shadowing in JS 🔥| Namaste JavaScript 🙏 Ep. 9	77
10	Closures in JS 🔥 | Namaste JavaScript Episode 10	105
11	setTimeout + Closures Interview Question 🔥 | Namaste 🙏 JavaScript Ep. 11	115
12	CRAZY JS INTERVIEW 🤯ft. Closures | Namaste 🙏 JavaScript Ep. 12	125
13	FIRST CLASS FUNCTIONS 🔥ft. Anonymous Functions | Namaste JavaScript Ep. 13	138
14	Callback Functions in JS ft. Event Listeners 🔥| Namaste JavaScript Ep. 14	149
15	Asynchronous JavaScript & EVENT LOOP from scratch 🔥 | Namaste JavaScript Ep.15	160
16	JS Engine EXPOSED 🔥 Google's V8 Architecture 🚀 | Namaste JavaScript Ep. 16	180

17	TRUST ISSUES with setTimeout() | Namaste JavaScript Ep.17	191
18	Higher-Order Functions ft. Functional Programming | Namaste JavaScript Ep. 18	197
19	map, filter & reduce 🙏 Namaste JavaScript Ep. 19 🔥(Season 1 ends)	201
20	Callback Hell | Ep 01 Season 02 - Namaste JavaScript (Season 2 begins)	207
21	Promises | Ep 02 Season 02 - Namaste JavaScript	211
22	Creating a Promise, Chaining & Error Handling | Ep 03 Season 02 Namaste JavaScript	220



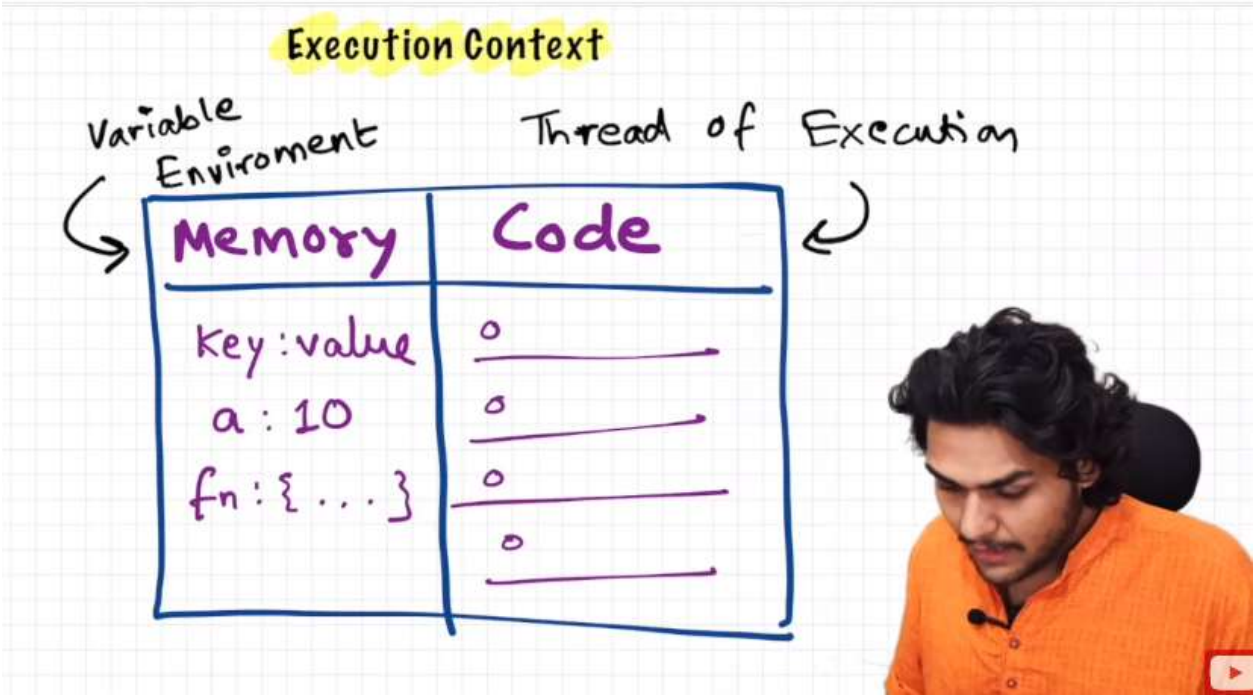
How JavaScript Works 🔥 & Execution Context | Namaste JavaScript Ep.1

Akshay Saini
283K subscribers
Subscribed
47K
Share
Download

How JavaScript Works 🔥 & Execution Context | Namaste JavaScript Ep.1

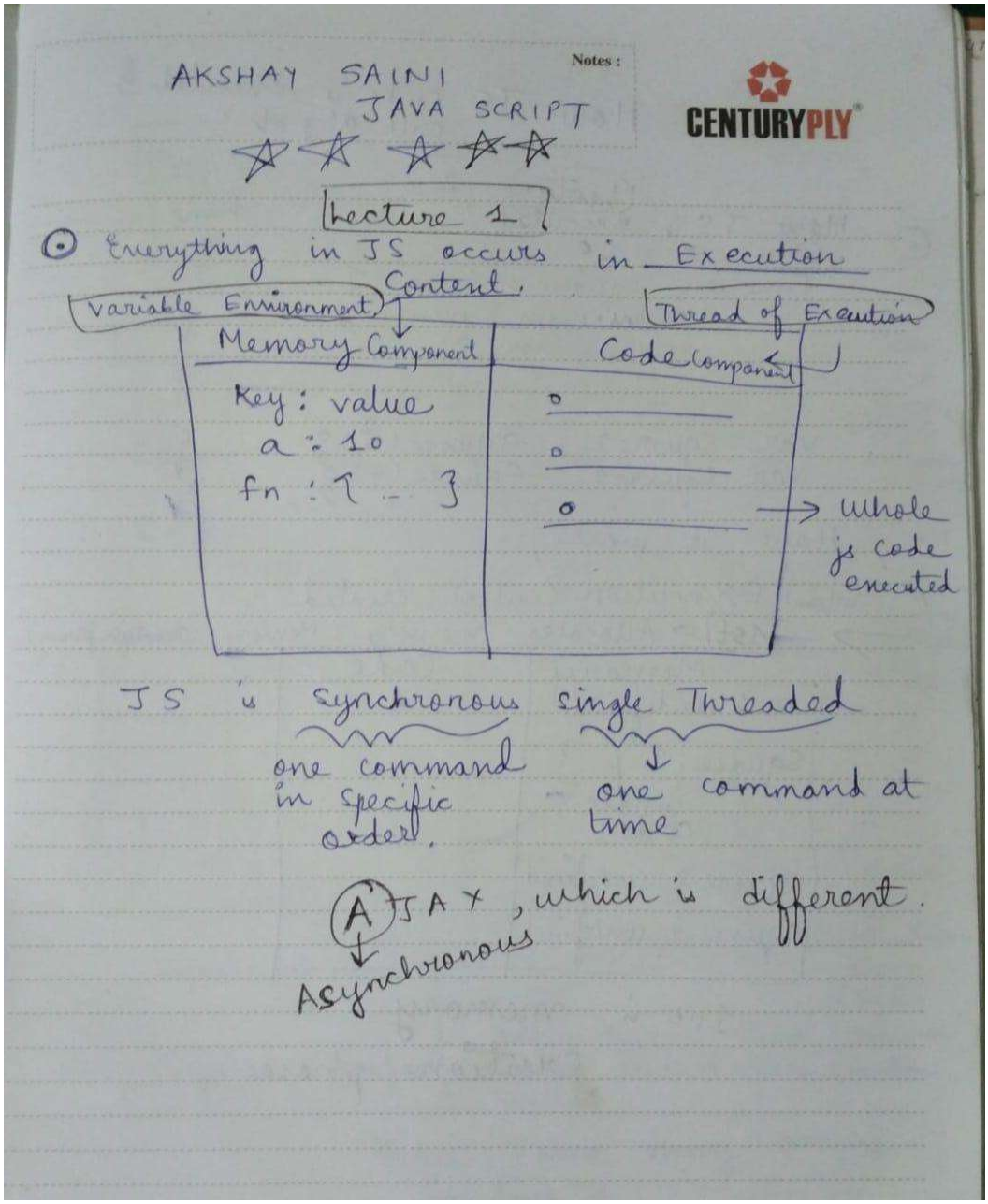
Youtube Link-https://youtu.be/ZvbzSrg0afE

- **Everything inside JavaScript happens inside the *execution context*. We can assume this execution context as a big box or whole container where this Java Script code is executed.**
- **Execution context has 2 components- *Memory Component* and *Code Component.***
- ***Memory component* is also known as a *variable environment*.**
- ***Code Component* is also known as *thread of execution*.**
- **In the memory component the variables and functions are stored as *key value pairs.***
- **In the code component, the whole code is executed *one line at a time*.**



- **Java is a *synchronous single threaded* language.**
- **Synchronous→commands are executed in a specific order.**
- **Single threaded→one command at a time.**
- **It means JS can go to the next line once the current line has finished executing.**
- **THEN what is *AJAX*? Something else known as Asynchronous JavaScript And XML.**

HANDWRITTEN NOTES-



How JavaScript Code is executed? ❤️& Call Stack | Namaste JavaScript Ep. 2

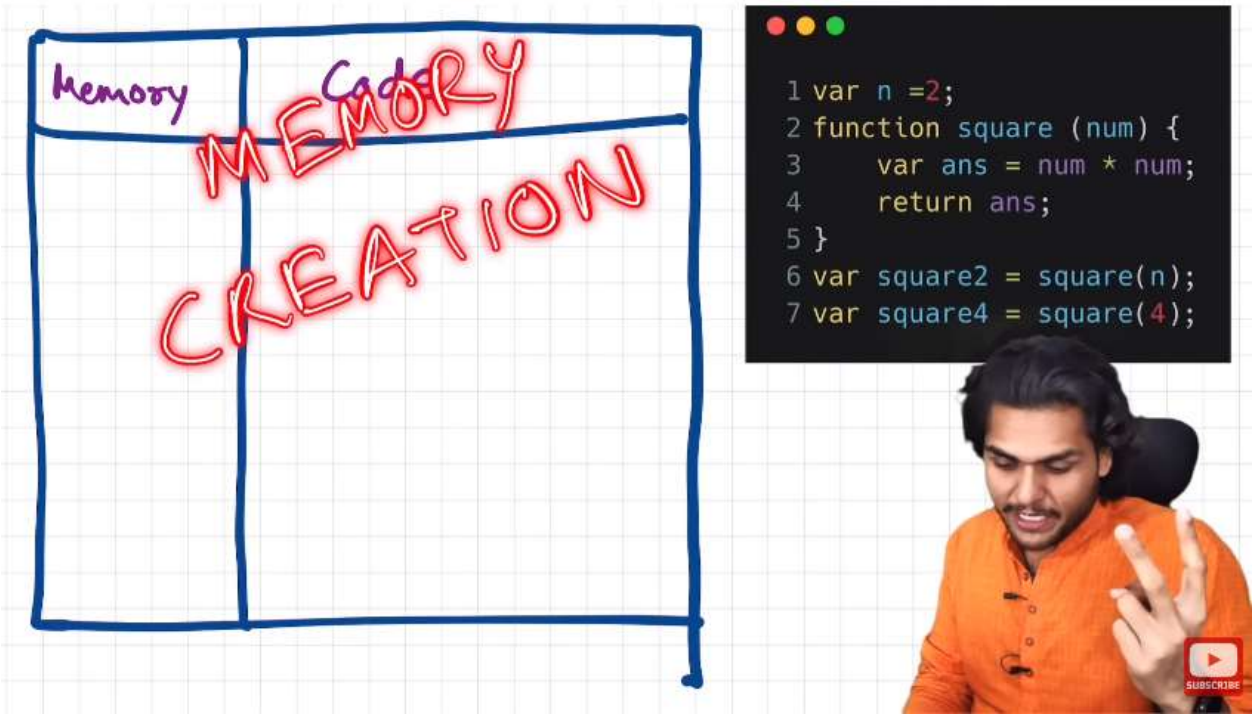
Youtube Link-https://youtu.be/iLWTnMzWtj4

https://ui.dev/javascript-visualizer Visualize JS code

Lets consider the following code and find out what happens behind the scenes.

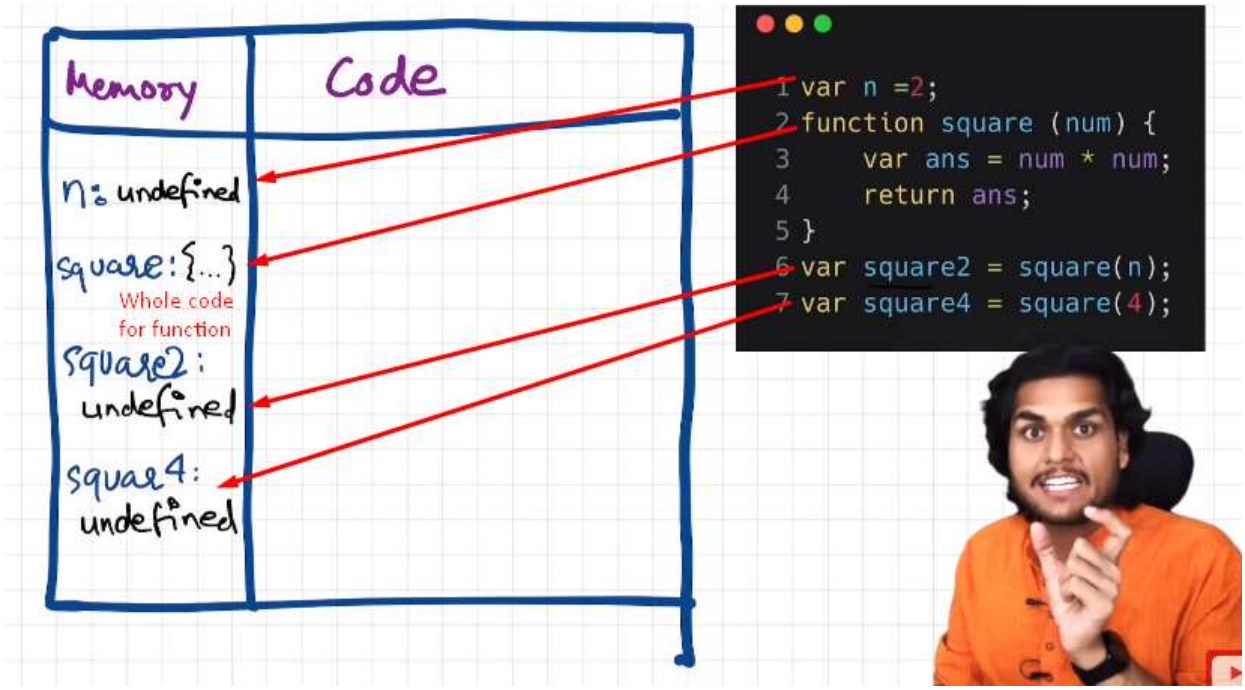
```
var n=2
function square(num){
    var ans=num*num
    return ans
}
var square2=square(n)
var square4=square(4)
```

At first, a global execution context is created.

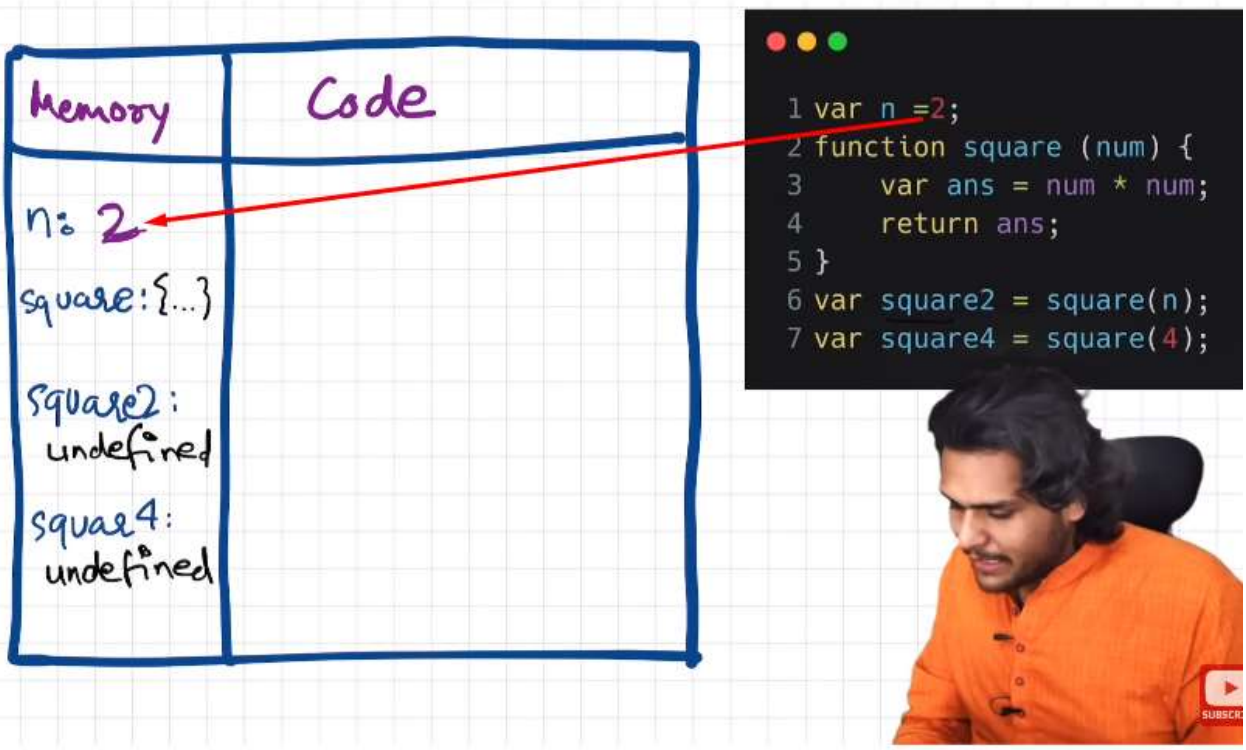


There is a memory component and a code component in the global execution context. There are 2 phases involved→1st one being *memory creation* and 2nd one being *code execution*.

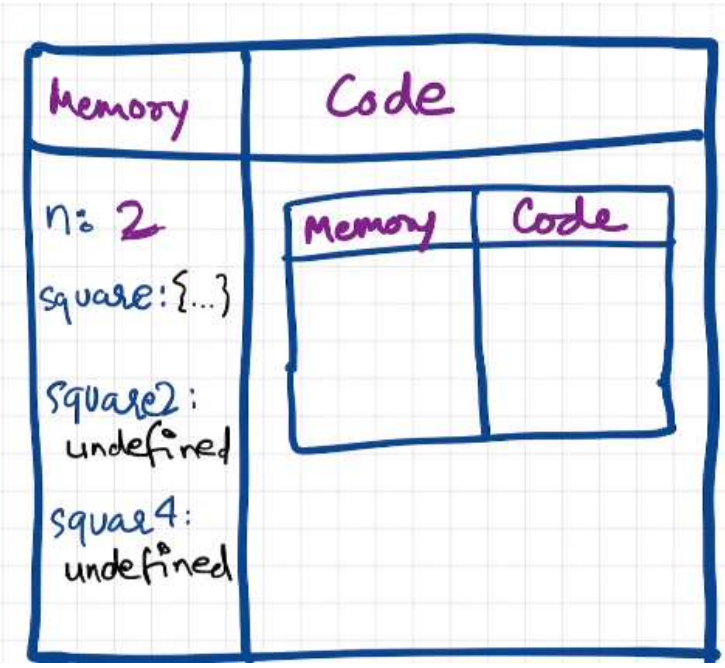
In the 1st memory creation phase JavaScript allocates memory for the variables and function and stores undefined as a placeholder for the variables.In case of function it stores the whole code.

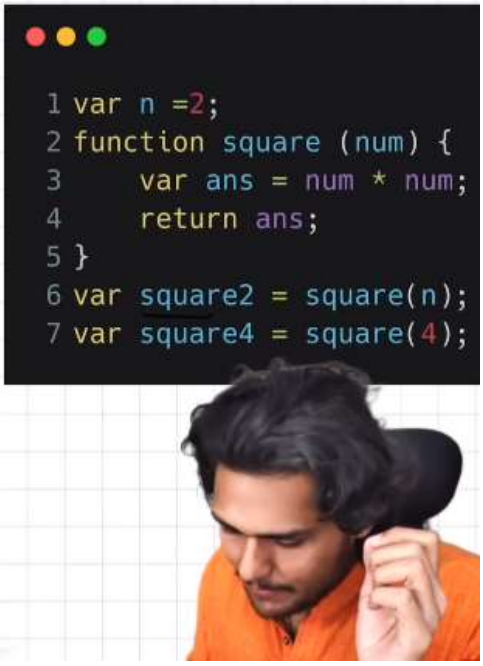


2nd phase is code execution.
Till now value of variables were having undefined. But now they will get the actual values from code.

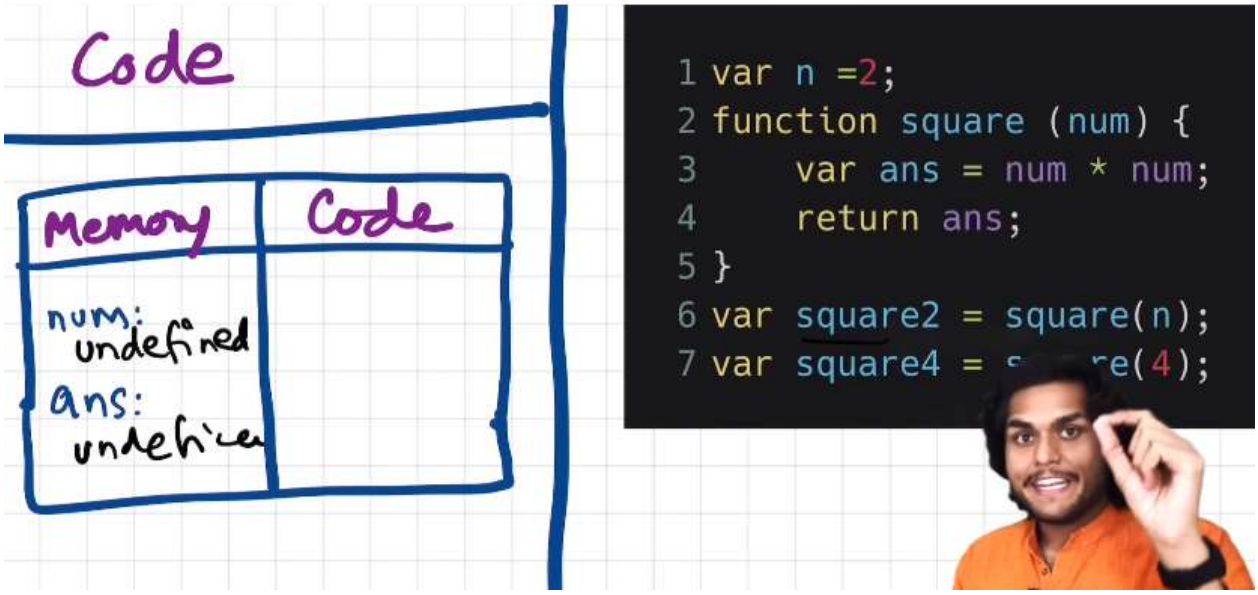


As flow of control moves below, it sees there is a function statement or function declaration.There is nothing much to do,so flow moves to line 6. And sees there is function invocation. Functions are heart of Javascript.They are like a mini program .
When they are invoked,altogether a new execution context is created. Earlier there was global execution context. Now this new execution context for function will also have 2 parts→ memory and code component. It will again have memory creation phase first and then the code execution phase.

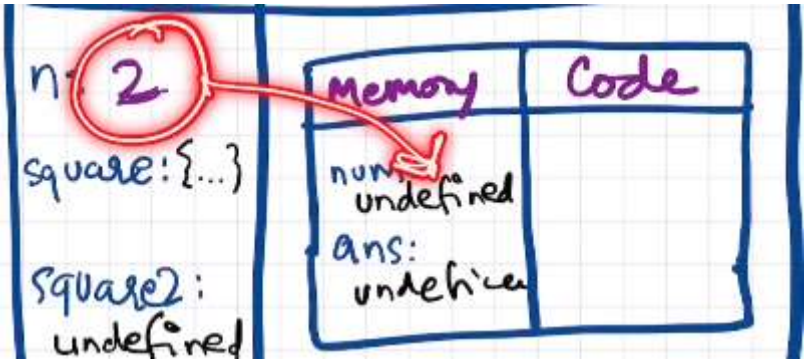




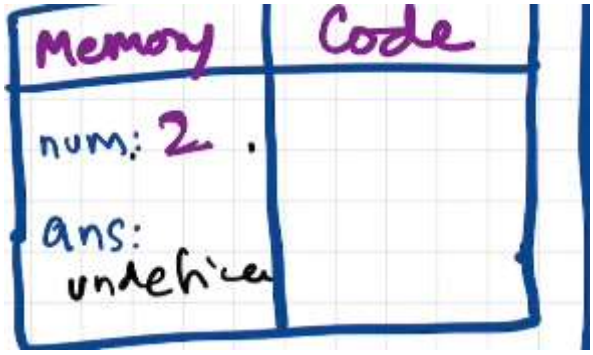
Lets see how memory is allocated here.



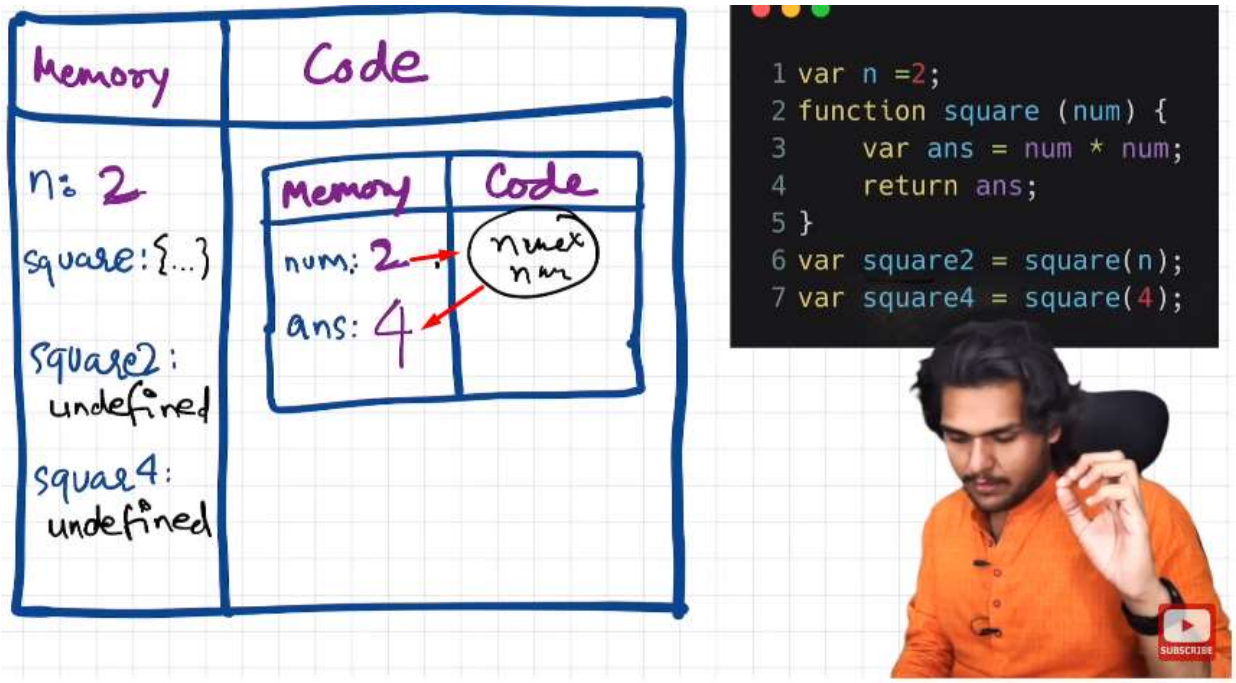
This was the 1st phase that happened.
Now code execution phase for this new execution context will begin where variables will get be allocated the value.



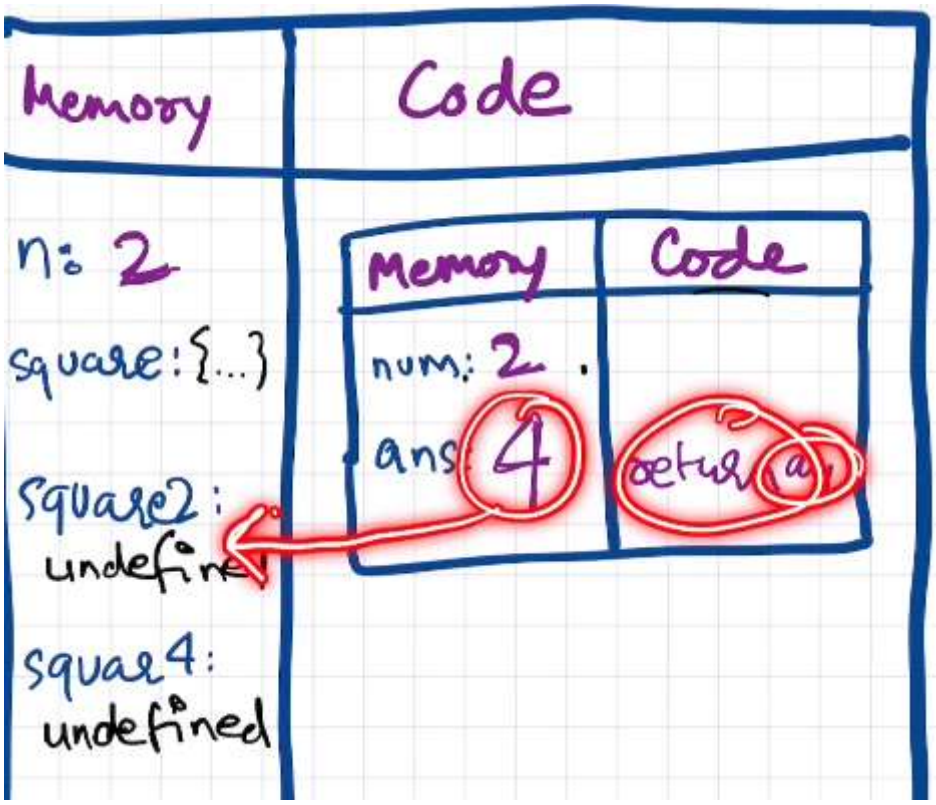
num is parameter, n is argument.

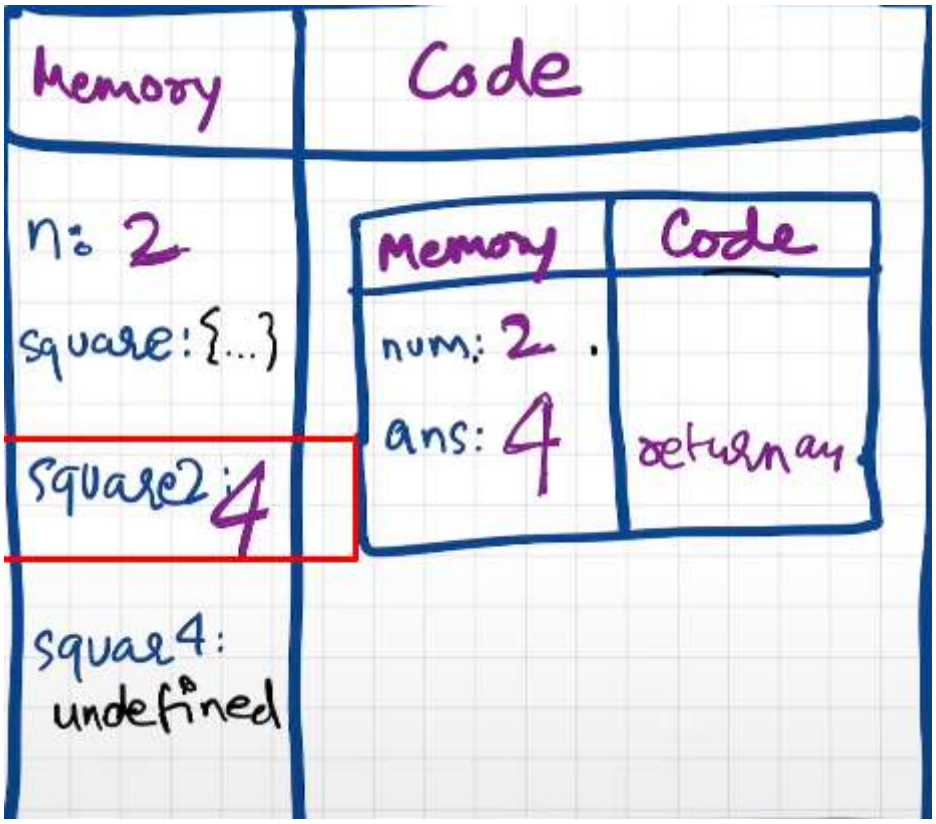


Now var ans=num*num code will be executed in code component and value will be placed in ans variable in memory component.



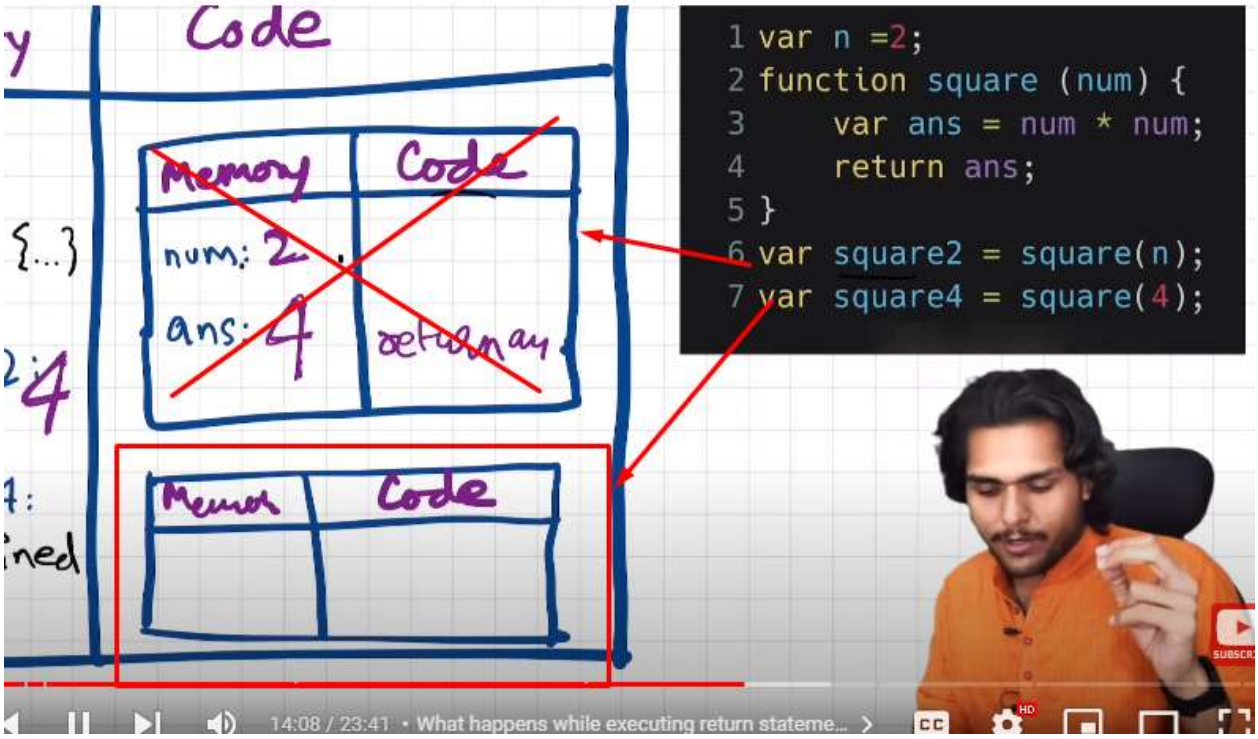
Now we encounter return ans after calculation is over.
Return tells us to return the control of the program to where this function was invoked,that is,back to global execution context over here.

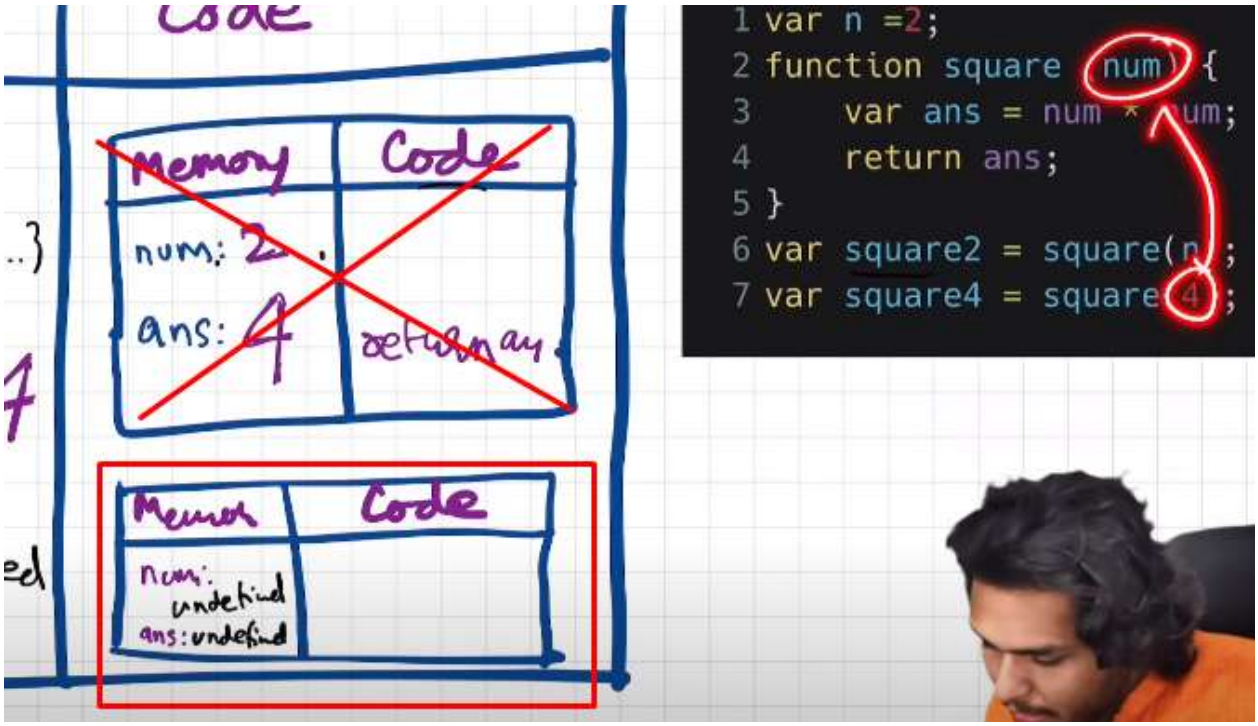




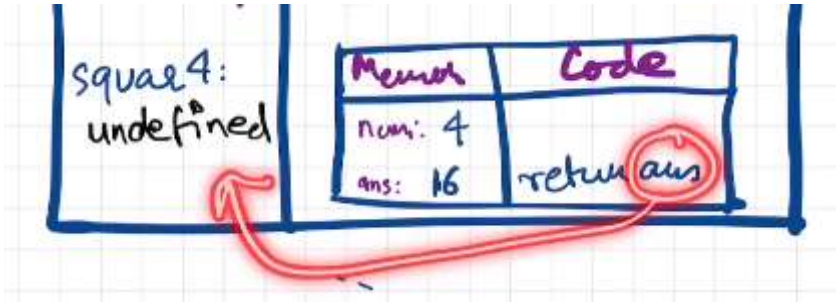
square 2 was replaced with undefined.

Now since the 1st function invocation is over ,it will be deleted.
On line 7,we see yet again the same function invocation happens,so again a new execution context will be created.
Again the memory creation and code execution phase will happen.

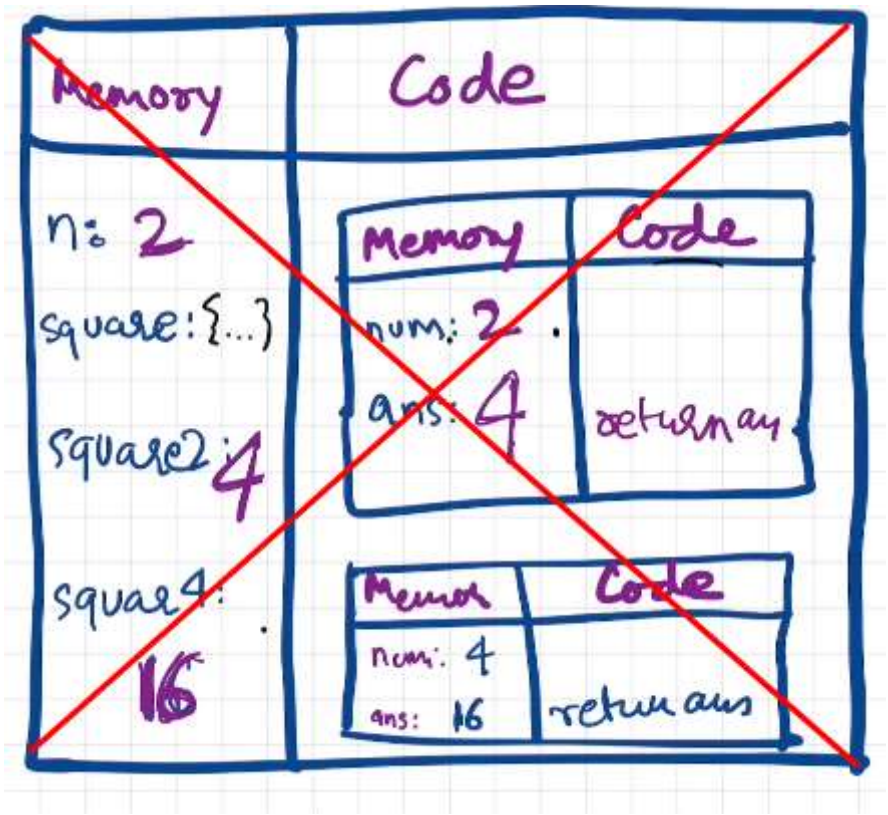




So in the 2nd execution context,variables are yet again assigned a placeholder *undefined.* The code execution happens.In code execution value allocation happens first ,then calculation and returning of value.

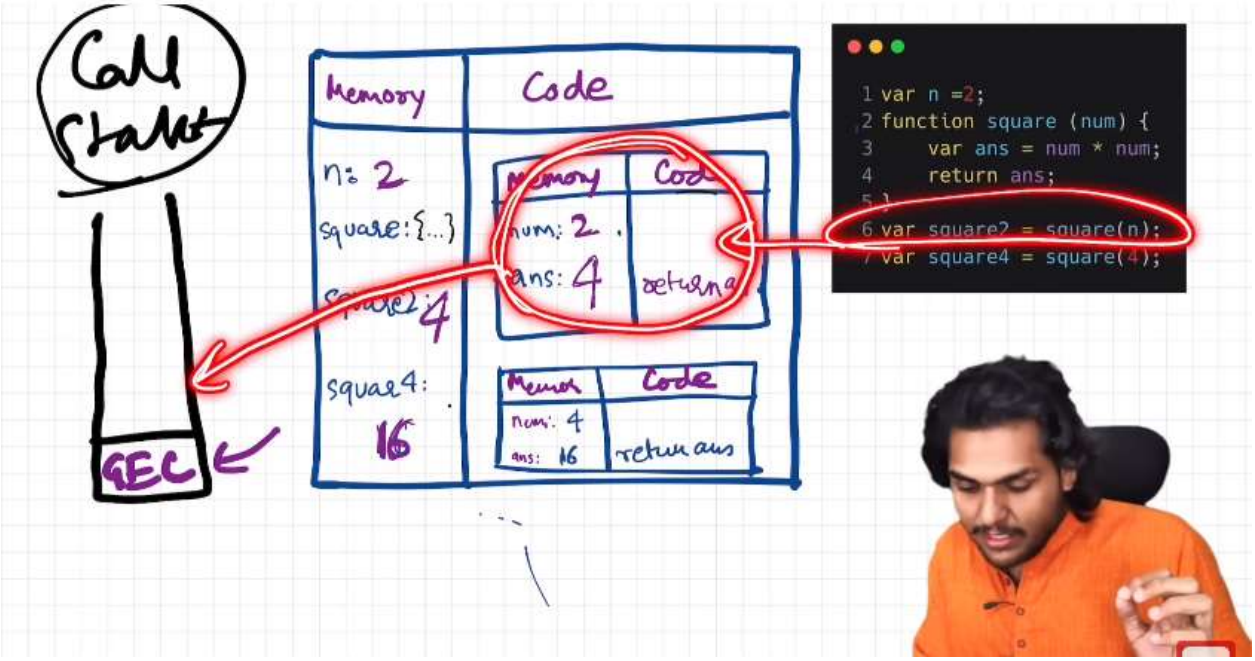


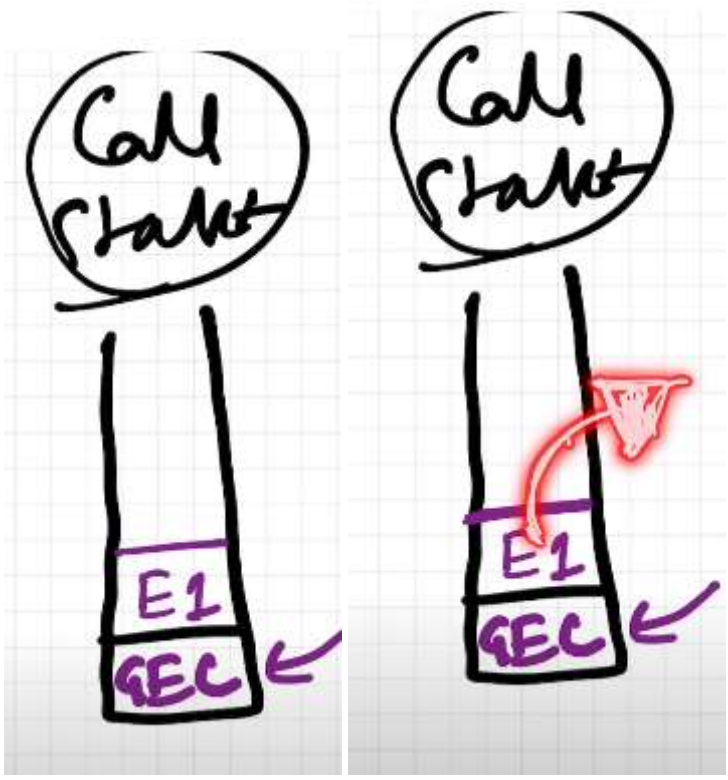
Now the program is finished.
The whole execution context is popped off or deleted.



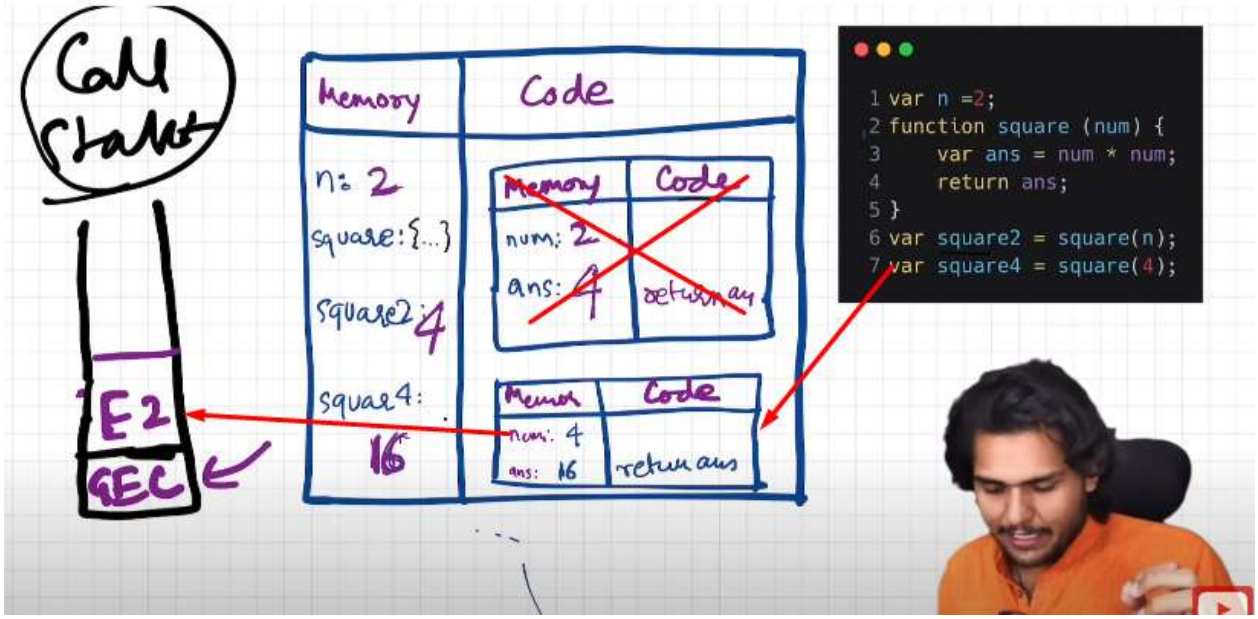
This code was an example of synchronous single threaded javascript.

Javascript manages so many things with the help of *call stack.* At the bottom of stack, known as *call stack,* we have the Global execution context pushed into it, as soon as program starts. During function innovation,when the execution context for the function is created,it is pushed into the stack.

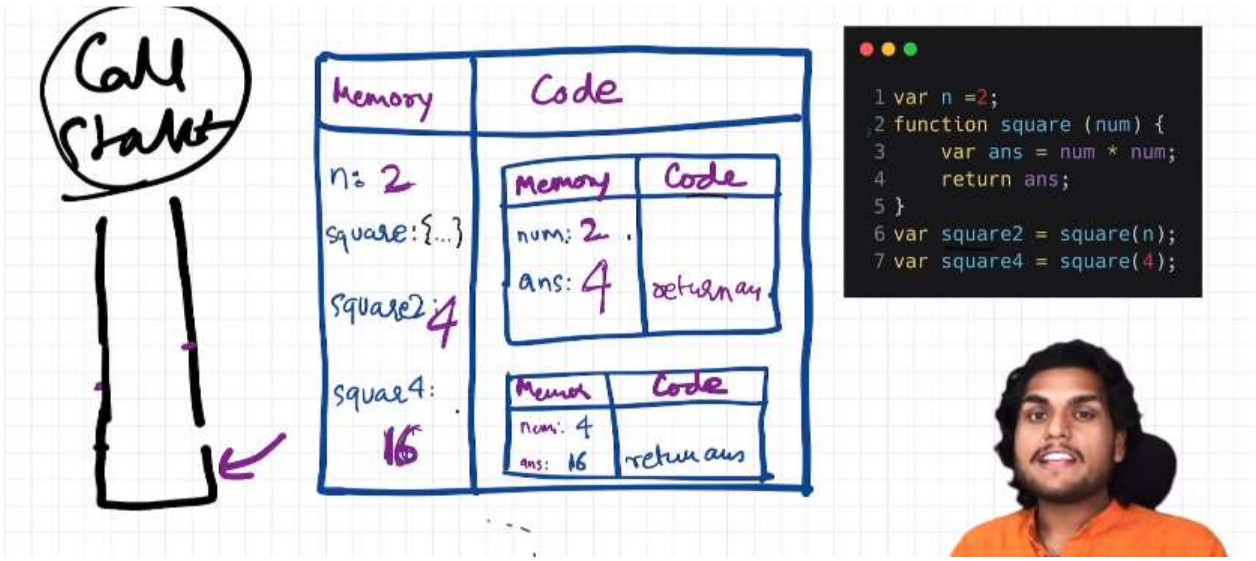




After function execution is over ,its execution context will leave the stack.
Then again when function invocation happens,again execution context for function will be created and it will again be put into the stack.



Then E2 will also be popped out when function has been executed and control will go back to global execution context.
So call stack maintain the order of execution of the execution contexts.
Call stack becomes empty at end as the program ends ,the global execution context is also popped off.

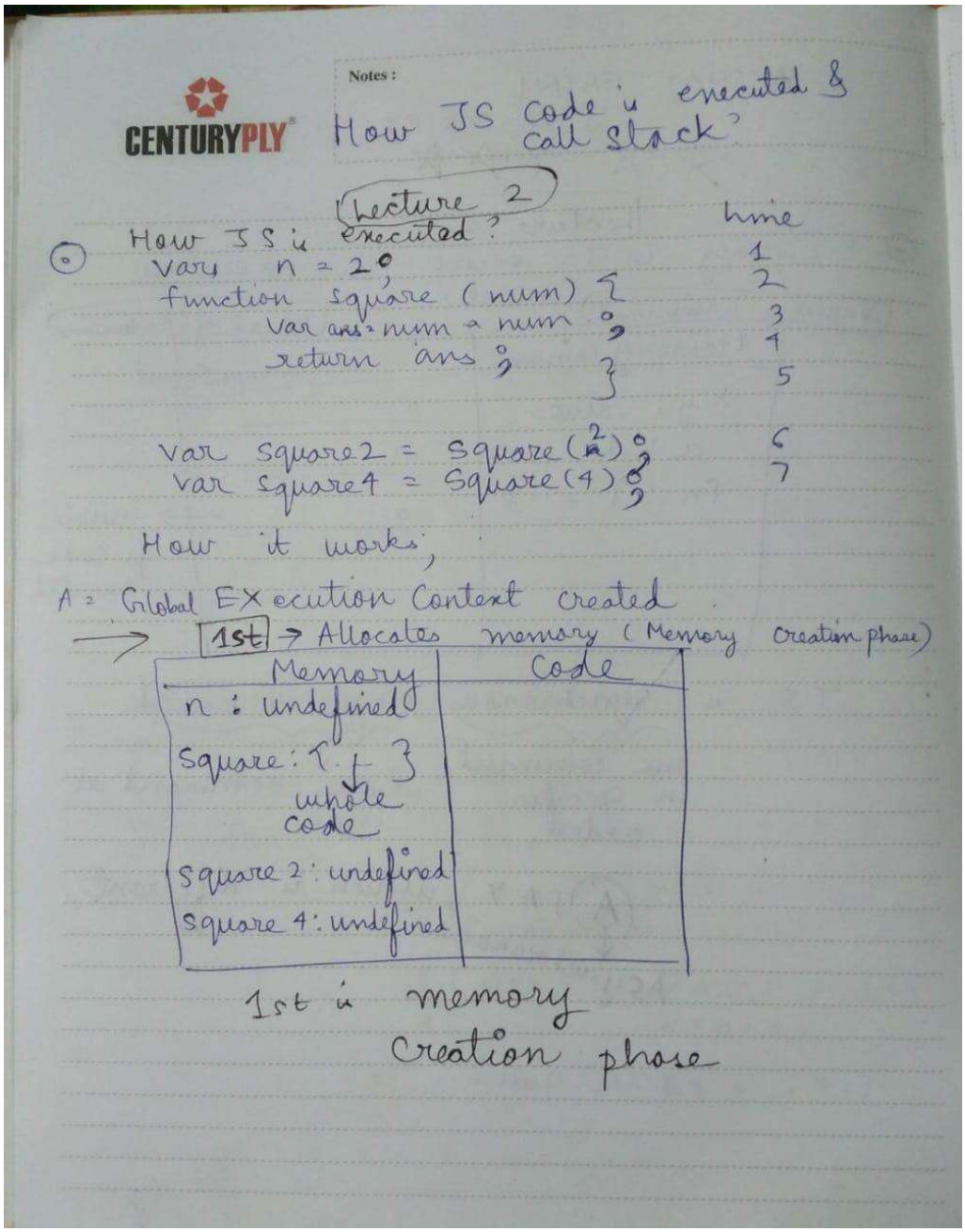


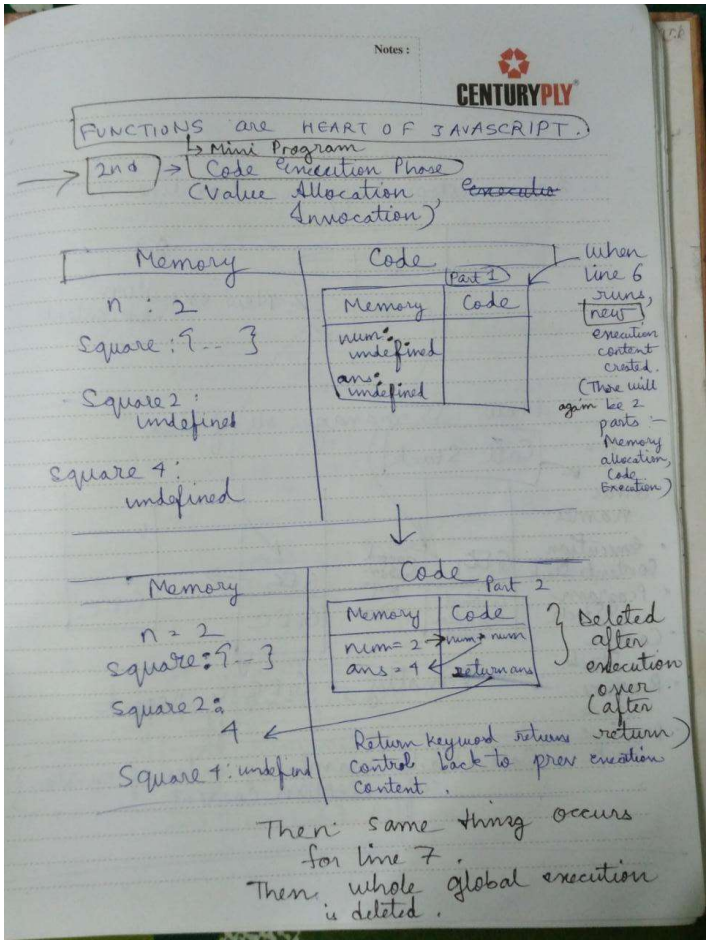
Call stack is also known as

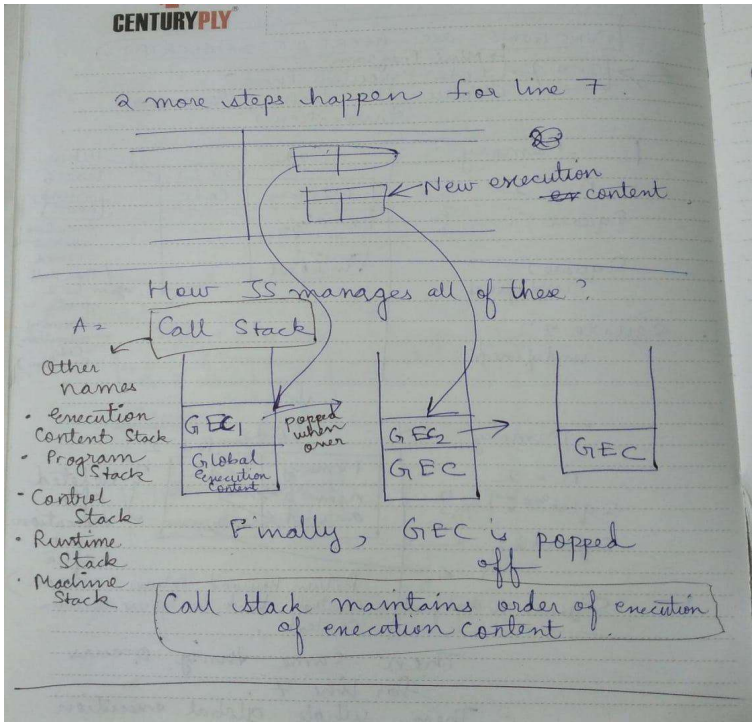
- **Execution Context Stack**
- **Program Stack**

- **Control Stack**
- **Runtime Stack**
- **Machine Stack**

HANDWRITTEN NOTES




Notes :
CENTURYPLY
FUNCTIONS are HEART OF JAVASCRIPT.
2nd → Code Execution Phase (Value Allocation & Invocation)
Memory
Code
Part 1
Memory | Code
num: undefined
ans: undefined
n : 2
Square : f -- }
Square2 : undefined
square 4 : undefined
When line 6 runs, new execution content created. (These will again be 2 parts — Memory allocation, Code Execution)
Part 2
num = 2
ans = 4
return ans
Deleted after execution over (after return)
Square2 : 4
Square 7 : undefined
Return keyword returns control back to prev execution content.
Then same thing occurs for line 7.
Then whole global execution is deleted.


CENTURYPLY
2 more steps happen for line 7.
New execution content
How JS manages all of these?
Call Stack
Other names
• Execution Context Stack
• Program Stack
• Control Stack
• Runtime Stack
• Machine Stack
GEC1
popped when over
Global Execution Context
GEC2
GEC
GEC
Finally, GEC is popped off
Call stack maintains order of execution of execution context.

Hoisting in JavaScript 🔥(variables & functions) | Namaste JavaScript Ep. 3

Youtube Link-https://youtu.be/FnJnw8uY6jo

Code

```
var x=7;
function getName(){
    console.log('Namaste JS')
}
console.log(x)
getName()
console.log(getName)
```

Output

```
7
Namaste JS
ƒ getName(){
    console.log('Namaste JS')
}
```

This was expected.

Now let's move the console logs and function calling to the top.

```
console.log(x)
getName()
console.log(getName)
var x=7;
function getName(){
    console.log('Namaste JS')
}
```

Here we are trying to access variable x and function getName even without initializing it.

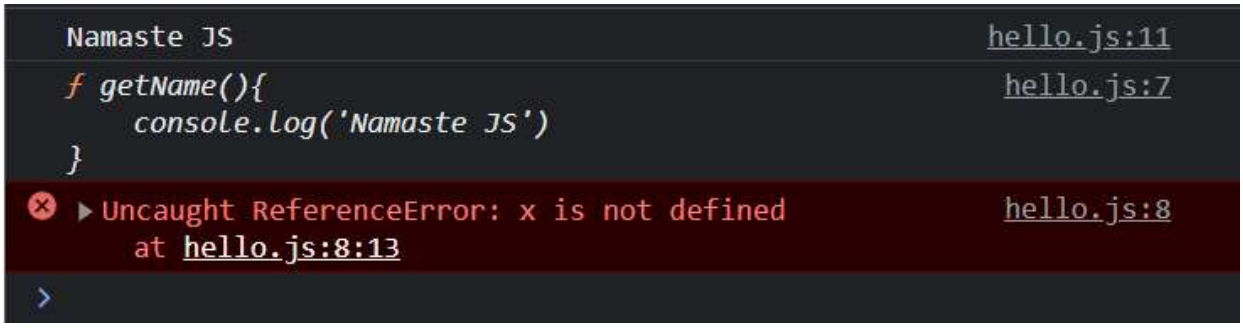
Output

We were able to access the function but not the variable x.
variable x is undefined.
Now let's remove the variable declaration and see the output.

```
getName()
console.log(getName)
console.log(x)
// var x=7;
function getName(){
    console.log('Namaste JS')
}
```

var x=7; has been commented.

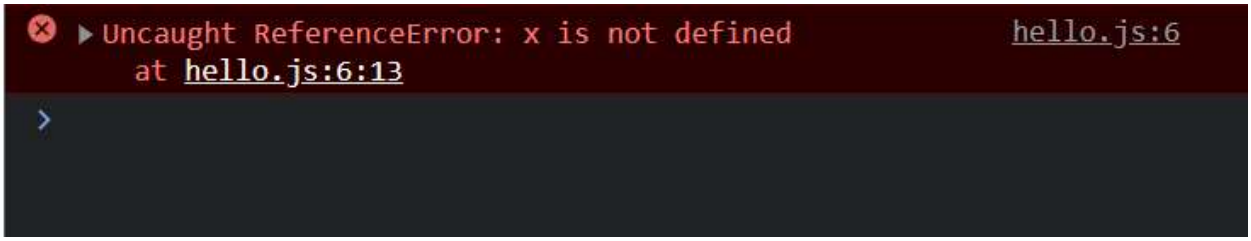
Output



Now we see Uncaught ReferenceError: x is not defined.
This is because we did not reserve memory for x as var x was commented.
SO *UNDEFINED* and *NOT DEFINED* are different things in JS.
Now for experimenting lets change the order of logs and log x first.

```
console.log(x)
getName()
console.log(getName)
// var x=7;
function getName(){
    console.log('Namaste JS')
}
```

Output



We see that getName() has not been called in above output.
This is because as soon as JS encounters an error in a line ,it stops execution then and there.
Error occurred because, later in the program(or anywhere), var x declaration was not there.
If var x declaration was there anywhere in the program ,then the rest of the code like function calling of getName would also have been executed.

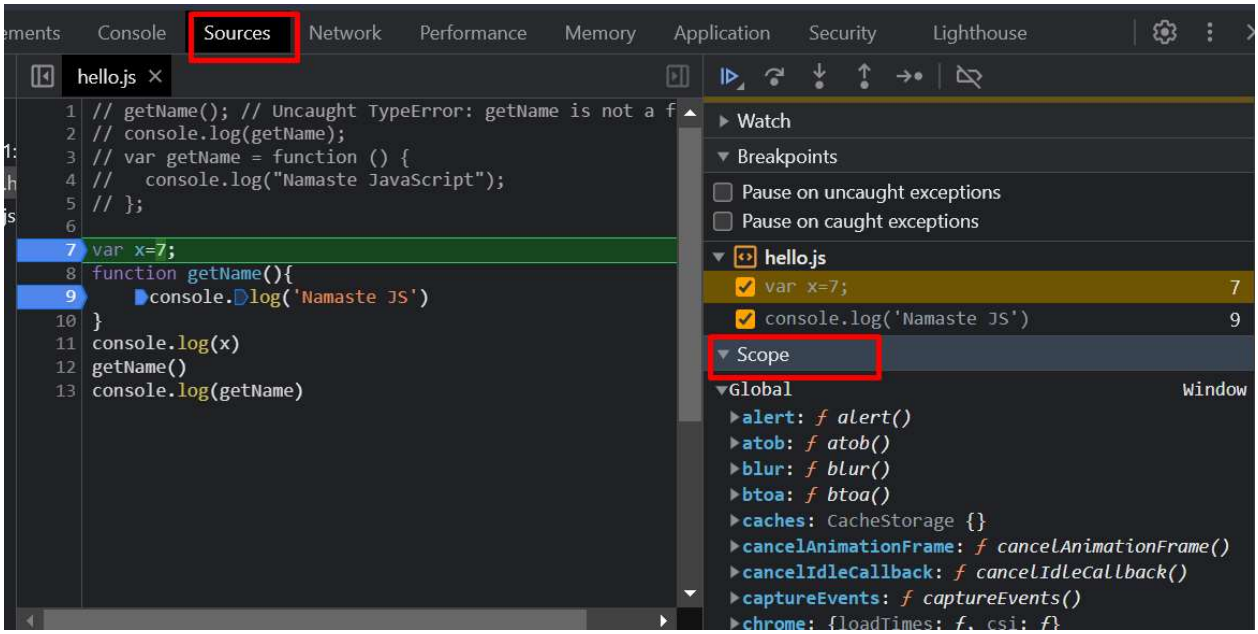
These things,where we try to log x,that is,access x, before declaration, and get undefined in the console ,happens due to hoisting.

HOISTING is phenomenon in JS by which you can access these variables and functions even before you have initialized it ,without any error.
Reason → Execution context gets created in two phases ->memory creation and code execution. So even before code execution, memory is allocated in a variable environment also called memory component. So in case of variable, its value will be initialized as undefined while in case of function the whole function code is placed in the memory component during memory allocation phase.

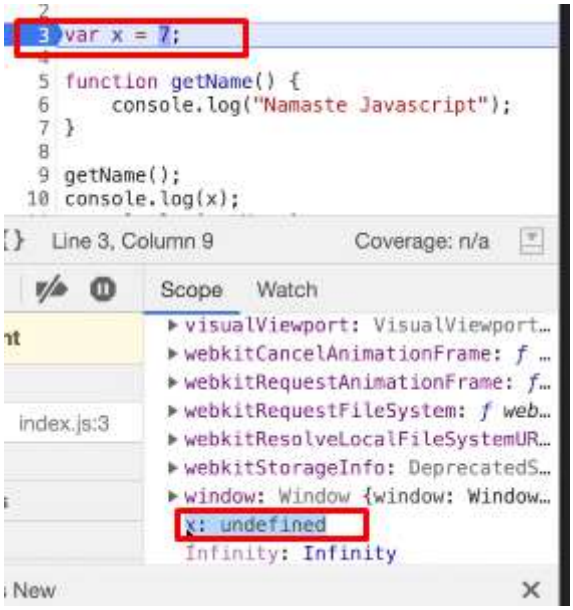
Now let's get back to the original code

```
var x=7;
function getName(){
    console.log('Namaste JS')
}
console.log(x)
getName()
console.log(getName)
```

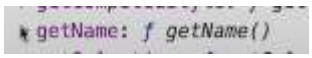
Let's place a debugger on var x=7 in sources of the console.



Inside the scope we look for the variable x.



We see x is getting value undefined in memory creation phase.
We had learnt that for functions we get the entire code for function.



So we see it as well→ the entire function is present as value.

Now let's make getName() an arrow function.

Code-

```
var x=7;
var getName=function(){
    console.log('Namaste JS')
}
console.log(x)
getName()
console.log(getName)
```

Output

```
7                                                    hello.js:5
Namaste JS                                           hello.js:3
ƒ (){                                                hello.js:7
    console.log('Namaste JS')
}
```

The above output was expected.

