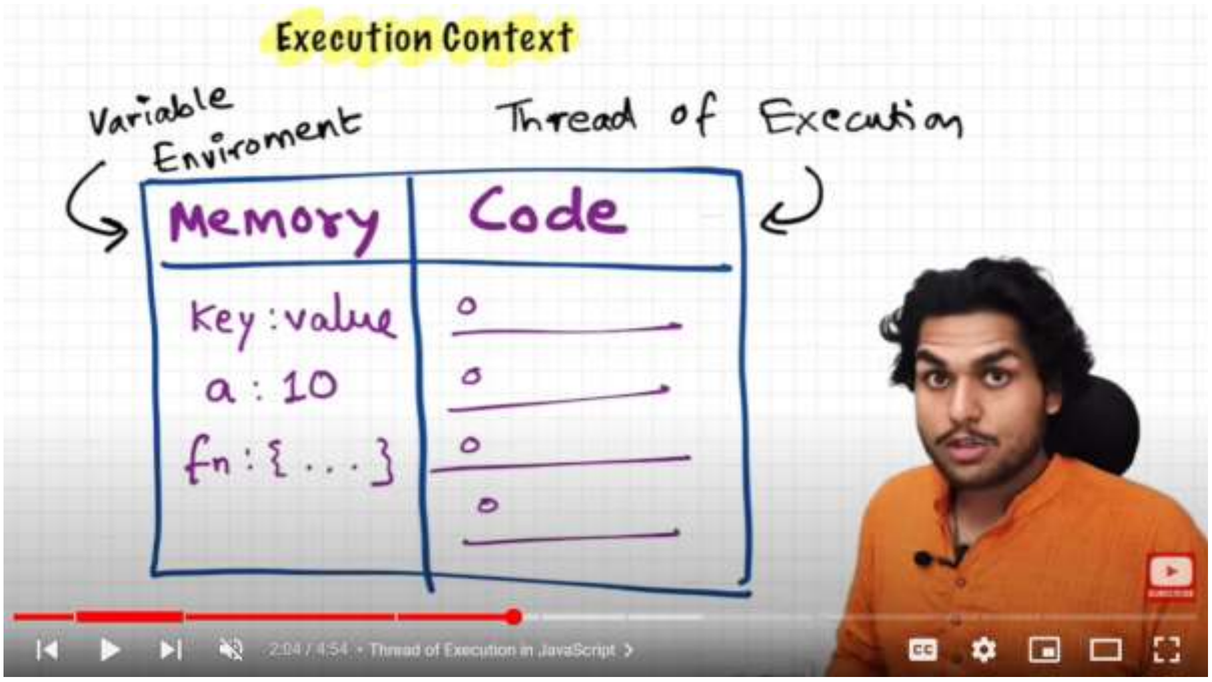
NAMASTE JS NOTES BY AAYUSH PAGARE

UNIT1 : EXECUTION CONTEXT

“ Everything in JavaScript happens inside this execution context “

Imagine Execution context is like a big box which has two components

1) Memory Component (*Variable Environment)*
2) Code Component (*Thread of Execution*)



Memory Component stores variable and functions as Key:Value pairs.

Code Component stores the whole code and executes it line by line .

NOTE : JavaScript is synchronous single-threaded language.

Single threaded : it can only execute one command at a time.

Synchronous :- it executes command in an specific order.

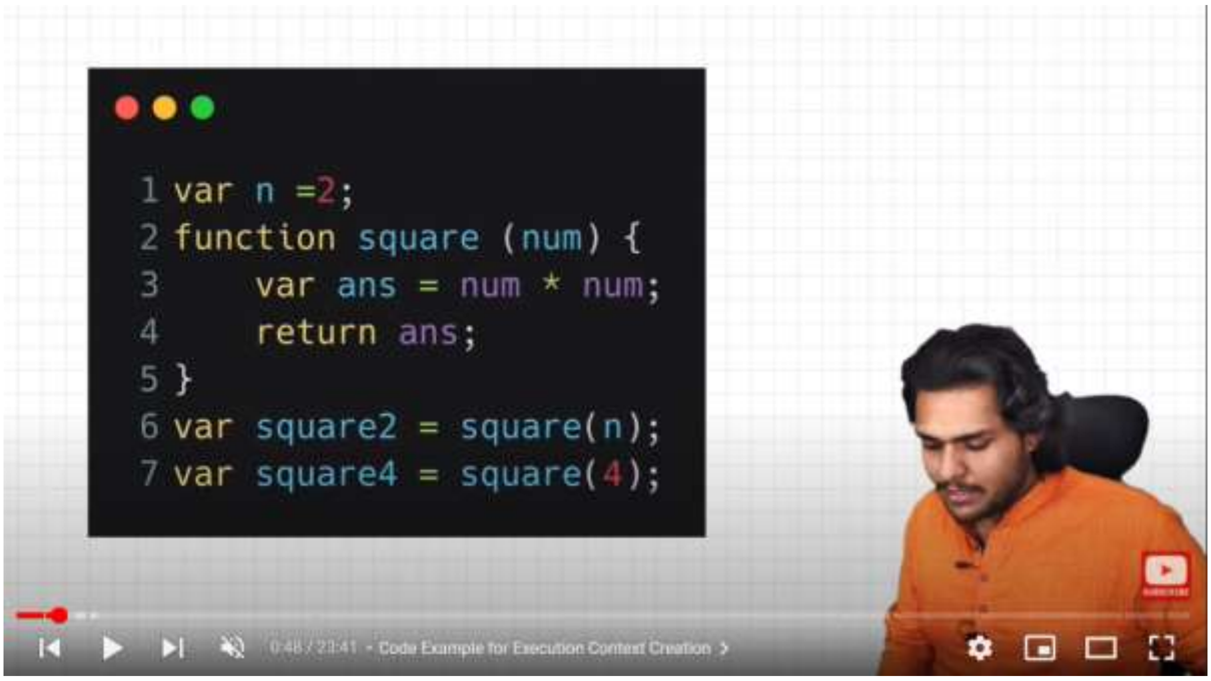
UNIT2 : HOW JS CODE IS EXECUTED :

How does JS Code is executes??

So, what happens when you run a JS program??.

“A brand new global execution context is created when we run a JS program”

To understand this les take an example



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

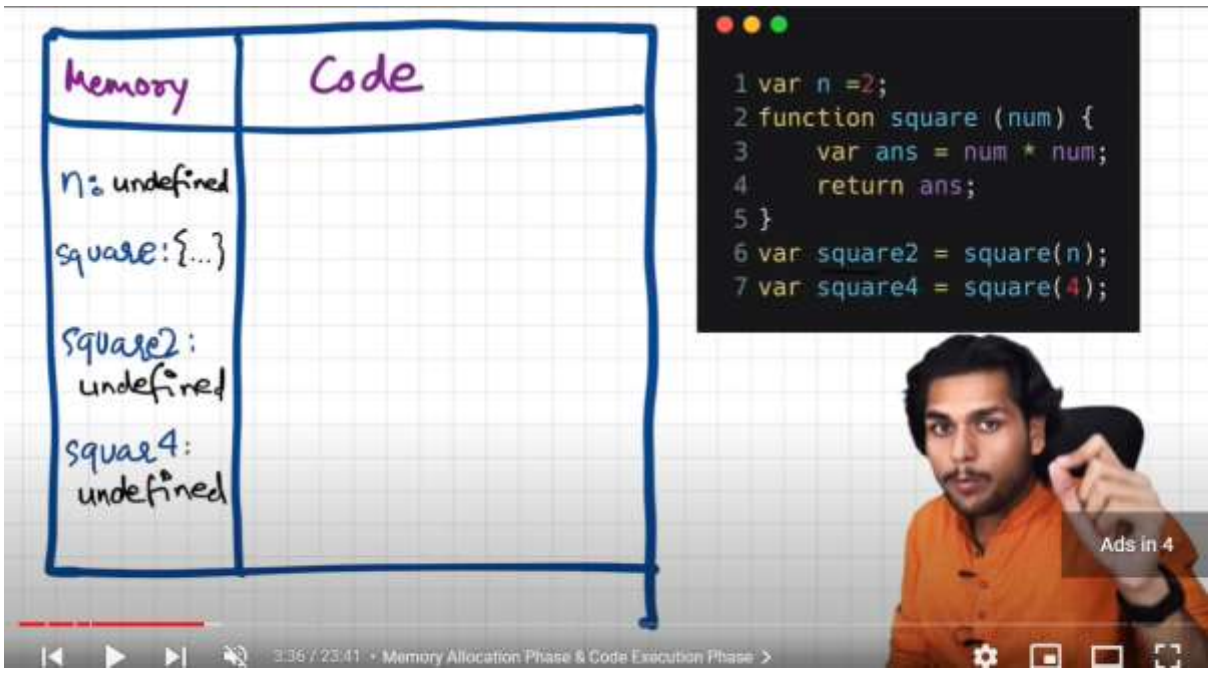
Now we will see how this code is executed by JS Engine

STEP 1 : AN EXECUTION CONTEXT IS CREATED IN TWO PHASES :

PHASE 1 : The Creation Phase: *(Memory Creation Phase)*

JavaScript scans the whole code and allocates memory to each variable and functions.

WHEN IT ALLOCALTES THE MEMORY TO VARIABLES INITIALLY IT STORES "undefined" AS THE VALUE IN THE KEY:VAL PAIR AND FOR FUNCTIONS IT LITERALLY STORES "the whole code of the function" AS THE VALUE IN THE KEY:VAL PAIR.

