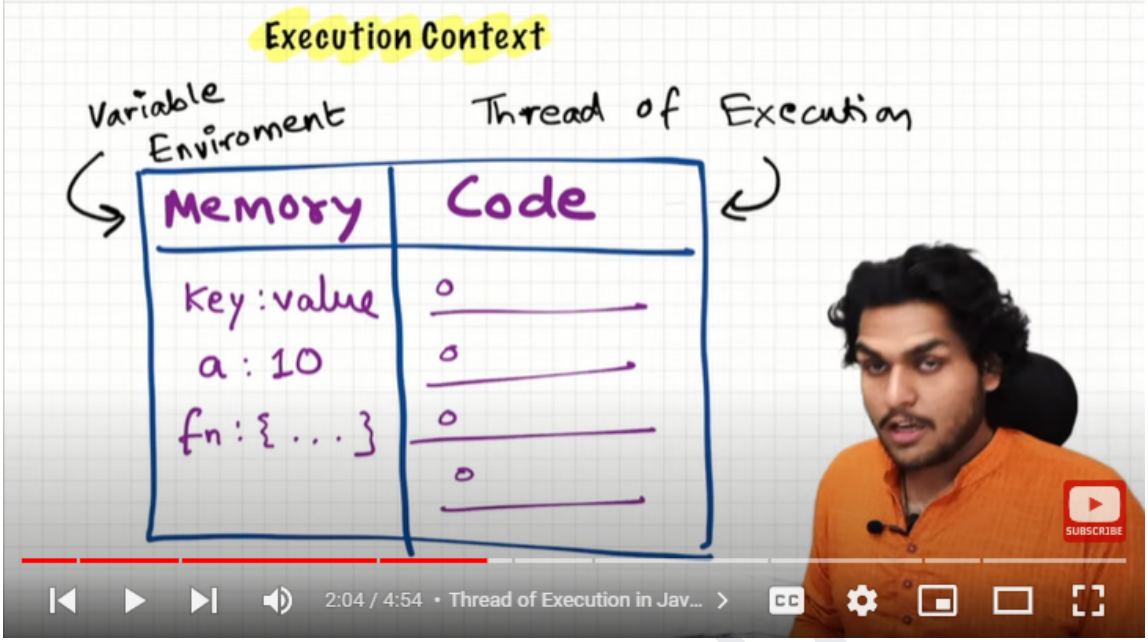
In the Namaste Javascript web series we will learn concepts like How Javascript Works ? Is javascript single threaded or multi-threaded ? Is javascript synchronous or Asynchronous ?

Lecture 1 :- How JavaScript Works 🔥& Execution Context

- Core fundamental concept 😀 :- Everything in javascript happens inside an **execution context.**

- We can assume this execution context as a big box or a container in which whole javascript code is executed.
- Execution context is like a big box that contain two components 1) Memory component 2) Code component
- **Memory component** is the place where all the variables and functions are stored as the key value pair. Memory component is also known as a **variable environment**.
- **Code component** is the place where the javascript code is executed one line at a time. There is one heavy word for this code component "**Thread of execution**".



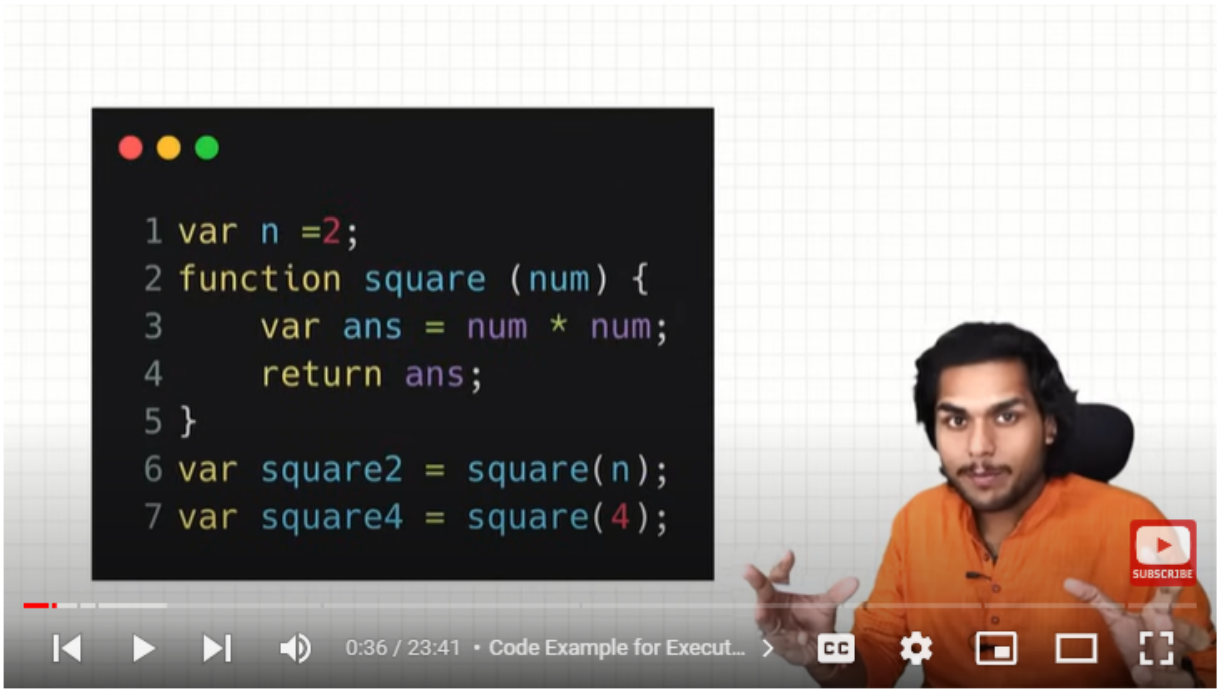
Another core fundamental concept 😀:- Javascript is synchronous single-threaded language

- **Single-threaded** means javascript can execute one command at a time.
- **synchronous** means javascript can execute one command at a time and that too in a specific order. It means javascript will go to the next line only when it is done executing the current line.

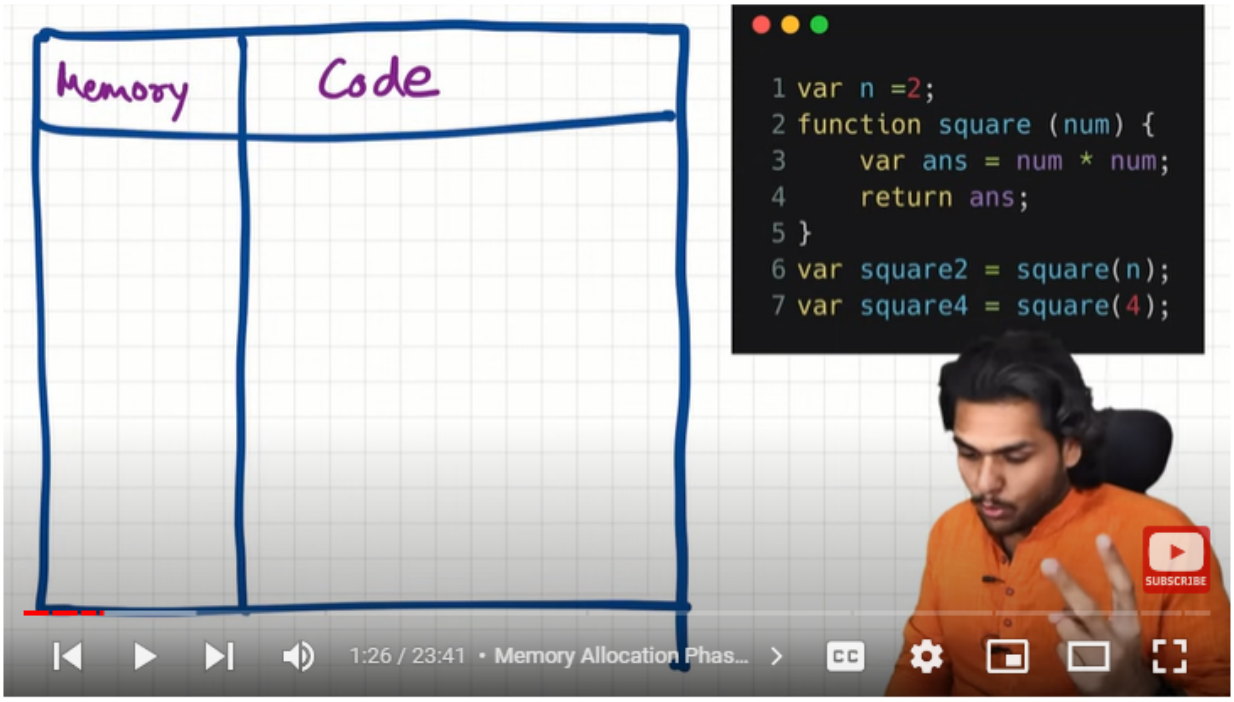
Lecture 2:- How JavaScript Code is executed? ❤️& Call Stack

What happens when you run Javascript code ?

- There are a lot of things happening behind the scenes inside the javascript engine when we run a javascript code.
- When we run a javascript program an **EXECUTION CONTEXT** is created inside the javascript engine. Let's understand how this execution context is created with an example.



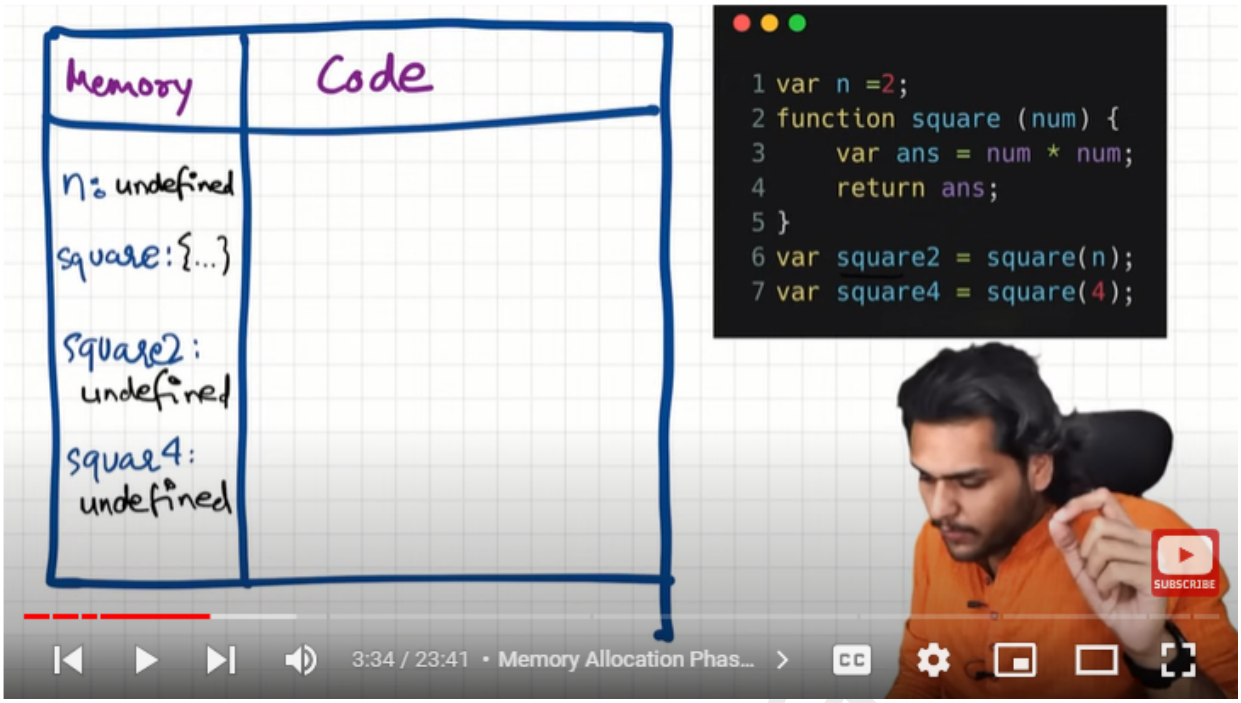
- When we run the above program a **GLOBAL EXECUTION CONTEXT** is created with a **memory component** and a **code component**



- This execution context is created in two phases 1) **Memory creation phase** and 2) **Code execution phase**.

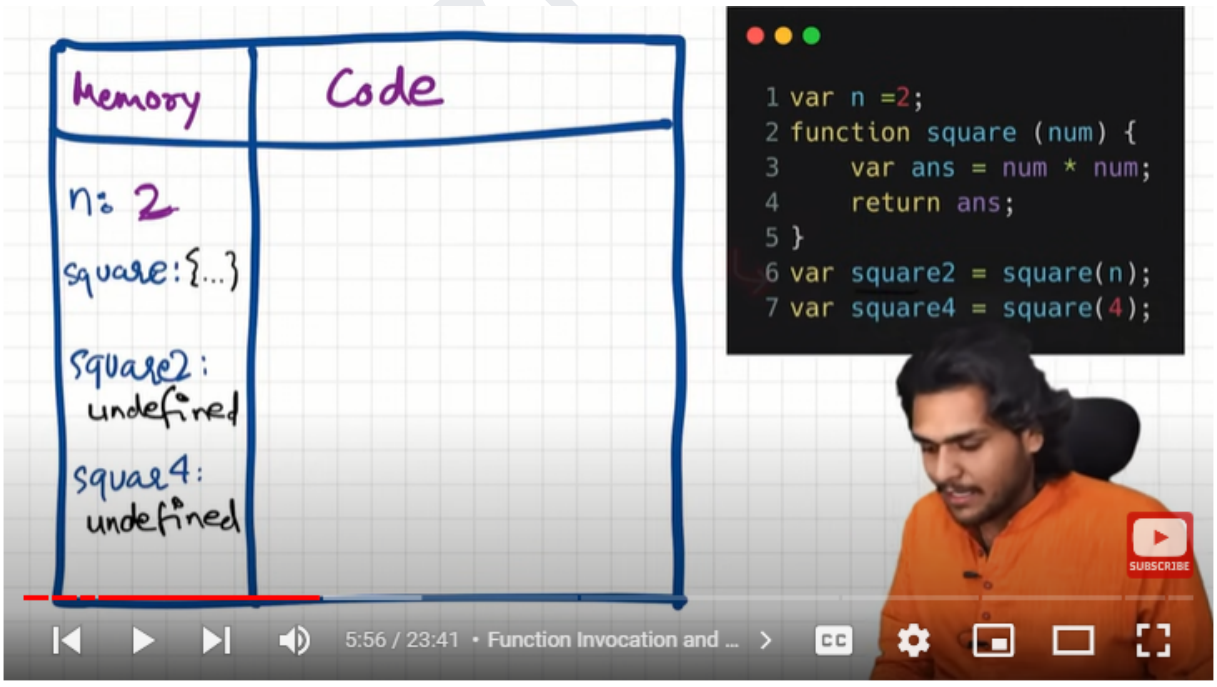
- During the **Memory creation** phase Javascript goes through the entire program and allocates memory to all the variables and functions available inside the program. It stores the variables and functions as the key value pairs. During the memory creation phase, javascript stores a special keyword **“undefined”** to the variables and to function it literally stores the entire code as the value of the function.