Now since we are working with arrow functions,let's remove variable x.

```
getName()
console.log(getName)
var getName=function(){
    console.log('Namaste JS')
}
```

Output

```
▶ Uncaught TypeError: getName is not a function      hello.js:1
    at hello.js:1:1
```

Or if we log the function and then call the function.

```
1    console.log(getName)
2    getName()
3    var getName=function(){
4        console.log('Namaste JS')
5    }
```

Output

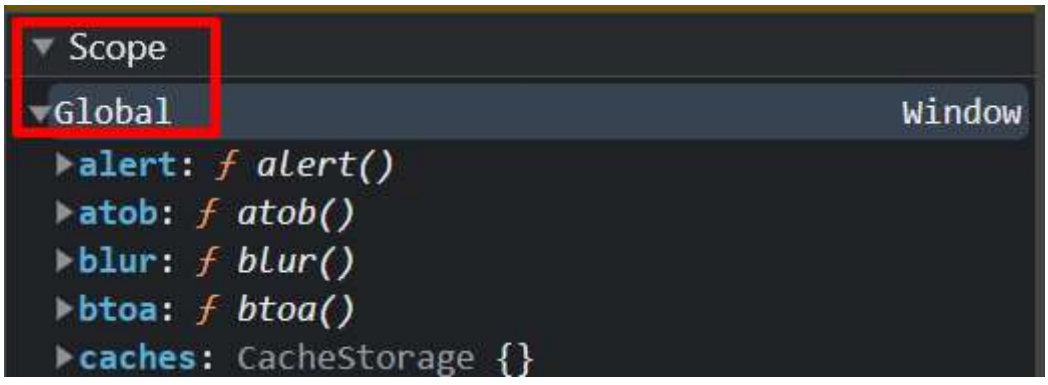
```
undefined                                                    hello.js:1
▶ Uncaught TypeError: getName is not a function              hello.js:2
    at hello.js:2:1
>
```

We see that on logging the function name, we get undefined,which means getName is behaving like a variable.
And when we do function calling on variable name getName, it obviously gives us the error message TypeError:getName is not a function.
So,when we are using arrow functions, then ,in memory creation phase, functions do not get the entire code of function as the value instead they get undefined as value. And in the code execution phase,they will get the entire function code as value(after memory creation phase). So before the execution phase ,if we are calling the function like above ,it is treated as a variable and an error is thrown. It is treated as a variable as a variable var getName is holding the function in the code.
Now to verify that arrow function gets undefined we use the following code

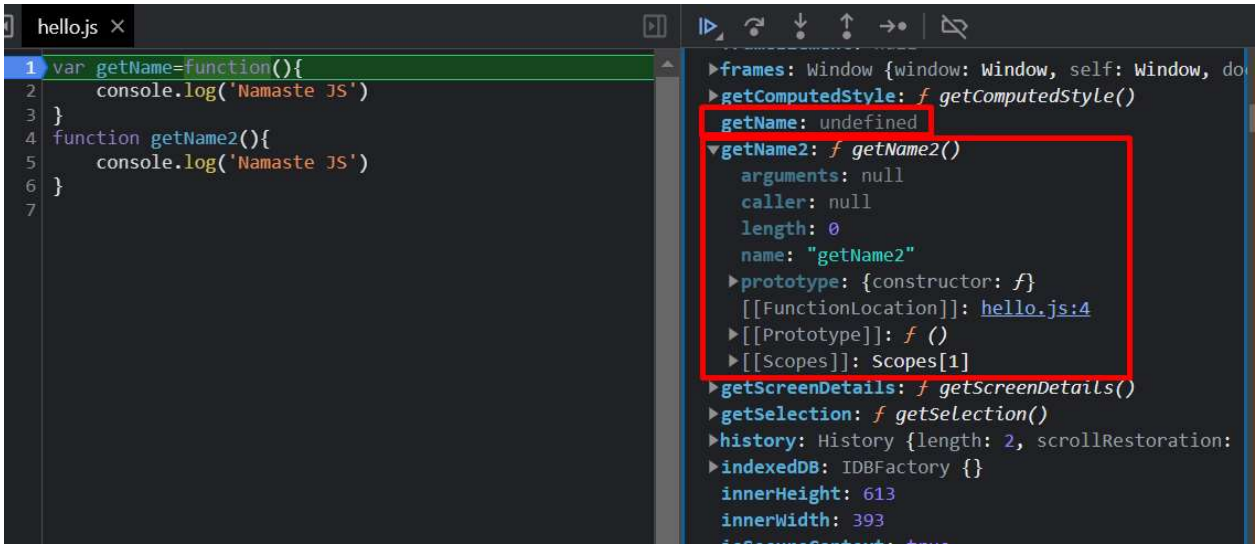
```
var getName=function(){
    console.log('Namaste JS')
}
function getName2(){
    console.log('Namaste JS')
}
```

We then apply a debugger

And then we move to **Global** inside **Scope**

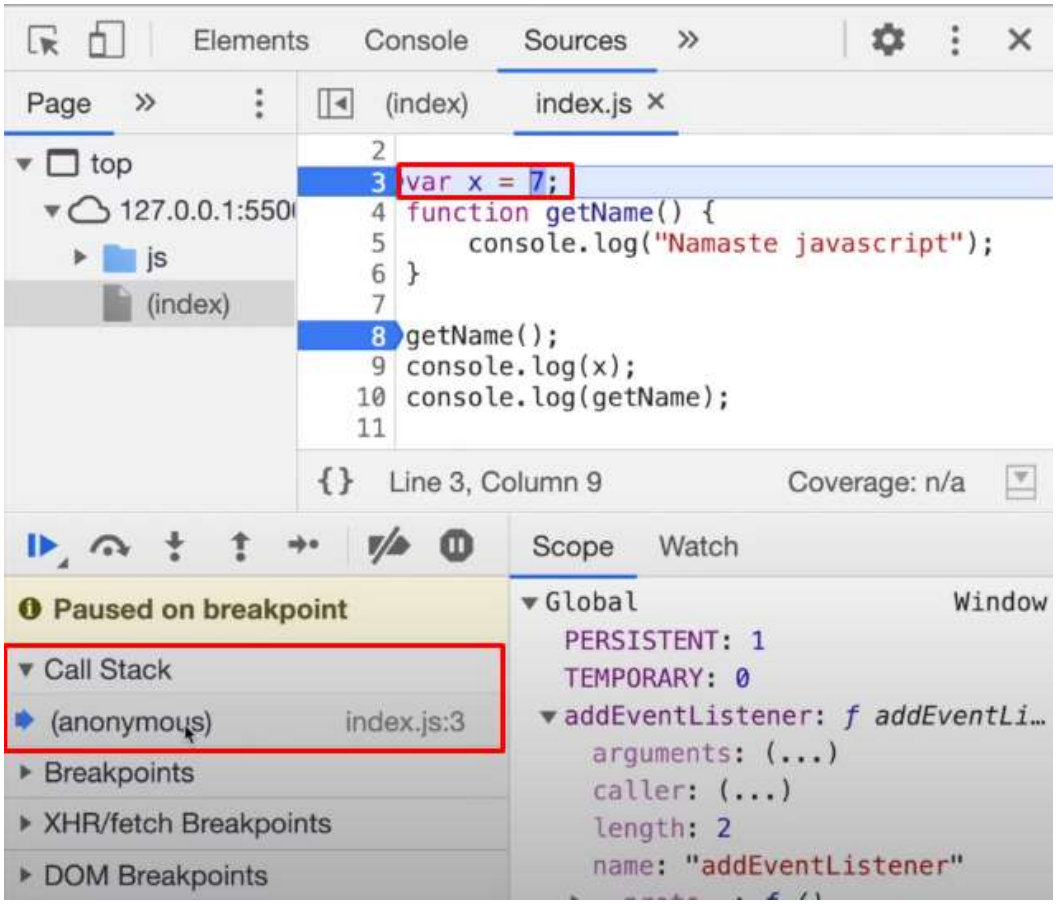


We proceed to find getName() and getName2() there.

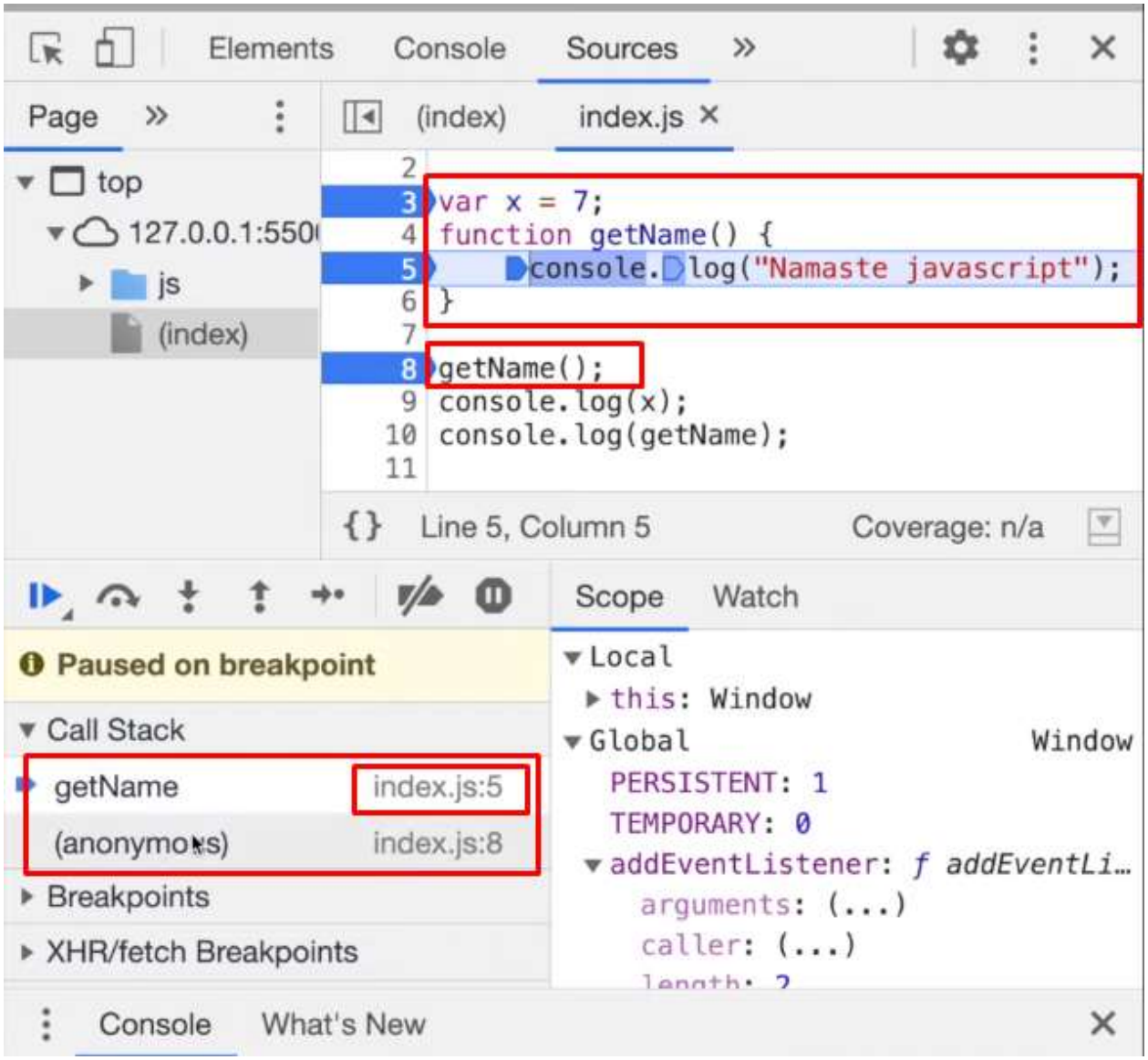


We see getName() being the arrow function gets an undefined value while getName2() gets the entire function code as value in memory creation phase.

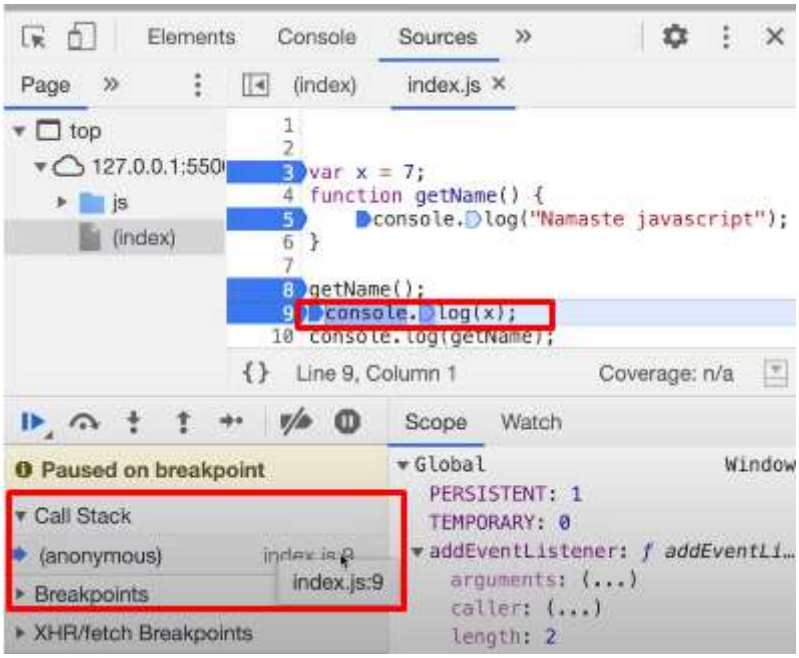
Now,let's see the global execution context in practical terms.



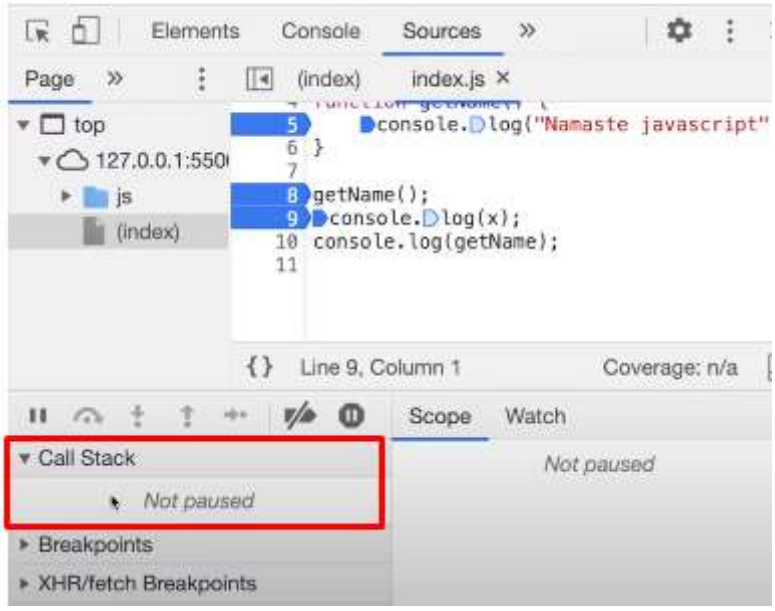
As we see control is on line 3, a global execution context is created and pushed onto the stack.
Now when function invocation happens, execution context for that function is created and put into the call stack.



So as we see the execution context of getName is put into call stack. And then it will be popped off.



As we see above, control is on line 9, hence getName execution context is popped off the call stack as function call is over.
And then finally,...

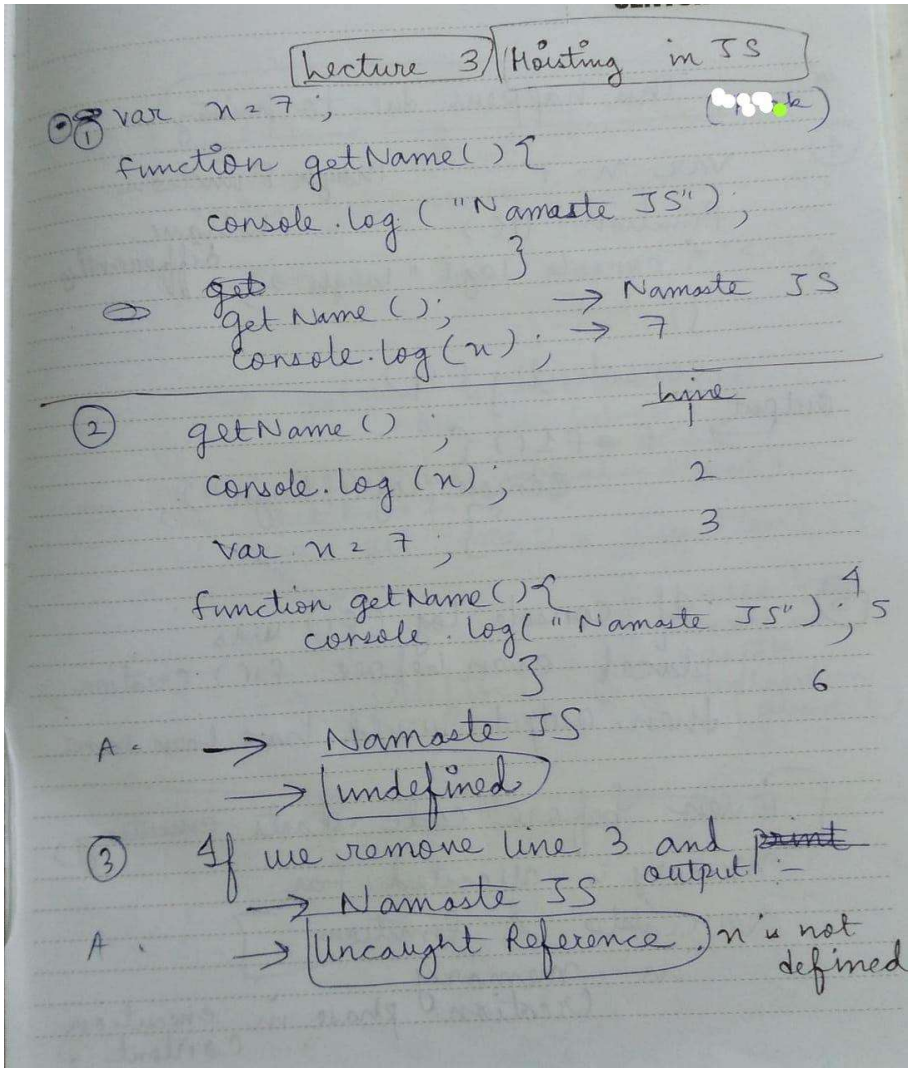


After program execution ends,global execution context is also popped off. And call stack is now empty.
Some things about hoisting from other sites→

- **Hoisting is JavaScript's default behavior of moving declarations to the top. JavaScript only hoists declarations, not initializations.**

- **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.**

HANDWRITTEN NOTES



This happens due to hoisting

④ var n = 7;
function f1()
{ console.log("laugh");
}
console.log(f1)

Though Functions behave differently

output → f f1(){
console.log("laugh");
}

⑤ If console.log(f1) was placed even before f1() creation, then output would have been same

[Even before code starts executing, memory is allocated for variables & functions in memory creation phase in execution content.]

⑥ getName();
// arrow function
var getName = () => {
console.log("Namaste JS");
}

→ getName is ~~not~~ undefined
(now it behaves like variable)

(Not defined and undefined are different).

⑦ getName2();
var getName2 = function(){
}

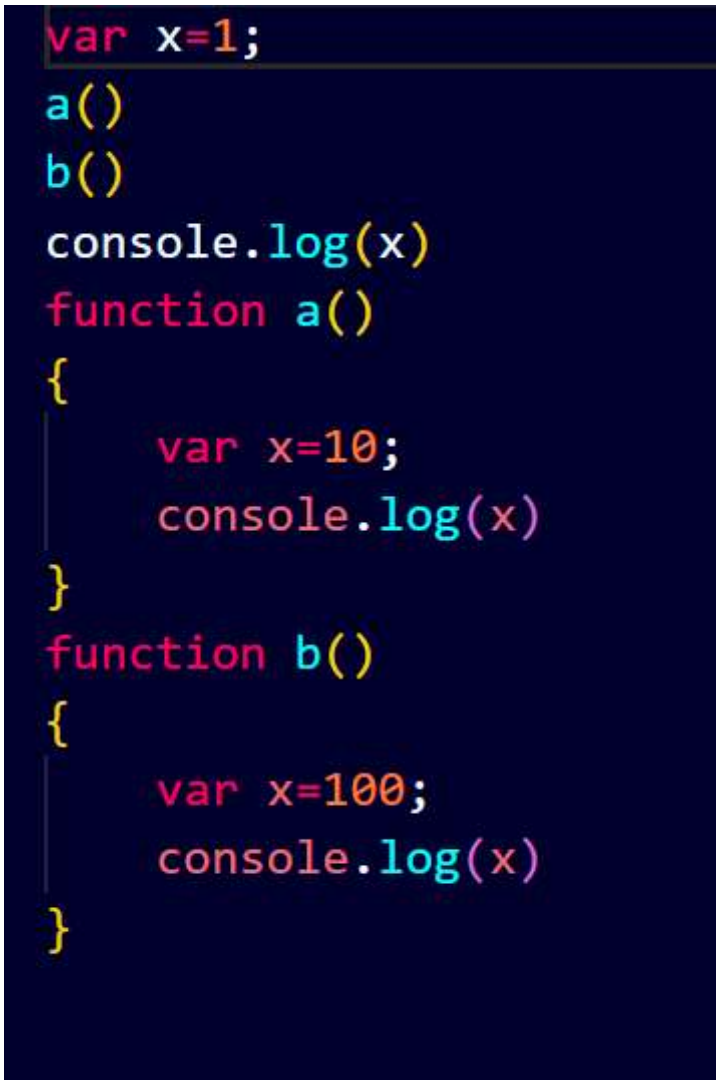
→ undefined

(video has more in depth explanation about it)

How functions work in JS ❤️ & Variable Environment | Namaste JavaScript Ep. 4

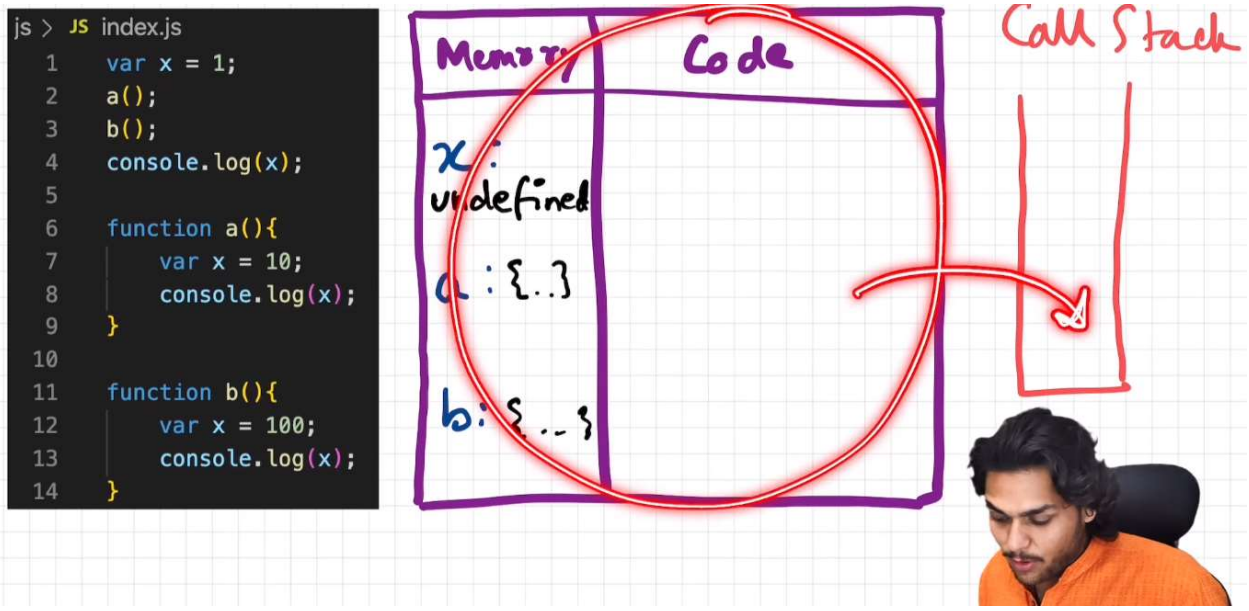
Today we will learn about function invocation and the variable environment.

For this code

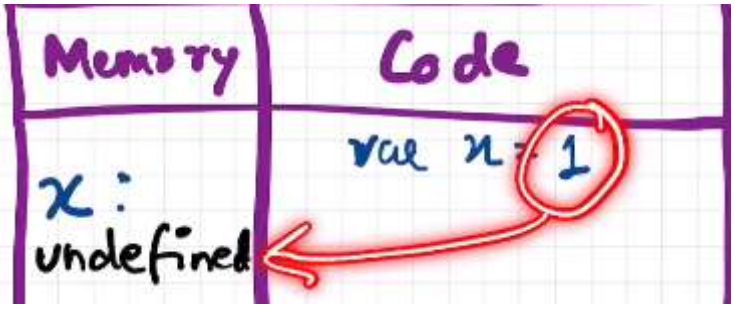


```
var x=1;
a()
b()
console.log(x)
function a()
{
    var x=10;
    console.log(x)
}
function b()
{
    var x=100;
    console.log(x)
}
```

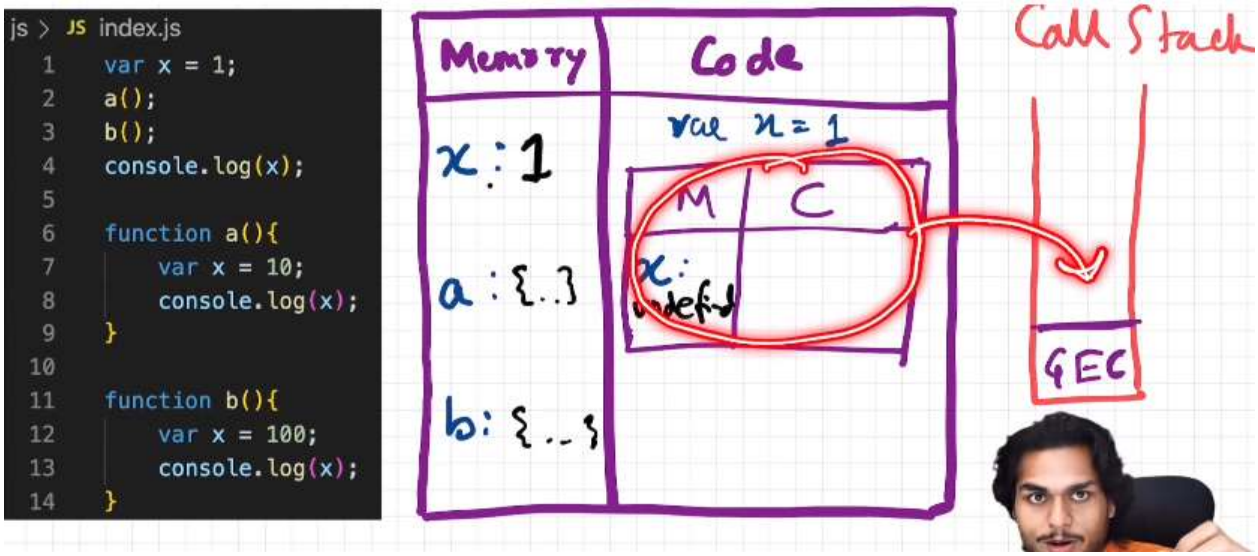
Lets see what happens in the global execution context.



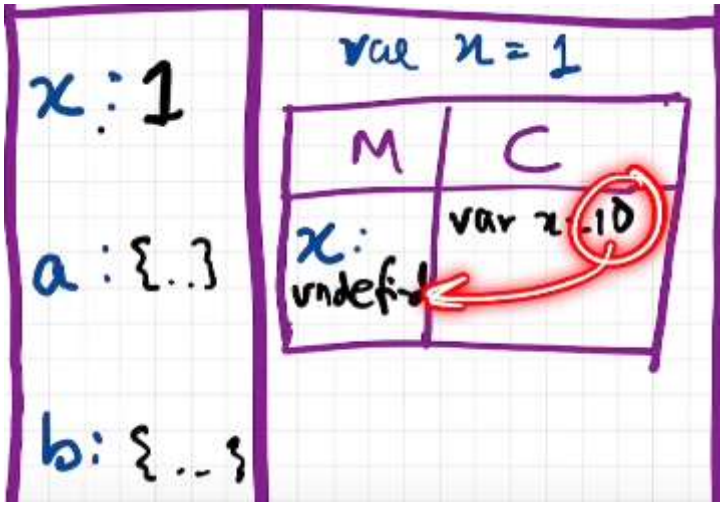
Memory creation phase takes place where variables get undefined value and entire function body is put beside function name as value. And in call stack, global execution context is pushed.



undefined will be replaced by 1 in value in x because of line 1 in code.

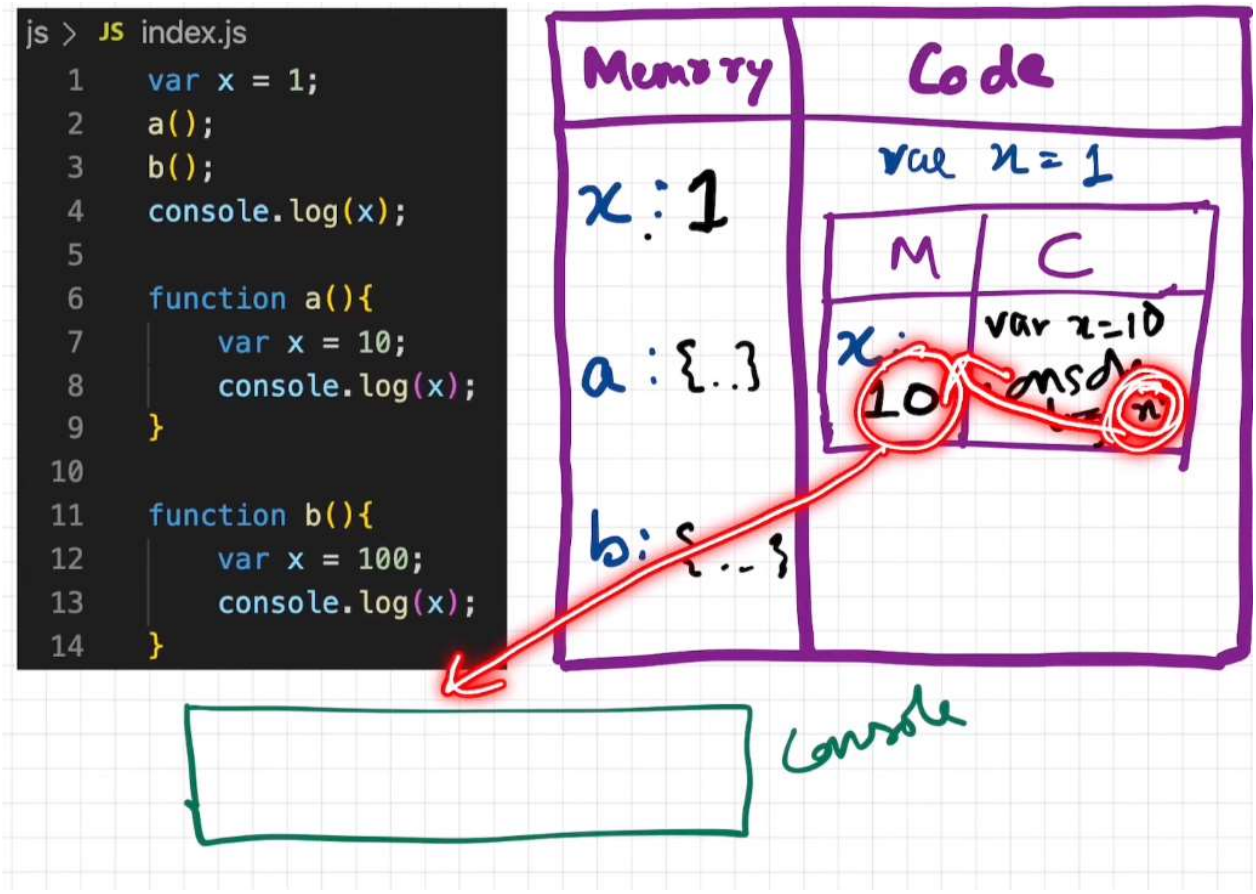


After function invocation, execution context is created and pushed into stack for function a().

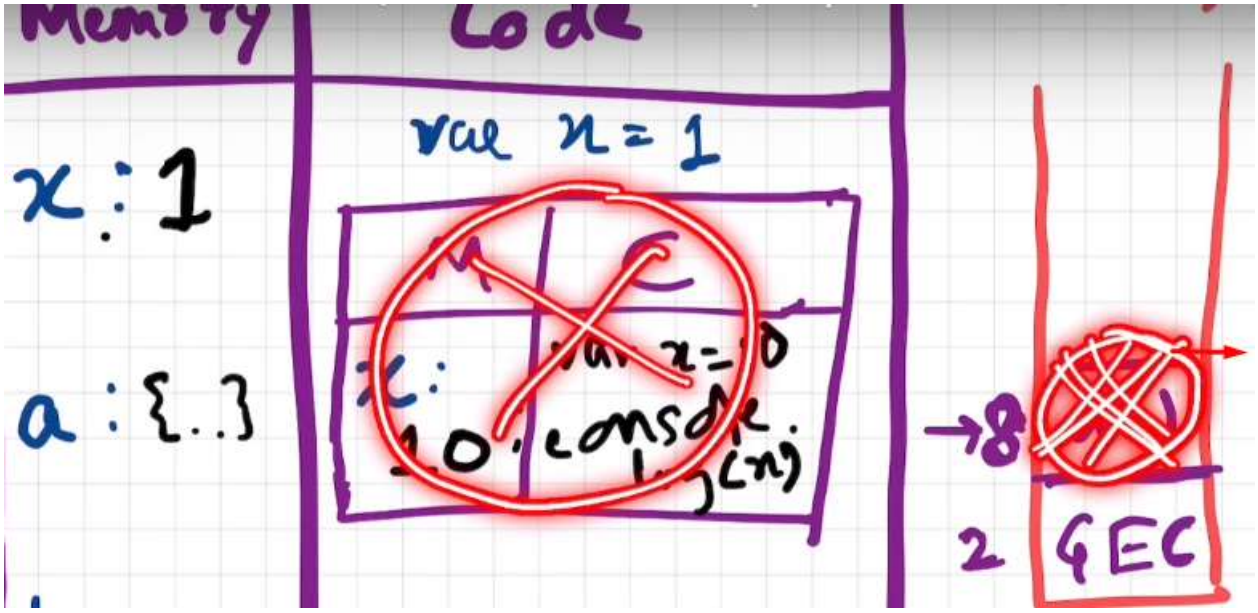


So the execution context also goes through memory creation and code execution phase. Inside it ,a separate copy of x was created and assigned value 10.

Now when we log x inside a(), JS will search for x in its local memory.



console will now have 10 printed.

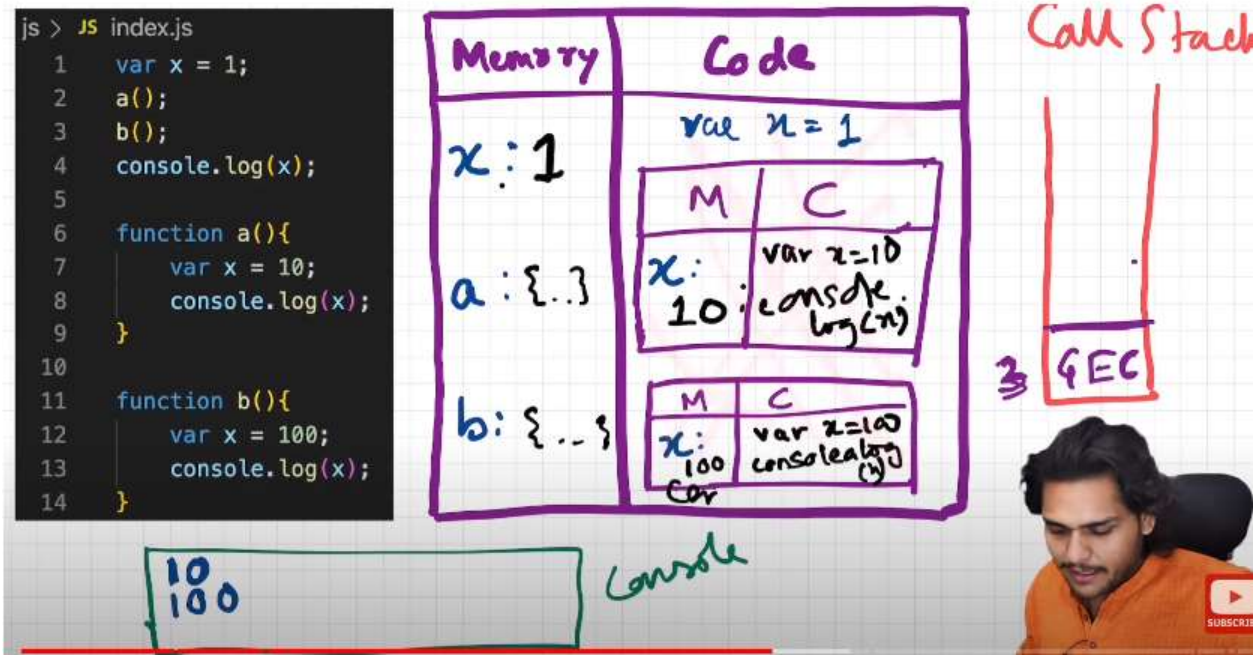


execution context of function a is popped off.

Control goes back to global execution context.

Again same thing happens.

- **execution context of b created and pushed into stack**
- **execution context of b has its own memory creation phase where x gets undefined**
- **in code execution phase it gets value 100.**
- **10 logged to console as JS find value of x in local execution context. Then execution context of b is popped off stack.**

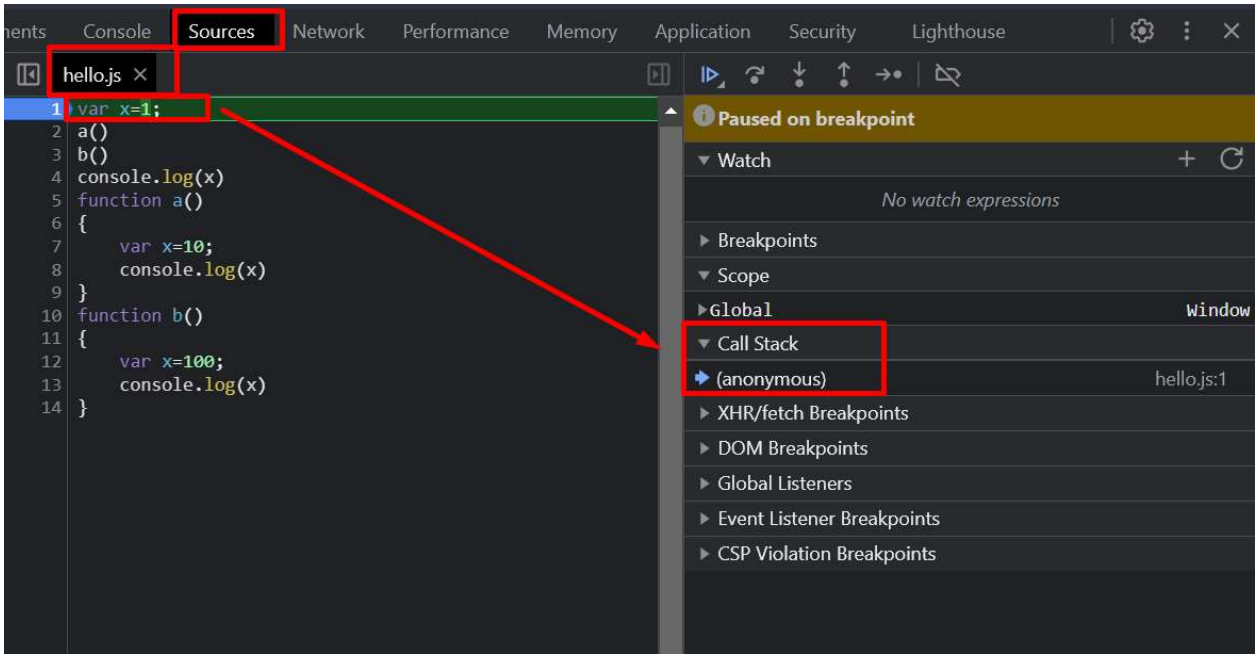


- **And then finally line 4 executes and 1 is printed.**
- **Since there is nothing more to be executed, global execution context is popped out.**

LETS SEE THIS INSIDE BROWSER ALSO.

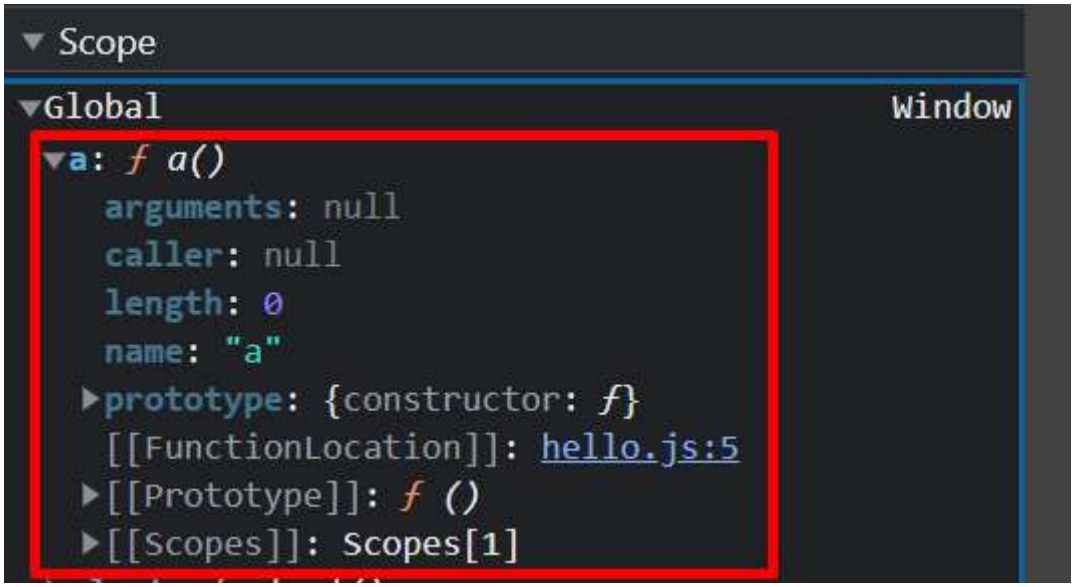
STEP 1

Global execution context created and pushed onto stack. We placed a debugger in line 1.



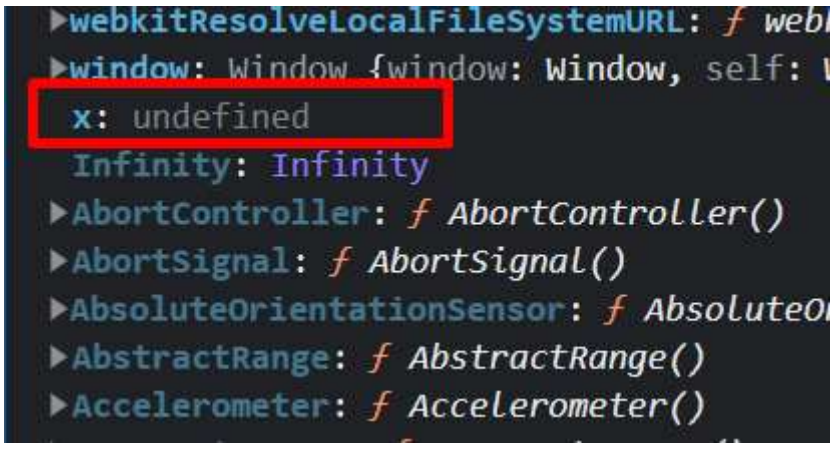
Lets find function a,b and variable x

a() in memory creation phase below



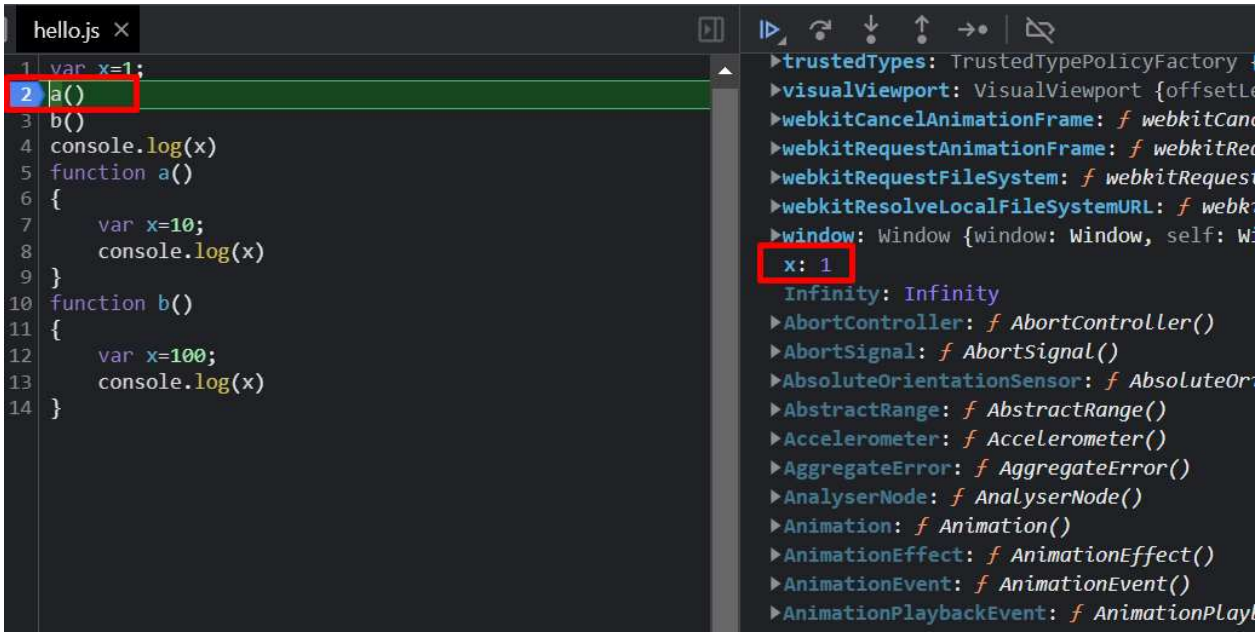
b() in memory creation phase

x as expected gets value undefined

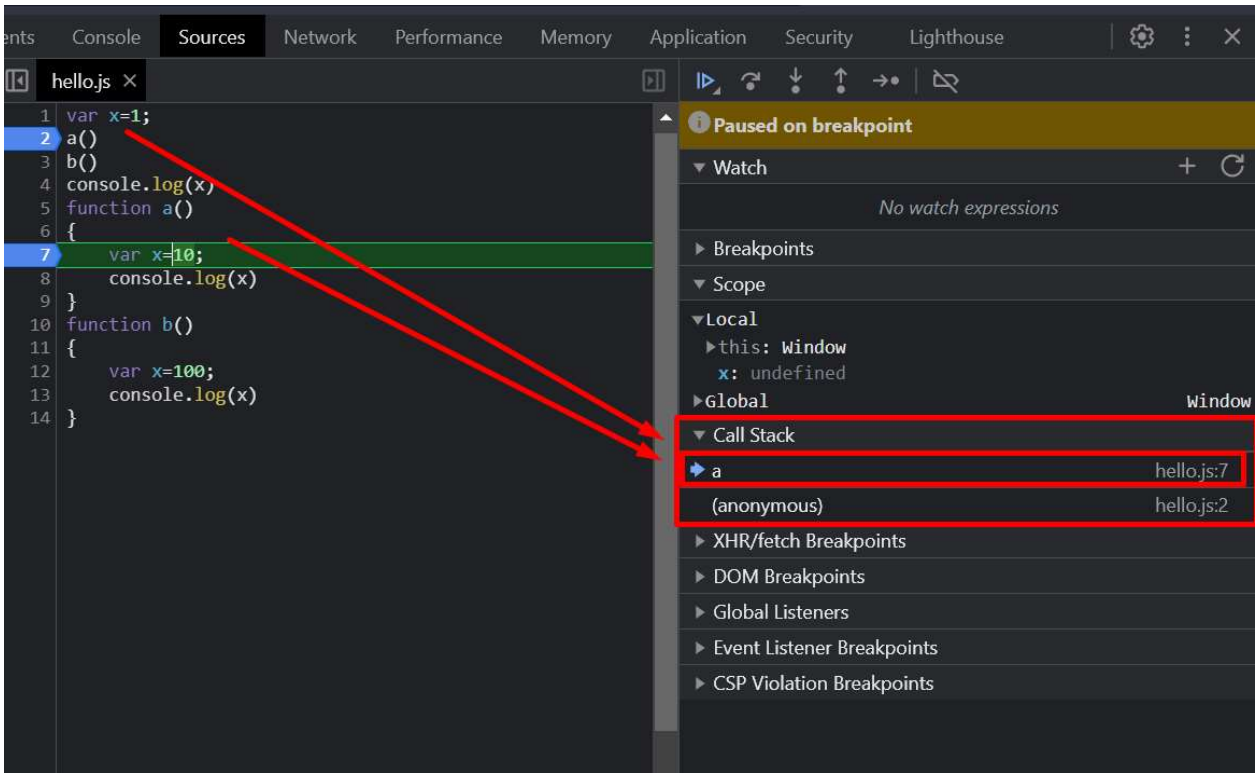


STEP 2

Now we place debugger on 2nd line



We see x has been replaced by 1.Now as soon as function a() is invoked ,its execution context will be put inside call stack.



Now if we see below, global execution context has its own memory space,

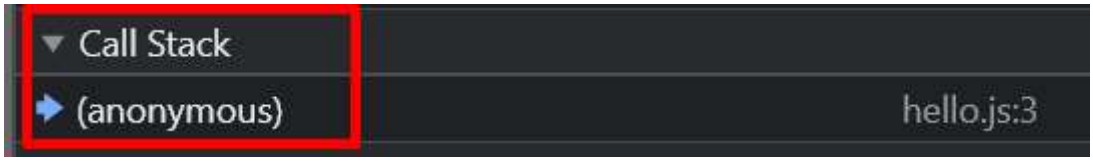
And if we click on the execution context of a, it has its own memory space.

Now if you see carefully in the above pic there is Local and Global memory.

In Local memory, x is undefined as local execution context of a has its own memory creation phase where local variables get allocated value of undefined.

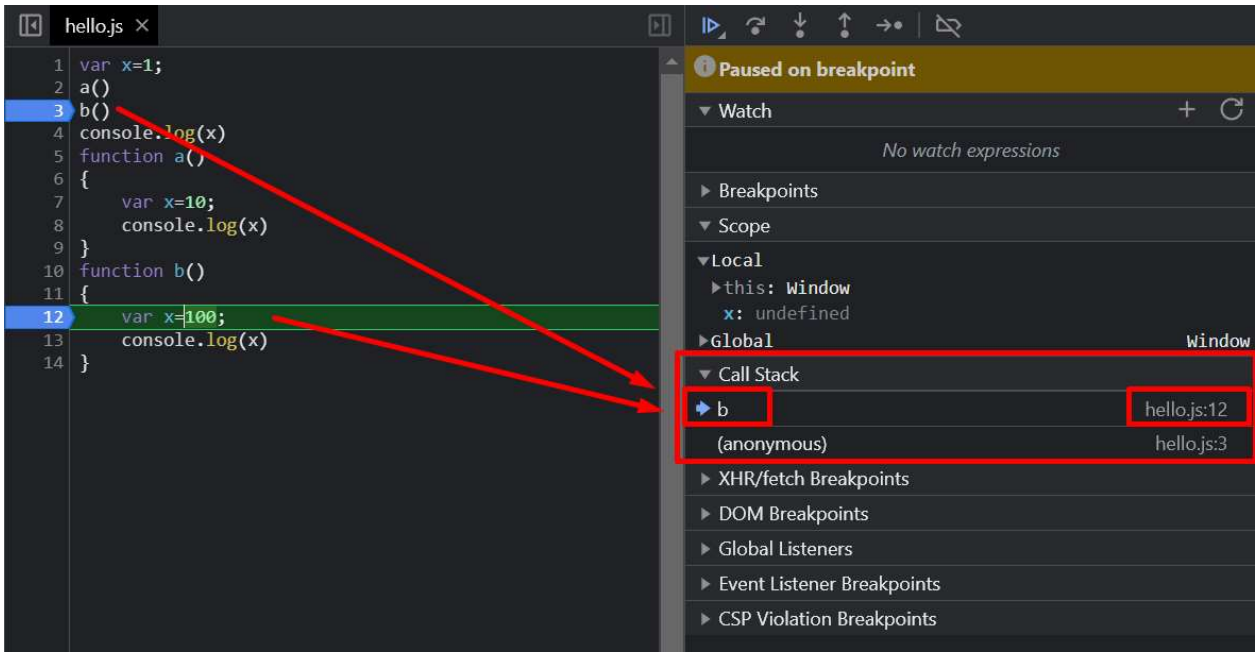
Now x gets value of 10 as we move ahead in code execution phase.

Now x will be logged onto console and control will go back to b().

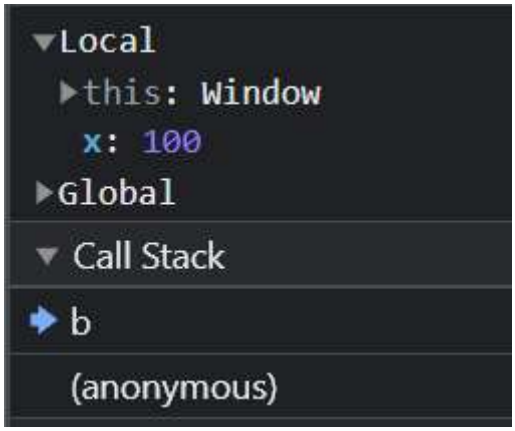


Execution context of a popped off from call stack.

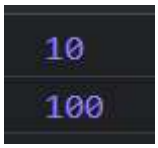
Control moves back to global execution context.



Since b() has been invoked, execution context for b created and pushed onto the call stack and x gets value of undefined in memory creation phase.And the process repeats.

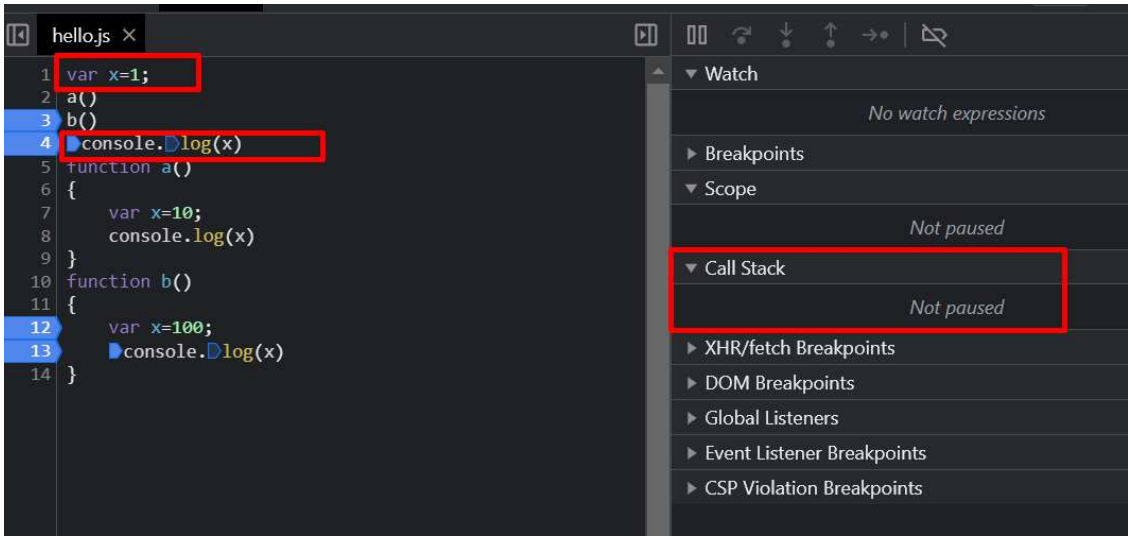


x gets value 100.



100 is logged onto console.

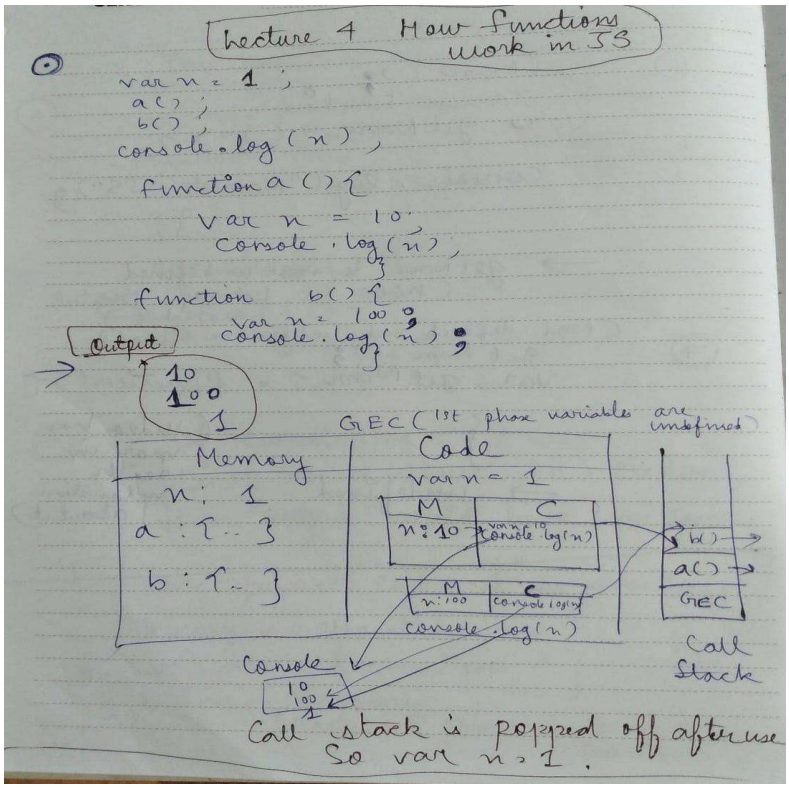
b is popped off from the call stack after its execution.



Finally call stack is empty after program ends by logging x on line 4.

```
10
100
1
```

Handwritten Notes



SUMMARY

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(EC) gets created having its own phases and is pushed into the call stack.

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

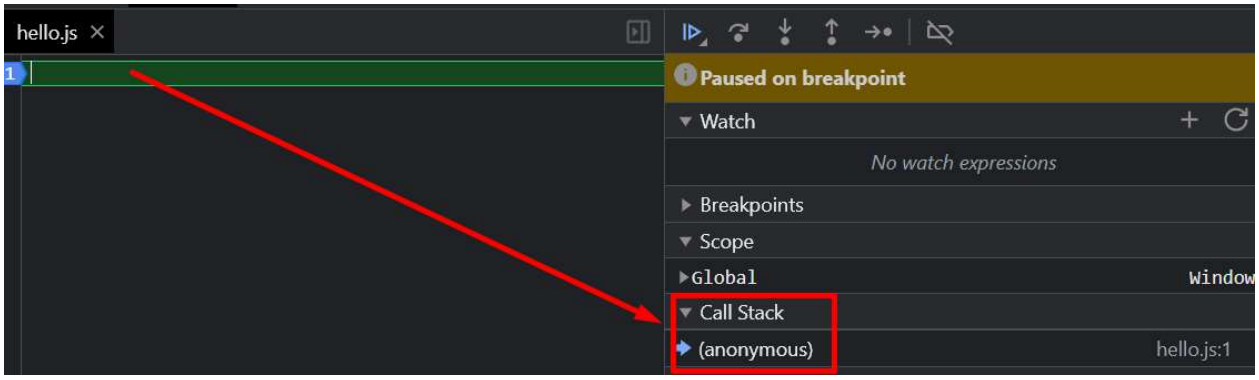
6. When the program ends, even the global EC is pulled out of the call stack.

SHORTEST JS Program 🔥window & this keyword | Namaste JavaScript Ep. 5

Youtube Link-https://youtu.be/QCRpVw2KXf8

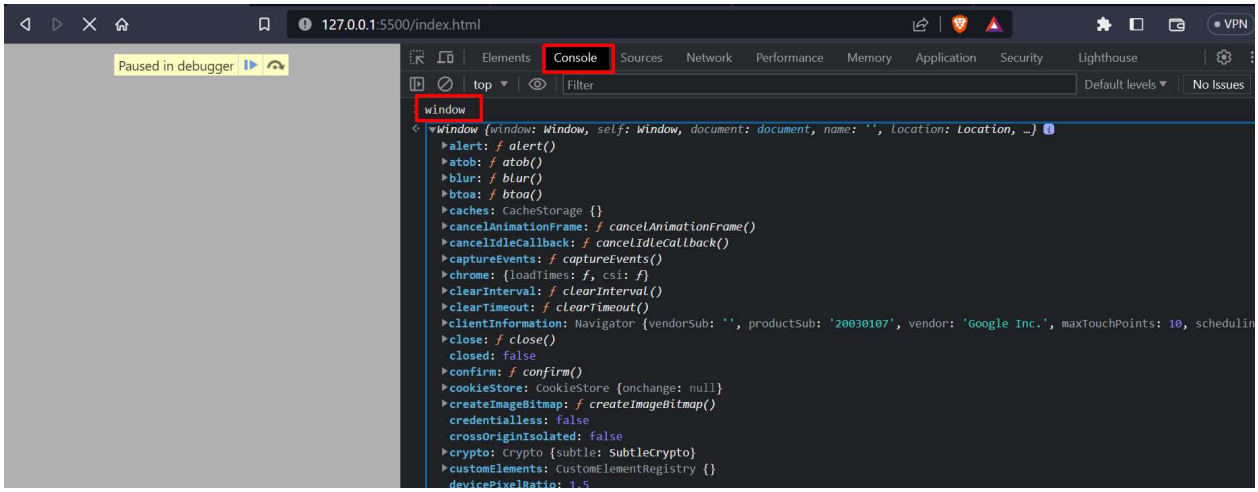
Shortest JS program is an empty file.

We put a debugger on an empty line.



And we see global execution context is created and pushed onto the call stack.

It still sets up the memory space. The JS engine creates something known as window.



- **window is like a big object with lots of functions, methods.**
- **These functions and variables are created by the javascript engine into the global space.**
- **The JS engine also creates this. At the global level this points to window objects.**
- **Window is a global object created along with the global execution context.**
- **JS is running on other devices like server apart from browsers as well. The devices should have their JS engine . So these JS engines always create a global object even if file is empty.**

```
> window
< ▶Window {window: Window, self: Window, document: document, name: '', location: Location, …}
> this==window
< true
> this===window
< true
```

What is global space being talked about?

Any code not inside a function is global space. Any variable we write in the global space gets attached to the global object window.

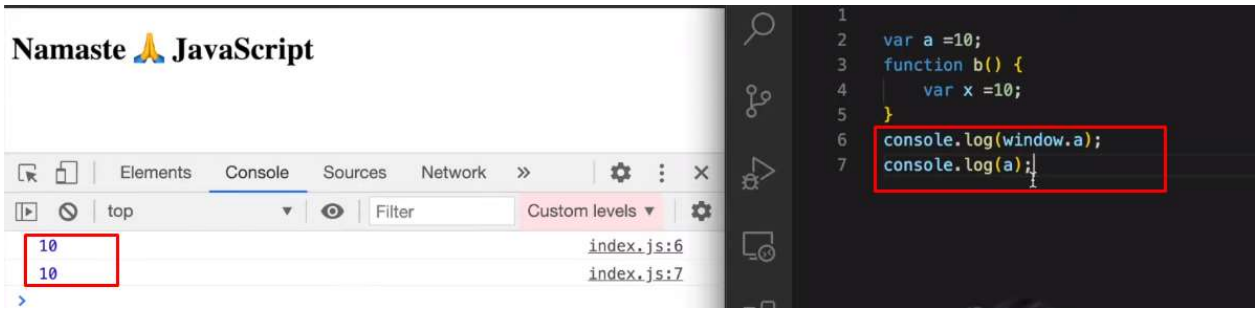
Example

```
var a = 1;

function c() {
  var d = 9;
}
```

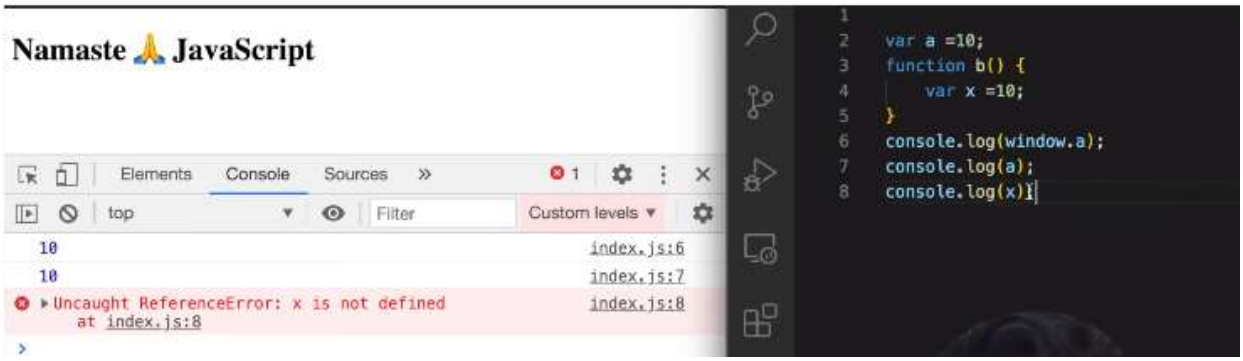
In console if we see window

```
> window
< ▾Window {window: Window, self: Window, document: document, name: '', location: Location, …}
    a: undefined
   ▸alert: ƒ alert()
   ▸atob: ƒ atob()
    b: undefined
   ▸blur: ƒ blur()
   ▸btoa: ƒ btoa()
   ▸c: ƒ c()
   ▸caches: CacheStorage {}
   ▸cancelAnimationFrame: ƒ cancelAnimationFrame()
   ▸cancelIdleCallback: ƒ cancelIdleCallback()
```

variable a and function c were in global space so we see them inside window. var d was there inside function c, hence we can't see it inside the global object window as it is not inside global space.

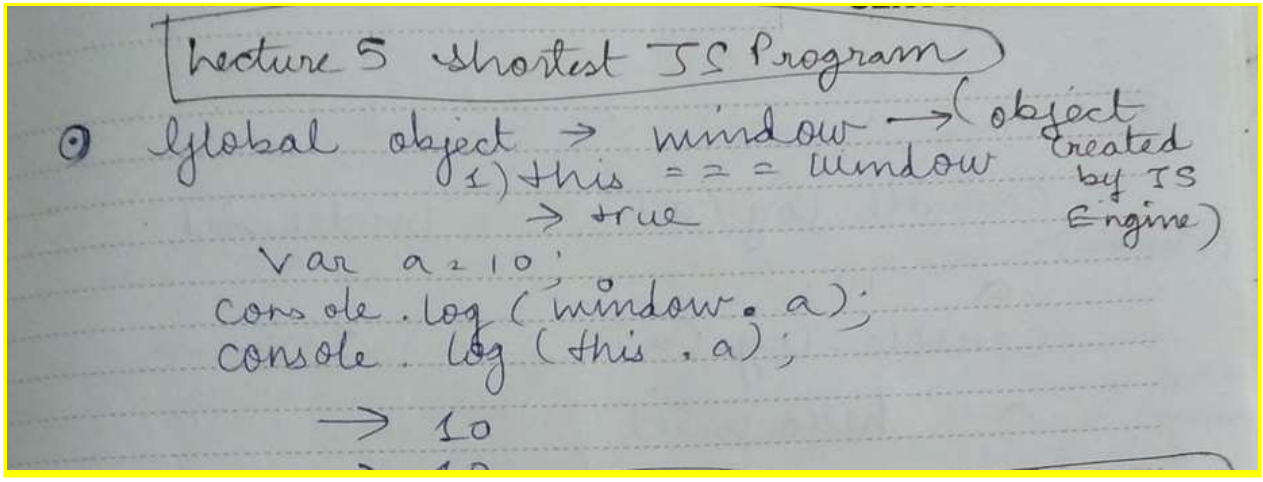


If we don't write anything before variable a while logging it ,assumes it is present in the window object.



Here it assumes x is present in window object and searches for x in the window but does not find it there and hence logs Uncaught ReferenceError: x is not defined.

Handwritten Notes



CONCLUSION -

1. Shortest Program in JS: Empty file. Still, browsers make global EC and global space along with window object.

2. Global Space: Anything that is not in a function, is in the global space.

3. Variables present in a global space can be accessed by a "window" object. (like window.a)

4. In global space, (this === window) object.

5.
```
var x = 10;

console.log(x); // 10

console.log(this.x); // 10
```

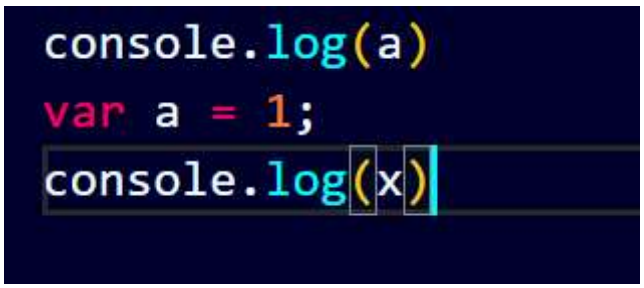
```
console.log(window.x); // 10
```

---

undefined vs not defined in JS 🤔 | Namaste JavaScript Ep. 6

Youtube Link-https://youtu.be/B7iF6G3EyIk

undefined is like a placeholder for a variable until the variable is assigned a value.



Output

We are trying to access before its declaration hence we get undefined due to concept of hoisting which allows us to access value of variables before it is declared due to

- **memory creation phase where undefined is assigned as a placeholder to variables**

- **moving of declarations to the top of code**

x is not declared anywhere hence we get not defined as it not allocated memory in memory allocation phase.

not defined means not being allocated memory.

Now for following code

```
console.log(a)
var a = 1;
console.log(a)
```

We get output

```
undefined
1
```

Once the flow of control reaches the 2nd line in above pic, a is assigned a value 1,thus removing the placeholder undefined previously there.

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

Now above if we never initialize a ,it will always have placeholder undefined.

```
undefined
```

For code **Output**

```
var a ;
console.log(a)
if(a===undefined)
    console.log('I am undefined')
else
    console.log('I am not undefined')
```

```
undefined
I am undefined
```

For code **Output**

```
var a ;
console.log(a)
a=7;
if(a===undefined)
    console.log('I am undefined')
else
    console.log('I am not undefined')
```

```
undefined
I am not undefined
```

JS is a loosely typed language(weakly typed). If I put a string in a variable,later we can assign an integer to it as well. So JS is flexible.

