

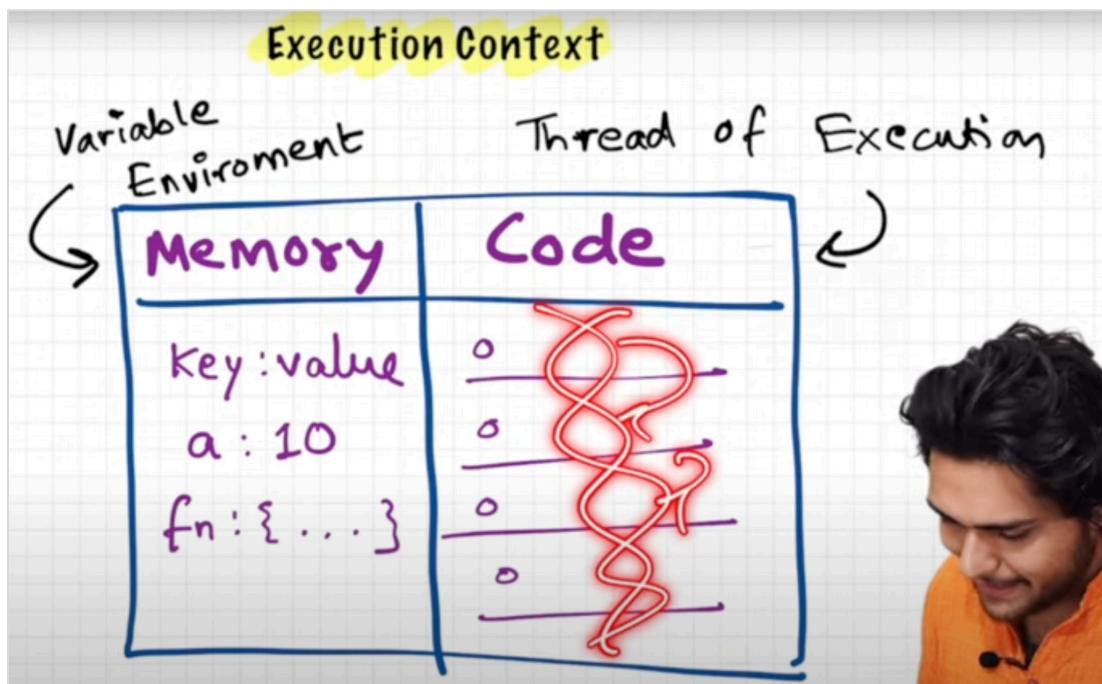
# Namaste JavaScript

By Akshay Saini during Nov 2024

- How JavaScript Works 🔥 & Execution Context
  - Everything in Javascript, happens inside an **Execution Context**



- Execution Context is like big box or container
- Execution Context has 2 major components
  - **Memory**, where all JS variables, functions stored as key, value pair and used to call as **Variable Environment**
  - **Code**, all code executed in code component, executed line by line, one line at a time, calling as **Thread of Execution**



**“JavaScript is a  
synchronous  
single-threaded  
language”**

- JS is **Single Threaded**, since JS executed one command at a time only once with **Synchronously** (orderly from top one by one)
- How JavaScript Code is executed? ❤️ & Call Stack
  - What happens when you run the JS code ? - **Global Execution Context will create with 2 phases**
    - Phase 1 - **Memory creation phase**, will allocate memory for all JS variables & functions
    - Phase 2 - **Code execution phase**, all computation & variable initialization will take care
  - **What happens when run below the piece of JS code ?**

```
var n = 2;

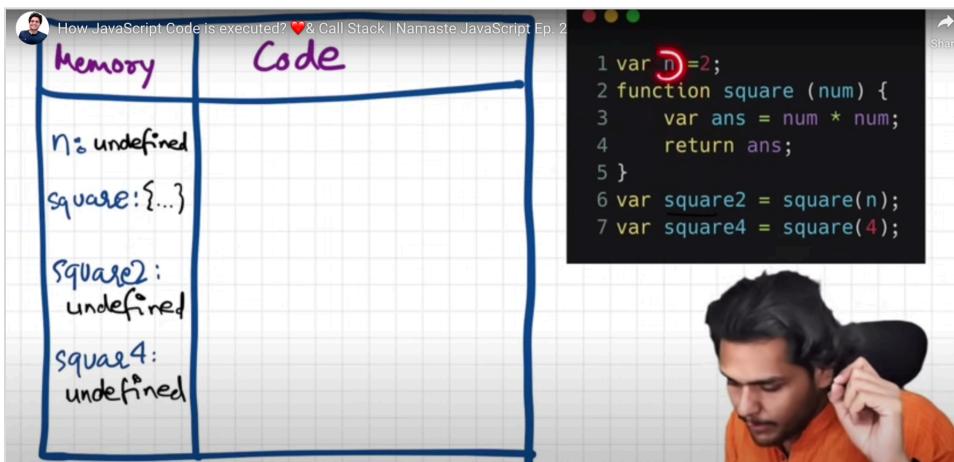
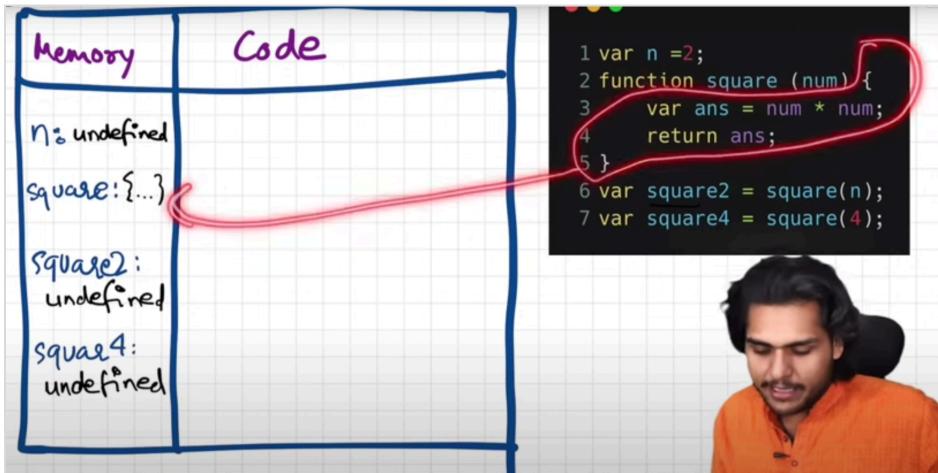
function square(num) {
    var ans = num * num;
    return ans;
}

var square2 = square(n);
var square4 = square(4);

console.log("square2", square2);
console.log("square4", square4);
```

- Initially Global Execution Context will create and start allocating memory for all JS variables & functions
- During Memory creation phase 1
  - Initially for all variables assign with **undefined**, **undefined is special keyword which initially during memory creation phase assign to JS variable**

- For functions, takes value as a whole function



- During Code execution phase 2, actual calculation will take care
  - When control encounters, For variables, start initialize the value in memory by replacing the undefined

The screenshot shows a video player interface. On the left, there is a hand-drawn diagram titled "Memory" and "Code". The "Memory" section contains variables: "n: 2", "square: {...}", "square2: undefined", and "square4: undefined". The "Code" section contains the following JavaScript code:

```

1 var n = 2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);

```

A pink arrow points from the value "2" in the "Memory" section to the variable "n" in the "Code" section. On the right, a video player interface shows a person speaking, with a "Share" button at the top right.

- Line #6, have a function invocation / calling / executed

- Functions are heart of Javascript, act as mini program & behave differently compare to other programming language

- Line #6, When invoke / call the function, brand new Execution Context will create within global and follow the same Memory creation phase & Code execution phase

The screenshot shows a video player interface. On the left, there is a more detailed hand-drawn diagram. The "Memory" section contains "n: 2". The "Code" section contains a sub-diagram with "Memory" and "Code" sections. The "Memory" section of the sub-diagram contains "num: undefined" and "ans: undefined". The "Code" section of the sub-diagram is empty. A red circle highlights the value "2" in the "Memory" section of the main diagram, and a red arrow points from it to the variable "n" in the sub-diagram's "Memory" section. The "Code" section of the main diagram contains the same JavaScript code as before. On the right, a video player interface shows a person speaking, with a "Share" button at the top right.