-

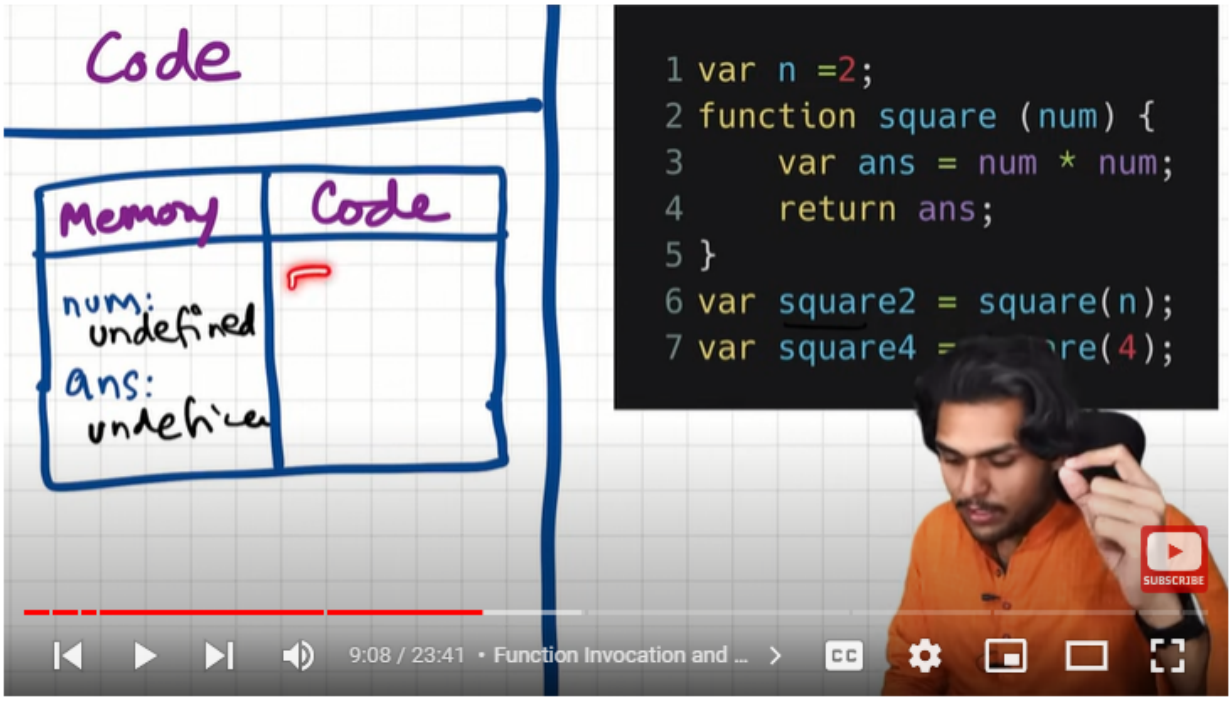


- During the code execution phase javascript again runs through the entire program line by line and executes the code. This is the phase where every calculation takes place
- As soon as it encounters the line 1 during the code execution phase it stores the value 2 inside the identifier n in the memory component.

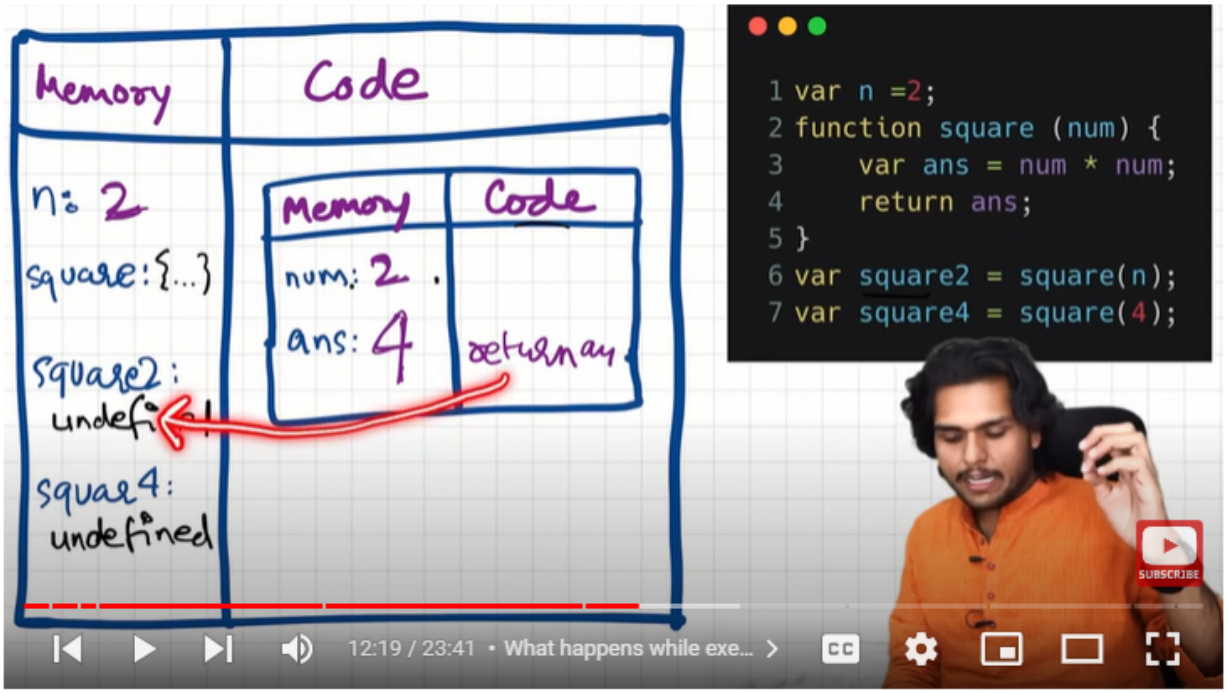
-



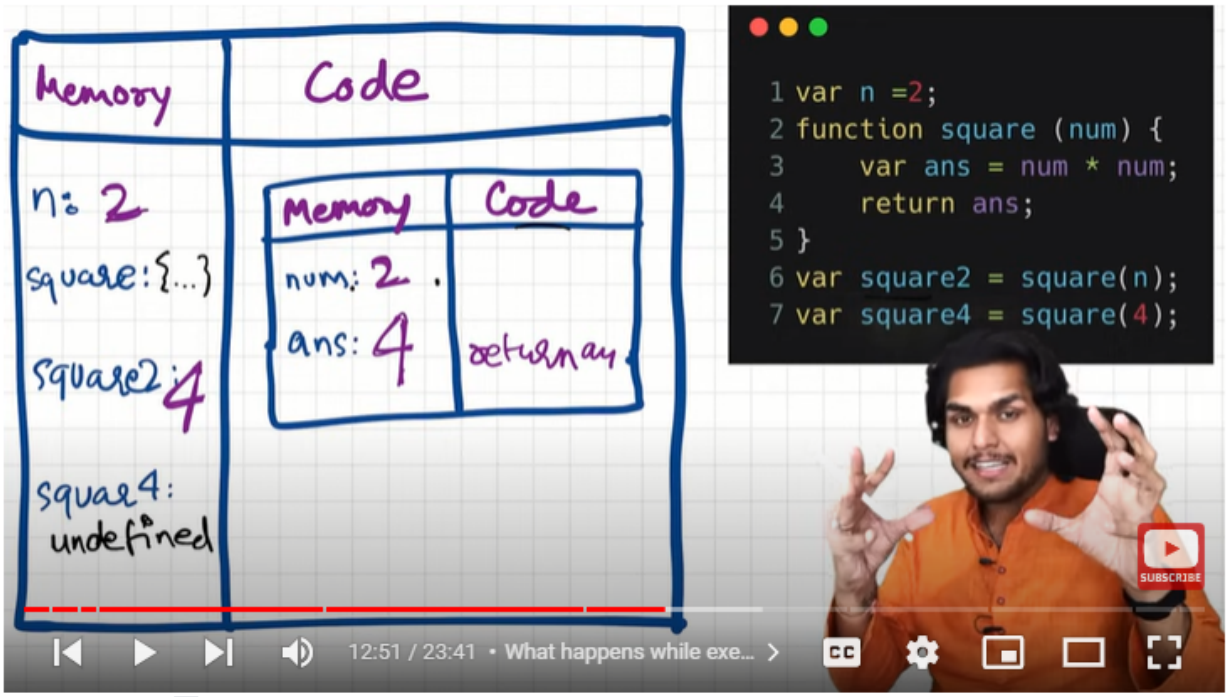
- Now from line 2 to line 5 there is nothing to execute literally so it will move to line 6
- At line 6 there is a function call and whenever a function is called in the javascript program a new **execution context** is created and this execution context contains same 2 component 1)Memory component and 2) Code component
- Now, again we will go through the creation of this execution context and again there are two phases involved in it 1) Memory creation phase and 2) Code Execution phase. Now as there is a function invocation so we will be concerned about the piece of code inside the function
- In **Memory creation** phase again memory is allocated to the variables and the functions and a special value “undefined” is stored inside the variables. We also allocate memory to the parameters passed to the function during the memory creation phase.



- During the code execution javascript runs through the entire program line by line and executes the code and stores the value inside the variables in the memory component after calculating the values.
- During the **execution phase** when javascript encounters the **return ans;** statement on the line 4 inside the function. It tells the function that your work is over now and now you can return the whole control to the execution context from where the function was invoked and the value of ans which is calculated 4 will be stored inside square2 variable in the memory component of **global execution context**.

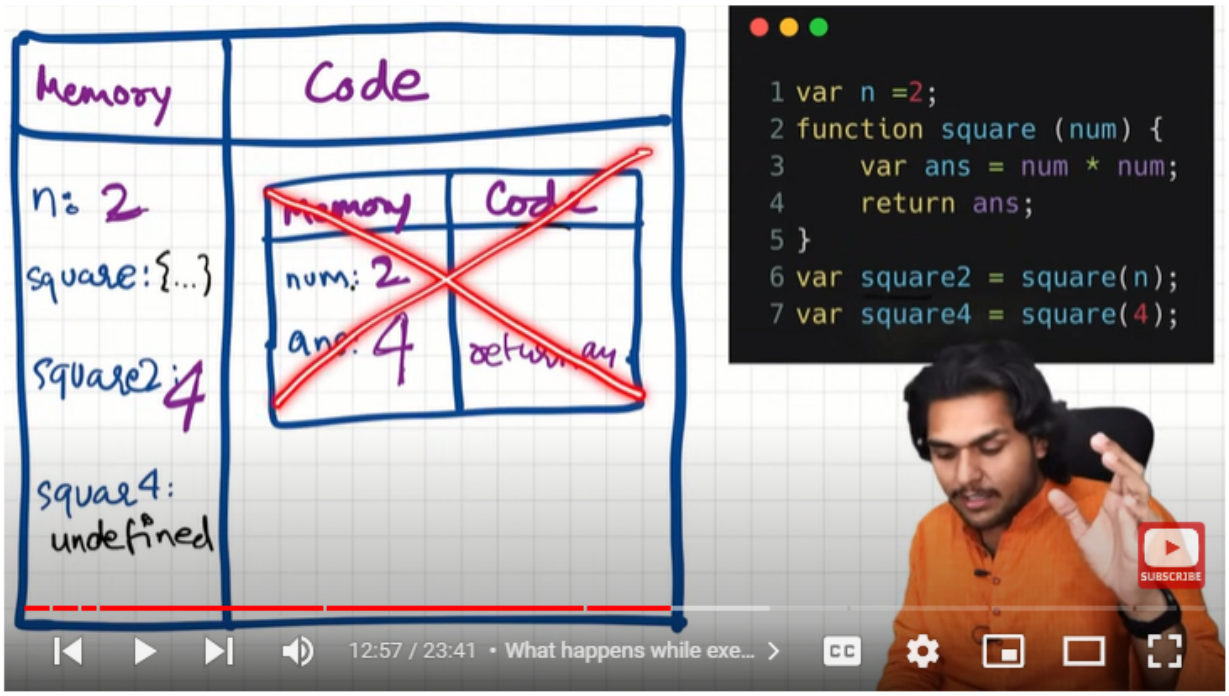


-

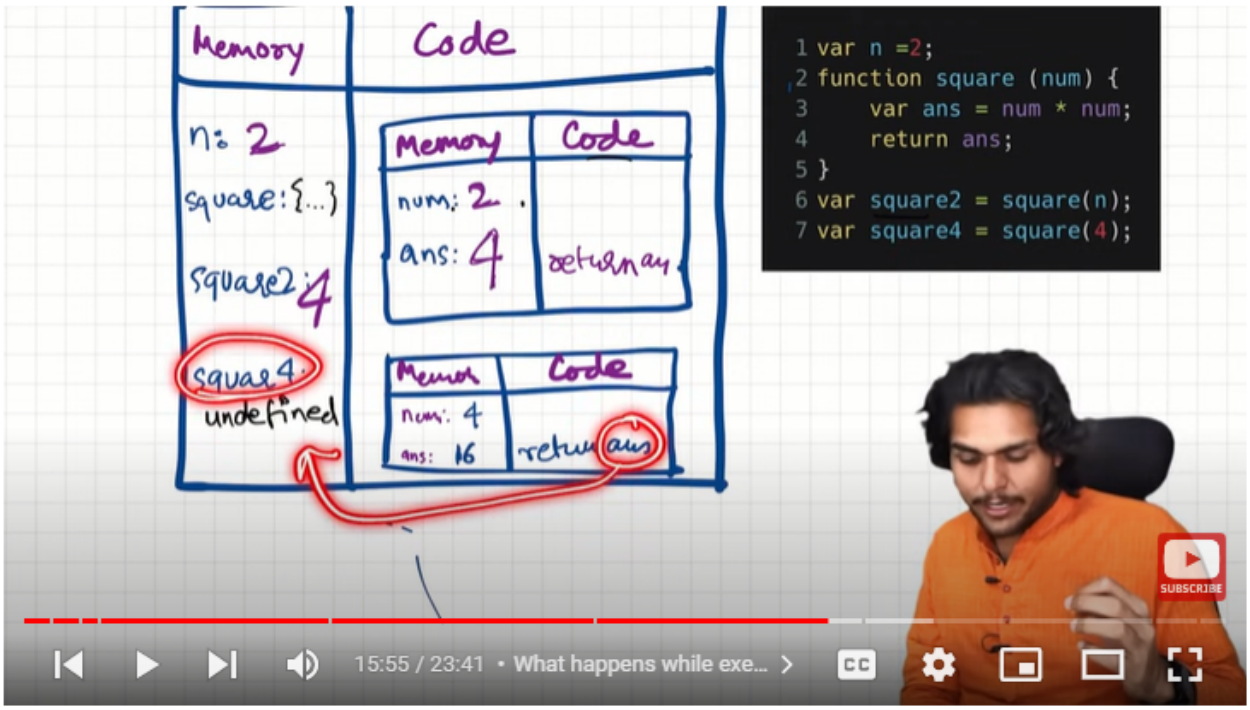


-

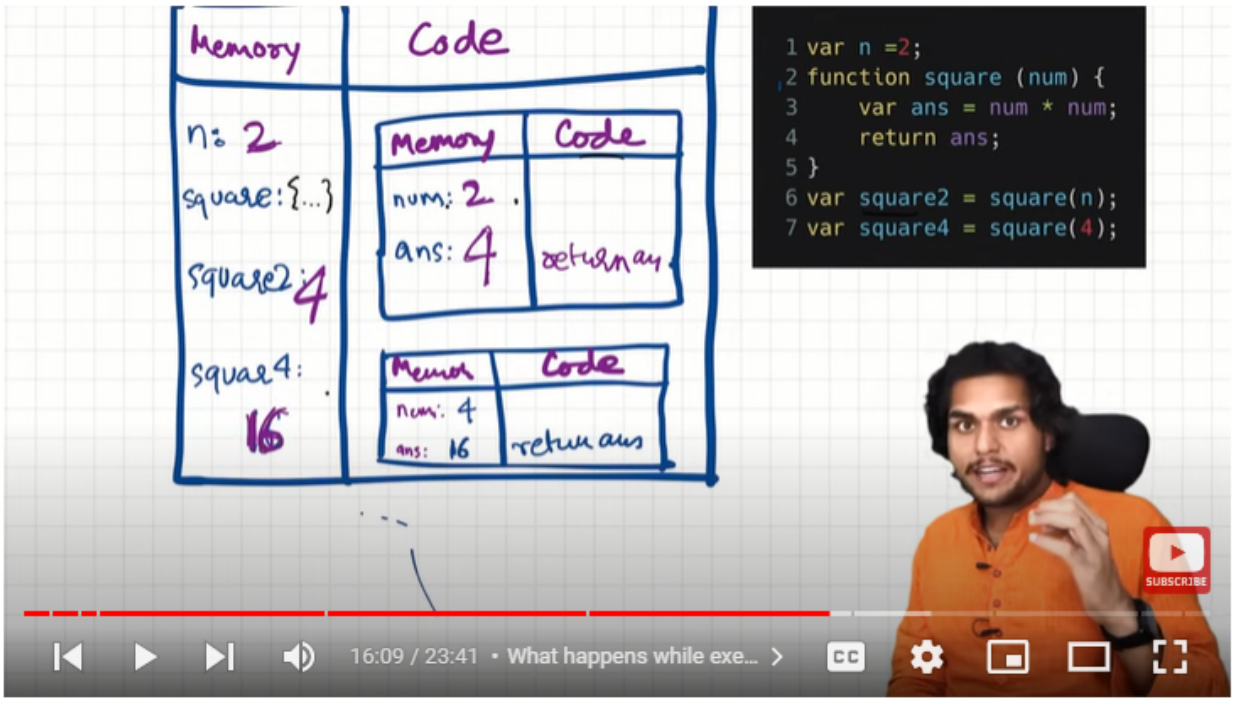
- One more thing that happens after the function is executed is that the execution context which is created due to the function call will be deleted.