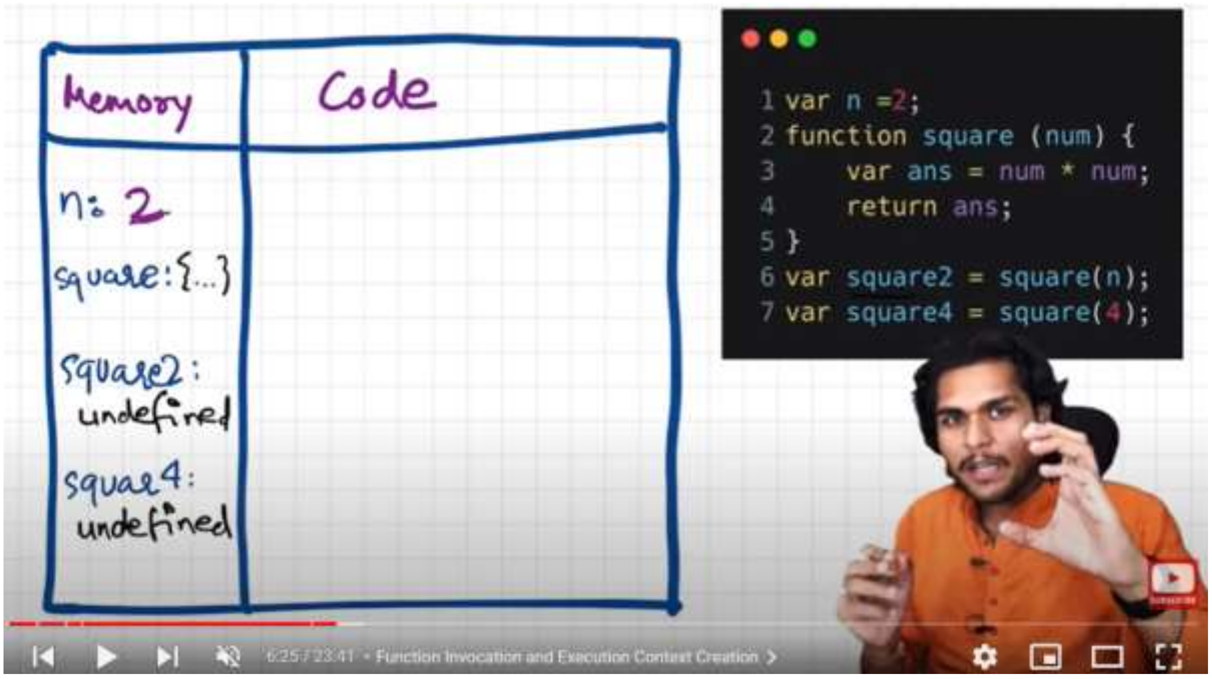


```
var n =2;
function square (num) {
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

CREATION PHASE OVER :

PHASE 2: The Code Execution Phase :

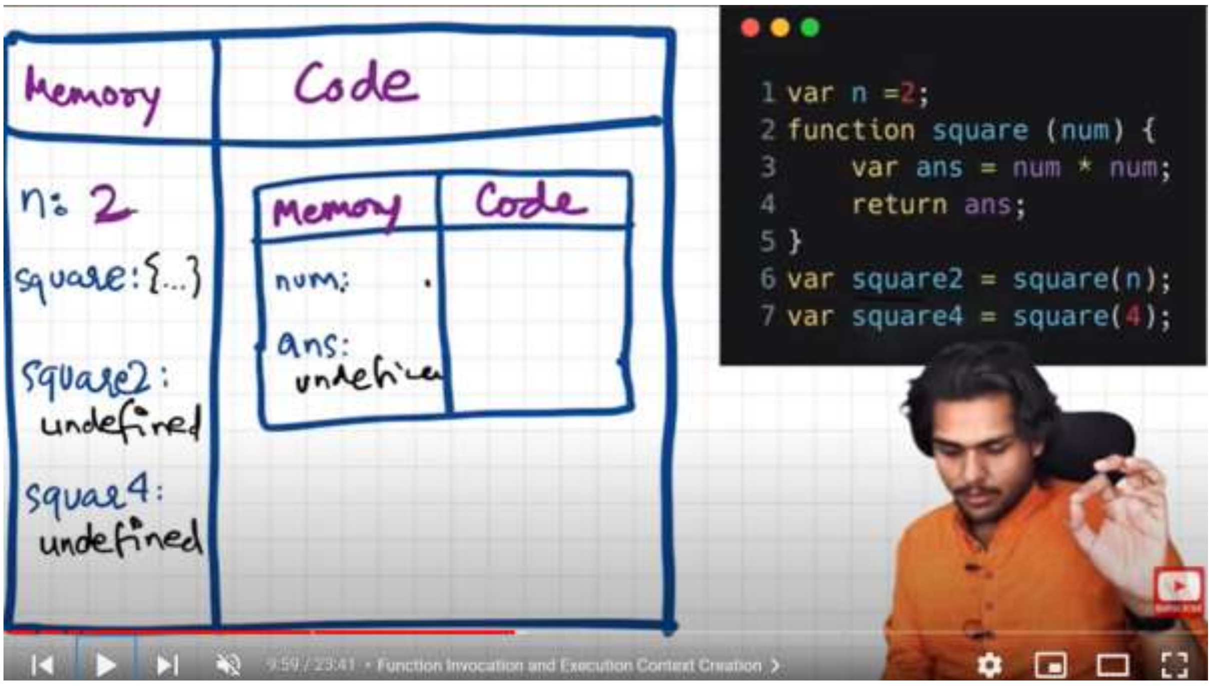
It scans the code line by line and changes the value of variables in the key:val pair from undefined to the actual value assigned to them in the code.


Memory
Code
n: 2
square: {...}
square2: undefined
squar4: undefined
6:25 / 23:41 • Function Invocation and Execution Context Creation

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

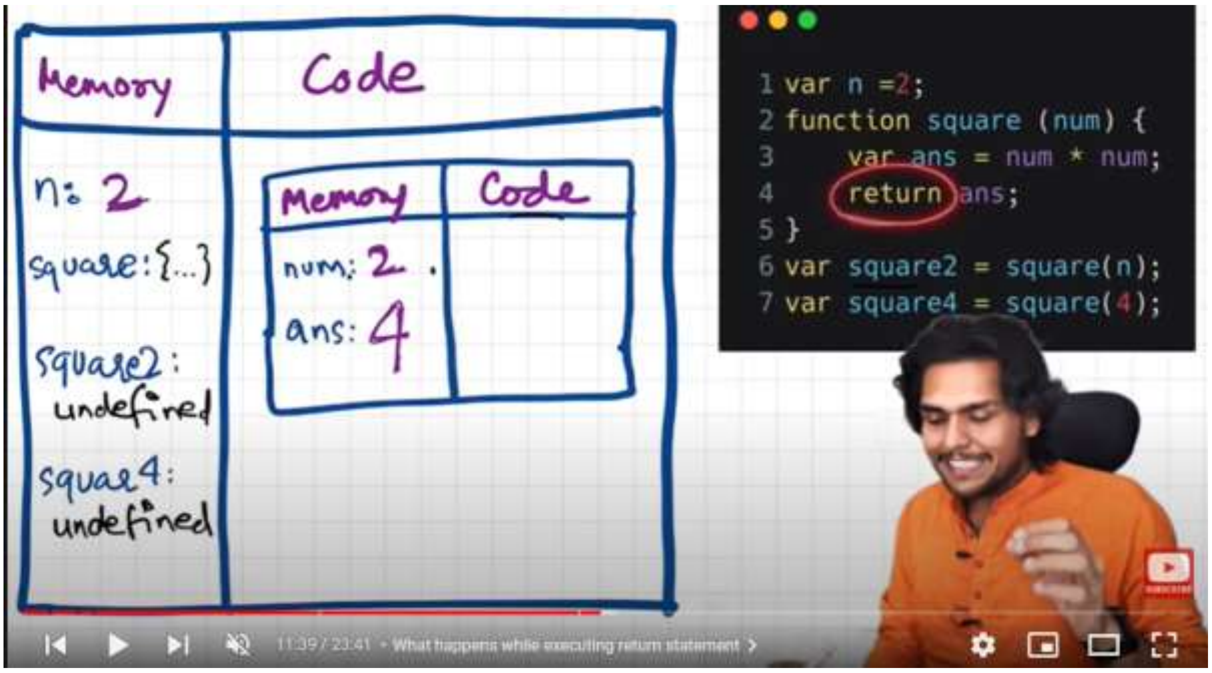
This was for variables now what for functions?.

Whenever a function is invoked a separate brand new execution context is created for that function for executing the code of that function and repeats both the phases of exec context.



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

PHASE 1 :



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

PHASE 2:

***NOTE : WHENEVER THE "return" KEYWORD IS ENCOUNTERED THE CONTROL OF THE PROGRAM IS RETURNED TO THE LINE FROM WHERE IT WAS ENCOUNTERED AND IT** DELETES THE EXECUTION CONTEXT OF THAT FUNCTION*

Concept of ***"CALL STACK"***

To manage this exec context creation, deletion JavaScript maintains a Stack known as Call Stack. Whenever a function is invoked it's execution context is created and pushed in the call stack and whenever the execution context is deleted it pops out from the call stack.

CALL STACK MAINTAINS THE ORDER OF THE EXECUTION OF EXECUTION CONTEXTS.

Execution context stack

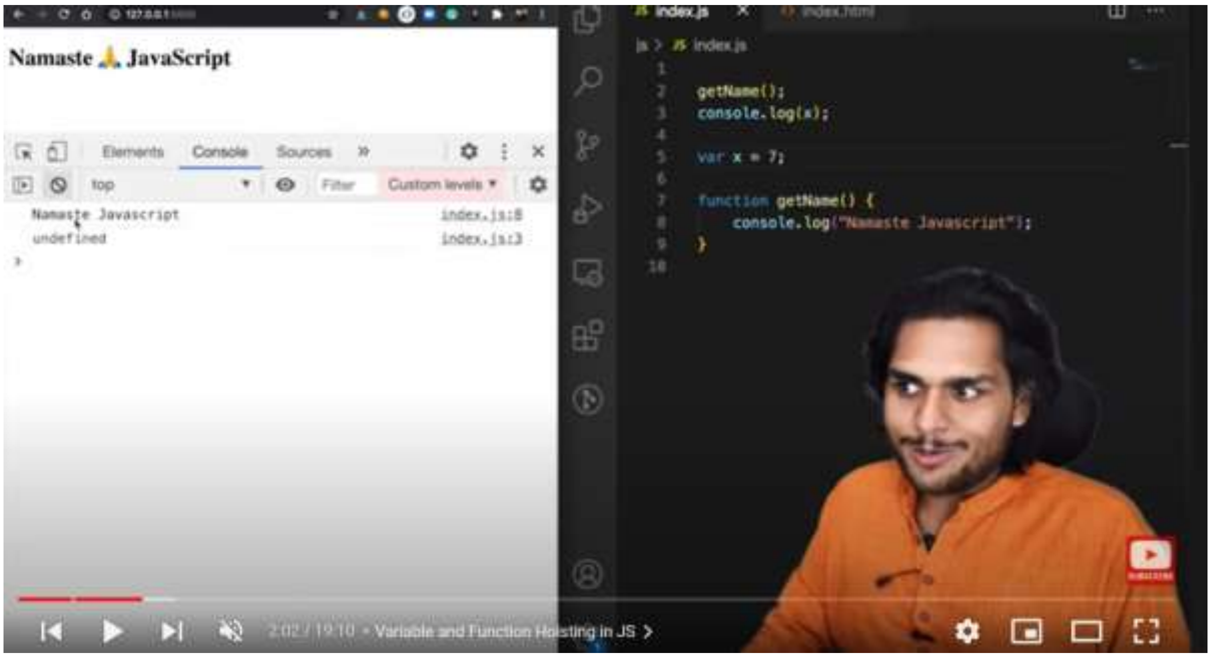
Program stack

Machine stack

Control stack

Runtime stack are some fancy names of call stack.

UNIT3 : HOISITING IN JAVASCRIPT



```
Namaste Javascript                    index.js:8
undefined                             index.js:3
```

```

getName();
console.log(x);

var x = 7;