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

```

1 var n =2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);

```

Share

Memory

Code

n: 2

square:{...}

square2:  
undefined

square4:  
undefined

MORE VIDEOS

ans: 4

return ans

square4

- As soon as function completed its execution, its execution context will be removed / deleted & give the control to from where the function is called

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

```

1 var n =2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);

```

Share

Memory

Code

n: 2

square:{...}

square2: 4

square4:  
undefined

MORE VIDEOS

ans: 4

return ans

square4

- Similarly in line#7 also will executed with brand new execution context & follow the same process

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

The screenshot shows a video interface. On the left, there's a small profile picture of a person. The main area has a title "How JavaScript Code is executed? ❤️ & Call Stack | Namaste JavaScript Ep. 2". Below the title, there's a hand-drawn diagram on lined paper. The diagram consists of two boxes: "Memory" and "Code". In the "Memory" box, there are two rows: "num: 2" and "ans: 4". In the "Code" box, the word "return" is written. To the left of the diagram, there are some handwritten notes: "n: 2", "var: {...}", "var2: 4", and "var4: undefined". On the right side of the video interface, there's a code editor window showing the following JavaScript code:

```

1 var n = 2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);

```

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

This screenshot is similar to the one above, showing the same hand-drawn memory diagram and code editor. The handwritten notes on the left now include: "n: 2", "square: {...}", "square2: 4", and "square4: 16". The "Code" box from the first diagram has been crossed out with a large red X. Instead, there is a new "Code" box below it, which also contains a red X. The code editor on the right remains the same.

- When whole JS code done with work / executed completely, global execution context will be removed

```

1 var n = 2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);

```

A man with dark hair and an orange shirt is visible on the right side of the screen.

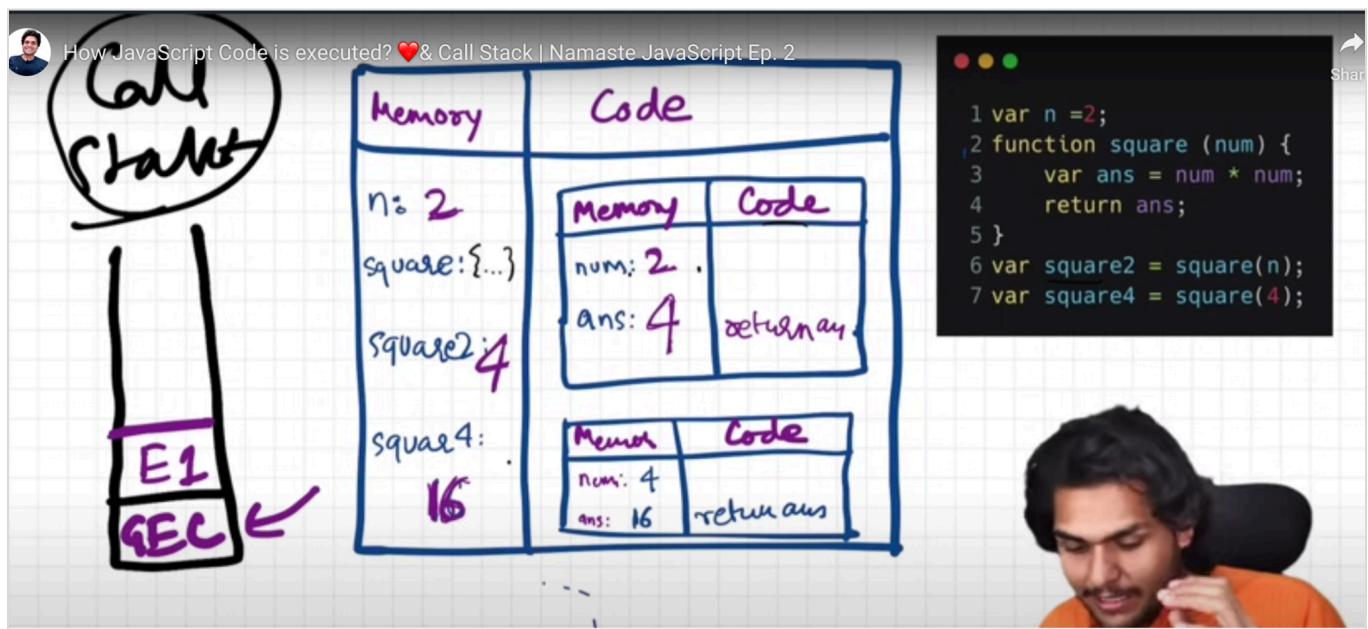
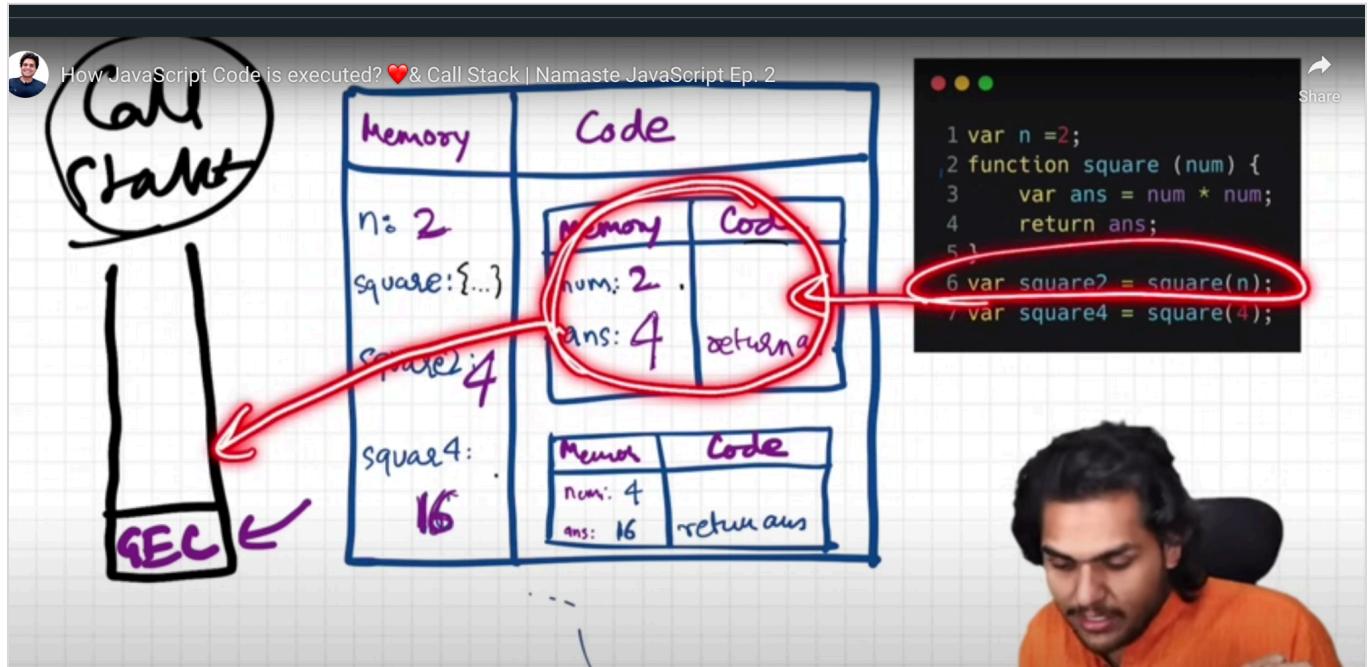
- How does JS manage creation and deletion of Execution context when ever JS functions called ?

```

1 var n = 2;
2 function square (num) {
3     var ans = num * num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4);

```

- Always Global Execution Context will be there at bottom of Call Stack
- Whenever new function is called, branch new Execution Context will pushed into Call Stack



- As soon as done with E1 pop from stack, control back to where it called in global execution context
  - Similarly will execute with line #7 as well
- As complete JS code execution done, Global EC also will removed & Call Stack becomes empty