Code **Output below**

```
var a ;
console.log(a)
a=7;
console.log(a)
a="Akash"
console.log(a)
```

```
undefined
7
Akash
```

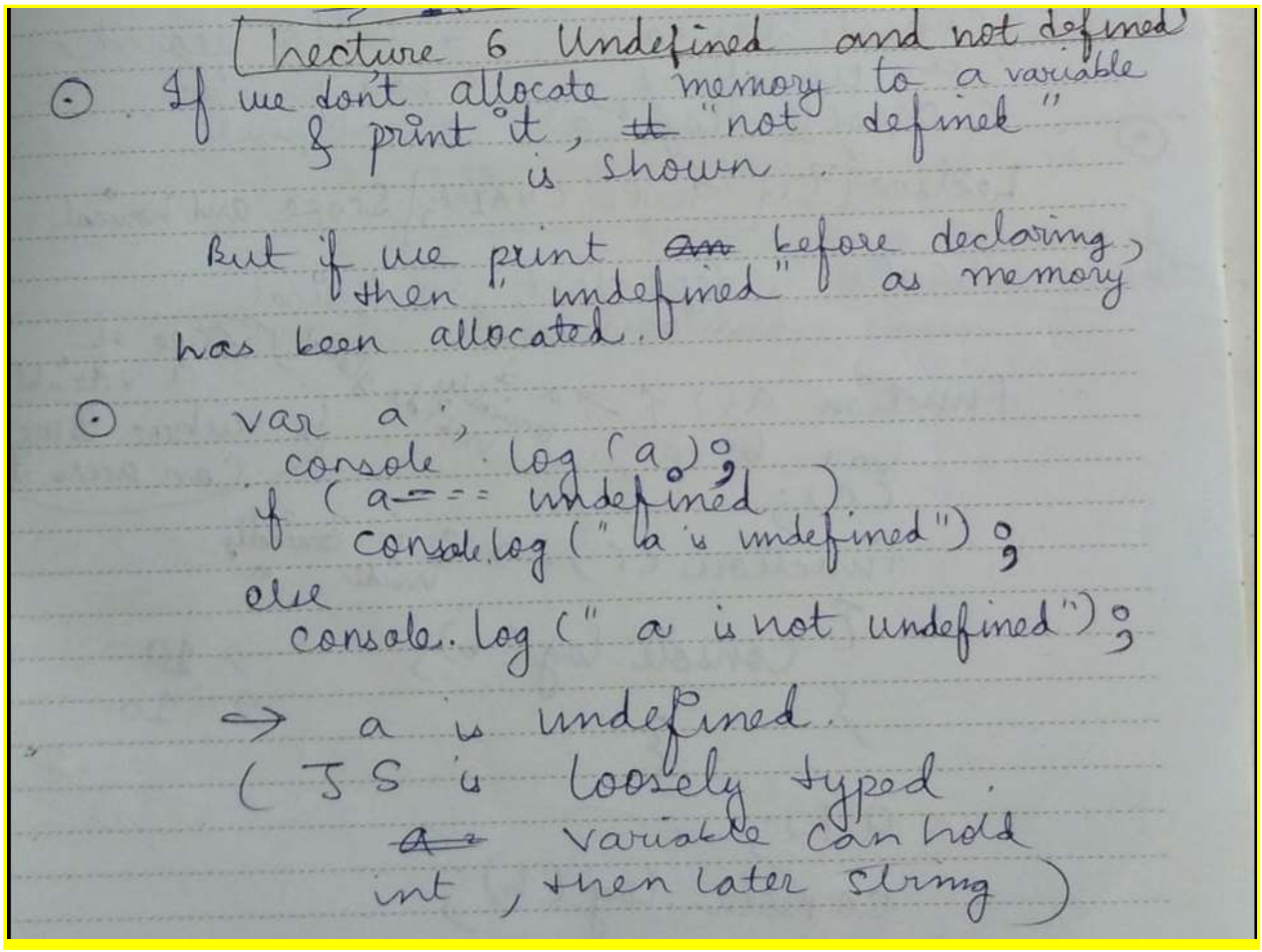
Never do this

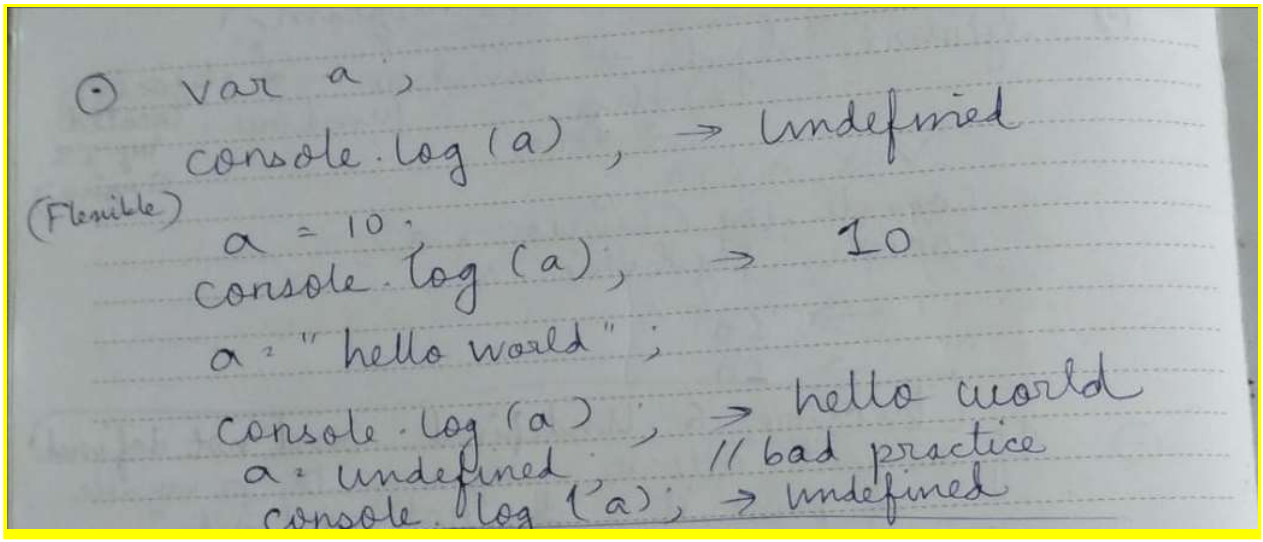
```
a=undefined
```

It syntactically correct but bad thing to do.

undefined was used to know whether that variable was assigned anything or not. So making a variable undefined after it was assigned a value does not make sense so better avoid it.

Handwritten Notes





SUMMARY **1. Undefined is like a placeholder till a variable is not assigned a value.**

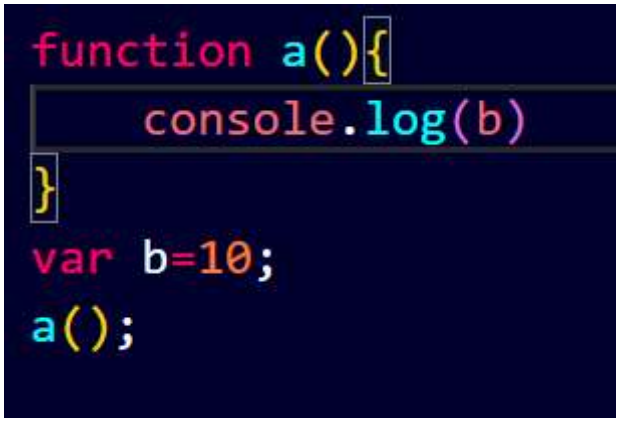
2. undefined !== not defined

3. JS- weakly typed language since it doesn't depend on data type declarations.

4.When a variable is declared but not assigned a value, its current value is undefined. But when the variable itself is not declared but called in code, then it is not defined.

The Scope Chain, 🔥Scope & Lexical Environment | Namaste JavaScript Ep. 7 Youtube Link-https://youtu.be/uH-tVP8MUs8

Scope in javascript is directly related to the lexical environment.



Question here is can we access variable b inside function a ?

JS will first try to find if b exists in the local memory space of function a or not,which means JS will try to find b inside the local memory of a's execution context.

Output

which means b gets a value of 10,so yes we can access b inside a.

Let's make the code complicated and log b inside function c which is inside function a.

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

Output

```
10
```

But if we do vice versa in code-

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

Can we access b ?

Output

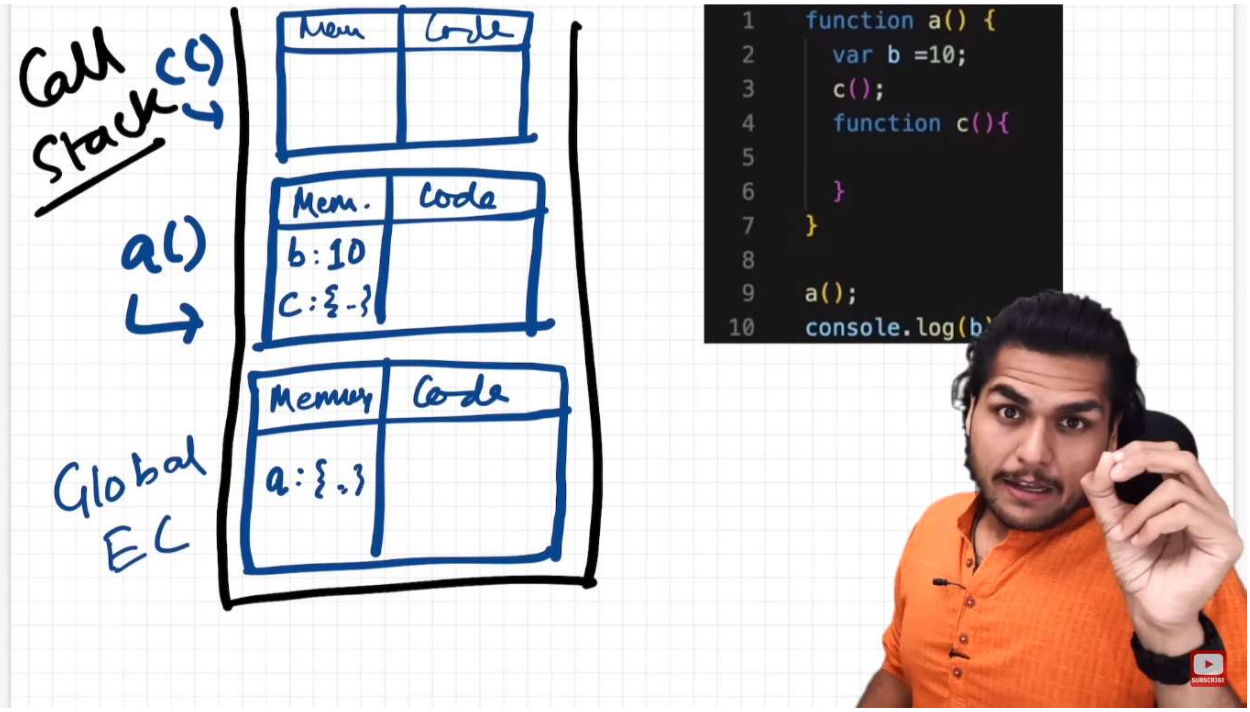
```
▶ Uncaught ReferenceError: b is not defined
    at hello.js:8:13
>
```

We cannot access b.

So here scope comes into picture.Scope means where you can access a specific variable or a function in our code.We can access a variable only if it is inside our scope.

So when someone says is b inside the scope of c ? That means can one access b inside c. Scope is directly dependent on the lexical environment.

Let's look at the below code



Whenever execution context is created , a lexical environment is also created.

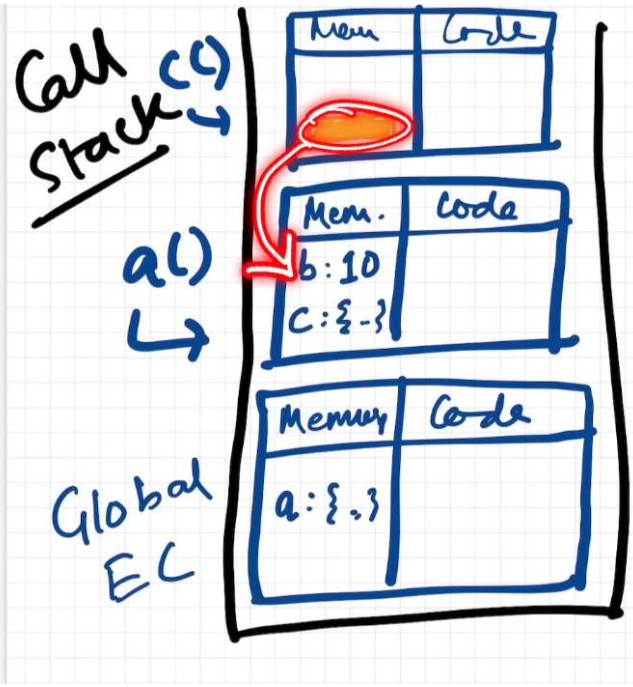
A reference to the lexical environment of the parent is also got.

Lexical environment=Local memory+reference to Lexical environment of its parent.

Lexical ,as a term,means hierarchy or in a sequence or in order.

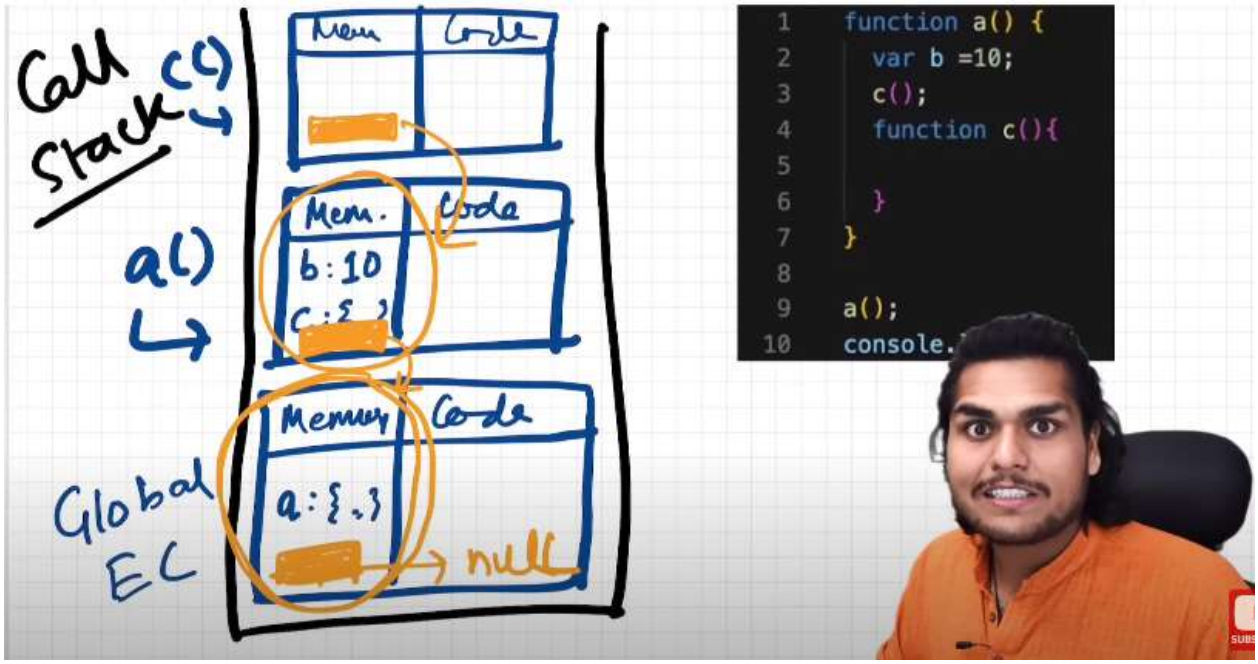
In terms of code,we can say the c function is lexically sitting inside a function and a is lexically inside global scope.

Lexical parent of c is a.




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

Lexical environment of a is its own memory space+lexical environment of a's parent.



At a global level,reference to outer environment points to null.

Lets see the following code

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

When we are logging b, JS first searches for b inside local memory of function c.It could not find b inside c.

JS now goes to the lexical parent of c which is a with the help of reference.Now it searches for b in the lexical environment of a and it prints the value of a in the console as it finds b there.

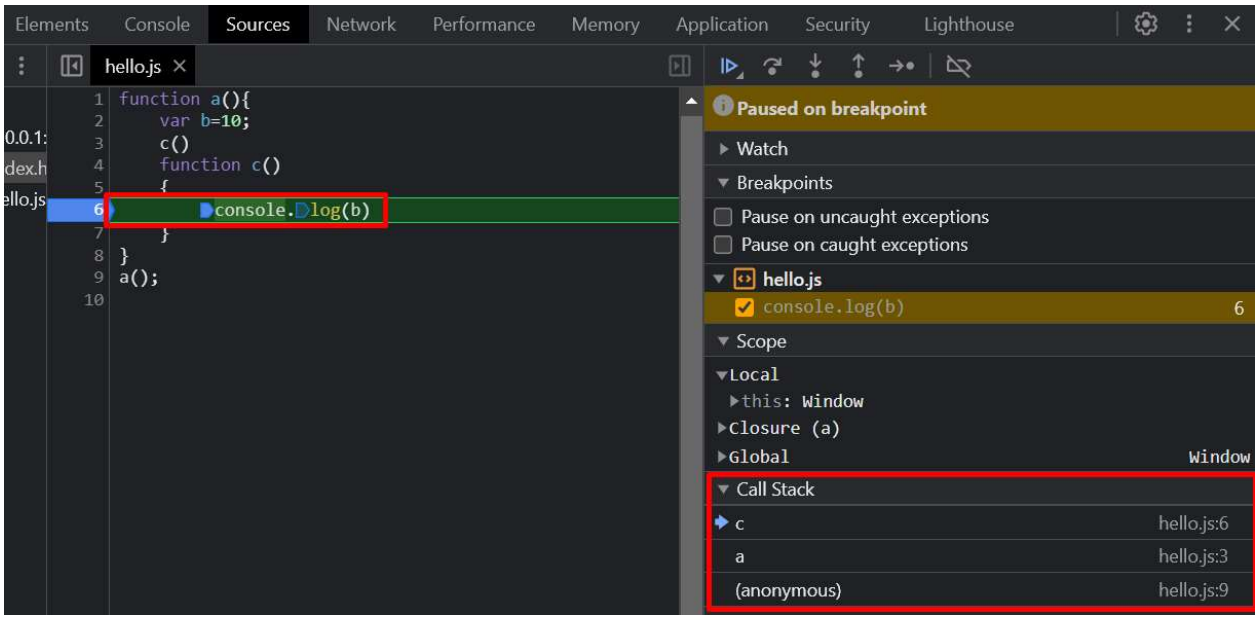
But LET'S SAY if b was not there inside function a as well,then JS would go to lexical environment of a's parent,that is, in the global execution context's lexical environment.

And now lets say b is not there inside the environment of global execution context as well,then JS goes to the reference of GEC and finds out it is null as GEC has no parent. So then we get the error b is not defined as it was not found anywhere.

This way of finding or mechanism is known as scope chain, the whole chain of all lexical environments. It defines whether a function or variable is present inside a scope or not.So if a variable is not found or shows *not defined error* we can say it was not there in the scope chain.

To summarize the above points in terms of execution context:
call_stack = [GEC, a(), c()]
Now let's also assign the memory sections of each execution context in call_stack.
c() = [[lexical environment pointer pointing to a()]]
a() = [b:10, c:{}, [lexical environment pointer pointing to GEC]]
GEC = [a:{}, [lexical_environment pointer pointing to null]]

Let's go to our console.



The call stack has 3 contexts→execution context for c,a and global execution context.

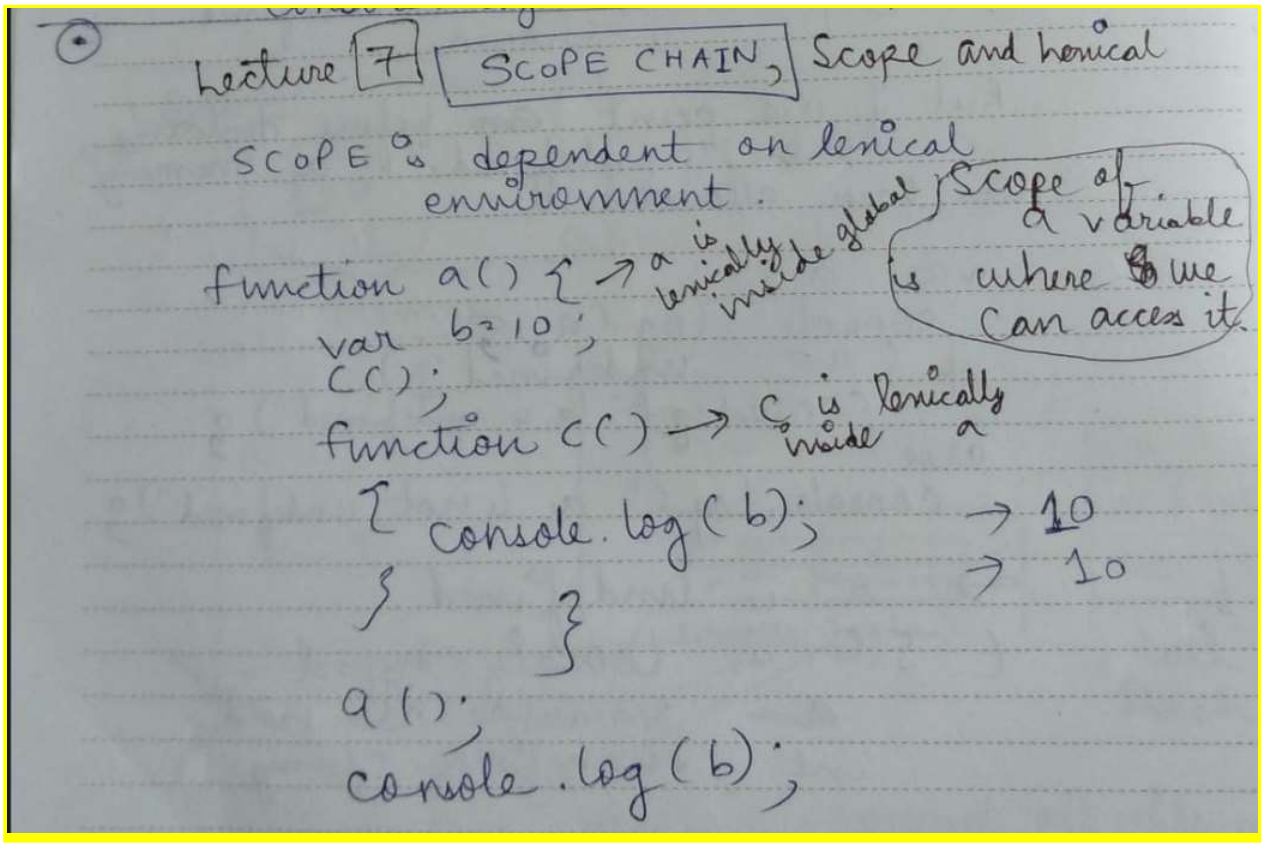
Portion marked in red is the lexical environment of global execution context, we got to see after clicking on (anonymous) in the Call stack portion.

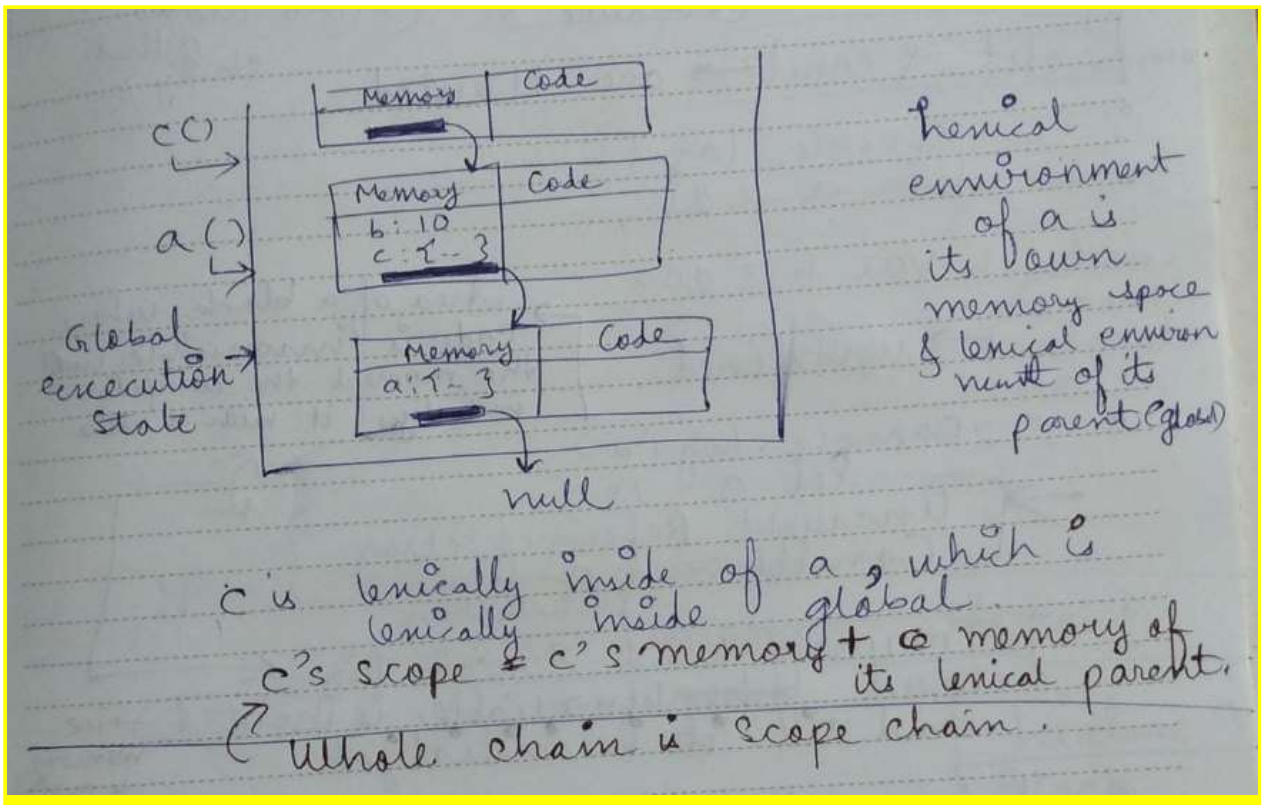
What's the lexical environment of a?

Local memory of a+ lexical environment of its parent

Lets see c's lexical environment .

Handwritten Notes





SUMMARY-

- **Scope is directly related to lexical environment.**
- **Scope is where you can access the variable or functions.**
- **Lexical ,as a term,means hierarchy or in a sequence or in order.**
- **Lexical environment = EC's Local Memory + Reference to Lexical Environment of its parent.**
- **The Lexical Environment of its parent is the scope where a function is physically present or defined. So, suppose a function x(), is defined and invoked in the GEC, when function x()'s EC is pushed in the call stack, it stores a reference to its parent's lexical environment i.e. the GEC's memory.**
- **Whenever a new Execution Context is pushed in the Call Stack it holds a reference to the Lexical Environment of its parent, i.e. the EC's memory from where it was invoked.**
- **Global execution context holds reference to null.**
- **Javascript engine first looks for the variable/function being accessed in the local scope of the function, and if not found, it keeps on searching the lexical environment of its parent until it finds the variable/function being accessed.**

- **This way of finding or mechanism is known as scope chain, the whole chain of all lexical environments. It defines whether a function or variable is present inside a scope or not.**
- **If the variable accessed is not found in the Scope Chain, then you will get the variable is not defined error in the browser's console.**
- **when one layer of onion(a variable) covers the layer inside it, it also covers the inner layers inside that layer and a point on our first layer of onion encloses inner layers our inner layer(variable and function inside the layer) gets covered by our first layer(can access the variable on the context) and the second layer(under first layer) encloses the third layer(innermost layer)still covered by 1st layer(can access the variable on layer).**
- **An inner function can access variables which are in outer functions even if inner function is nested deep. In any other case, a function can't access variables not in its scope.**
- **Global**
  {
  Outer
  {
  Inner
  }
  }
  Inner is surrounded by lexical scope of Outer

---

let & const in JS 🔥Temporal Dead Zone | | Namaste JavaScript Ep. 8

Youtube Link -https://youtu.be/BNC6slYCj50

- **What are temporal dead zones?**
- **Are let and const hoisted?**
- **Difference between SyntaxError , ReferenceError , TypeError ?**

To start off with,

Yes, let and const declarations are hoisted but they are in a temporal dead zone for the time being.

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

We can access b before declaring it due to hoisting. In other languages this gives us an error.

If b is hoisted, we should be able to access a as well due to hoisting?

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

Output

```
Uncaught ReferenceError: Cannot access 'a' before initialization
    at hello.js:1:13
```

Uncaught ReferenceError occurs.

Now

1

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

```
undefined
undefined
```

2

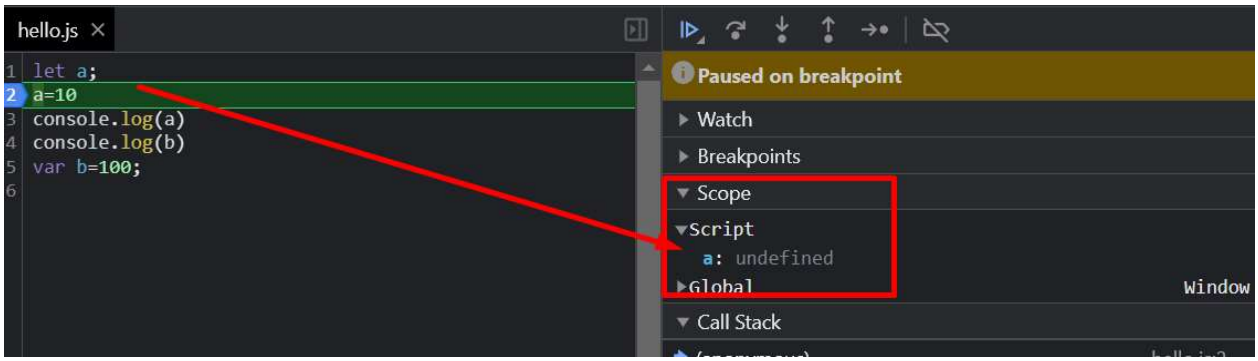
```
let a;
a=10
console.log(a)
console.log(b)
var b=100;
```

```
10
undefined
>
```

So we see that only after we have declared let a and initialized some value; then only we can access it.

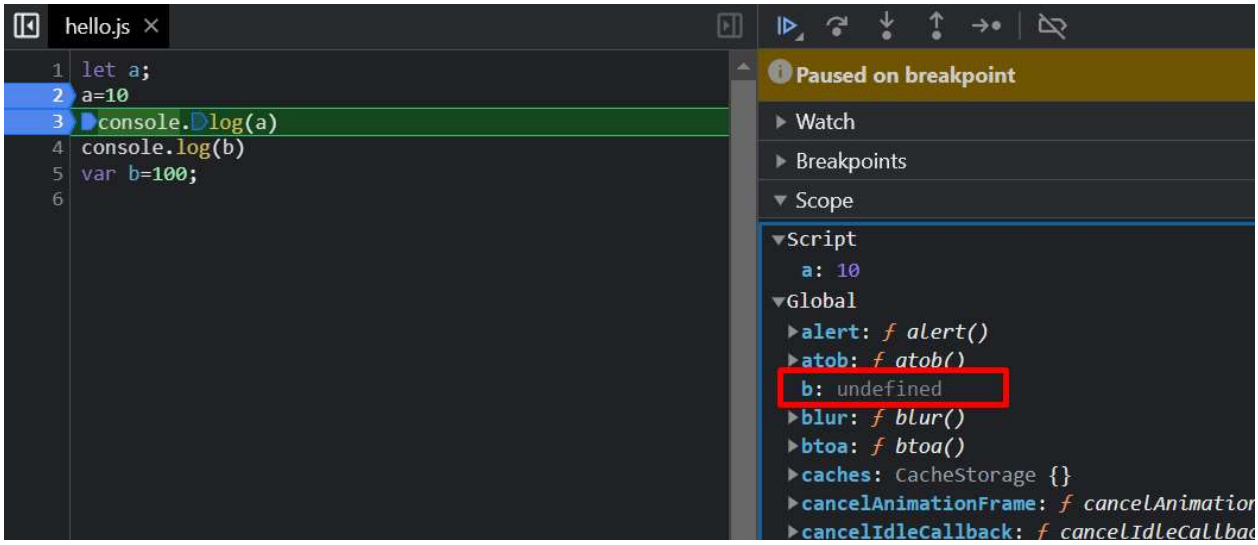
So how do we know let is actually hoisted?

Let's put a debugger and see in sources.



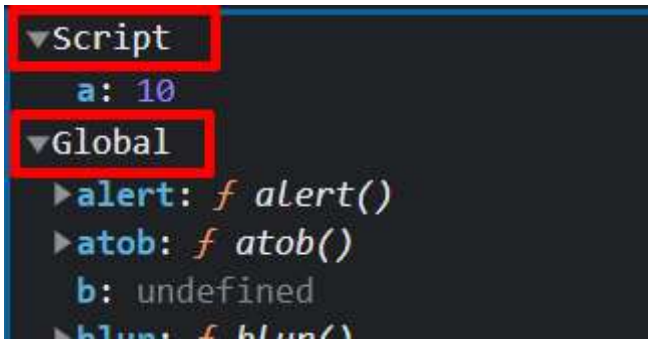
We see that a is already there as undefined.When flow of control moves to next line,then a gets a value of 10, thus showing the concept of hoisting.

We know that var b has been hoisted so let's find it out as well.



We found b which is undefined as well due to hoisting.

But if we see, there are 2 sections.



Script and Global.

Variable b was attached to the global object. Let and const are also allocated memory but they are stored in separate memory space called script,so one cannot access let and const before you have put some value in them.

Temporal dead zone is

- **Time from when it was hoisted till it was initialized some value**
- **Area where let variable cannot be accessed**

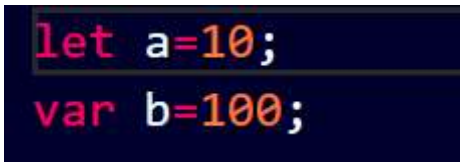
Above, the area in red is the temporal dead zone for let variable a.

When line 4 is executed,then temporal dead zone ends for a.So before line 4,everything was temporal dead zone for a.

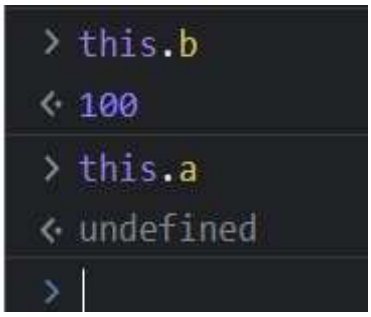
var variables are attached to window object as they are present in the Global space and window is global object, so everything in Global space can be accessed with window.

But let,const cannot be accessed using window.

For code



In console:-



let is more strict.

We can redeclare var.

```
var b=100;
var b=200;
```

But we cannot redeclare let.

```
1    let a=10;
2    let a=20;
```

Output

```
Uncaught SyntaxError: Identifier 'a' has already been declared (at hello.js:2:5)
```

In VS code error was shown on both lines but in console, error was shown on line 2 only.

Now

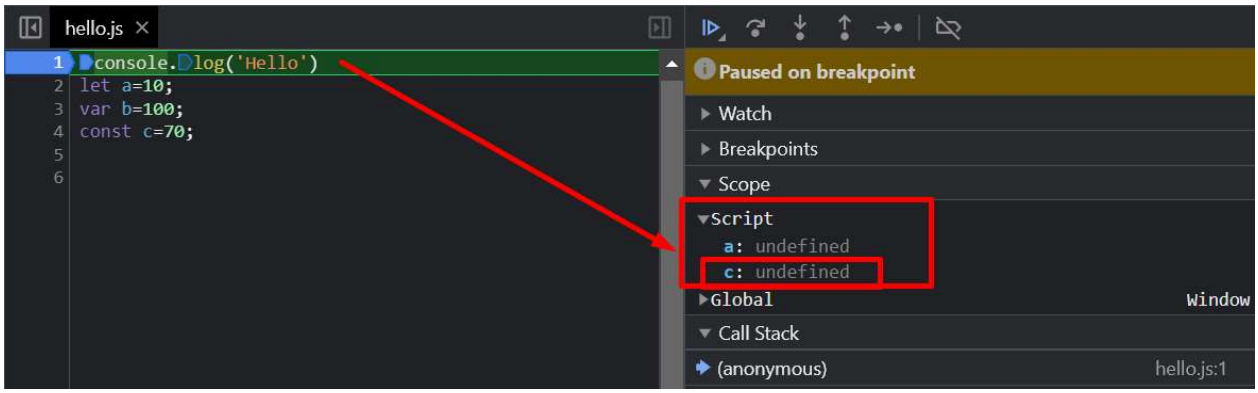
```
console.log('Hello')
let a=10;
let a=20;
```

Even if we try to log something before the error, javascript outright rejects it even before reaching the line where error occurs. So in case of SyntaxError no part of code is executed.

```
Uncaught SyntaxError: Identifier 'a' has already been declared (at hello.js:3:5)    hello.js:3
```

Talking about const also behaves the same way as let and goes into temporal dead zone. But const is more stricter than let.

```
1    console.log('Hello')
2    let a=10;
3    var b=100;
4    const c=70;
```



We see that const c is stored in script just like let a and c is hoisted as well.

We can do below with let

```
1    let a;
2
3    a=10;
```

Can we do same with const?

```
1    const a;
2
3    a=10;
```

NO we CANNOT because we need to initialize const variables while declaring it.

Output

```
Uncaught SyntaxError: Missing initializer in const declaration (at hello.js:1:7)
>
```

Error is Syntax Error: Missing initializer in const declaration.

Now,

```
1   const a=10;
2   a=10;
```

Is this ok?

NOO, NOT OK

```
Uncaught TypeError: Assignment to constant variable.
    at hello.js:2:2
>
```

Here error is TypeError which is assignment to constant variable. As we cannot re-assign constant variables.

SUMMARY of ERRORS

1)

```
1   console.log(a)
2   let a=5
```

```
Uncaught ReferenceError: Cannot access 'a' before initialization                hello.js:1
    at hello.js:1:13
>
```

2)

```
console.log(y)
```

```
Uncaught ReferenceError: y is not defined
    at hello.js:1:13
>
```

3)

```
console.log('Hello')
let a=10;
let a=20;
```

```
Uncaught SyntaxError: Identifier 'a' has already been declared (at hello.js:3:5)        hello.js:3
>
```

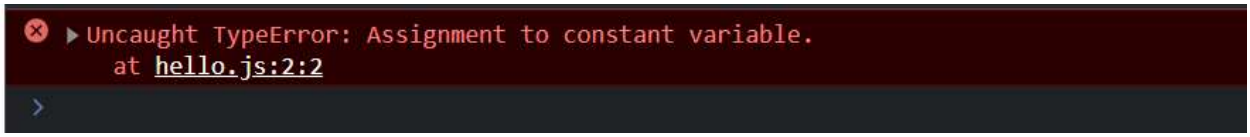
4)

```
1   const a;
2
3   a=10;
```

```
Uncaught SyntaxError: Missing initializer in const declaration (at hello.js:1:7)
>
```

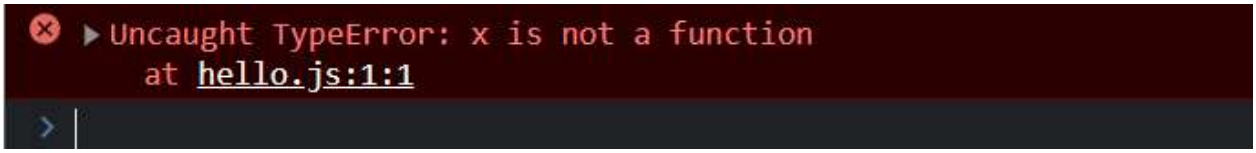
5)

```
1   const a=10;
2   a=10;
```



6)

```
x()
var x=function(){}
```



GOOD PRACTICE->

It is recommended to use const.

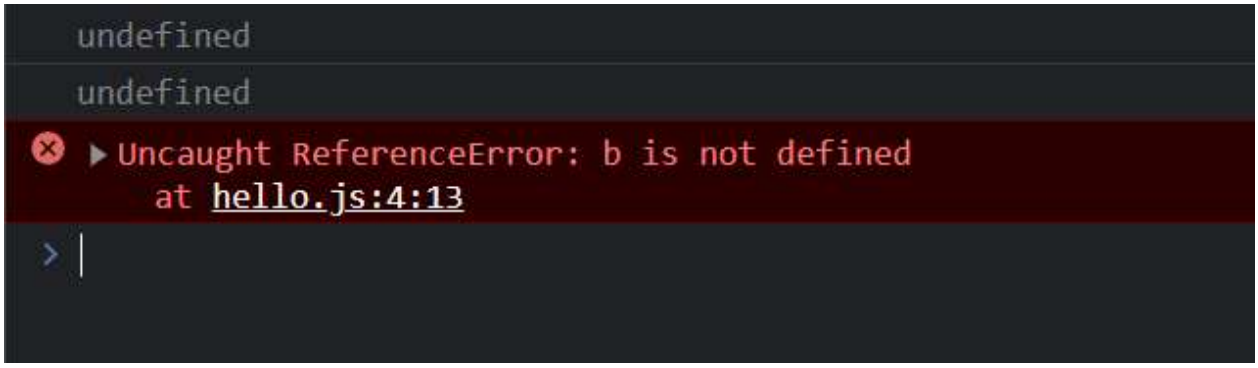
If not const, then use let. Keep aside var.

To avoid temporal dead zones,keep your declarations and initializations at the top.

So as soon as your code starts executing,it hits initialization part first.Thus, this is shrinking of the temporal dead zone.

One weird output

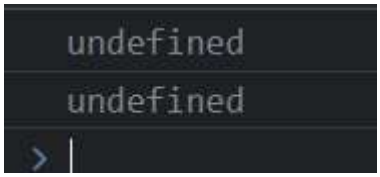
```
console.log(this.a)
console.log(window.a)
console.log(b)
```



code

```
let a=5
console.log(this.a)
console.log(window.a)
```

Output

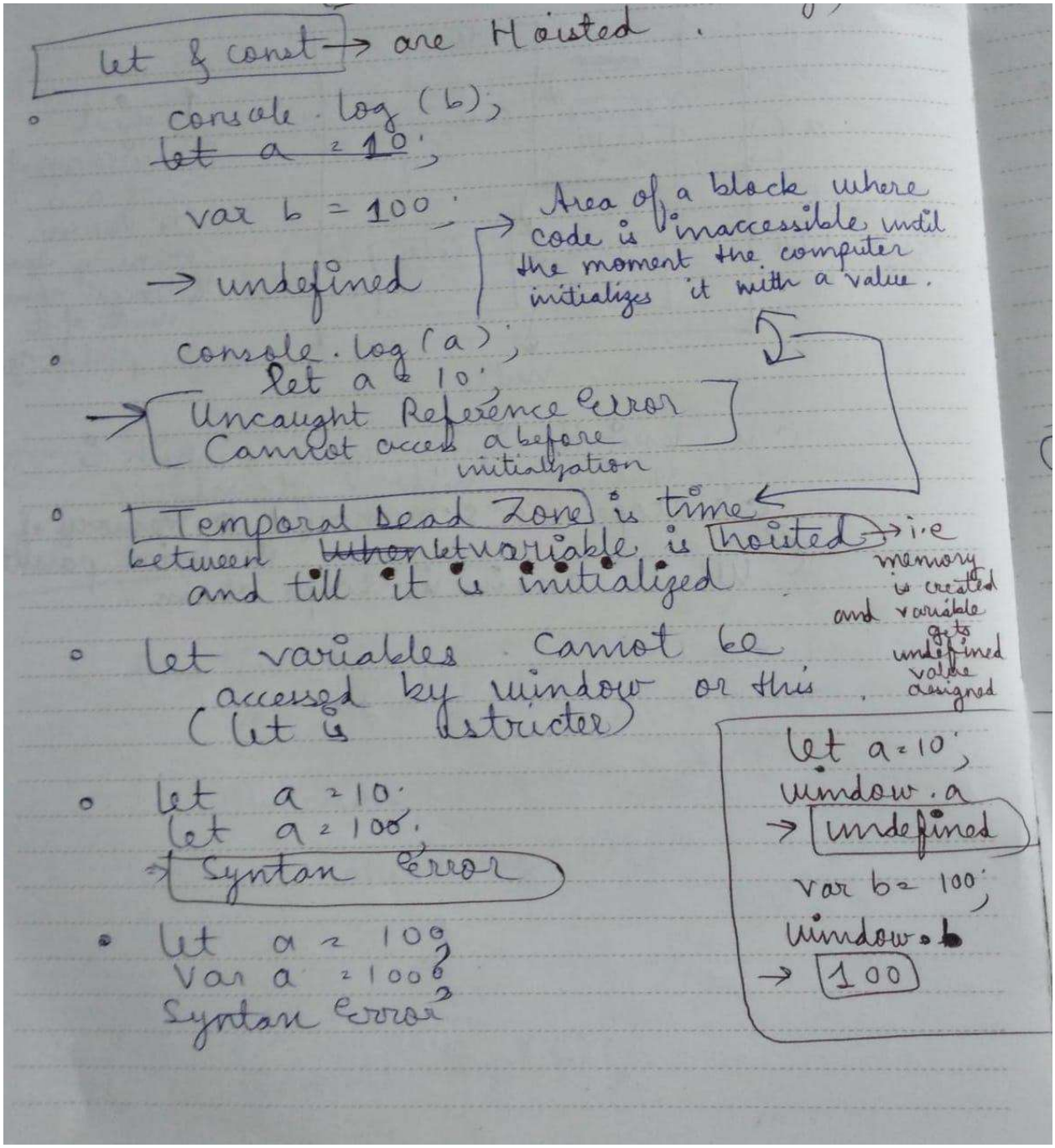


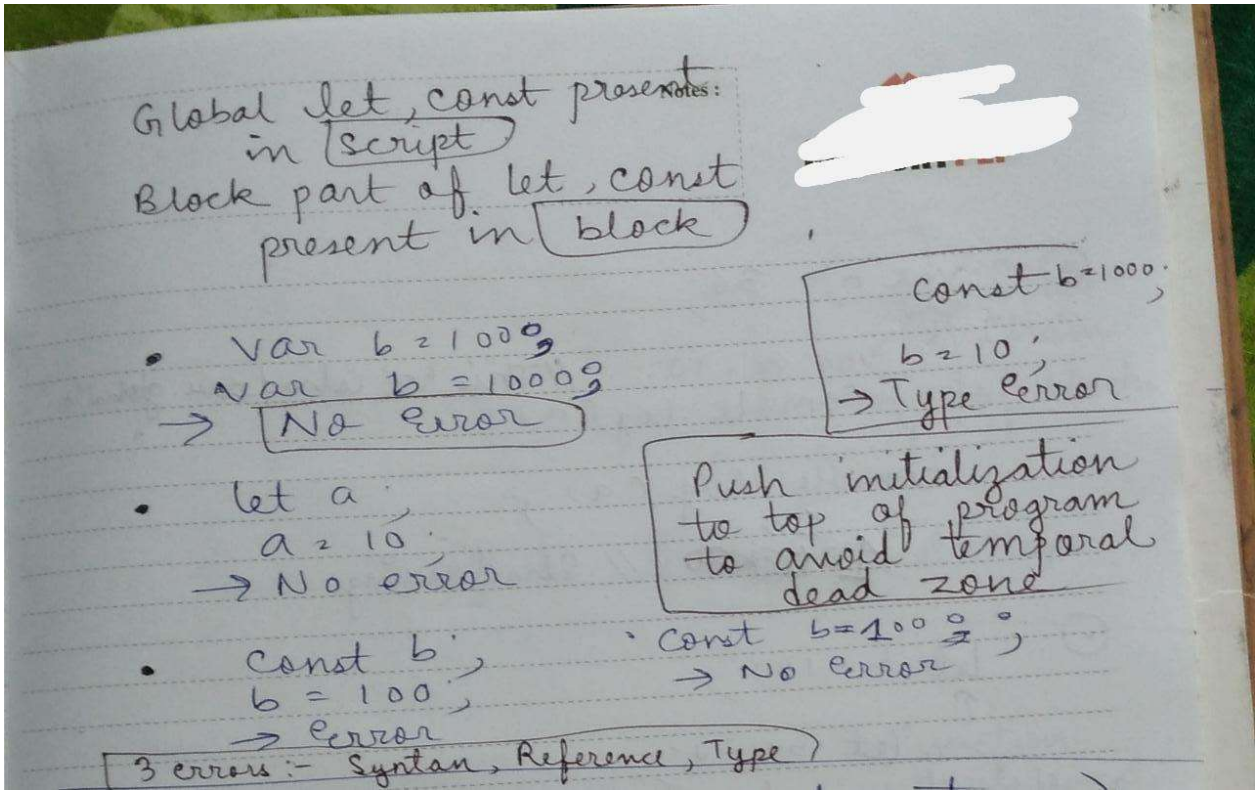
code

```
const a=5
console.log(this.a)
console.log(window.a)
```

Output

Handwritten Notes





SUMMARY

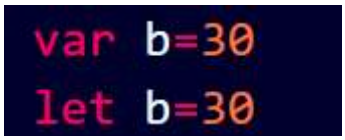
- let and const are hoisted but its memory is allocated at other place known as script which cannot be accessed before initialisation.
- Temporal dead zone is time from when it was hoisted till it was initialized some value.
- Temporal Dead Zone exists until variable is declared and assigned a value.
- window.variable OR this.variable will not give value of variable defined using let or const.
- We cannot redeclare the same variable with let/const(even with using var the second time).
- const variable declaration and initialisation must be done on the same line.
- 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}.
- Use const wherever possible followed by let, Use var as little as possible(only if you have to). It helps avoid errors.
- Initializing variables at the top of code is a good idea, helps shrink the temporal dead zone to zero.
- let and const are hoisted. we can't use them before initialization is the result of "temporal dead zone".

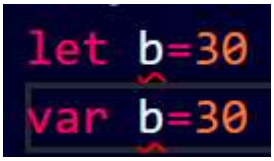
- **js use diff memory known as script rather than global execution context to store let and cost which is reason behind "temporal dead zone"**
- **level of strictness ... var<<let<<const.**
- **var //no temporal dead zone, can redeclare and re-initialize, stored in GES**
- **let //use TDZ, can't re-declare, can re-initialize, stored in separate memory**
- **const //use TDZ, can't re-declare, can't re-initialize, stored in separate memory**
- **syntax error is similar to compile error. while type and reference error falls under run time error.**
- **syntax error → violation of JS syntax**
- **type error → while trying to re-initialize const variable**
- **reference error →while trying to access variable which is not there in global memory.**

BLOCK SCOPE & Shadowing in JS 🔥 | Namaste JavaScript 🙏 Ep. 9

Youtube Link -https://youtu.be/lW_erSjyMeM

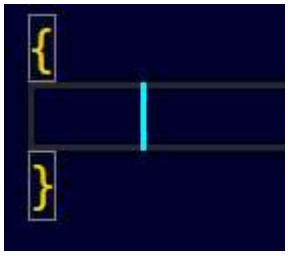
Before starting





Above are wrong.

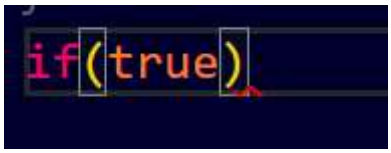
Block and scope are 2 different things.



A block is defined by curly braces. This is a perfectly valid JS code.

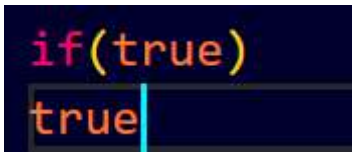
Block is also known as compound statement,that is, combining multiple javascript statements into a group.Why we do this? ->so that we use the group of multiple statements where javascript expects one statement.

Example



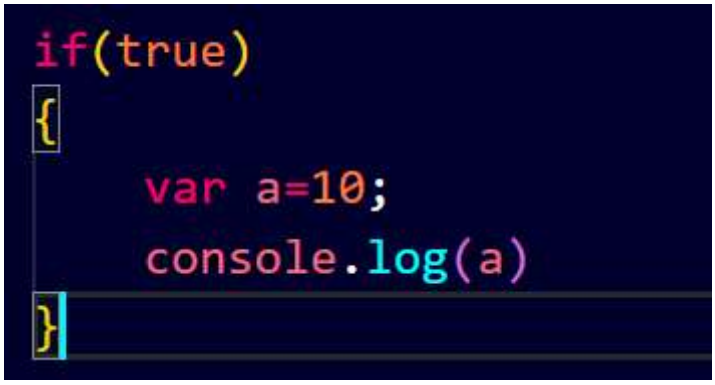
Output

It is an error as if is expecting one statement.

Now this is valid 

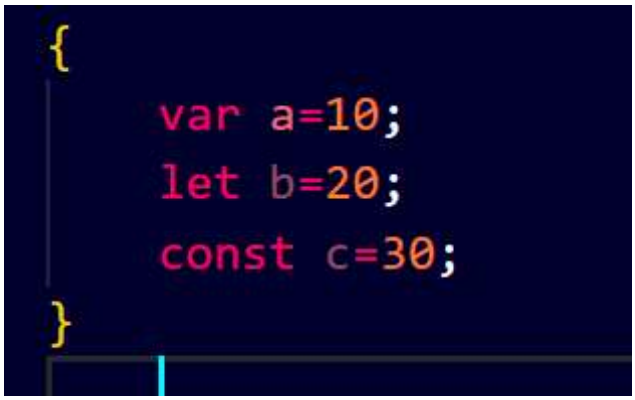
.

But if we want to write multiple statements after we can do that by grouping them together inside curly braces.

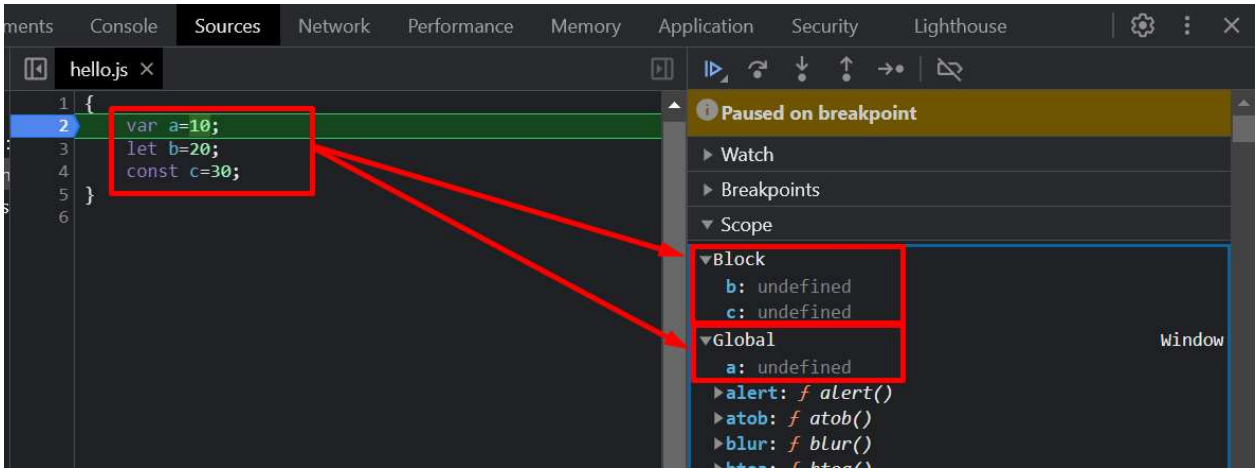


So we can use a block to combine multiple statements where JS expects a single statement.

Block scope is , what all variables, functions , we can access inside a block.



Let's understand this code.



Let and const are stored in Block space.

var is stored in Global space.

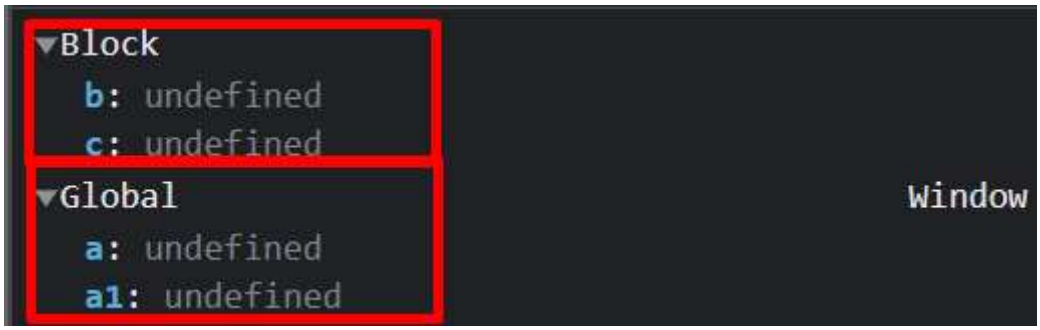
So b and c are hoisted in Block and assigned undefined.

Block is separated memory space reserved for block.

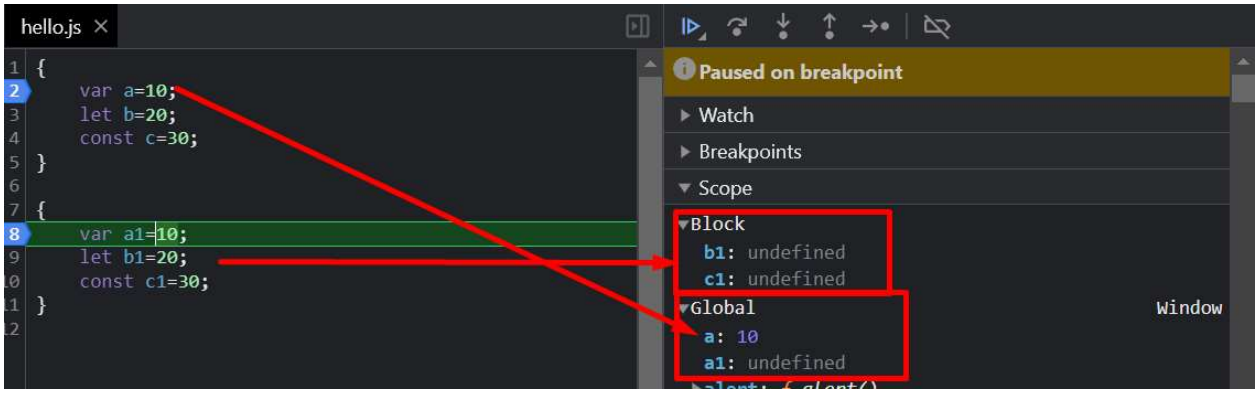
For multiple blocks,

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

{
    var a1=10;
    let b1=20;
    const c1=30;
}
```

We see b1,c1 are not hoisted as of now when flow of control enters 1st block.



But as soon as flow of control enters 2nd block, 👇

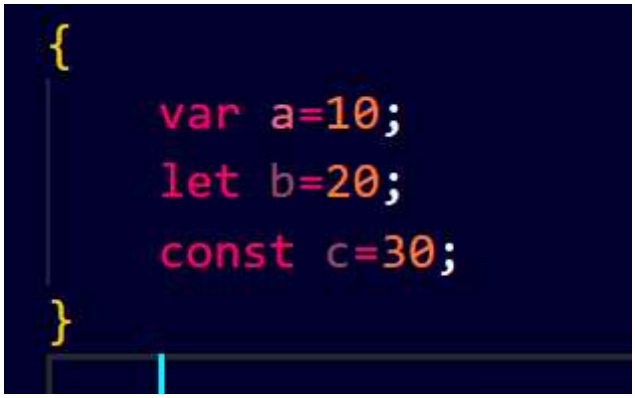


We see b,c are removed from Block Area as soon as flow of control exits first block.

b1 ,c1 are hoisted as well in the Block area. a is assigned 10 due to the previous block.

So we can say let,const inside a block will be hoisted only when flow of control enters that block while var of all blocks will be hoisted at beginning.

Coming back to our original example



We cannot access let and const outside the block meaning let and const are block scoped while var is Global scoped so it can accessed outside the block as well.

If we try to do this

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

Output

As expected b and c cannot be accessed outside.

If you have a variable outside the block then the variable with same name inside the block shadows the variable outside.

Code-

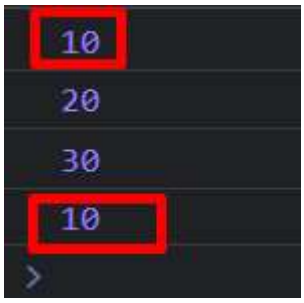
```
1    var a=100
2    {
3        var a=10;
4        let b=20;
5        const c=30;
6        console.log(a)
7        console.log(b)
8        console.log(c)
9    }
```

```
10
20
30
```

So we see the value of 10 was overwritten,i.e. var a inside block shadowed var a outside block.

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

Output



So we see 10 twice because var is present only in the Global scope and value in Global scope persists even after block ends, so value 10 remains. Thus,var inside and outside any block ,point to same memory location.

Now lets experiment with let.

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

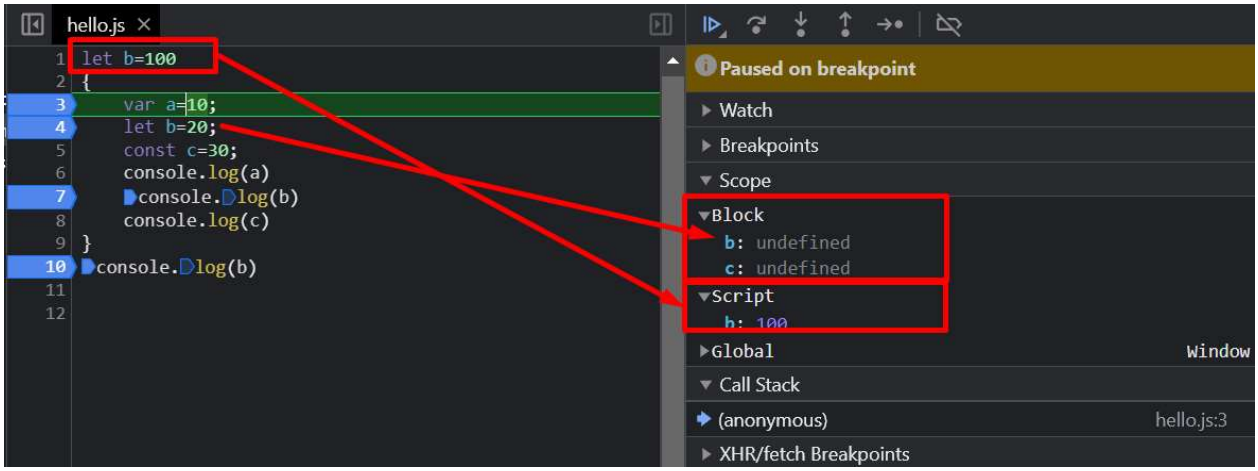
Output

```
10
20
30
100
>
```

This was shadowing.

We can understand that let inside block and let outside block are at different memory locations unlike var. Let outside block,as we studied before,is stored in script space.Let inside block is stored inside in block space.

So lets use a debugger and analyze.



b is in two different places.

All 3 separate memory spaces

Notice var is in Global space.

b in block memory space shadows b present in script memory space.

As soon as block ends, Block memory space is erased hence b and c inside Block also get removed, thus the let inside Script is used to log the value of b.

Let's experiment with const.

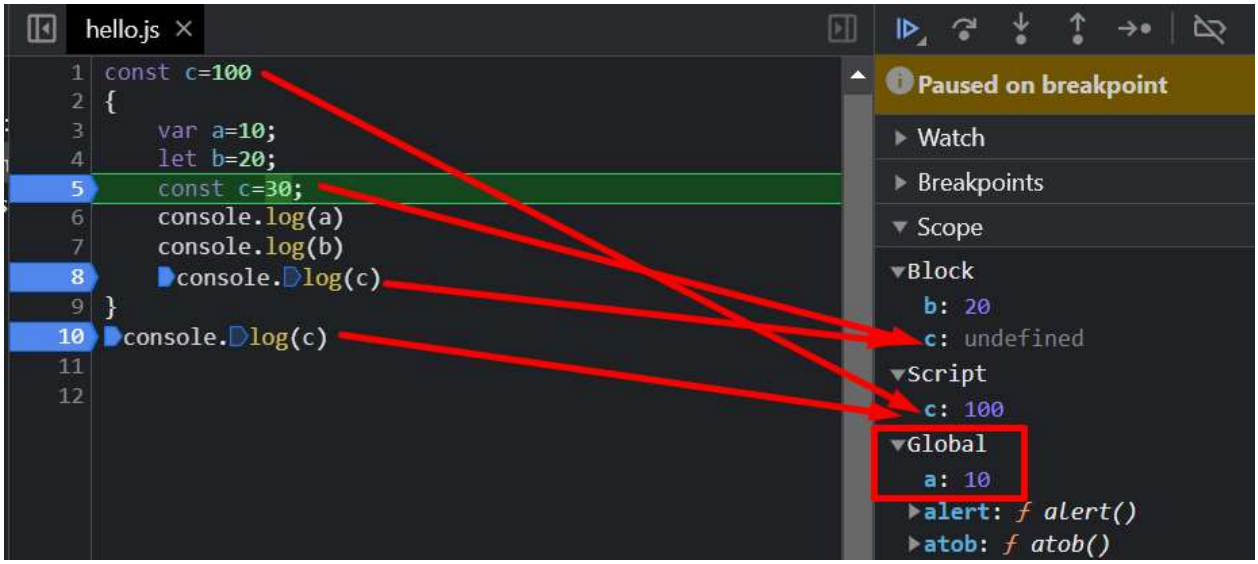
```
const c=100
{
    var a=10;
    let b=20;
    const c=30;
    console.log(a)
    console.log(b)
    console.log(c)
}
console.log(c)
```

Output

```
10
20
30
100
```

Like let, the const c inside block shadows const c outside block.

With debugger,



Now we will shadowing is not just a concept of block it can be implemented in block also.

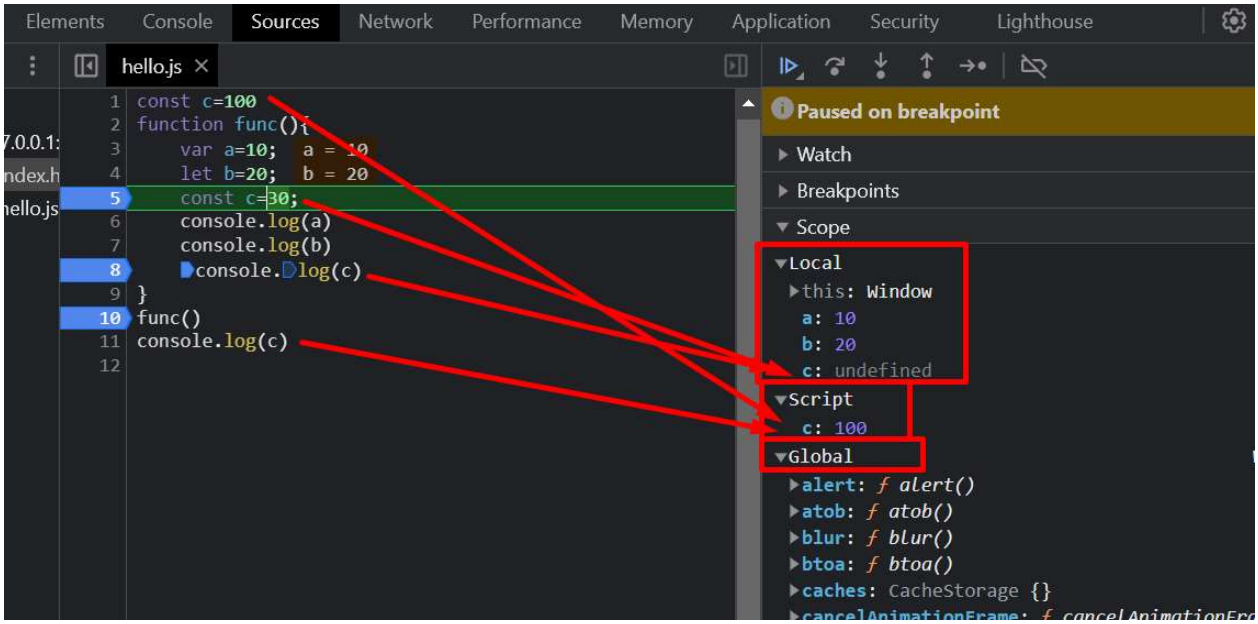
Code -

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

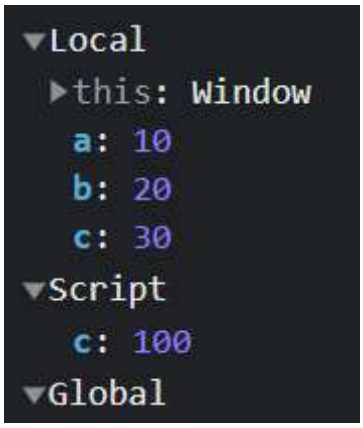
Output

```
10
20
30
100
```

Using debugger,



We see const and let inside function are stored in the Local space of that function.



And when function invocation and execution is over.

Local environment of function is removed, and only Script and Global space remain.

Lets experiment var with function.

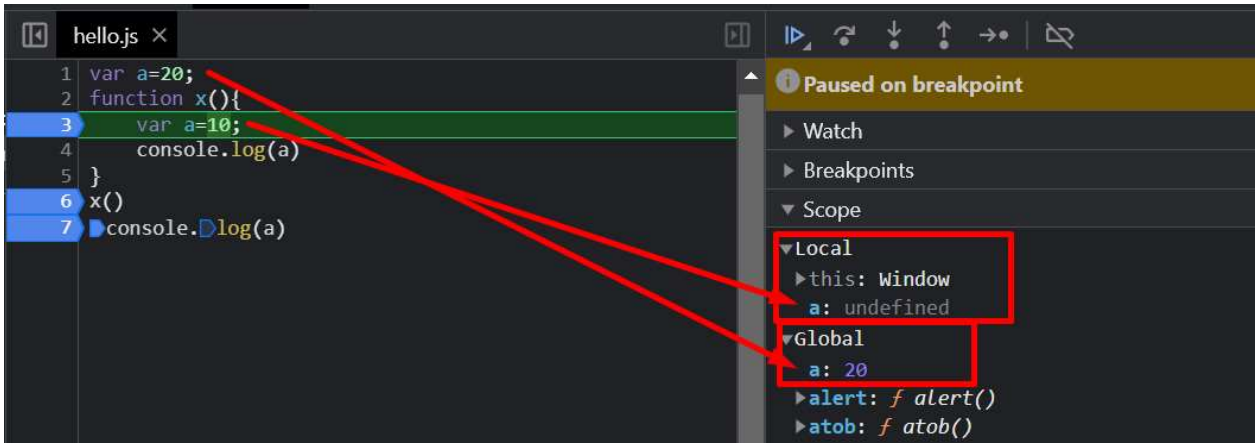
```
var a=20;
function x(){
    var a=10;
    console.log(a)
}
x()
console.log(a)
```

Output

```
10
20
```

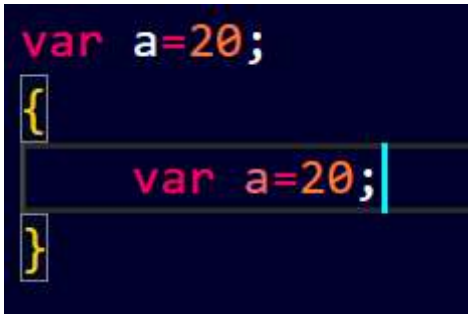
Surprising isnt it? Because when we used block instead of function both outputs were supposed to be 10. That was because var was global scoped.

Here, var is function scoped as well as global scoped, which means Local space of function will have separate copy of a unlike in blocks where there was only Global space.Here there is Local environment of function and Global space both due to usage of functions.

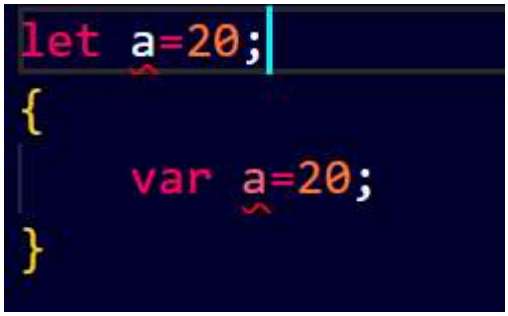


Both the var variables have been hoisted in their respective memory areas.On logging a inside function ,it searches in the lexical environment of the function first present inside the execution context of the function x. Since it finds a inside x no more searches x in the lexical environment of the parent,which is the global scope. And after function has executed,Local space of the function gets erased as the execution context of x() pops off the call stack. And then when we log a ,value of a present in Global space of global execution context is taken.

LETS see what is ILLEGAL SHADOWING.



We can do the above.



But we cannot do this.This is Illegal Shadowing.

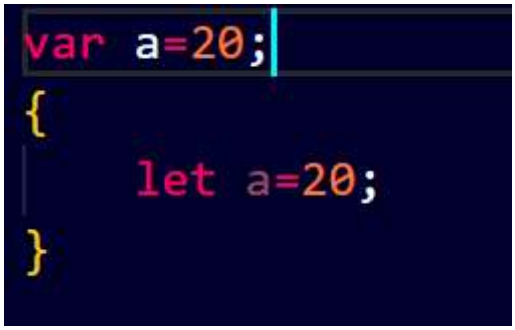
Output

```
Uncaught SyntaxError: Identifier 'a' has already been declared (at hello.js:3:9)
>
```

We can shadow let using let but we cannot shadow let using var.

Illegal Shadowing: Now, while shadowing a variable, it should not cross the boundary of the scope, i.e. we can shadow var variable by let variable but cannot do the opposite. Var is crossing the boundary of let so let cannot be shadowed by var.

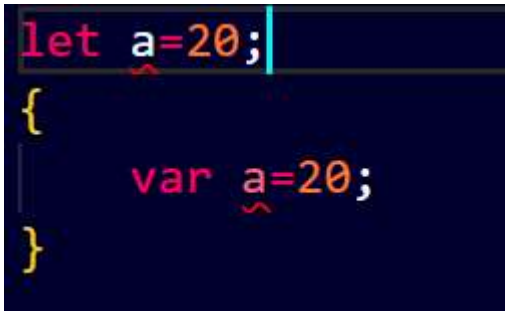
Is vice-versa possible?



Yes ,possible.

In blocks:-

- **let can be shadowed by let,const but not var.**
- **const can be shadowed by let,const but not var.**
- **var can be shadowed by var,let,const.**
- **We can draw the conclusion-**
- **More strict like let,const can shadow the less strict and themselves.**



This was giving us an error.

So let's try the same thing with function.