function getName() {
    console.log("Namaste Javascript");
}

```

Look at the code:

Here we have tried to access a variable even before we have declared it in most of the programming language it would give us a “Runtime Error”

But JavaScript instead prints “undefined”.

WHY UNDEFINED WAS PRINTED ??

Reason is HOISTING :

JAVASCRIPT HOISTING REFERS TO THE PROCESS WHEREBY THE INTERPRETER APPEARS TO MOVE THE DECLARATIONS OF THE FUNCTIONS, VARIABLES OR CLASSES TO THE TOP OF THEIR SCOPE, PRIOR TO THE EXECUTION OF THE CODE.

MY definition :

HOISTING IS THE PROCESS WHERE JAVASCRIPT SKIMS THROUGH THE CODE BEFORE THE EXECUTION AND IT ALLOCATES MEMORY TO THE VARIABLES AND FUNCTION EVEN BEFORE EXECUTING THE CODE

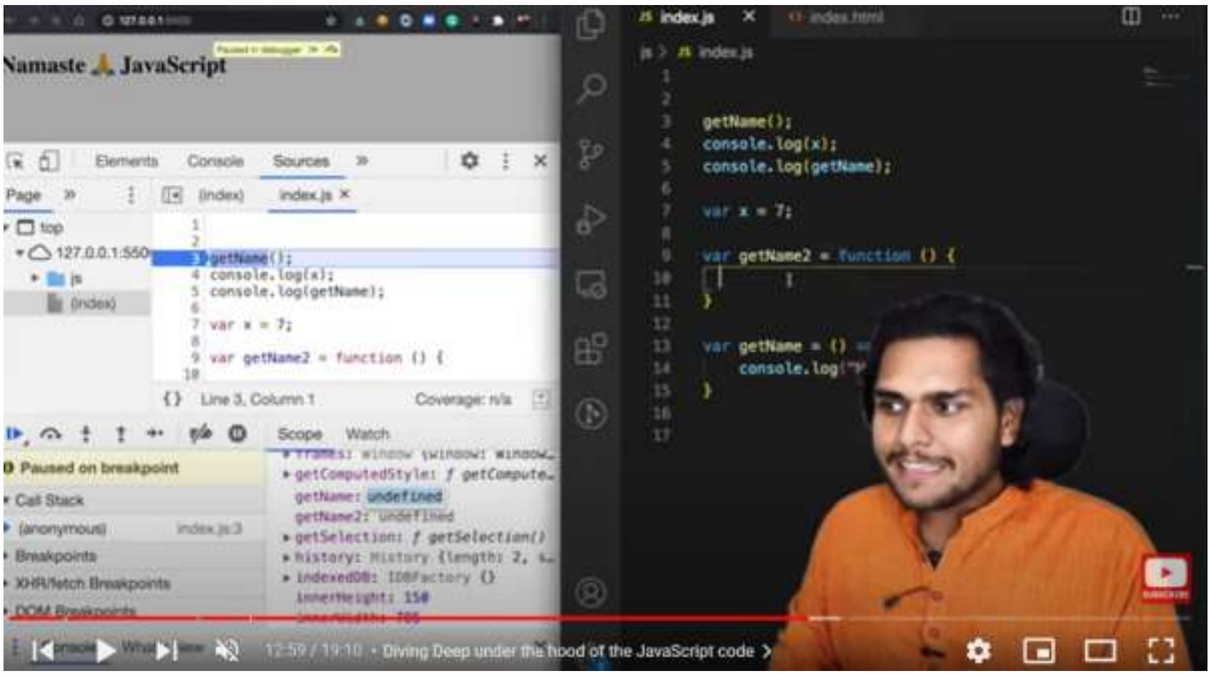
HOSITING ALLOWS FUNCTIONS TO BE SAFELY USED IN CODE BEFORE THEY ARE DECLARED

The code printed undefined because as execution context was created and it had assigned the value “undefined” to variable x in the creation phase.

So until and unless the line var x = 7; will not be encountered all the clogs accessing x would log undefined on the o/p screen, and for the functions if invoked will be executed it logged will print the whole function body.

NOTE :

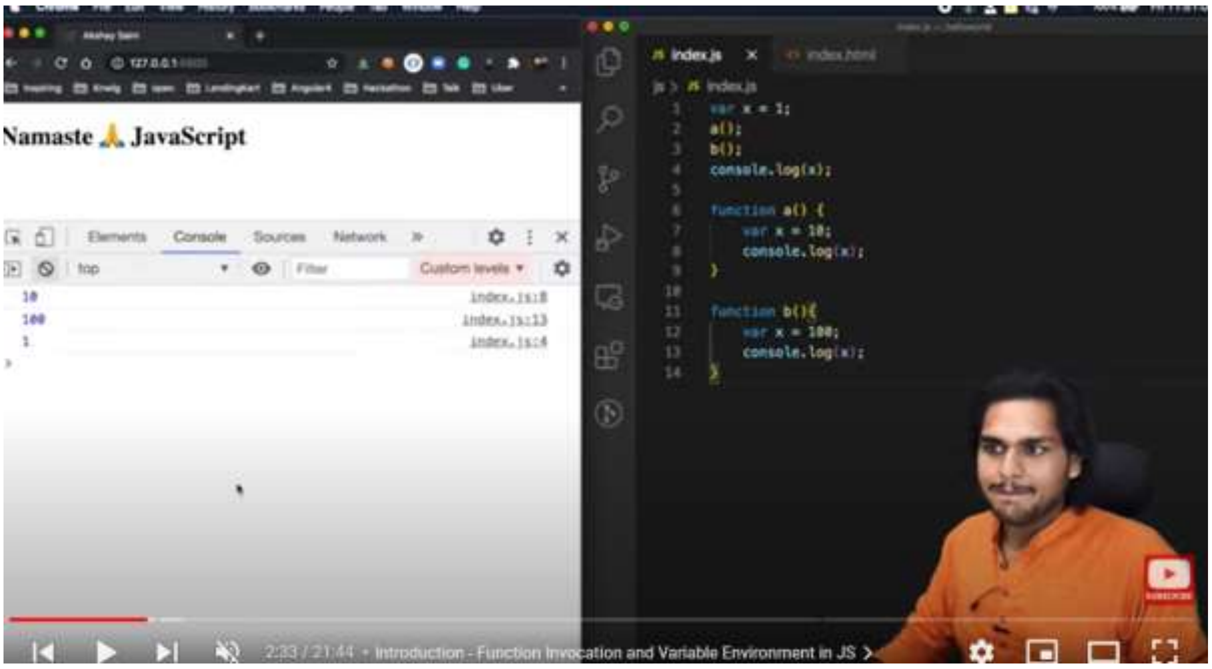
ARROW FUNCTIONS **AND FUNCTIONS WRITTEN IN SYNTAX LIKE** ***“ var xyz = function abc () {…..} ”*** **WILL BEHAVE AS VARIABLES AND NOT AS FUNCTIONS**



```
getName();
console.log(x);
console.log(getName);

var x = 7;

var getName2 = function () {

}

var getName = () =>
    console.log("
}
```

```
Paused on breakpoint
getName: undefined
getName2: undefined
```

UNIT 4 : HOW FUNCTIONS WORK IN JS



Namaste 🙏 JavaScript

```
10                                   index.js:8
100                                  index.js:13
1                                    index.js:4
```

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

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

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

Elements
Console
Sources
Network
top
Filter
Custom levels
index.js
index.html
2:33 / 21:44 • Introduction - Function Invocation and Variable Environment in JS

EXPLAINATION:

A global exec is created which looks like (push global exec to call stack)

- ⇨ Creation Phase for exec global

  x : undefined

  a : {…}

  b {….}

- ⇨ Execution Phase for exec global

  x : 1

  Creates new exec for a (push exec a to call stack)

  - Creation Phase for exec a

x : undefined

- Execution phase for exec a

  x : 10

  LOGS 10 TO CONSOLE

DELETE exec a (pop exec a from call stack)

Creates new exec for b (push exec b to call stack)

- Creation Phase for exec b

  x : undefined

- Execution phase for exec b

  x : 100

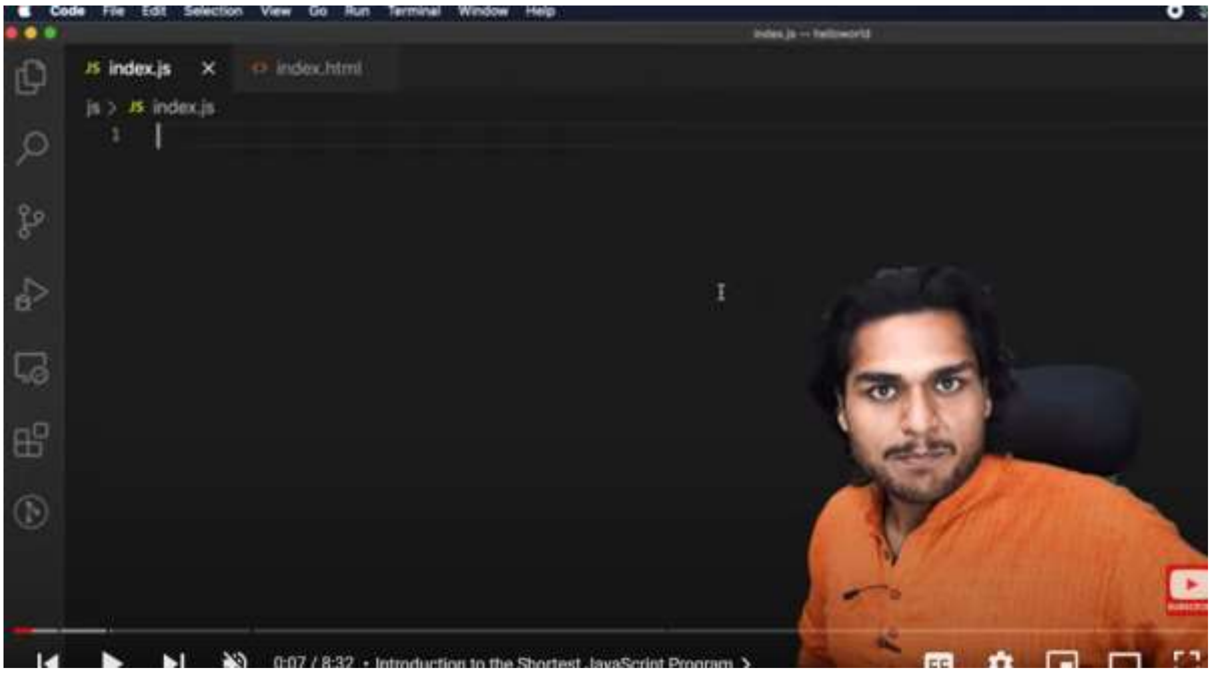
  LOGS 100 TO CONSOLE

DELETE exec b (pop exec b from call stack)

LOGS 1 TO CONSOLE

DELETE exec global (pop exec global from call stack)

UNIT 5 : SHORTEST JAVASCRIPT PROGRAM


index.js
index.html
js > index.js
0:07 / 8:32 • Introduction to the Shortest JavaScript Program

AN EMPTY FILE : SHORTES JS PROGRAM

Even though there is nothing to execute still JavaScript Engine executes the code by creating a ***"window object"*** and also a ***"this keyword"*** , at global level this keyword points to the global object.

"window" object is created by JavaScript Engine which is a big object with lots of functions, methods and variables, which could be accessed anywhere and anytime.

Whenever a JS program is created a window object is created, a this variable is created and a global exec is created.

JavaScript runs everywhere like on browsers, servers, mobile applications. So wherever JS runs there must be a JS engine over there for e.g. “chrome” has “V8”. All the JS engine has the responsibility to create a global object. In browser it’s known as window, in node something else.

At global level this === window

UNIT 6 : UNDEFINED V/S NOT-DEFINED

If you'll try to access a variable before it's declaration or initialization, but the variable is surely declared in the script in the later part it will print *"undefined"*

If you will access a variable which is never declared in a program it will give an *REFERNCE ERROR : "not defined"*

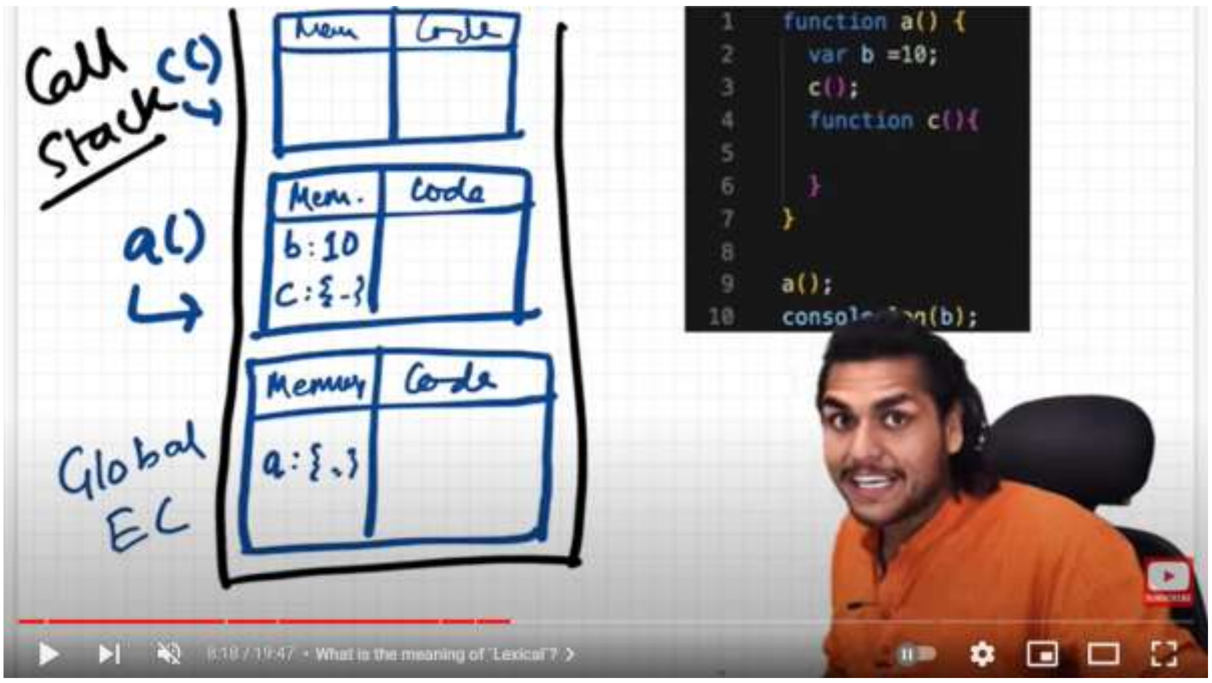
Script :

```
Console.log(a);
Var a = 10;
Outputs undefined
```

Script :

```
Console.log(a)
Outputs not-defined
```

UNIT 7 : THE SCOPE CHAIN , SCOPE AND LEXICAL ENVIRONMENT



```
function a() {
  var b =10;
  c();
  function c(){

  }
}

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

This explains how execution contexts are pushed in the call stack.

SCOPE : Scope of variable is where the variable could be accessed.

Is b inside the scope of function c means is b could be accessed inside c.