“Call stack maintains the  
order of execution of  
execution contexts”

- Call Stack calling with different names

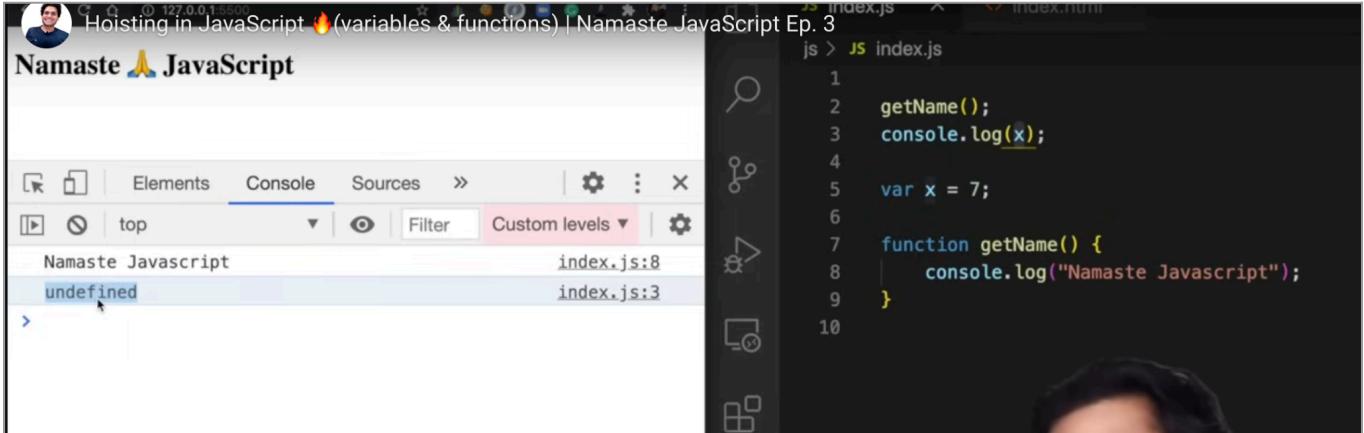
## 0. Call Stack

1. Execution Context Stack
2. Program Stack
3. Control Stack
4. Runtime Stack
5. Machine Stack



- Hoisting in JavaScript 🔥 (variables & functions)
  - Answer in Memory creation phase & code execution phase

- Magic of JS hoisting

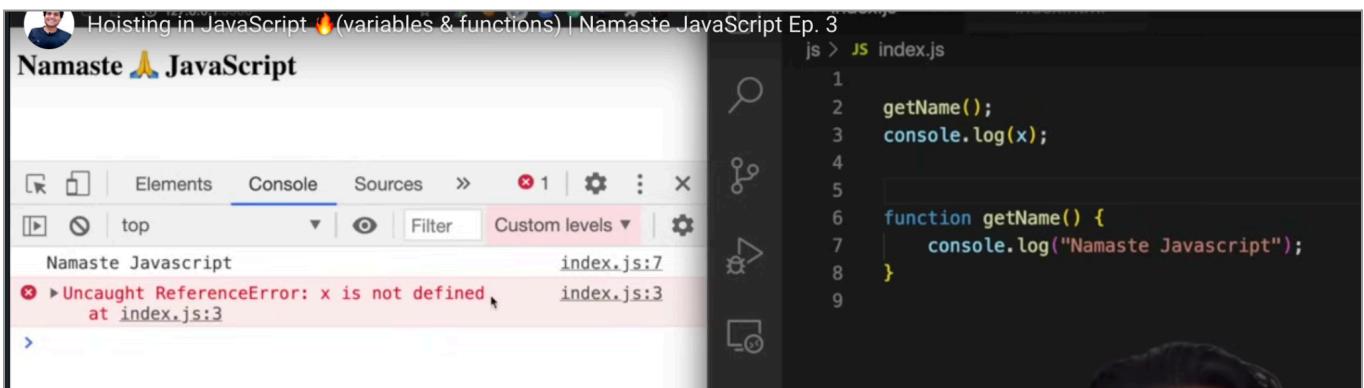


The screenshot shows a browser developer tools interface with the "Console" tab selected. The output pane displays:

```
Namaste 🙏 JavaScript
index.js:8
undefined
index.js:3
```

The source code pane shows:

```
js > JS index.js
1
2 getName();
3 console.log(x);
4
5 var x = 7;
6
7 function getName() {
8   console.log("Namaste Javascript");
9 }
10
```

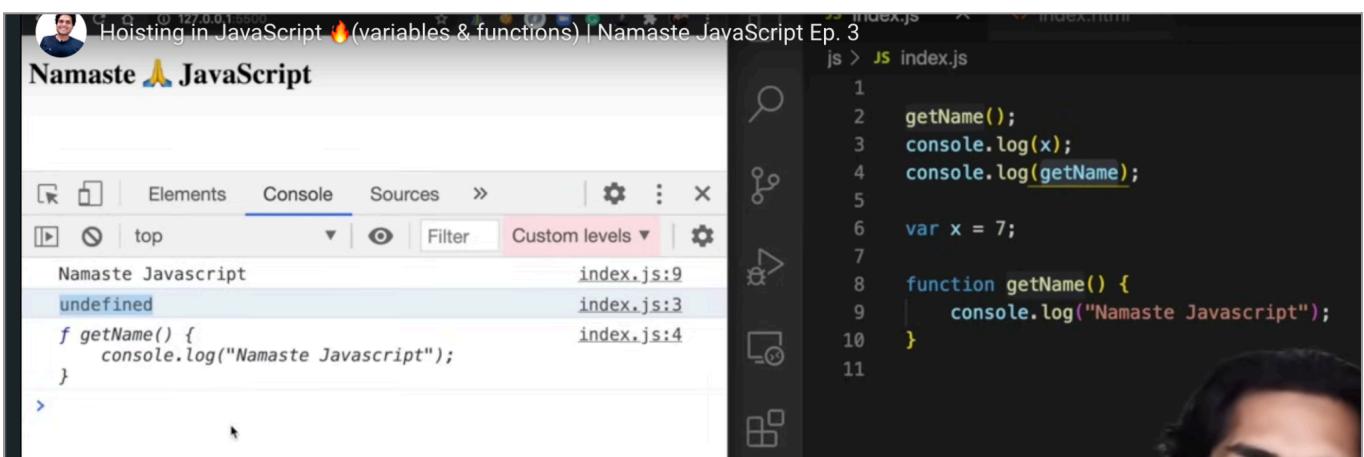


The screenshot shows a browser developer tools interface with the "Console" tab selected. The output pane displays an error message:

```
Uncaught ReferenceError: x is not defined
at index.js:3
```

The source code pane shows:

```
js > JS index.js
1
2 getName();
3 console.log(x);
4
5
6 function getName() {
7   console.log("Namaste Javascript");
8 }
9
```