- Now, we will move to line 7 and again the same thing happens because on line 7 also there is a function invocation. So, again a execution context is created and again the same thing happens which we have already seen during the execution of line 6.

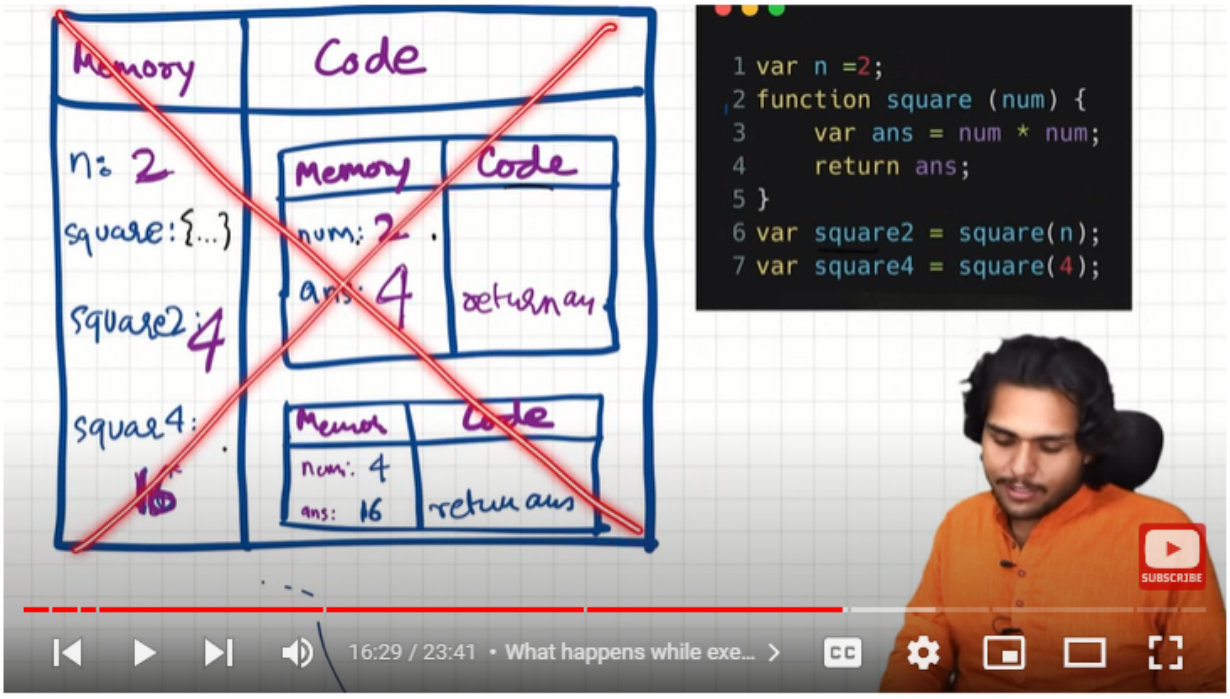


-



- Now, as the line 7 is also executed. Therefore javascript is done executing the whole program so now the whole global execution context will be deleted.

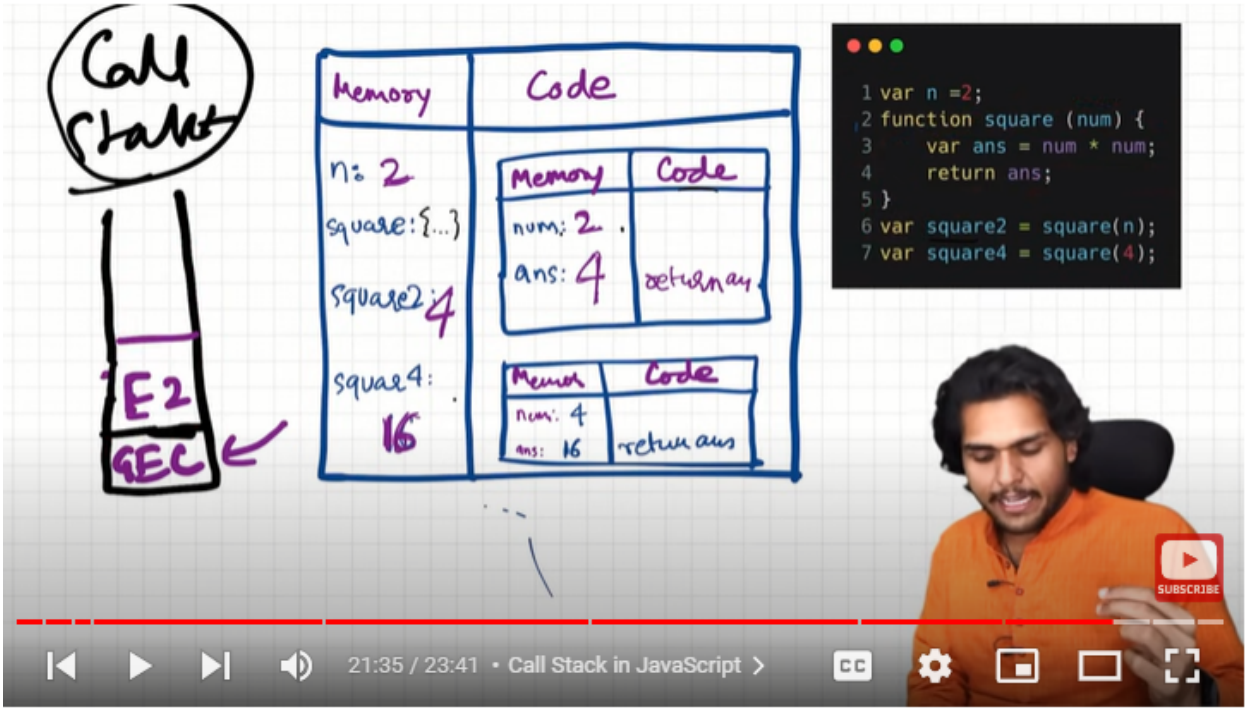
-



- This is how an entire program is executed in the javascript

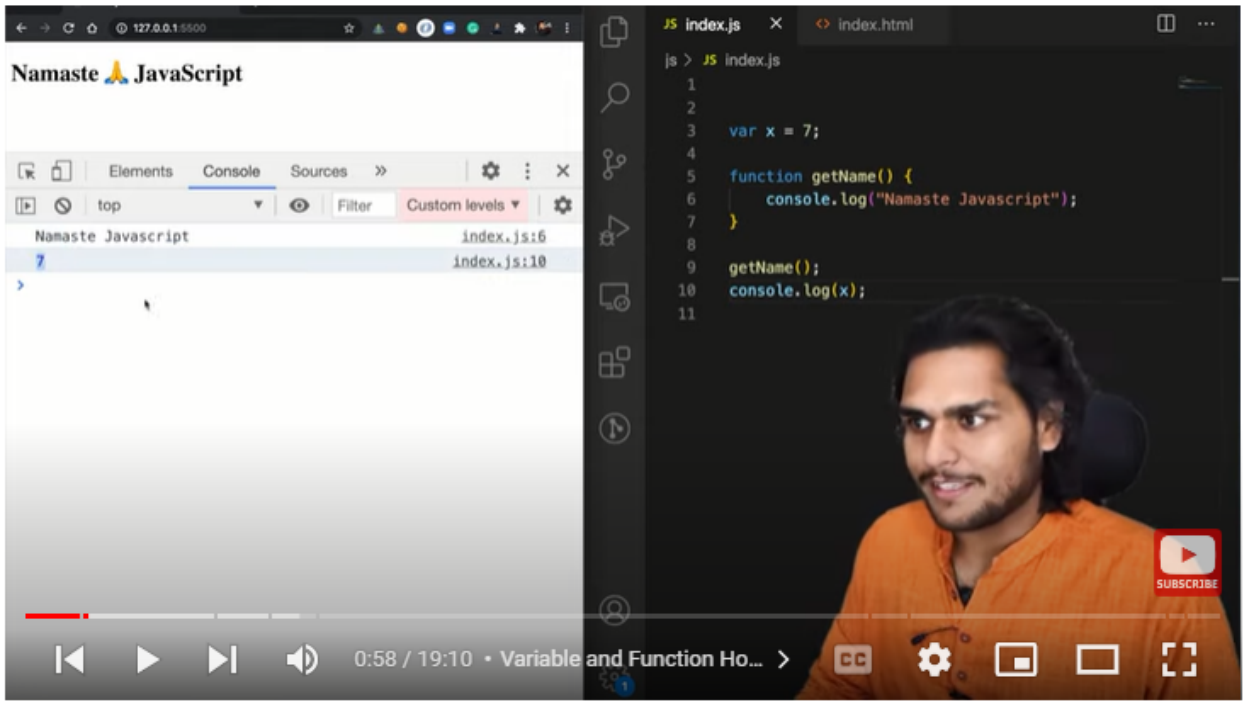
CALL STACK

- **Call stack** is for managing these **execution contexts**. So whenever an execution context is created it will be pushed to the stack and when the execution context is deleted it will be pushed out of the stack. So with the help of **CALL STACK** javascript is able to manage the deletion and the creation of these **execution contexts.**

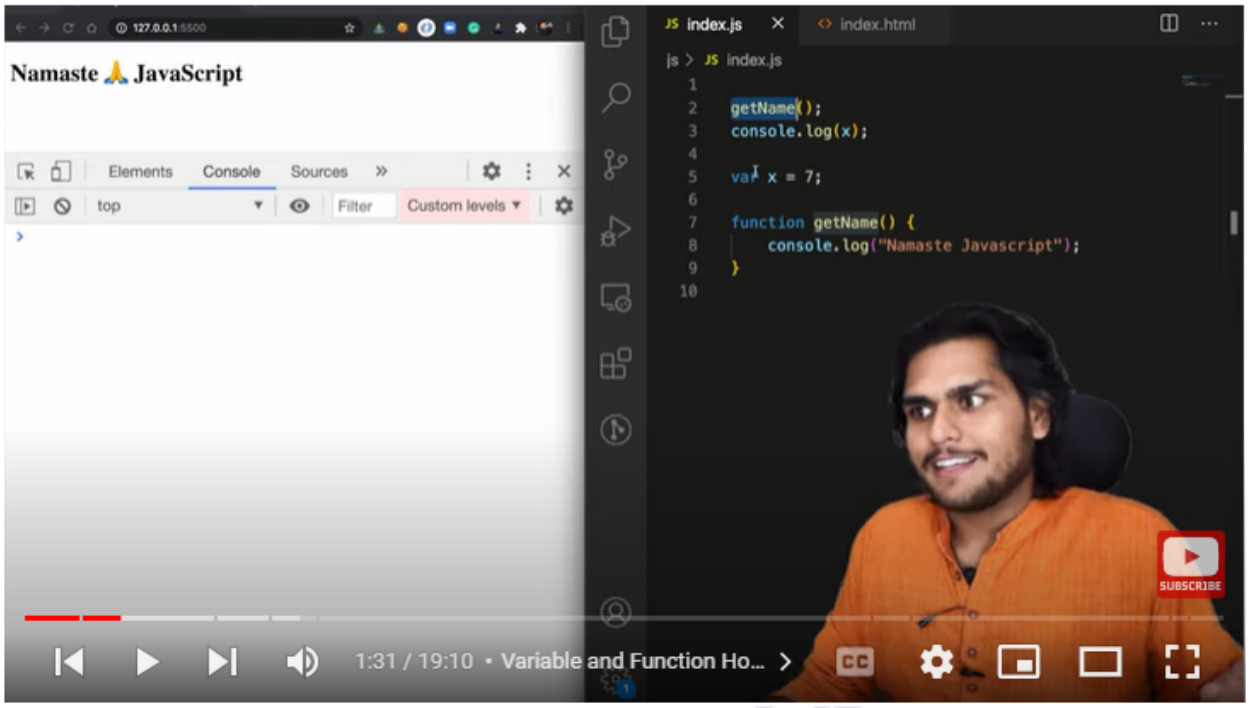


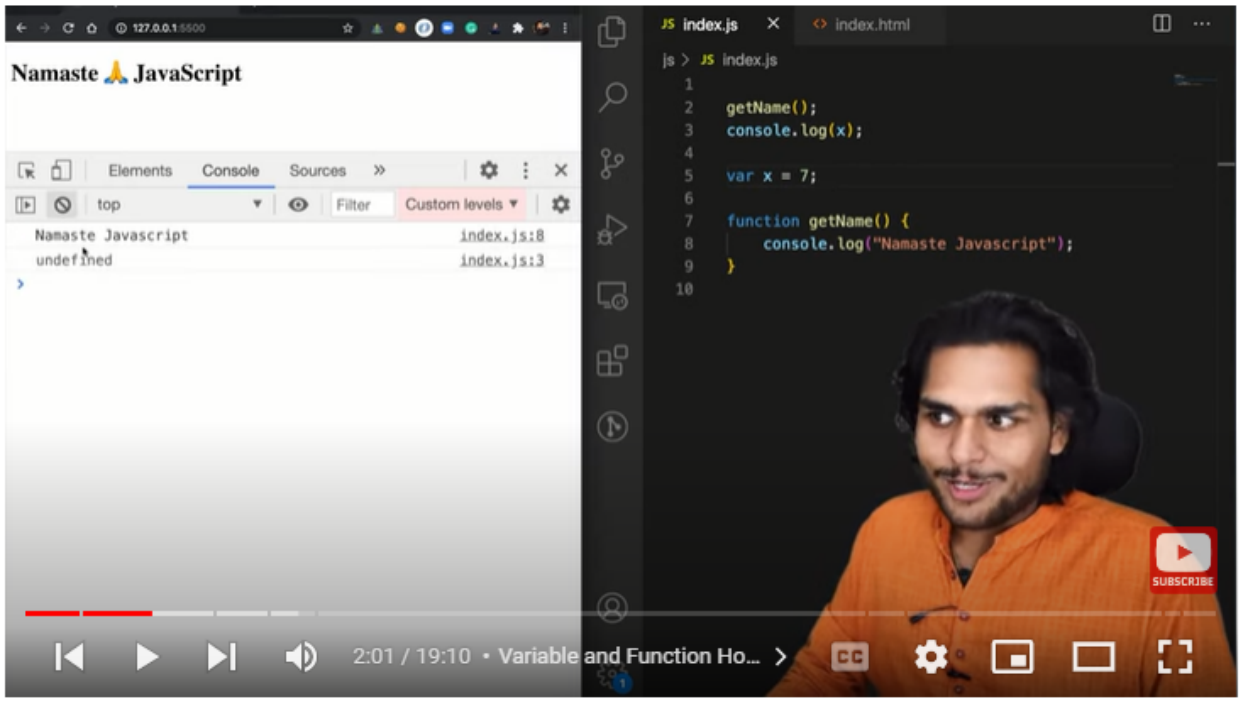
-
- **CALL STACK maintains the order of execution of the execution contexts.**
- **CALL STACK is also known by few other names such as Execution context stack, Program Stack, Control stack, Runtime stack, Machine stack**