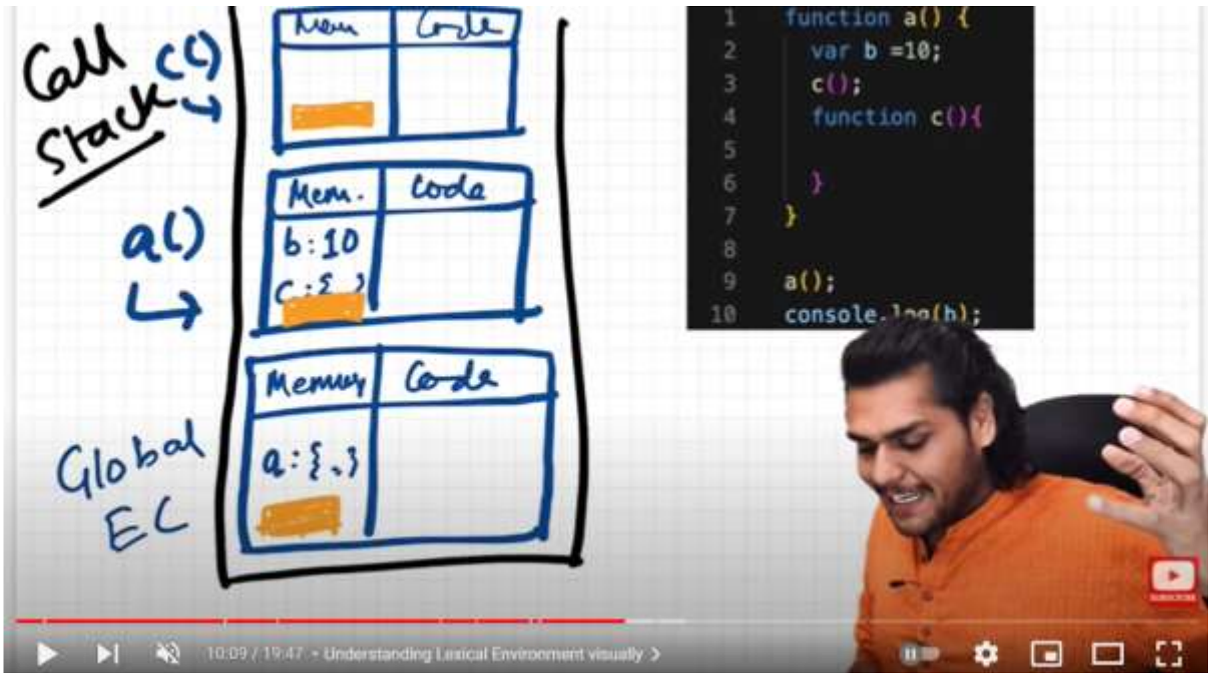
Scope depends directly on lexical environment.

LEXICAL ENVIRONMENT : Lexical environment is local memory + lexical environment of it's parent.

Meaning of lexical is hierarchy

In this case function a is lexically inside global, function c is lexically inside a.

Global is lexical parent of function a, function a is lexical parent of function c.



```
function a() {
  var b =10;
  c();
  function c(){

  }
}

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

Memory space of execution context consists of its variables, functions and also lexical environment of its lexical parent.

That orange thing represents lexical environment.

In this example when b is accessed in the global scope, engine searches lexical environment of global i.e. (VE(global) + LE(global)). It doesn't finds b in LE(global) hence it logs reference error of not -defined

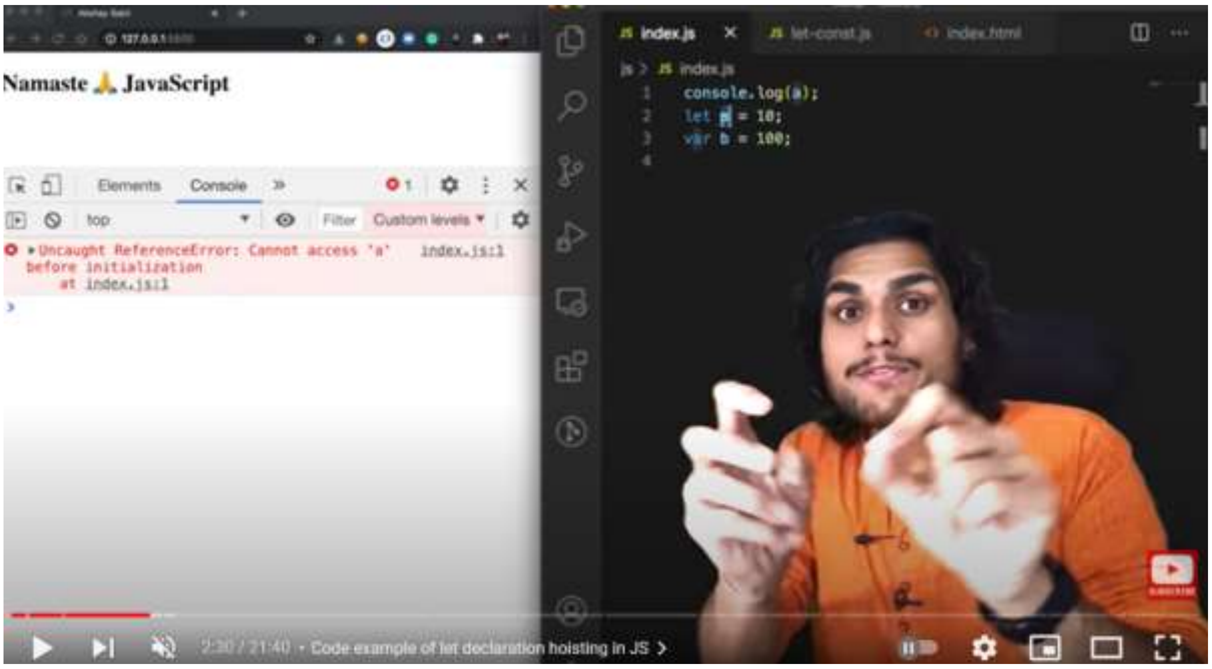
If b is accessed inside c then

b is searched in LE(c), result not found

b is searched in lexical parent of c, which is a so b is searched in LE(a), result found

UNIT 8 : LET VS CONST, TEMPORAL DEAD ZONE..

LET AND CONST DECLARATIONS ARE HOISTED BUT THEY ARE HOISTED DIFFERNETLY THAN VAR, INSTEAD THEY ARE IN TEMPORAL DEAD ZONE FOR A TIME BEING.

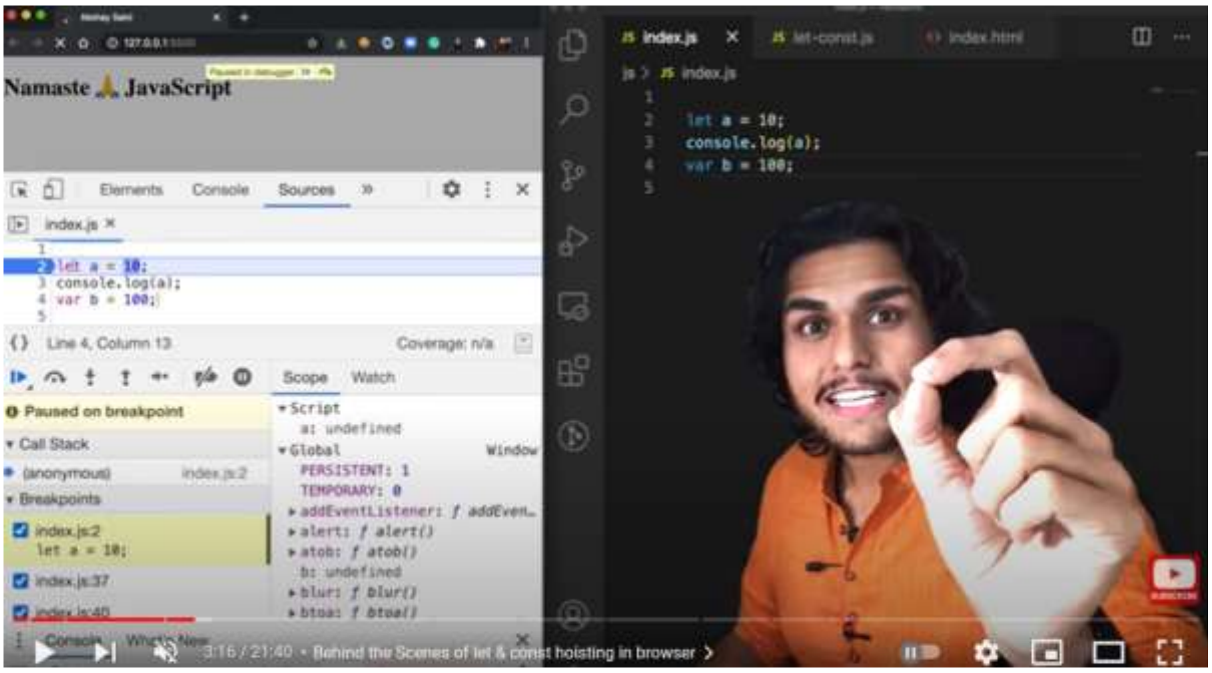


```
console.log(a);
let a = 10;
var b = 100;
```

```
▸Uncaught ReferenceError: Cannot access 'a'    index.js:1
 before initialization
     at index.js:1
```

Namaste 🙏 JavaScript
2:30 / 21:40 • Code example of let declaration hoisting in JS

Now if let variables are hoisted then why we got REFERNCE ERROR ?



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