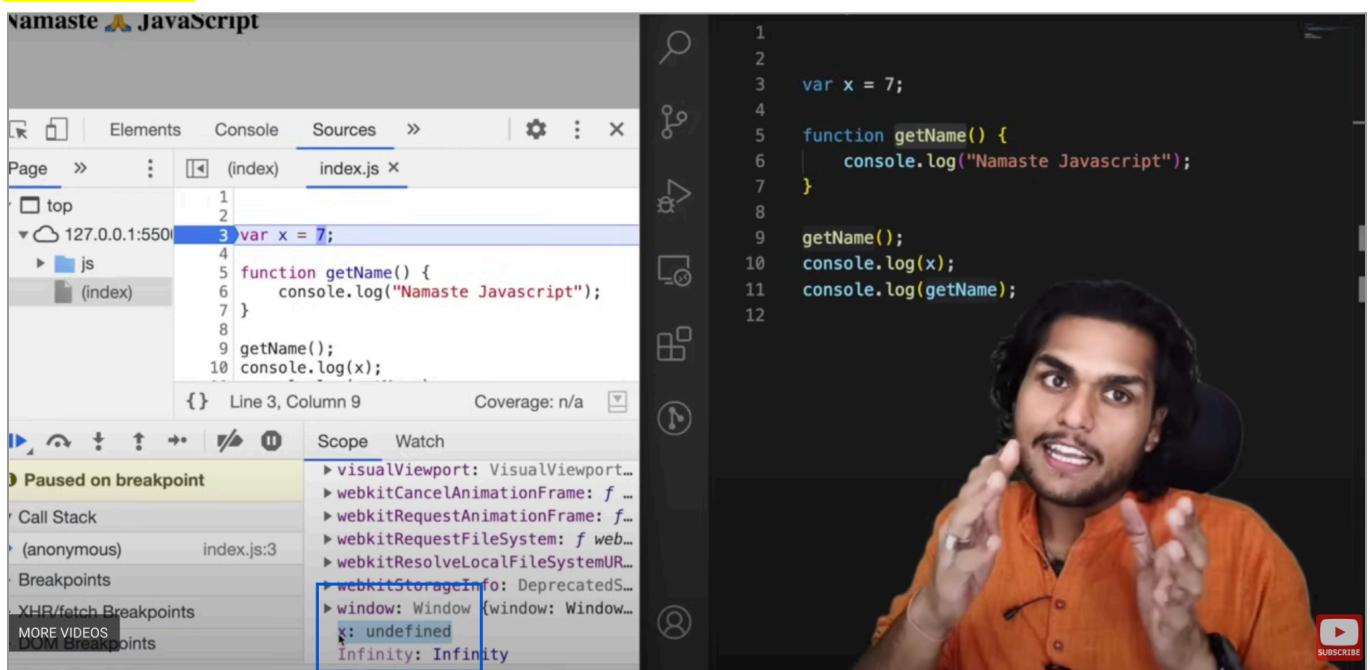
The screenshot shows a browser developer tools interface with the "Console" tab selected. The output pane displays:

```
Namaste 🙏 JavaScript
index.js:9
undefined
index.js:3
f getName() {
  console.log("Namaste Javascript");
}
```

The source code pane shows:

```
js > JS index.js
1
2 getName();
3 console.log(x);
4 console.log(getName());
5
6 var x = 7;
7
8 function getName() {
9   console.log("Namaste Javascript");
10 }
11
```

- Is it **undefined & not defined** both are same ? And: NO  
Both are different
- **JS Hoisting is behaviour where access all JS variables & function even before initialized it, without any error**
- Even before code starts executing, memory is allocating to all JS variables & function during memory creation phase



The screenshot shows a browser developer tools debugger interface. The title bar says "Hoisting in JavaScript (variables & functions) | Namaste JavaScript Ep. 3". The "Sources" tab is selected, showing the file "index.js". The code is:

```
1
2
3 getName();
4 console.log(x);
5 console.log(getName());
6
7 var x = 7;
8
9 function getName() {
10   console.log("Namaste Javascript");
}
```

The line 3, "getName();", has a yellow highlight. A tooltip above it says "undefined". In the bottom left, the "Call Stack" panel shows "(anonymous)" at index.js:3. In the bottom right, the "Scope" and "Watch" panels are visible. The "Scope" panel shows "Global" with "PERSISTENT: 1" and "TEMPORARY: 0". The "Watch" panel is empty. On the right side of the screen, there is a video player showing a man with long hair and a beard, wearing an orange shirt, pointing towards the camera.

- **Not defined**

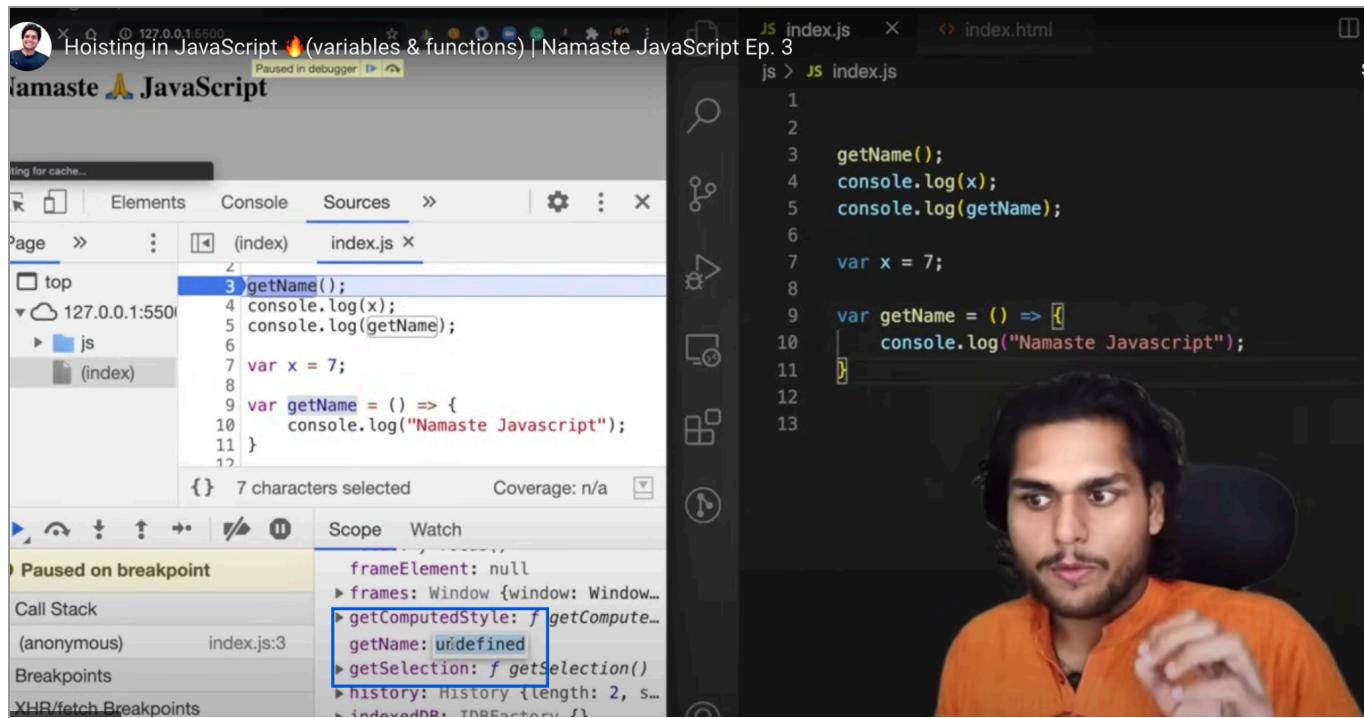
When memory is not allocated for variable x, since not defined, will get error

The screenshot shows a browser developer tools debugger interface. The title bar says "Hoisting in JavaScript (variables & functions) | Namaste JavaScript Ep. 3". The "Sources" tab is selected, showing the file "index.js". The code is identical to the previous screenshot:

```
1
2
3 getName();
4 console.log(x);
5 console.log(getName());
6
7 var x = 7;
8
9 function getName() {
10   console.log("Namaste Javascript");
}
```

The line 4, "console.log(x);", has a red highlight. A tooltip above it says "Uncaught ReferenceError: x is not defined". In the bottom left, the "Call Stack" panel shows "Not paused". In the bottom right, the "Scope" and "Watch" panels are visible. The "Scope" panel shows "Global" with "PERSISTENT: 1" and "TEMPORARY: 0". The "Watch" panel is empty. On the right side of the screen, there is a video player showing a man with long hair and a beard, wearing an orange shirt, looking slightly to the side.