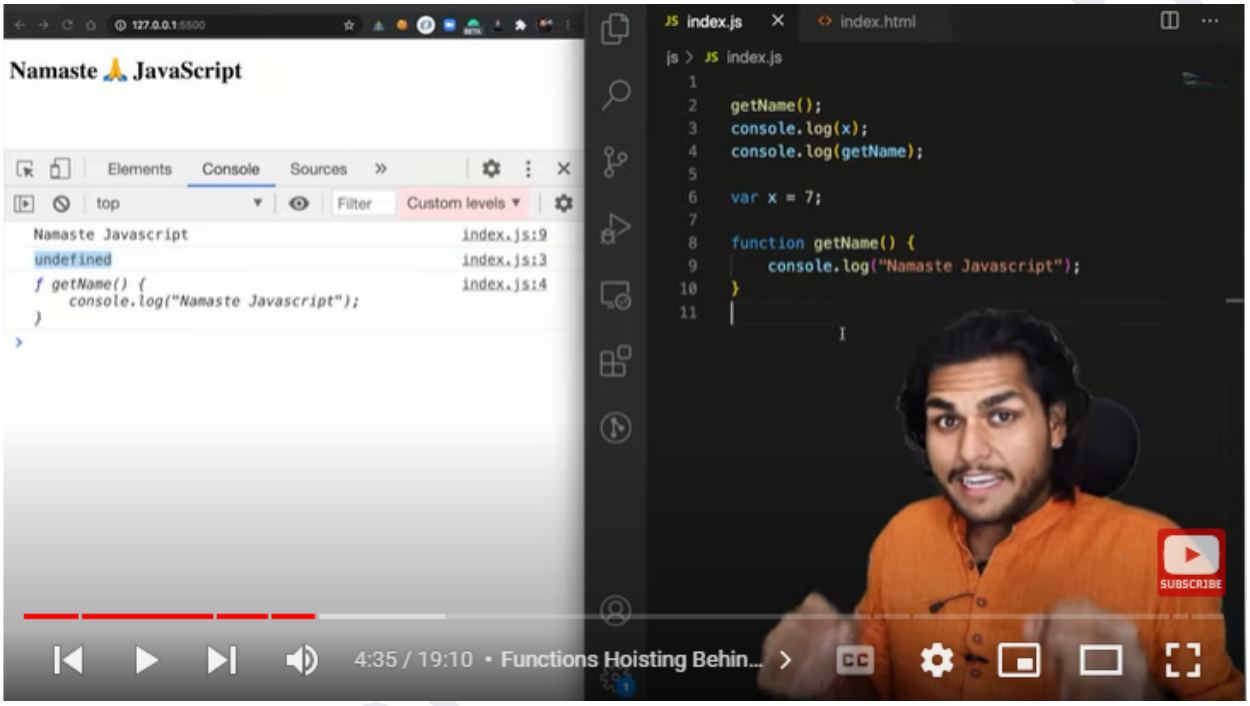
Lecture 3 :- Hoisting in JavaScript 🔥(variables & functions)

- Consider the following code and observe its output in the console. You will see the expected output.
- 

- Now, just copy the code at line 9 and line 10 and paste at line 1. Now, what output do you expect to see ? Here, we are trying to access the function even before we have initialized it and we are trying to access the variable x even before we have put some value in it. In most of the programming languages this will result in an error but Javascript deals with it differently. Let's look at the output.

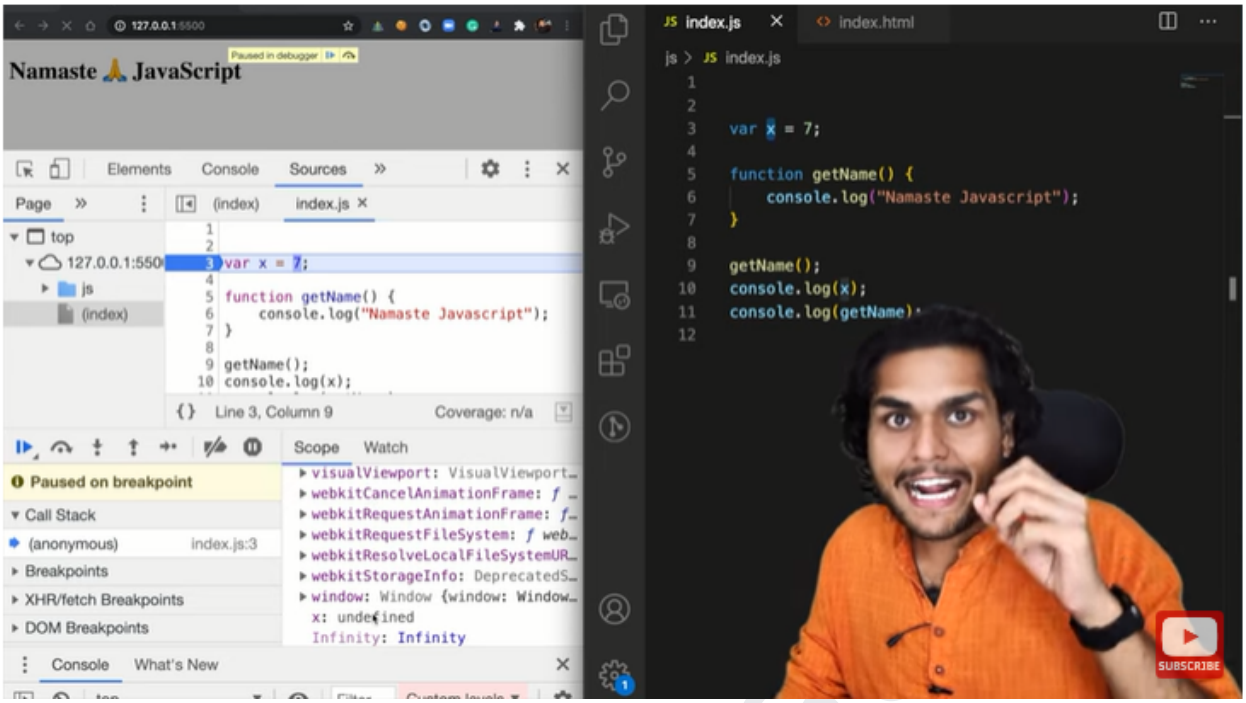
- 

- The output in the console will look like this. Isn't it something different? We get undefined when we are trying to access the variable x before it is initialized and when we are trying to access the function before its initialization we are somehow getting the desired output. Isn't it weird ? 😁
- 

- Well this happen because of an interesting concept in javascript known as **HOISTING** in javascript
- **HOISTING** is the phenomenon in javascript by which we can access the function and variables even before we have initialized them or put some value in it without getting an error.
- Let us try to **console.log getName** function before its initialization. In this case we will see the whole function as the output in the console and not undefined like the variable x. Isn't it weird ?
- 

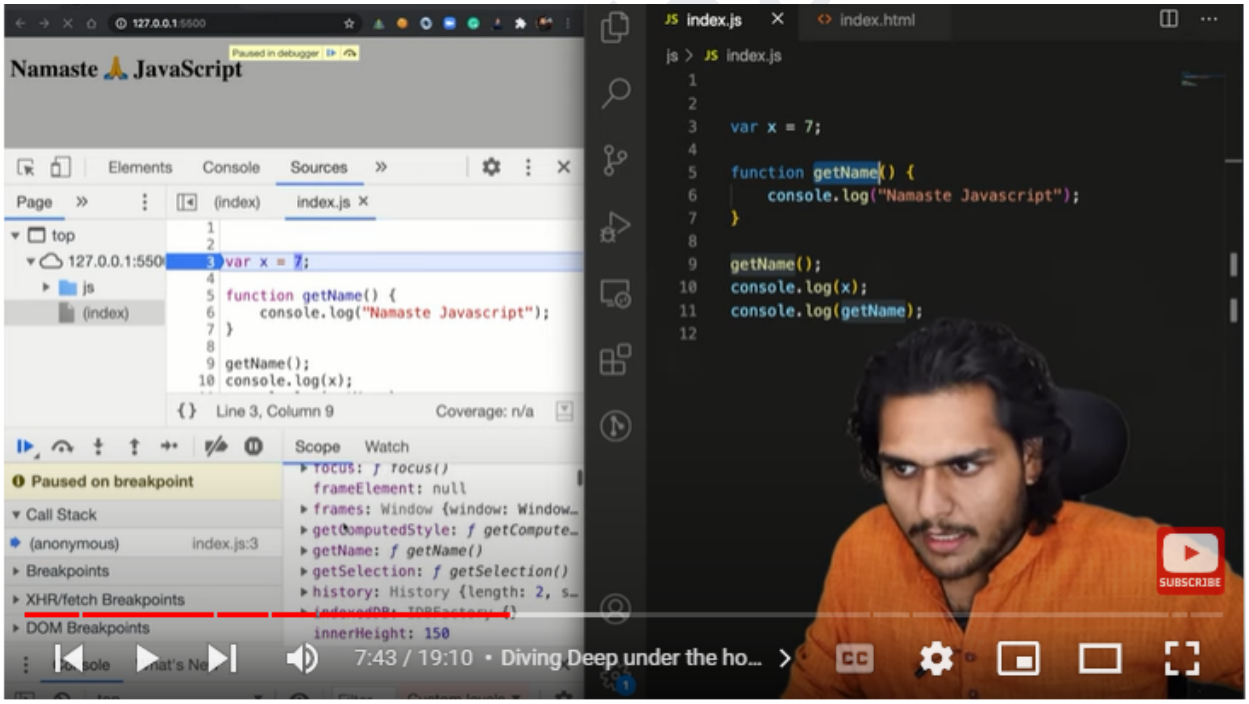
- Let's understand why this program is behaving like this
- We have discussed an important concept in the last episode that whenever javascript runs a program an execution context is created and it is created in two phases 1) Memory creation phase and 2) Code execution phase. So the answer of why the program is behaving like this lies in that concept only.
- We know that even when the whole code in javascript starts executing, memory is allocated to each and every variable and functions present in the code. Inside variables it stores a special keyword "undefined" and inside the functions it stores the entire code even before the execution of the code

-



-

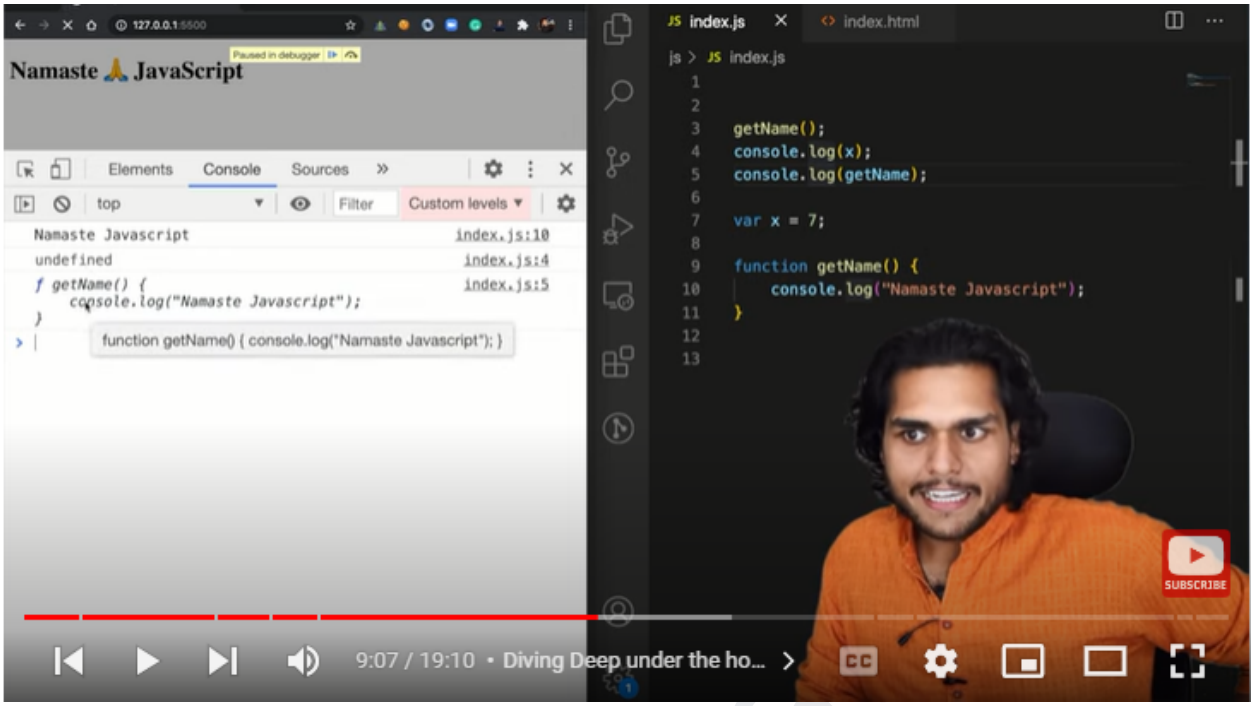
-



-

- So, now if we try to access the function and the variables before it is initialized then it will print “undefined” for variables and for function it will print the entire code.