```
let a=20;
function x(){
    var a=10;
    console.log(a)
}
x()
console.log(a)
```

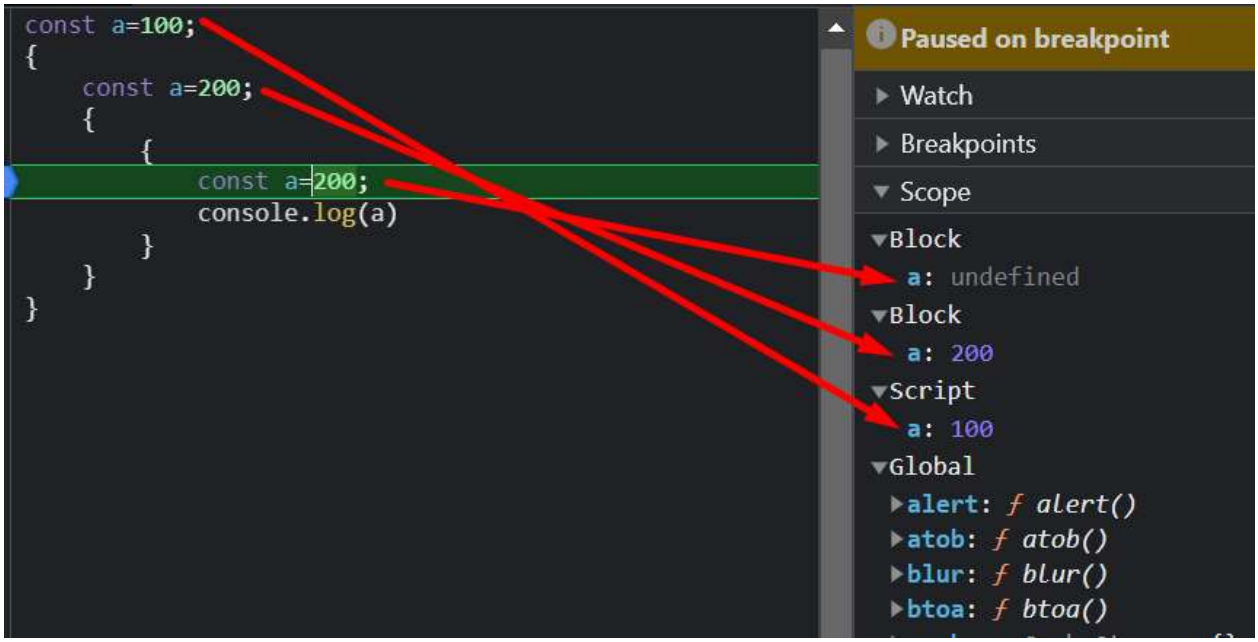
It has no error. And gives output

```
10
20
```

Inside a function anyone can shadow anyone.As let,const,var all are function scoped.

There is something known as lexical block scope as well.

```
const a=100;
{
    const a=200;
    {
        {
            const a=200;
            console.log(a)
        }
    }
}
```

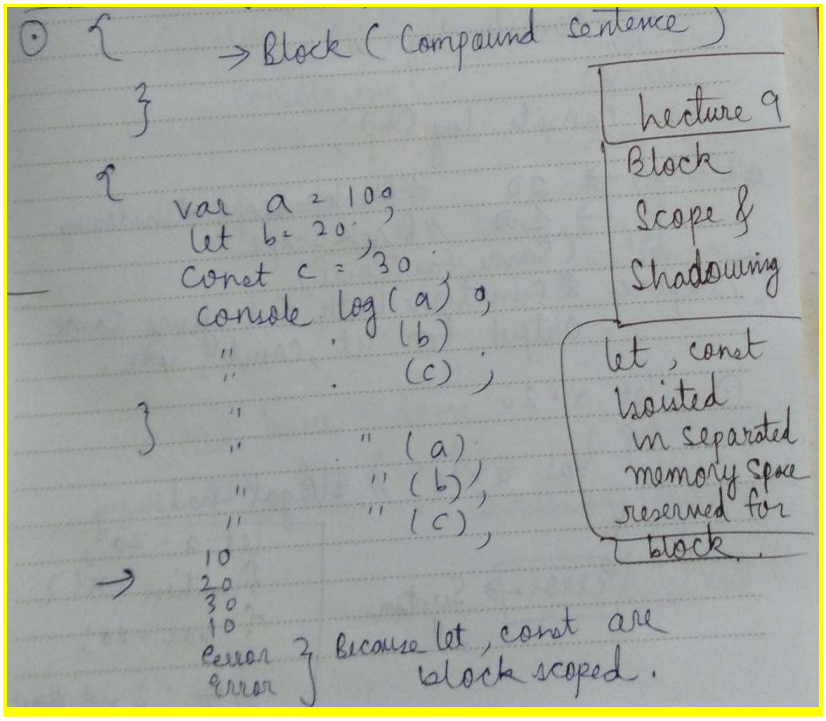


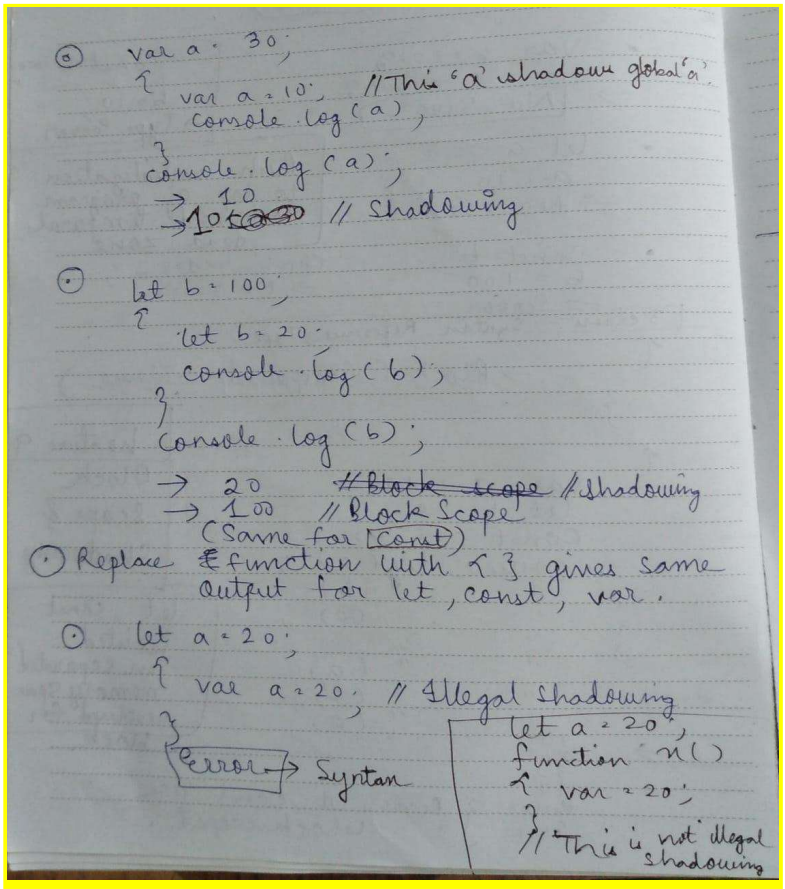
When we log a then JS tries to find a in the current lexical block.It finds a inside the current lexical block and prints it. Lets say a was not found in current lexical block ,then JS would proceed to find a in its parent lexical blocks environment, the mechanism which we read was called scope chaining. If it the entire scope chaining was exhausted and a was till not found then we could have said a was not present in any of lexical scope of blocks.

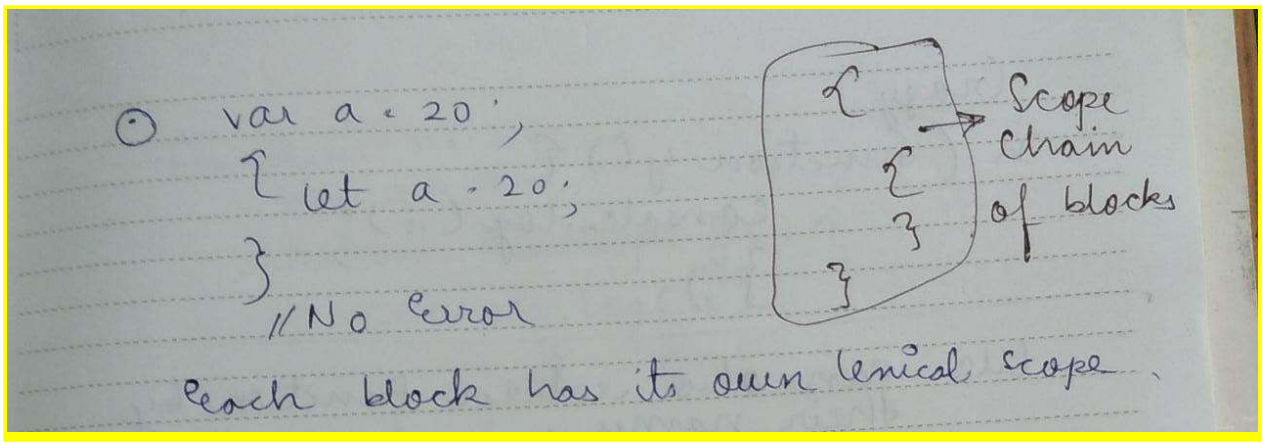
Each block of code has its own Block span as seen above.

All the scope rules that work for arrow functions work same for arrow functions as well.

Handwritten Notes







SUMMARY

- **Code inside curly bracket is called block.**
- **Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if, else, loop, function etc.**
- **Block values are stored inside separate memory than global. They are stored in block. (the reason let and const are called block scope)**
- **The shadow should not cross the scope of original otherwise it will give error.**
- **shadowing let with var is illegal shadowing and gives error.**
- **var value stored in global memory and hence can be accessed outside block as well whereas same is not the case with let and const.**
- **Script,block,local,global are 4 memory areas.**

Now for a little bit of experiment apart from the video.

```
{
    function func()
    {
        console.log('I am inisde parent lexical block')
    }
    {
        func()
    }
}
```

We can call func() present in lexical environment of parent.

Output

```
I am inisde parent lexical block
>
```

Code-

```
{
    function func()
    {
        var x=10
        console.log('I am inisde parent lexical block')
    }
    console.log(x)
}
```

Output-

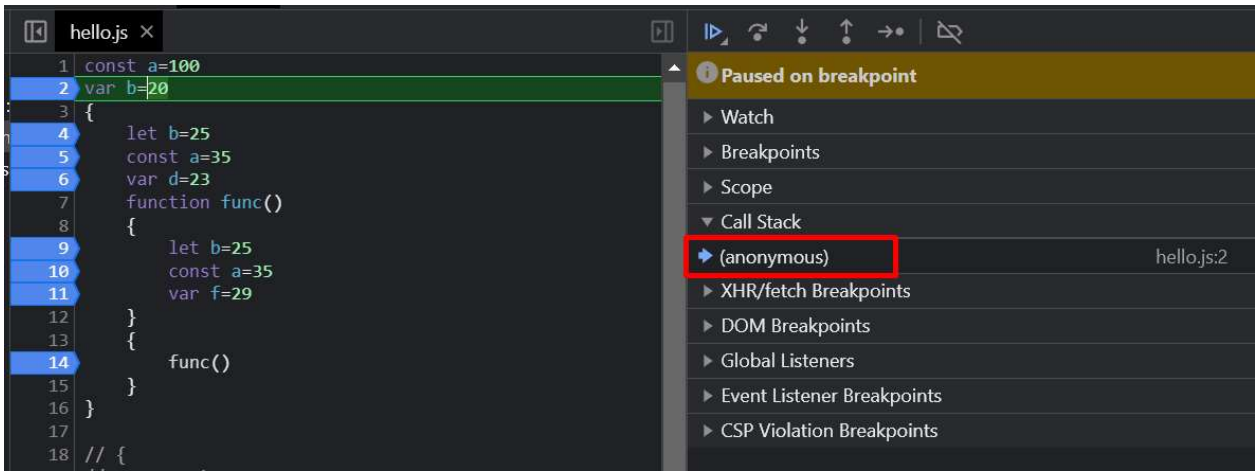
```
Uncaught ReferenceError: x is not defined
    at hello.js:7:17
```

Even if we used let or const instead of var,output would still be same as all are function scoped.

For the below complicated code we analyze line by line what happens with help of debugger. We will see all possible memory areas being created and the call stack.

```
const a=100
var b=20
{
    let b=25
    const a=35
    var d=23
    function func()
    {
        let b=25
        const a=35
        var f=29
    }
    {
        func()
    }
}
```

STEP 1

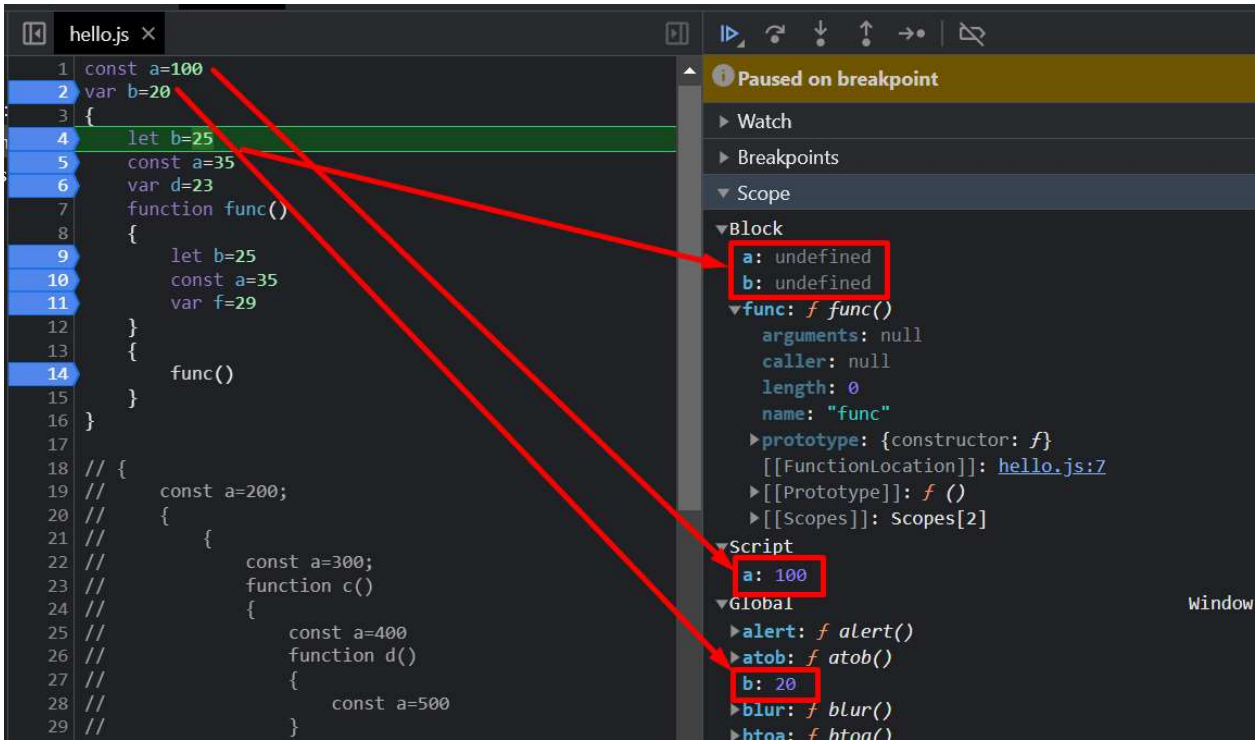


Global execution context created and pushed to call stack.

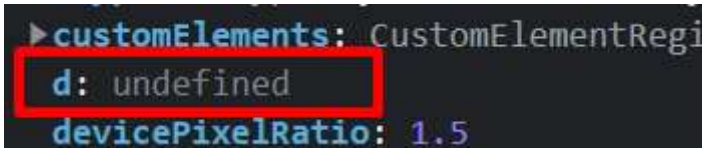
2 memory areas created Script having const a , Global having var b.

STEP 2

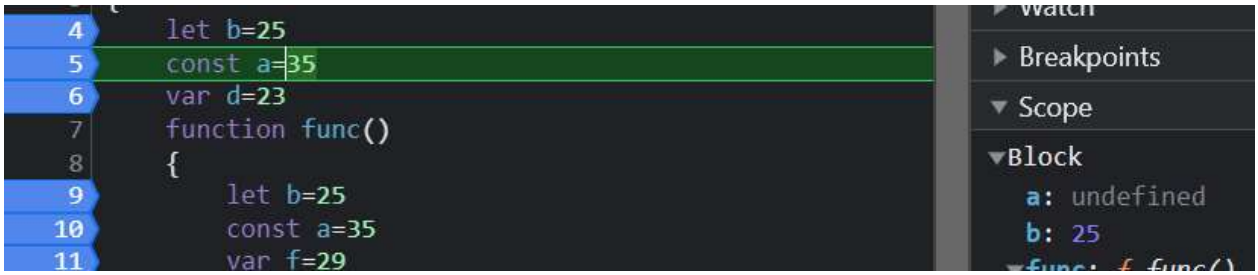
As control enters line 4



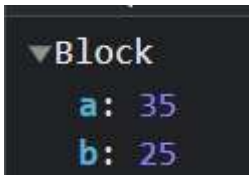
a,b are hoisted in Block getting undefined.



var d is also hoisted in Global if we scroll down.

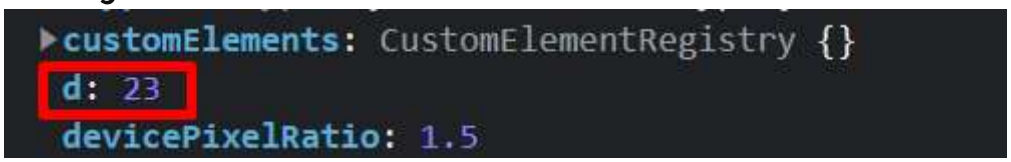


Then as flow moves to line 4, b gets value 25.



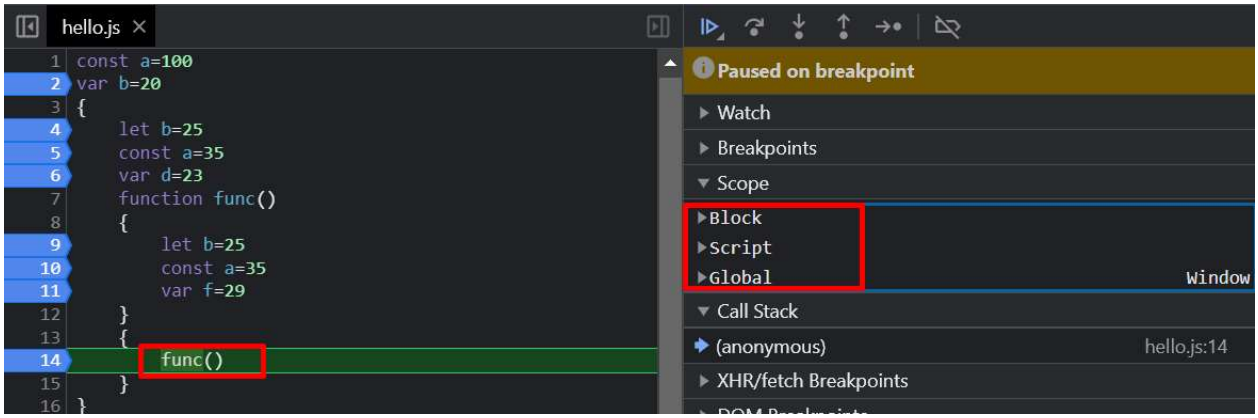
Both a and b get their values. d is till undefined.

Now d gets value 23 as well

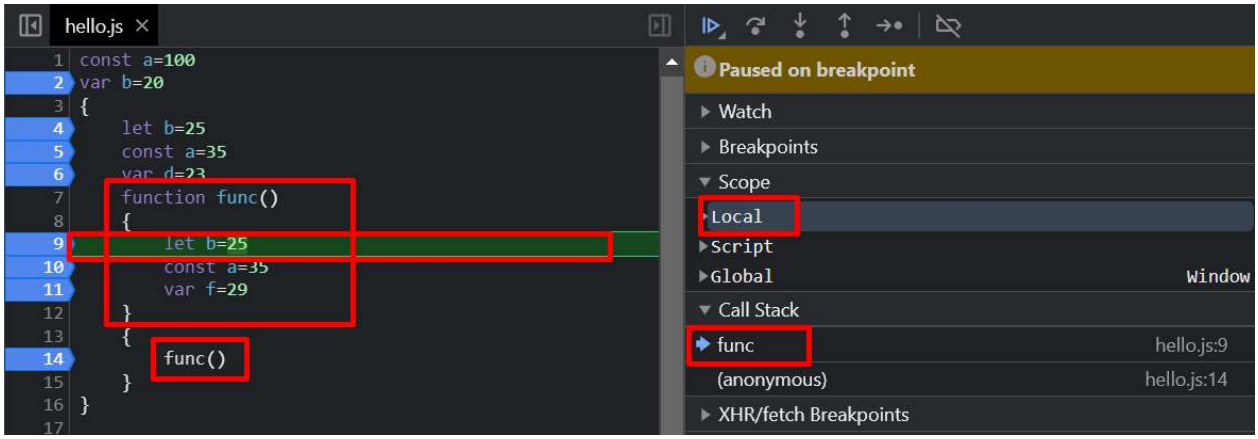


STEP 3

Debugger moves to line 14. We have 3 memory areas.



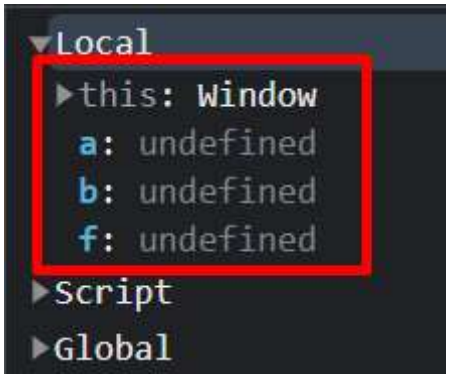
Now, many things happen in the pic below as flow reaches line 9.



Execution context of func created and pushed inside call stack.

If one notices,in the scope the Block memory has been removed and Local Memory inserted.

And we see Local

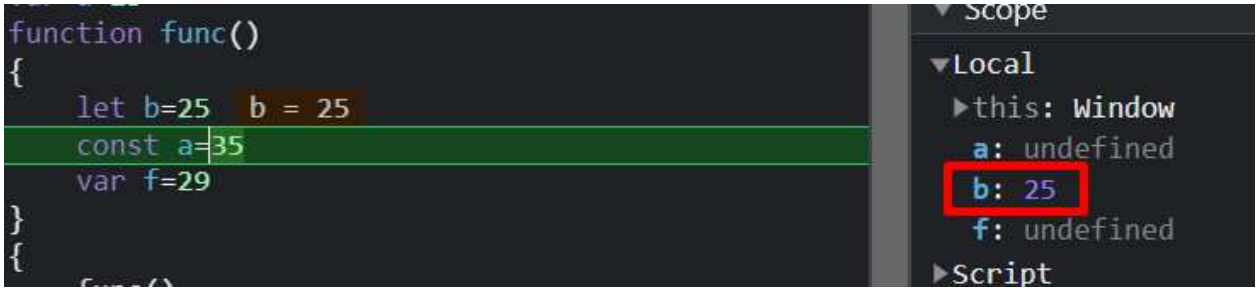


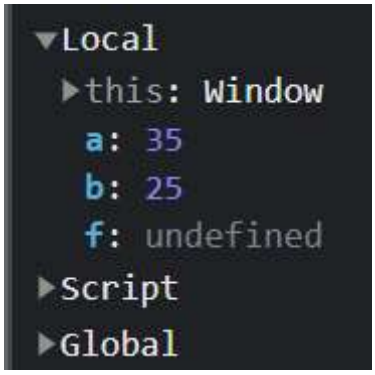
We will notice const a,let b,var f are hoisted inside function as they are function scoped.

Notice we declared them in order b,a,f but they appear as a,b,f inside Local space of function func.

STEP 4

As flow proceeds ,we see the variables getting their respective values.





STEP 5

And finally as function execution ends ,execution context of func() is popped off.

Local space of function is erased.

Thus this was one experimentation.

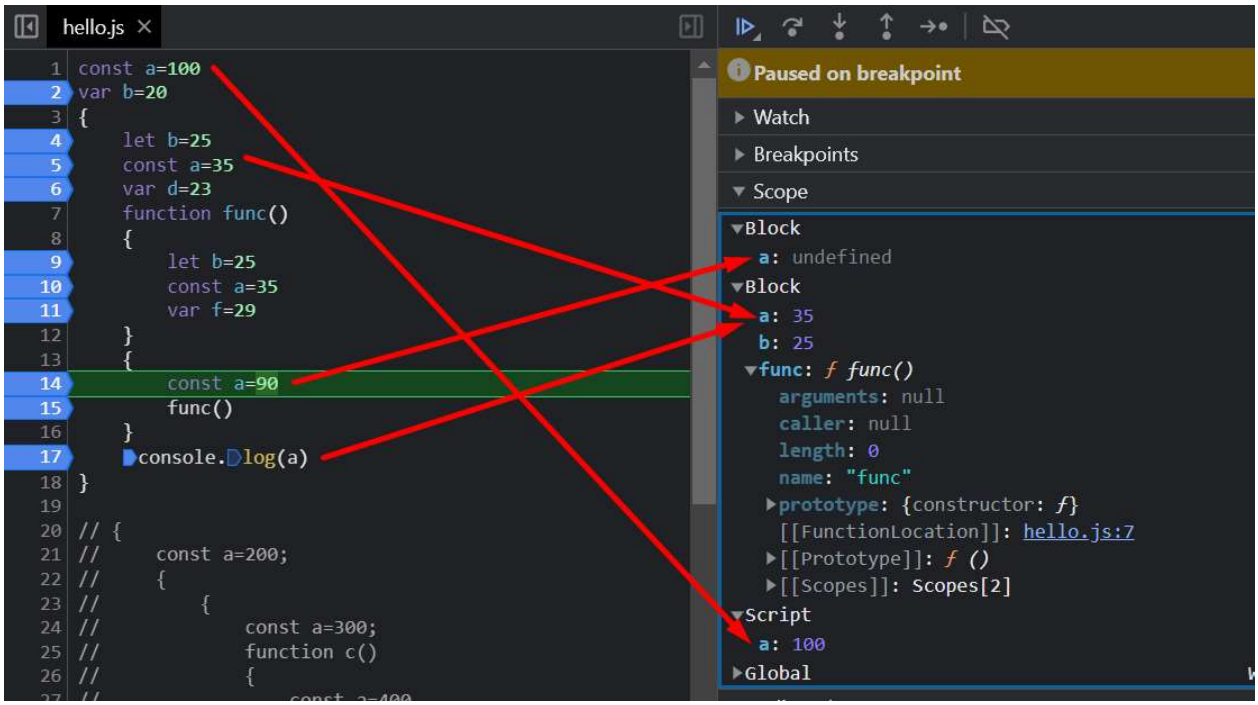
—----------

To further explore stuff,

Let's add 2 more lines,

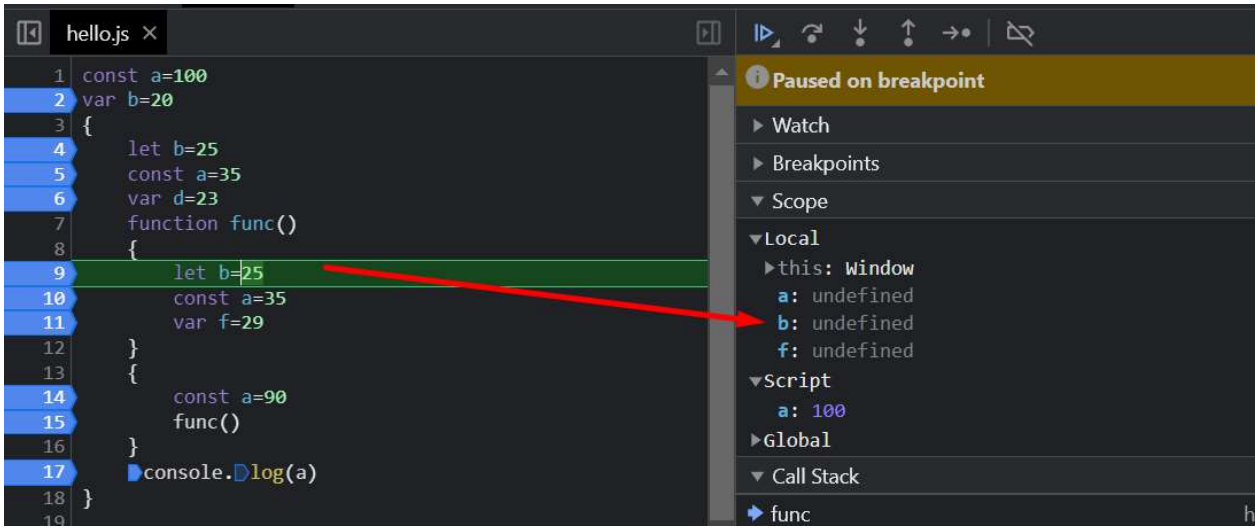
```
const a=100
var b=20
{
    let b=25
    const a=35
    var d=23
    function func()
    {
        let b=25
        const a=35
        var f=29
    }
    {
        const a=90
        func()
    }
    console.log(a)
}
```

When control reaches const a=90



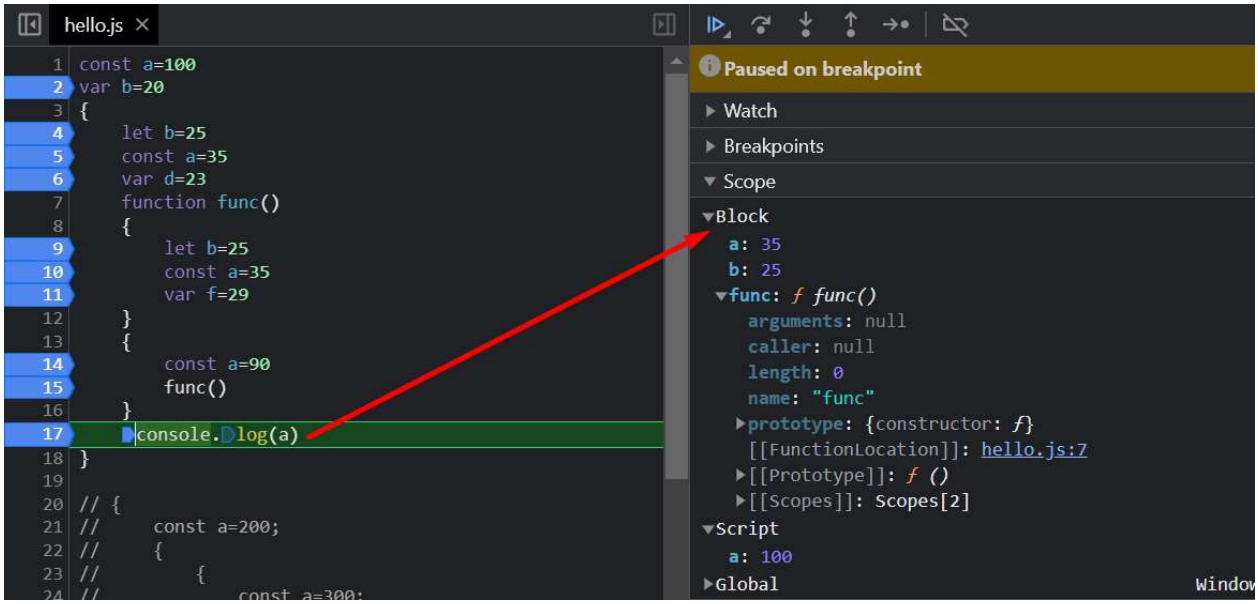
We see that in Scope, 2 Block memory areas are created.

And as soon as function call happens and control flow enters the function,



Again both the Block areas are erased and function func's Local Area is seen inside Scope.

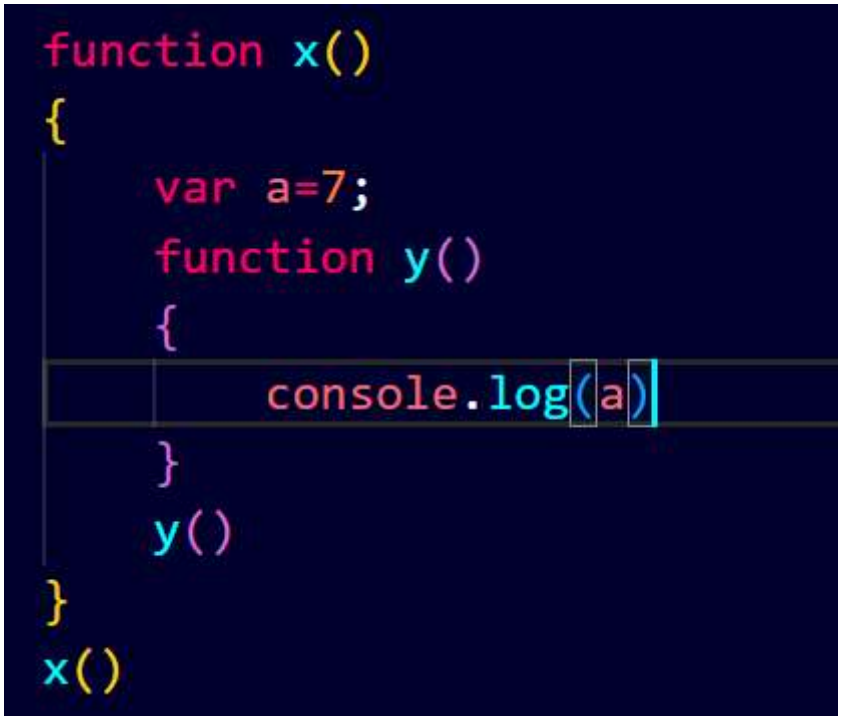
But after function call is over and control flow reaches line 17,again the Local area of function is erased,the execution context of a is popped and Block area of block is seen inside the Scope.



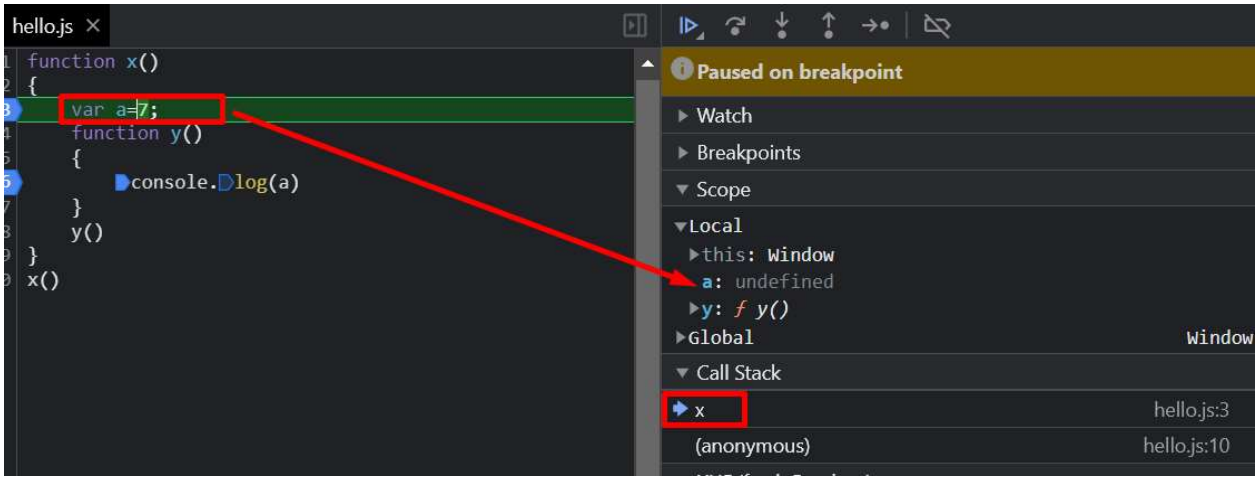
Thus was all about let,const,var,shadowing, the practical of Scope areas.

Closures in JS 🔥 | Namaste JavaScript Episode 10

Youtube Link-https://youtu.be/qikxEIxsXco

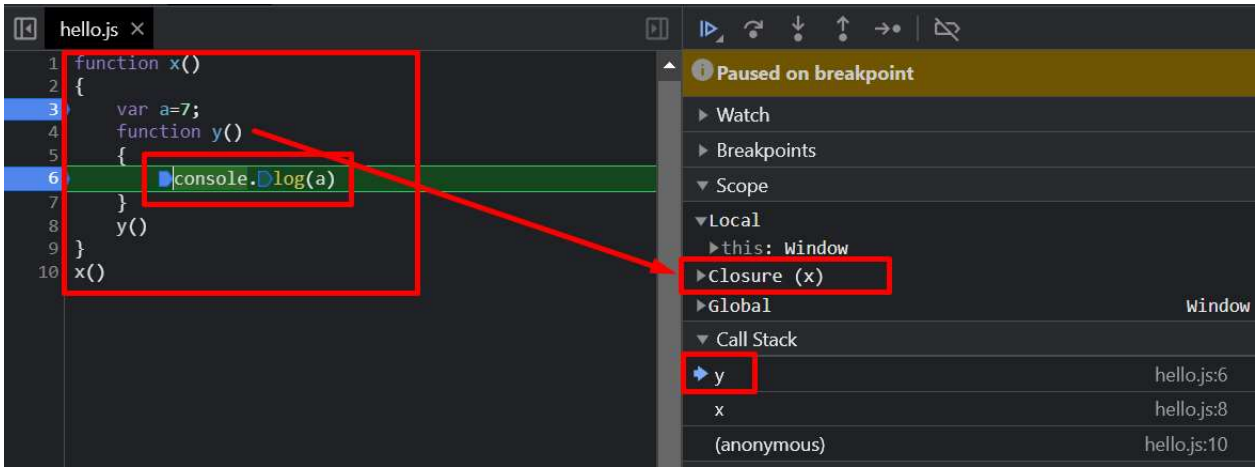


Placing the debugger

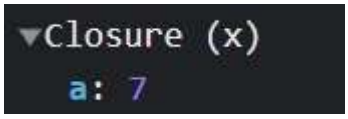


We see execution context for x function pushed into stack.

But if we see that when control flows moves to nested function



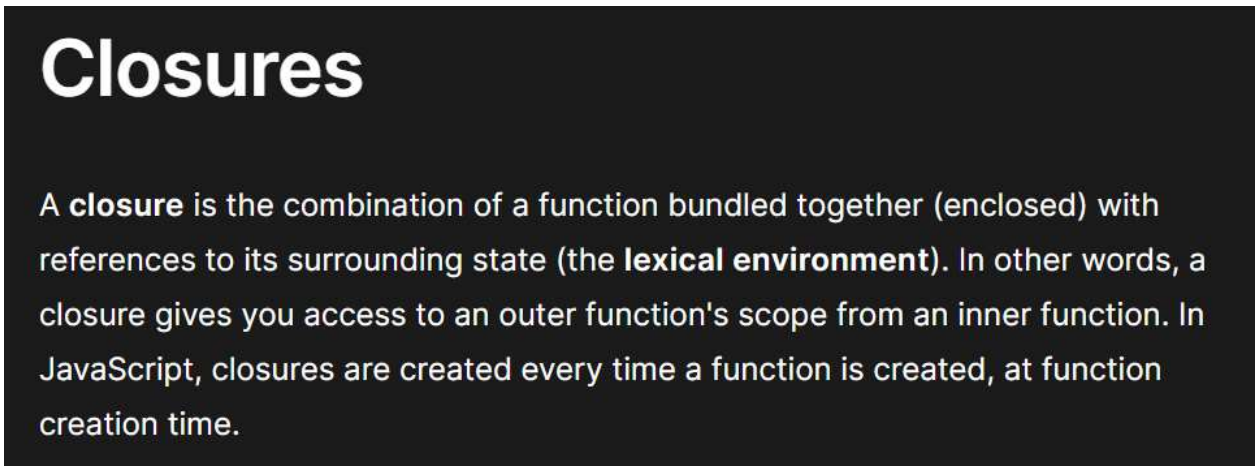
We see that nested function has Closure.



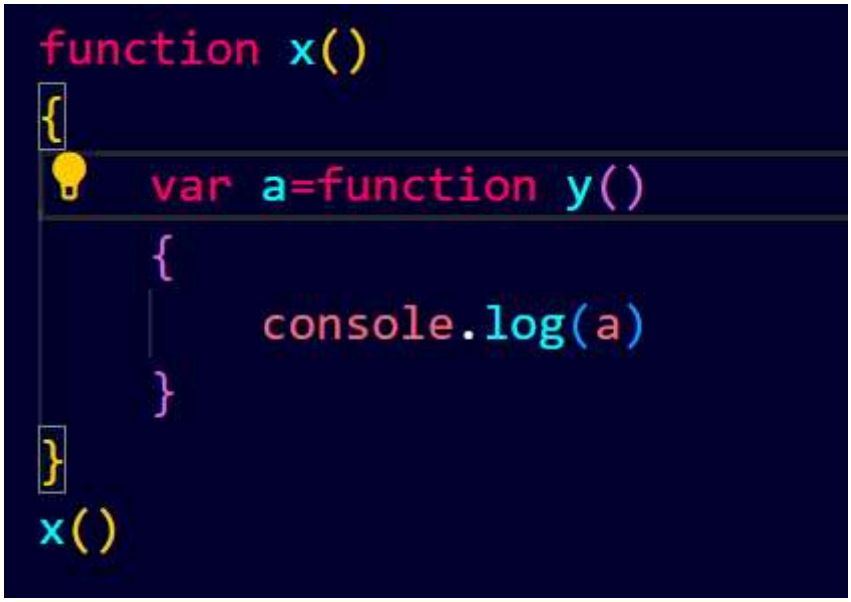
Closure-> function bind together with its lexical environment.(function along with its lexical scope)

function y forms closure with variables inside function x,i.e. with its lexical environment of its parent.

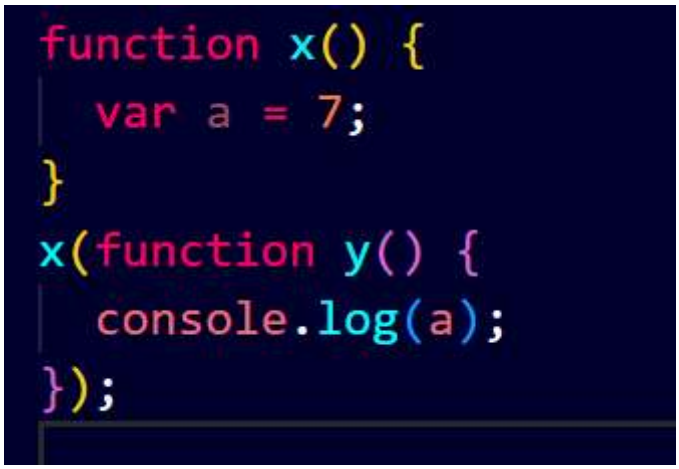
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures



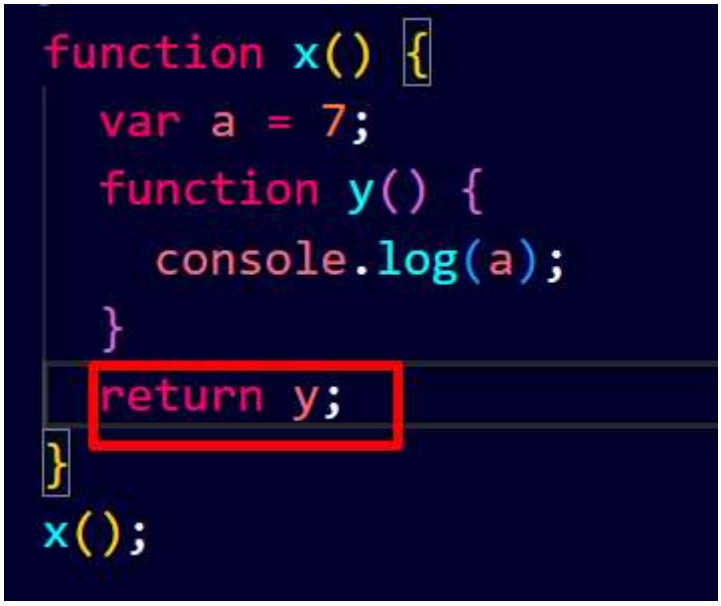
We can assign functions to variables as well in javascript



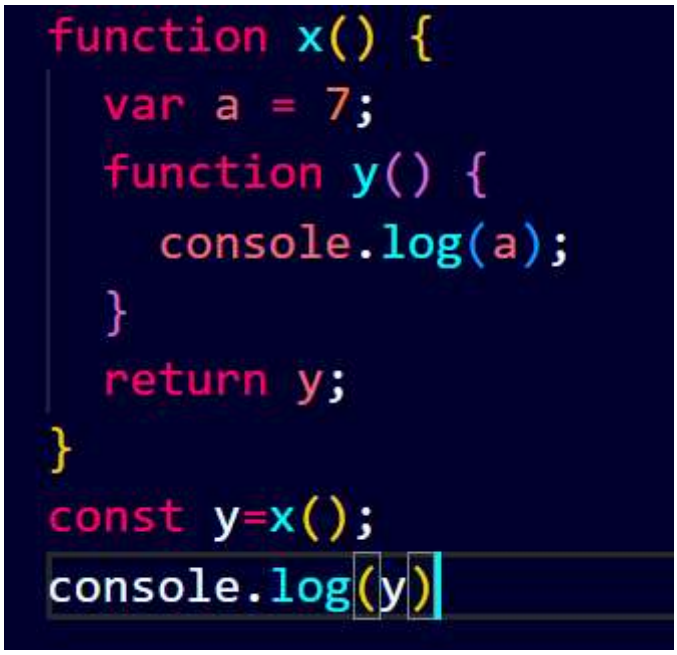
We can also pass functions as arguments.



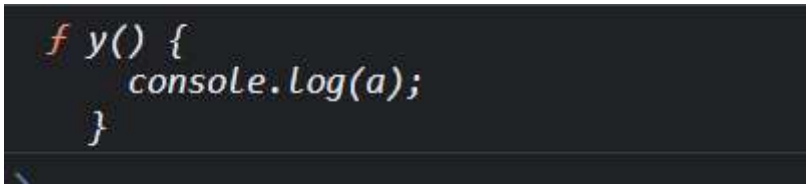
We can return function from a function also.



Now lets print function



Output



Our variable const y contains the function y. Function x is gone now from the call stack.So how will y behave now? If we call y will it print variable a? which was inside function x but function x has been popped off the call stack.

```
function x() {
  var a = 7;
  function y() {
    console.log(a);
  }
  return y;
}
const y=x();
y()
```

Output

```
7
```

So even though variable a was popped off with execution context of x, function y still remembered the value of variable a since it had performed closure with its parents lexical environment i.e. with a ,hence it remembered a and printed a in the console.

Functions maintain their lexical scope and remember where they actually existed even after being returned.

Some cool developers directly return function as shown below.

```
function x() {
  var a = 7;
  return function y() {
    console.log(a);
  }
}
const y=x();
y()
```

Now for code

```
1   function x() {
2     var a = 7;
3      function y() {
4        console.log(a);
5     }
6     a=100
7     return y
8   }
9   const y=x();
10  y()
```

Output

```
100
```

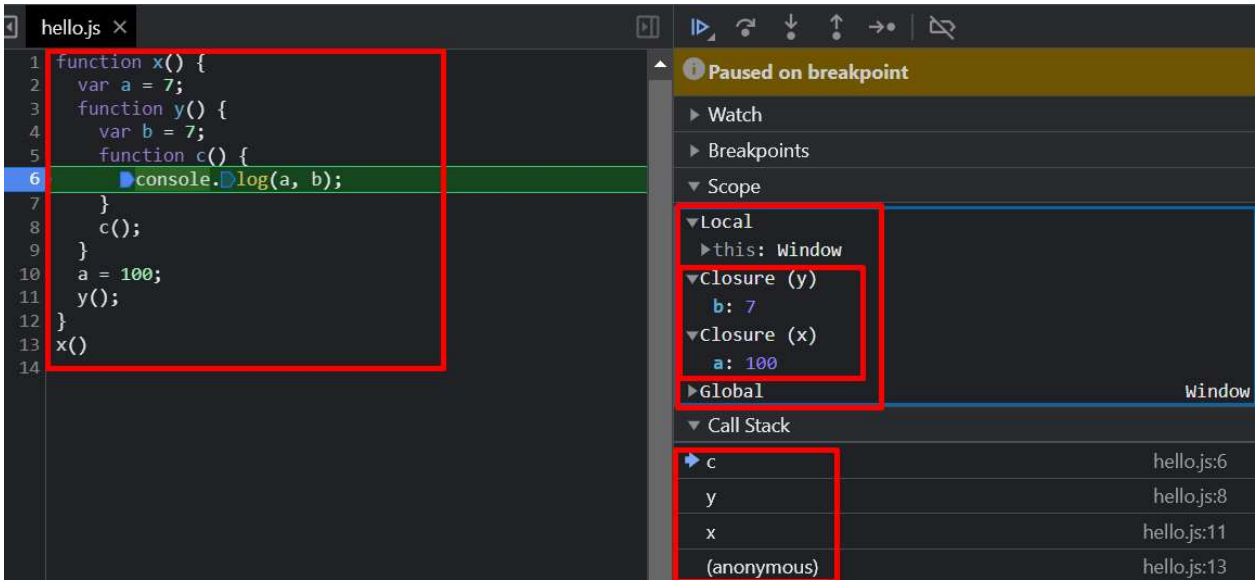
So how did it print 100 anot 7?

This is because reference to a persists and not reference to value 7 in the closure of function y.

Now let's take a deeply nested code for functions.

```
function x() {
  var a = 7;
  function y() {
    var b = 7;
    function c() {
      console.log(a, b);
    }
    c();
  }
  a = 100;
  y();
}
x()
```

Output



function c forms closure with scope of x and scope of y.

Output

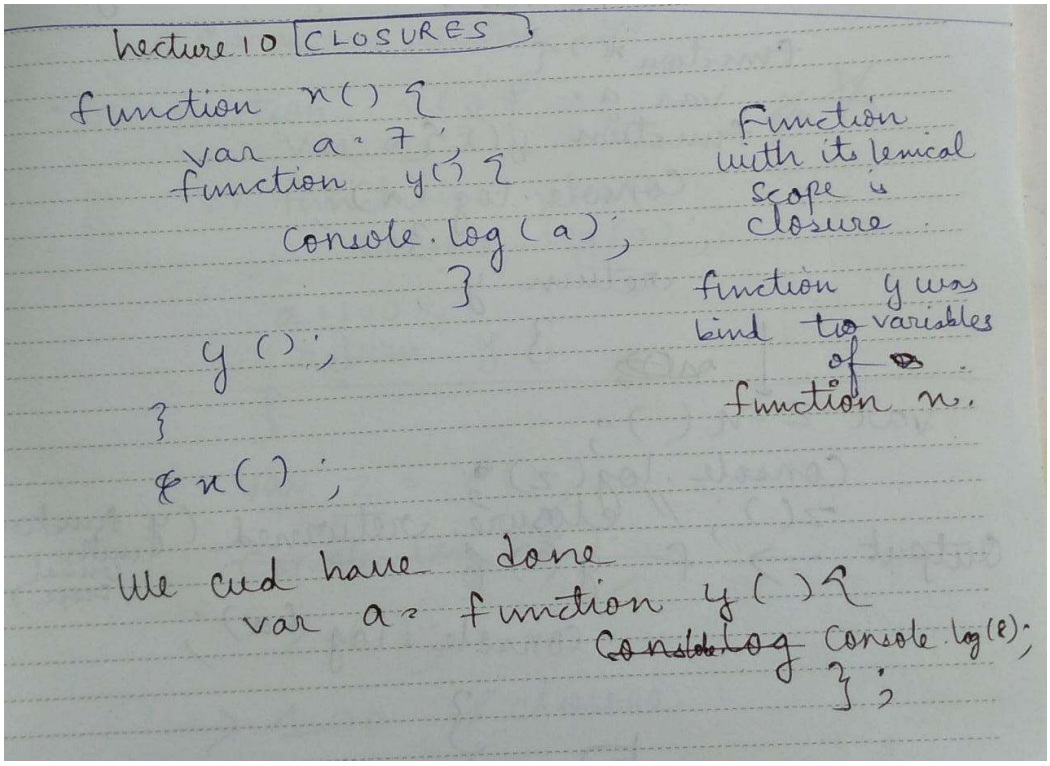
USES OF CLOSURES→

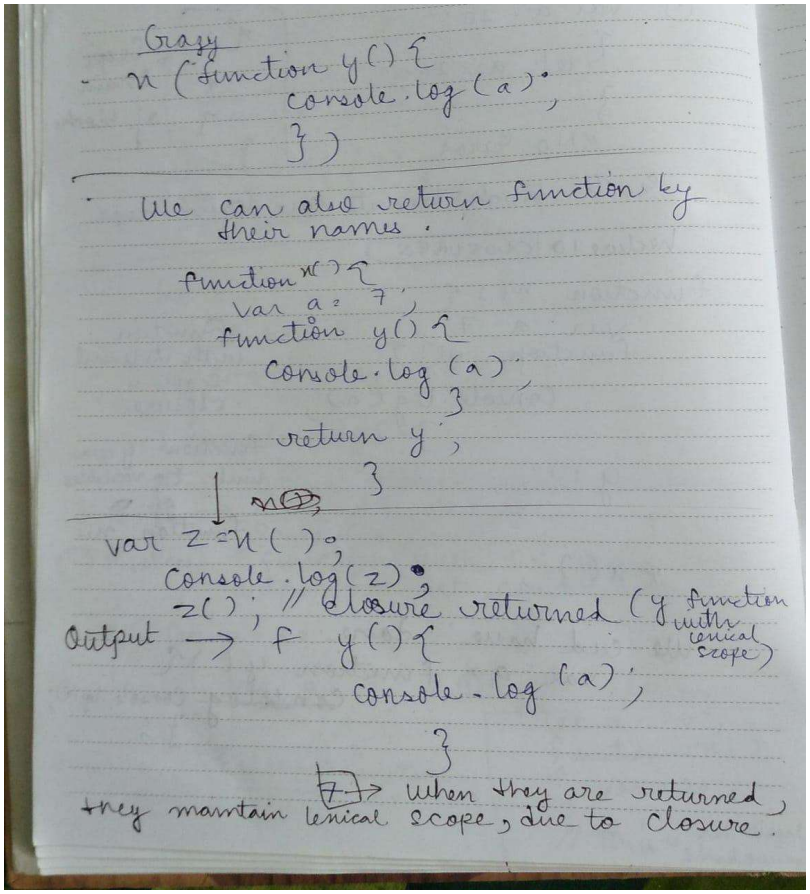
Uses of Closures:
- Module Design Pattern:
- Currying
- Functions like once
- memoize
- maintaining state in async world
- setTimeouts
- Iterators
- and many more...

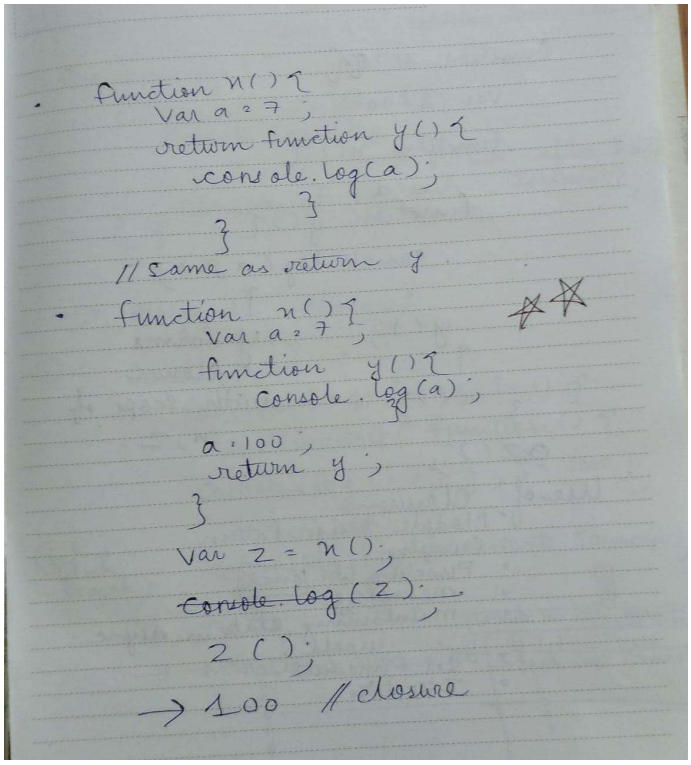
Disadvantages of Closure:

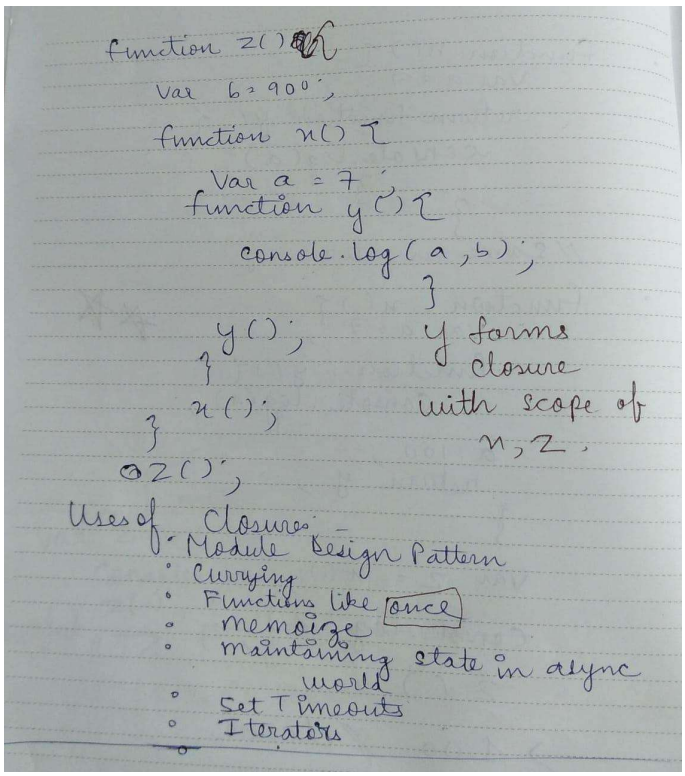
- **Over consumption of memory**
- **Memory Leak**
- **Freeze browser**

HANDWRITTEN NOTES






function x() {
var a = 7;
return function y() {
console.log(a);
}
}
// same as return y
function x() {
var a = 7;
function y() {
console.log(a);
}
a = 100;
return y;
}
var z = x();
console.log(z);
z();
→ 100 // closure


function z() {
var b = 900;
function x() {
var a = 7;
function y() {
console.log(a, b);
}
y();
}
x();
}
z();
y forms closure with scope of x, z.
Uses of Closures -
1. Module Design Pattern
2. Currying
3. Functions like once
4. memoize
5. maintaining state in async world
6. Set Timeouts
7. Iterators

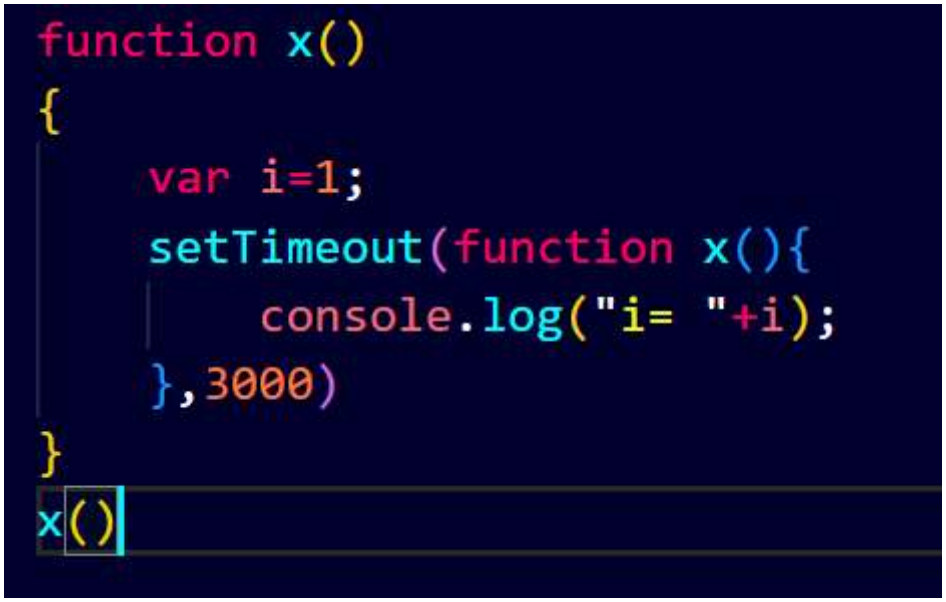
CONCLUSION

Closure Conclusion :Function bundled with its lexical environment is known as a closure. Whenever function is returned, even if its vanished in execution context ,still it remembers the reference it was pointing to.

---

setTimeout + Closures Interview Question 🔥 | Namaste 🙏 JavaScript Ep. 11

Youtube Link-https://youtu.be/eBTBG4nda2A



We see that when outer function x is popped off the call stack, setTimeout calls callback function x after 3 seconds to print value of i as 1 even when execution context of x and global execution context were popped off.

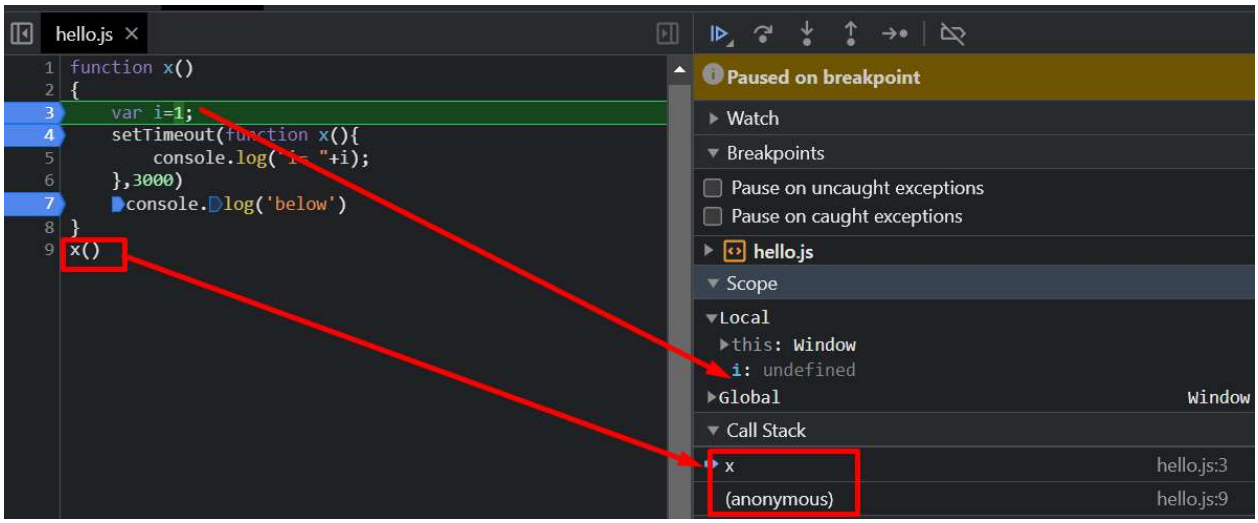
Time tide and JS waits for none.

```
function x()
{
    var i=1;
    setTimeout(function x(){
        console.log("i= "+i);
    },3000)
    console.log('below')
}
x()
```

So for above code output is

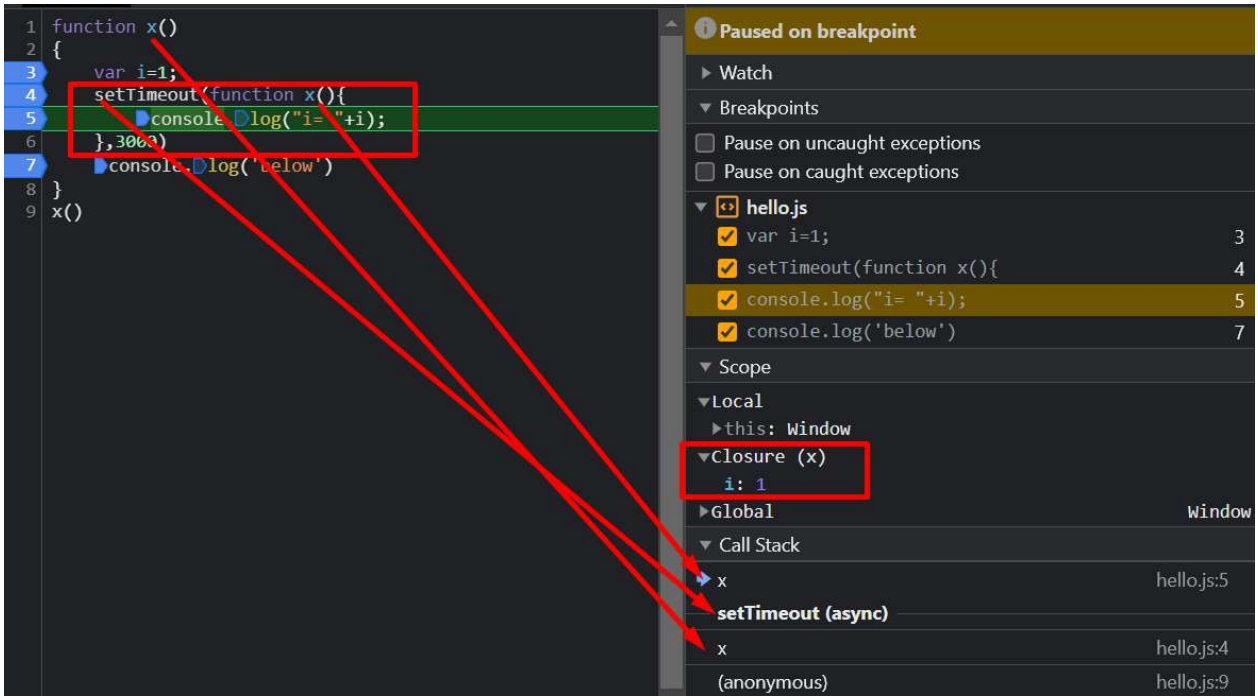
```
below
i= 1
```

Placing a debugger we see.

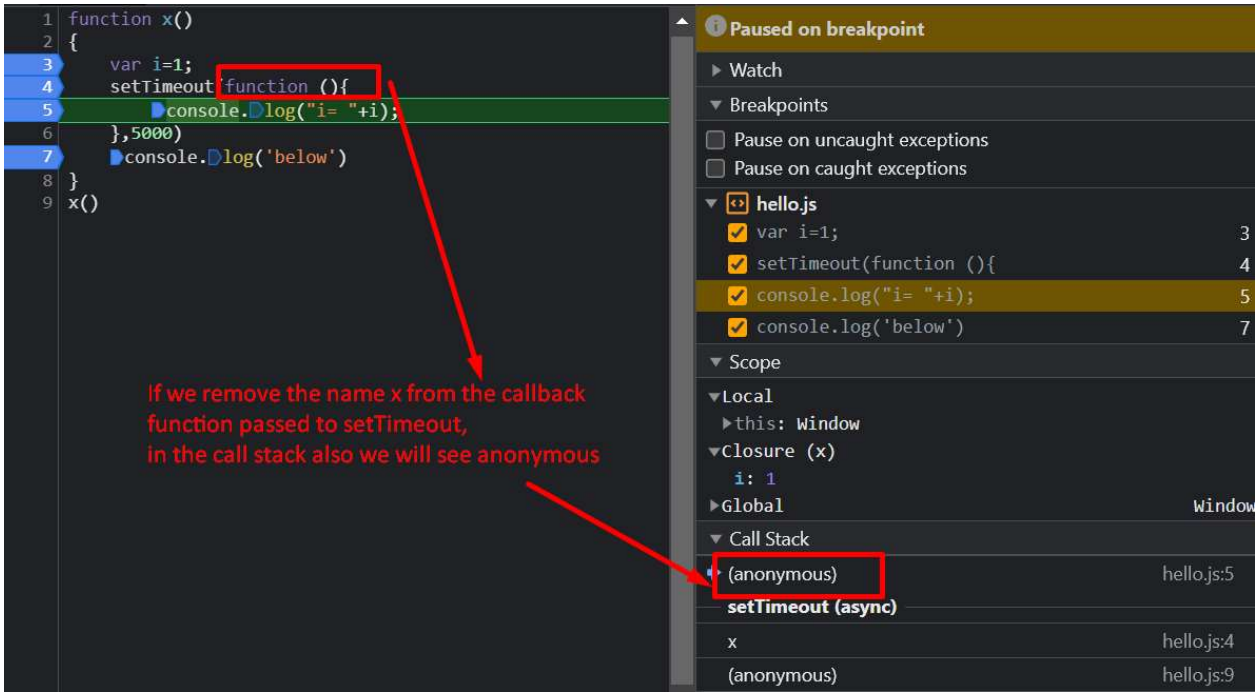


Debugger jumps from line 4 to line 7 prints below then jumps back to line 5.

Callback function x inside setTimeout forms a closure with scope of outer function x hence callback function x remembers reference to i even when function x has been popped off.



setTimeout() takes a callback function and attaches a timer to it and stores it in the web APIs environment. When timer expires, JS puts the function into the callback queue. When event loop sees the call stack empty,it puts the callback function from callback queue back into the call stack to be executed.(Discussed in event loop video https://youtu.be/8zKuNo4ay8E).



i=1 is logged when call stack is empty.

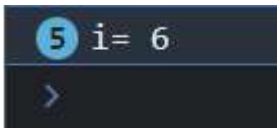
Now,

Print 1 after 1 seconds,2 after 2 seconds,3 after 3 seconds and so on..

We might think to do the below

```
function x()
{
    for(var i=1;i<=5;i++)
    {
        setTimeout(function (){
            console.log("i= "+i);
        },i*1000)
    }
}
x()
```

Output



6 is printed 5 times.

Wrong output

The callback function inside setTimeout forms a closure with the lexical scope of outer function x and remembers the reference to i.

So each time,setTimeout creates a copy of this callback function,attaches a timer and also remembers reference to i. So all 5 copies of callback functions with their respective timers point to the same reference because the environments for all of these functions are same.

JS will not wait for timer to expire. It will quickly register the 5 copies of callback functions. Since the loop was constantly running, i becomes ultimately 6. When timer expires all of them see the value of i as 6 and hence all 5 copies of callback functions print 6.

