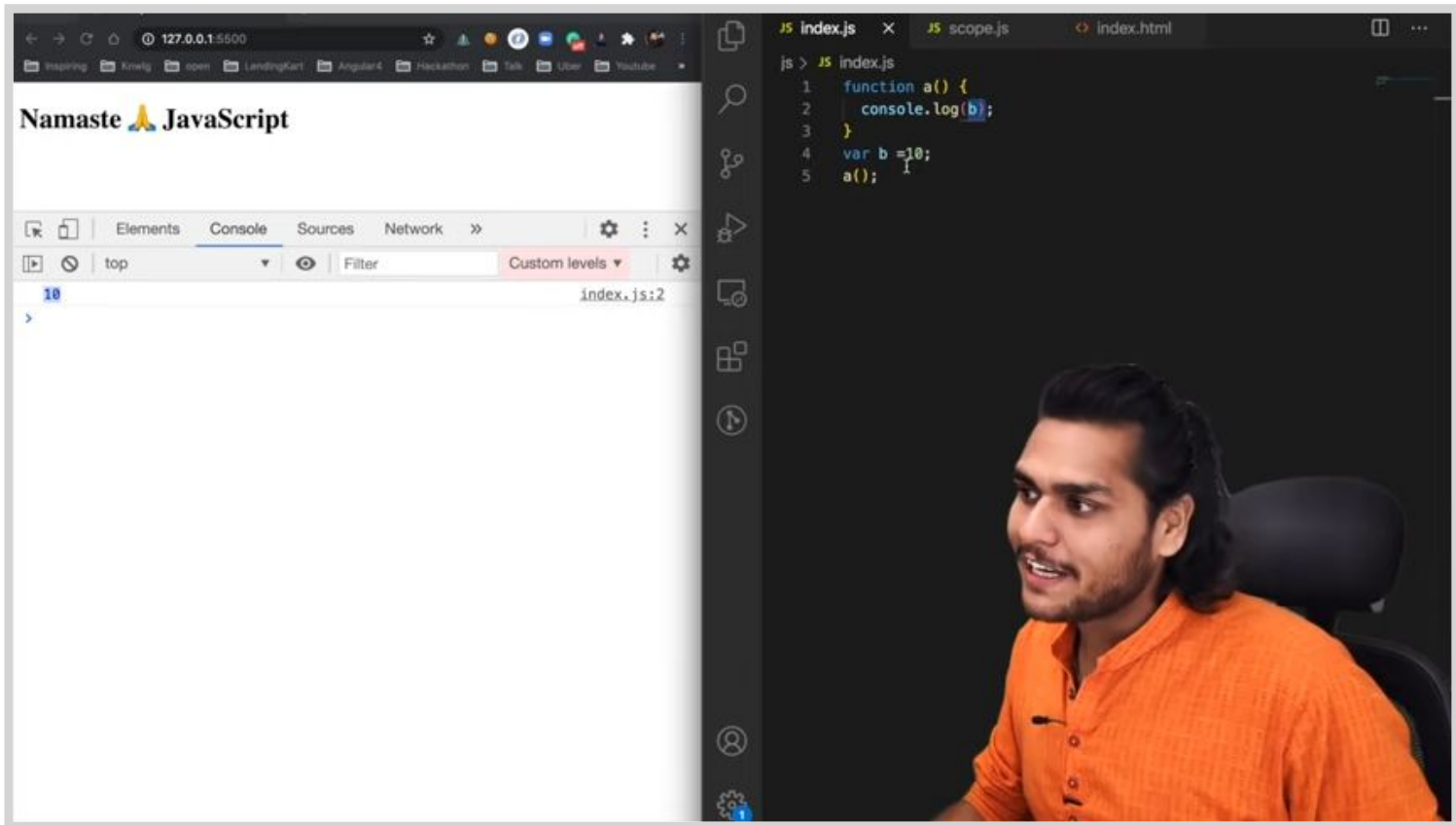




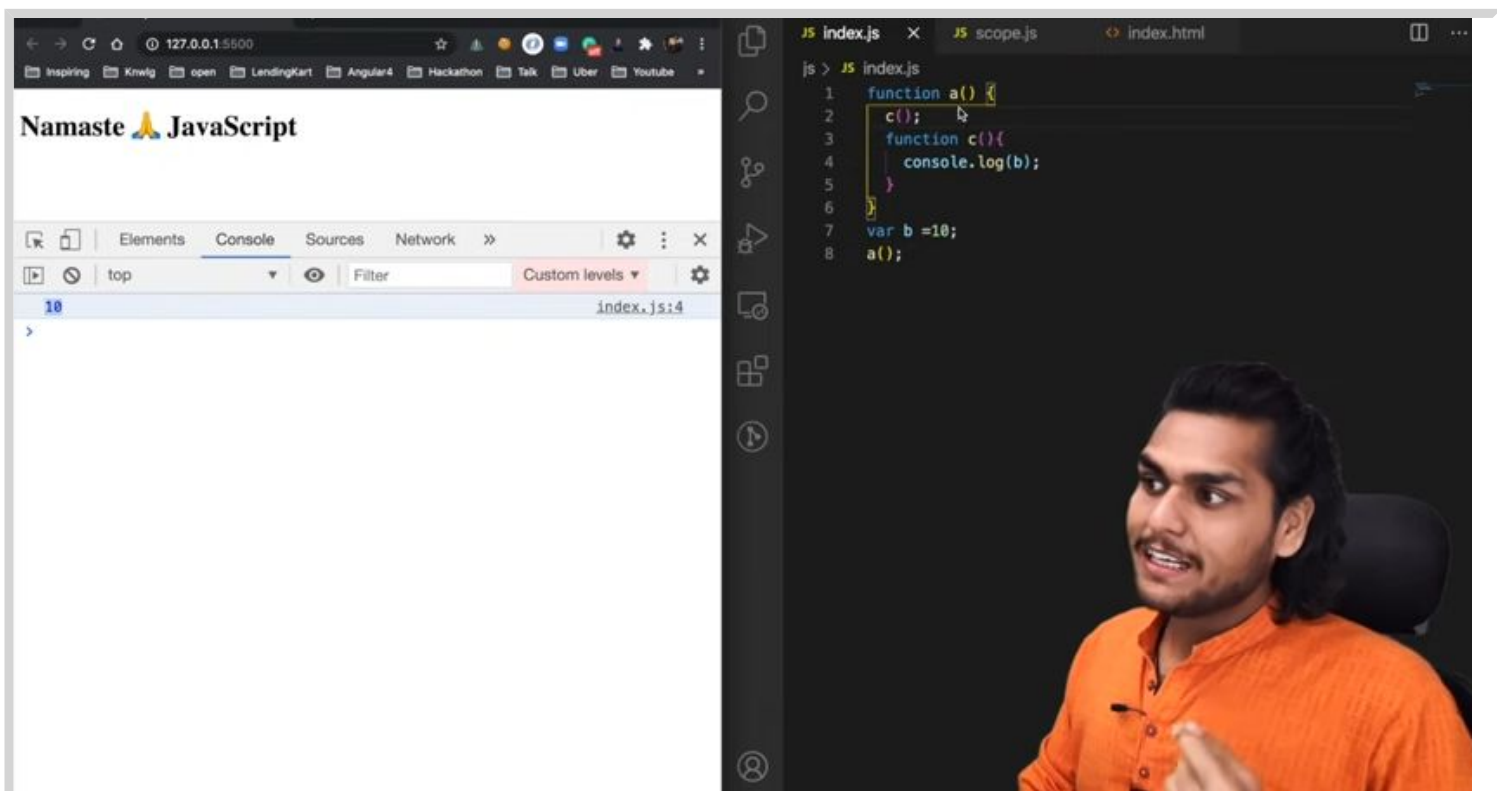
Make Notes from Youtube Videos

Scope in JS is directly dependent on Lexical environment.