```
Paused on breakpoint
▾Script
   a: undefined
▾Global                              Window
   PERSISTENT: 1
   TEMPORARY: 0
  ▸addEventListener: f addEven…
  ▸alert: f alert()
  ▸atob: f atob()
   b: undefined
  ▸blur: f blur()
  ▸btoa: f btoa()
```

In case of var b , b is allocated memory in the global space but let a is allocated memory in the a different space (Script). Hence they are also hoisted.

They are hoisted in different memory space and you can't access that memory space before initialization and declaration.

TEMPORAL DEAD ZONE :-

The time between the let variable was hoisted and was initialized.

If let variables are accessed during the temporal dead zone it will give us reference error.

Script :

Console.log(x)

Console.log(a)

Console.log(b)

Var a = 0;

Let b = 90;

REFERENCE ERROR x is not defined

REFERNCE ERROR a is not defined as a is in temporal dead zone

Undefined

NOTE :
Var variables are attached to the window object

Let and const are not attached to window object.

LET is a little more strict than var

Let a = 90;

Let a = 89; or var a = 78;

Not possible, throws SYNTAX ERROR

You can't use the name in the same scope again

But

Var a = 90;

Var a = 89;

Possible with var

Const is most strict, it requires initialization with declaration.

Const a ;

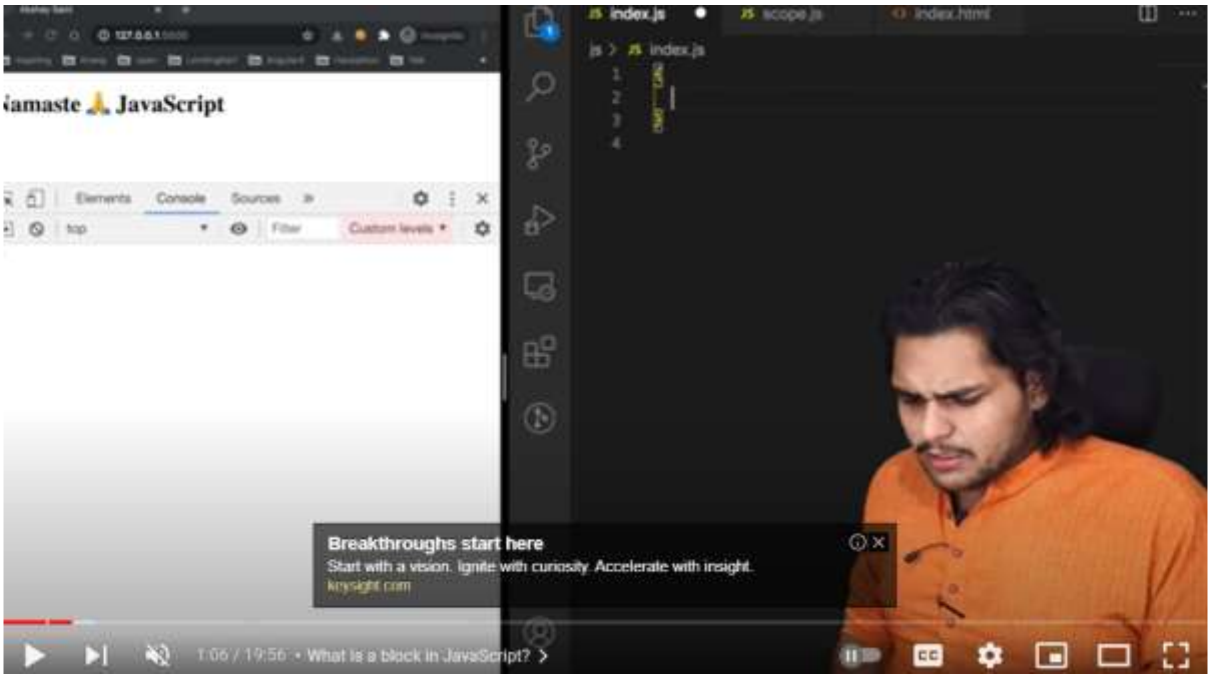
SYNTAX ERROR : missing initializer in the const declaration

xConst a = 90;

a = 909;

TYPE ERROR : assignment to const variable

UNIT 9 : BLOCK IN JAVASCRIPT


Namaste 🙏 JavaScript
Elements
Console
Sources
Breakthroughs start here
Start with a vision. Ignite with curiosity. Accelerate with insight.
keysight.com
1:06 / 19:56 • What is a block in JavaScript?
index.js
scope.js
index.html

Anything which is enclosed inside curly braces is known as block.

Block is used to combine multiple JavaScript statements into a group.

We need to group this statements in a group because we can use this statements as a whole where JavaScript expexcts them to be used as whole.

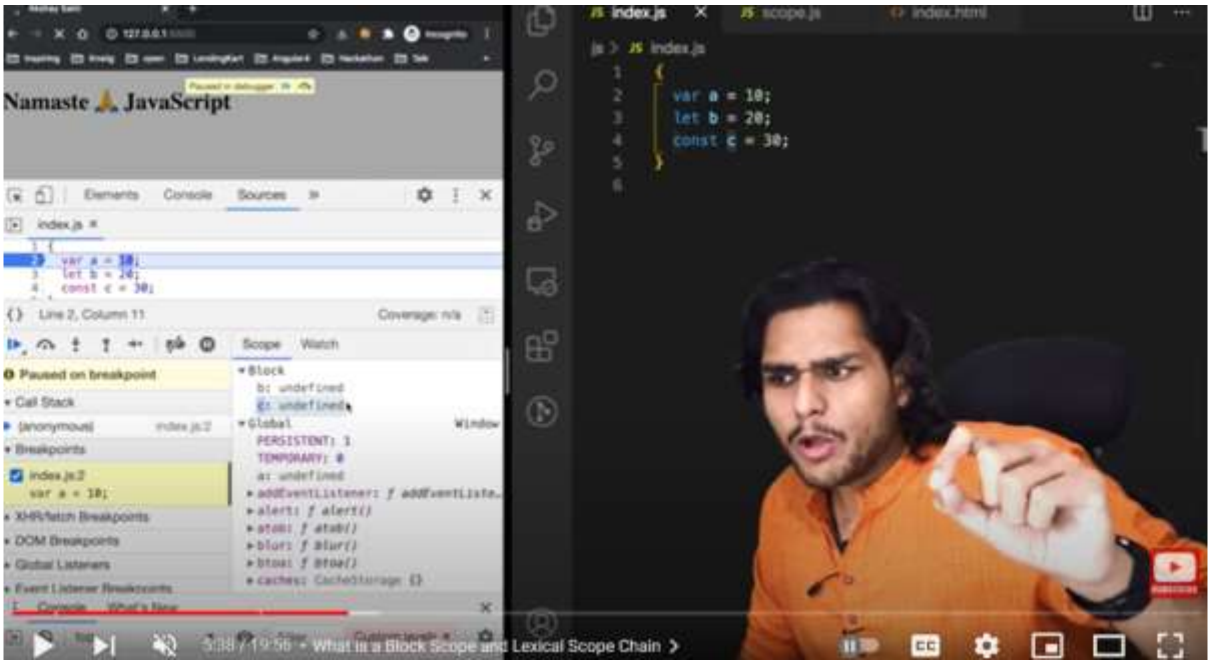
For e.g

If (a === i) expects only one statement

If (a == i) k = true; //true

```
If (a == i) {
        D = k;
        K = l;
}  //expects only single statement and they are grouped as a single statement by block
```

BLOCK SCOPE : What all variables and functions we could access inside this block.

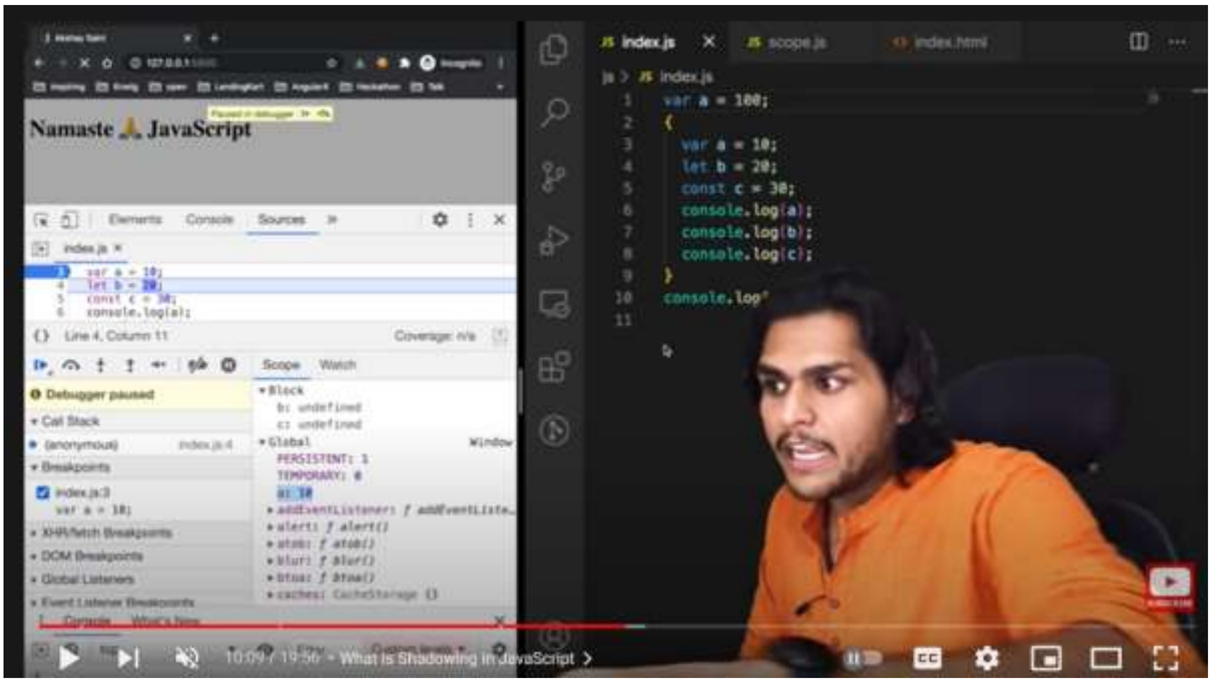


```
{
  var a = 10;
  let b = 20;
  const c = 30;
}
```

```
▾Block
  b: undefined
  c: undefined
▾Global                                   Window
  PERSISTENT: 1
  TEMPORARY: 0
  a: undefined
 ▸addEventListener: ƒ addEventListe…
 ▸alert: ƒ alert()
 ▸atob: ƒ atob()
 ▸blur: ƒ blur()
 ▸btoa: ƒ btoa()
 ▸caches: CacheStorage {}
```

Here b and c are hoisted in the block scope but a in a global scope.

LET AND CONST ARE BLOCK SCOPED. Because they are hoisted in a seprate memory space reserved for a block, Let and const can't be accessed outside the block.



```
 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  console.log
11
```

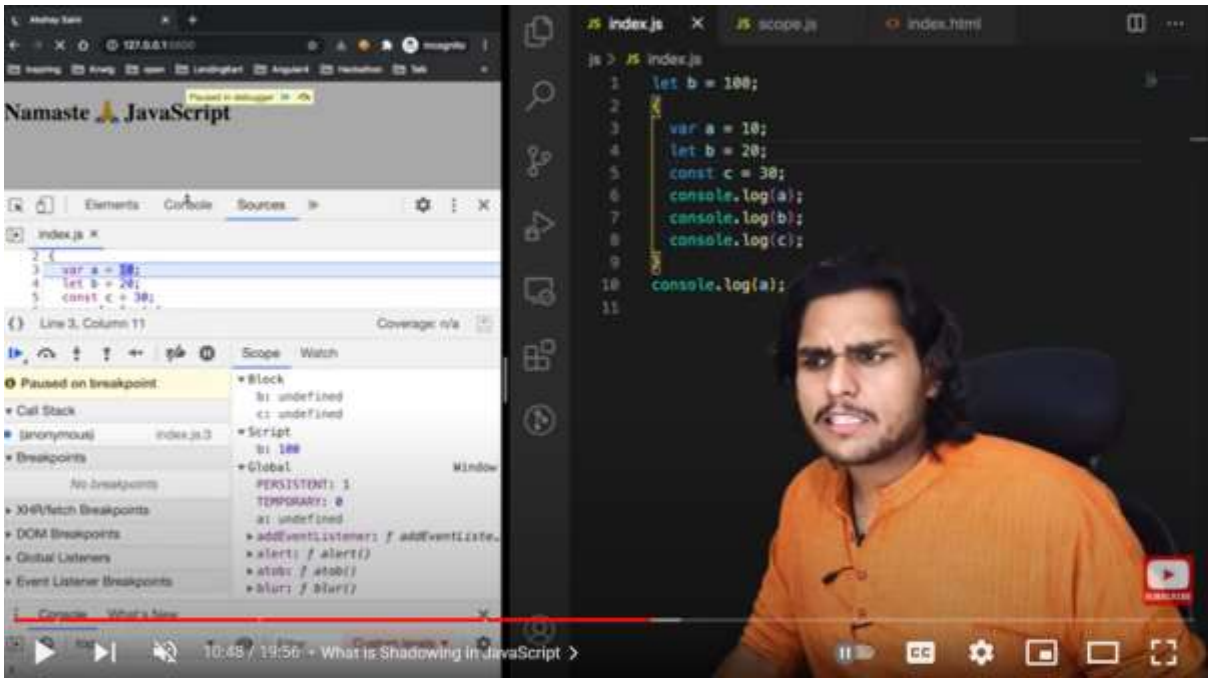
```
Line 4, Column 11
Debugger paused
▾ Call Stack
  (anonymous)        index.js:4
▾ Breakpoints
  index.js:3
    var a = 10;

Scope
▾ Block
    b: undefined
    c: undefined
▾ Global                         Window
    PERSISTENT: 1
    TEMPORARY: 0
    a: 10
  ▸ addEventListener: ƒ addEventListe…
  ▸ alert: ƒ alert()
  ▸ atob: ƒ atob()
  ▸ blur: ƒ blur()
  ▸ btoa: ƒ btoa()
  ▸ caches: CacheStorage {}
```

As line number 1 executes var a = 100;

Line number 3 changes value of a as 10 in the global space even though this stament is inside the block no seprate memoery will be allocated to var a in the block scope.

It modifies the value of a which we call as SHADOWING

100 would be printed both inside and outside block for var a



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