```
var a=2;let b=3
function x()
{
    var c=10
    let d=90
    for(let i=1;i<=5;i++)
    {
        setTimeout(function (){
            console.log("i= "+i);
        },i*1000)

    }
}
x()
```

Here the output comes i=1 2 3 4 5 as expected

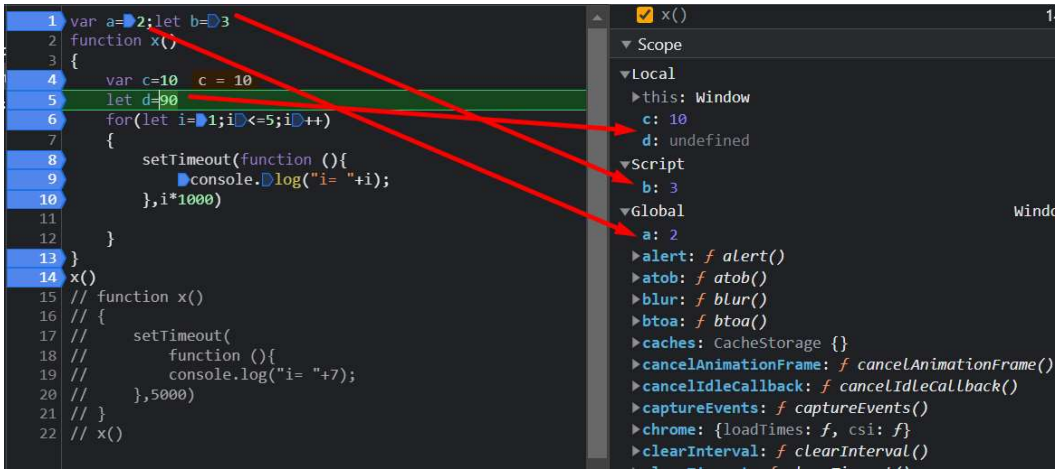
let forms a block scope.

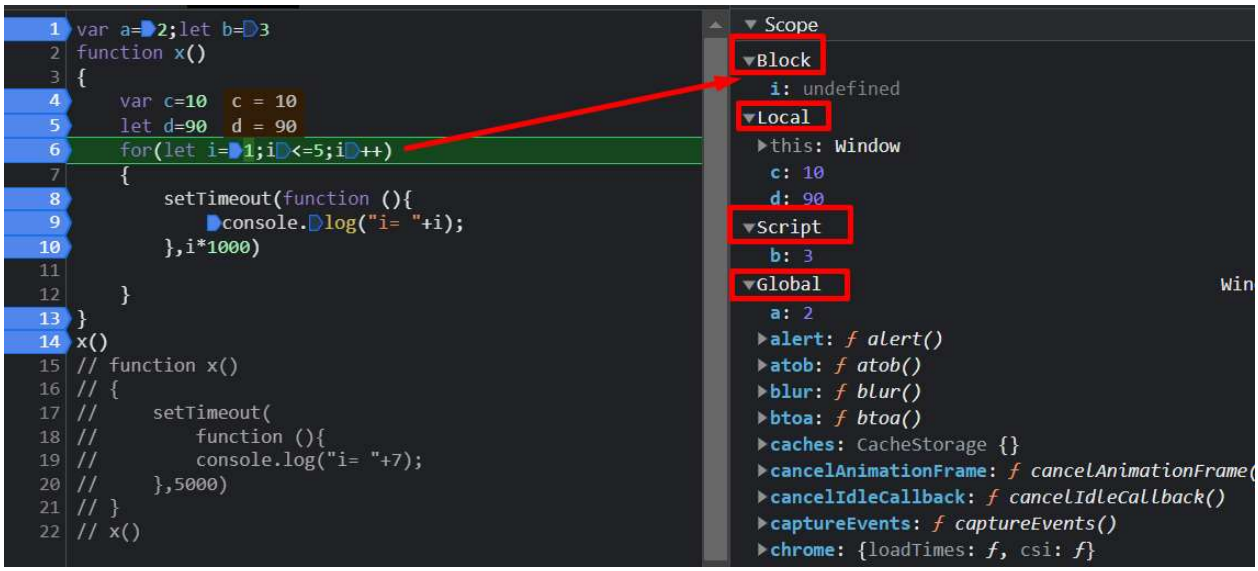
So in each iteration,the callback function registers that value of i which is currently in the loop,that is,value which is in that block.

In the next iteration, altogether a different block memory area is created for the block .As block memory due to previous iteration was deleted as soon as that previous block ended.So in each iteration,previous block memory was destroyed and new block memory was created with new value i which got incremented.The block memory creation and deletion happens very fast for us to notice. For each iteration, the i is a new variable altogether(new copy of i). Everytime setTimeout is run, the inside function forms closure with new variable i.

Hence each time,callback function gets different value from block.

Output





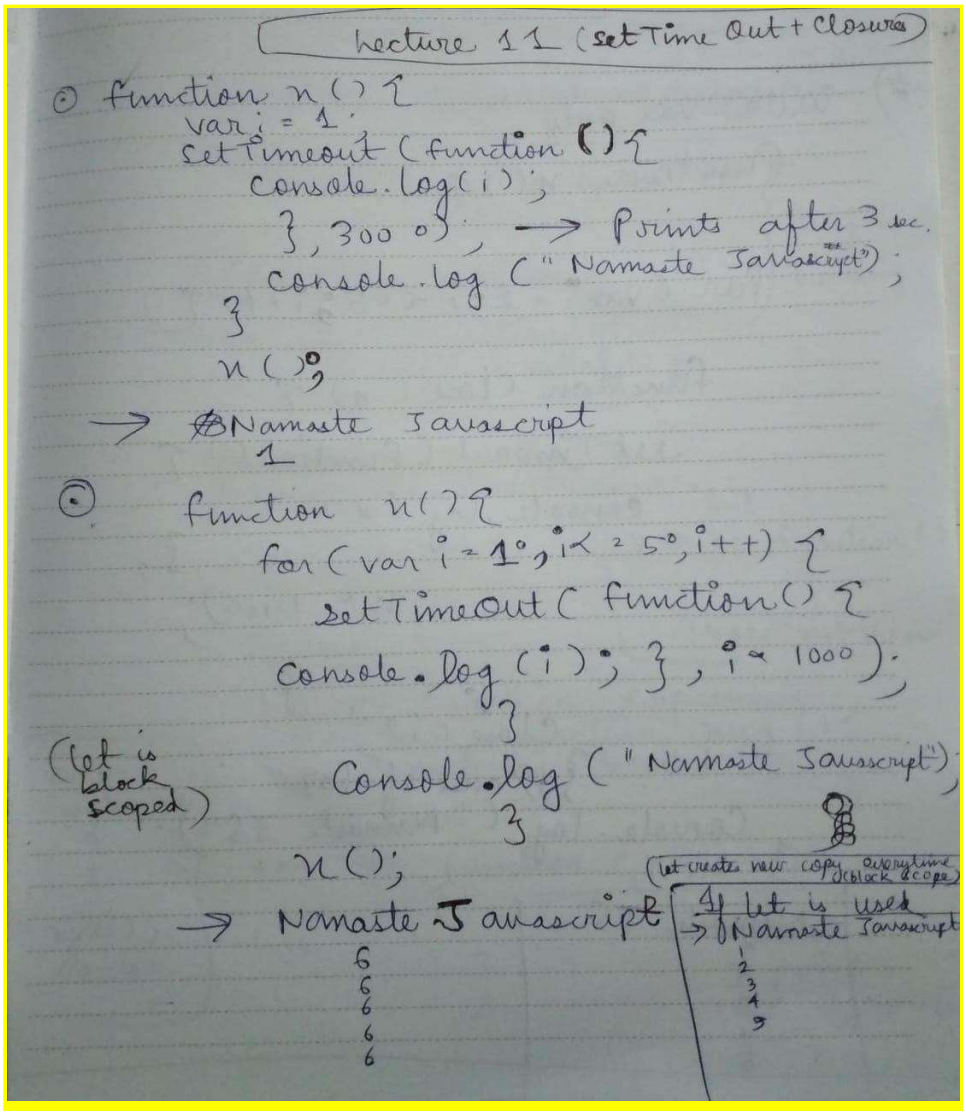
But what if we needed to do a program using var?

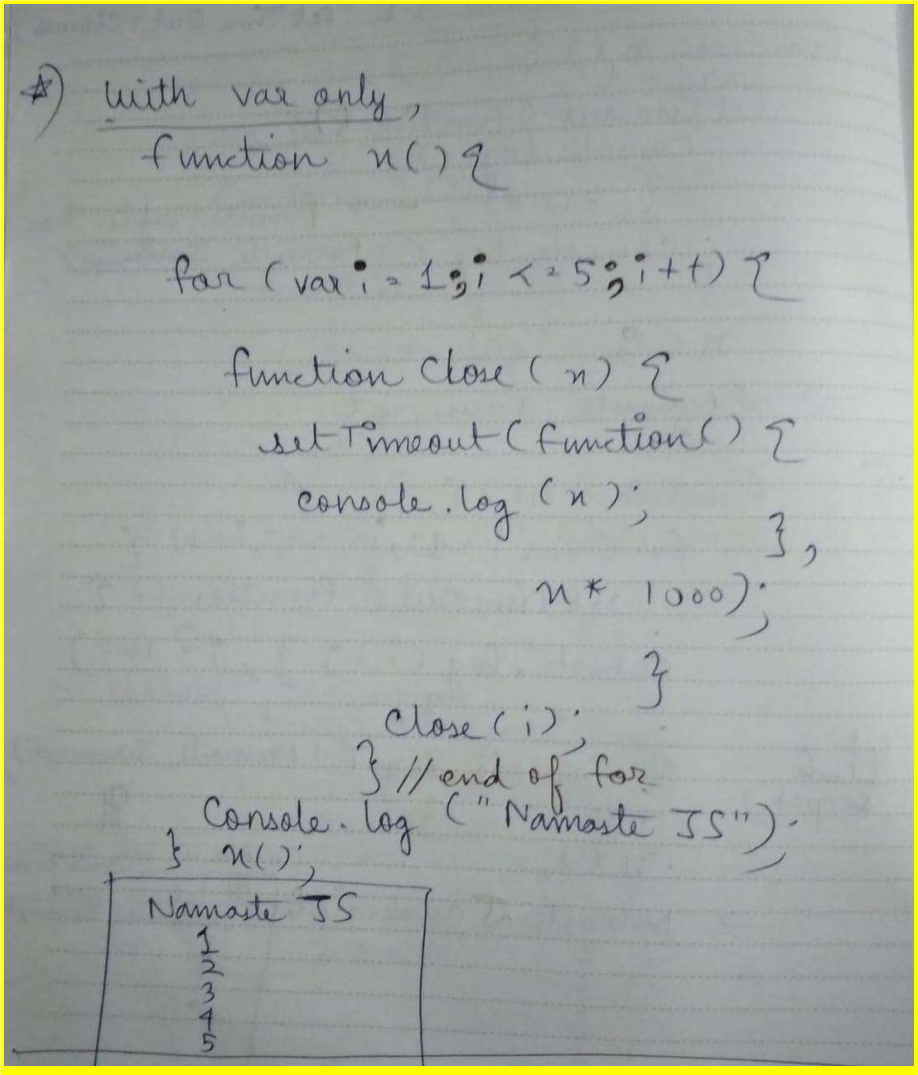
```
var a=2;let b=3
function x()
{
    var c=10
    let d=90
    for(var i=1;i<=5;i++)
    {
        function call(i)
        {
            setTimeout(function (){
                console.log("i= "+i);
            },i*1000)
        }
        call(i)
    }
}
x()
```

```
i= 1
i= 2
i= 3
i= 4
i= 5
```

Now callback function inside setTimeout forms a closure with function call.Hence even if call function is popped off the stack,the registered callback in API environment will remember the reference to its environment.

Handwritten Notes





SUMMARY

1. setTimeout stores the function in a different place and attached a timer to it, when the timer is finished it rejoins the call stack and executed.

2. Without closure the var reference gives the latest value as it does not retain the original value but rather has the reference so any update in value after timeout will be shown.

3. If we use let/const because they have block scope, every time a new copy of variable is attached, thus this can be done without closure.

(to be experimented upon)

CRAZY JS INTERVIEW 🤯ft. Closures | Namaste 🙏 JavaScript Ep. 12

Youtube Link-https://youtu.be/t1nFAMws5FI

What is a closure?

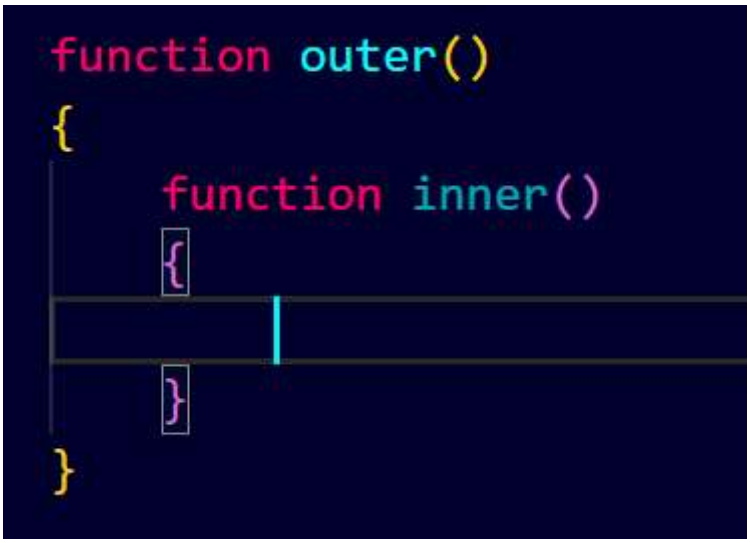
- **function with reference to outer environment together forms a closure**
- **closure is combination of function and its lexical scope bundled together**

Explain a little bit more about it.

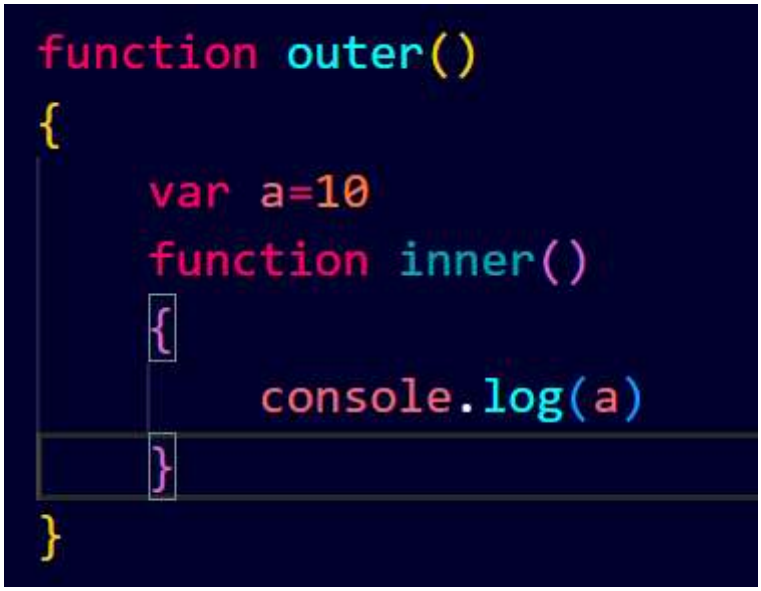
- **each and every function in javascript has access to its outer lexical environment,that is,it has access to the variables and functions present in the environment of its parent**
- **Even when this function is executed in some other scope and not in its own original scope ,it still remembers its outer lexical environment where it was originally present in the code.**

Give an example.

-



The inner function has access to outer environment.



The function inner+lexical outer environment is known as closure.

Function inner forms a closure with environment of its parent.

Now let's return inner function also. And try to access it from somewhere else then the function inner will still remember what the value of a was.

```
function outer()
{
    var a=10
    function inner()
    {
        console.log(a)
    }
    return inner
}
outer()()
```

It still prints 10.

```
10
```

So even if we call inner from outside,it still remembers the value of a.

<u>What are the 2 parentheses for?</u>

```
outer()()
```

It is like calling the inner function.

Simply calling outer by → `outer()` **returns us inner function so we call it again by using another parentheses.**

We can modify it this way→

```
var inner=outer()
inner()
```

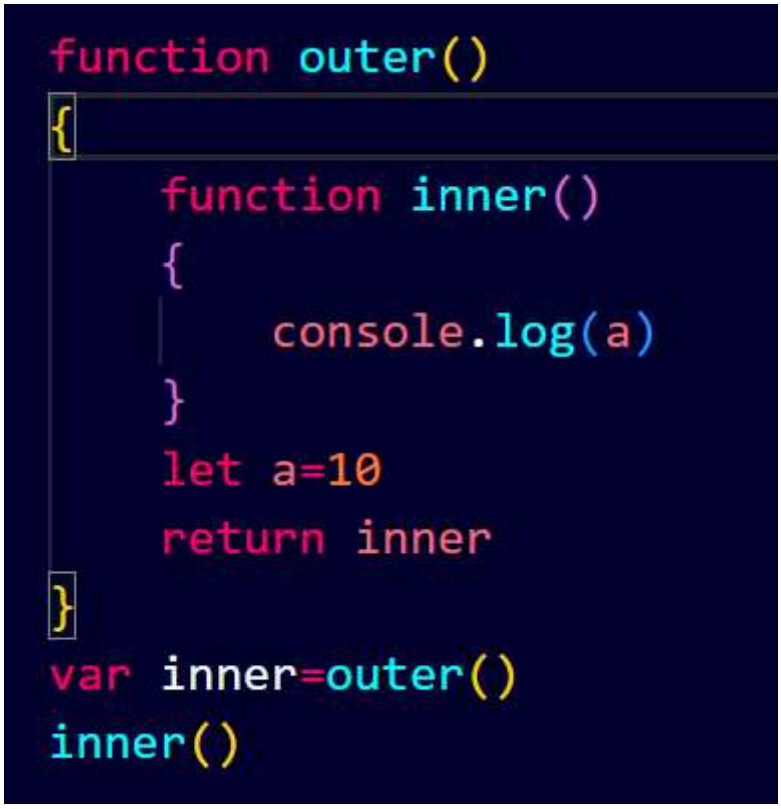
.

Now, what if we move var a just before return inner and just after function inner(){} ends? Will it still form a closure?

```
28  function outer()
29  {
30      function inner()
31      {
32          console.log(a)
33      }
34      var a=10
35      return inner
36  }
37  var inner=outer()
38  inner()
```

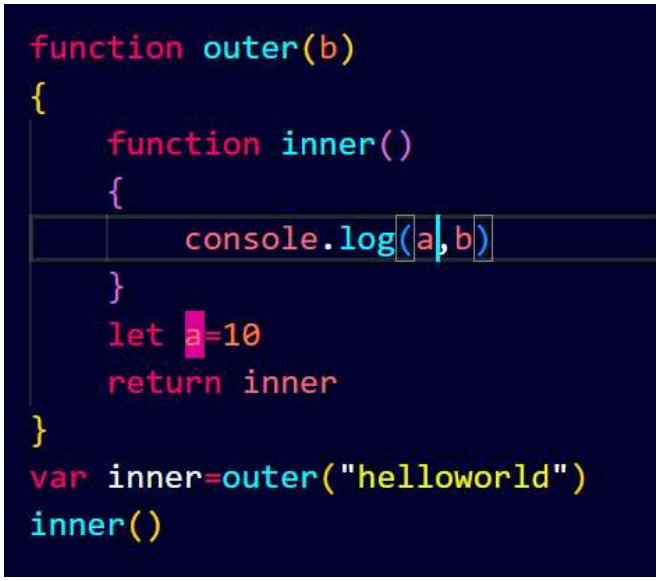
Answer-> It will still form a closure. If we run it ,it will still print 10. Even if we place var a=10 before an inner function or after an inner function,it is still within the outer function,that is the Local space of outer will always have a,irrespective of the fact that a is declared below or above. Hence inner will also form a closure

Now,if we make var to let will there be any change?



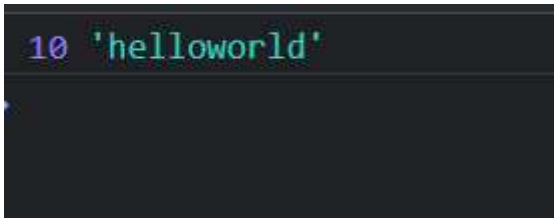
Answer-> inner still forms a closure even though let is block scoped.

Qs-> What if outer has a parameter b? Then what will happen?



So,answer is it behaves the same way, inner still forms closure with the outer environment and b is present in the outer environment. So inner remembers b somewhere else as well.

Output



Qs-> What if outer is nested inside another function? Will inner it still form closure with the environment of that function?

```
function outest()
{
    var c=20
    function outer(b)
    {
        function inner()
        {
            console.log(a,b,c)
        }
        let a=10
        return inner
    }
}
var inner=outer("helloworld")
inner()
```

Output

SO yes,inner() will still remember c as inner forms closure with outest() along with outer().

Lets properly write the code and get the output.

```
function outest()
{
    var c=20
    function outer(b)
    {
        function inner()
        {
            console.log(a,b,c)
        }
        let a=10
        return inner
    }
    return outer
}
var inner=outest()("helloworld")
inner()
```

calling outer(b)

```
10 'helloworld' 20
>
```

outest() returns outer function so apply another bracket () to call outer function as well on same line. It returns inner function. Now we call inner on next line.

SO ABOVE WAS THE EXAMPLE OF SCOPE CHAIN WITH CLOSURES.

Qs->What will happen if we have a conflicting global variable name like lets say let a=100?

```
function outest()
{
    var c=20
    function outer(b)
    {
        function inner()
        {
            console.log(a,b,c)
        }
        let a=10
        return inner
    }
    return outer
}
let a=100
var inner=outest()("helloworld")
inner()
```

So inner() remembers let a=10 inside the outer(b) function and not let a=100 in script memory area. 

is printed. If let a=10 was absent in parent lexical environment of inner then ,scope chaining would find in parent's parent lexical environment and let a=100 and 100 would be printed instead of 10. And if let a=100 was also not declared then Uncaught ReferenceError:a is not defined would have been printed.

<u>What are the Advantages of Closure?</u>

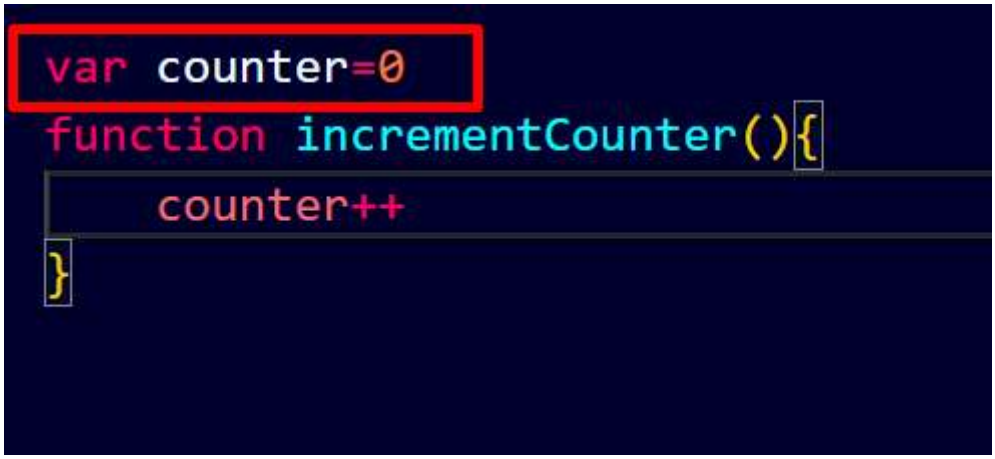
Function currying, module patterns, used in higher order functions like memoized ,

Helps us in data hiding and encapsulation.

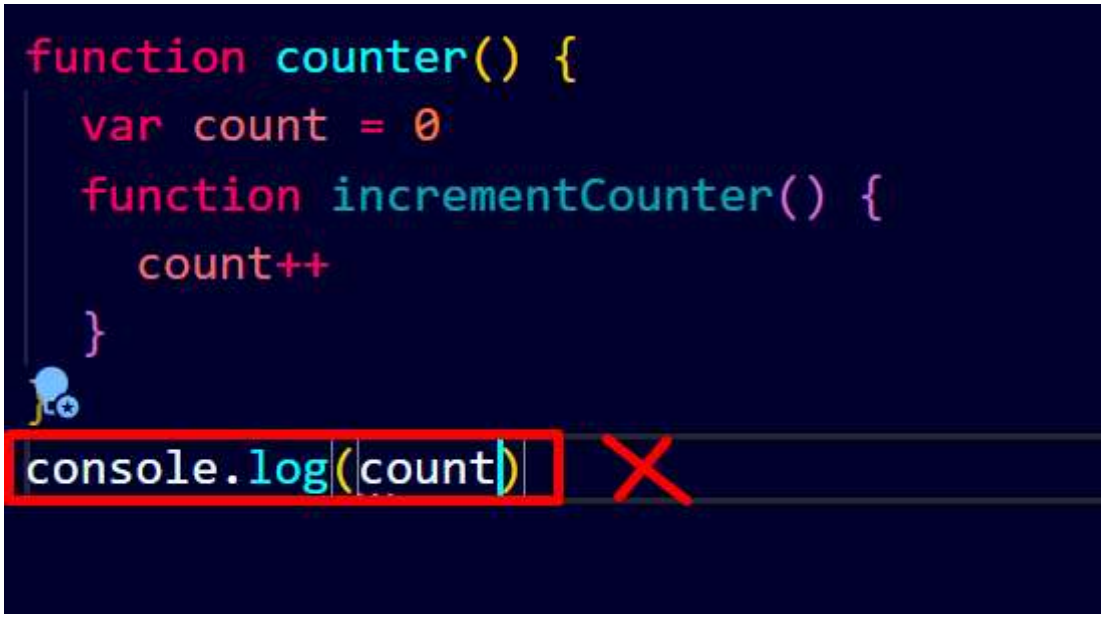
Qs- <u>Tell more about data hiding and encapsulation</u>

Lets say we have a variable holding some data. And we want to have some data privacy so that other pieces of functions cannot have access to that particular data, then it is data hiding.

Qs- Give example of it



Anybody can access the counter variable. So here data hiding comes into the picture.We want to ensure nobody else can modify that counter variable and we can only do so using increment counter function. We can do that using CLOSURES.



So nobody can access the count variable directly. We are thus hiding our data and having privacy over count. When we return the function it forms a closure with count.

```
function counter() {
  var count = 0
  return function incrementCounter() {
    count++
    console.log(count)
  }
}
var counter1=counter()
counter1()
counter1()
counter1()
```

So with the help of closure of incrementCounter() with count variable,we can access count even though count is actually hidden and cannot be accessed directly.

counter1 holds incrementCounter(). So when we call counter1, incrementCounter() actually remembers the variable count and increments count.when we call counter1 again incrementCounter() yet again remembers the previous incremented value of count variable and increments it further.

```
1
2
3
>
```

Qs->if there was var counter2=counter(), what would happen?

```
function counter() {
  var count = 0
  return function incrementCounter() {
    count++
    console.log(count)
  }
}
var counter1=counter()
counter1()
counter1()
counter1()
var counter2=counter()
counter2()
counter2()
counter2()
```

counter2 won't touch counter1 . It is a fresh one.In counter2() ,a new count will form another closure with incrementCounter() different from closure in counter1.

OUTPUT

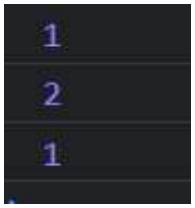
```
1
2
3
1
2
3
```

Qs->Is your code scalable in the sense when you need to create decrementCounter?

No,not scalable, we need to create constructor function and we should have separate functions for increment or decrement.

```
function Counter() {
  var count = 0
  this.incrementCounter=function() {
    count++
    console.log(count)
  }
  this.decrementCounter=function() {
    count--
    console.log(count)
  }
}
var counter1=new Counter()
counter1.incrementCounter()
counter1.incrementCounter()
counter1.decrementCounter()
```

We need to use new keyword for functional constructor.



What are the disadvantages of closure?

-overconsumption of memory

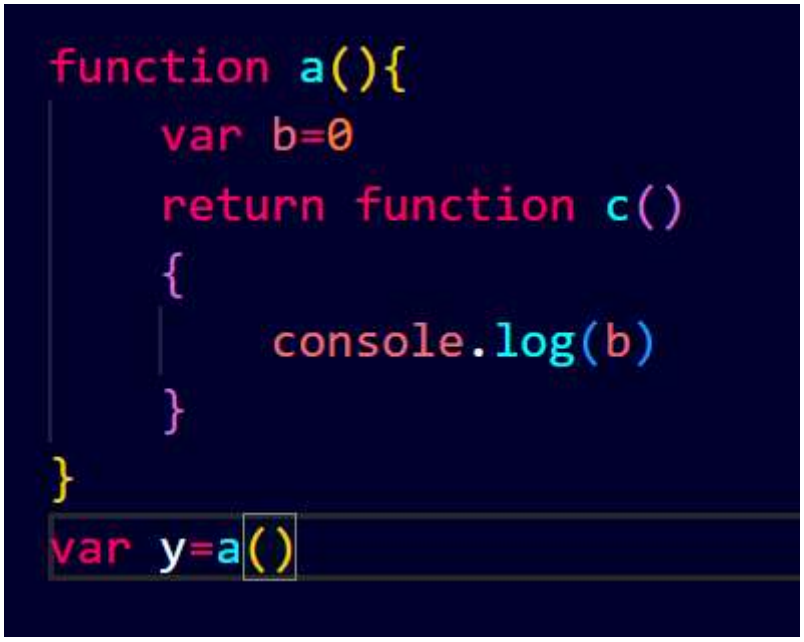
-those closed over variables are not garbage collected

-if not handled properly, browsers can be freezed and memory leaks can happen.

What is garbage collector and what does it do?

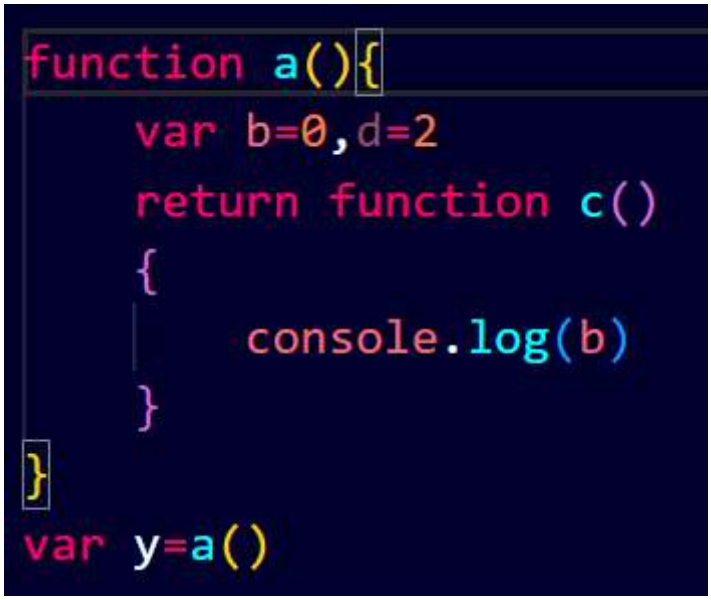
Garbage collector is like a program in browser or JS engine which frees up the unutilized memory. In C/C++ ,it is upto developers how we allocate or deallocate memory, but in high level programming languag most work is done by JS engine. Whenever there are unused variables or if variables ar unused, JS engine takes it out of memory.

How are closures and garbage collector related?

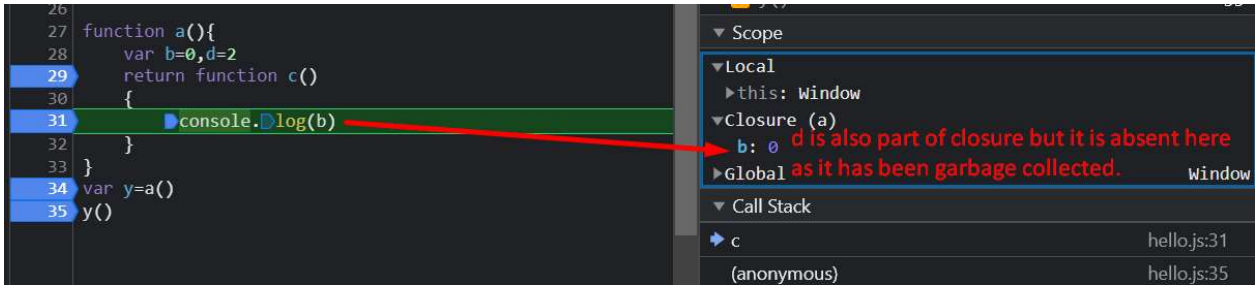


Here memory of b cannot be free as it forms closure with function c. function c is inside y. But V8 and modern browsers have smart garbage collection and they garbage collect these variables if they are unreachable.

Qs) What do you mean by smart ?



Here d will be smartly garbage collected . Though both b,d form closure with c; d has no usage inside c.



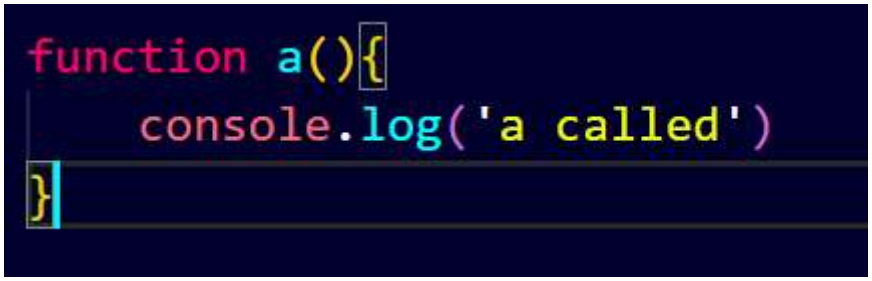
At that point if we want to log b, and log d

d should have been present but is actually absent.

FIRST CLASS FUNCTIONS 🔥ft. Anonymous Functions | Namaste JavaScript Ep. 13

Youtube Link-https://youtu.be/SHINoHxvTso

What is a function statement?(also known as function declaration)



What is a function expression?

```
var b=function(){
    console.log('b called')
}
```

Above 2 are ways to create a function.

Difference between these 2?

Hoisting.

Code

```
a()
b()

function a(){
    console.log('a called')
}
var b=function(){
    console.log('b called')
}
```

Output

```
a called                                                    hello.js:6
▶ Uncaught TypeError: b is not a function                   hello.js:3
    at hello.js:3:1
>
```

In memory creation phase a,b are present as keys in memory component.

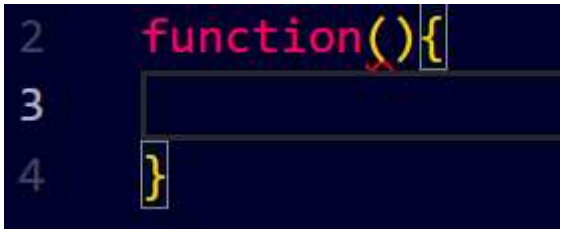
Value of a is the entire code for function a.

Value of b is undefined initially as we have used arrow function so b is treated like a variable. When we do variablename() it ought give us TypeError.

So until the flow reach var b=function() ,value of b will be undefined.And as we are calling b before var b=function() has been encountered we get TypeError.

<u>What is anonymous function?</u>

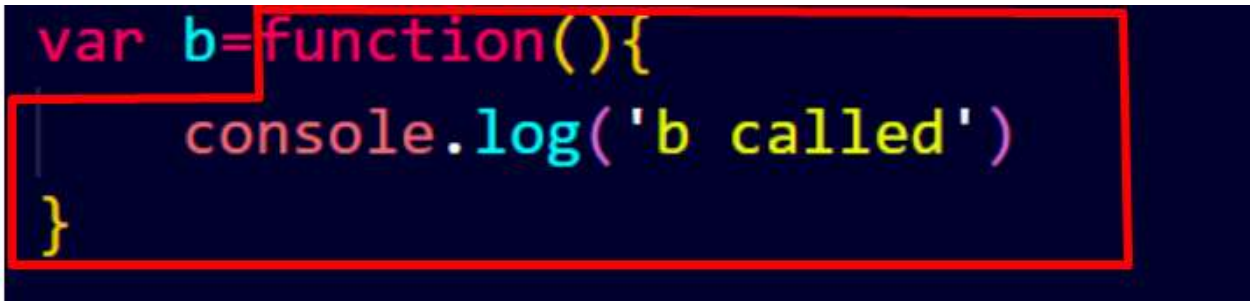
A function that has no identity.



Above is a syntax error. It looks similar to a function statement. But according to ECMAScript specification, a function statement should always have a name. So its an error.

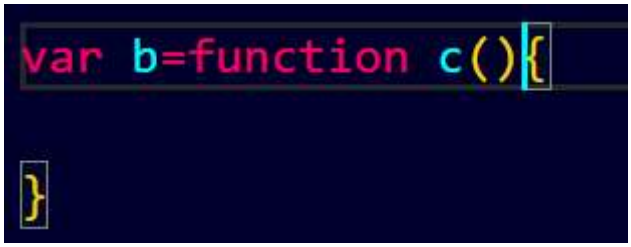
So where are anonymous function used ?

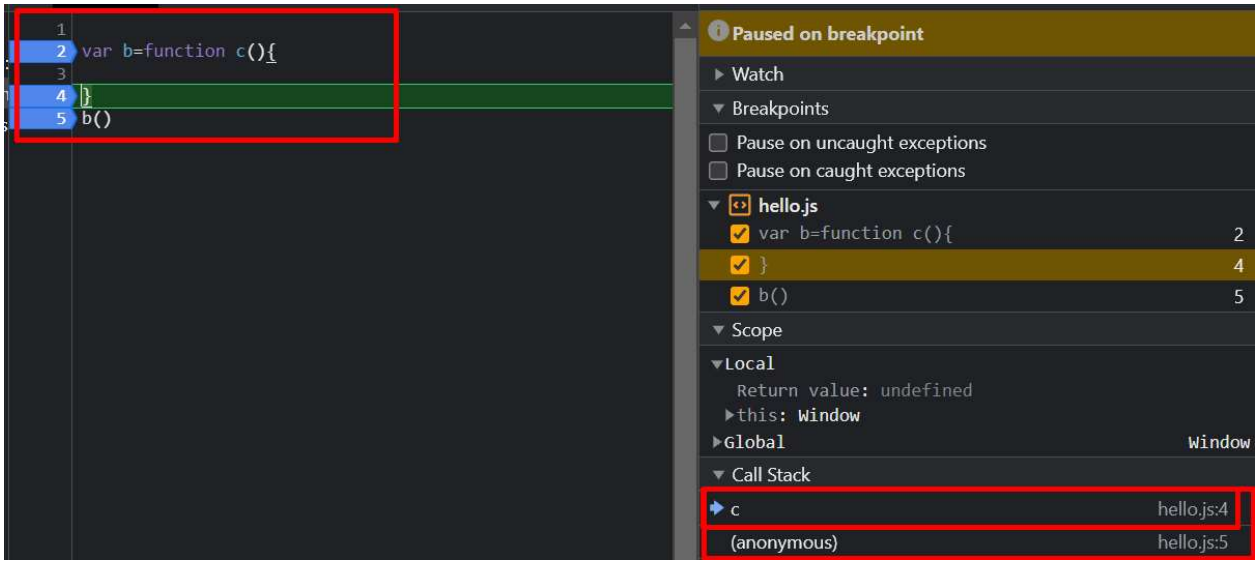
So anonymous functions are used where functions are used as values.



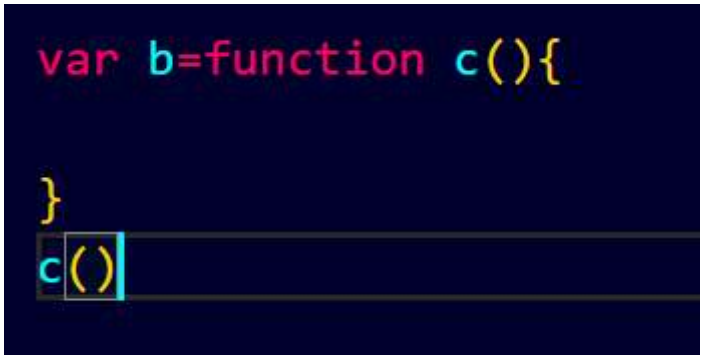
Here anonymous are assigned as value to variable.

<u>What are named function expression?</u>





Note- In above pic,in call stack,execution context for c is being created, but if we use c() for function calling then it results in error as seen below.



Reason - c is not a function in outer scope but it is created as a local variable.

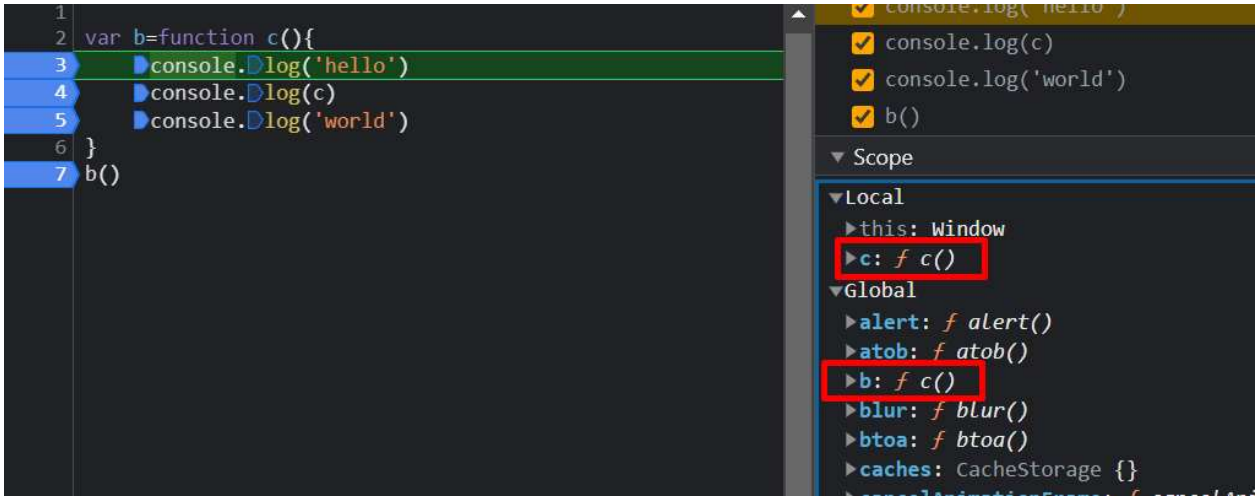
On the other hand if we access the c function inside it, we can access it.

```
var b=function c(){
    console.log('hello')
    console.log(c)
    console.log('world')
}
b()
```

Output

```
hello                                                    hello.js:3
ƒ c(){                                                   hello.js:4
    console.log('hello')
    console.log(c)
    console.log('world')
}
world                                                    hello.js:5
>
```

Using debugger we see→



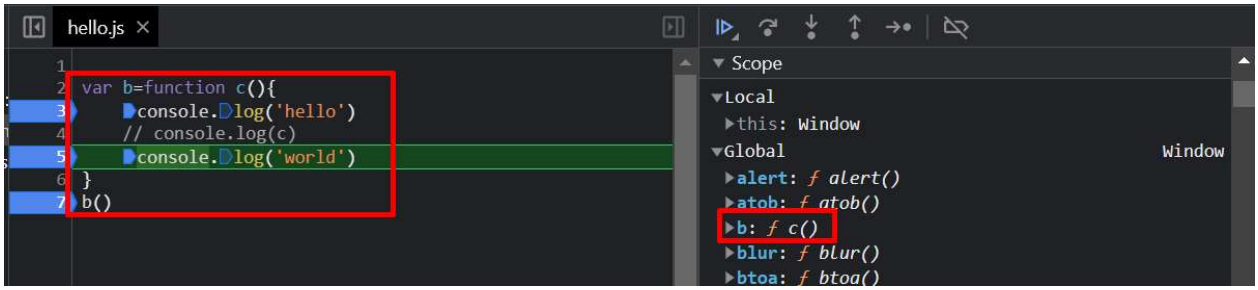
If we notice→ Inside Global we have b as key and function c as value.

While inside Local space of function → we have c as key and function c as value.

Now for an experiment,lets remove console.log(c)

```
var b=function c(){
    console.log('hello')
    // console.log(c)
    console.log('world')
}
b()
```

Using debugger



Notice that Local space does not contain c as key anymore.

And if we try to log c

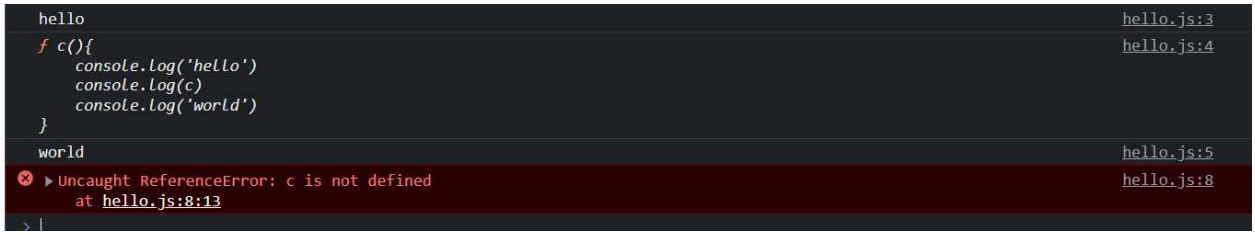
We see an error when we try to log c. Reason is since we did not use c inside function c(), c was not required hence it was garbage collected. So ,though the flow control was inside c,and we tried to log c,it resulted in error.

Now, find out the error in following code

```
var b=function c(){
    console.log('hello')
    console.log(c)
    console.log('world')
}
b()
console.log(c)
```

Error is we are trying to use c outside function c(). We saw that c is used as a key only inside the Local space.In Global space d is used as key. So outside we will get an error if we try to log c but no error if we log d.

OUTPUT



These were the corner cases of named function expression.

<u>What is difference between parameter and argument ?</u>

```
function c(parameter)
{

}
var argument=5;
c(argument)
```

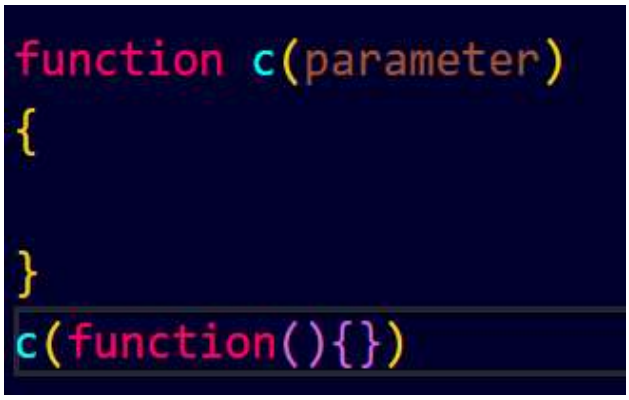
The parameter is local variable inside the function. We cannot access parameter outside.They can be called identifiers as well as labels.

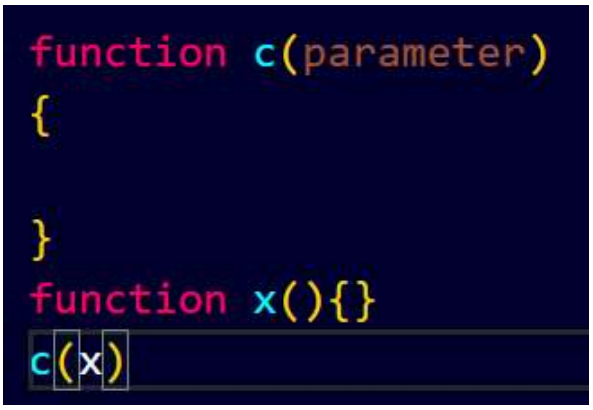
Values which we pass are arguments.

The labels,identifiers which gets those values are parameters.

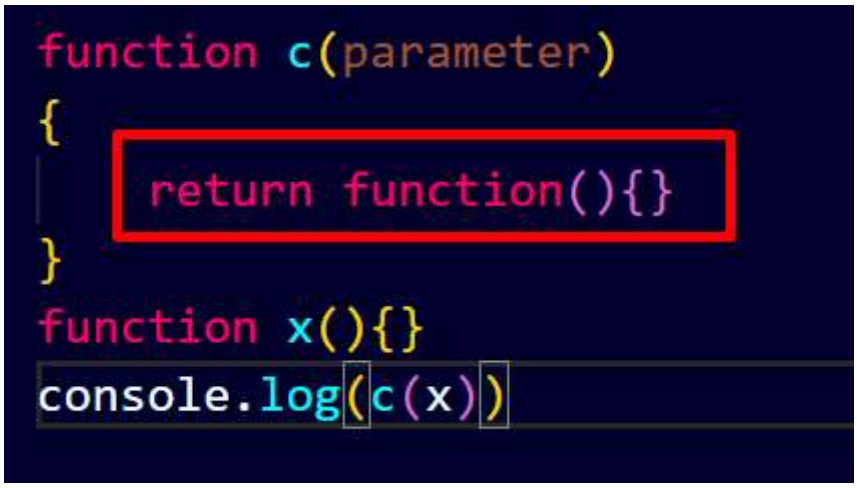
<u>What are first class functions ?</u>

Functions are very beautiful. We can pass functions inside functions as arguments.



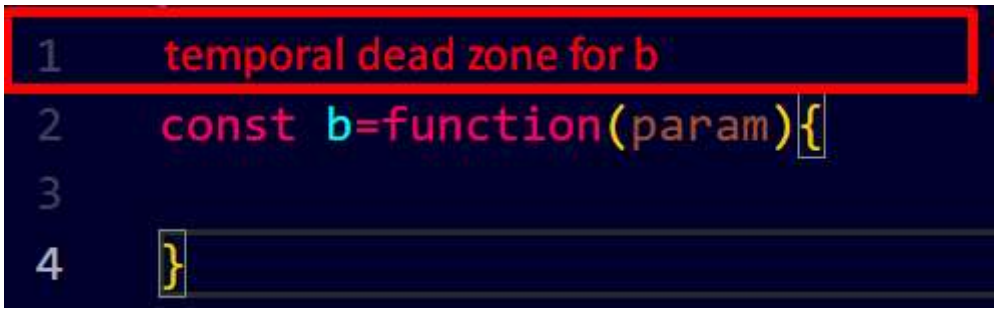


We can also return a function.



First class functions is the ability to use functions as values by passing them as argument,returning them from function,assign a function to a variable.

FUNCTIONS ARE FIRST CLASS CİTİZENS. It is same term as first class functions.



Coming to arrow functions, it has been part of ES6. ECMAScript 2015.

SUMMARY-

1. What is a Function Statement ?

A. A normal function that we create using Naming convention. & By this we can do the Hoisting.

```
For Ex  -  function xyz(){

                console.log("Function Statement");

           }
```

2. What is Function Expression ?

A. When we assign a function into a variable that is Function Expression. & We can not do Hoisting by this because it acts like variable.

```
For Ex - var a = function(){

                console.log("Function Expression");

          }
```

3. What is Anonymous Function ?

A. A Function without the name is known as Anonymous Function. & It is used in a place where functions are treated as value.

For Ex - function(){

}

4. What is Named Function Expression ?

A. A function with a name is known as Named Function Expression.

For Ex - var a = function xyx(){

console.log("Names Function Expression");

}

5. Difference b/w Parameters and Arguments ?

A. When we create a function & put some variables in this ( ) that is our Parameters.

For Ex - function ab(param1, param2){

console.log("

}

& When we call this function & pass a variable in this ( ) that is our Arguments

For Ex - ab(4, 5);

6. What is First Class Function Or First class citizens?

A. The Ability of use function as value,

* Can be passed as an Argument,
* Can be executed inside a closured function &
* Can be returned.

For Ex - var b = function(param){

return function xyz(){

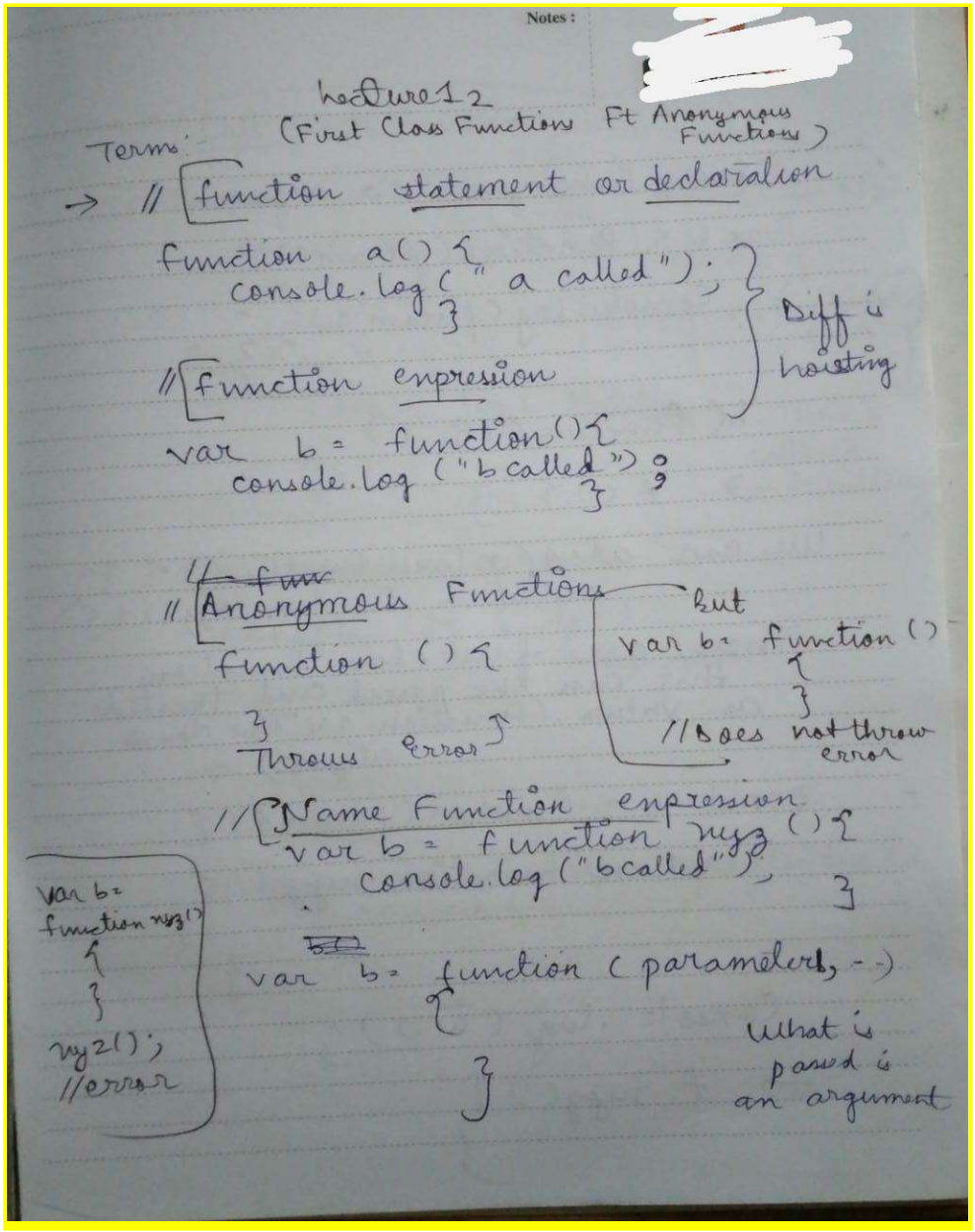
console.log(" F C F ");

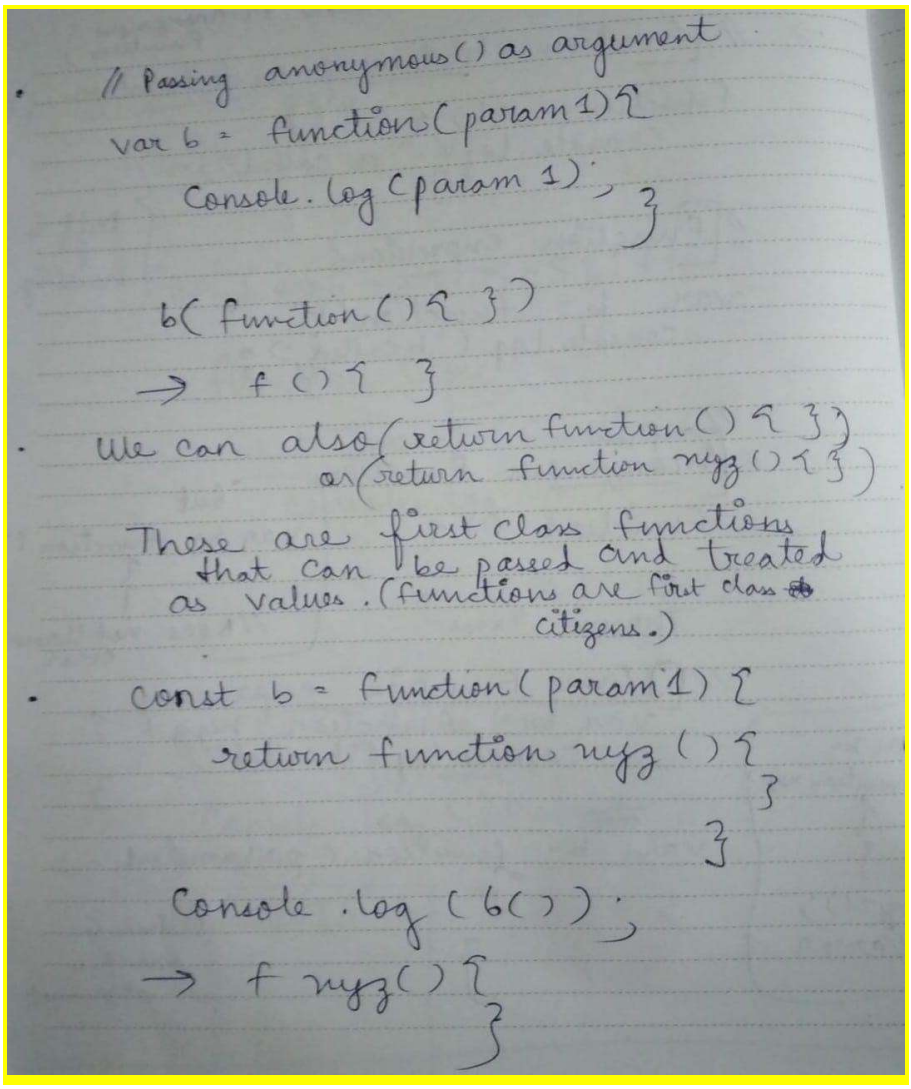
}

}

7. Function are heart of JS. They are called first class citizens or first class functions because they have the ability to be stored in the variables, passed as parameters and arguments. They can also be returned in the function.

Hand Written Notes





Callback Functions in JS ft. Event Listeners 🔥 | Namaste JavaScript Ep. 14

Youtube Link- https://youtu.be/btj35dh3_U8

What is a callback function?

When you pass a function to another function,the function that you pass is known as callback function.

Callback functions give us access to an asynchronous world in a synchronous single threaded language. Due to callbacks we can do async things inside JS.

```
function x(){

}
x(function y(){

})
```

y is the callback function.

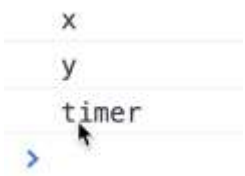
Why the name callback? Because we call the function sometime else in your code.

```
function x(y){

}
x(function y(){

})
```

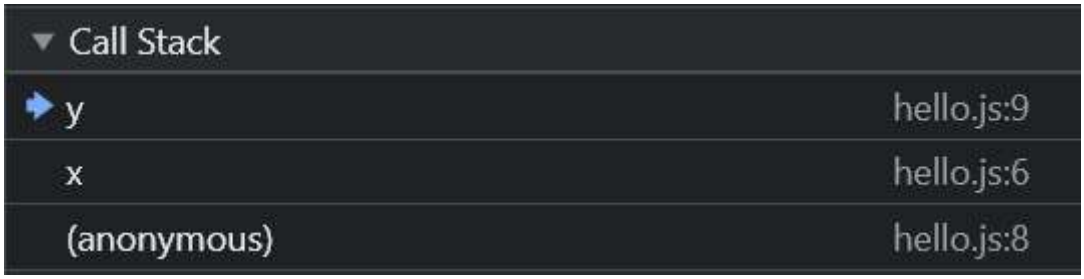
We simply passed y to x. Now it is upto x when it wants to call this y.

```
setTimeout(function () {
  console.log("timer");
}, 5000);

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

Here, 1st thing is register of the callback function and a timer will be attached to it. JS won't wait for setTimeout to execute. It will quickly execute the below code 1st and when timer expires after 5 seconds, timer will be printed. Time,Tide and JS wait for None.



This is the call stack



Then after 5 seconds, the callback function appears in call stack.

Everything is executed through call stack. If any operation blocks the call stack it is called blocking the main thread.

So if I have a function which performs a very heavy task, then the main thread will be blocked as the call stack will be executing that function while other functions may not be getting a chance.

So due to

- **functions being first class citizens,**
- **Callback functions**

We can do asynchronous operations in JS.

Now lets study about event listeners

```
index.html > html > head > title
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>Document</title>
7   </head>
8   <body>
9       <button id="clickMe">Click Me</button>
10      <script src="hello.js">
11      </script>
12  </body>
13  </html>
```

We create a button in index.html file.

In js file,

```
document.getElementById('clickMe')
.addEventListener("click",function(){

})
```

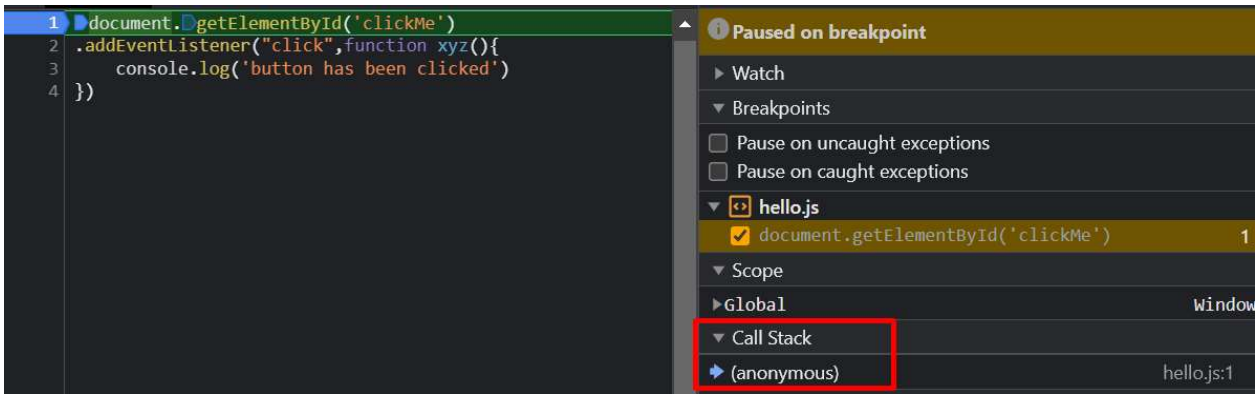
We attach event listener to the button with id clickMe. That event is click. What will happen if the the event occurs? It will call the callback function. So the callback function will be stored somewhere and when the click happens it will come into our call stack automatically

```
document.getElementById('clickMe')
.addEventListener("click",function xyz(){
    console.log('button has been clicked')
})
```

Lets see our console .

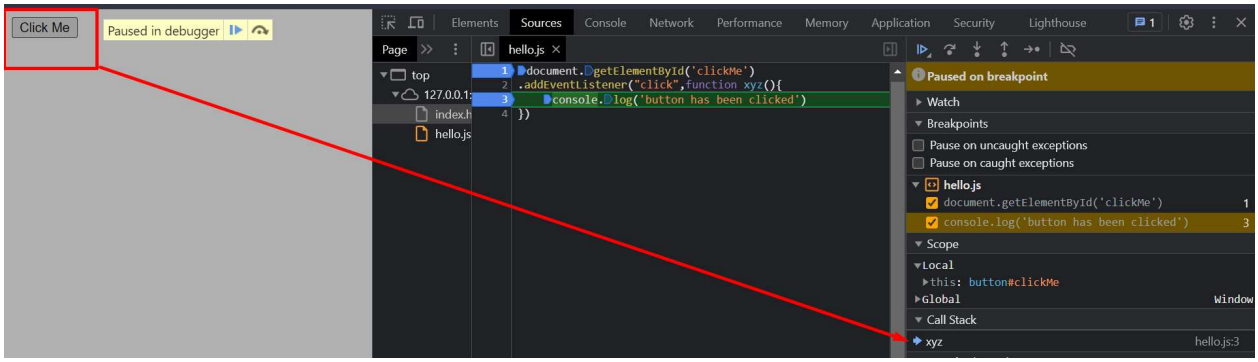
On clicking Click Me , we see output in the console.

So

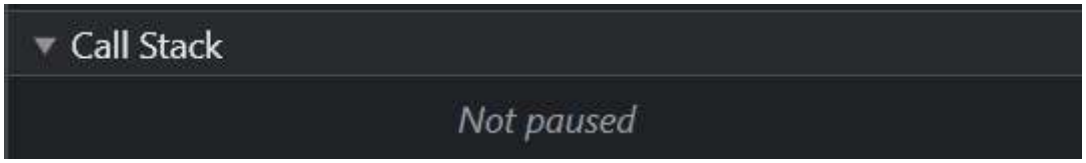


Call stack just has global execution context .

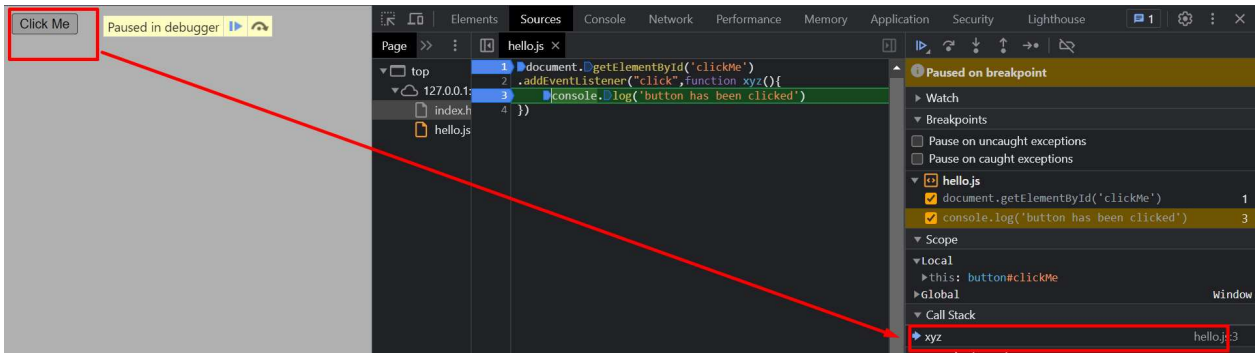
And when i click..



Execution context of xyz appears in the call stack but global execution context cannot be seen as it was popped off.Output was printed in console and then call stack become empty.



And when i click again



The call stack has got xyz execution context again.

Now we will see closures demo with event listeners. Lets say we need to count how many times a button has been clicked.

```
var count=0
document.getElementById('clickMe')
.addEventListener("click",function xyz(){
    console.log('button has been clicked',++count)
})
```

But using global variable is not good solution.

Now if we remember, closures was used for data hiding.

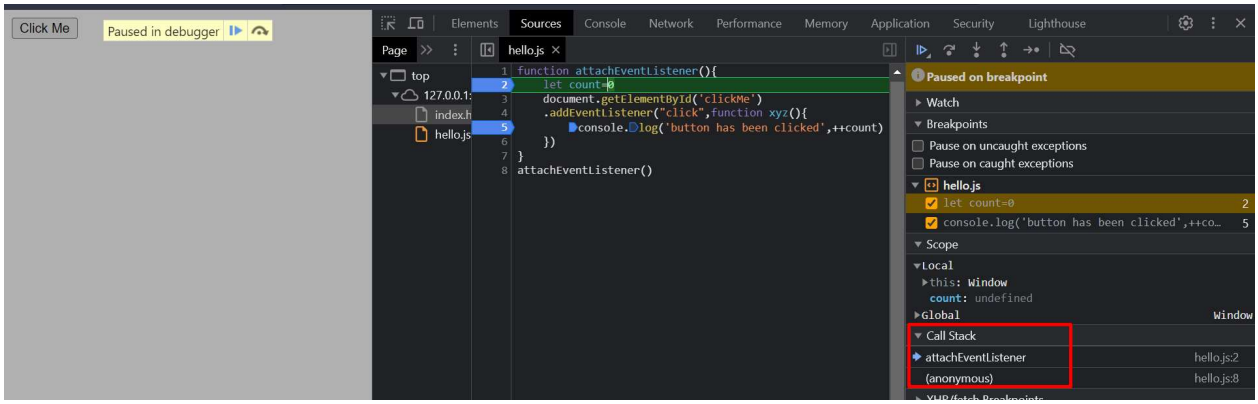
```
function attachEventListener(){
    let count=0
    document.getElementById('clickMe')
    .addEventListener("click",function xyz(){
        console.log('button has been clicked',++count)
    })
}
attachEventListener()
```

Now the callback function forms a closure with count,that is,the callback function remembers the value of count.

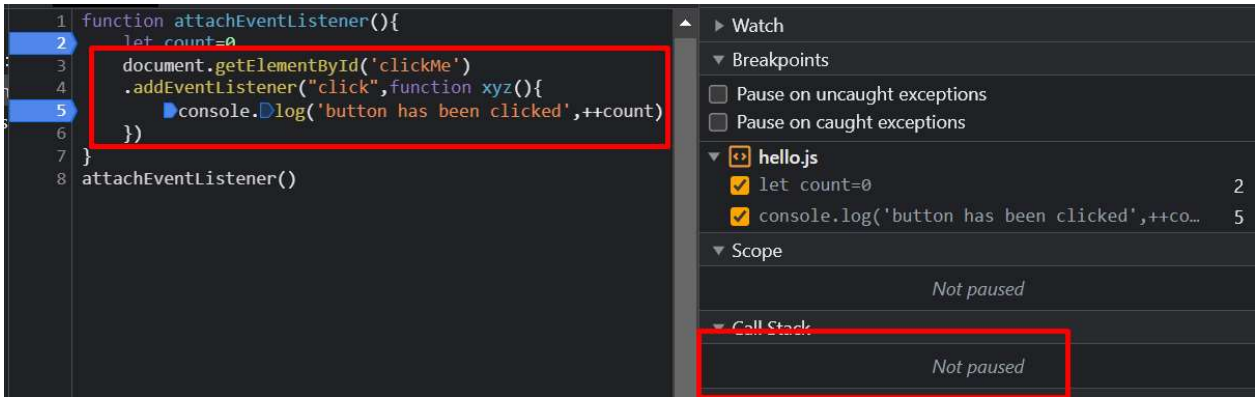
Output

```
button has been clicked 1                    hello.js:5
button has been clicked 2                    hello.js:5
button has been clicked 3                    hello.js:5
button has been clicked 4                    hello.js:5
```

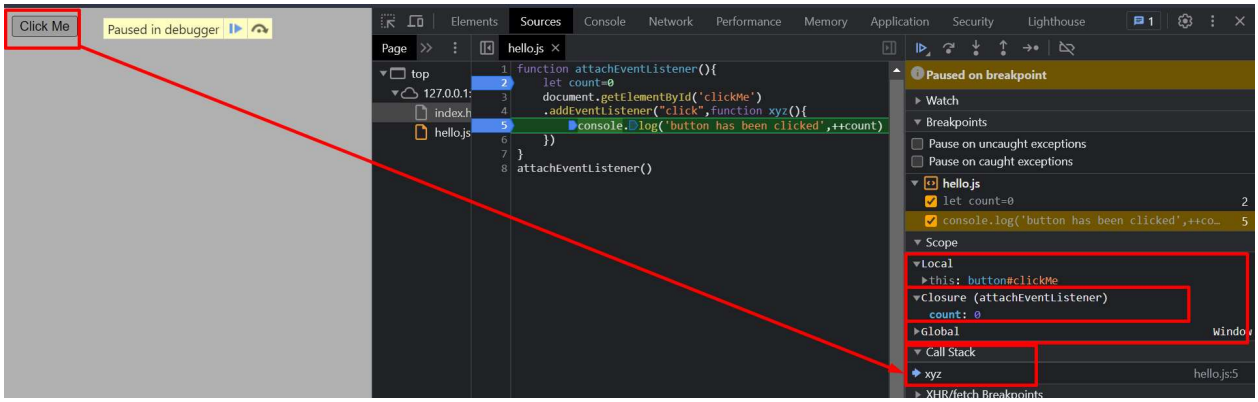
So seeing the debugger



The call stack has global execution context and attachEventListener() initially.Now the callback function will be registered and call stack will be empty.