-

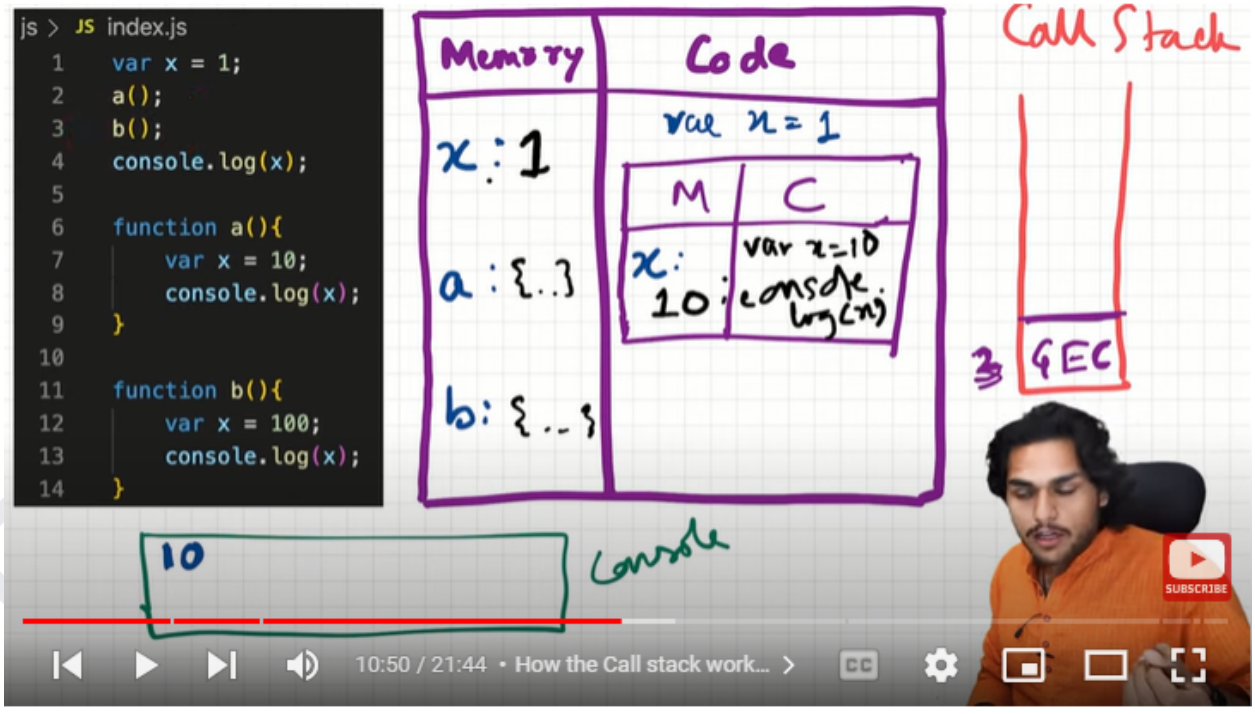


1. Difference between undefined and Not defined ? Explain with a coding example
2. What happens if we try to console.log the variables before they are initialized with some value in the code ? Explain with a code example ? Also explain why it is happening
3. What happens if we try to console.log or invoke a proper function before it is initialized in the code ? Explain with a code example ? Also explain why it is happening
4. Why do we get undefined if we try to log to the console the variables before they are initialized in the code ?
5. if we try to call the function before its initialize inside a javascript program than why we are able to call that particular function and it works but in other programming language it will give an error if we try to call function before its initialization
6. What is HOISTING ? Explain with a coding example
7. How HOISTING in javascript works. Explain in details
8. Why is HOISTING happening in the javascript ?
9. Explain the two phases of execution context ?
10. Name the two components of the execution context ?
11. Explain the whole concept of how code is executed in javascript behind the scenes ?

12. What happens if we try to call an ARROW function before it is initialized in the code. Explain it ?
13. How does the ARROW function behave if we try to call or try to log to the console before its initialization ?
14. Why does the Arrow function behave as a variable if we try to call or try to log to the console before its initialization ?
15. Show the demo how the call stack looks like inside the browser with a code example. (Time stamp :- 15:00 for revising this concept of episode 3)

Question from EPISODE 4 :-How functions work in JS ❤️ & Variable Environment

1. Explain how the code given below is executed with each and every step starting from creation of the global execution to the removal of Global execution from the CALL stack and observe the behavior of how functions are working ?



2. Show the demo of executing this code inside the browser by putting a debugger and explain each step how the code is being executed inside the javascript engine(check video for revision).

Question in EPISODE 5 :- SHORTEST JS Program 🔥window & this keyword

1. What is the shortest program in javascript ?
    - The shortest program in the javascript is the empty file. Even though the file is empty and there is nothing to execute, the javascript engine is doing a lot of things behind the scenes.
2. What happens behind the scenes when you run the shortest program in javascript ?
    - Whenever we run an empty file even though there is nothing to execute, the javascript engine still creates a global execution context and sets up the memory. Javascript engine also creates a **window object** which contains all the methods and functions which we can use anywhere inside our program. It also creates a **this** variable which points to the window object at the global space.
    - **Window is the global object which is created along with the global execution context**
    - **Note :- Whenever any javascript program is run a GLOBAL WINDOW OBJECT is created, a GLOBAL EXECUTION CONTEXT is created and a THIS variable is also created which points to the WINDOW Object at the global level.**
    - **Note :- Javascript is running in a lot other devices and places and it is not just limited to the browser and wherever javascript is running there must be a javascript engine and all these javascript engines have the responsibility of creating a global object. In case of the browser this GLOBAL OBJECT is known as WINDOW OBJECT**
3. **At global level is this === window ?**

4. **NOTE :- Whenever an execution context is created along with it a THIS variable is also created and it is also true for functional execution context.**
5. **What is GLOBAL SPACE ?**
    - **Any code that we write inside a javascript file which is not inside a function is known as the global space. Everything at the top level is the global space.**
6. **What happens if we create variables and functions in the global space ?**
    - **Whenever we create variables and functions inside the global space it will get attached to the global window object.**

7. **If we create a variable x inside the global space and we want to access it than what all options we have to access the x variable**
    - **console.log(x), console.log(window.x) and console.log(this.x) these are all the same.**

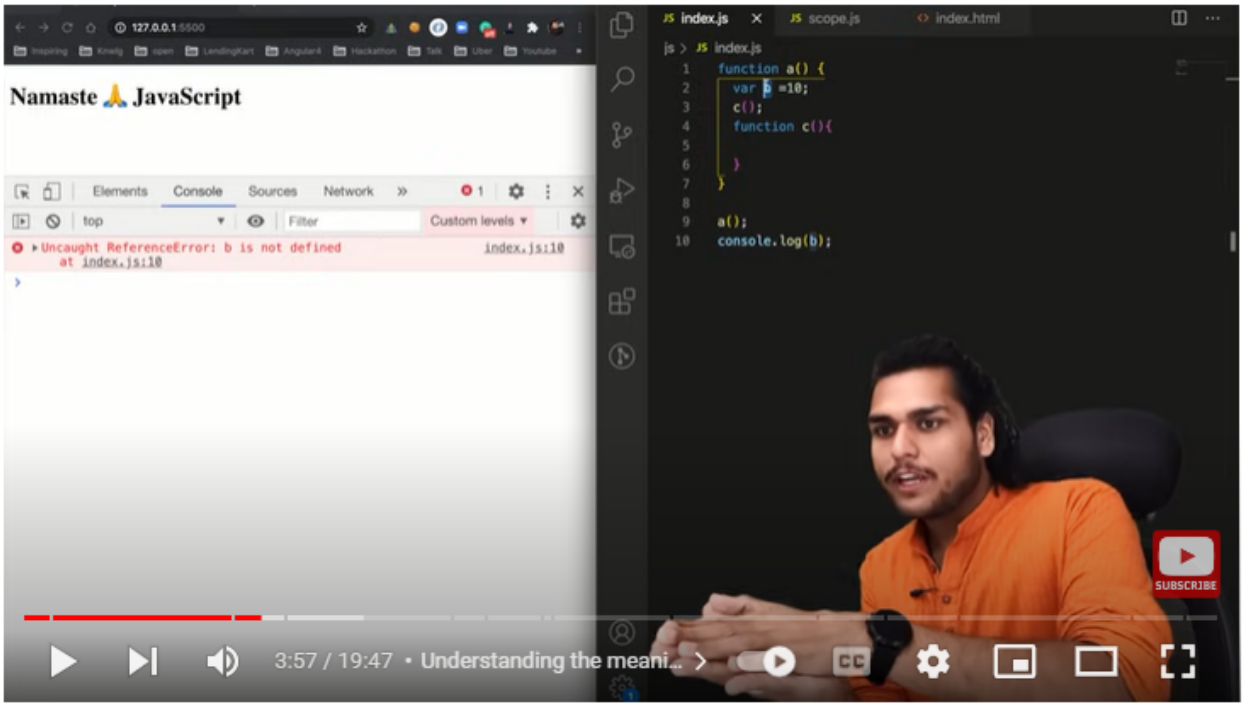
Question in EPISODE 6 :- undefined vs not defined in JS 🤔

1. **What is “undefined” in javascript ?**
    - **During the memory creation phase memory is allocated to all the variables and the functions and during this memory creation javascript stores a special placeholder “undefined” to all the variables. So, when we try to access a particular variable even before its initialization or if we try to access a variable which is never been initialized in the program than we will get “undefined” as the output**
2. **What is “not defined” in javascript ?**
    - **“Not defined” is something which has not been allocated any memory during the memory creation phase and we are trying to access that particular variable then it will give the error message that the variable is not defined.**
3. **Difference between undefined and not defined ?**
4. **Does undefined means empty ?**
    - **No, undefined does not mean empty. It is just a placeholder which is used to keep inside the variables during the memory creation phase; in fact it occupies some memory so it does not mean empty.**
5. **Javascript is a loosely typed or weakly typed language. Explain.**

Questions in EPISODE 7 : -The Scope Chain, 🔥Scope & Lexical Environment

1. **Note :- Scope in javascript is directly related to its lexical environment**
2. **What is scope ? Explain with a code example.**
    - **Scope is something where we can access a specific function or variables in our code. There are two aspects to it 1) What is the scope of this variable that means where can I access the variable b. 2) Is b inside the scope ? when we say:- is b inside the scope of function c ?**

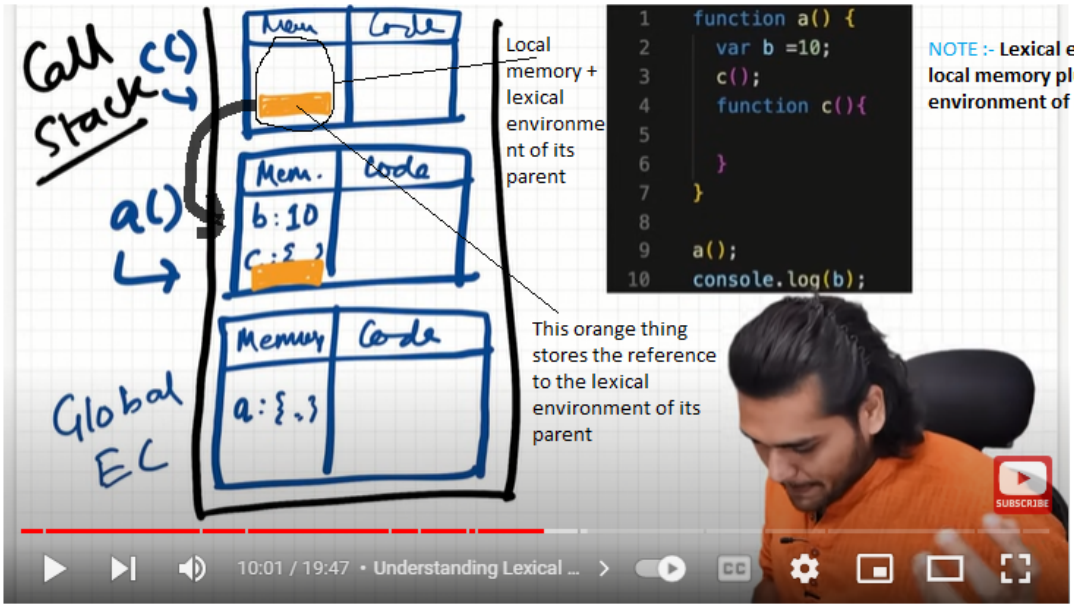
-



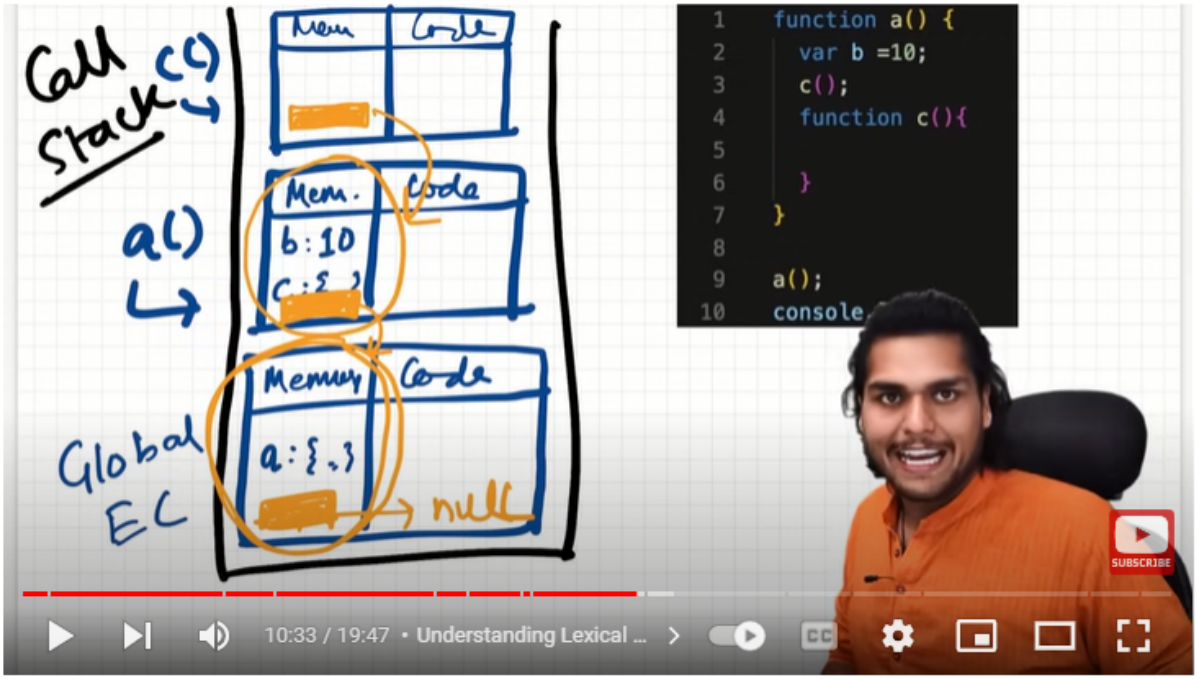
2) What is the lexical environment ?

- When we run a program in javascript a global execution context is created and it is pushed onto the call stack. Whenever an execution context is created a **lexical environment** is also created. **Lexical environment** is the local memory plus the lexical environment of its parent. Lexical means in order or in hierarchy. In the code example, we can say that the function c() is lexically sitting inside the function a() and function a() is lexically inside the global scope.