- If function defined with fat arrow function or variable access



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

Namaste 🙏 JavaScript

Paused on breakpoint

Call Stack

(anonymous) index.js:3

Breakpoints

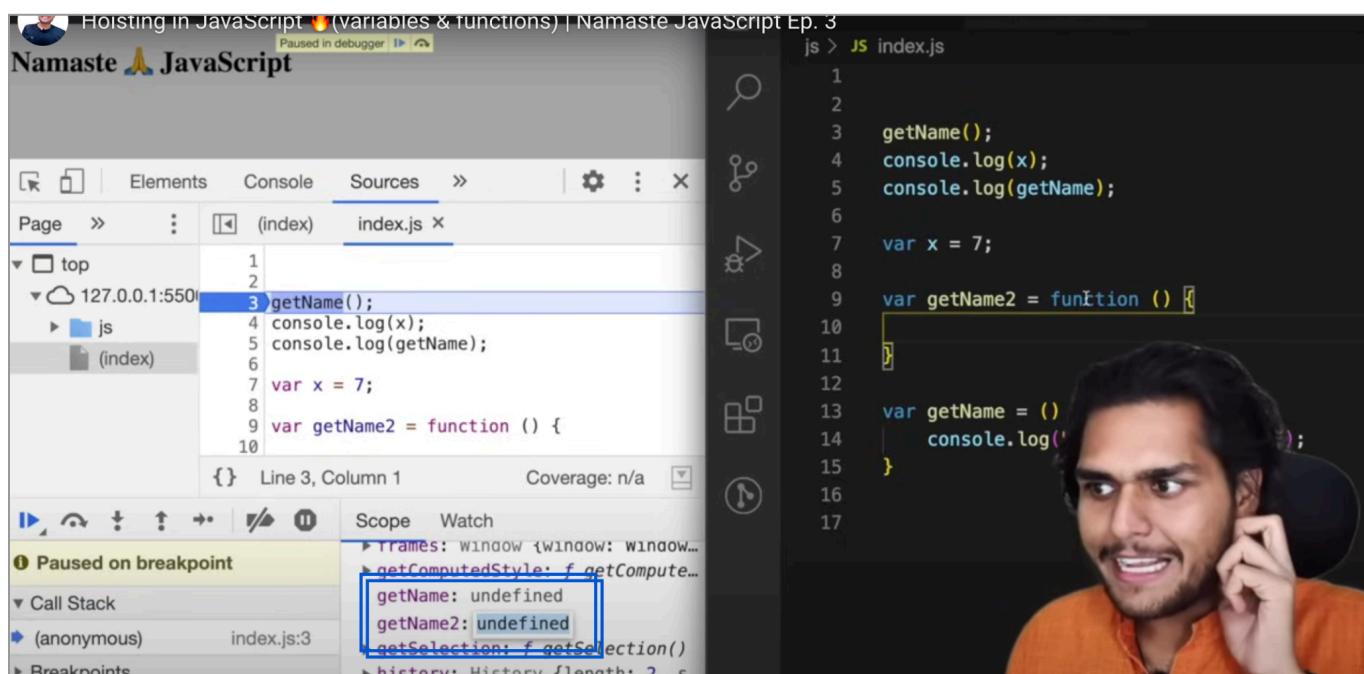
XHR/fetch Breakpoints

Scope Watch

```

1
2
3 getName();
4 console.log(x);
5 console.log(getName);
6
7 var x = 7;
8
9 var getName = () => {
  console.log("Namaste Javascript");
10}
11
12
13
    
```

frameElement: null  
 frames: Window {window: Window...}  
 getComputedStyle: f getCompute...  
 getName: undefined  
 getSelection: f getSelection()  
 history: History {length: 2, s...}  
 indexedDB: IDBFactory f



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

Namaste 🙏 JavaScript

Paused on breakpoint

Call Stack

(anonymous) index.js:3

Breakpoints

Scope Watch

```

1
2
3 getName();
4 console.log(x);
5 console.log(getName);
6
7 var x = 7;
8
9 var getName2 = function () {
10}
11
12
13 var getName = () {
  console.log('
14')
15
16
17
    
```

frames: WINDOW {WINDOW: Window...}  
 getComputedStyle: f getCompute...  
 getName: undefined  
 getName2: undefined  
 getSelection: f getSelection()  
 history: History {length: 2, s...}

- Call Stack in browser - maintains the order of Execution of Execution contexts

The screenshot shows the Chrome DevTools debugger. The 'Sources' tab is selected, displaying the file 'index.js'. A breakpoint is set on line 5 of the 'getName' function. The 'Call Stack' panel on the left shows two entries: 'getName' at index.js:5 and '(anonymous)' at index.js:8. The 'Scope' panel on the right shows the current execution context, which is a 'Window' object with properties like 'PERSISTENT: 1', 'TEMPORARY: 0', and 'addEventListener: f'. The code editor on the right shows the same code with the breakpoint highlighted.

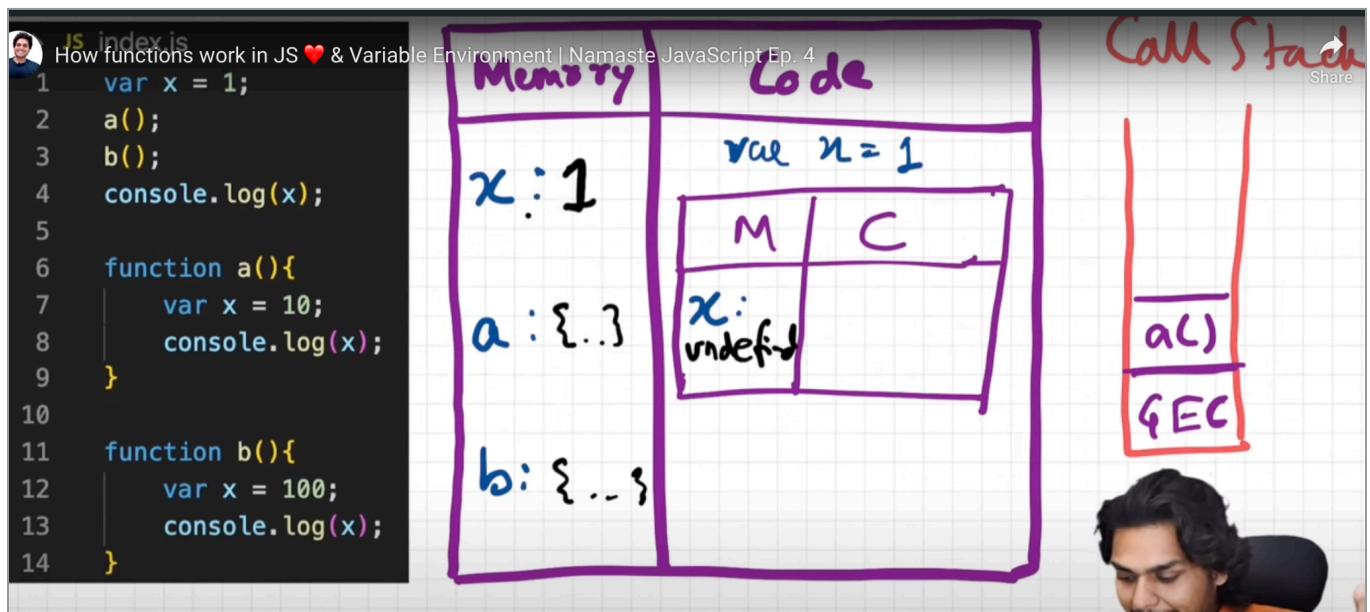
## ● How functions work in JS ❤️ & Variable Environment

- Functions are heart of javascript, act as mini program
- Global Execution context & functions