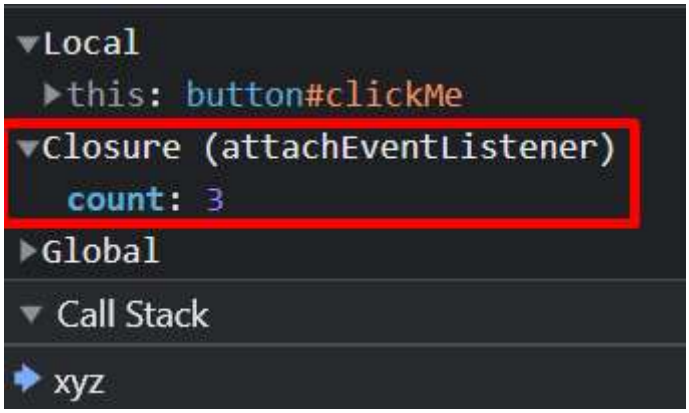
Now when we click on button,



We see that the execution context of xyz has been pushed onto the stack. And also that xyz forms a *closure* with count variable.

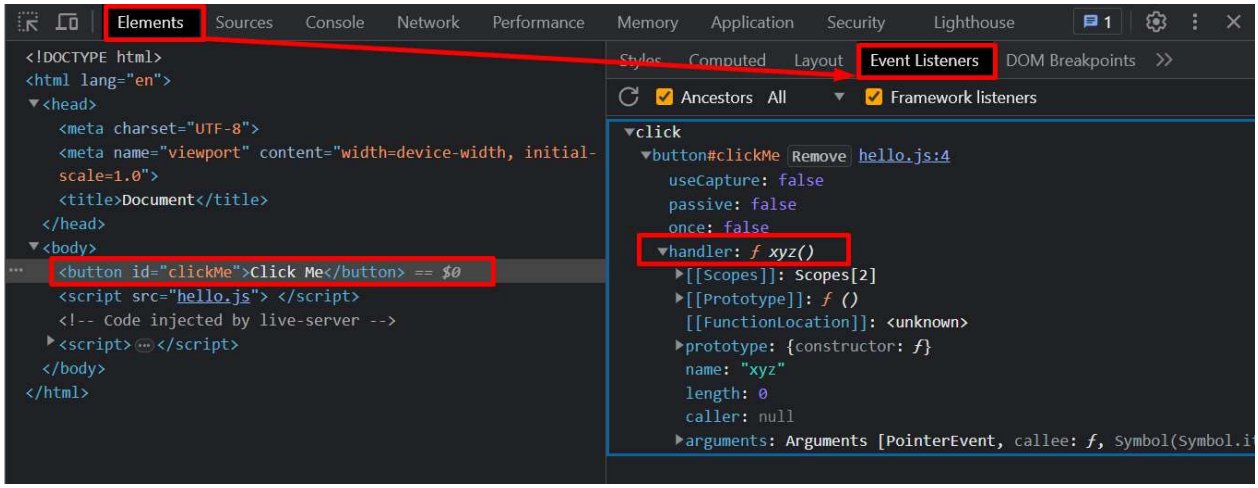
After logging the output,call stack becomes empty.

And if we keep on clicking,

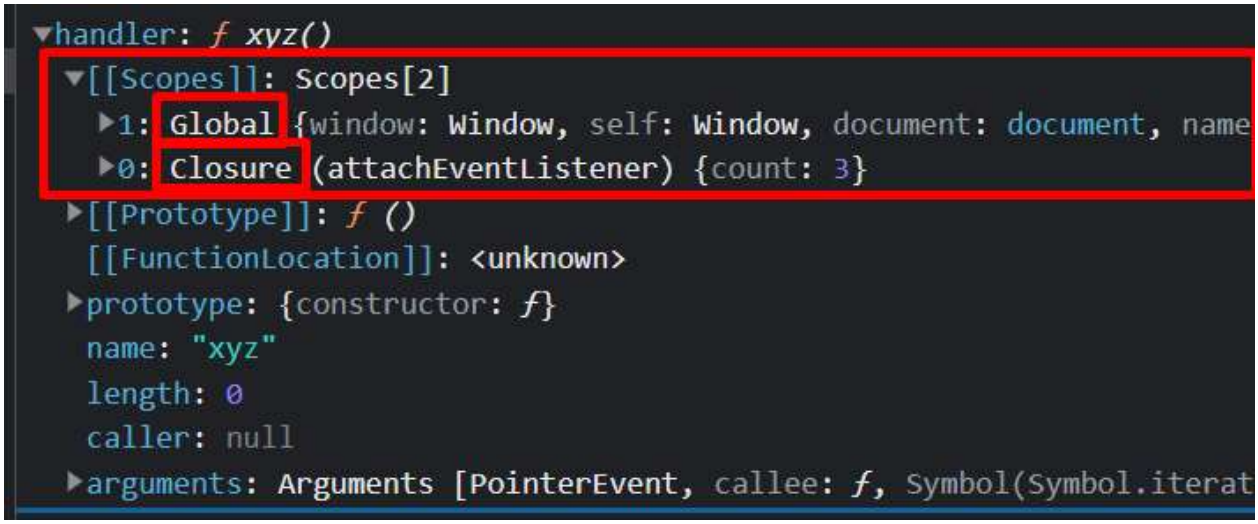


The value of count increases.

Now, Go to *elements*. And click on the button code in HTML, we will see *Event Listeners*. Click on it to see event Listeners.



Now what is the scope in the handler? Its the same lexical we had studied in earlier videos.And guess what will be inside the scope? Global and Closure

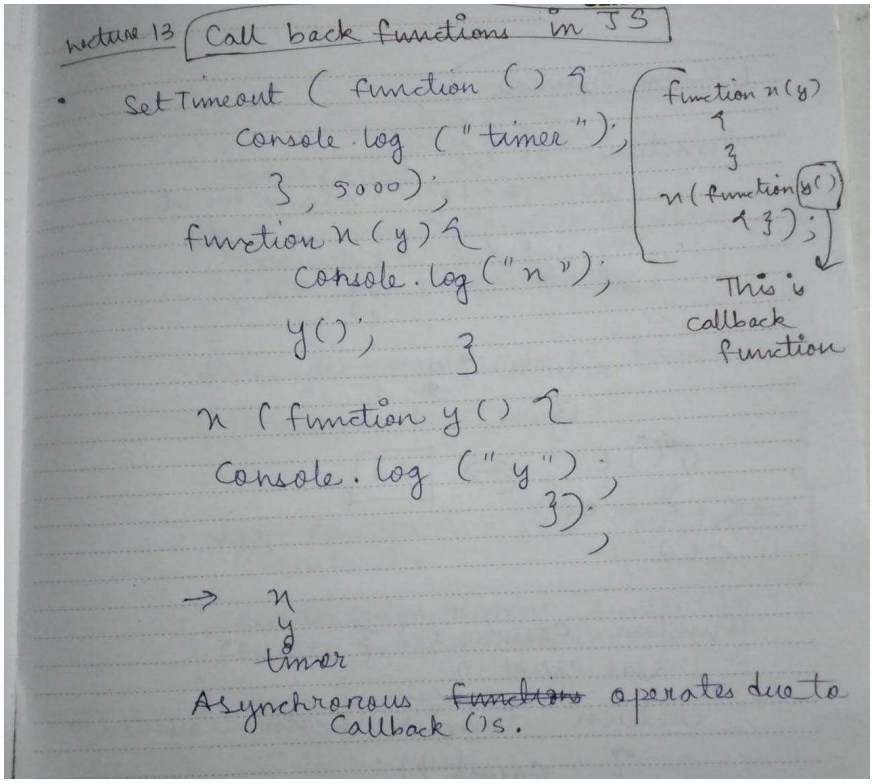


So whenever the callback function is executed , it has the scope attached to it.

Qs-> Why do we need to remove event Listeners?

Event listeners are heavy. They form a closure. Even when call stack is empty and we are not executing any code but still the program cannot free up the count because we never know when someone will click on the button. If many event listeners are attached then our page will become slow because of the memory of so many closures holding so many scopes. Therefore,we need to remove event listeners.

HANDWRITTEN NOTES



SUMMARY

1. Function that is passed on as argument to another function, is called callback function.

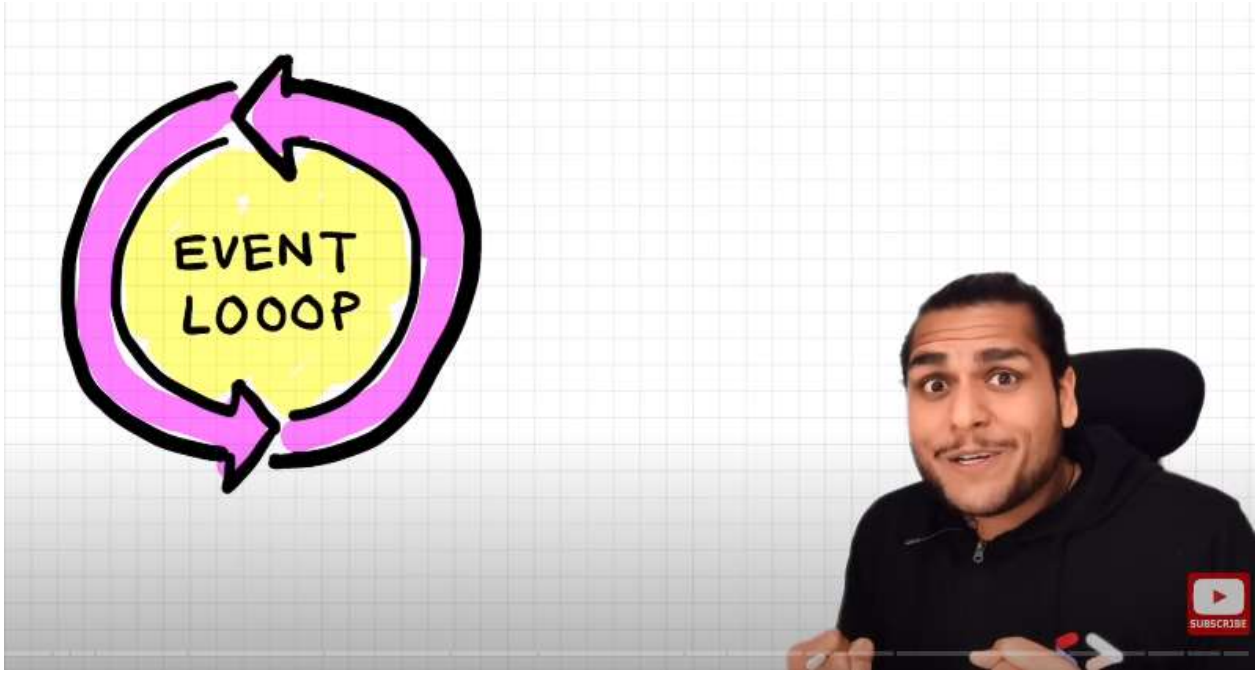
2. setTimeout helps turn JS which is single threaded and synchronous into asynchronous.

3. Event listeners can also invoke closures with scope.

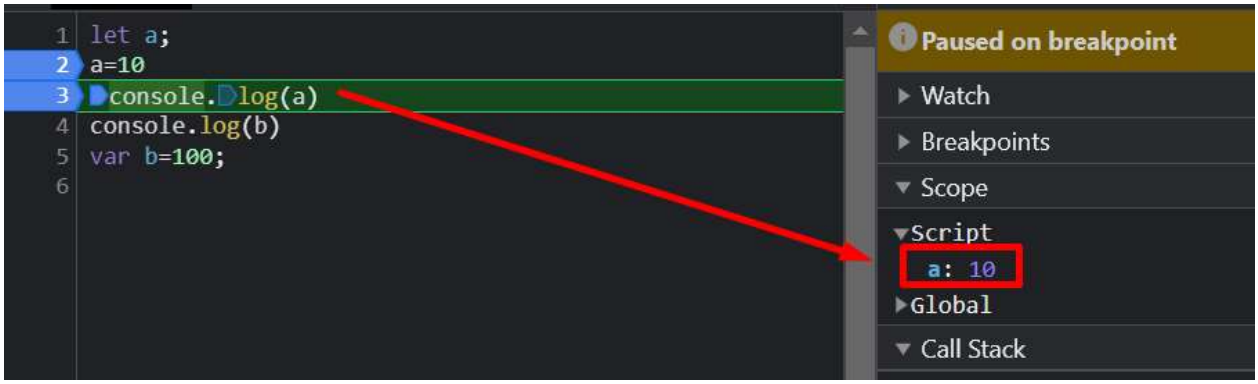
4. Event listeners consume a lot of memory which can potentially slow down the website therefore it is good practice to remove if it is not used.

Asynchronous JavaScript & EVENT LOOP from scratch 🔥 | Namaste JavaScript Ep.15

Youtube Link-https://youtu.be/8zKuNo4ay8E



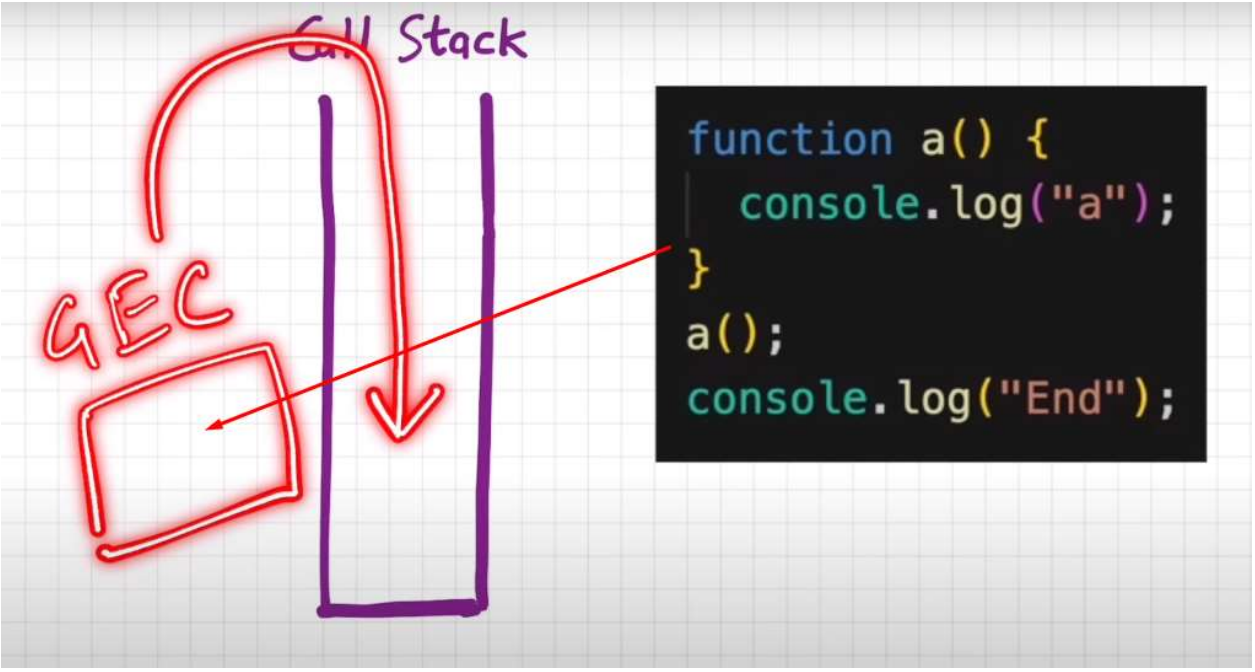
What do we learn?



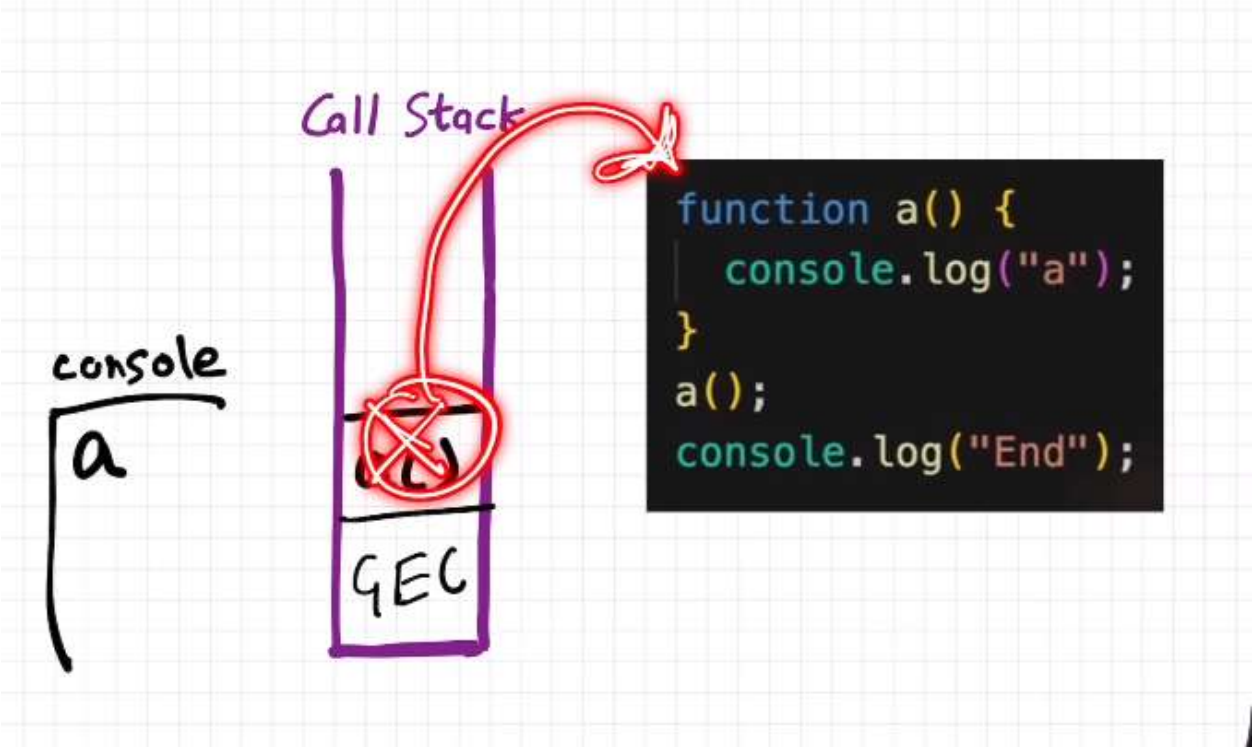
Event Loop

- **CallbackQueue**
- **Microtask Queue**
- **And how everything works inside browser.**

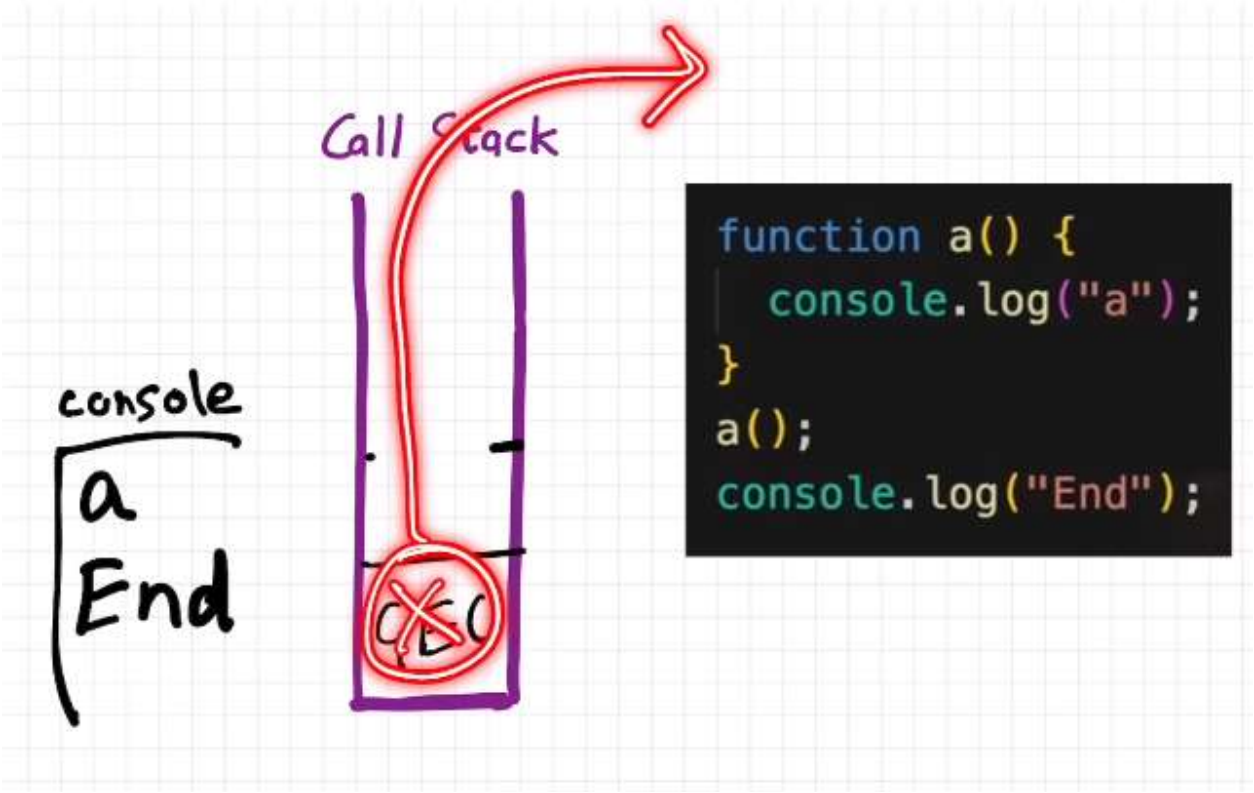
JS is synchronous single threaded language and it can do 1 thing at a time.
It has 1 call stack present inside the Java script engine.
Whenever code is run Global execution context is created and put inside call stack.



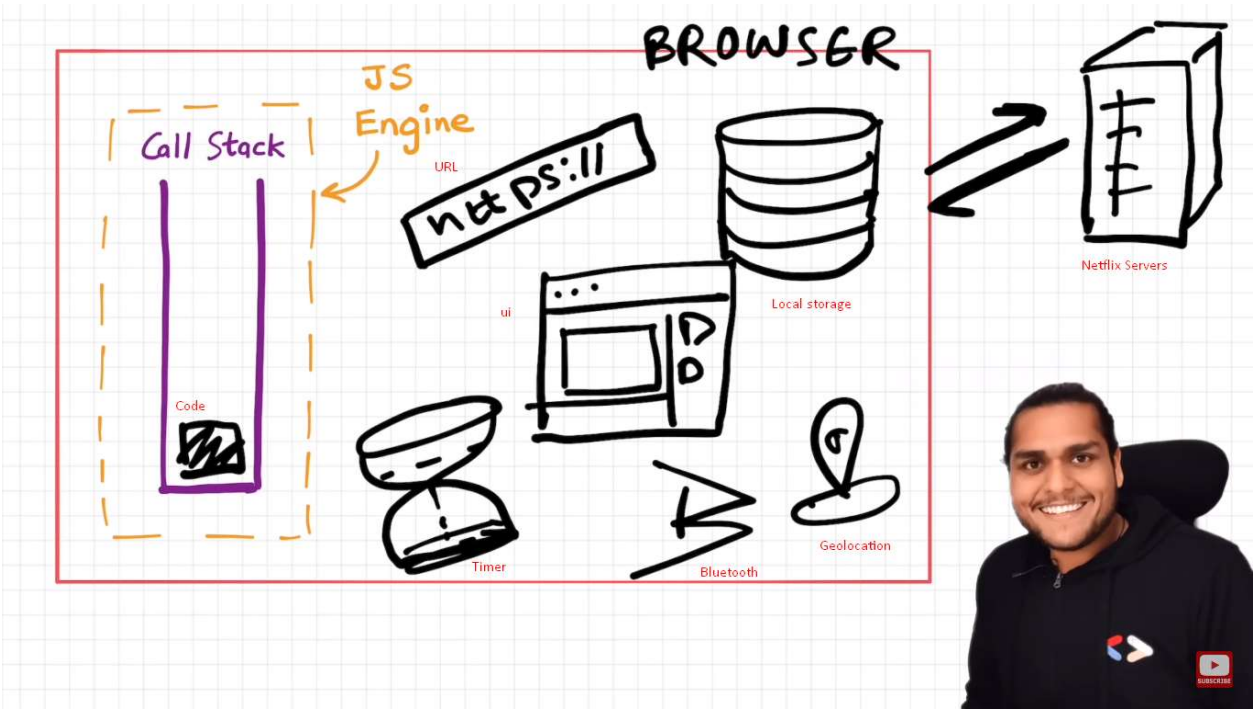
Execution context for a() created during function invocation of code execution phase.
a is logged onto console.



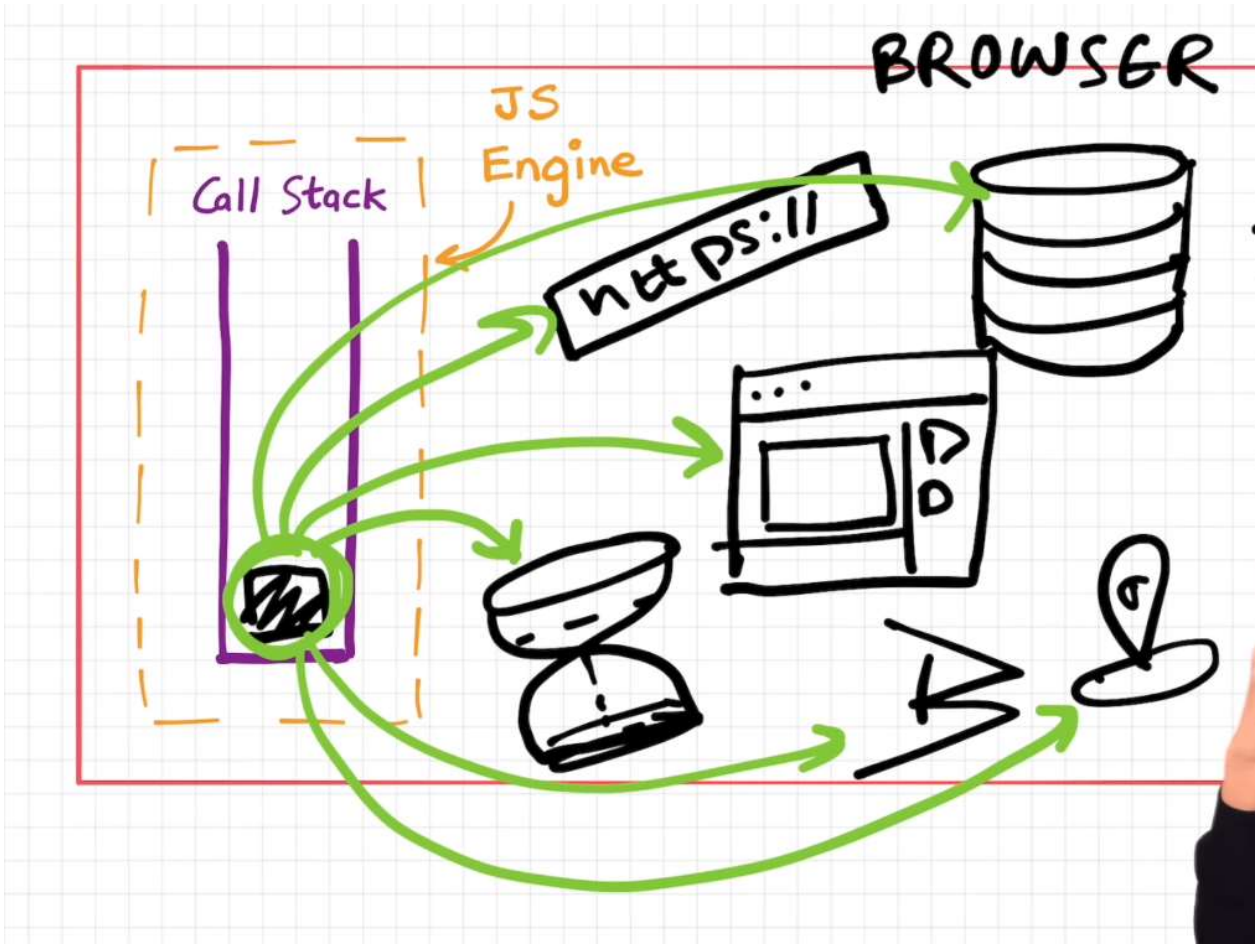
Then a context is popped off.



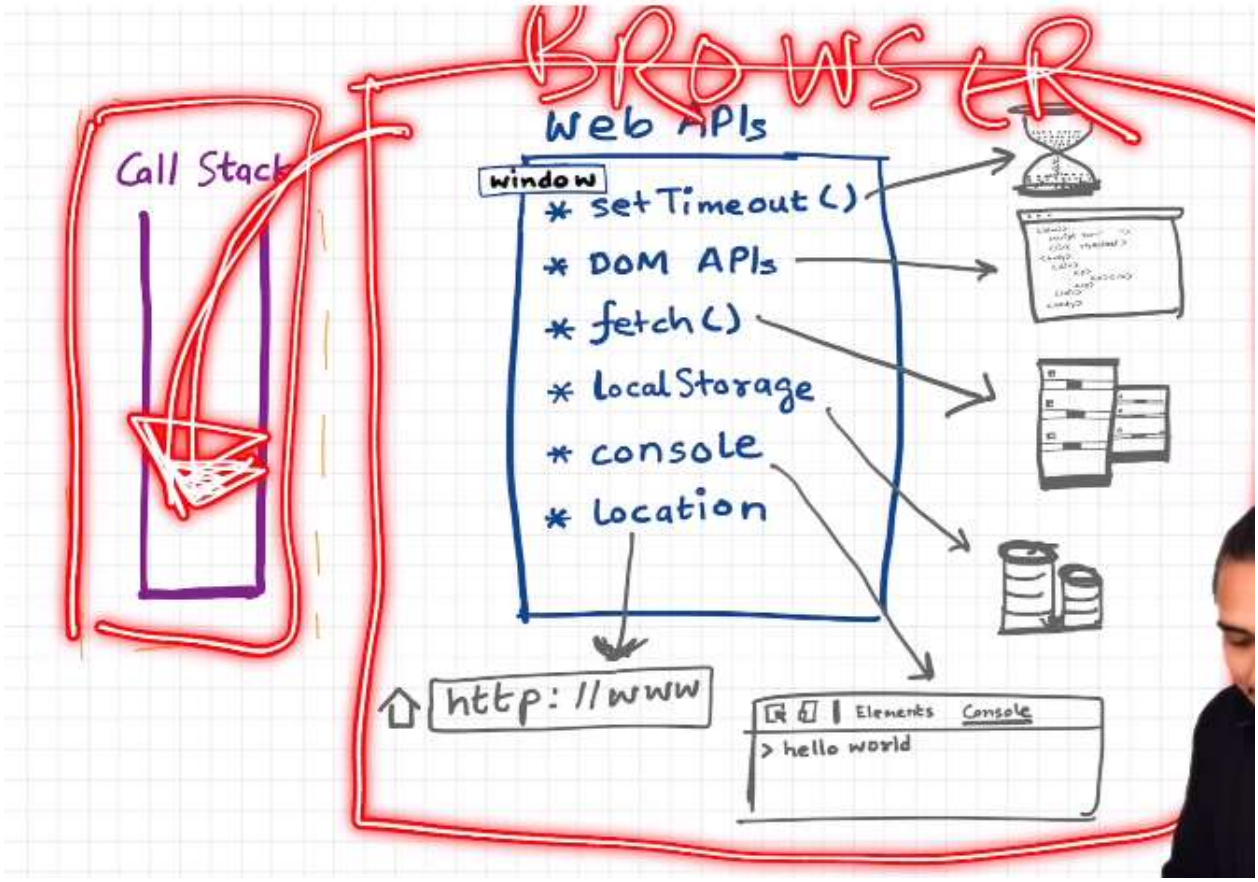
Finally End is printed and GEC is popped off too.
Call stack does not have timers. So we need something else.



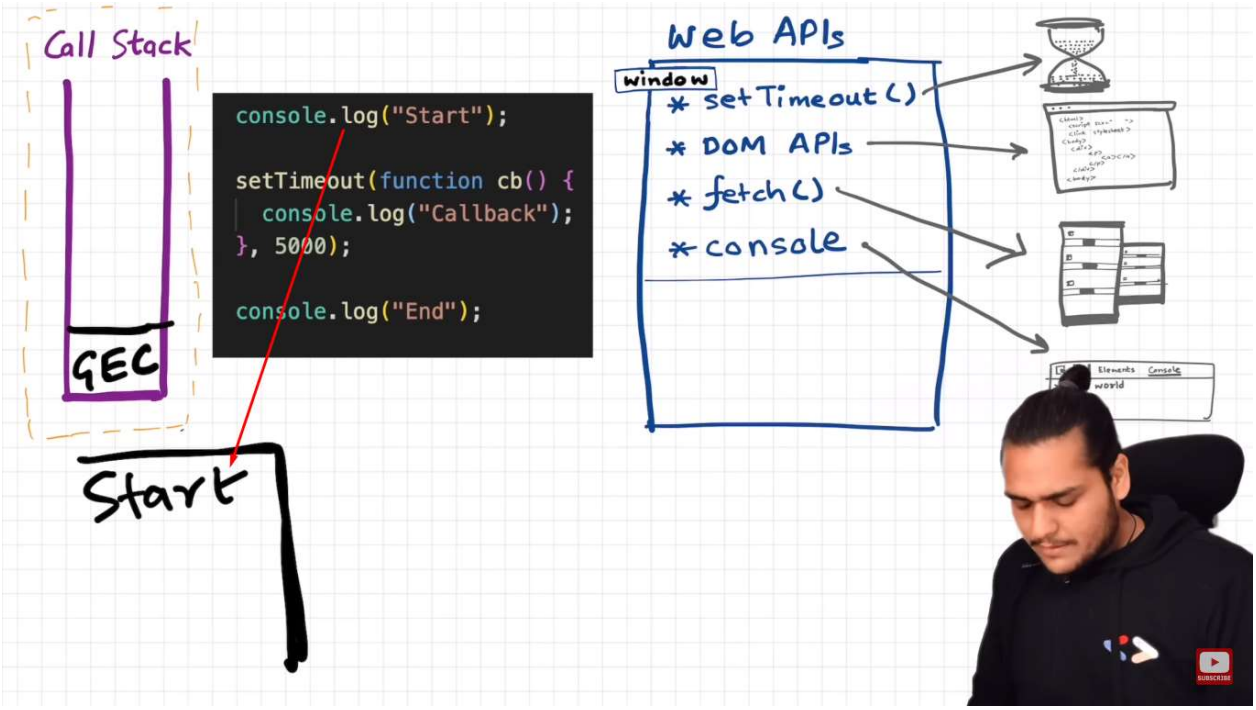
Superpowers of Browser



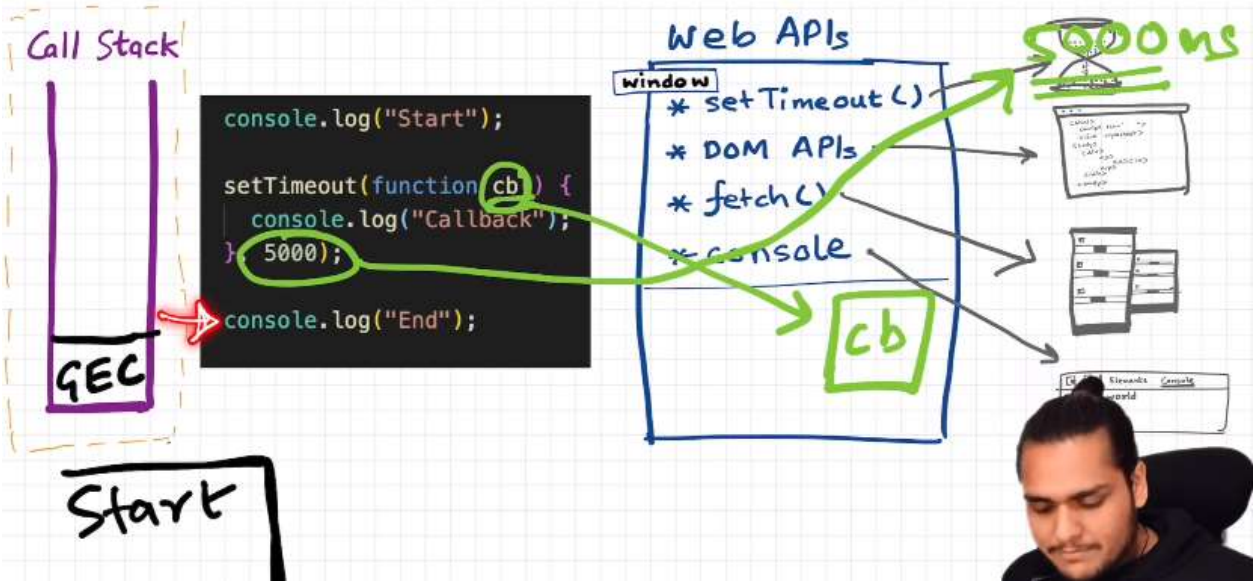
JS needs a way to access the superpowers.
These are the web apis.
setTimeout, document,console are not JS, they are part of browsers.



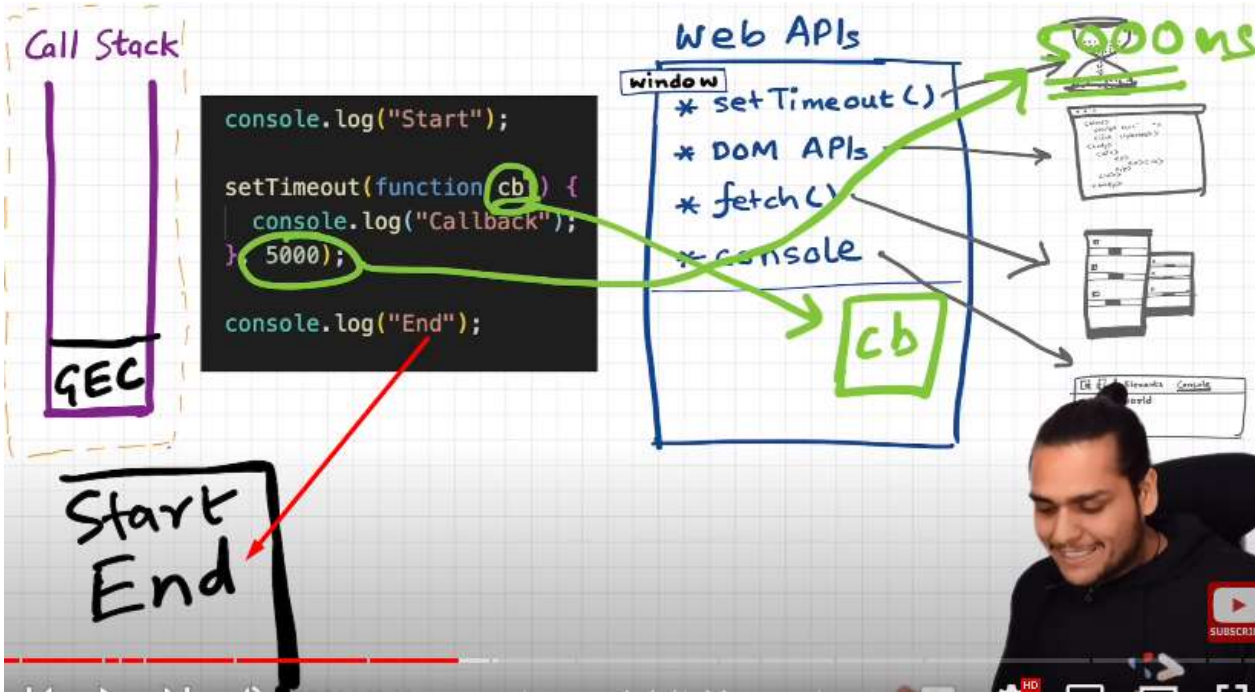
JS gets access to these powers via the browser with help of global object.
That global object is window.
example-> window.console.log()
But since window is present in global scope we can access it without writing it.



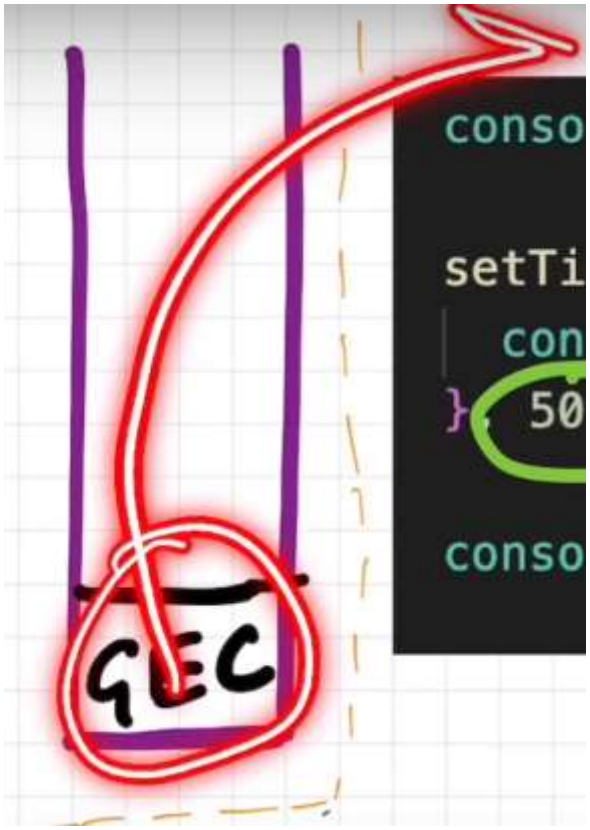
Next is setTimeout. It will be registered as shown below in web APIs environment by setTimeout() and timer starts as shown.



Since JS waits for nobody, End is printed on the console.

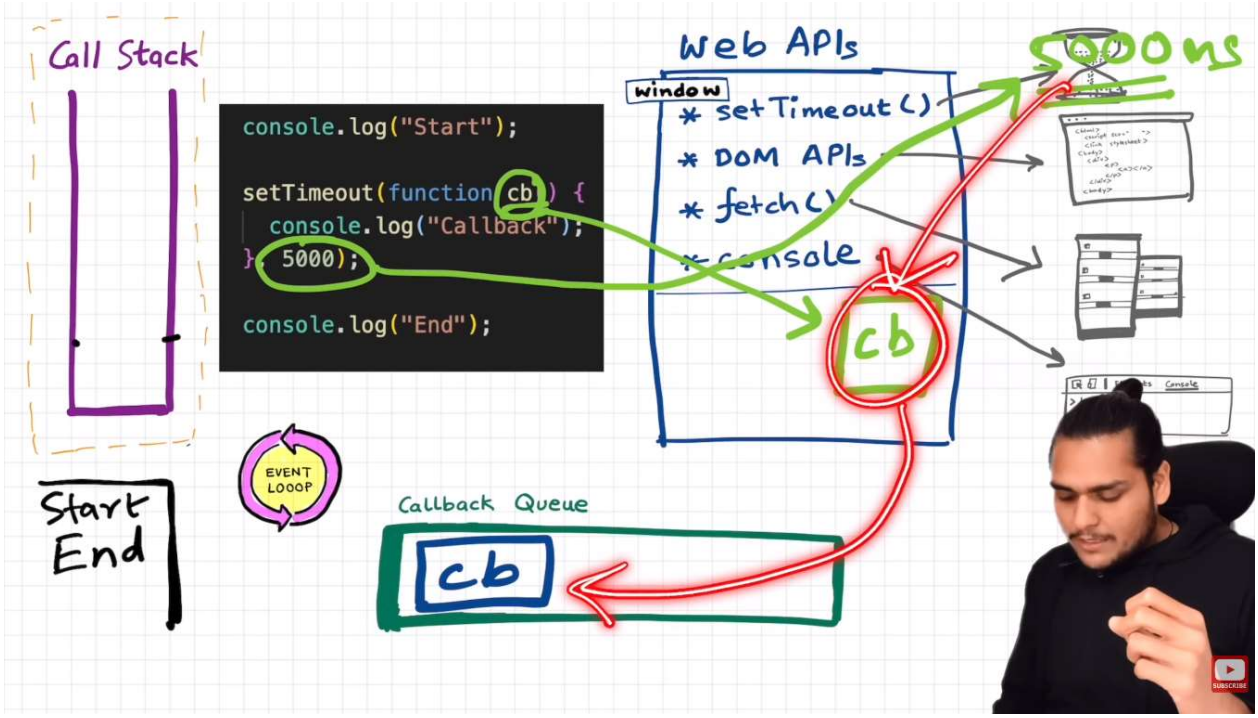


GEC popped off.

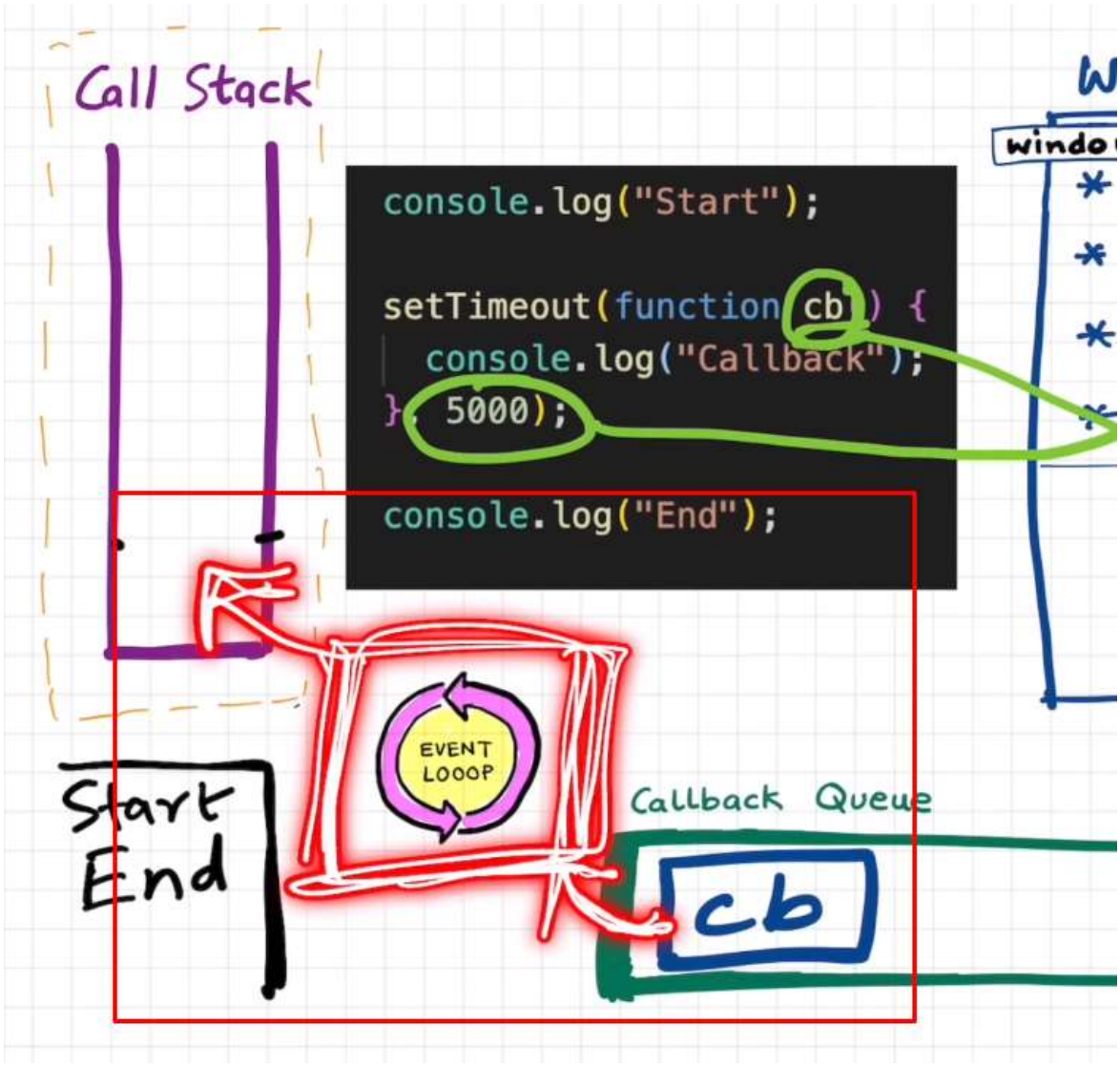


Meanwhile the timer is till running.

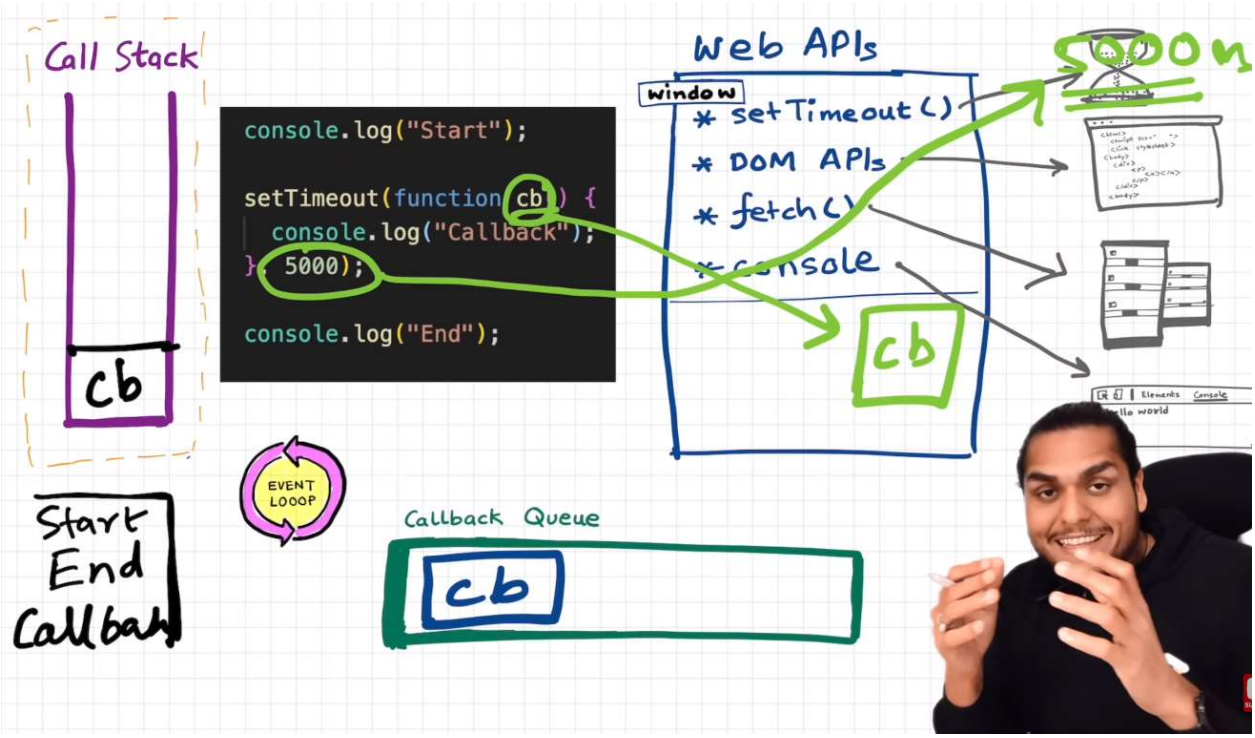
So we need the callback function inside the call stack then only the code inside callback can be executed, since we can execute JS code only if it is inside the call stack.



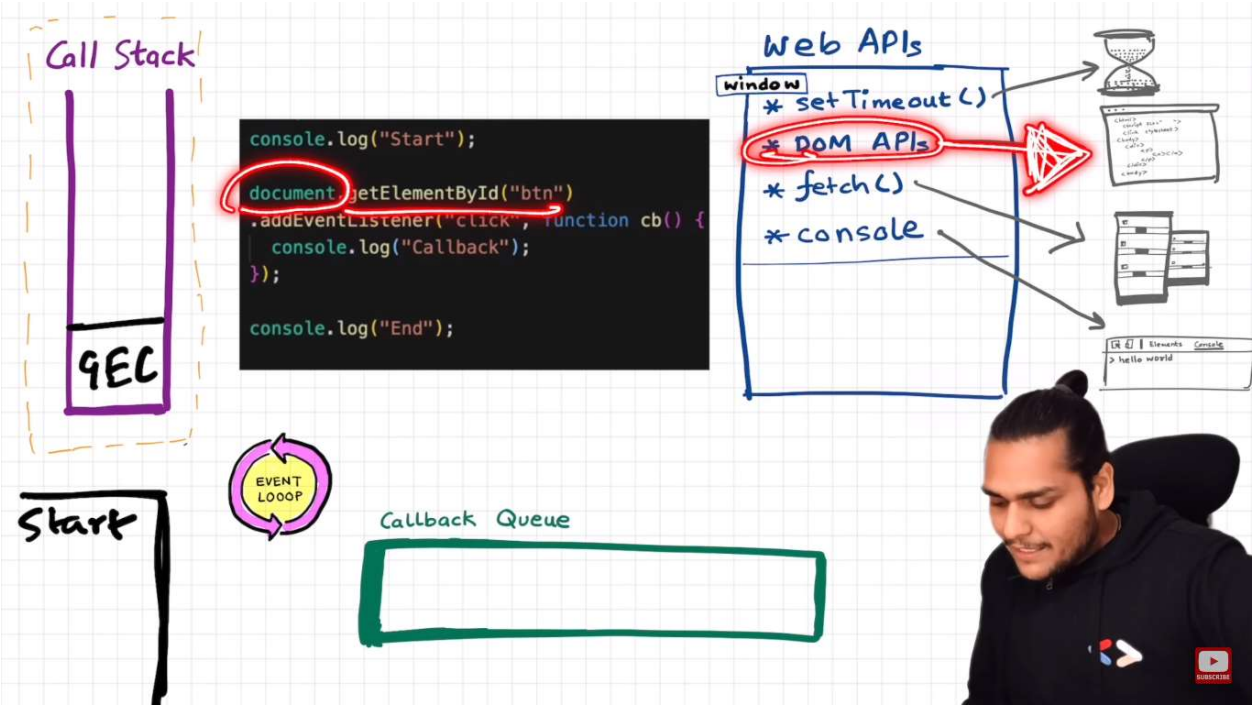
When timer expires callback function is placed in callback queue.
Job of event loop is to check callback queue and push the function from callback queue to call stack.



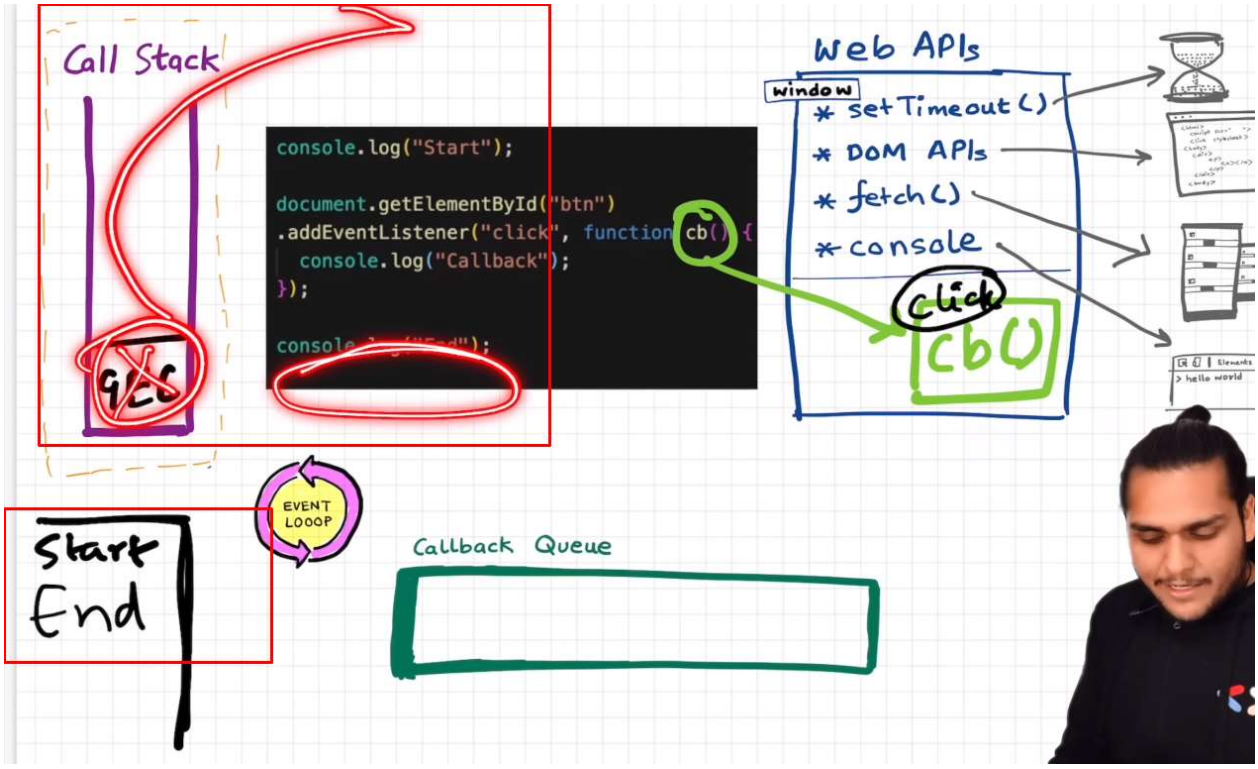
Now “callback” will be printed on the console



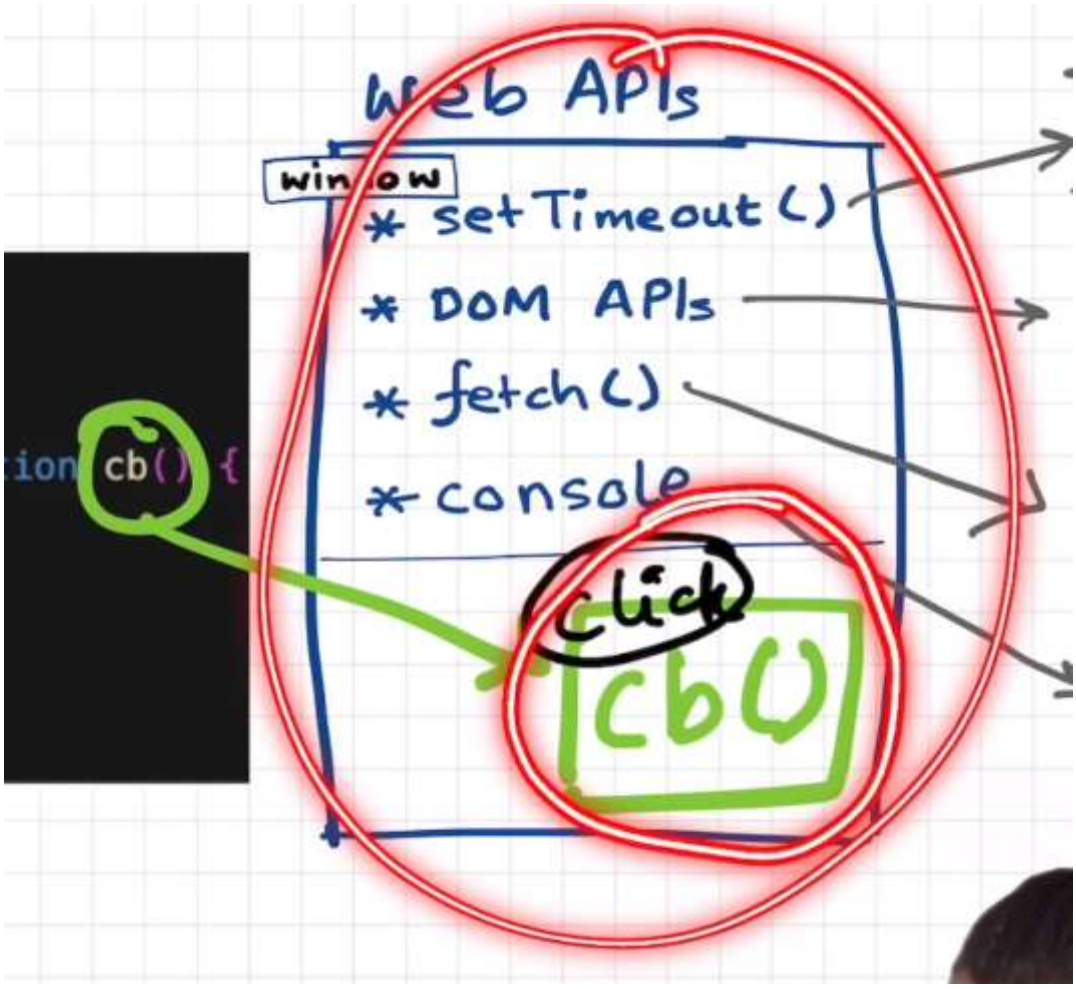
Now we will take another example.



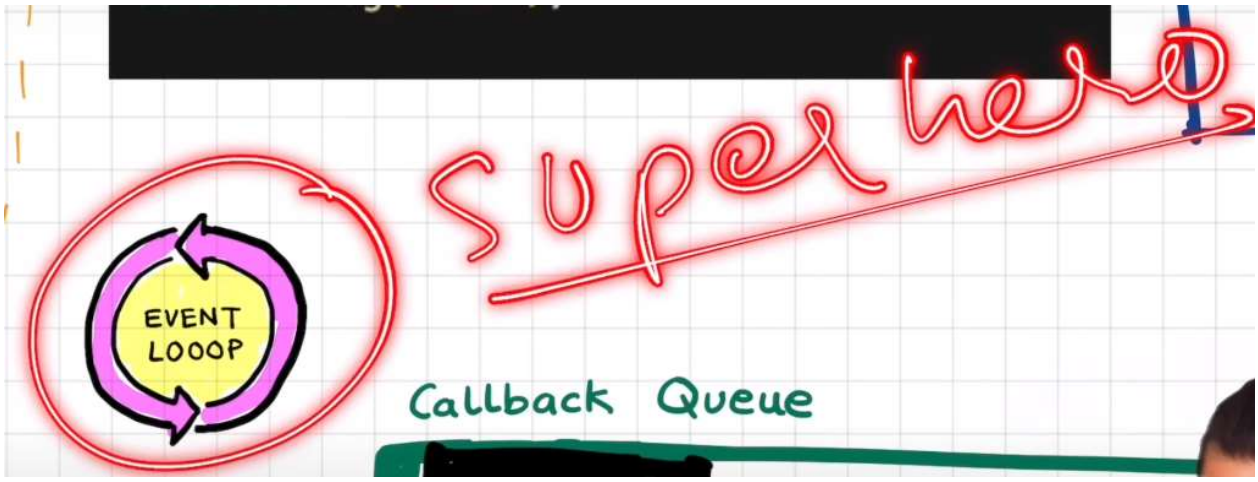
Start and end are printed on console and GEC is popped off.



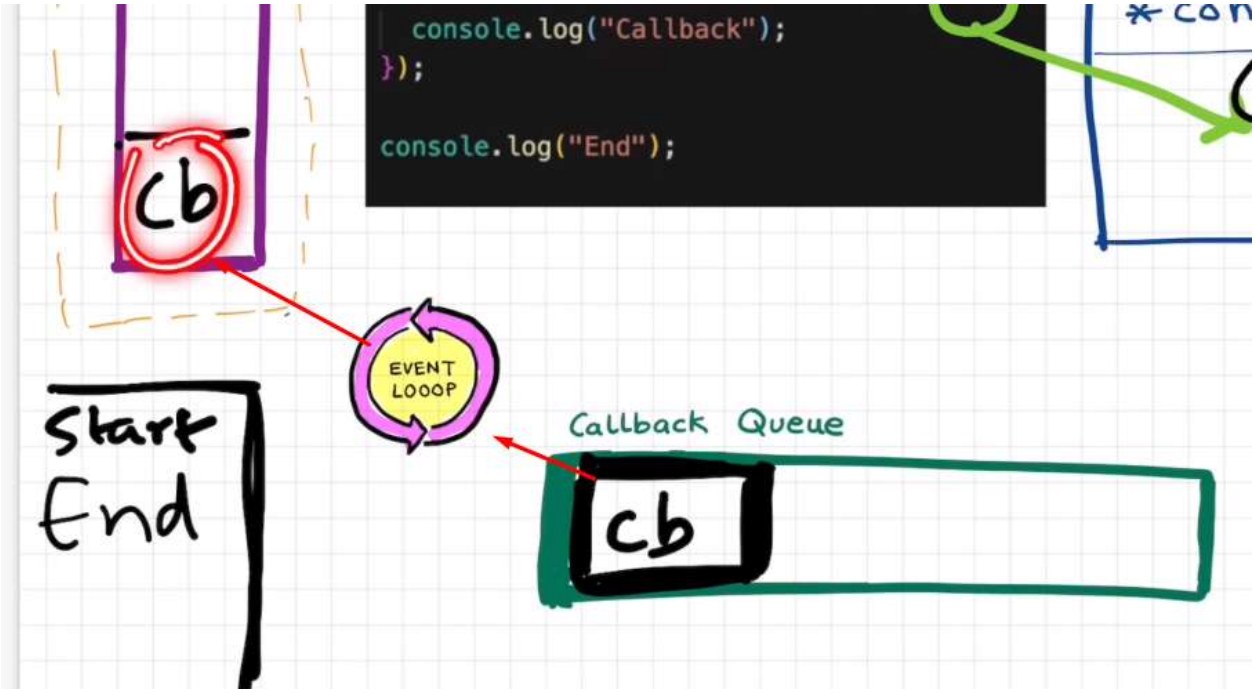
The callback () that was registered will remain in environment until we explicitly remove it.



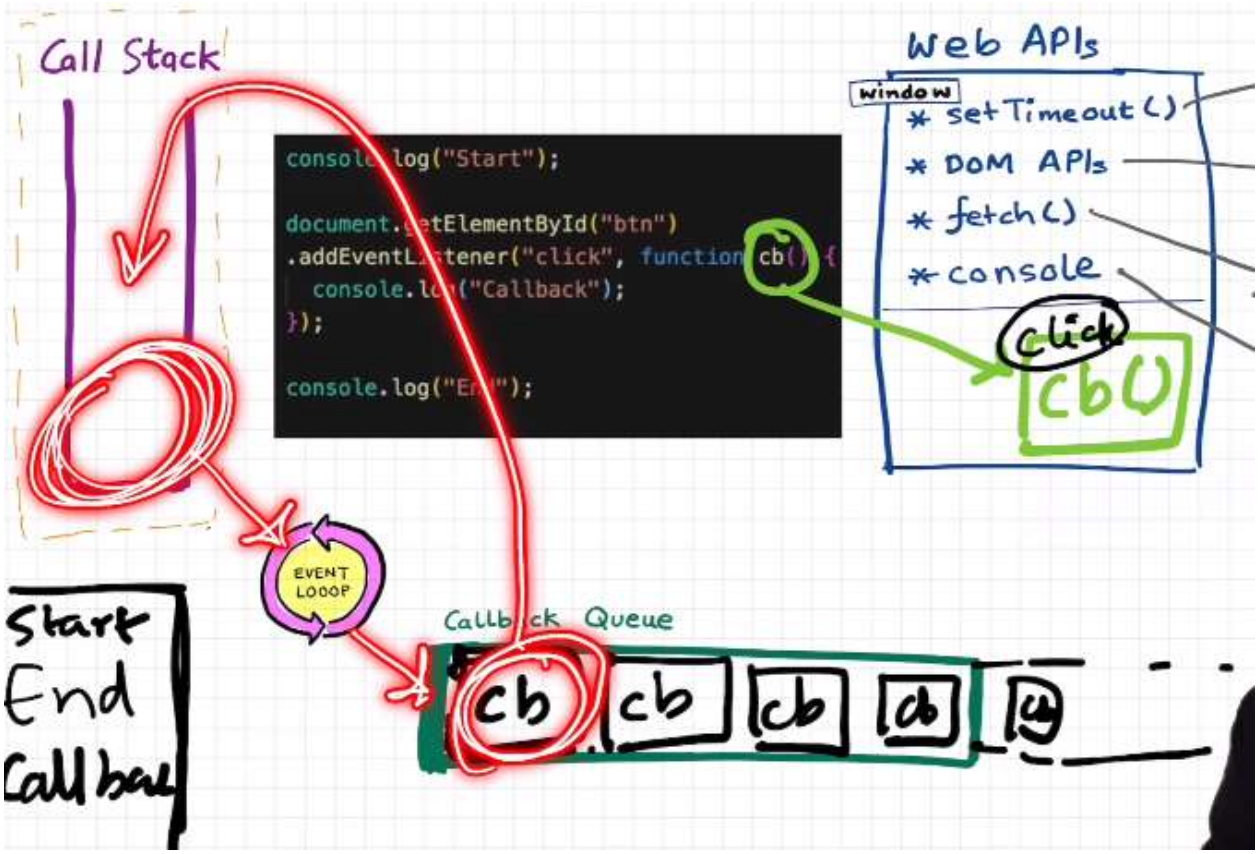
When user clicks on button, cb() is move to callback queue and waits for its turn to get executed.



Event loop constantly monitors the callback queue and pushes callback() to call stack.