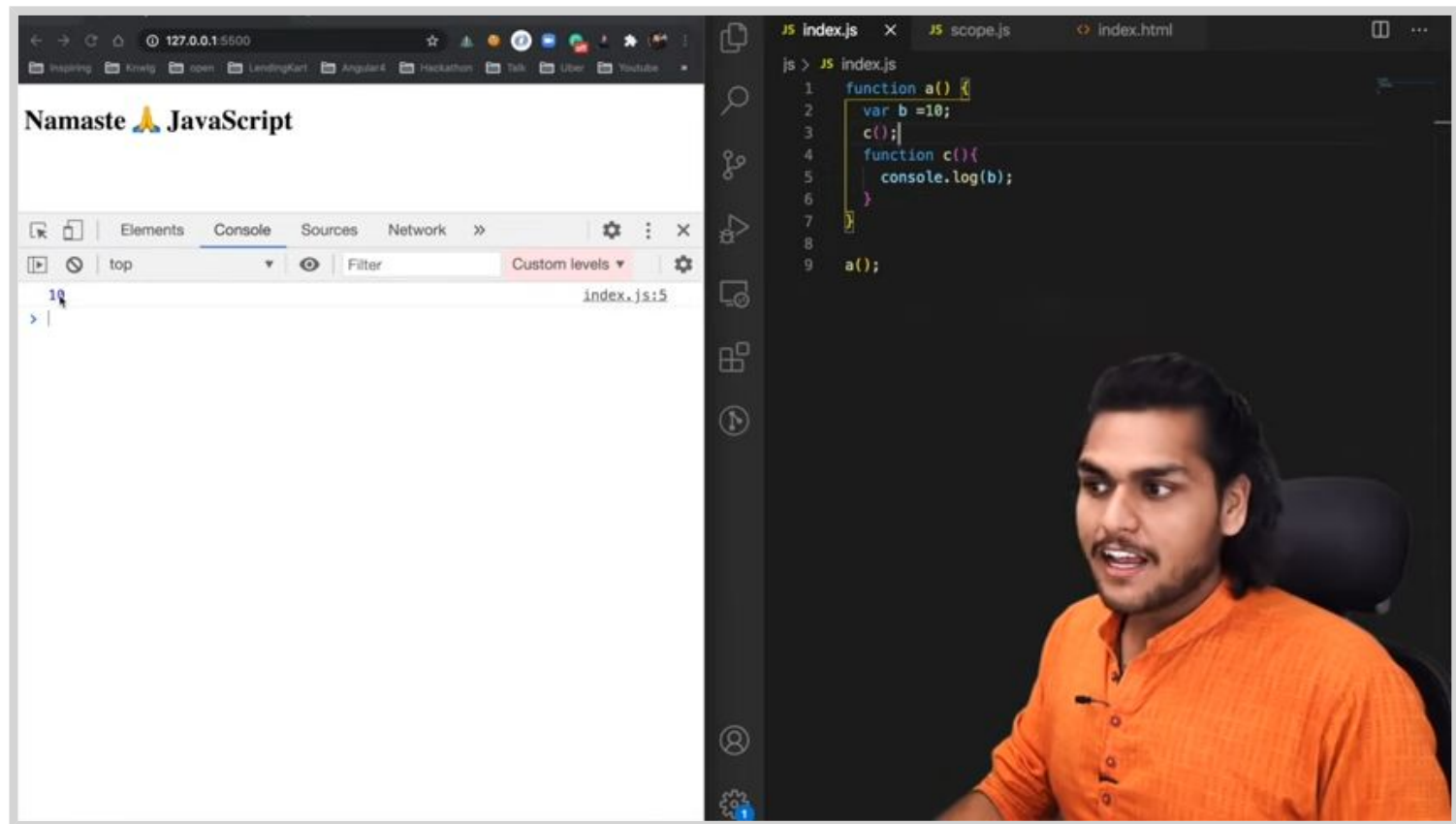
02:04



02:49



03:11

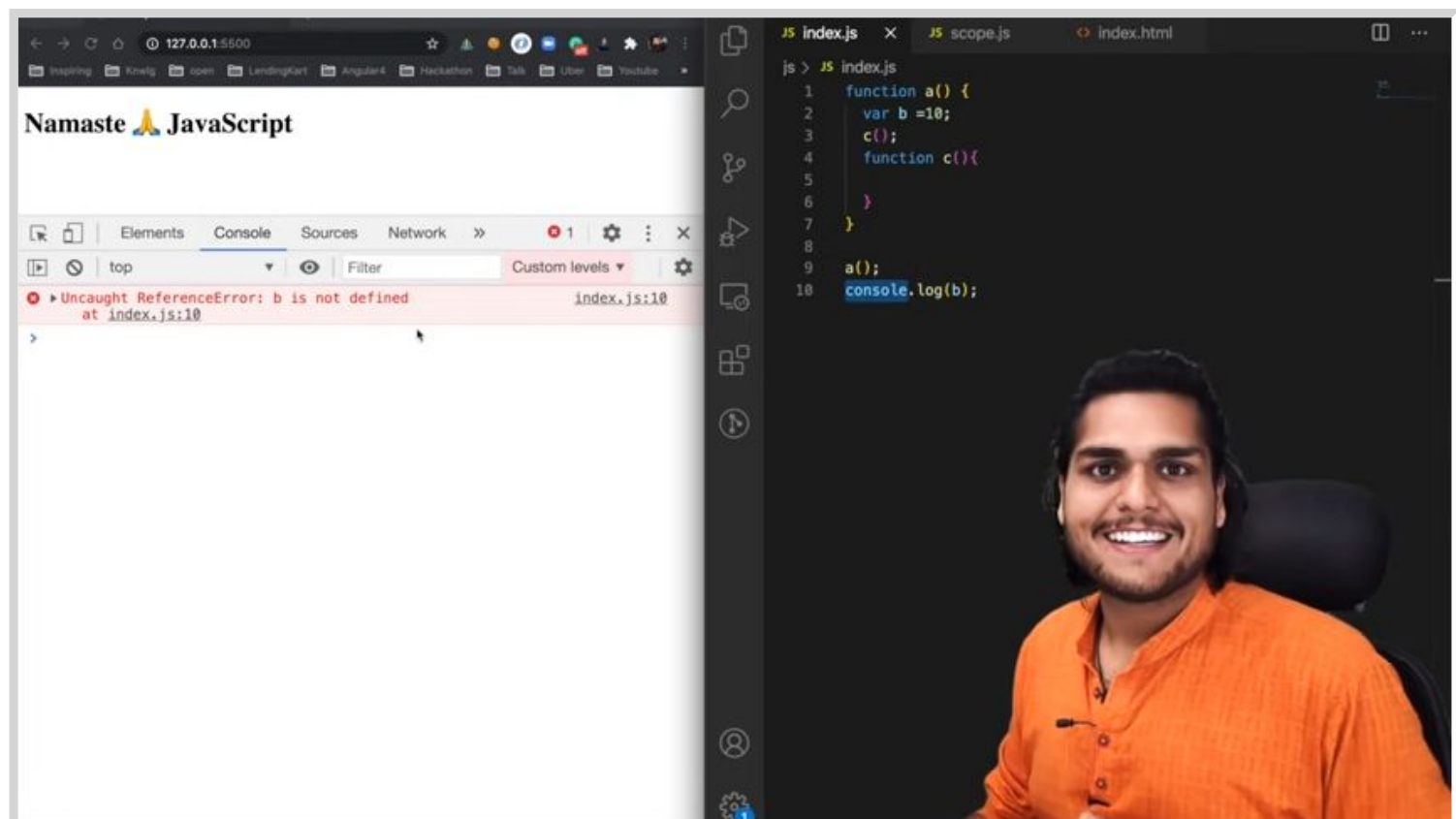


The screenshot shows a web browser window on the left and a code editor on the right. The browser displays "Namaste 🙏 JavaScript" and the console shows a successful execution of a script. The code editor shows the following JavaScript code in `index.js`:

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

The code defines a function `a` that declares a variable `b` with the value 10 and calls a function `c`. Function `c` logs the value of `b` to the console. The function `a` is then called at the end of the script.