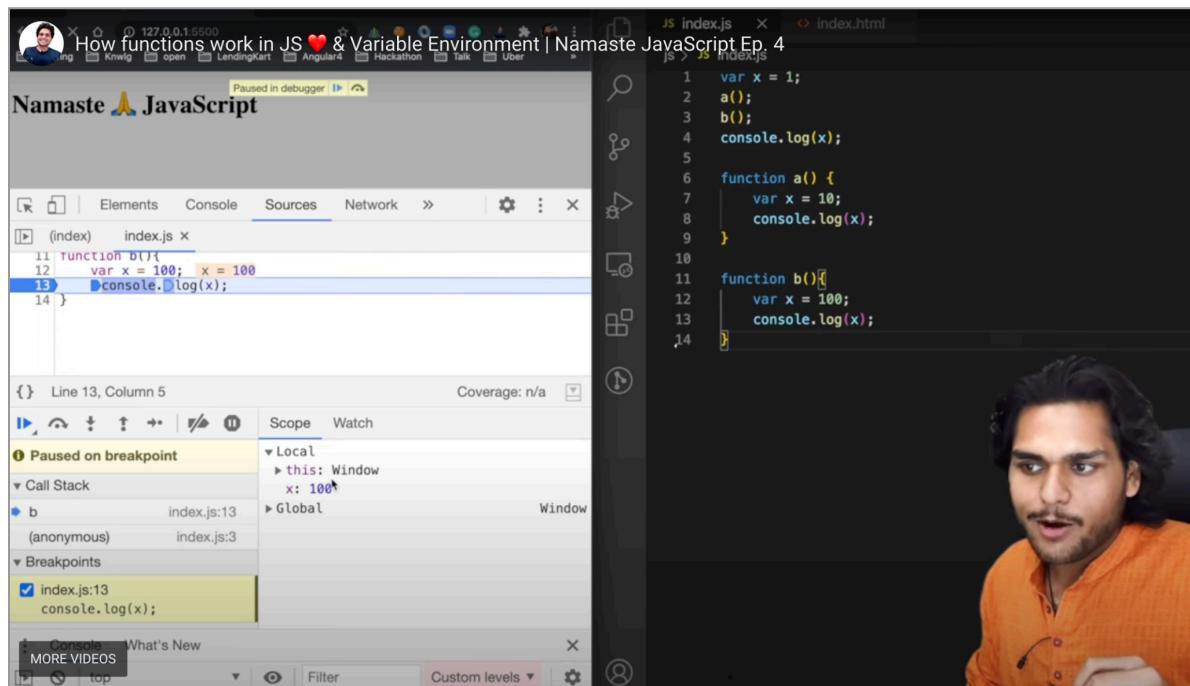
The diagram illustrates the variable environment and call stack. On the left, a code editor shows a script with variables 'x' being assigned values 1, 10, and 100 in different scopes. To the right, a large box is divided into three sections: 'Memory' (containing 'x: undefined', 'a: {}', and 'b: {}'), 'Code' (empty), and 'Call Stack' (containing 'GEC'). A portrait of the instructor is visible at the bottom right.

```

js > JS index.js
1  var x = 1;
2  a();
3  b();
4  console.log(x);
5
6  function a(){
7      var x = 10;
8      console.log(x);
9  }
10
11 function b(){
12     var x = 100;
13     console.log(x);
14 }
  
```



- While executing Each function has its own / separate memory space, which is independent
- By executing JS code, will understand
  - Call stack, where is the controls with line number
  - Separate memory space for function (local, global memory)
  - Memory creation & Code execution phase for Execution contexts



- **SHORTEST JS Program 🔥 window & this keyword**

- Shortest JS program is empty JS file
- What happens when run Empty JS file in browser
  - Still Global Execution will create & global spaces
- JS engine will create global window object
- window is global object which created along with Global Execution Context when you run any JS code
- All JS engines have responsibility to creates the window global object
- In global level, window is equals to this  
window === this ⇒ returns true

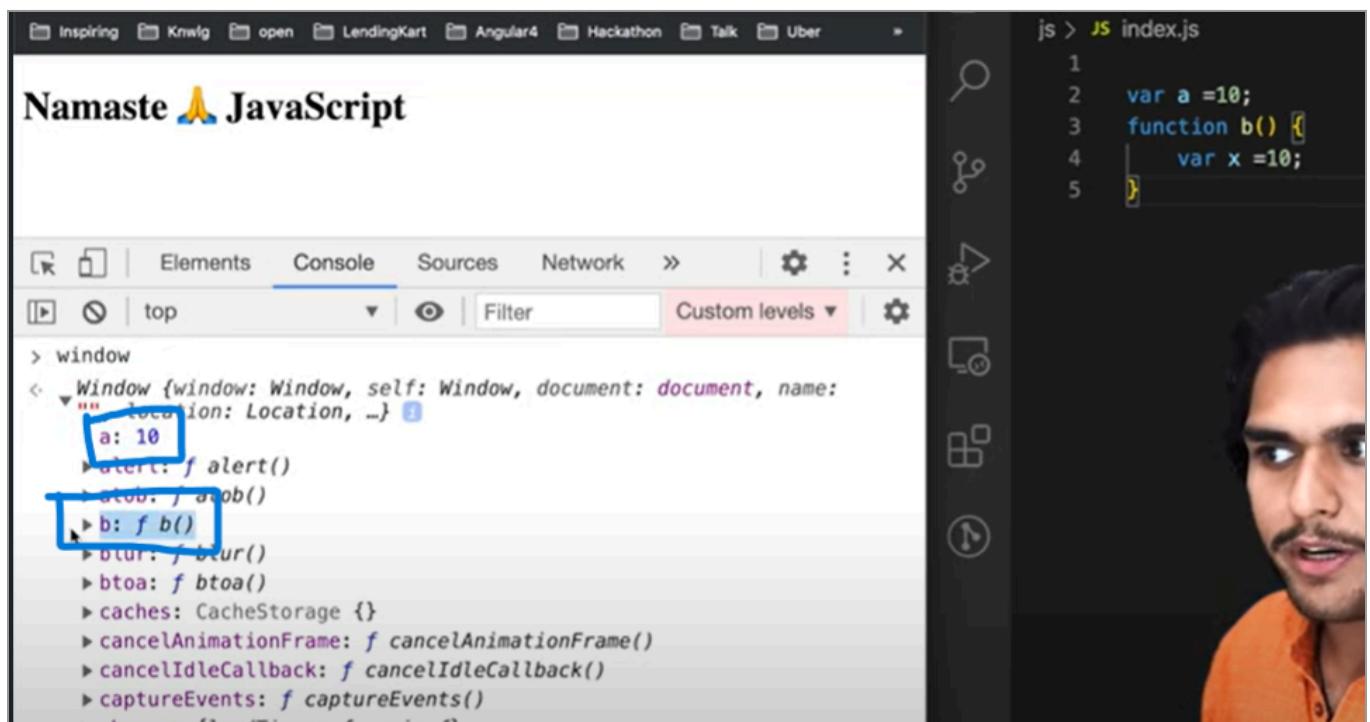
The screenshot shows a browser developer tools console with the title "Namaste ⚒ JavaScript". The console tab is selected. The output area displays the following JavaScript session:

```
> window
< Window {window: Window, self: Window, document: document, name: "", location: Location, ...}
> this
< Window {window: Window, self: Window, document: document, name: "", location: Location, ...}
> this === window
< true
>
```

The right side of the developer tools interface shows a sidebar with various icons for navigation and search.

- Whenever **global execution context created**, along with **window & this will create** and even for function execution context **this will create**
- Global Space
  - Any code write in JS which is not inside the functions

- All variables & function which is in Global spaces, are attached to global window / this object