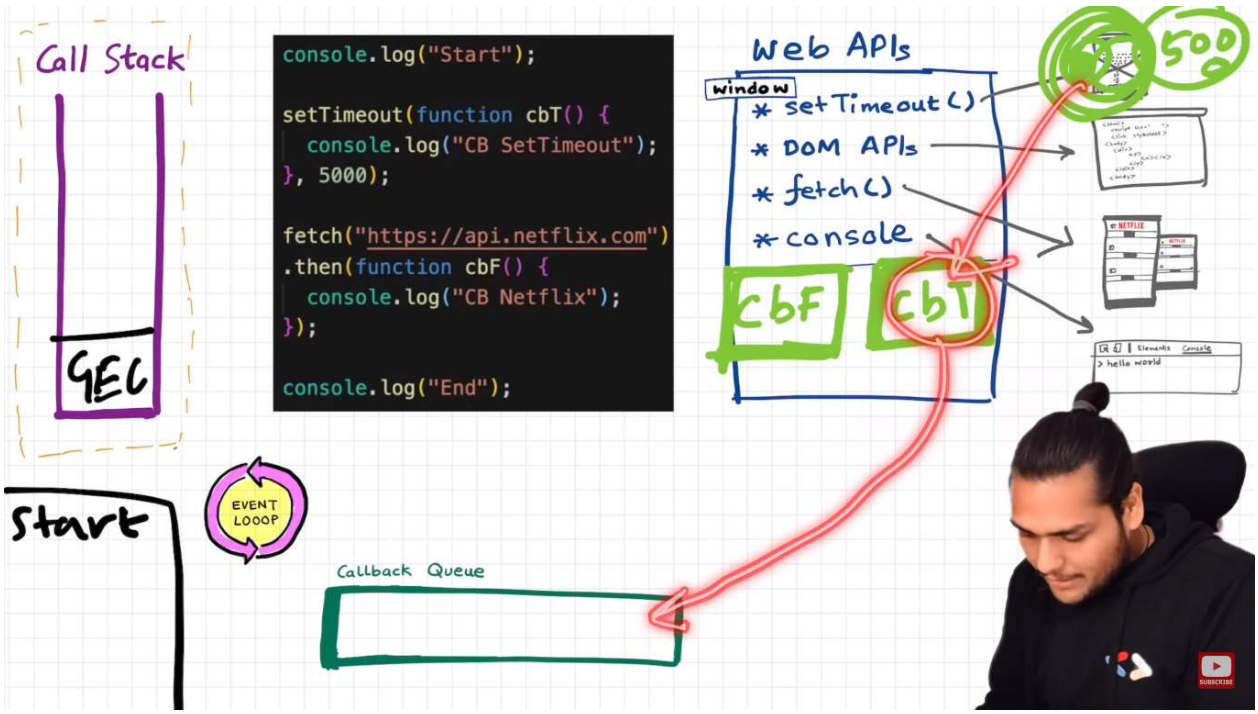


Then in console, “callback is printed” and callback queue is empty,call stack is empty.
Now if user continuously clicks button 4 or 5 times then callback() enters queue many times.

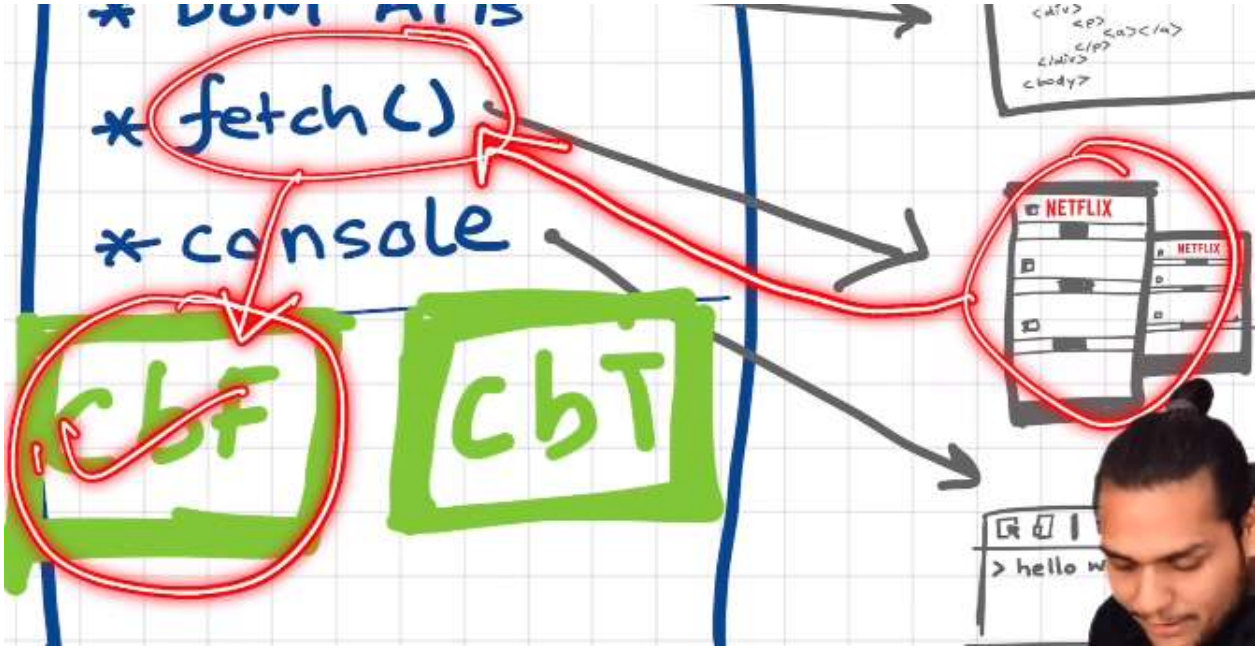


Then slowly every callback() is popped off from queue slowly one by one.

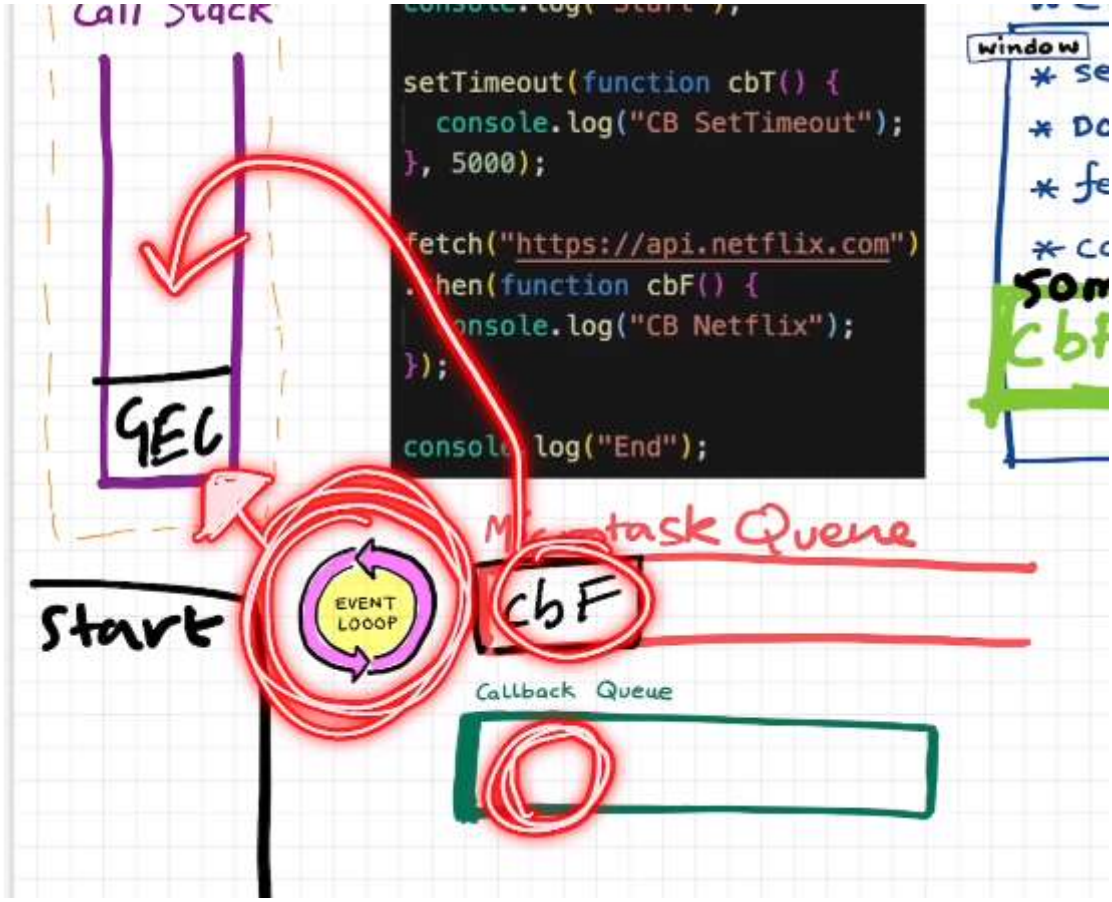
NOW WE WILL UNDERSTAND FETCH().



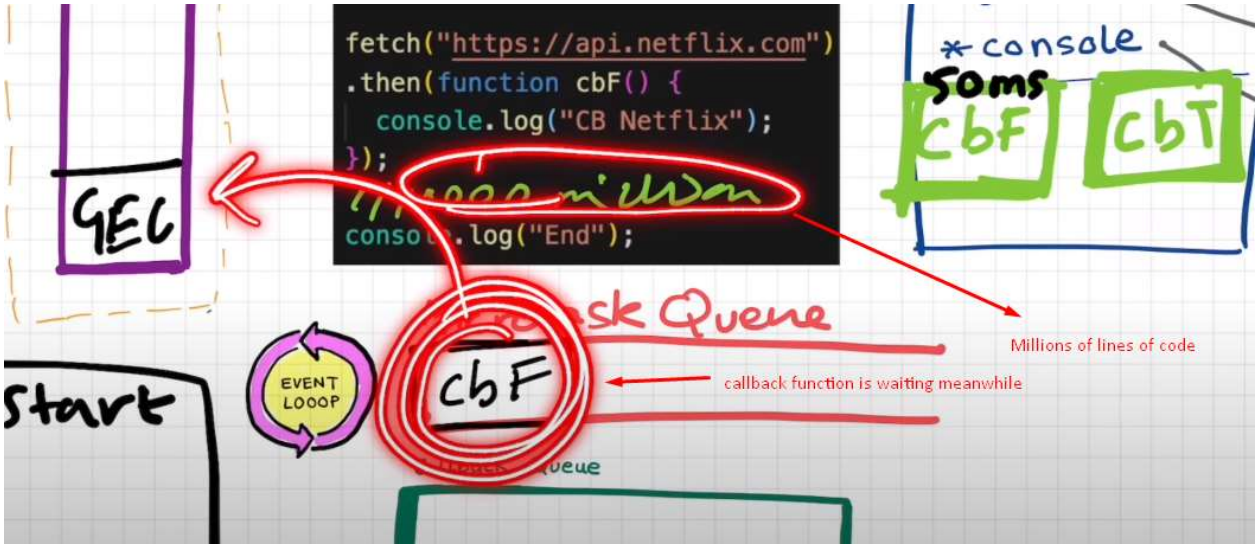
cbF and cbT are registered as callback functions in web api via fetch() and setTimeout() respectively.
cbT is waiting for its timer to expire before entering callback queue.
cbF is waiting for data to be returned from server.



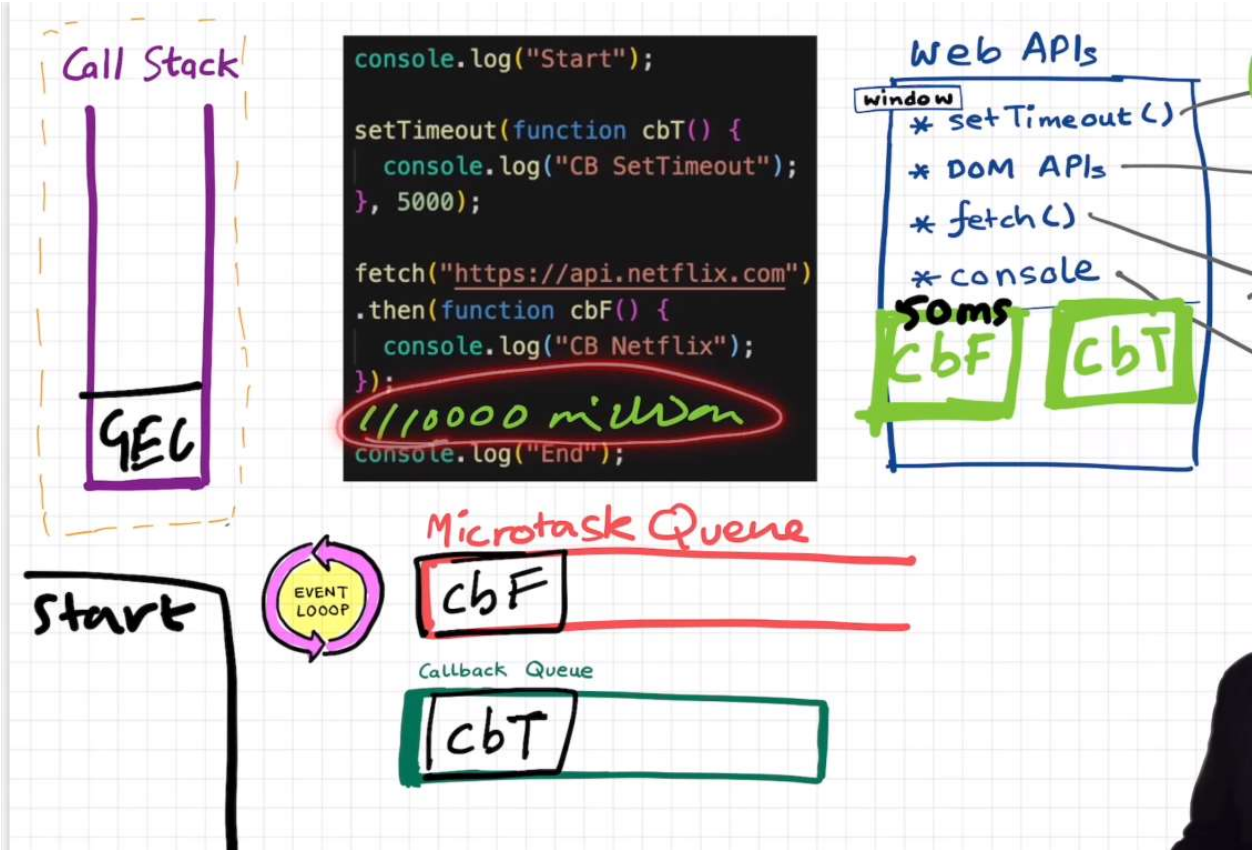
Will cbF from fetch enter callback queue?
NO
We have microtask queue which has higher priority,means callback functions in it executed first and functions in callback queue executed later.



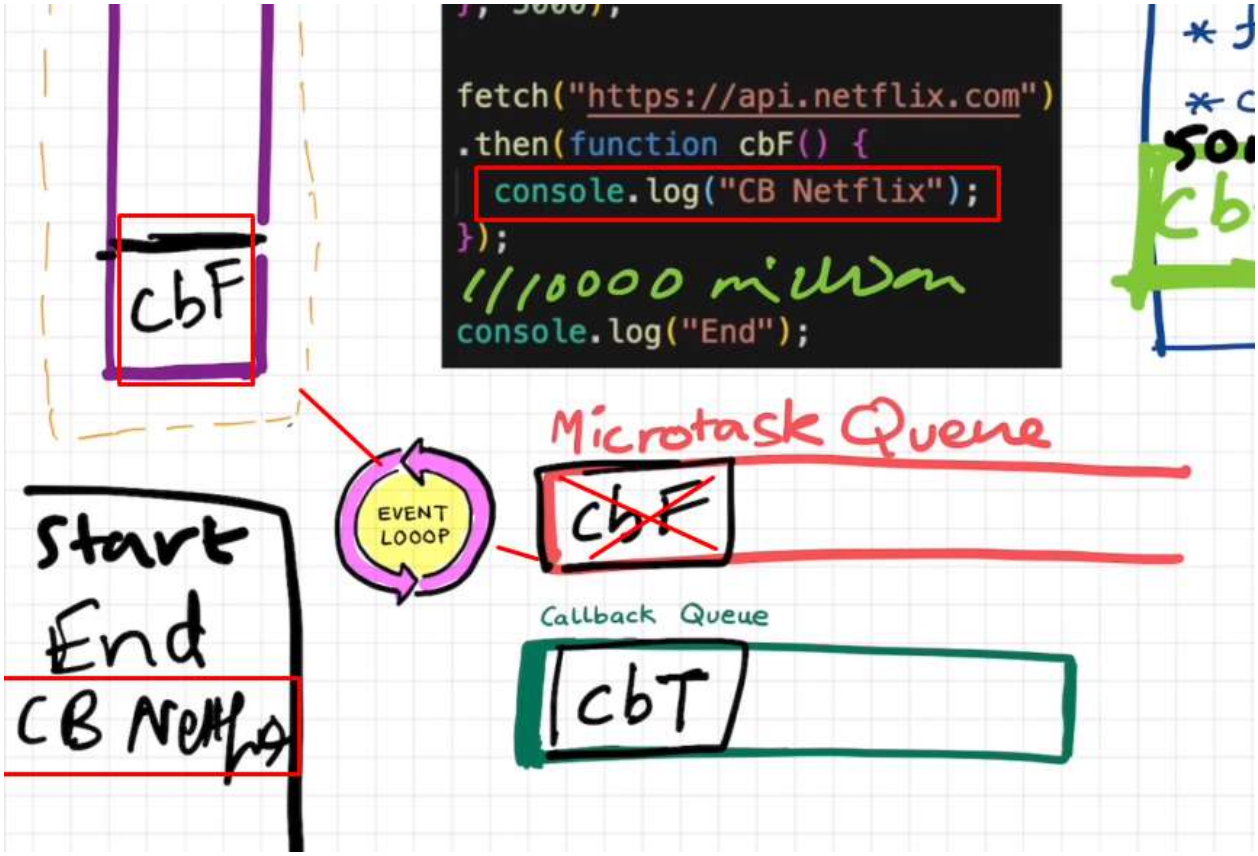
Lets say netflix servers were really fast and we got data so we push the callback function cbF into microtask queue quickly.



JS is doing one thing at a time but browser is doing lots of things.
cbF remains waiting in microtask queue as JS continues to execute 10000 lines of code in call stack.
Now lets say cbT also goes inside callback queue as its timer expires.

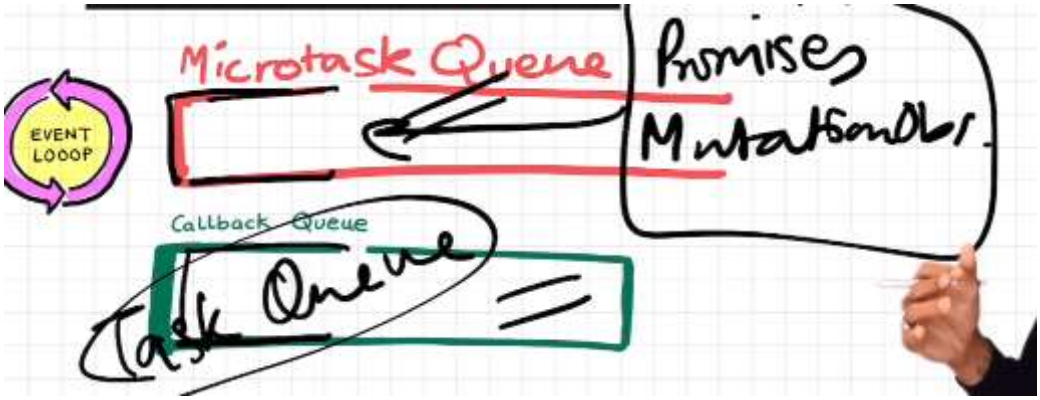


cbF waits in microtask queue,cbT waits in callback queue.
Event loop continues to check if call stack is empty or not.
When GEC is popped off, event loop pushes cbF first into call stack due to its higher priority of microtask queue.
Then "CB Netflix is printed on console."



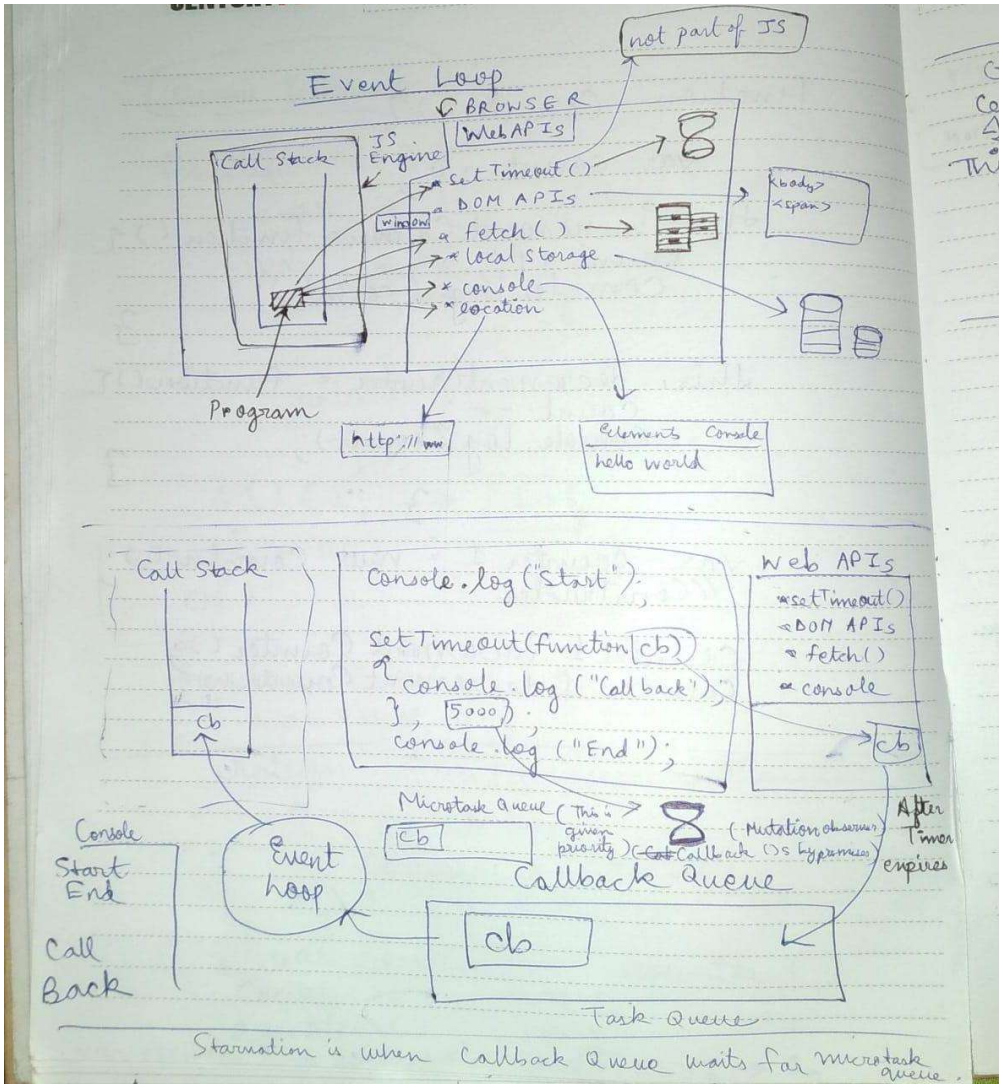
SIMILARLY cbT is pushed into call stack via event loop and then popped off.

All callback functions that come through promises goes into microtask queue.
There is something known as mutation observer which checks for mutation in DOM tree.
If there is mutation in DOM tree,it goes inside microtask queue.
Microtask queue gets promises and mutation observer callback functions.
DOM APIs go inside callback queue also known as task queue.



Once all tasks in microtask queues have executed i.e when microtask queue becomes empty then only callback queue gets opportunity.
So callback queue can go into starvation.

HANDWRITTEN NOTES



RECAP-

1. Browser has superpowers that are lent to JS engine to execute some tasks, these superpowers include web API's such as console, location, DOM API, setTimeout, fetch, local storage.
2. Asynchronous callback functions and event handers are first stored in Web API environment and then transferred to callback queue.
3. Promises and mutation observer are stored in API environment and then transferred to microtask queue.

4. Event loop continuously observes call stack and when it is empty it transfers task to call stack.
5. Micro task is given priority over callback tasks.
6. Too many micro tasks generated can cause Starvation (nit giving time to callback tasks to execute).
7.The Event Loop pushes the "queue" into the Call Stack only when the Call Stack is empty (i.e. the global execution context has been pushed off the call stack).
The order in which the Event Loop works is:
Call Stack→ Microtask Queue→Callback Queue

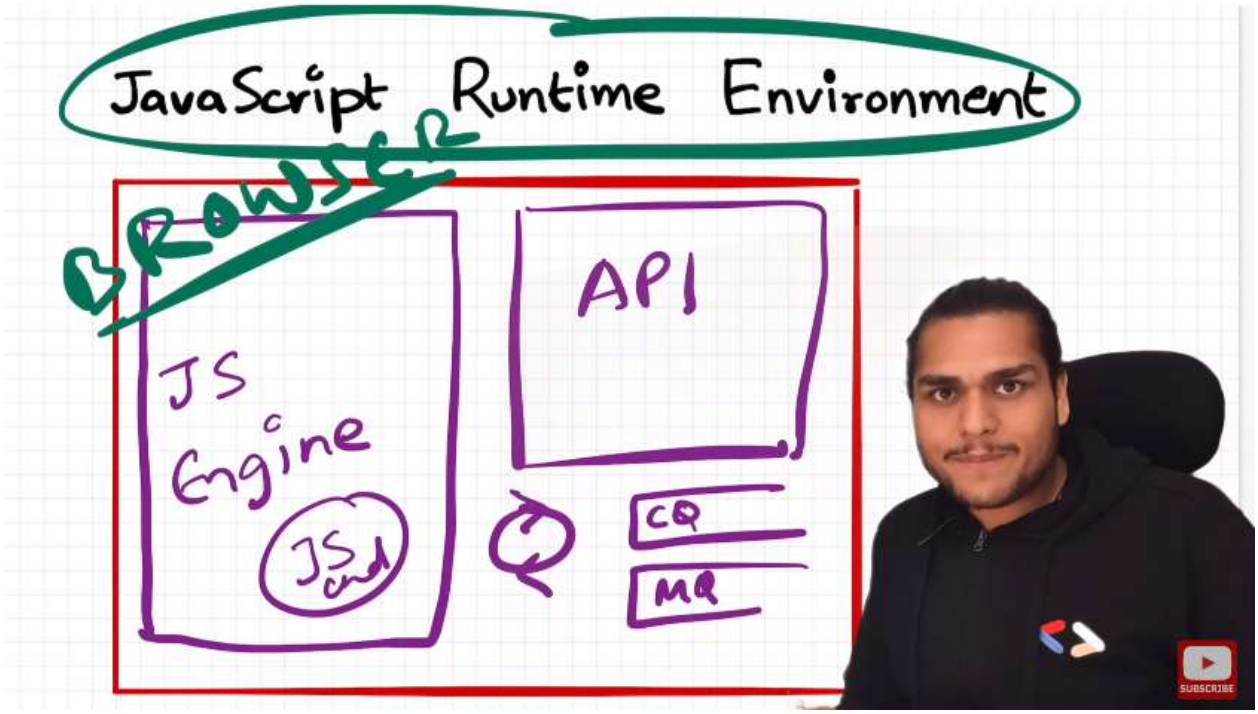
SOME DOUBTS CLARIFICATION-
1. When does the event loop actually start? - Event loop, as the name suggests, is a single-thread, loop that is `almost infinite`. It's always running and doing its job.
2. Are only asynchronous web API callbacks are registered in the web API environment? - YES, the synchronous callback functions like what we pass inside map, filter, and reduce aren't registered in the Web API environment. It's just those async callback functions that go through all this.
3. Does the web API environment stores only the callback function and pushes the same callback to queue/microtask queue? - Yes, the callback functions are stored, and a reference is scheduled in the queues. Moreover, in the case of event listeners(for example click handlers), the original callbacks stay in the web API environment forever, that's why it's advised to explicitly remove the listeners when not in use so that the garbage collector does its job.
4. How does it matter if we delay for setTimeout would be 0ms. Then callback will move to queue without any wait?
No, there are trust issues with setTimeout() . The callback function needs to wait until the Call Stack is empty. So the 0 ms callback might have to wait for 100ms also if the stack is busy.

JS Engine EXPOSED 🔥 Google's V8 Architecture 🚀 | Namaste JavaScript Ep. 16

Youtube Link- https://youtu.be/2WJL19wDH68

Javascript can run inside a browser, server,smartwatch, lightbulb, robots.
How is this possible?

- **Due to the javascript runtime environment.**
- **The JS runtime environment is like a big container that is required to run JS code.**
- **There is an event loop,callback queue,microtask queue inside the environment.**
- **It can have a set of APIs to connect to the outer environment.**
- **A JS environment is not possible without a JS engine. The JS engine is the heart of the environment.**



Browser can execute your JS code as it has a JS engine.
Node.js->It is open source javascript runtime used to run javascript code outside the browser.
Let's say we want to run JS inside a watercooler , we would need a JS runtime environment inside the cooler.
The APIs might be different though.
The APIs give us superpowers which we can use inside our JS code.
The browsers have an API known as localStorage. It may be different in nodejs.
setTimeout() might have different implementations inside node and browser.

List of JS engines
https://en.wikipedia.org/wiki/List_of_ECMAScript_engines

Notable engines [edit]

Further information: List of ECMAScript engines

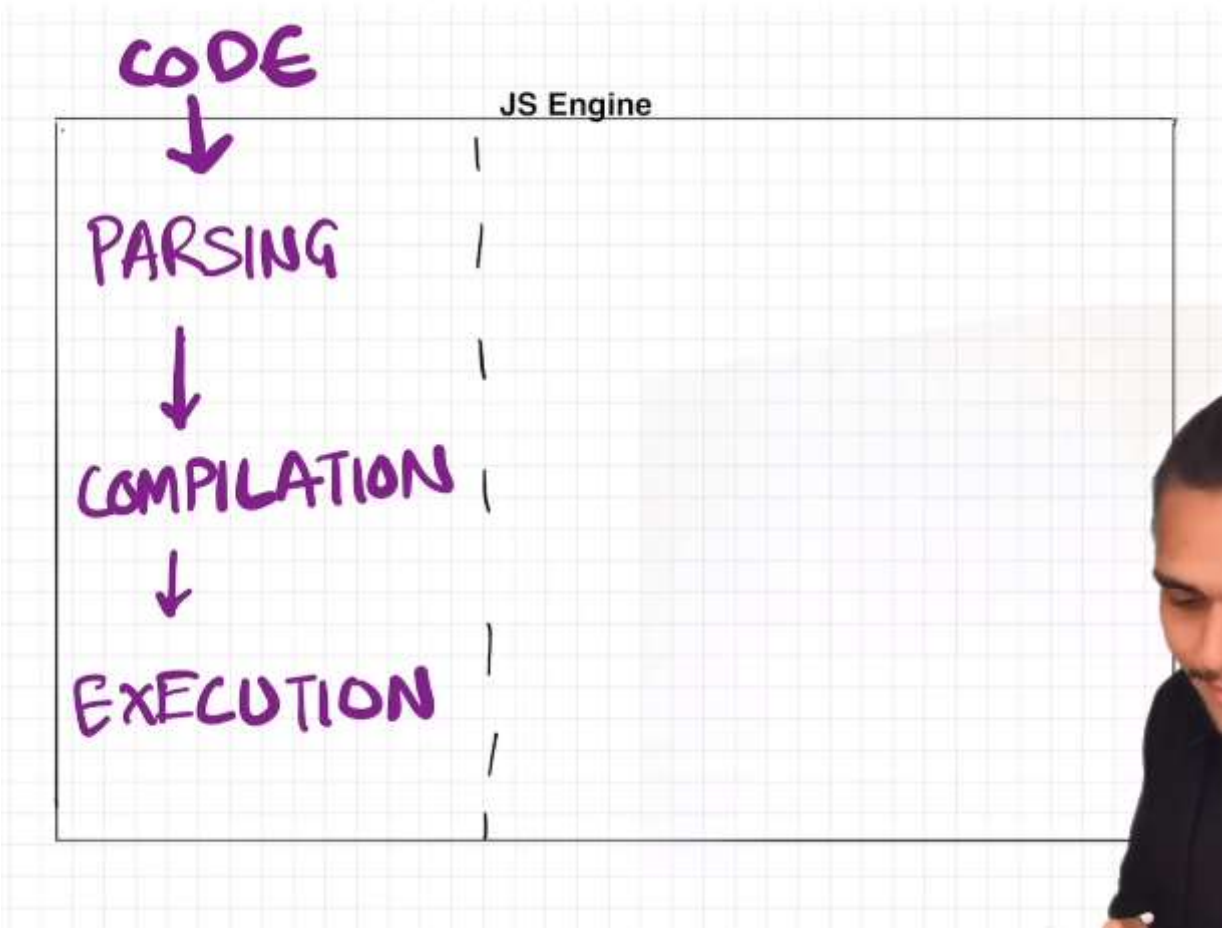
- V8 from Google is the most used JavaScript engine. Google Chrome and the many other Chromium-based browsers use it, as do applications built with CEF, Electron, or any other framework that embeds Chromium. Other uses include the Node.js and Deno runtime systems.
- SpiderMonkey is developed by Mozilla for use in Firefox and its forks. The GNOME Shell uses it for extension support.
- JavaScriptCore is Apple's engine for its Safari browser. Other WebKit-based browsers also use it. KJS from KDE was the starting point for its development.[7]
- Chakra is the engine of the Internet Explorer browser. It was also forked by Microsoft for the original Edge browser, but Edge was later rebuilt as a Chromium-based browser and thus now uses V8.[8][9]

https://www.geeksforgeeks.org/introduction-to-javascript-engines/

We can create our JS engine as well by following standards set by ECMAScript which is the governing body for javascript.
The JS engine was created by the creator of JS->Breandan Eich.This engine,built at Netscape Communications Corporation, is now evolved and known as SpiderMonkey.

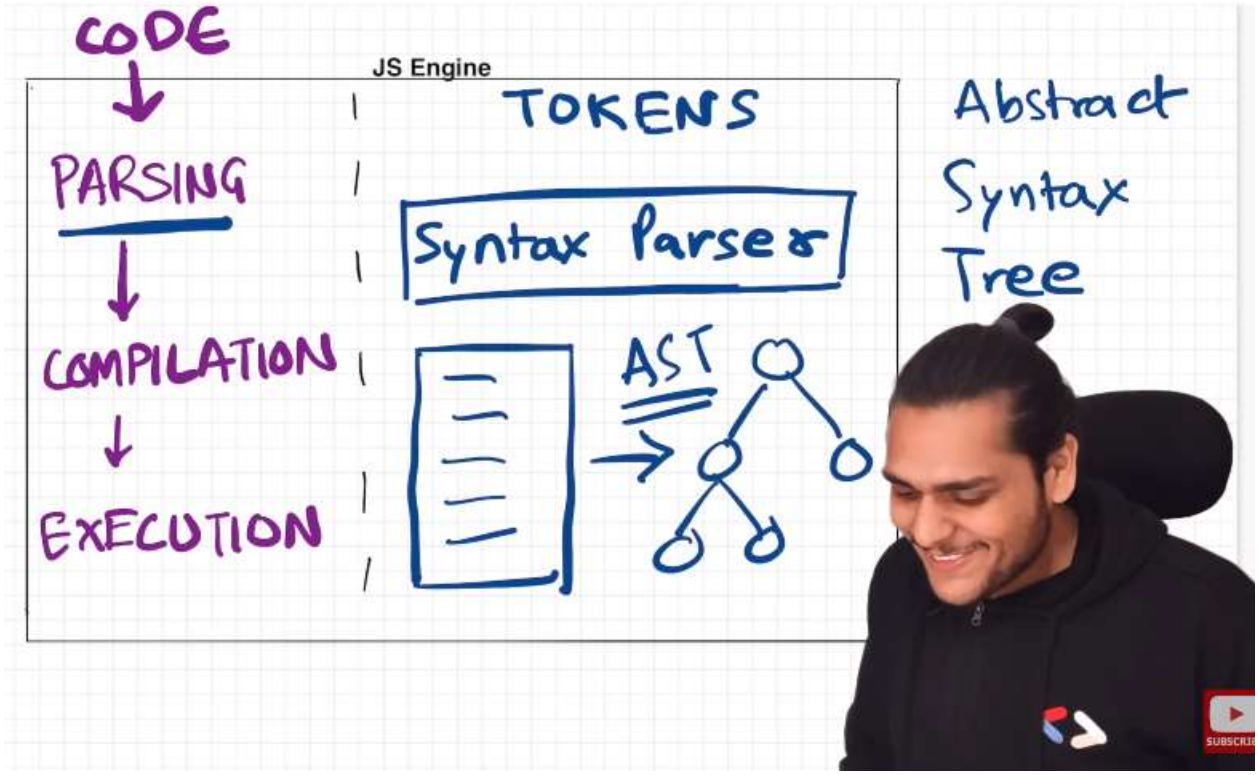
JS ENGINE IS NOT A MACHINE.
It is a code. The V8 engine is written in C++.

JS ENGINE ARCHITECTURE

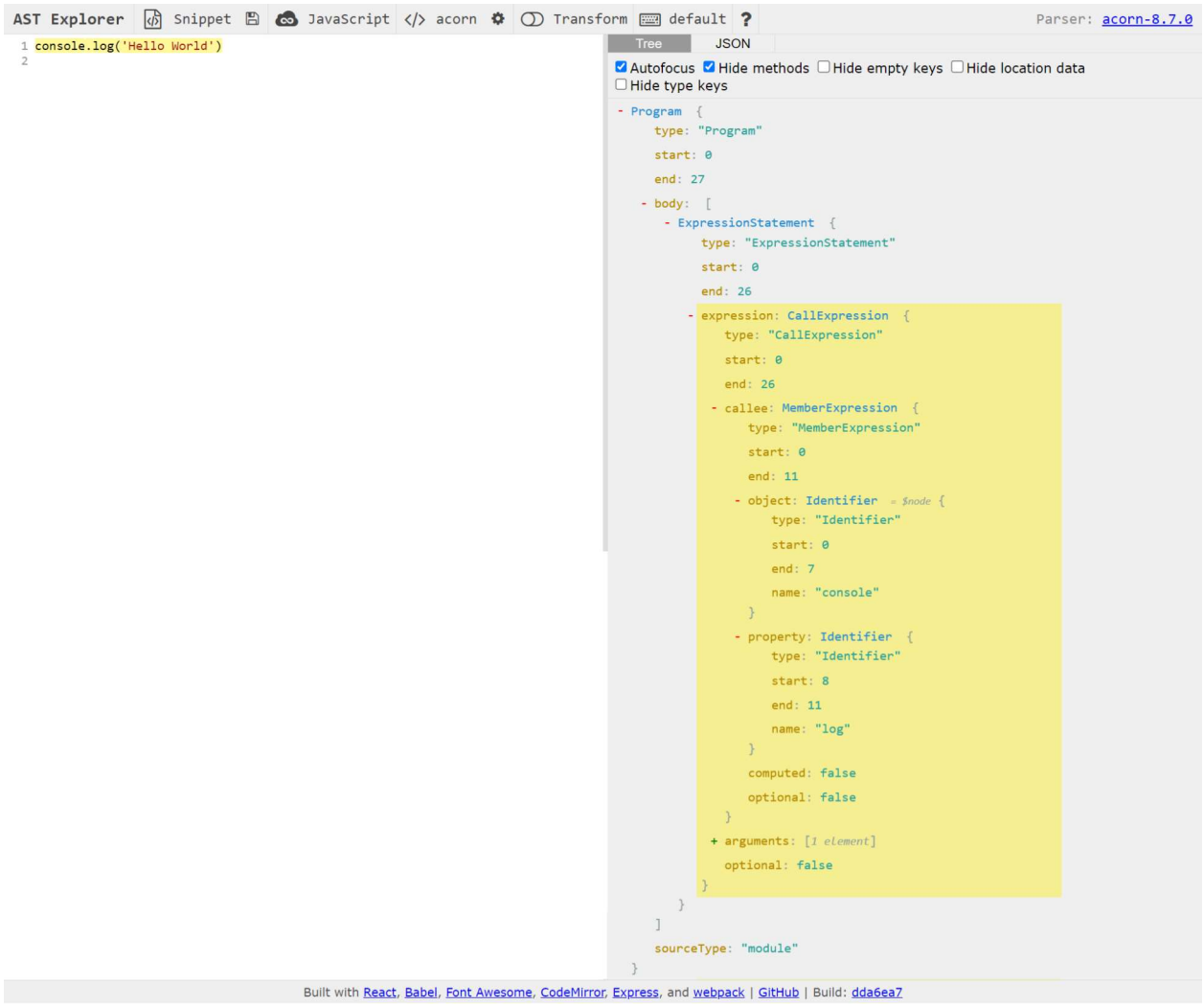


During the parsing phase, code is broken down into tokens.

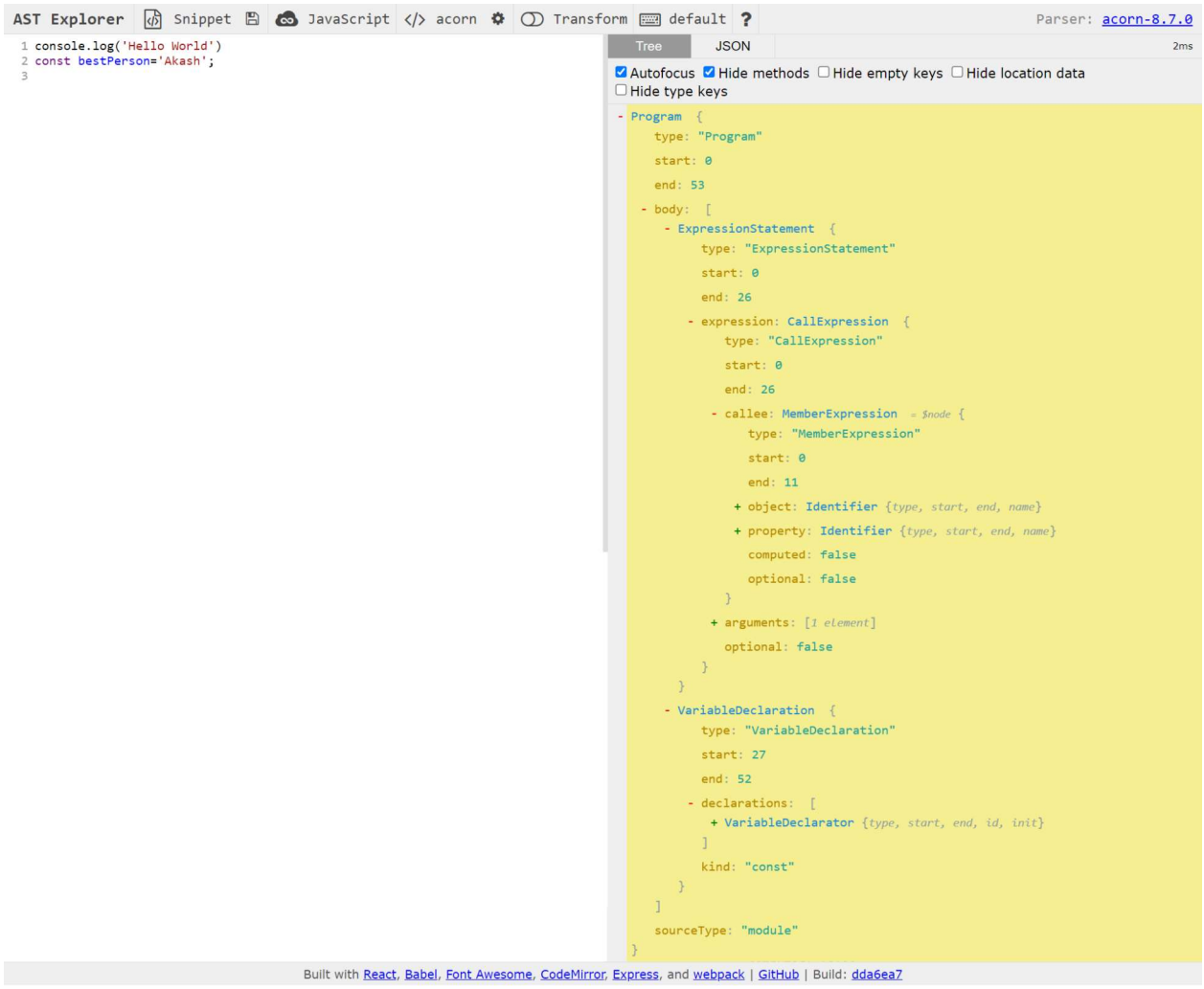
There is syntax parser.
It takes code and converts it to an abstract syntax tree(AST).

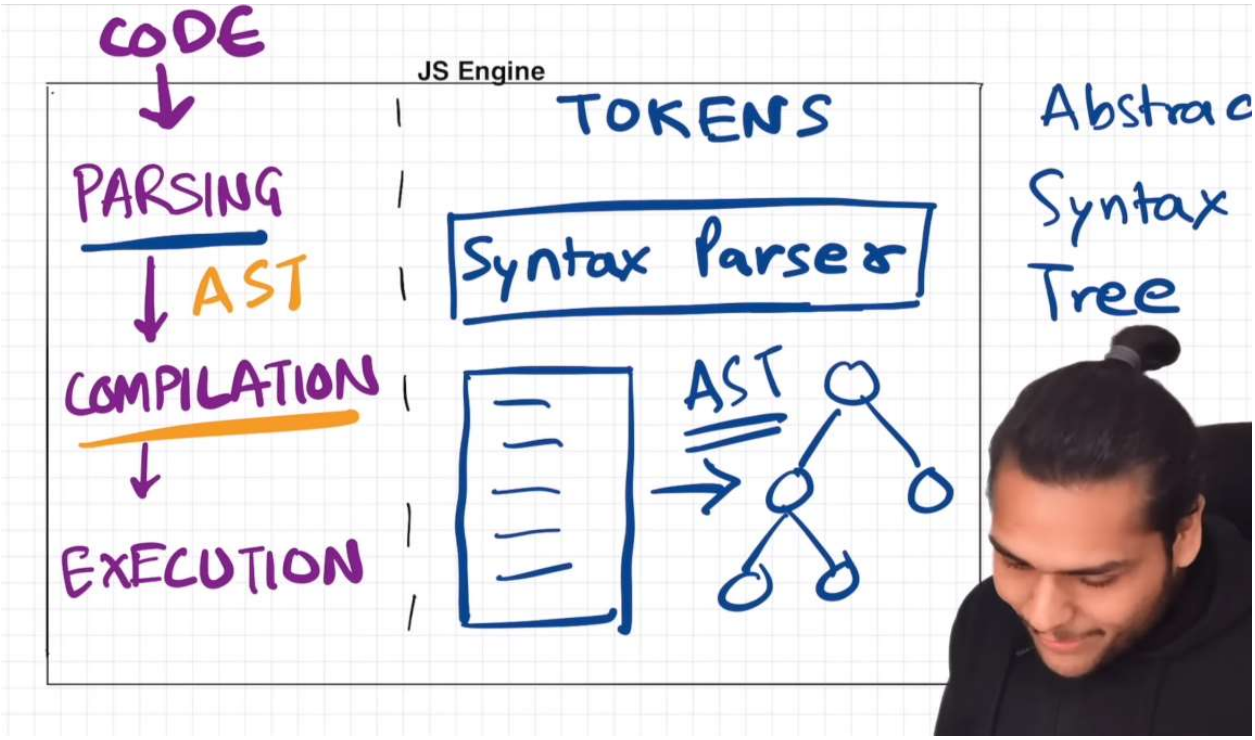


On this site we can get the abstract syntax tree
https://astexplorer.net/
For the code in left hand side we get abstract syntax tree on right hand side.



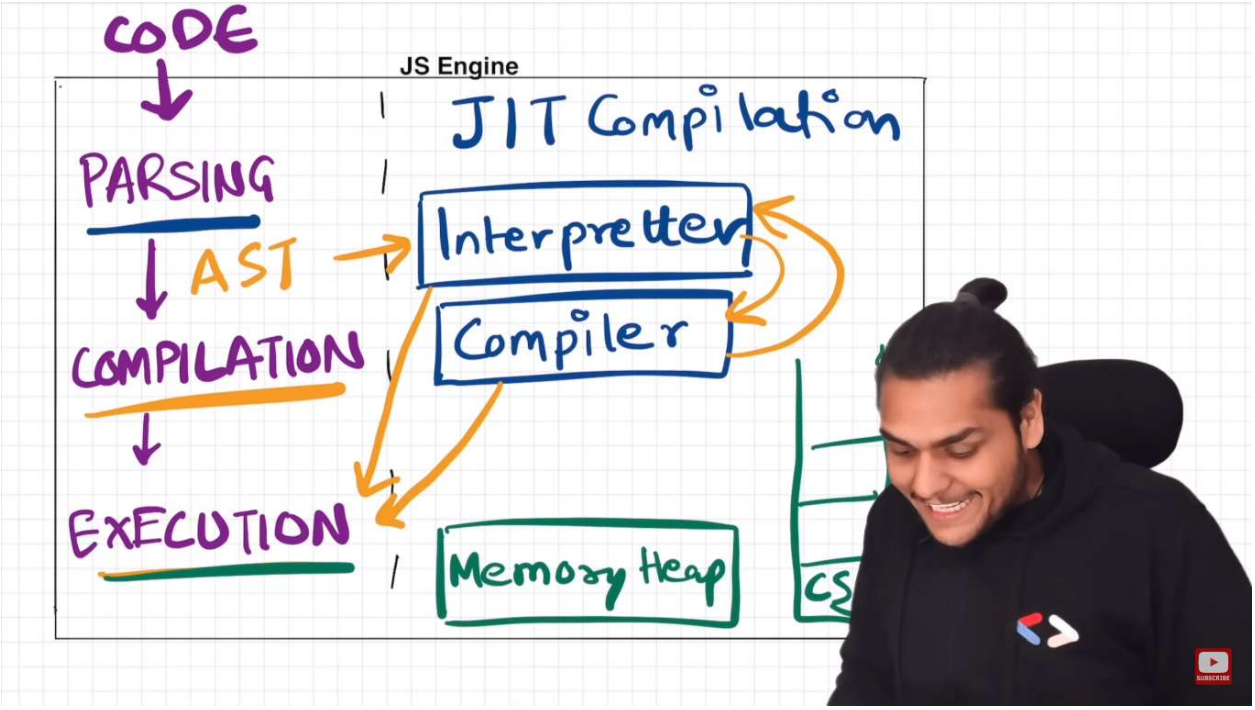
Syntax Parser generates the abstract syntax tree after parsing phase. Another one


AST Explorer
Snippet
JavaScript
acorn
Transform
default
Parser: acorn-8.7.0
1 console.log('Hello World')
2 const bestPerson='Akash';
3
Tree
JSON
2ms
Autofocus
Hide methods
Hide empty keys
Hide location data
Hide type keys
- Program {
type: "Program"
start: 0
end: 53
- body: [
- ExpressionStatement {
type: "ExpressionStatement"
start: 0
end: 26
- expression: CallExpression {
type: "CallExpression"
start: 0
end: 26
- callee: MemberExpression = $node {
type: "MemberExpression"
start: 0
end: 11
+ object: Identifier {type, start, end, name}
+ property: Identifier {type, start, end, name}
computed: false
optional: false
}
+ arguments: [1 element]
optional: false
}
}
- VariableDeclaration {
type: "VariableDeclaration"
start: 27
end: 52
- declarations: [
+ VariableDeclarator {type, start, end, id, init}
]
kind: "const"
}
]
sourceType: "module"
}
Built with React, Babel, Font Awesome, CodeMirror, Express, and webpack | GitHub | Build: dda6ea7

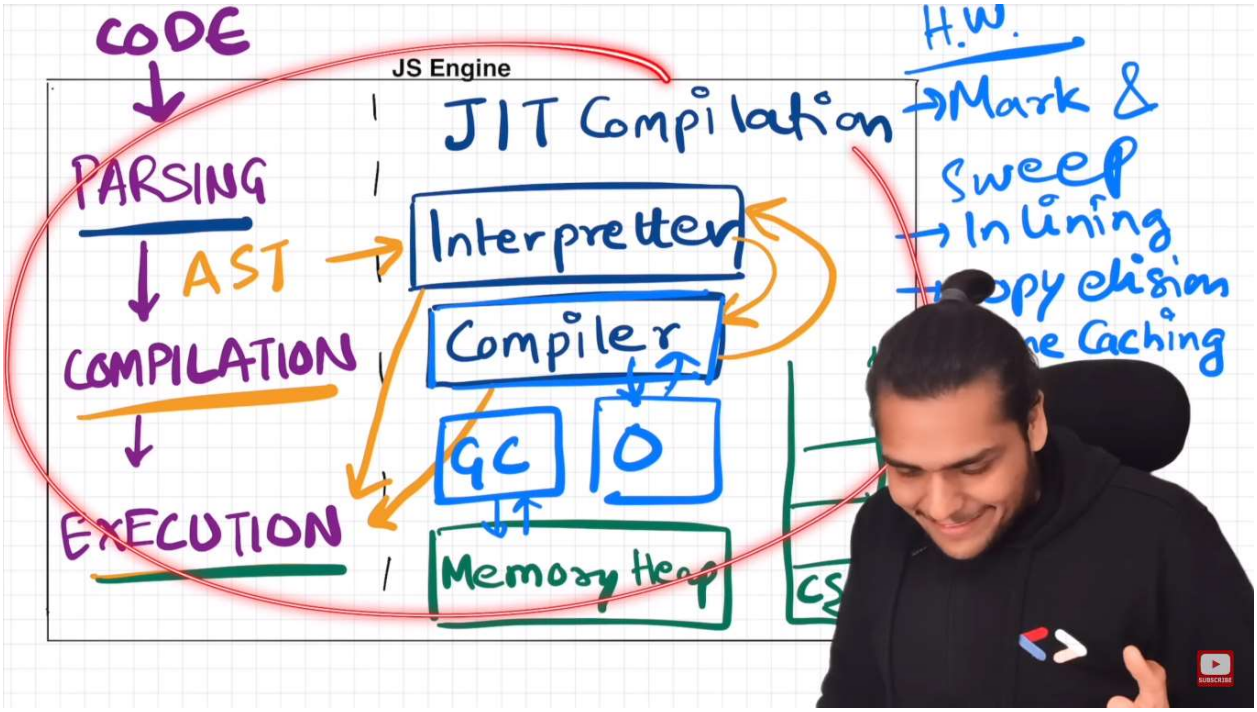


Coming to the compilation phase,
JS has Just in time compilation.
Now let's see what is interpreter and compiler.
Interpreter executes code line by line.
In compilation whole code is compiled before executing.
The compiled code is optimized version of this code that has performance improvements.
https://www.geeksforgeeks.org/difference-between-compiler-and-interpreter/
JS can behave as both compiled and interpreted.
When it was created, it was supposed to be interpreted language.
Because JS would run on browsers and browsers cannot keep the machine code and then execute it.
Modern browsers uses compiler and interpreter both.
This is done by JIT compilation.
Abstract Syntax Tree goes to interpreter. Interpreter converts our high level code to byte code.Byte code moves to execution step.
Compiler basically tries to optimize the code(inlining,inline caching,copy elision) as much as it can while interpreter is optimizing the code.This is not a 1-phase process but happens in multiple phases.
In some engines there is AOT Ahead of Time compilation. Compiler takes a piece of code which is going to be executed later and tries to optimize it as much as it can.

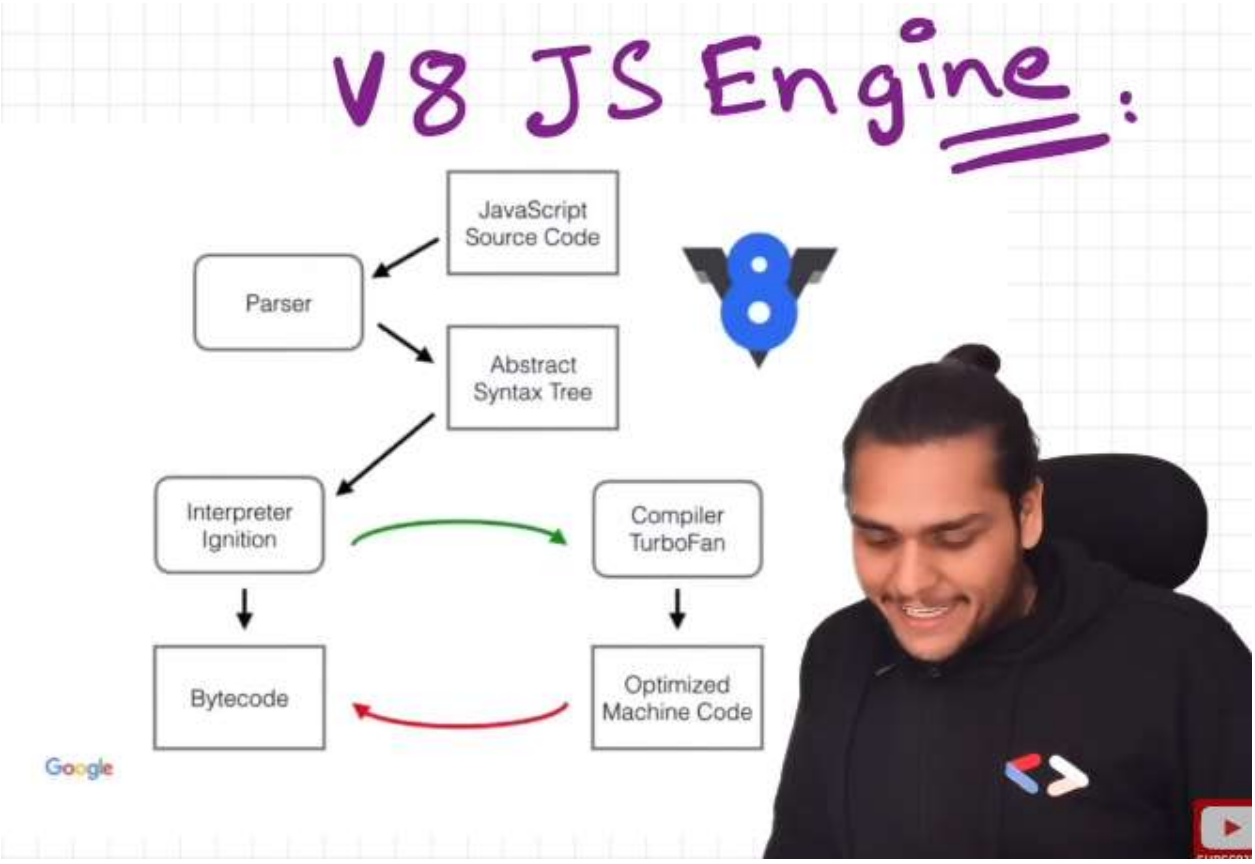
In the execution phase , there is memory heap and call stack.



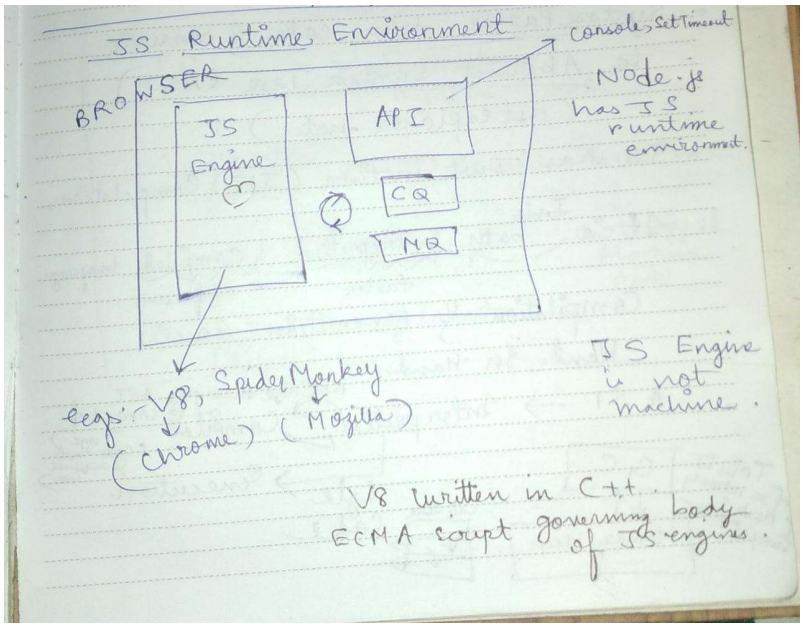
Memory heap is in sync with garbage collector, call stack and lot of other things.
In heap,variables and functions are assigned memory.
Garbage collector uses mark and sweep algorithm to sweep unused intervals,objects,functions.
FACTS->
V8 has ignition interpreter.
Turbo Fan is optimizing compiler of V8
Orinoco is V8garbage collector.
Oilpan is a garbage collector .

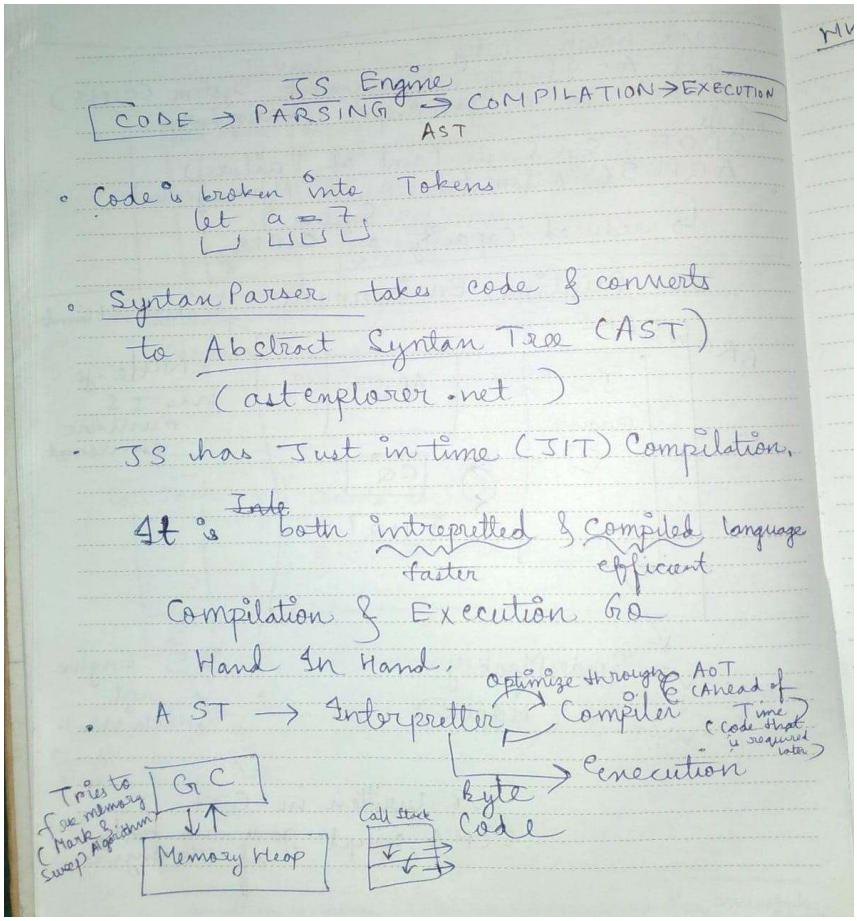


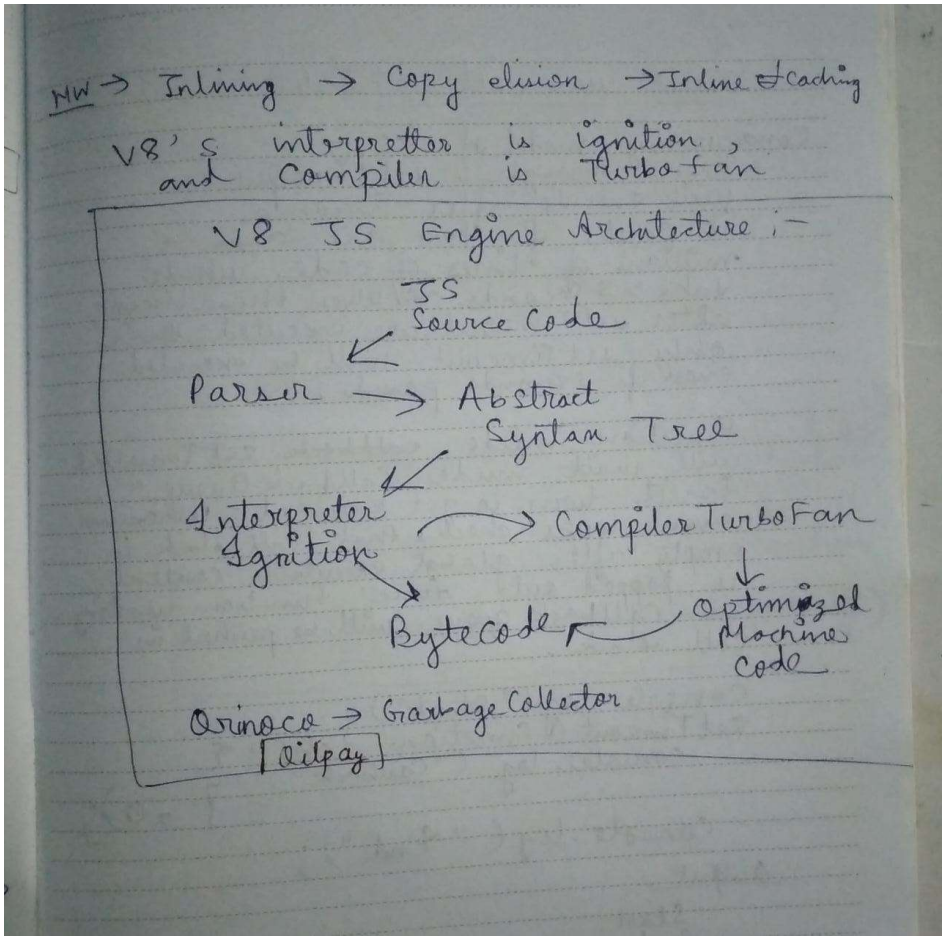
V8 engine architecture.



HANDWRITTEN NOTES





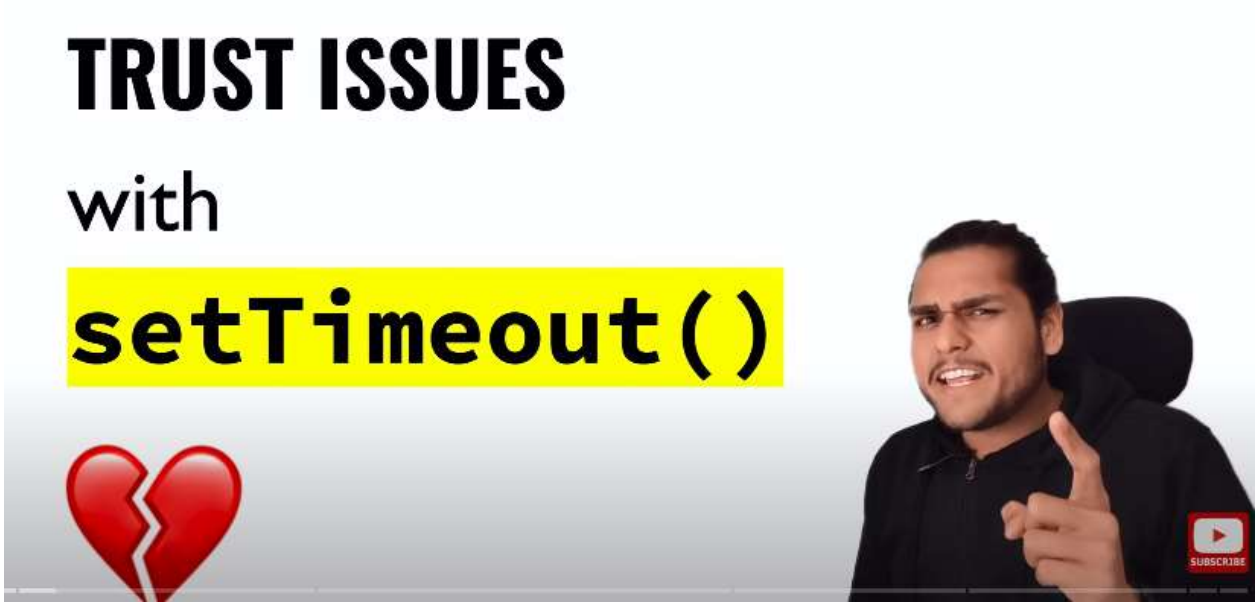


Conclusion

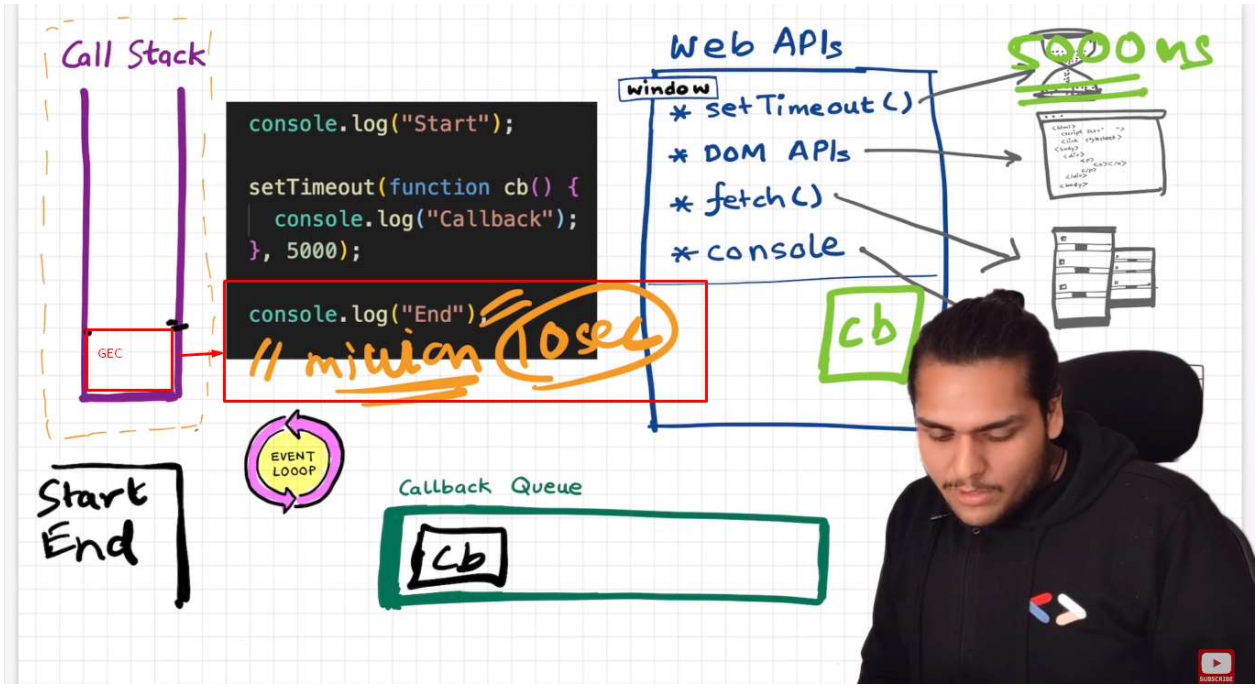
1. The JS runtime environment contains all elements required to run JS.
2. It contains a JS engine, set of API's, callback queue, microtask queue, event loop.
3. The JS engine is a piece of code.
4. Process includes Parsing ---> Compilation -----> Execution.
5. Parsing breaks code into tokens and converts it into AST(Abstract Syntax Tree).
6. Modern JS engine follows JIT compilation, it interprets while it optimises code as much as it can.
7. Execution and Compilation are done together.
8. Execution has Garbage collector and other optimisation such as inlining, copy elusion, inline caching etc.