```
Paused on breakpoint
▾Block
   b: undefined
   c: undefined
▾Script
   b: 100
▾Global                Window
   PERSISTENT: 1
   TEMPORARY: 0
   a: undefined
```

Here let b outside block will have a seprate memory space

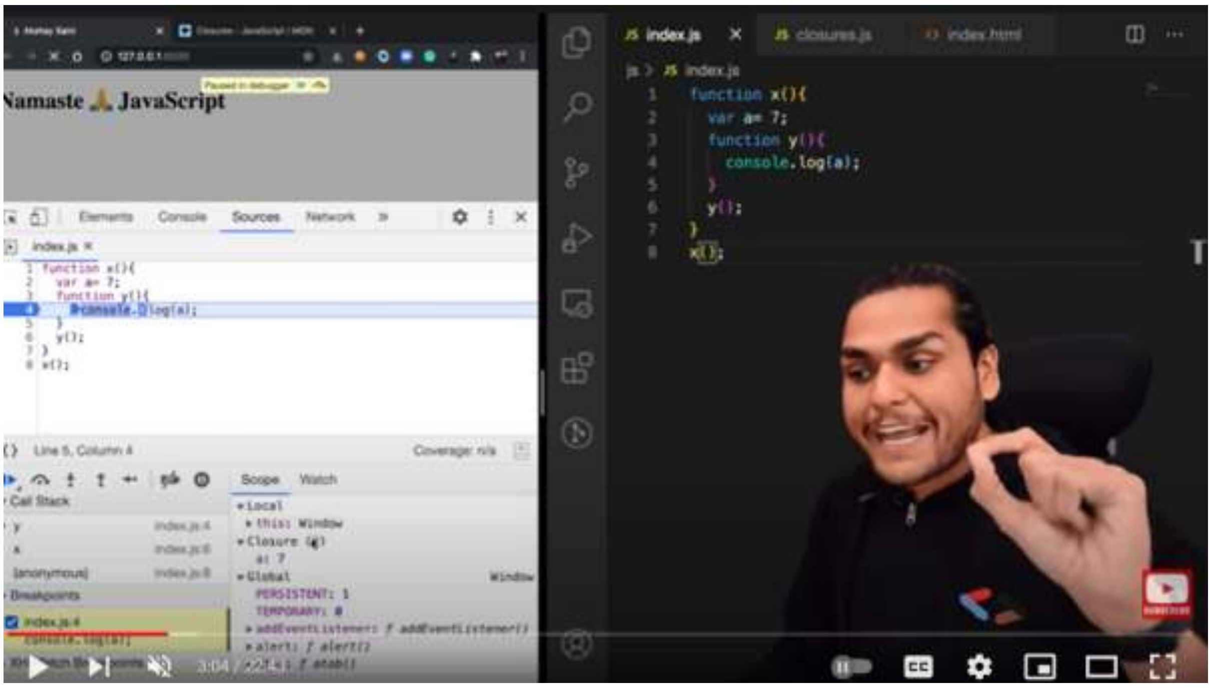
Let b inside the block will have a different memory space

So outside block 100 will be printed inside block 10 would be printed

SHADOWING HAPPENS ONLY WITH VAR DECLARATIONS AND NOT WITH LET

UNIT 9 : CLOSURES

CLOSURE IS FUNCTION BIND TOGETHER WITH IT'S LEXICAL ENVIRONMENT



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

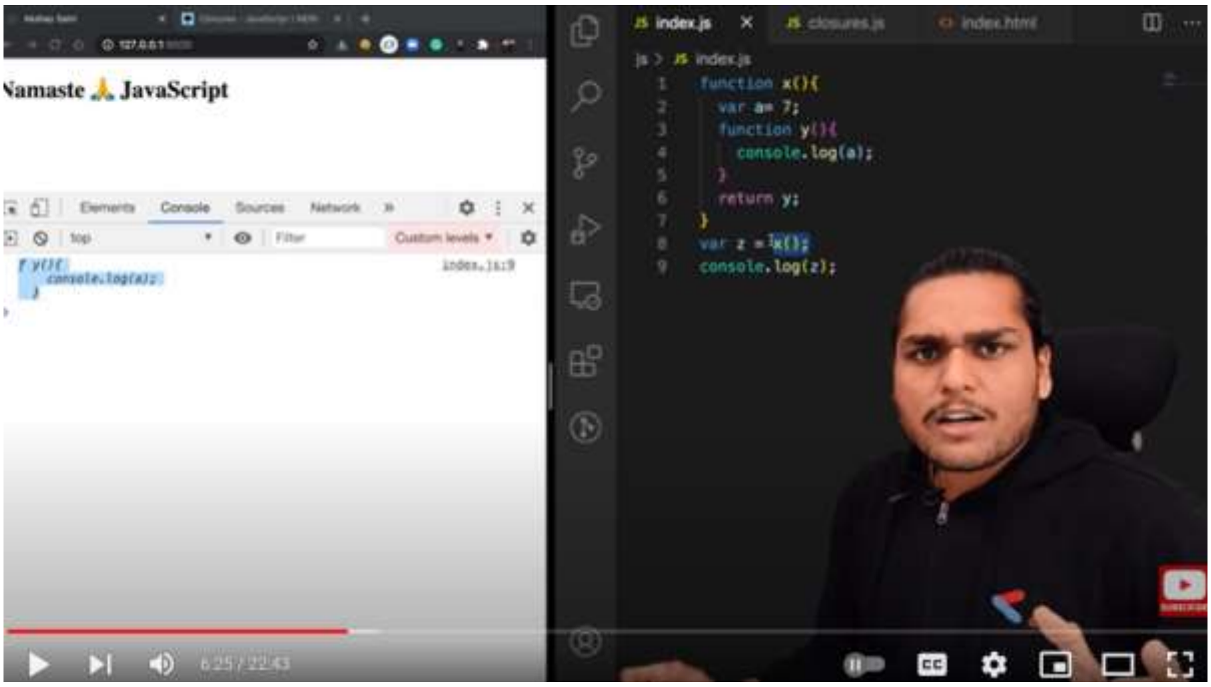
```
▾Local
  ▸this: Window
▾Closure (x)
    a: 7
▾Global                                   Window
    PERSISTENT: 1
    TEMPORARY: 0
  ▸addEventListener: ƒ addEventListener()
  ▸alert: ƒ alert()
```

Closure(x) is created when the control goes to execution of line3.

Closure(x) means function y binded with it's lexical environment means it has access to it's parent's lexical environment also.

FUNCTIONS ARE THE HEART OF JAVASCRIPT, YOU CAN EVEN ASSIGN FUNCTIONS TO A VARIABLE, YOU CAN ALSO PASS THE FUNCTION INSIDE A FUNCTION AS A PARAMETERS, YOU CAN EVEN RETURN THIS FUNCTION.

When you return the function closures come into picture and things get complex



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

```
f y(){
    console.log(a);
  }                                        index.js:9
```

In this script function x() has returned function y() and function y() is stored in variable z.

Once function x() has returned the function y() it vanishes and it's exec is no longer there in the call stack.

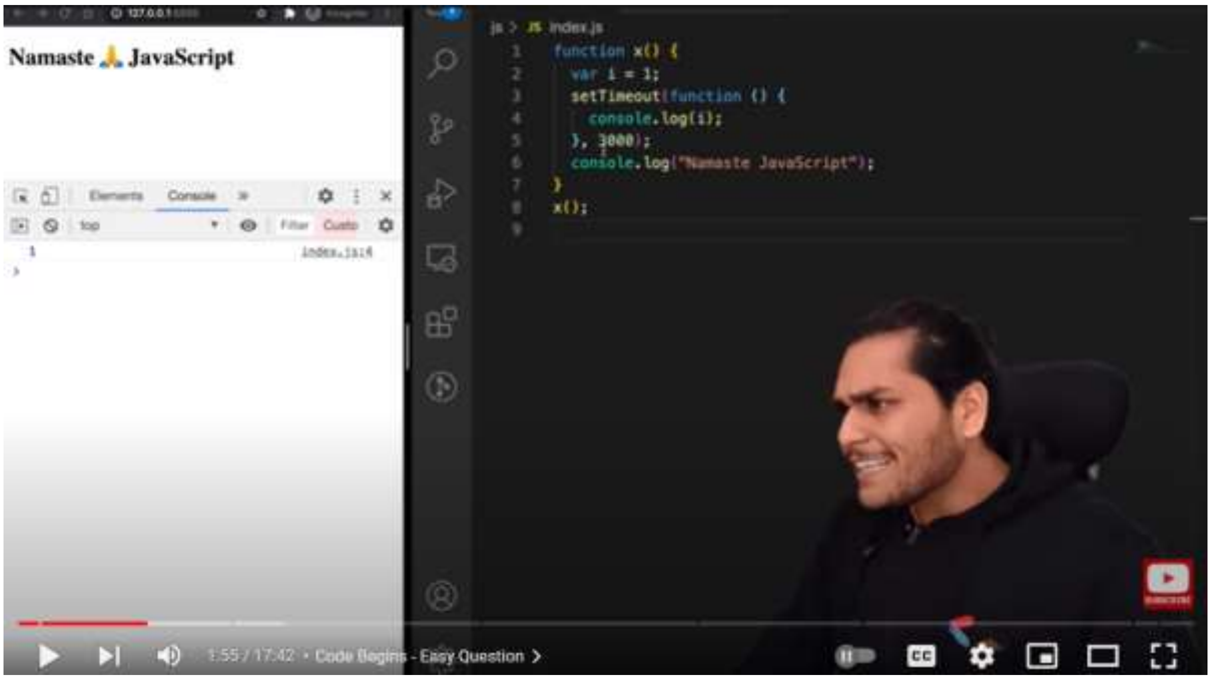
Now our variable z contains function y(). Now we can execute function y() in the global space. (Beauty of JavaScript).

So if z is executed in the global space it will try to access var a, but where is var a it must have vanished because function x()'s exec is poped out from the call stack. So it should give a refernce error right??. But still it prints 7.

Reason :

When you return a function , a closure is returned and not the function. Closure is function binded with it's lexical environment. Hence function y()'s closure was function y() + lexical environment of y that's why it could acces var a even after a is completely erased from memory.

UNIT 10 : setTimeout + closures



```
function x() {
  var i = 1;
  setTimeout(function () {
    console.log(i);
  }, 3000);
  console.log("Namaste JavaScript");
}
x();
```

```
1                                index.js:4
>
```

Here setTimeout works like , it takes a call back function and attaches a timer with it and call it whenever the timer expires.

UNIT 11 : FIRST CLASS FUNCTIONS

1. FUNCTION STATEMENT :

   ```
   function a () {

   }
   ```
   This way of creating a function is called as function statement

2. FUNCTION EXPRESSION :

   ```
   var b = function () {

   }
   ```
   Assingning a function to a variable is called as function expression

   Major difference between this two is hoisting.
   If a is called before it's creation it will execute.
   If b is called before it's creation it will throw a TypeError. Function b is treated as a variable.

3. FUNCTION DECLARATION :

   same as function statement.

4. ANONYMOUS FUNCTION :

   A Function with no name is anonymous function.

   ```
   Function() {

   } //Syntax error : function statement requires a name.

   Var h =  function() {
   ```

```
} // anonymous function.
```

ANONOYMOUS FUNCTION ARE USED WHEN THEY ARE USED AS VALUES OR USED IN FUNCTION EXPRESSIONS.

5. NAMED FUNCTION EXPRESSION :

```
var b = function xyz() {

}

b(); // executed
xyz(); /REFERENCE ERROR;
```

6. PARAMETERS ARE RECEIVED AND ARGUMENTS ARE PASSED

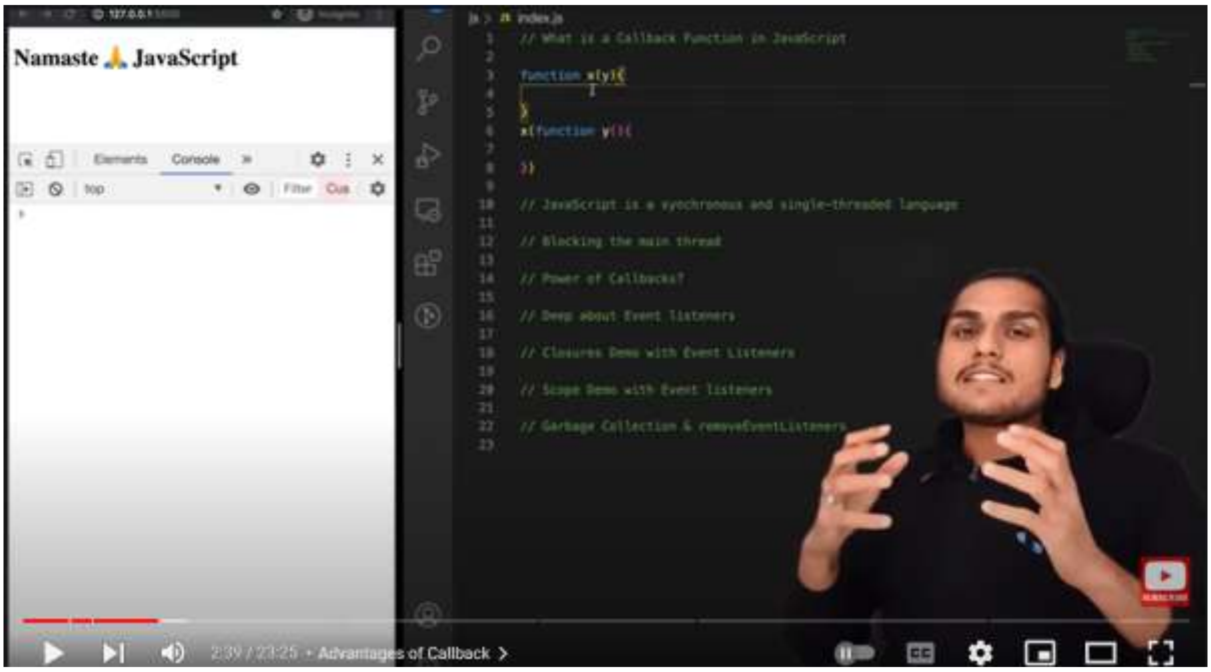
Parameters are local variable in the function.

7. FIRST CLASS FUNCTIONS

The ability to use functions as values and to be passed in a arguments and to be returned from a function is known as FIRST CLASS FUNCTIONS.

Functions are “first class citizens” is same as first class functions.

UNIT 11 : CALLBACK FUNCTIONS AND EVENT LISTENERS.