-



-

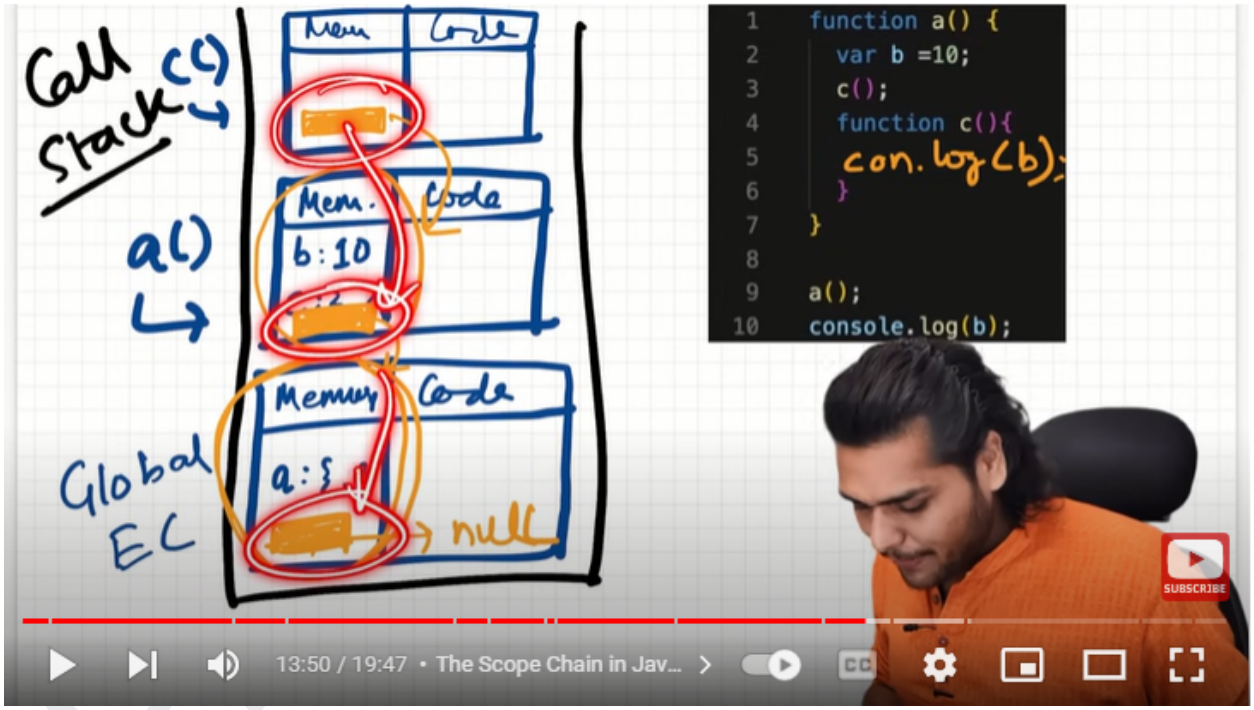


-
- At global level this reference to the outer environment points to NULL as it has no parent.
- This is how lexical environment looks like in memory.

3. How is this lexical environment used ? Explain how scope chaining works ?

- Let's say we want to console.log(b) inside the function c(). Now, javascript engine try to find b in the local memory of the function c() and as variable b is not present inside the local memory, javascript engine will go to the reference to the

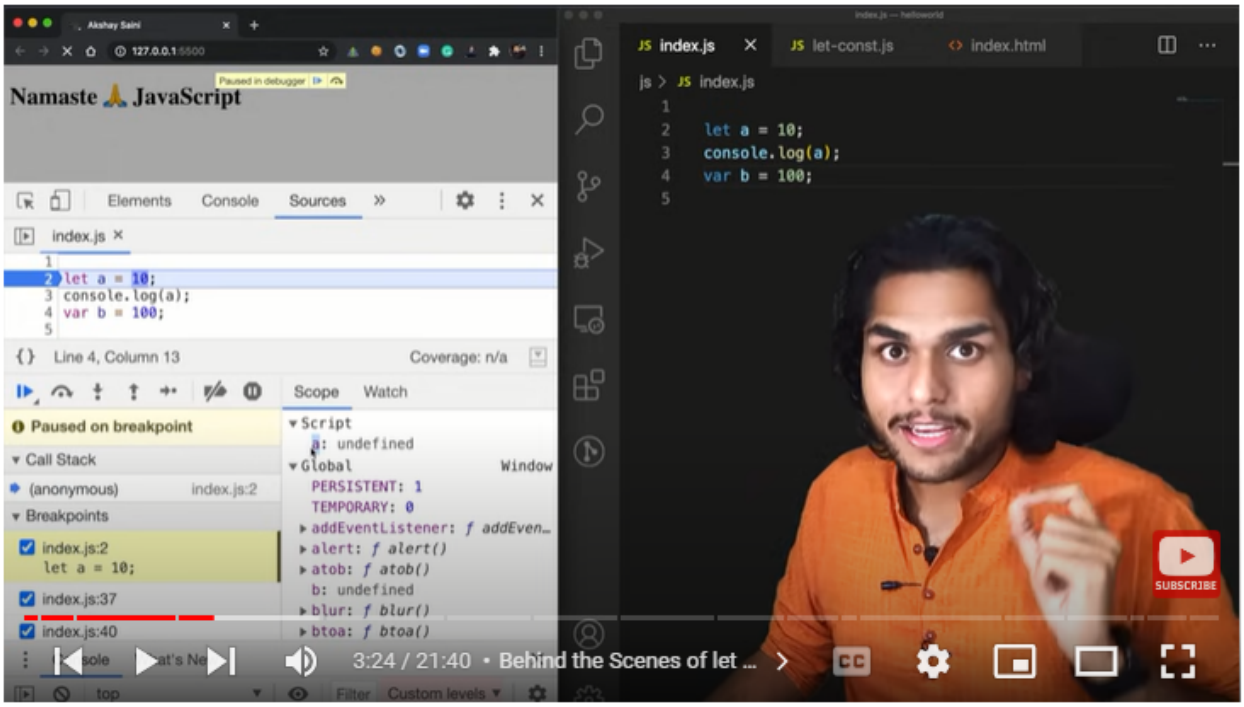
lexical environment of its parent and will try to find out b their and b is present their so it will print the value of b.

- Let suppose b is not present in the lexical environment of function a() than javascript engine will go to the reference to the lexical environment of a() parent which is the global execution context and try to find out b their and if it is not able to find b their than it will go to the reference of the lexical environment of the global parent and as at global level the reference points to the NULL, javascript engine fails to find b and it will throw an error to the console and will say b is not present in the scope. This is how the javascript engine goes to the next level of scope chain if it is not able to find a particular variable inside the local memory. So this way of finding a variable in a hierarchical way is known as the **scope chaining. Scope chaining is the chain of this lexical environment and the parent references**
- 

4. **Show the demo of how the lexical environment works inside the browser ?**

Question in EPISODE 8 :- let & const in JS 🔥Temporal Dead Zone

1. **Are let and const declaration hoisted ?**
    - Yes let & const declarations are hoisted but they are hoisted very differently than the var declaration.
2. **What happens if we try to access the variables declared with let and const before they are initialized in the program ?**
    - It will throw a reference error that we cannot access it before initialization.
3. **Why does it throw a reference error if we try to access variables declared with let & const even though they are hoisted ?**
    - Let's understand this with a code example



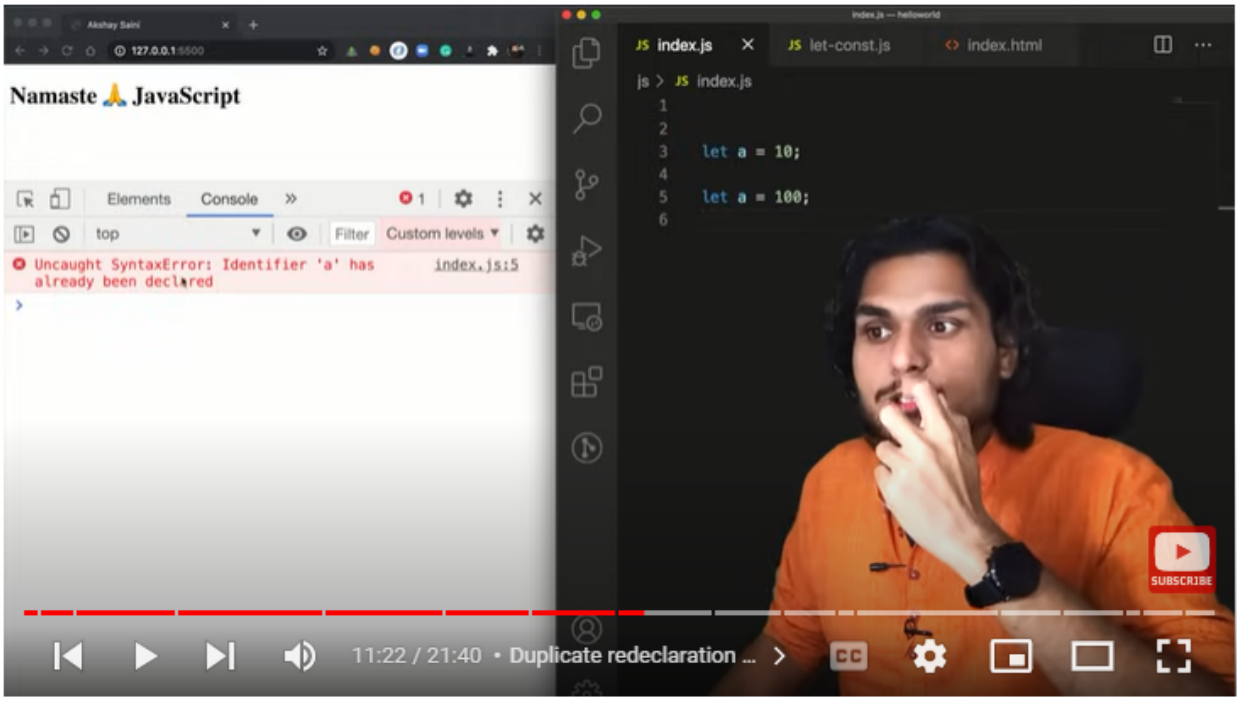
-

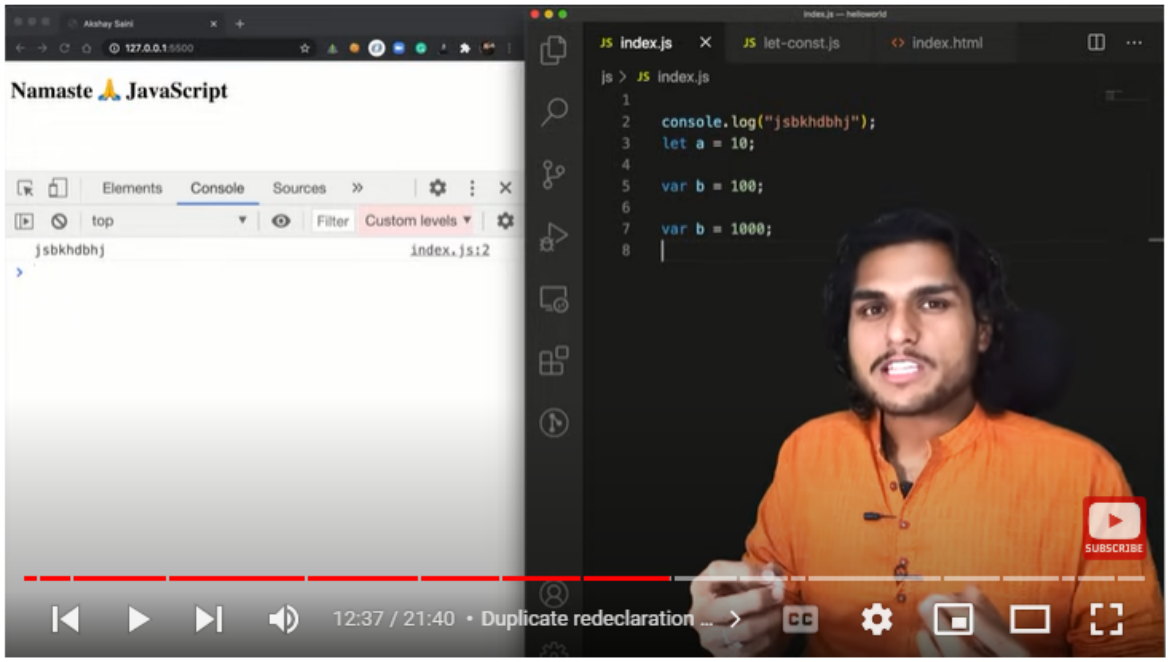
- Even before a single line of code is executed javascript has allocated memory for variable **a** which is declared with **let declaration** and javascript has also allocated memory for variable **b** which is declared with **var declaration** and stored a placeholder "**undefined**" into them.
- Memory was assigned to variable **b** which is declared with **var** declaration and this variable **b** is attached to the global object and incase of the variable declared with let and const they are also allocated memory that means they are also hoisted but they are stored in some other place and not attached to the global object and we cannot access these variables

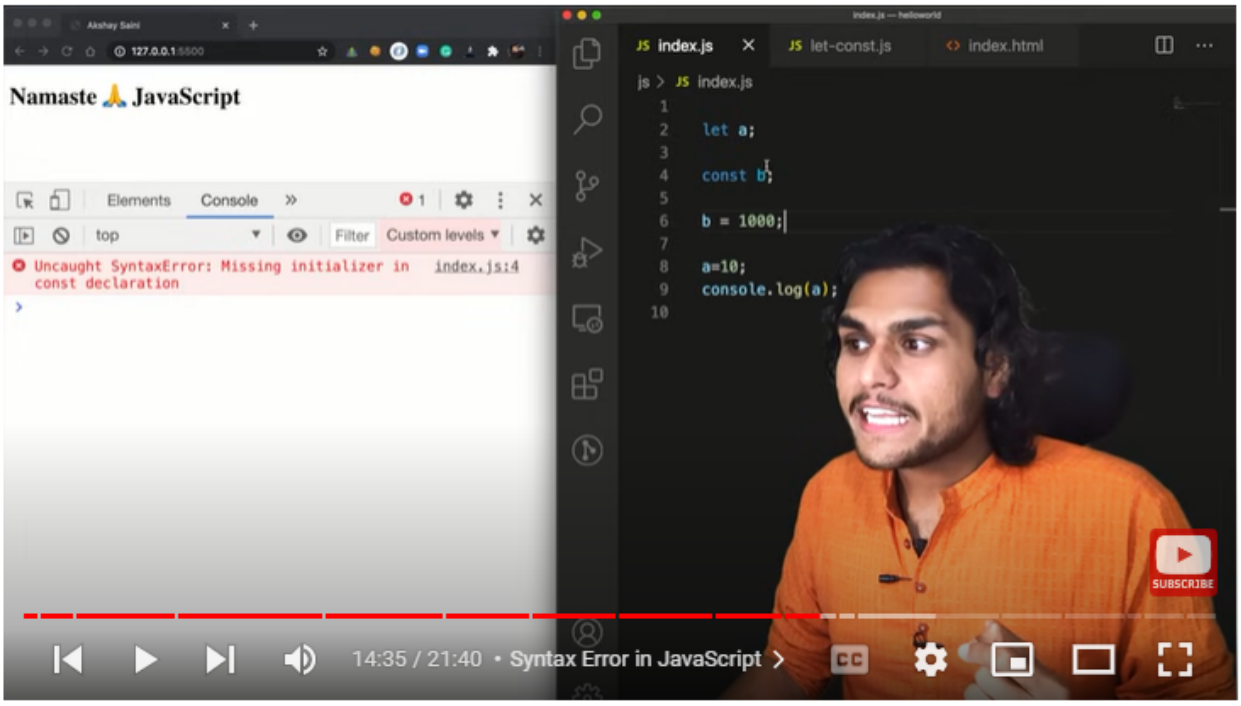
declared with let & const or we cannot access this memory place before we have put some value inside it.