TRUST ISSUES with setTimeout() | Namaste JavaScript Ep.17

Youtube Link-https://youtu.be/nqsPmuicJJc

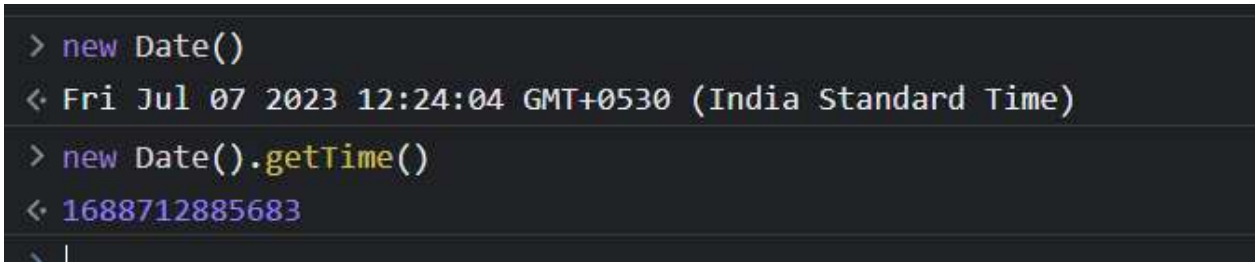


A setTimeout with a delay of exactly 5 seconds does not always exactly wait for 5 seconds.It might also take 6 seconds.It depends on call stack.



In the above diagram , we see callback function being registered in web APIs environment and getting inside the callback queue.
Now let's say there are millions of lines of codes below, which take 10 seconds to be executed. So then,even though the timer of callback function expires,it has to wait until the call stack is empty. When the global execution context is popped off after 10 seconds, then event loop will push the callback function to the call stack. This is known as *CONCURRENCY MODEL* in javascript. This was why we had trust issues with setTimeout. We also should not block the main thread because if call stack does not get empty it won't be able to process any other event.

Now,let us simulate above by blocking main thread for 10 seconds,so that we can setTimeout does not always wait for 5 seconds.
We are just seeing how Date API works.



getTime() returns the number of milliseconds since January 1, 1970 00:00:00.
Now we want a loop to run for 10 seconds.

```
console.log('Start')
setTimeout(function cb(){
    console.log('Callback')
},5000)
console.log('End')

let startDate=new Date().getTime()
let endDate=startDate
while(endDate<startDate+10000){
    endDate=new Date().getTime()
}
console.log('while expires')
```

The while loop runs for 10 seconds.
We have added 10,000 to startDate as it is 100000 milliseconds.
endDate keeps on calculating the current number of seconds. As soon as endDate is 10 seconds more than startDate,loop is over,. While expires will be printed on console and global execution context is popped off.Then ,callback function,which was waiting in callback queue will be put in call stack after 10 seconds even though it had expired after 5 seconds.

This was an example of blocking the main thread.
Why JS has only 1 call stack?
It has only 1 main thread. That is why it is synchronous single threaded.It is fast,interpreted.

Now what is setTimeout with timer of 0 seconds?

```
console.log('Start')
setTimeout(function cb(){
    console.log('Callback')
},0)
console.log('End')
```

Even though timer is 0 seconds,the callback function will be registered in the web APIs environment.
It has to still go through the callback queue.
It will be then executed only when the call stack is empty.

```
Start
End
Callback
>
```

Qs is why we set timer of 0?

We could have done this also

```
function cb() {
  console.log("Callback");
}
cb();
```

.

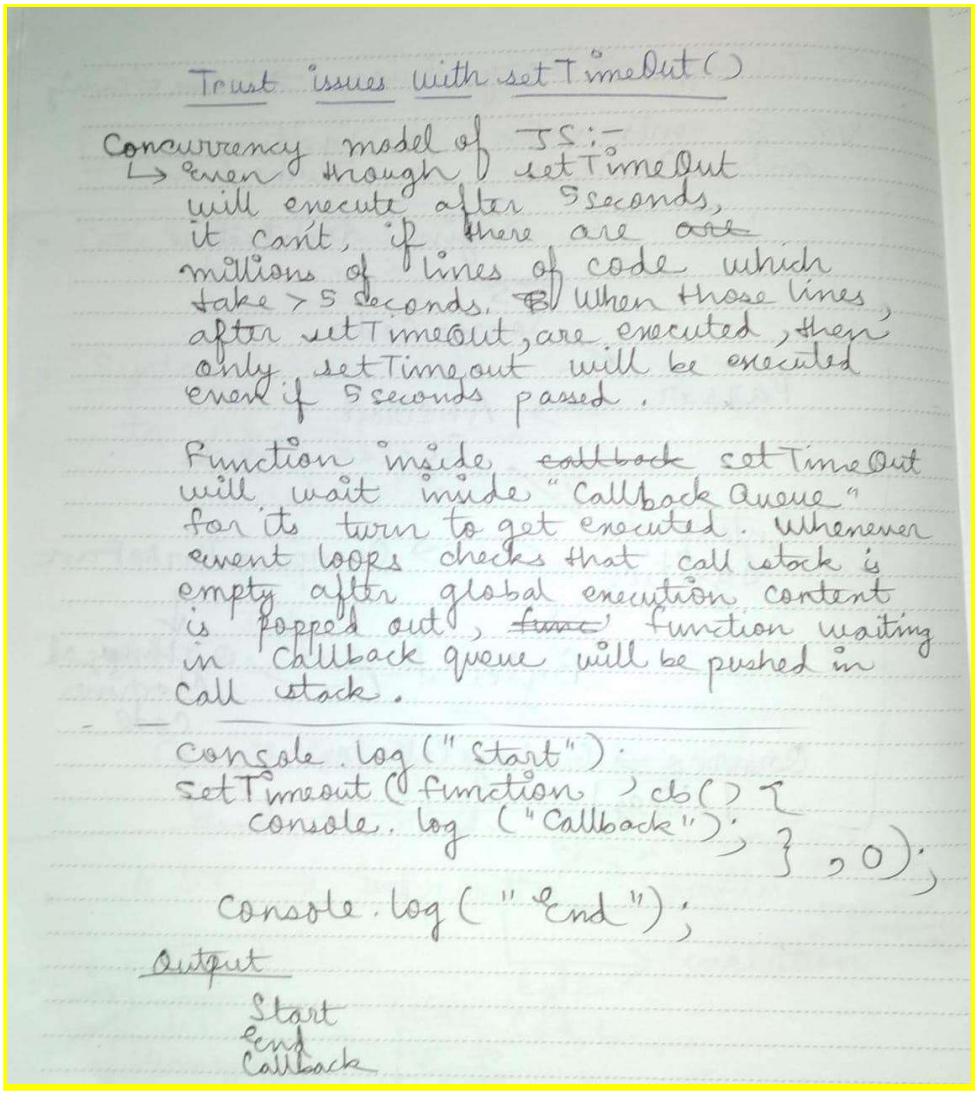
We use timer of 0 when we want to *defer a piece of code.*
Defer means→to leave something until a later time
May be when cb is not that important,then we can defer the function.
For example if we want to show something on our page at the end,then we can defer it.

```
console.log('Start')
setTimeout(function cb(){
    console.log('Callback1')
},0)
setTimeout(function cb(){
    console.log('Callback2')
},0)
console.log('End')
```

```
Start
End
Callback1
Callback2
```

The output is always this. 2 will never come before 1.

HANDWRITTEN NOTES



Trust issues with setTimeOut()

Concurrency model of JS:-
↳ even though setTimeOut will execute after 5 seconds, it can't, if there are millions of lines of code which take > 5 seconds. When those lines after setTimeOut, are executed, then only setTimeout will be executed even if 5 seconds passed.

Function inside ~~callback~~ setTimeOut will wait inside "Callback Queue" for its turn to get executed. Whenever event loops checks that call stack is empty after global execution content is popped out, ~~func~~ function waiting in Callback queue will be pushed in Call stack.

```
console.log("Start");
setTimeout(function cb(){
    console.log("Callback");
},0);
console.log("End");
```

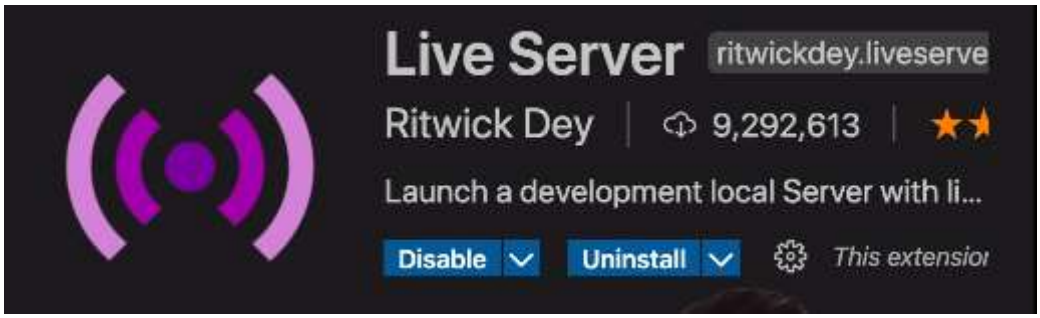
Output

Start
End
Callback

Setup used

VS CODE, Google Chrome.

2 files index.html,index.js and extension live server on vs code.



Higher-Order Functions ft. Functional Programming | Namaste JavaScript Ep. 18

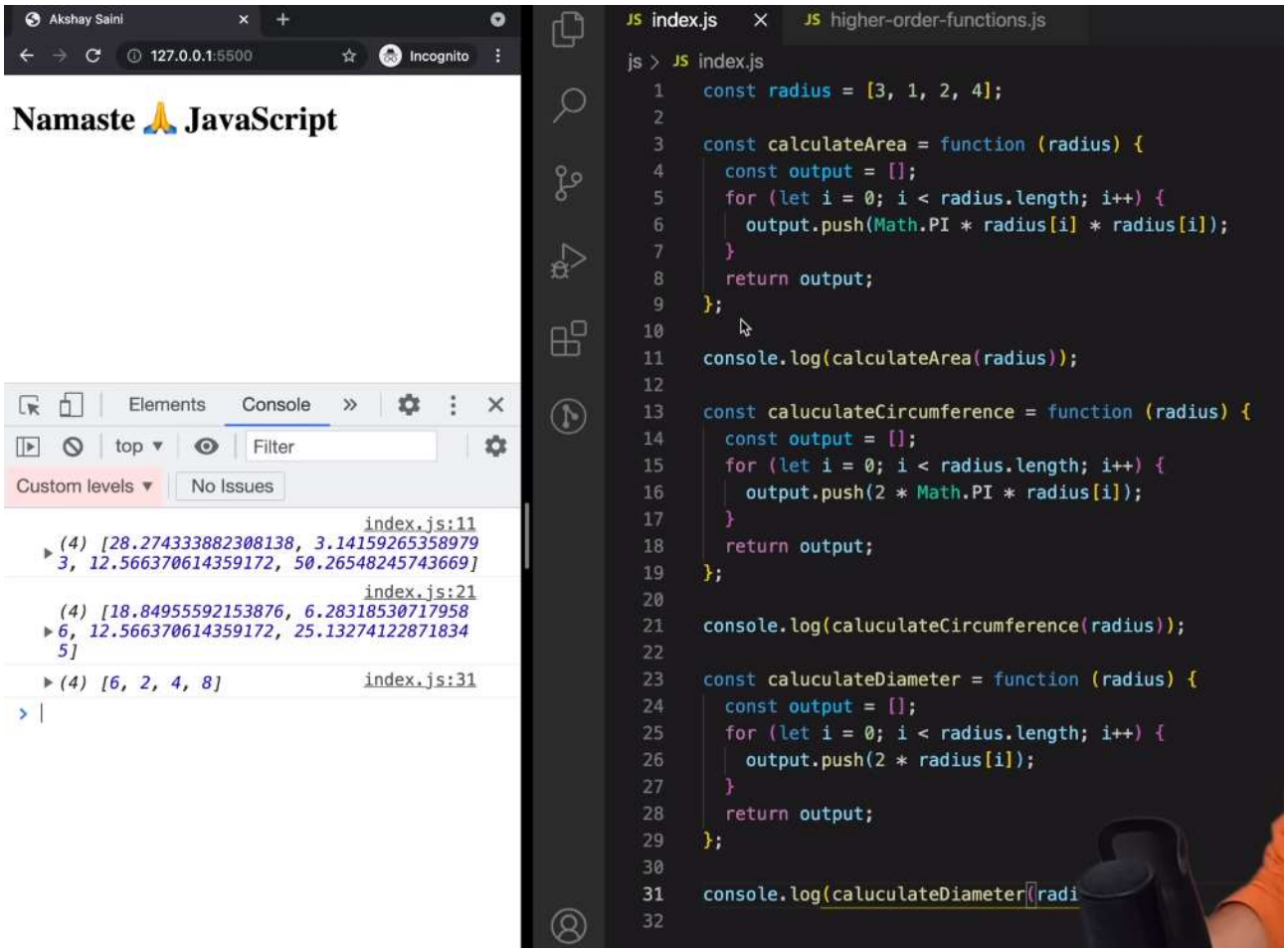
Youtube Link-https://youtu.be/HkWxvB1RJq0

A function that takes another function as an argument or returns another function from it is higher order function.

```
function x() {
  console.log("Namaste");
}

function y(x) {
  x();
}
```

y is higher order function. x is callback function.

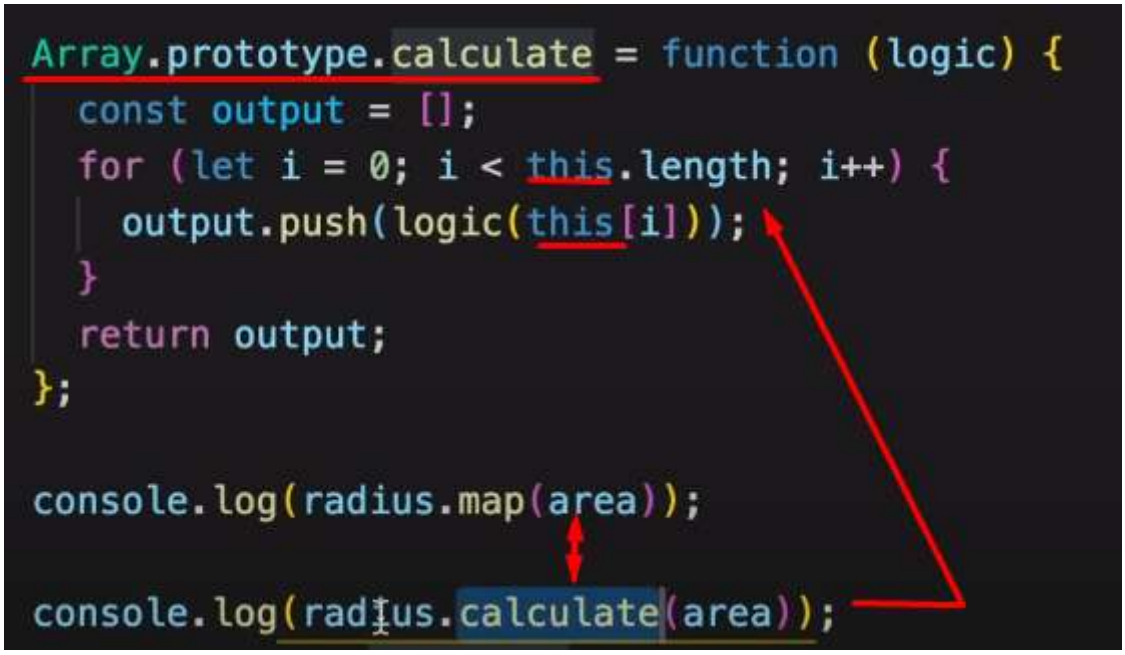
Lets say we are asked to calculate area,circumference,diameter of each circle.



Writing code in above way would be a problem as we do not write in optimized way and we do not follow DRY principle.
We are running similar type of loop,inserting in similar way again and again.
The only thing that changes is what we insert in output array.

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
  return Math.PI * radius * radius;
};

const cicumference = function (radius) {
  return 2 * Math.PI * radius;
};

const diameter = function (radius) {
  return 2 * radius;
};

const calculate = function (radius, logic) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(logic(radius[i]));
  }
  return output;
};

console.log(calculate(radius, area));
console.log(calculate(radius, cicumference));
console.log(calculate(radius, diameter));
```

This is good code.
Calculate() is higher order function.
And area,circumference,diameter is callback function.
Lets make our function calculate same as map function.



Using Array.prototype is like writing a polyfill for map. Now it will be available for all arrays.
This was essence of functional programming->thinking of logic in terms of smaller functions and passing them into another function and treating functions as values.

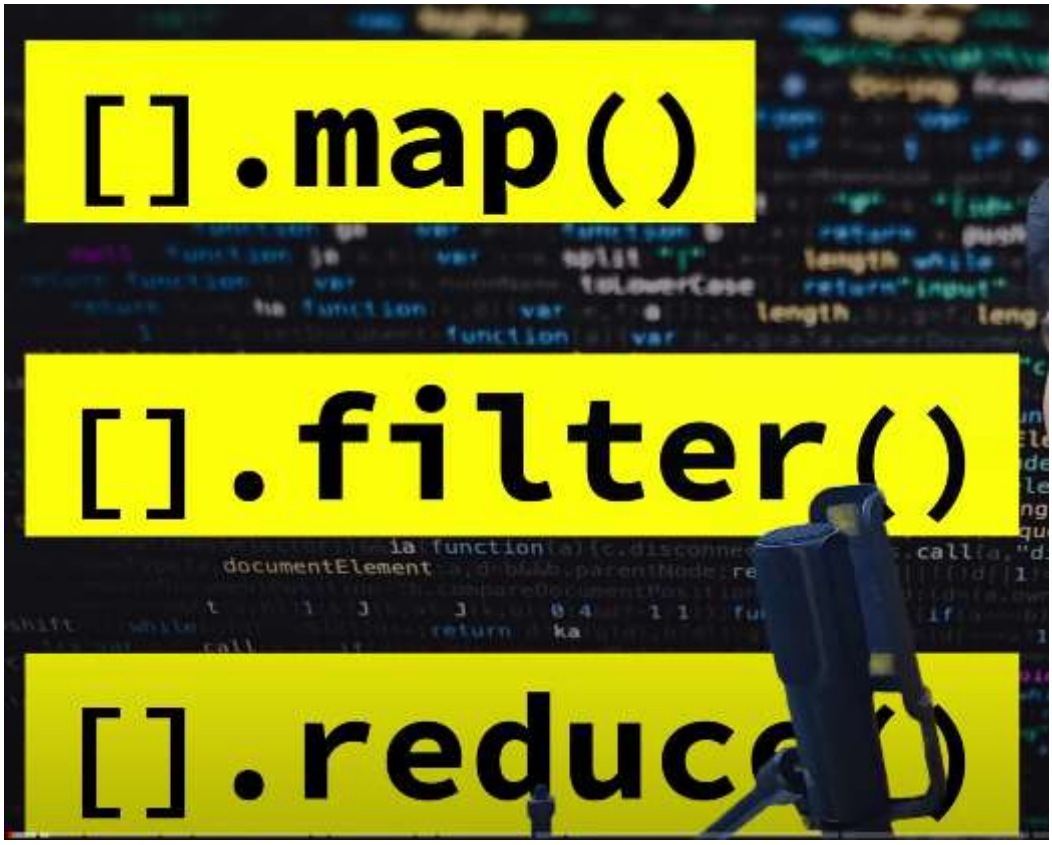
SUMMARY->

1. Follow DRY(Don't Repeat Yourself) principle while coding.
2. Use function to stop writing repeating line of codes.
3. Function that takes another function as argument(callback function) is known as Higher order functions.
4. It is this ability that function can be stored, passed and returned, they are called first class citizens.
5. If we use Array.property.function-name. This function is accessible to any array in your code.
6.First-class functions are JavaScript functions that can behave like variables. They can also be passed as arguments to higher-order functions.

---

map, filter & reduce 🙏 Namaste JavaScript Ep. 19 🔥

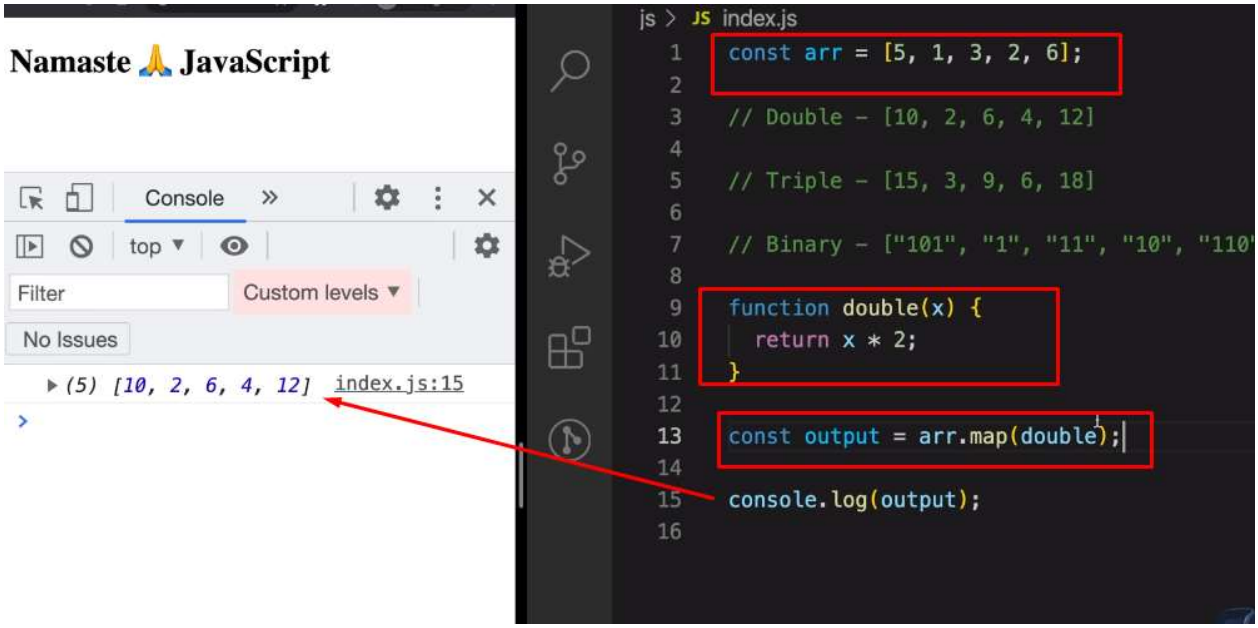
Youtube Link- https://youtu.be/zdp0zrpKzlE



These are higher order functions in javascript.
Suppose we want to transform an array.
Like double or triple all elements then we use map.

```
const arr = [5, 1, 3, 2, 6];

// Double - [10, 2, 6, 4, 12]

// Triple - [15, 3, 9, 6, 18]

// Binary - ["101", "1", "11", "10", "110"]
```

Map takes a function as an argument which will be applied to each element in array.



We can write it like this as well for converting decimal to binary

```
const output = arr.map(function binary(x) {
  return x.toString(2);
});
```

Or like this

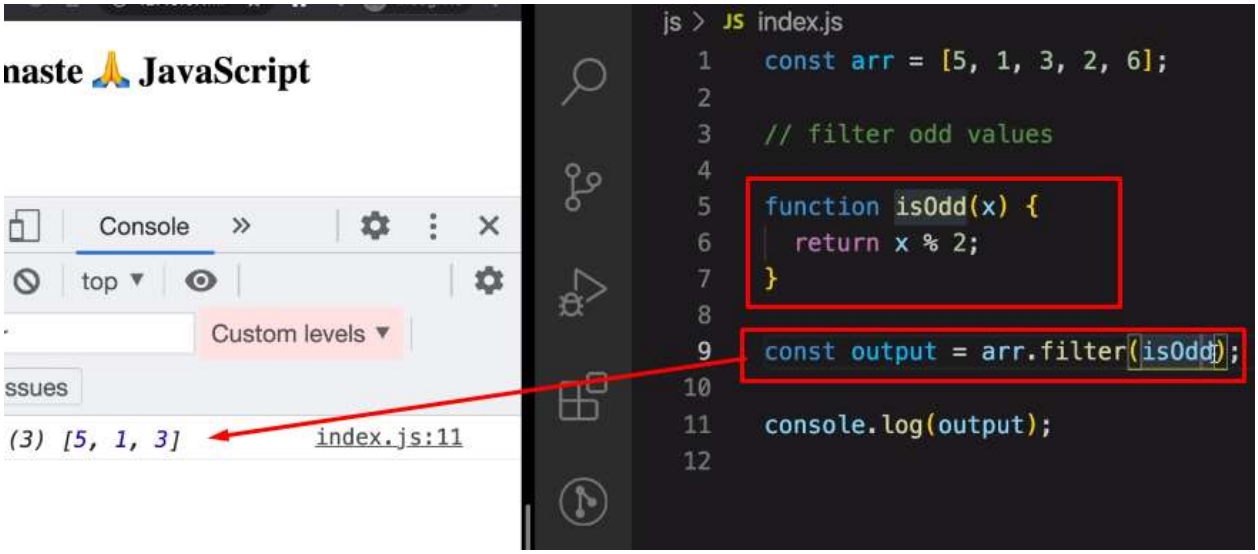
```
const output = arr.map((x) => {
  return x.toString(2);
});
```

Or like this

```
const output = arr.map((x) => x.toString(2));
```

NOW WE USE FILTER

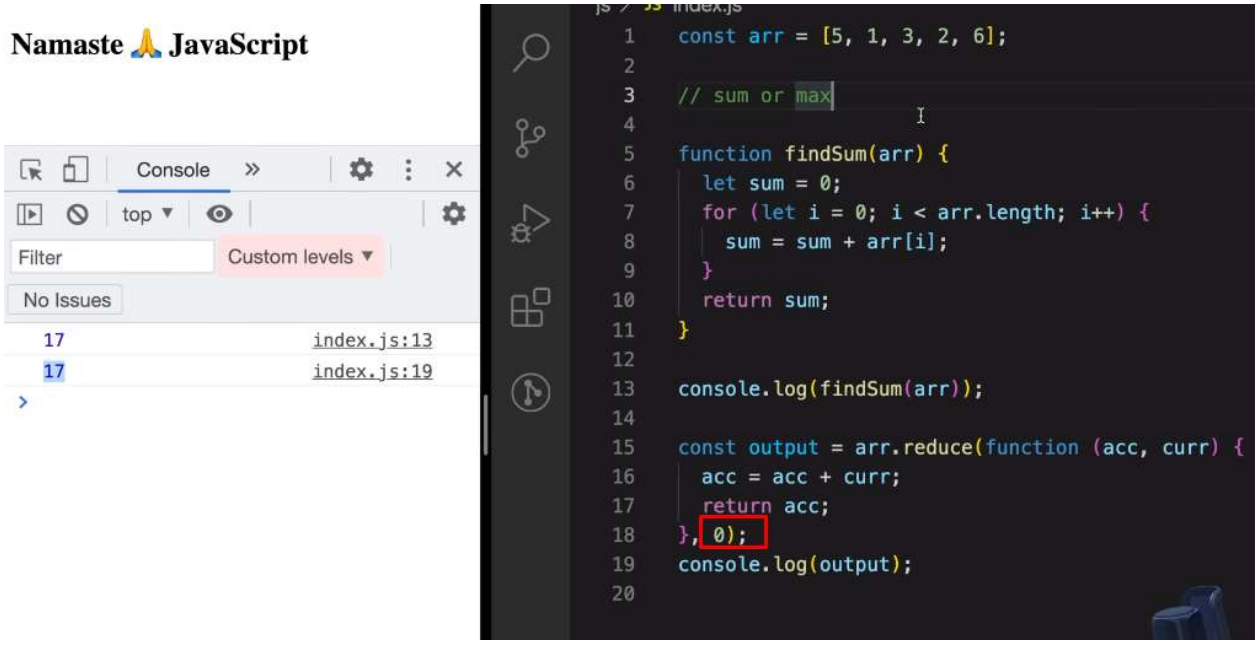
Filters out the values as per condition..returns an array of filtered values



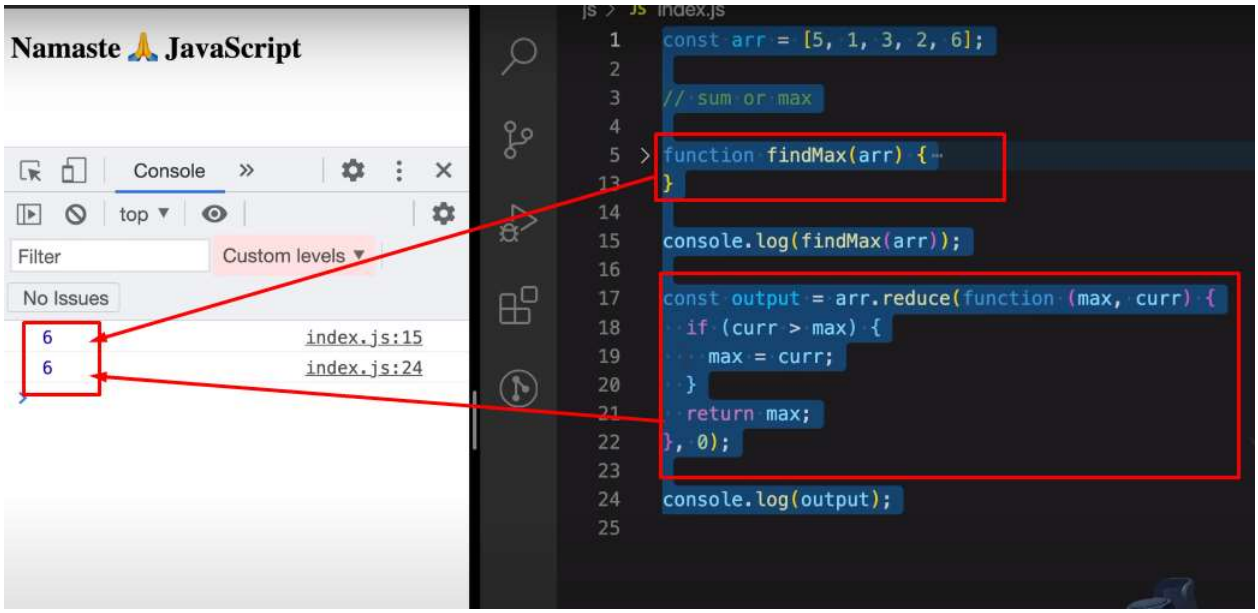
All values greater than 4 need to be filtered out.

```
const arr = [5, 1, 3, 2, 6];

const output = arr.filter((x) => x > 4);

console.log(output);
```

NOW WE WILL SEE REDUCE FUNCTION.
reduce() returns a value so we use output variable to hold it.



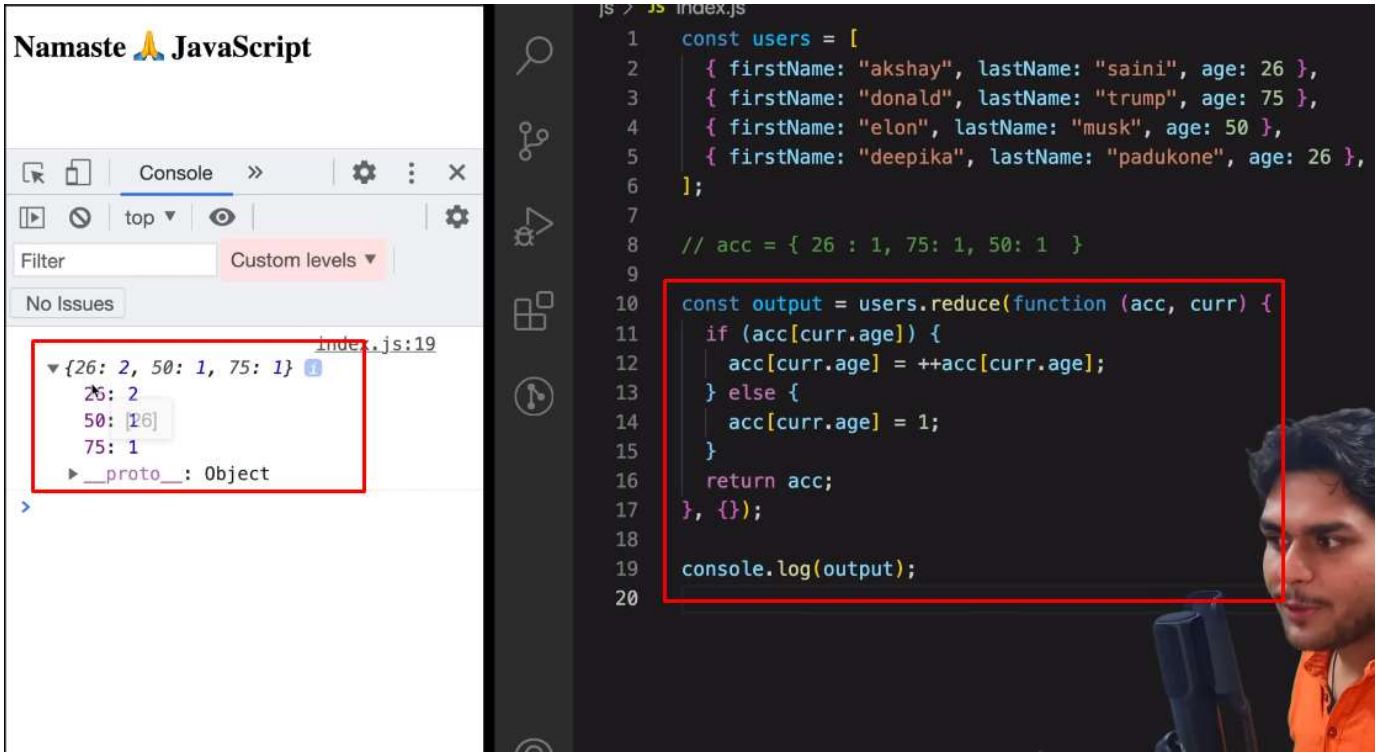
acc is accumulator.
curr is current.
curr is arr[i]
acc is sum.
We also pass 0 as an initial value to acc as 2nd argument to reduce.
Also do not forget to return acc inside function.



We find max value in arr using reduce function.
We do not get an array in output instead we get a value.

Listing the full names of people by mapping over array of objects.

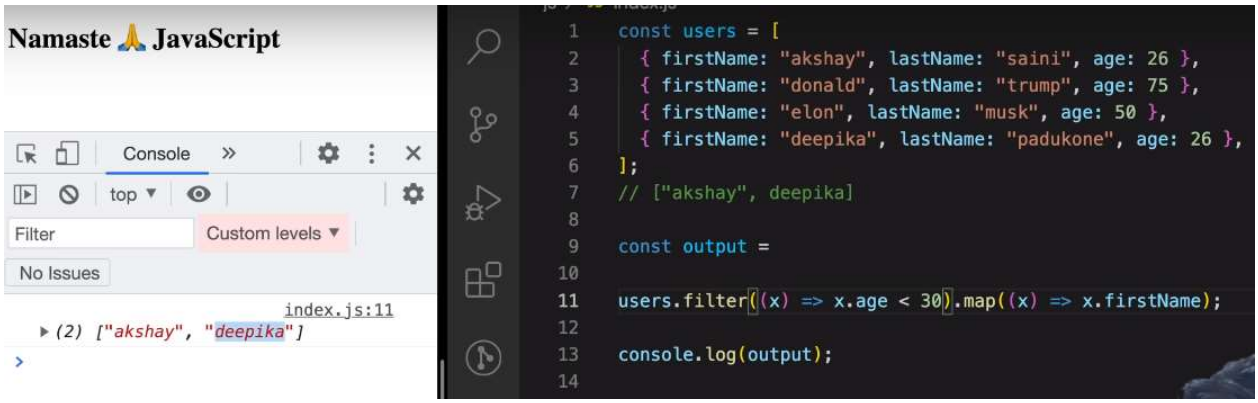
Now find the frequency of age i.e. create key value pair of age and its frequency.



Thus we can create a dictionary type of thing or a map like thing.

Now we will learn chaining of map, filter, reduce.

Suppose we want the first names of people having age <30



<u>Do above using reduce</u>

```
const output = user.reduce(function(acc, curr){
    if (curr.age < 30){
        acc.push(curr.firstName);
    }
    return acc;
```

}, [ ])

RECAP-

- **Return type of map,filter is an array of values**
- **Return type of reduce is value.**
- **All are higher order function that take function as argument.**
- **Map(),filter() require 1 argument.**
- **reduce() takes 2 arguments-1 function,1 initial value of acc.**
- **map method is used when we want transformation of whole array.**
- **filter is used when we want to filter the arrar to obtain required value.**
- **reduce is used when we want to reduce the array to single value eg (max, min, avg, sum, difference etc).**
- **reduce passes two arguments one function(which includes accumulator and initial value as argument itself) and another initial value of accumulator.**

See polyfill of map(),filter(),reduce().

NAMASTE JAVASCRIPT SEASON 1 ENDS HERE.

Akshay Saini Namaste Javascript Season 2

https://www.youtube.com/playlist?list=PLlasXeu85E9eWOpw9jxHOQyGMRiBZ60aX

Playlist Link

Callback Hell | Ep 01 Season 02 - Namaste JavaScript

Youtube Link- https://youtu.be/yEKtJGha3yM

We are going to cover
Good and Bad Part of Callbacks

Callbacks are important in writing asynchronous code in JavaScript.
2 issues we face-

- **Call Back Hell**
- **Inversion of Control**

```
setTimeout(function () {
    console.log("JavaScript");
},5000)
```

Here function() is callback function that needs to be executed after 5 seconds by set timeout,

---

We need to create an order using create order api.
And once order is created then only we can proceed to payment.
There is a dependency between them.

```
const cart = ["shoes", "pants", "kurta"];

api.createOrder()
api.proceedToPayment()
```

So to perform such async operations we need to take help of callback functions.

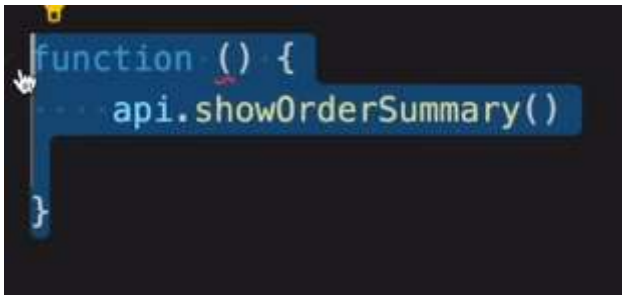
```
const cart = ["shoes", "pants", "kurta"];

api.createOrder(cart, function () {

    api.proceedToPayment()

})
```

So we pass proceedToPayment() as callback function to create order api.
Now its responsibility of createOrder api to create an order and call the proceedToPayment() .
Now lets say we want to show an order summary page too.

So we call the order summary api.

```
api.createOrder(cart, function () {

    api.proceedToPayment()

})


api.showOrderSummary()
```

But again orderSummary() needs to be called only after payment happens.
So we wrap it up inside a function ..



And pass it as callback function

```
const cart = ["shoes", "pants", "kurta"];

api.createOrder(cart, function () {
        any
    api.proceedToPayment(function () {

        api.showOrderSummary()

    })

})
```

Now we want to update wallet

Passing the updateWallet() to showOrderSummary().

```
api.createOrder(cart, function () {

    api.proceedToPayment(function () {

        api.showOrderSummary(

            function () {
                api.updateWallet()
            }

        )

    })

})
```

This is callback hell.
-One callback inside another callback
-Code starts to grow horizontally
-Unmaintainable code
-Structure is known as pyramid of dome.

INVERSION OF CONTROL
-Its like you loose the control of your code when you use callback.

While passing the callback function to another function,
Like while passing proceedToPayment() to createOrder(), we are becoming entirely dependent on createOrder() and we are sitting back and relaxing thinking createOrder will do the job and we do not know what goes on behind the scenes. Control is entirely in hands of createOrder function which is not good.
THIS IS RISKY. What if createOrder() never calls our callback function back or calls it twice.

RECAP-

1.) "Time, tide and JS waits for none"
2.) Callback function enables us to do async programming in JS. We use this for some functions that are interdependent on each other for execution. For eg: Ordering can be done after adding items in cart. So we pass callback functions as argument to functions which then call the callback function passed. However this causes some problems:

a.) Callback Hell: When a callback function is kept inside another function, which in turn is kept inside another function. (in short, a lot of nested callbacks). This causes a pyramid of doom structure causing our code to grow horizontally, making it tough to manage our code.

b.) Inversion of control: This happens when the control of program is no longer in our hands. In nested functions, one API calls the callback function received but we don't know how the code is written inside that API and how will it affect our code. Will our function be called or not? What if it is called twice? What if it has bugs inside it? We have given control of our code to other code.

Promises | Ep 02 Season 02 - Namaste JavaScript

Youtube Link-https://youtu.be/ap-6PPAuK1Y

Promises are used to handle asynchronous programming.
So to solve the problem of callback hell and inversion of control we use something called promises.

Now createOrder() will no longer take callback function. It will just take cart details and return us a promise.

```
const promise = createOrder(cart);
```

Promise is nothing but an empty object with some data value in it.
Data value will hold whatever create order api will return.

```
// {data: undefined }
```

After some time, createOrder will replace the undefined with some value.
This means program keeps executing,whenever data is got,it is put in value.
Now we attach a callback function to this promise object.
We use then to do this available on promise object.

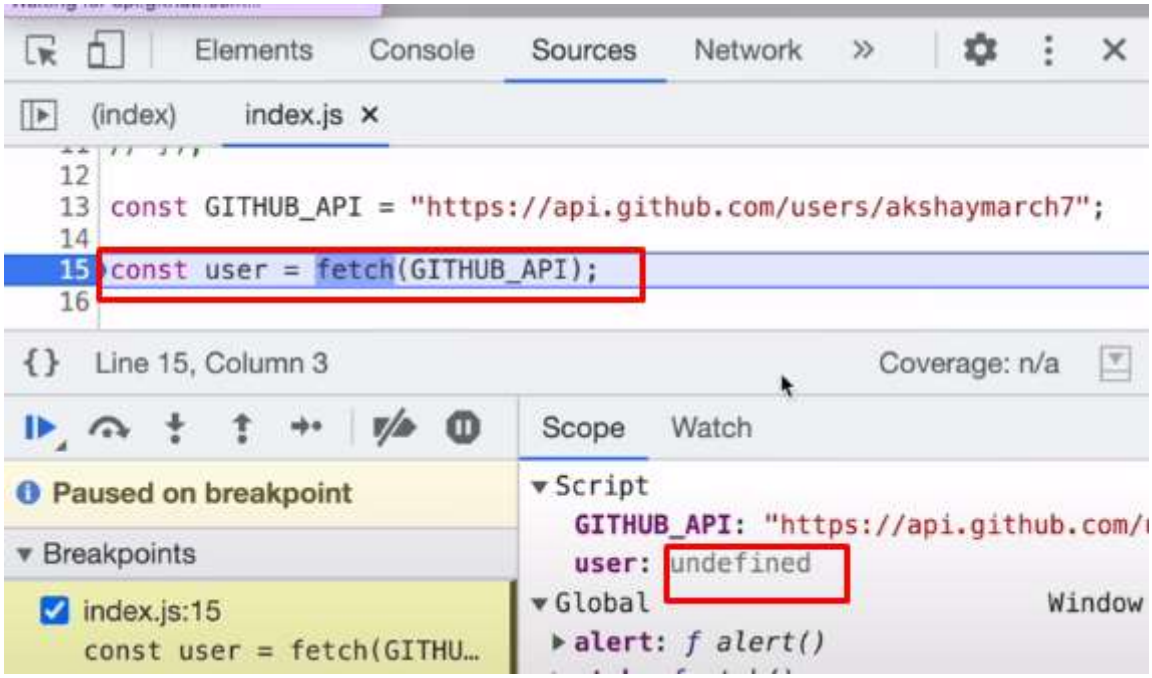
```
const promise = createOrder(cart);

promise.then(function (orderId) {
  proceedToPayment(orderId);
});
```

Now what's the difference with previous callbacks and promise shown now?
Earlier,
We were passing callback function to another function.
And here,
We are attaching callback function to a promise object.
Promises gives us the trust and guarantee that it will call the callback function back for us.
The control of program is now with us.
Whenever there is data inside promise object, it will call the callback function.
And it will call it only once.

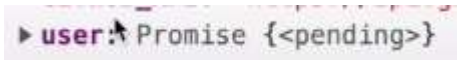
We will now study Promise in detail.
We will use fetch now.
Fetch will return us a promise and fetch is async operation.
So we fetch akshay sainis github details using github api.

```
const GITHUB_API = "https://api.github.com/users/akshaymarch7"

const user = fetch(GITHUB_API);
```

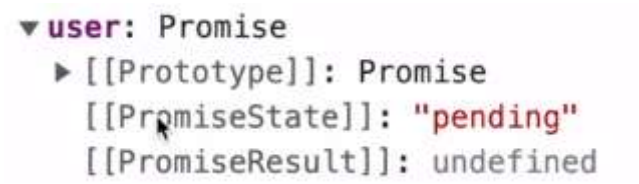
Now we will apply a debugger in browser



We see that initially the user is undefined when line 15 has not been executed.
But a little later we see below,after line 15 has been executed.



And if we expand it we see



But now, we see promise object inside user is in a pending state.
There is a state of a promise and there is a result of promise.
Result will store data. And state tells the state like pending or fulfilled.

```
const GITHUB_API = "https://api.github.com/users

const user = fetch(GITHUB_API);

console.log(user);
```

Now lets log the promise user.
What should we expect to see pending or fulfilled?
OUTPUT

```
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
  ▶ [[PromiseResult]]: Response
```

Above it shows pending but on expanding, below it shows fulfilled.
This inconsistency is part of chrome browser.
Why does this happen?
JS V8 engine quickly logs user in console . At that time user is in pending state due to fetch being an asynchronous operation.
Eventually,after sometime, data comes into the promise object user and hence PromiseState is shown as fulfilled.

Now let's attach a callback function to the promise object.

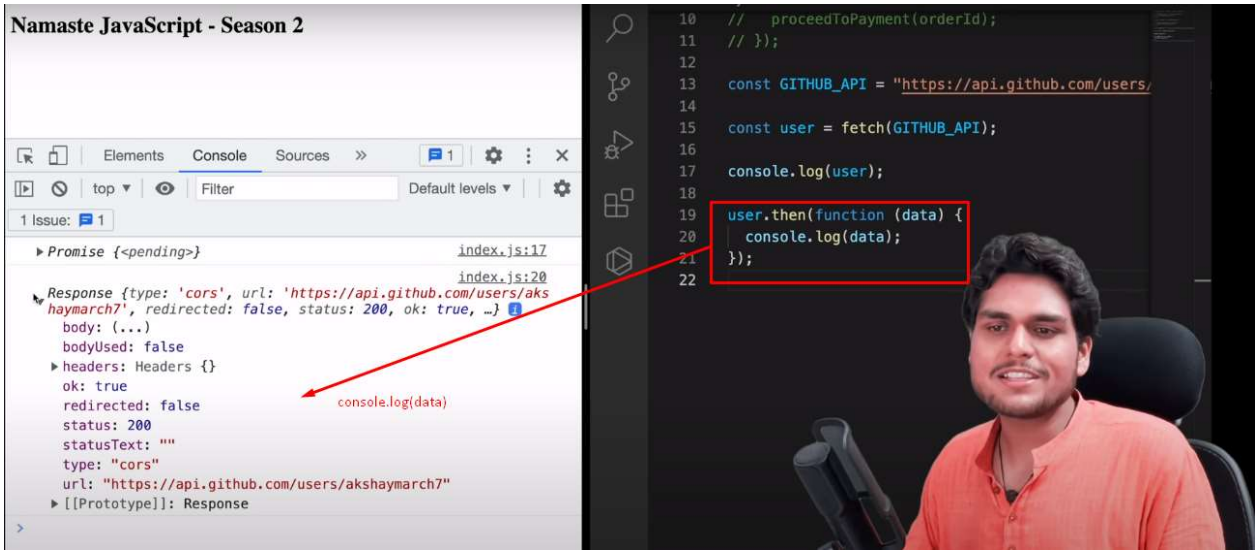
```
user.then(function(data) {
    console.log(data);
})
```

The data passed as parameter is the data we get in PromiseResult.

```
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
  ▼ [[PromiseResult]]: Response
    ▼ body: ReadableStream
        locked: false
      ▶ [[Prototype]]: ReadableStream
      bodyUsed: false
    ▶ headers: Headers {}
      ok: true
      redirected: false
      status: 200
      statusText: ""
      type: "cors"
      url: "https://api.github.com/users/akshaymarch7"
    ▶ [[Prototype]]: Response
```

DATA

And finally the output



There can be 3 states of promise-
Pending , fulfilled, rejected

A `Promise` is in one of these states:

- *pending*: initial state, neither fulfilled nor rejected.
- *fulfilled*: meaning that the operation was completed successfully.
- *rejected*: meaning that the operation failed.

Promise object is immutable.
So we can pass it anywhere without fearing that it may be changed.

For interviews,
What is Promise?
-Placeholder that will be filled later with a value.
-Placeholder for a certain period of time until we receive a value from a asynchronous operation
-Container for a future value

Using Promises

« Previous Next »

A `Promise` is an object representing the eventual completion or failure of an asynchronous operation. Since most people are consumers of already-created promises, this guide will explain consumption of returned promises before explaining how to create them.

For the last part of video we will understand Promise chaining.

Earlier we did this

```
createOrder(cart, function (orderId) {

  proceedToPayment(orderId, function (paymentInfp) {

    showOrderSummary(paymentInfp, function (){

        updateWalletBalance();

    });

  });

});
```

This code grows horizontally, becomes ugly and hard to maintain.
Promise chaining helps here.

This is 1 way to do below

```
const promise = createOrder(cart);

promise.then(function (orderId) {
  proceedToPayment(orderId);
});
```

But we can also do-

```
createOrder(cart).then(function (orderId) {
  proceedToPayment(orderId);
});
```

So now the entire code

But is the chaining right ?
NOO
Why?
Because we are not passing the data down the chain, so we need to use return to pass data down the chain and most developers forget to use it.

```
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    return showOrderSummary(paymentInfo);
  })
  .then(function (paymentInfo) {
    return updateWalletBalance(paymentInfo);
  });
```

Now code does not grow horizontally.
We can also use arrow function inside then to increase readability..

```
createOrder(cart)
  .then((orderId) => proceedToPayment(orderId))
  .then((paymentInfo) => showOrderSummary(paymentInfo))
  .then((paymentInfo) => updateWalletBalance(paymentInfo));
```

This gets us out of callback hell and generates trust in the transaction.
We saw how we consume a promise, which will be executed only once.

RECAP-

1. Before promise we used to depend on callback functions which would result in 1.) Callback Hell (Pyramid of doom) | 2.) Inversion of control
2. Inversion of control is overcome by using promise.
2.1) A promise is an object that represents eventual completion/failure of an asynchronous operation.
2.2) A promise has 3 states: pending | fulfilled | rejected.
2.3) As soon as promise is fulfilled/rejected => It updates the empty object which is assigned undefined in pending state.
2.4) A promise resolves only once and it is immutable.
2.5) Using .then() we can control when we call the cb(callback) function.

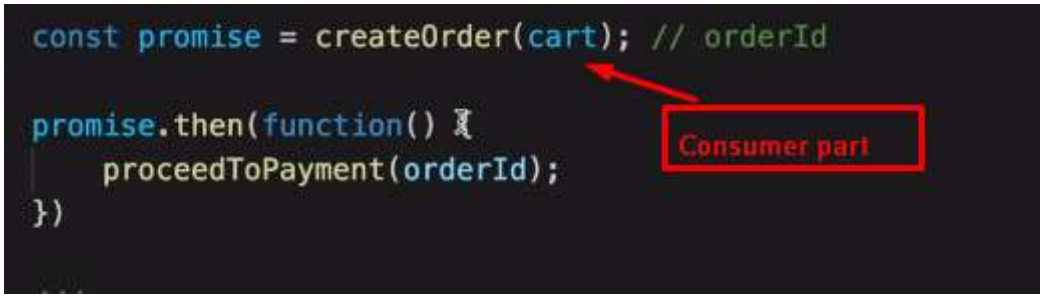
3. To avoid callback hell (Pyramid of doom) => We use promise chaining. This way our code expands vertically instead of horizontally. Chaining is done using '.then()'
4. A very common mistake that developers do is not returning a value during chaining of promises. Always remember to return a value. This returned value will be used by the next .then()

Creating a Promise, Chaining & Error Handling | Ep 03 Season 02 Namaste JavaScript

Youtube Link-https://youtu.be/U74BJcr8NeQ

We will learn the following-

We had done consuming the promise in previous part-

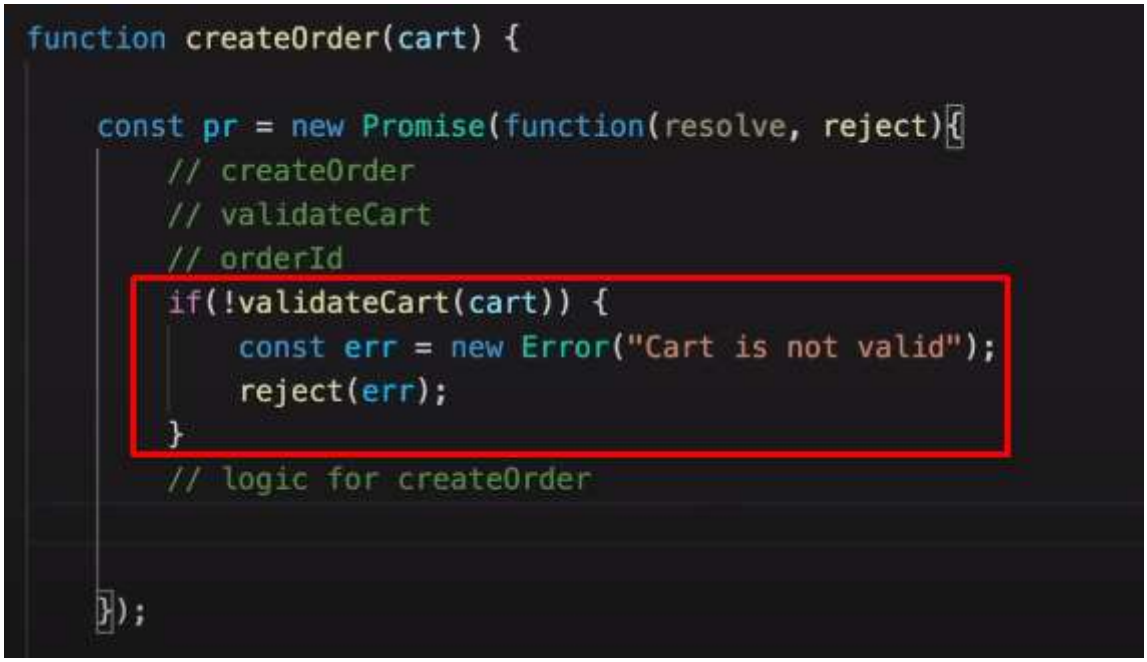


Now we want to write the producer part which is the code for createOrder that will return us a promise.
We use new keyword to create Promise.

Promise constructor will take function that has 2 parameters resolve and reject.

```
function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){

    });


    return pr;

}
```

By design of promise api, we get resolve and reject functions given by javascript.

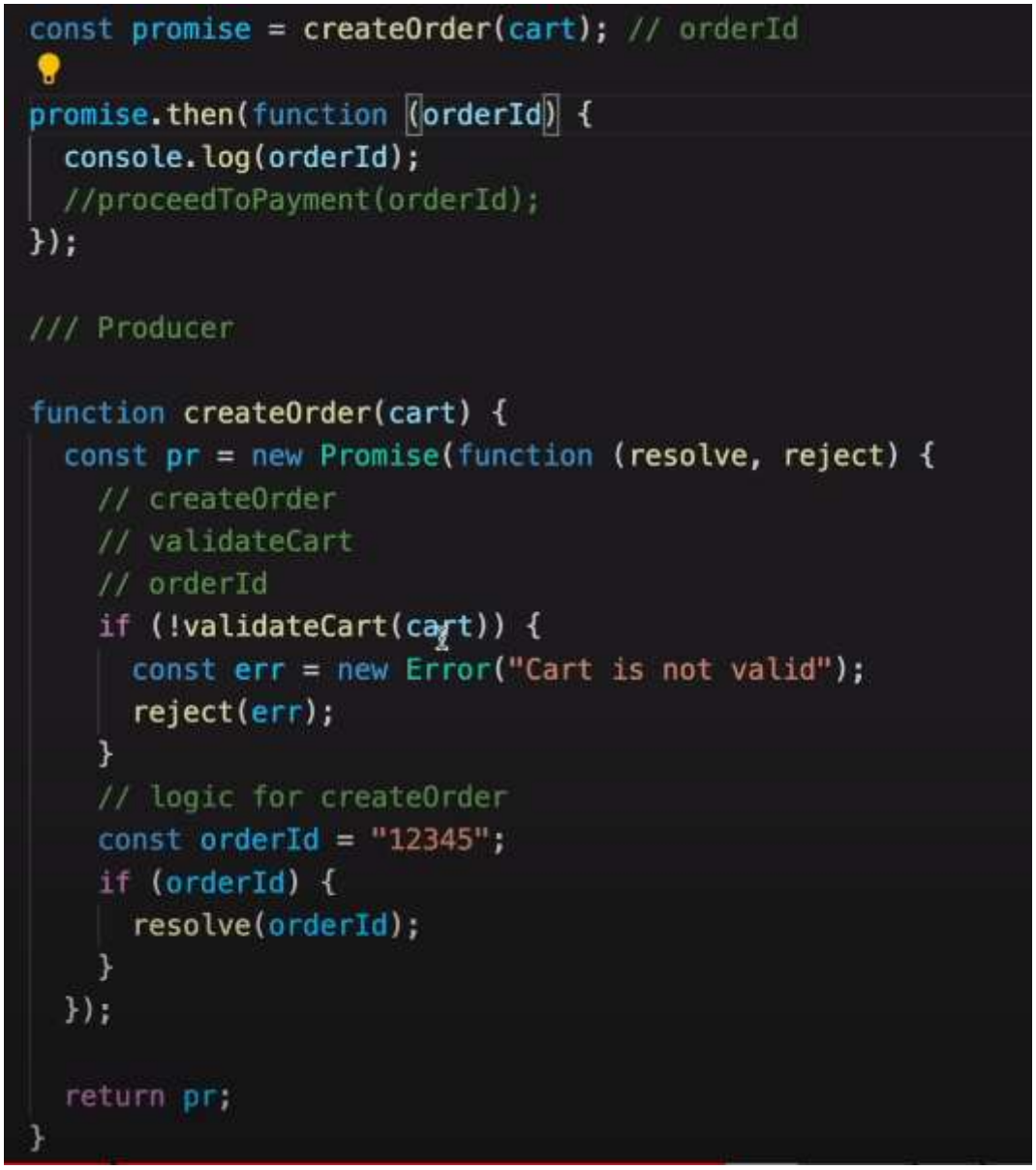


```
function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){
        // createOrder
        // validateCart
        // orderId
        if(!validateCart(cart)) {
            const err = new Error("Cart is not valid");
            reject(err);
        }
        // logic for createOrder


    });
```

We simply reject Promise if cart is not validated. To reject we use reject function provided by promise.But before that we create an error using new Error and then pass it to reject function.

```
function validateCart(cart) {
    return true;
}
```

For simplicity, let the validateCart() return true.



```
const promise = createOrder(cart); // orderId

promise.then(function (orderId) {
  console.log(orderId);
  //proceedToPayment(orderId);
});

/// Producer

function createOrder(cart) {
  const pr = new Promise(function (resolve, reject) {
    // createOrder
    // validateCart
    // orderId
    if (!validateCart(cart)) {
      const err = new Error("Cart is not valid");
      reject(err);
    }
    // logic for createOrder
    const orderId = "12345";
    if (orderId) {
      resolve(orderId);
    }
  });

  return pr;
}
```

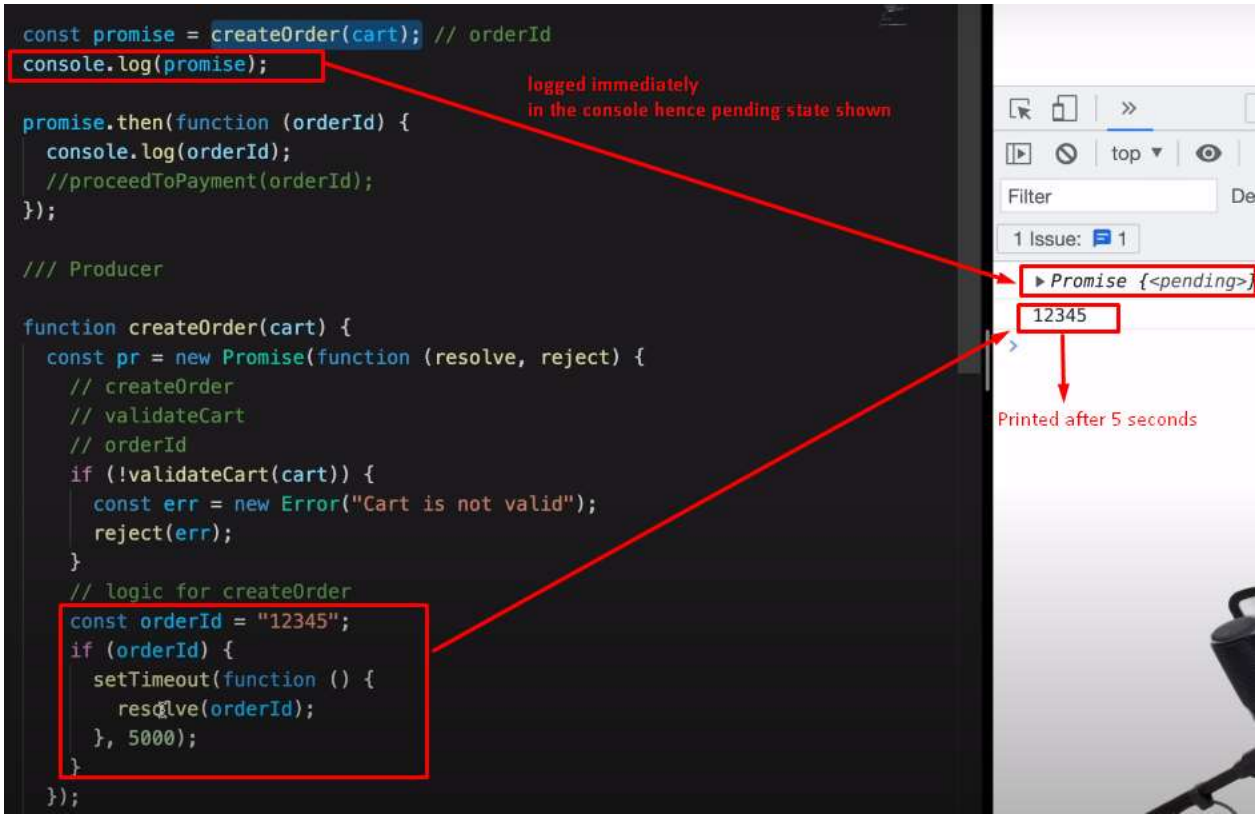
Explanation for above
-createOrder() returns a promise.
-if promise is resolved then we print orderID.
-Promise is resolved if orderId is true as seen in the code.
-validateCart() returns true so promise is not rejected.
-simply using resolve and passing orderId as parameter works
-the orderId passed below is accepted by the callback function used above using then
-Promise is resolved or rejected as per condition but value we pass an argument inside resolve or reject goes as parameter to the callback function inside then.

In console we get orderID

```
12345
```

```
const orderId = "12345";
if (orderId) {
  setTimeout(function () {
    resolve(orderId);
  }, 5000);
}
```

If we add a setTimeout ,then orderId will printed on console after 5 seconds.



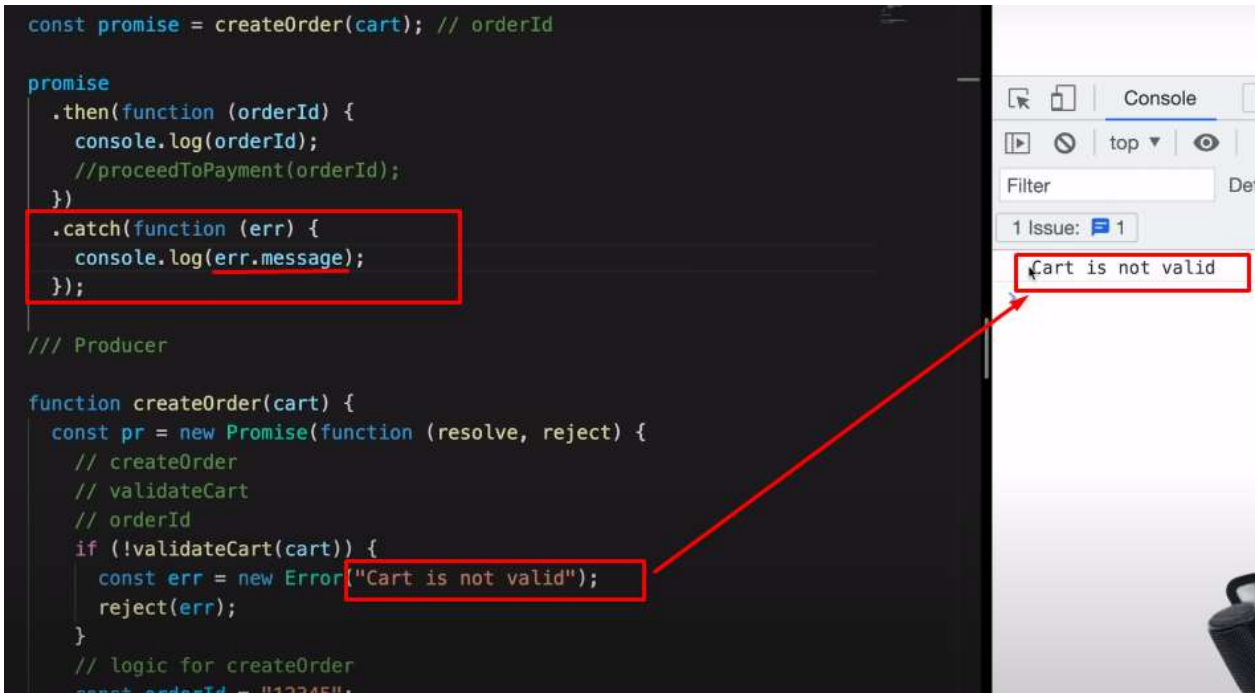
This is how we create a promise and consume it.

NOW LETS REJECT OUR PROMISE.
Just make validateCart() return false.

```
function validateCart(cart) {
  return false;
}
```

```
▶Uncaught (in promise)  index.js:18
Error: Cart is not valid
    at index.js:18:19
    at new Promise (<anonymous>)
    at createOrder (index.js:13:14)
    at index.js:3:17
```

In browser we see red coloured error.
This is because we did not handle the error.
And if such error happens, user will never get to know about the error as this will be silently be there in the console.



We handled the promise using catch and we get the message in console.

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
  })
  .then(function() {
    proceedToPayment(orderId);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

Lets chain things up and lets write code for proceedToPayment

So we write the code for returning promise from proceedToPayment

```
function proceedToPayment(orderId) {
    ///
    return new Promise( function(resolve, reject) {
        resolve("Payment Successful");
    })
}
```

But there is again an error.
Whatever value we need to pass down the chain we need to return it.

So we return on each step.
Now, after proceedToPayment we attached another then with function(paymentInfo) as callback function to which we pass the msg "Payment Successful" passed as parameter inside resolve() of proceedToPayment.

We could have attached then immediately after proceedToPayment(orderId) as in red box shown below.

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function(orderId) {
    return proceedToPayment(orderId).then(function(paymentInfo){
        console.log(paymentInfo)
    });
  })

  .catch(function (err) {
    console.log(err.message);
  });
```

But this would become promise hell just like callback hell.
But promise api was designed in such a way that we handle it in next level of chain-

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

So if promise is rejected in any of the steps we go to catch immediately for any of the promise in promise chain.
But what if we need to go to payment or to bottom of chain even if cart validation or anything above of chain fails instead of directly going to the catch at the bottom of every promise chain?

So what we can do is attach a catch with any then. The attached catch will only check for promise rejections above it only.

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .catch(function (err) {
    console.log(err.message);
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  });
```

```
Cart is not valid
Payment Successful
>
```

Cart was not valid still payment was successful as catch above payment handled the rejection of cart validation. Then flow reached payment normally after going through the catch.
If we write a catch in between,then after it will be called.

SUMMARY-

1. Promise can be created using a new Promise() constructor function.
2. This constructor function takes a callback function as argument.
3. The callback function has 2 arguments named 'resolve' and 'reject'. Resolve and reject are the keywords provided by JS.
4. We can only resolve or reject a promise. Nothing else can be done.
5. An error can also be created using new Error('error message').
6. There is also .catch() which is used to attach a failure callback function that handles any error that pops up during the execution of promise chain.
7. .catch only handles error of .then() that are present above it. If there is any .then() below it, catch will not handle any error for that, also that ,then will get executed no matter what.
8. It can be useful in a way if we want to catch error for a particular portion of a chain.
9. We can have multiple catch based on requirement and then a general catch at the end.
10. Always remember to return a value in the promise chain for the next .then to use .

11. If it returns a value => It will be used as a parameter in next function. If it is a promise then the next .then in the promise chain is attached to the promise returned by the current callback function.

Homework qs-
Create a chain for these APIs

```
createOrder,
proceedToPayment,
showOrderSummary,
updateWallet
```

Homework answer-

```
const cart = ['shoes', 'pants', 'kurta'];

createOrder(cart)
 .then(function(orderId) {
   console.log(orderId);
   return orderId;
 })
 .then(function(orderID) {
   return proceedToPayment(orderID)
 })
 .then(function({ message, amt }) {
   console.log(message, 'of amount:', amt);
   return showOrderSummary(message, amt);
 })
 .then(function({ message, amt }) {
   updateWallet({message,amt});
 })
 .catch(function(err) {
   console.log(err.message);
 })
```

```
 .then(function() {
   console.log('No matter what happens, I will get executed');
 });
function createOrder(cart) {
 const pr = new Promise(function(resolve, reject) {
   // create order
   // Validate Cart
   // orderId
   if (!validateCart(cart)) {
     const err = new Error('Cart is not valid!');
     reject(err);
   }
   // logic for createOrder
   const orderId = '12345';
   if (orderId) {
     setTimeout(function() {
       resolve(orderId);
     }, 5000)
   }
 });

 return pr;
}

function proceedToPayment(orderID) {
 // Logic for handling payment.
 // This function returns a promise
 return new Promise(function(resolve, reject) {
   // logic
```

```
    resolve({ message: `Payment Successful for order id: ${orderID}`,
amt: 2500 });
  })
}

function showOrderSummary(paymentInfo, amt) {
  return new Promise(function(resolve, reject) {
    // console.log(amt);
    if (amt >= 2000) {
      resolve({ message: 'You have ordered items that cost ${amt} RS',
amt });
    } else {
      reject(new Error('Please buy more for discount'));
    }
  })
}
function updateWallet({ message, amt }){
    console.log('Your wallet has beed debited by:', amt);
}
function validateCart(cart) {
  // code to validate cart.
  return true;
  // return false;
}
```