```
// What is a Callback Function in JavaScript

function x(y){

}
x(function y(){

})

// JavaScript is a synchronous and single-threaded language

// Blocking the main thread

// Power of Callbacks?

// Deep about Event listeners

// Closures Demo with Event Listeners

// Scope Demo with Event listeners

// Garbage Collection & removeEventListeners
```

When you pass a function as argument it's called as call-back function.

Fy() over here is call-back function.

e.g. while using set timeout we use a call-back function. Now it's upto setTimout when it uses that particular function or calls it.

BLOCKING THE MAIN THREAD :

Each and every function executes through callstack in JavaScript. Callstack is also known as main thread. If any operation blocks the callstack we say it as it blocked the main thread. Suppose we have a function that takes 20-30 mins to execute it will block our main thread. That's why we say that we should never block our main-thread. This kind of operations should be performed using asynchronous programming only.

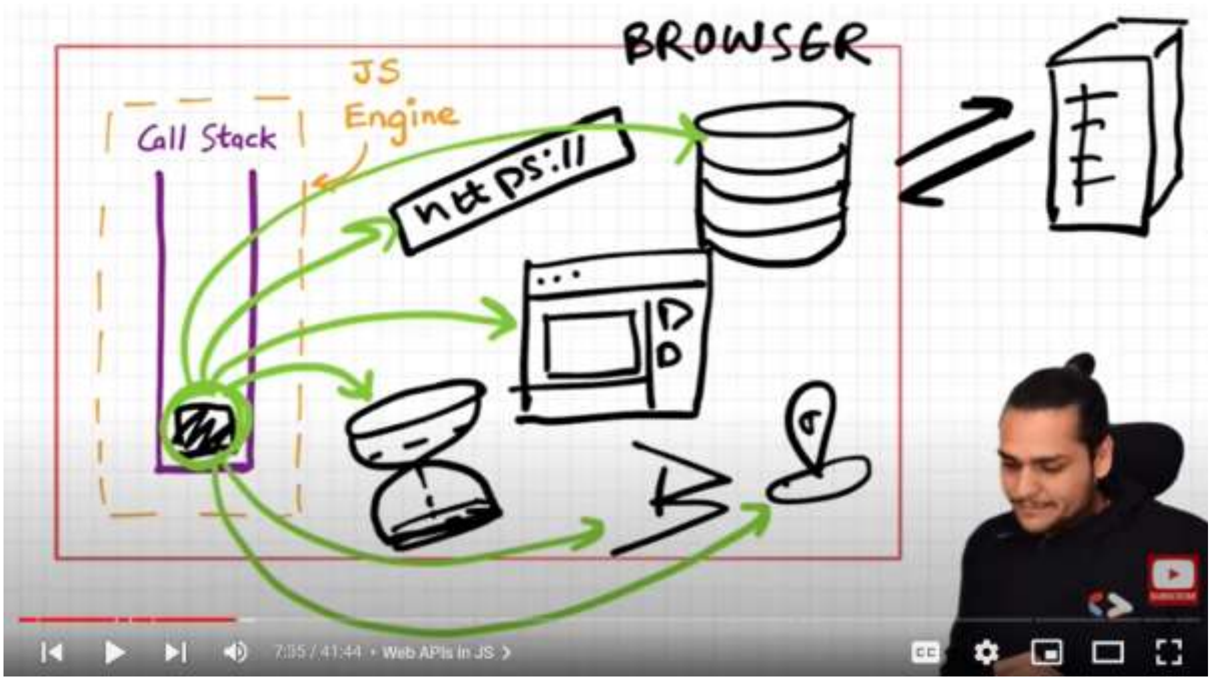
UNIT 12 : ASYNCHRONOUS JAVASCRIPT AND EVENT LOOP

" Time, Tide and JavaScript waits for none. "

Every function which is at the top of callstack would be immediately executed without any delay.

So what if you want to execute some part of script after 5mins or 10 mins or 15 mins.

We would require power of timers in JavaScript

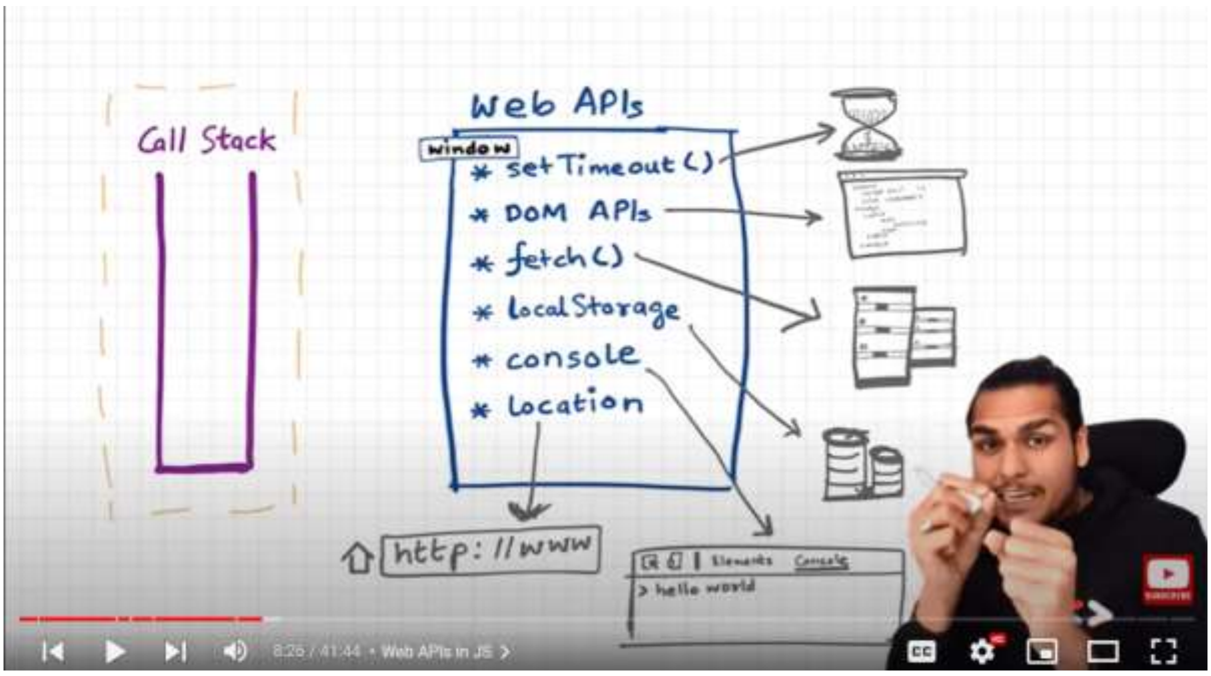


"Brower is the most remarkable creation in the history of mankind 😋"

Call Stack is inside JavaScript Engine and JavaScript Engine is inside the browser. Brower does a lots of tasks like.

1. Have Session and Local Storages.
2. You can also have a timer inside the browser.
3. Browser also has a superpower to communicate to outside world (e.g. Netflix Servers).
4. Bluetooth access, Geolocation access.

Now , JS Engine needs a way to access those superpowers of browser. The way is WEB API's.

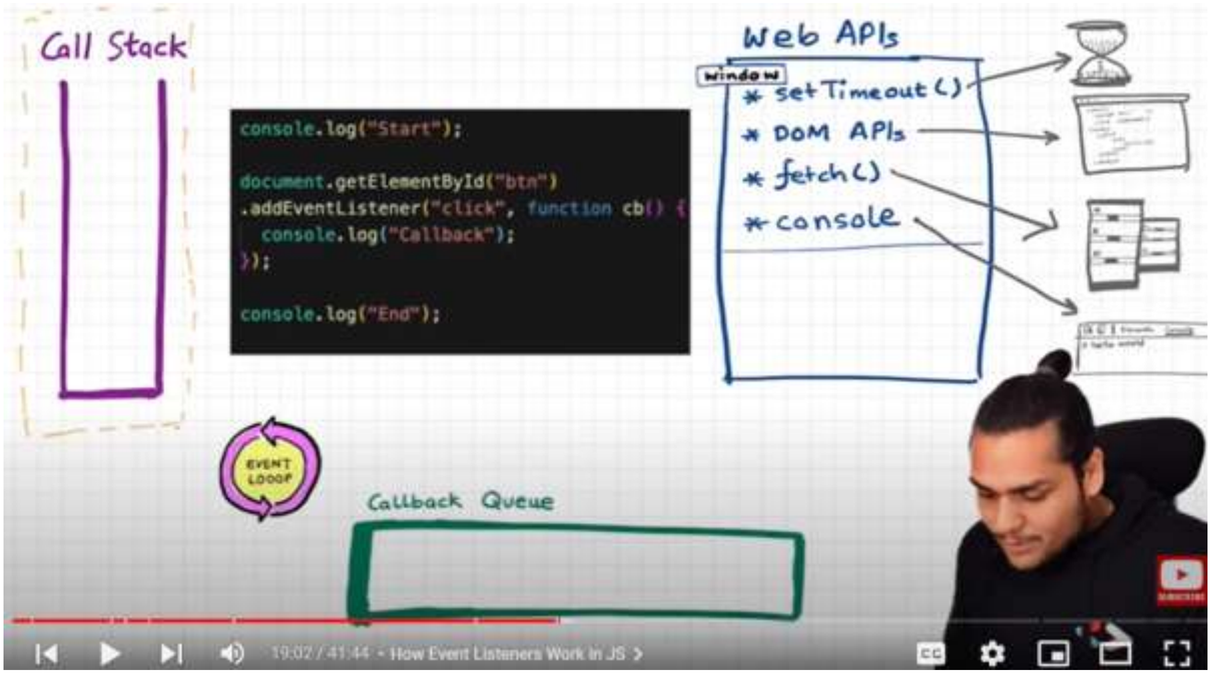


setTimeout, DOM API's, fetch(), local storage, console, Location this are all not a part of JavaScript. This all are WEB API's.

This all are part of Browsers.

So if you want to use this powers of browser in you JS code. You have to access them using Window keyword.

Concept of Callback Queue and Event Loop.