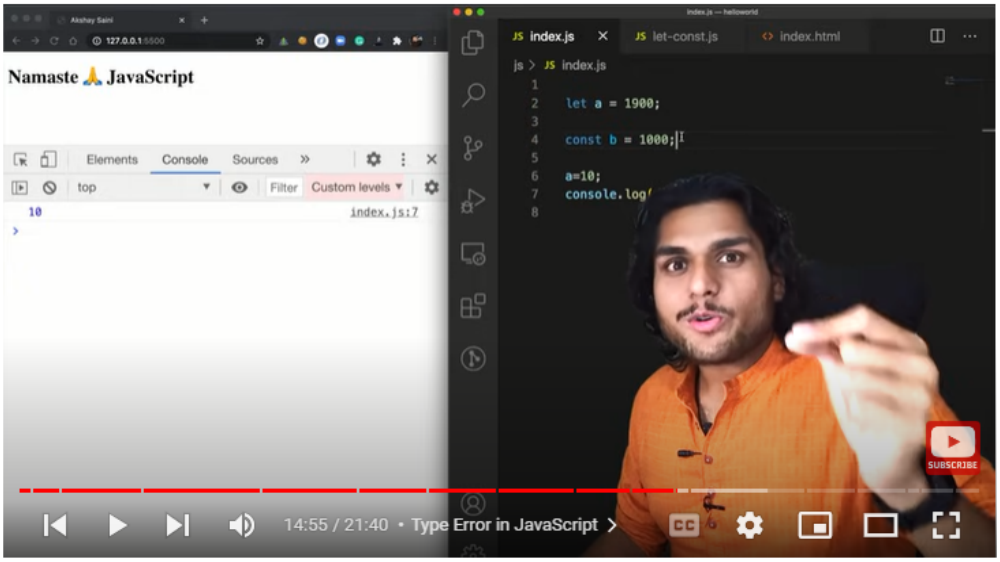
- So, we can say that variables declared with let & const are hoisted but they are stored at different location other than global object and we cannot access that memory location before we have initialized the variables with some values.
- After a value is allocated to the variables declared with let and const than we are able to access that memory location and we can access those variables declared with let & const as well.
- **NOTE :- we have seen that variable declared with var are attached to global object i.e window object in case of browser but variables declared with let and const are not attached to window object they are placed at separate location. There if we try to access variable declared with let or const like console.log(window.a) we will not be access it and it will throw a reference error.**

4. **What is the Temporal Dead Zone ?**
    - **A temporal dead zone (TDZ) is the area of a block where a variable declared with let & const is inaccessible until the moment the computer completely initializes it with a value.**
    - **We can say that the temporal dead zone is the area in which the variable is inaccessible before it is initialized with some values.**
    - **Whenever we try to access a variable inside the temporal dead zone it will give a reference error.**

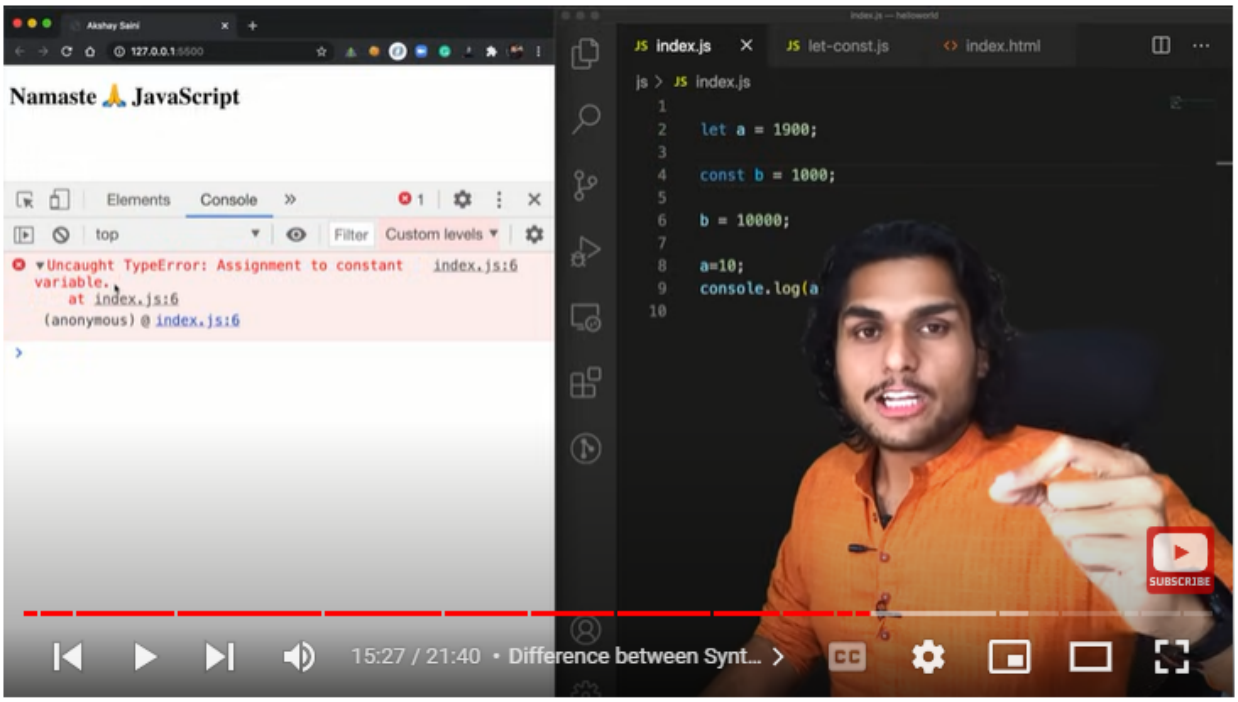
- **Let is more strict than var so if we try to redeclare a variable declared with let then it will throw a syntaxError and not a single line of code will run.**
- **Therefore javascript will not execute a single line of code if it sees a redeclaration or a duplicate declaration of let in the same scope.**
- 

- **But redeclaration or duplicate of var is possible and javascript will not throw a syntax error**
- 

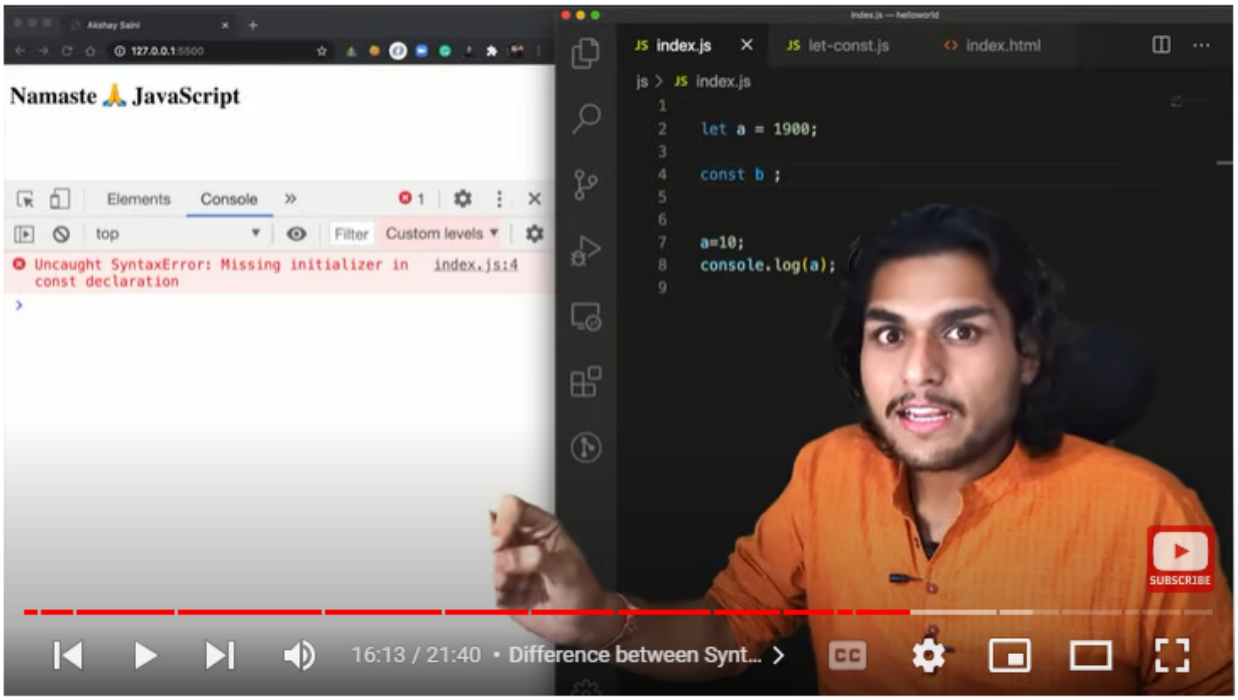
- **In case of const declaration, it behaves similarly as let declaration but it is even more stricter than let declaration. Const declarations are stricter in a sense that we have to initialize them in the same line in which they are declared or else it will throw a syntaxError. In case of let declaration we can declare let in some line and than we can initialize it in some other line as well. In the below image a variable is declared with const in line 4 and it is initialized in line 6. Hence, it will throw a syntaxError.**
- 

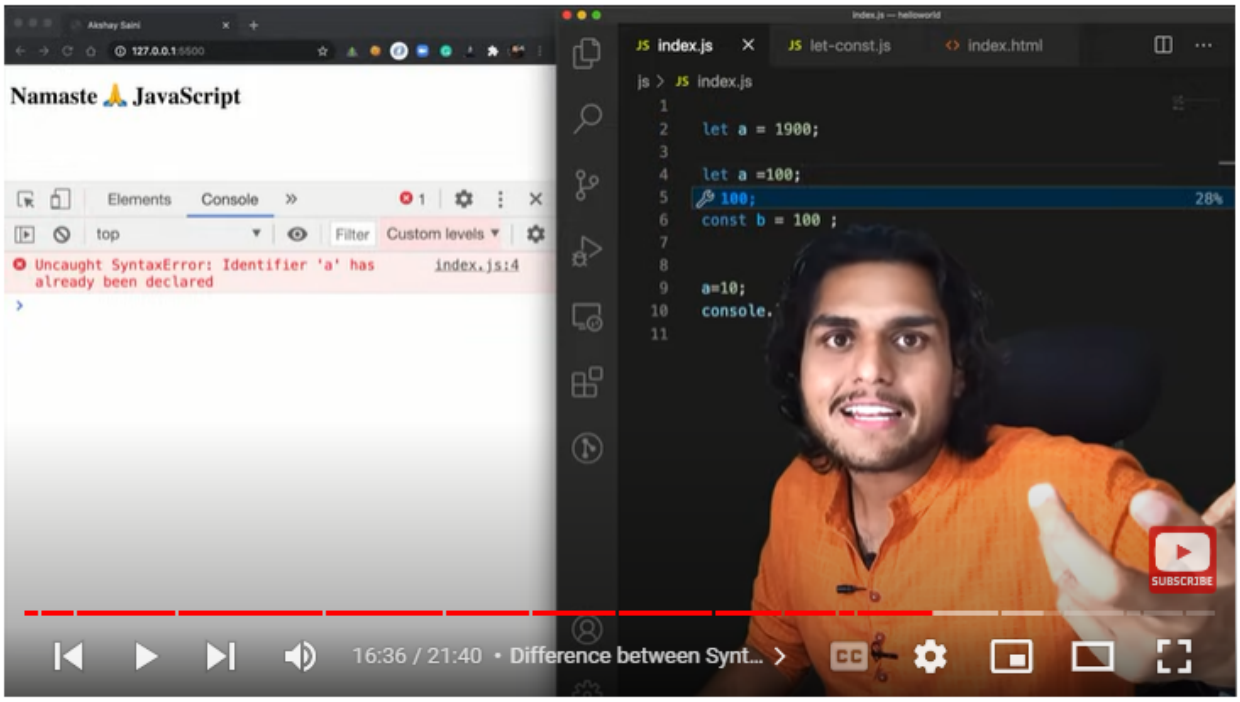
- **So incase of const we have to definitely declare and initialize on the same line.**
- 

5. **Difference between syntaxError, typeError and referenceError ?**
    - **TypeError :- it says that assignment to const variable is done on the line in which it is declared and if we try to assign a different value to a variable declare with const than it will throw the typeError.**



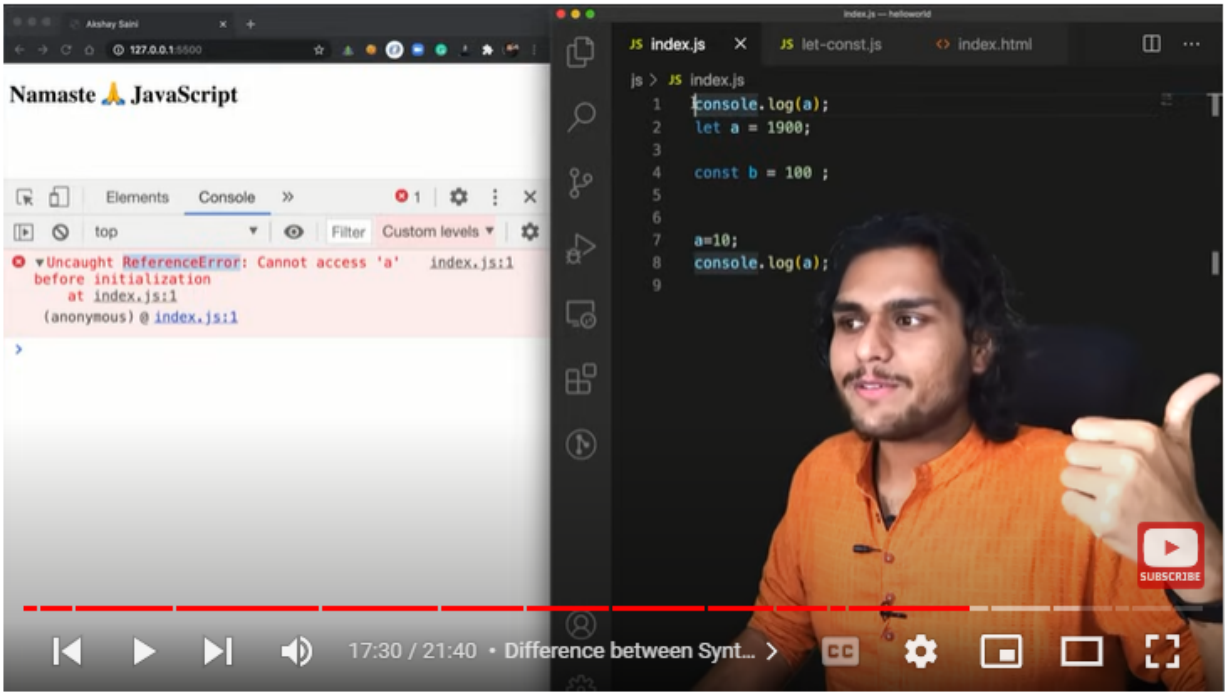
- **syntaxError :- if we do not initialize a const variable at the time of its declaration then it will throw a syntaxError : Missing initializer in const declaration. It will also throw a syntaxError if we try to redeclare or duplicate the variable declared with let.**

- 

- 

- **referenceError :- when javascript engine tries to find out a variable inside the memory and we cannot access it then it will throw a referenceError. It will also throw a reference error if we try to access the variable declared with let or const before they are initialized in the program.**

- **If we try to access a variable which we have never declared in our program then also it will throw a referenceError.**