03:25



The screenshot shows a web browser window on the left and a code editor on the right. The browser displays "Namaste 🙏 JavaScript" and the console shows an error: "Uncaught ReferenceError: b is not defined at index.js:10". The code editor shows the following JavaScript code in `index.js`:

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

The code is identical to the previous screenshot, but with an additional line at the bottom: `console.log(b);`. This line attempts to log the value of `b` after the function `a` has finished executing, but `b` is not in the global scope, leading to the error.

HERE comes Scope into the picture


05:40

Call Stack

Memory	Code
a: { }	

Global EC

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



06:23


Call Stack

Mem.	Code
b: c: { }	

Memory	Code
a: { }	

Global EC

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



07:09

The diagram illustrates the state of the JavaScript engine's memory at 07:09. On the left, a vertical stack of three frames is shown. The top frame is labeled 'Call Stack c()' and contains two columns: 'Mem' and 'Code'. The middle frame is labeled 'a()' and contains 'Mem.' and 'Code' columns; the 'Mem.' column has 'b:10' and 'c:{-}' listed. The bottom frame is labeled 'Global EC' and contains 'Memory' and 'Code' columns; the 'Memory' column has 'a:{-}' listed. To the right, a code editor shows the following JavaScript code:

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

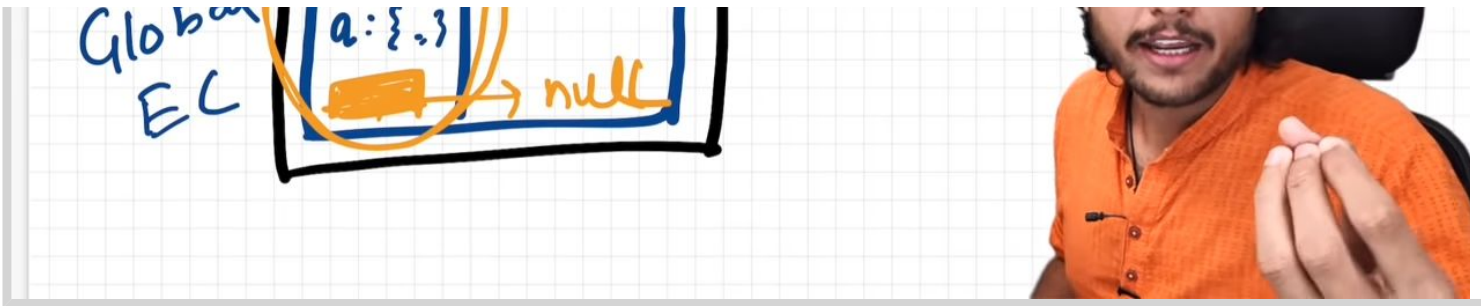
Whenever an Execution context is created , a lexical environment is also created.

- Lexical Environment is the local memory along with the lexical environment of its parent.
- Lexical means Hierarchy or in-order
- here, Function C() is lexically inside A () function & A() is lexically inside the Global Scope

10:45

The diagram illustrates the state of the JavaScript engine's memory at 10:45. It is similar to the previous diagram, but with additional annotations. In the 'a()' frame, the 'Mem.' column contains 'b:10' and 'c:{-}', and the 'Code' column is empty. An orange box highlights the 'c:{-}' entry, and an orange arrow points from it to the 'Code' column of the 'Call Stack c()' frame. The 'Global EC' frame remains the same. The code editor on the right shows the same JavaScript code as before:

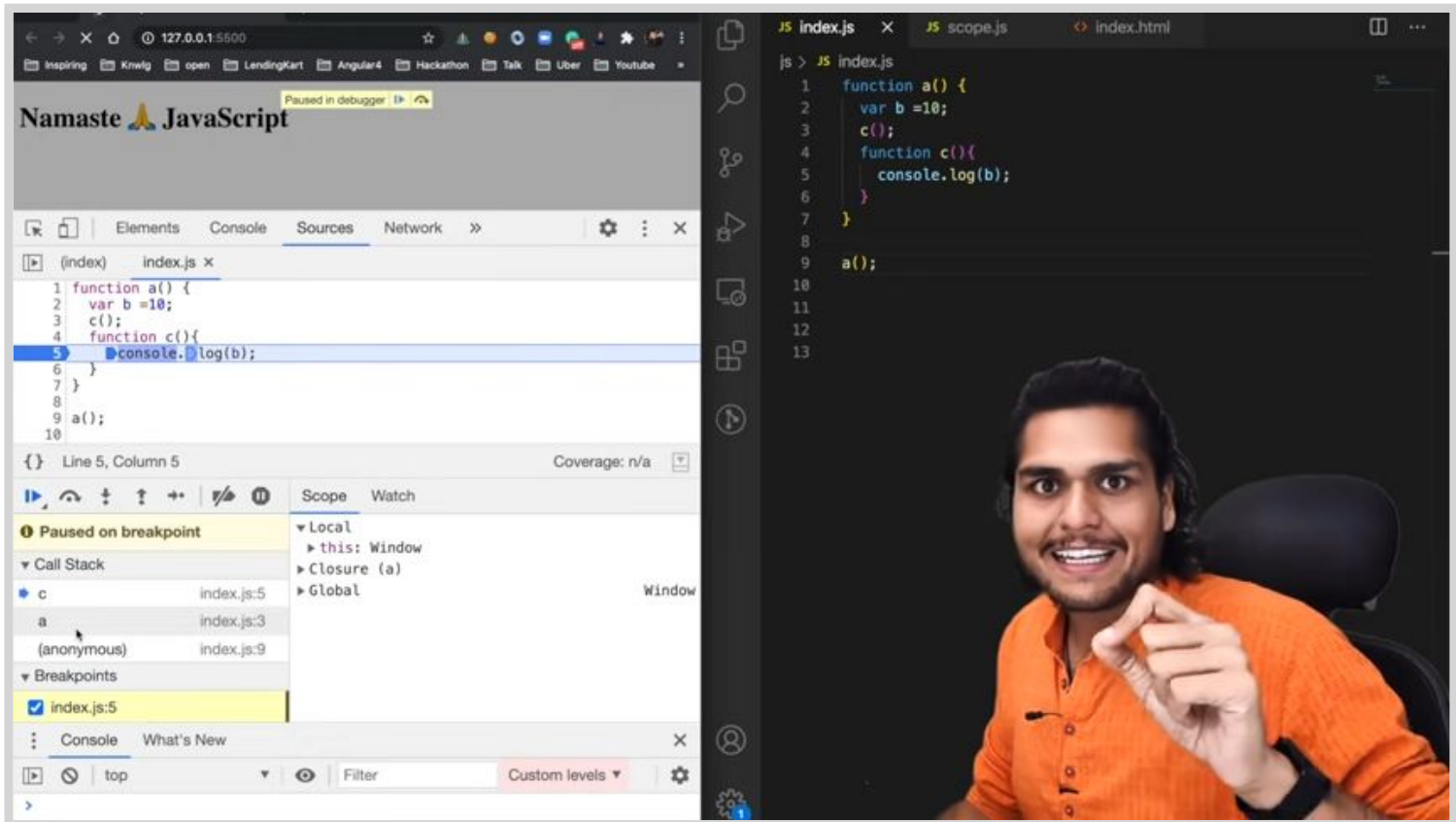
```
1 function a() {  
2   var b =10;  
3   c();  
4   function c(){  
5  
6   }  
7 }  
8  
9 a();  
10 console.log(b);
```



(suppose - var b = 10 is not there) - it will give null

This way of finding ,where we go from 1 lexical environment to parent of that lexical environment. is known as **SCOPECHAIN** .

16:02



"anonymous" - global execution context