```
console.log("Start");

document.getElementById("btn")
.addEventListener("click", function cb() {
  console.log("Callback");
});

console.log("End");
```

As we know there is a condition attached with callback functions, and whenever the condition become true the callback function is pushed inside the callstack.

But it's not pushed directly. Its first pushed inside the callback queue and then its pushed inside the callstack

Event loop checks the callback queue and whatever function is in the front of the Queue it pushes them in the callstack.

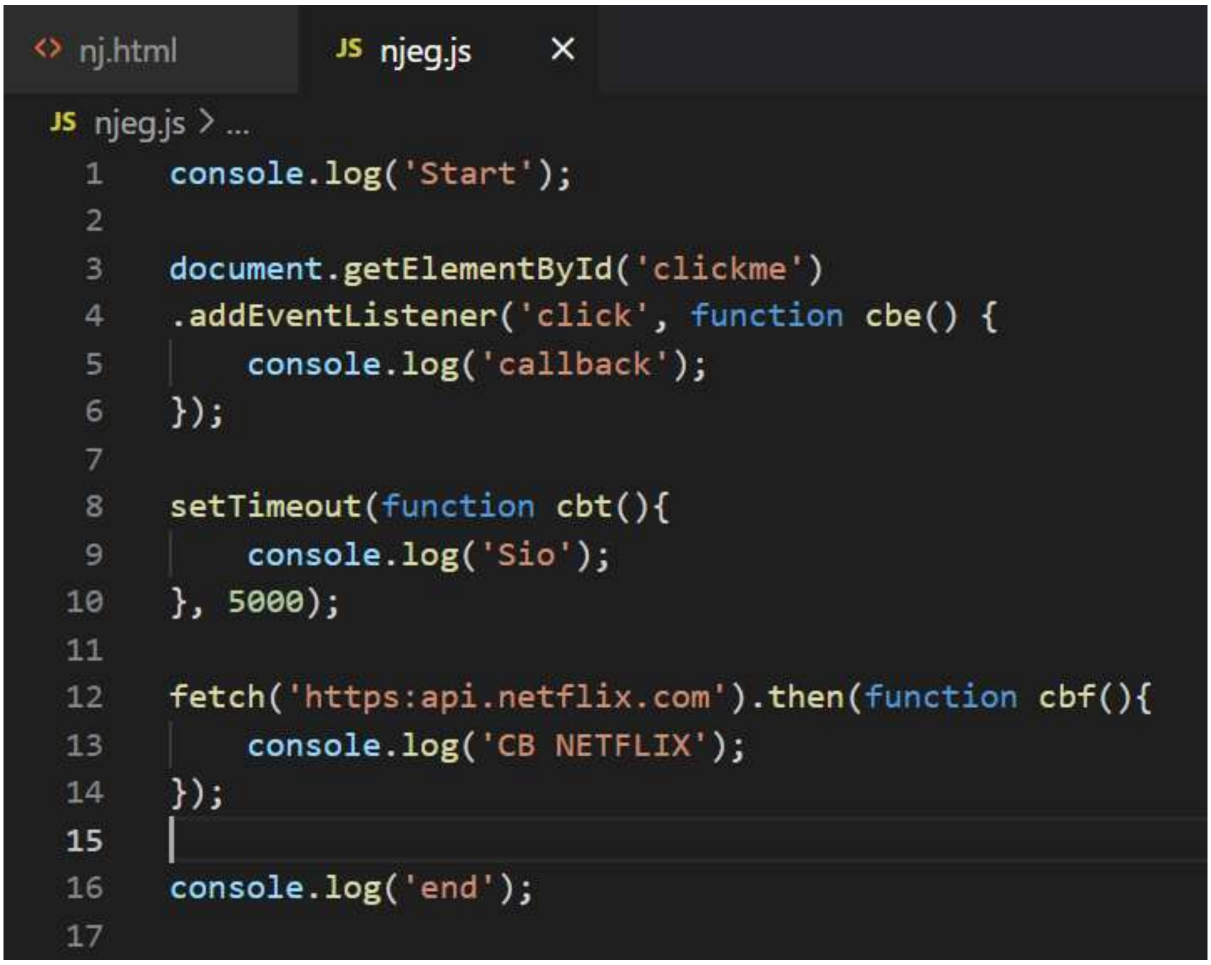
How fetch works?

Fetch is a web api which is used to make network connections.

Fetch basically goes and request api calls.

Fetch returns a promise and we have to pass a callback function once the

the promise is resolved we have to execute the callback function.



```
console.log('Start');

document.getElementById('clickme')
.addEventListener('click', function cbe() {
    console.log('callback');
});

setTimeout(function cbt(){
    console.log('Sio');
}, 5000);

fetch('https:api.netflix.com').then(function cbf(){
    console.log('CB NETFLIX');
});

console.log('end');

```

Now we have this three functions queued.

1) Cbe function is waiting for the button to be clicked
2) Cbt function is waiting for the timer to get expired
3) Cbf function is waiting for the data to be returned

Concept of microtask queue

Microtask queue has a higher priority. The queue is similar to callback queue but the queue has higher priority.

This function cbf which was a callback function in case if promises the callback function goes into microtask queue.

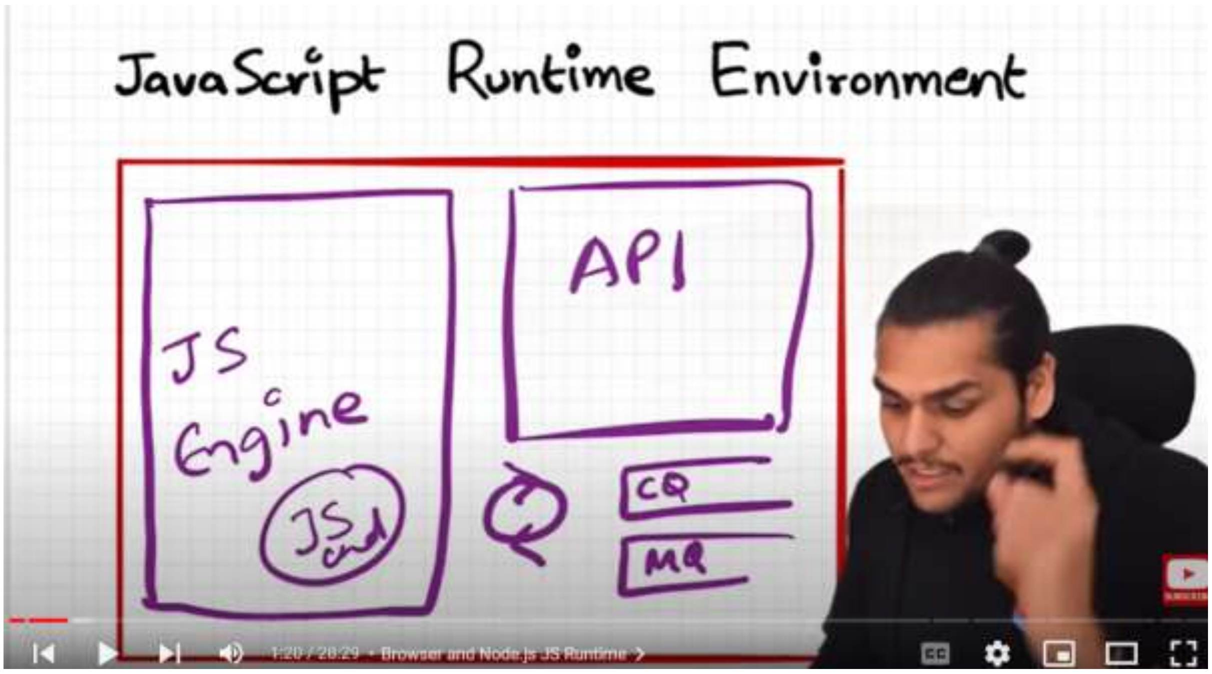
Now after a while the timer expires and cbt function is pushed inside the callback queue. Now after a while the button is clicked and so cbe is pushed inside the callback queue.

UNIT 13. JAVASCRIPT ENGINE

“JavaScript is everywhere”

JavaScript can run inside a browser, inside a server, inside your smart-watch, it can also run inside a light bulb, inside a robot.

Its all possible because of “JavaScript Runtime Environment” .


JavaScript Runtime Environment
JS Engine
JS code
API
CQ
MQ
1:20 / 28:29 • Browser and Node.js JS Runtime

JavaScript Runtime Environment is having all the things required to run a JS code. It consists of JavaScript Engine, set of web api,callback queue, event loop, microtask queue and many more things.

JavaScript Runtime Environment isn’t possible without JRE.

Every Browser has a JRE.

Node JS it is an open source JavaScript runtime. Means it has everything which is required to run a JS code and nodejs could run JS anywhere.

JavaScript could also be run inside a water cooler. It may have different api's over there for e.g getWaterLevel, setWaterLevel etc as in case of browser we have console

Every Browser has it's own JS Engine.

Chakra is a JS Engine of MS Edge.

V8 is a JS Engine of Google Chrome.

SpiderMonkey is JS Engine of Mozilla Firefox.

You can even follow EcmaScript Protocols and create your own JS Engine.

What was the first JS Engine developed?

The first JS Engine was created by the creator of JS itself. Brendan Eich was the creator of JavaScript and also the creator of first JS Engine which is known as SpiderMonkey.

JAVASCRIPT ENGINE

JavaScript Engine is like a normal program which is written in any low level language.

For e.g. Chrome's V8 is written in C++.

JavaScript Engine just takes the JS code and converts it into machine level instructions.

JAVASCRIPT ENGINE ARCHITECTURE

JavaScript Engine takes the code which we write and then it makes the code pass through three phases

1. Parsing
2. Compilation
3. Execution

Parsing

Code is broken down into tokens for e.g let q = 90; (let is a token q is another token ,= is a token)

Syntax Parser converts code into AST Abstract Syntax Tree. (JSON)

COMPILATION

AST is passed to compilation phase.

Interpreter : It's executes the code line by line. Faster than Compiler

Compiler : Whole code is compiled first before executing. An optimized version of code is created and then it's executed. Efficent than Interpreter.

"JavaScript is both compiled and interpreted"

JIT COMPILATION (Just In time compilation) :

EXECUTION

Execution takes the compiled code and executes it.

Execution isn't possible without call stack and memory heap.

Garbage Collector in JavaScript.

Garbage collector frees up the space whenever possible whenever the function is not being used or we clear the timeout. So basically it collets all the garbage and sweeps it out.

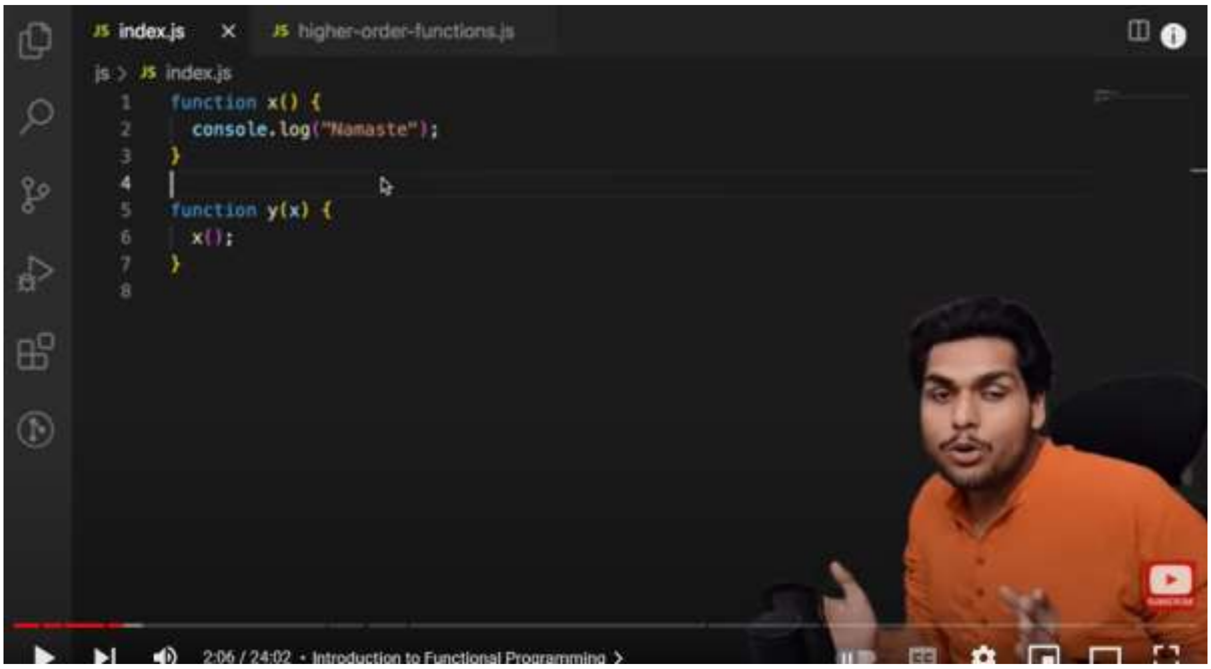
MARK AND SWEEP ALGORITHM is used in JS Garbage Collection.

GOOGLE’S V8 ENGINE IS FASTEST JAVASCRIPT ENGINE.

IGNITION IS THE INTERPRETER AND TURBO FAN IS THE OPTIMIZING COMPILER.

UNIT 14 : HIGHER ORDER FUNCTIONS

A function which takes another function as an argument or returns another function is a Higher Order Function In JavaScript.



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

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

index.js
higher-order-functions.js
js > index.js
2:06 / 24:02 • Introduction to Functional Programming

Here fx is a callback function and fy is a higher order function.

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

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

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

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

```
const calculate = function (radii, logic) {

    let output = [];

    for (let i = 0; i < radii.length; i++) {
        output.push(logic(radii[i]));
    }

    return output;
}

console.log(calculate(radii, area));
console.log(calculate(radii, perimeter));
console.log(calculate(radii, diameter));
```

EXAMPLE OF FUNCTIONAL PROGRAMMING .