Namaste  JavaScript

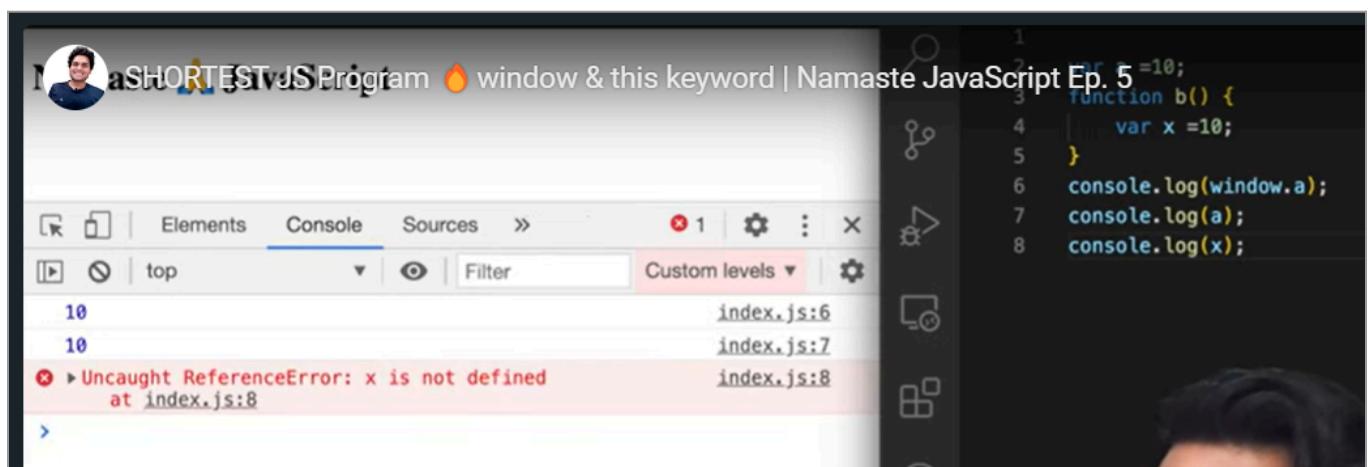
Elements Console Sources Network

top Custom levels

```
> window
< Window {window: Window, self: Window, document: document, name:
  a: 10
  ▶ alert: f alert()
  ▶ blob: f blob()
  ▶ b: f b()
  ▶ blur: f blur()
  ▶ btoa: f btoa()
  ▶ caches: CacheStorage {}
  ▶ cancelAnimationFrame: f cancelAnimationFrame()
  ▶ cancelIdleCallback: f cancelIdleCallback()
  ▶ captureEvents: f captureEvents()
```

js > JS index.js

```
1 var a =10;
2 function b() {
3   var x =10;
4 }
```



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

Elements Console Sources

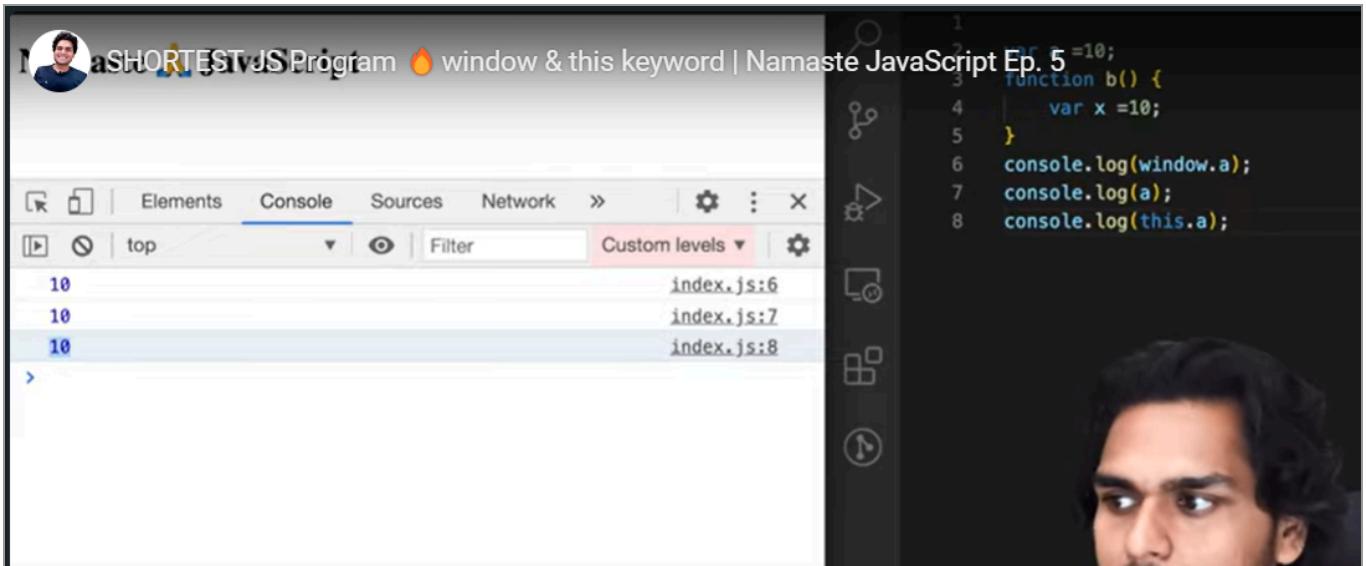
top Custom levels

```
10
10
✖ ▶ Uncaught ReferenceError: x is not defined
  at index.js:8
```

index.js:6
index.js:7
index.js:8

js > JS index.js

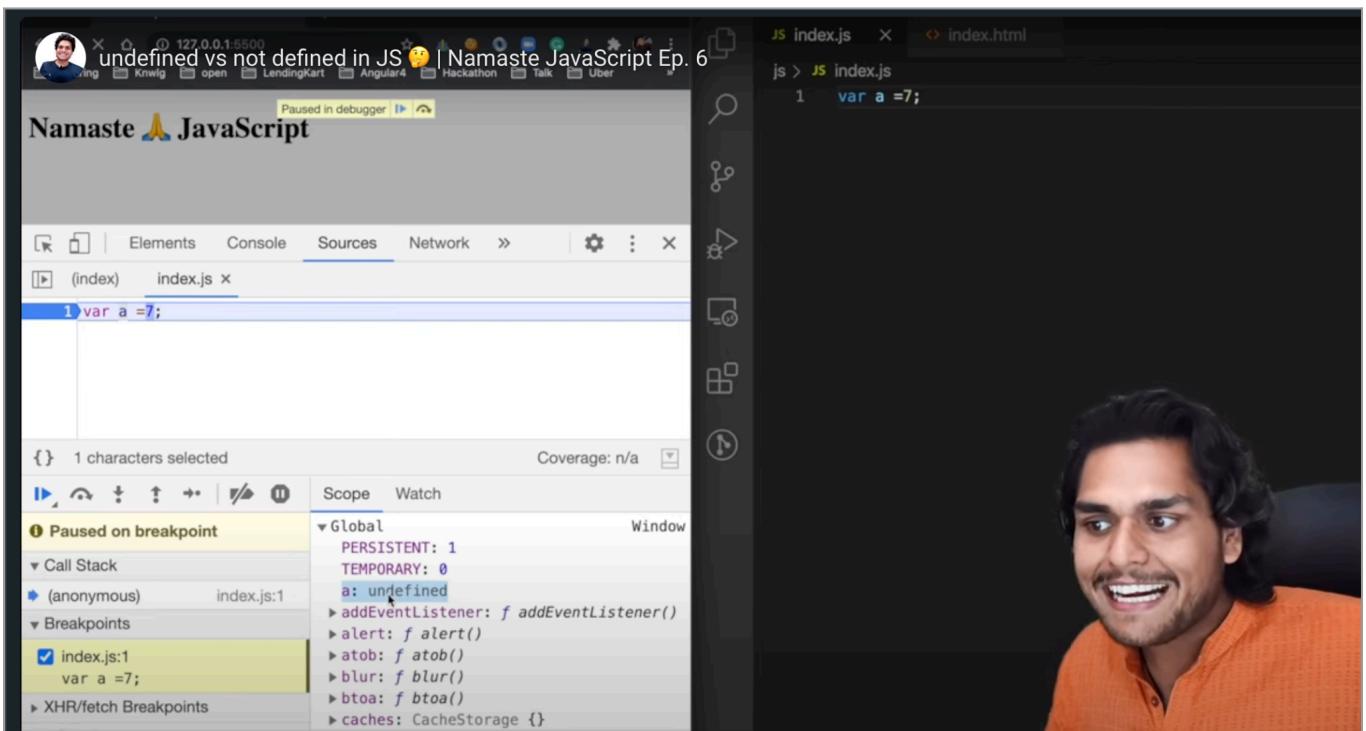
```
1 var a =10;
2 function b() {
3   var x =10;
4 }
5
6 console.log(window.a);
7 console.log(a);
8 console.log(x);
```



A screenshot of a browser's developer tools showing the 'Console' tab. The output area displays three identical entries: '10'. To the right of the console, there is a video player interface showing a man with dark hair and a beard, likely the video host.

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

- **undefined vs not defined in JS 🤔**
  - Undefined is very special keyword in JS language
  - Undefined comes into picture during memory allocation phase, before starts executing the code

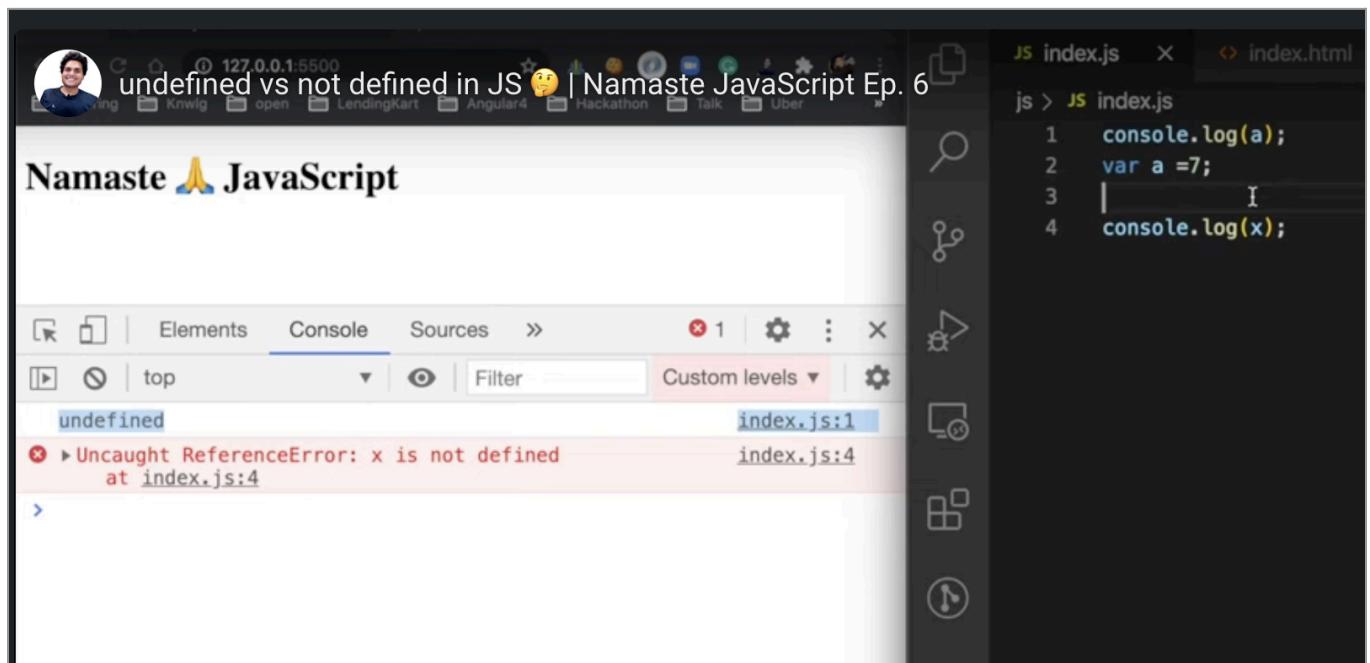


A screenshot of a browser's developer tools showing the 'Sources' tab with a breakpoint set at 'index.js:1'. The code 'var a =7;' is visible. In the bottom right corner, there is a video player interface showing a smiling man with dark hair and a beard.

```
1 var a =7;
```

- **undefined is like placeholder placed in the memory** during JS variables memory allocation, until assign some values to JS variables

- “undefined” is completely different from “not defined”  
“not defined” will get which is not allocated memory



The screenshot shows a browser window with the title "Namaste JavaScript". The developer tools' "Console" tab is active, displaying the following output:

```

undefined
Uncaught ReferenceError: x is not defined
at index.js:4
>

```

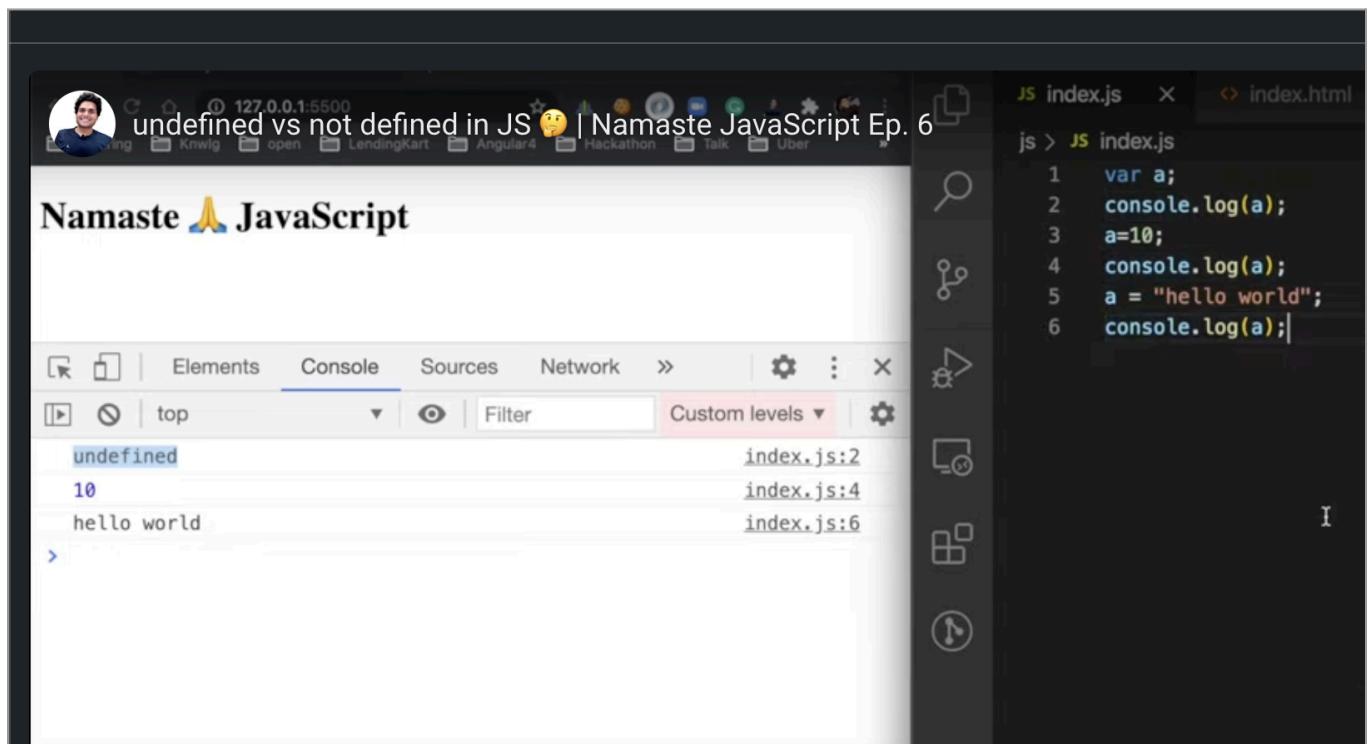
At the top right of the developer tools interface, there is a code editor window titled "JS index.js" containing the following code:

```

1 console.log(a);
2 var a =7;
3
4 console.log(x);

```

- JS is loosely typed / weakly typed language, since JS does not attach its variable to any specific data types  
JS is very flexible to changes its data types for variable, calling dynamically typed language



The screenshot shows a browser window with the title "Namaste JavaScript". The developer tools' "Console" tab is active, displaying the following output:

```

undefined
10
hello world
>

```

At the top right of the developer tools interface, there is a code editor window titled "JS index.js" containing the following code:

```

1 var a;
2 console.log(a);
3 a=10;
4 console.log(a);
5 a = "hello world";
6 console.log(a);

```

- Never do the mistake by assigning “undefined” to variable, not a good practice

Like

```
var a = undefined
```

- **The Scope Chain, 🔥 Scope & Lexical Environment**

- Sdfasdf
- Dsfasd
- Dsfas

- sdASF
- FDSsf
- Sfaf
- S
- Ff
- FDSAF