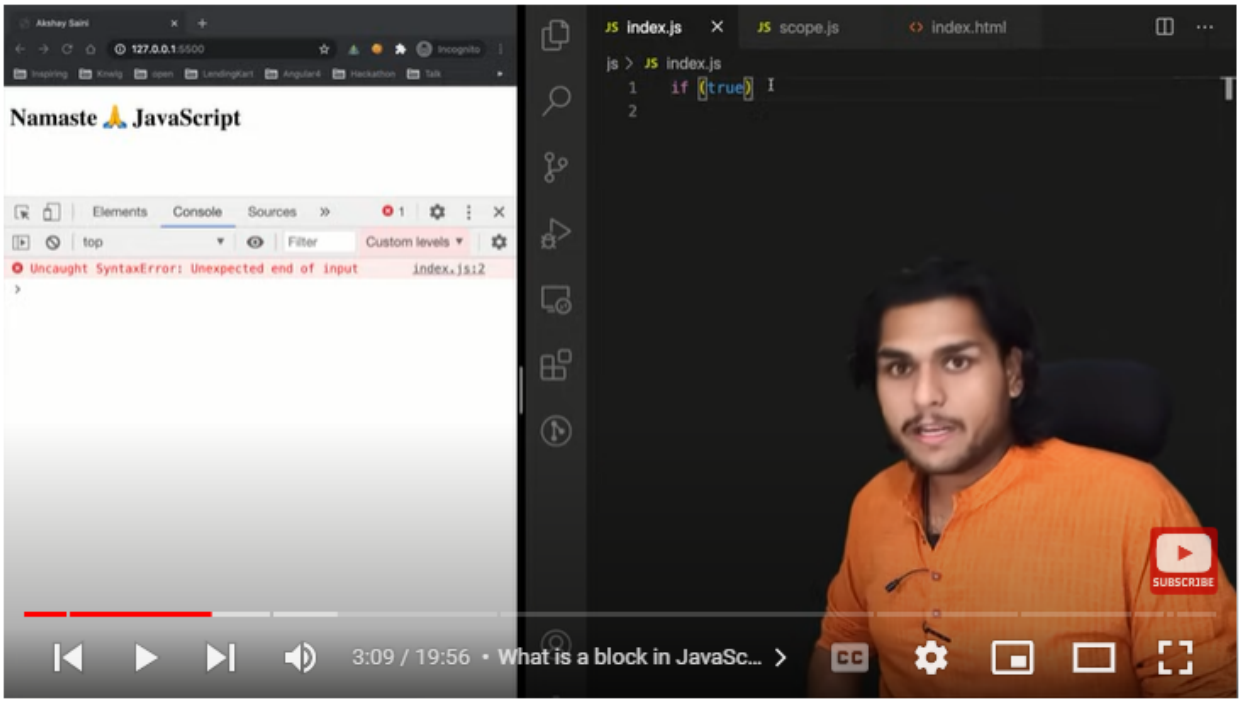


-

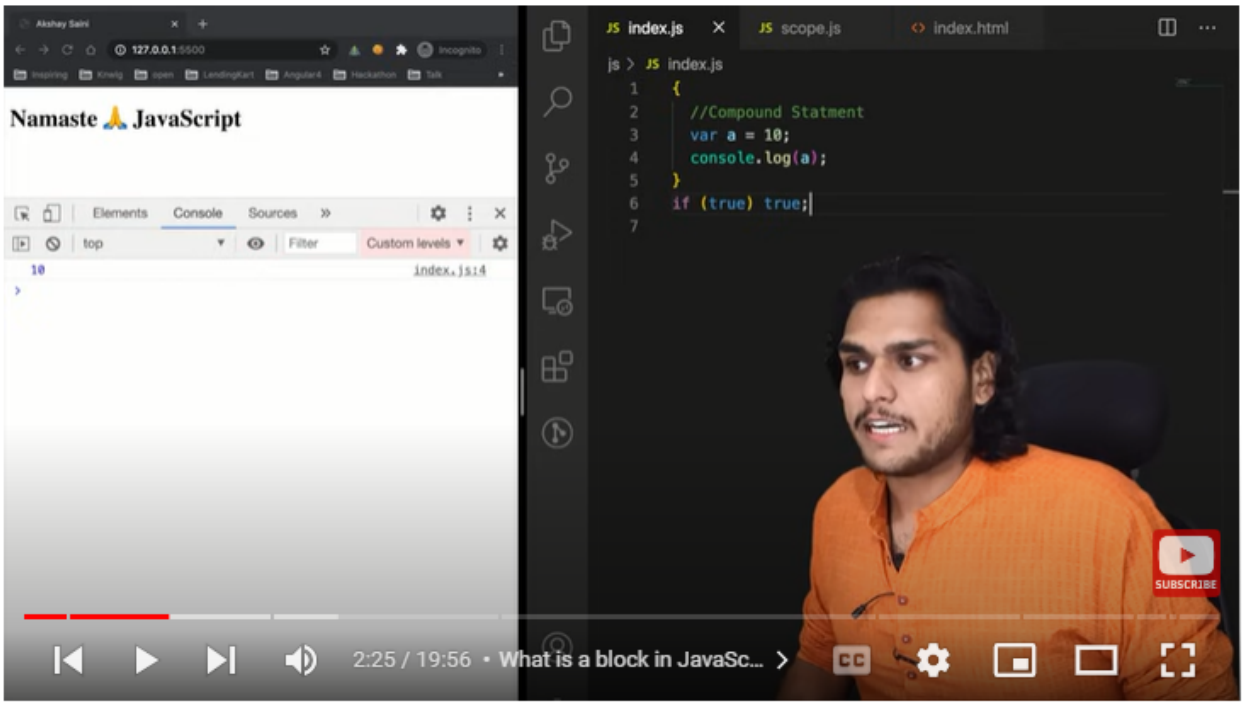
6. **We have three ways of declaring a variable so which is more preferable**
    - **Const > let > var follow**
    - **Try to declare a variable with const only wherever possible but if we want to change the value of the variable later in our program then only declare it with let. Try to avoid declaring a variable with var as much as possible.**
    - **Prefer to use just let and const in day to day coding.**
7. **How to avoid a temporal dead zone(TDZ) which may lead to unexpected errors?**
    - **By initializing all the variables at the top of the scope we can avoid TDZ. It also means that we have shrunk the TDZ window to zero by declaring and initializing all the variables at the top of the scope.**

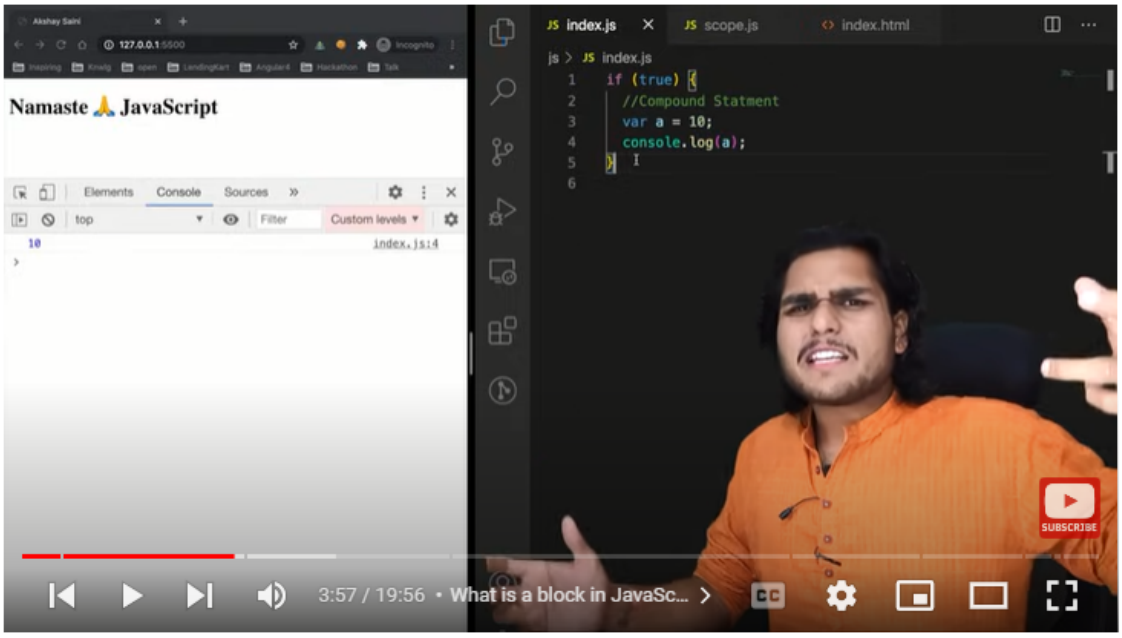
Question in EPISODE 9 :- BLOCK SCOPE & Shadowing in JS 🔥

1. **Let & const are block scoped**
2. **What is a BLOCK ?**
    - **A block is defined by curly braces. Block is also known as COMPOUND STATEMENT.**
    - **A block is used to combine multiple javascript statements into one group but why do we need to group multiple statements ? we need to group multiple statements together so that we can use multiple statements in a place where javascript expects a single statement.**
    - If we observe the syntax of **if statement** than it expects a single statement after the condition and if we don't write a single statement after the condition than it will give an error



- It has thrown a syntaxError as we have not written any statement after the **if(true)** . So the if statement expects a single statement after the condition.

- 

- We have written **if(true)true;** and now it will not throw an error as we have written a single statement after the condition in **if statement.** The if statement expects a single statement after the condition but if we want to use multiple statements in place of a single statement then a **block** comes into the picture as Block is the only way to group multiple statements together. **Block** groups multiple statements together and it can be used in the place where javascripts expect a single statement.
- 

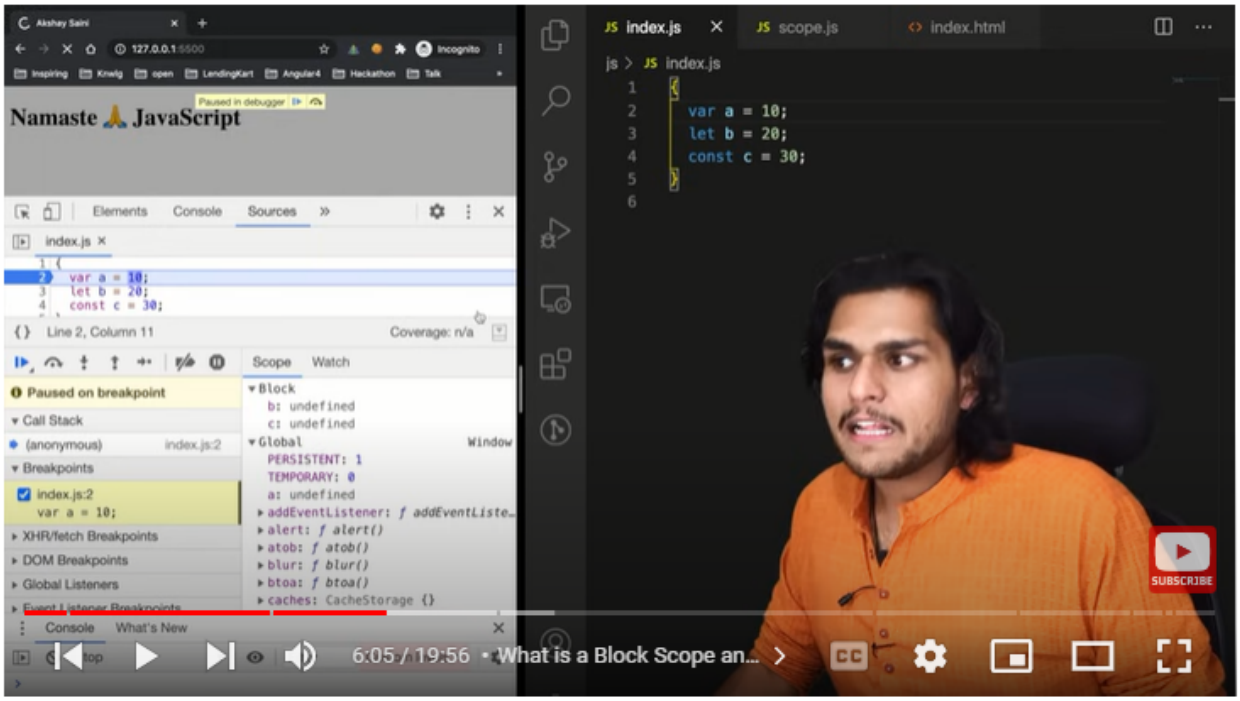
- So, we can use this **Block** to group multiple statements and use it in **while loops, for loop, if else statements.**

2. What is a BLOCK SCOPE ?

- **What all variables and functions we can access inside the block is known as BLOCK scope.**

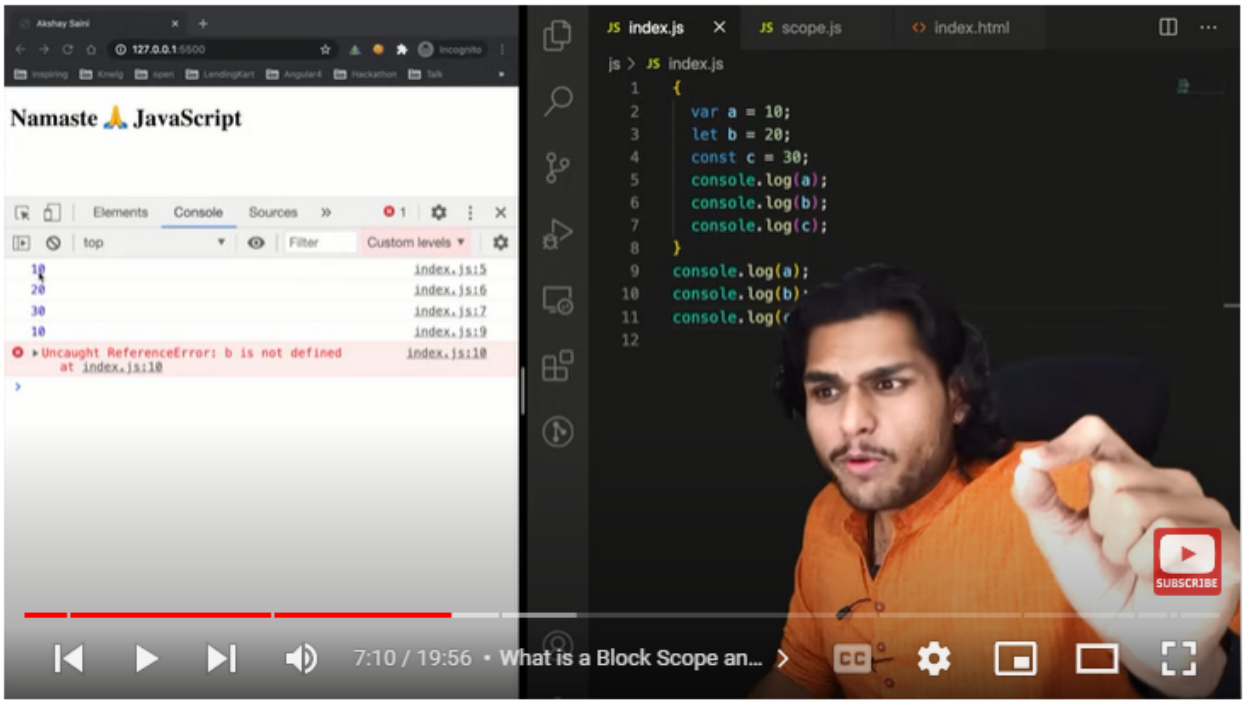
3. Why are let and const are BLOCK SCOPE ?

- If we declare variables with let and const inside a block and when we observe behind the scene behavior than we will see that variables declared with let and const are allocated a separated memory location i.e they are hoisted at different memory location known as block and the variable declared with var will still be allocated the memory in the global scope.



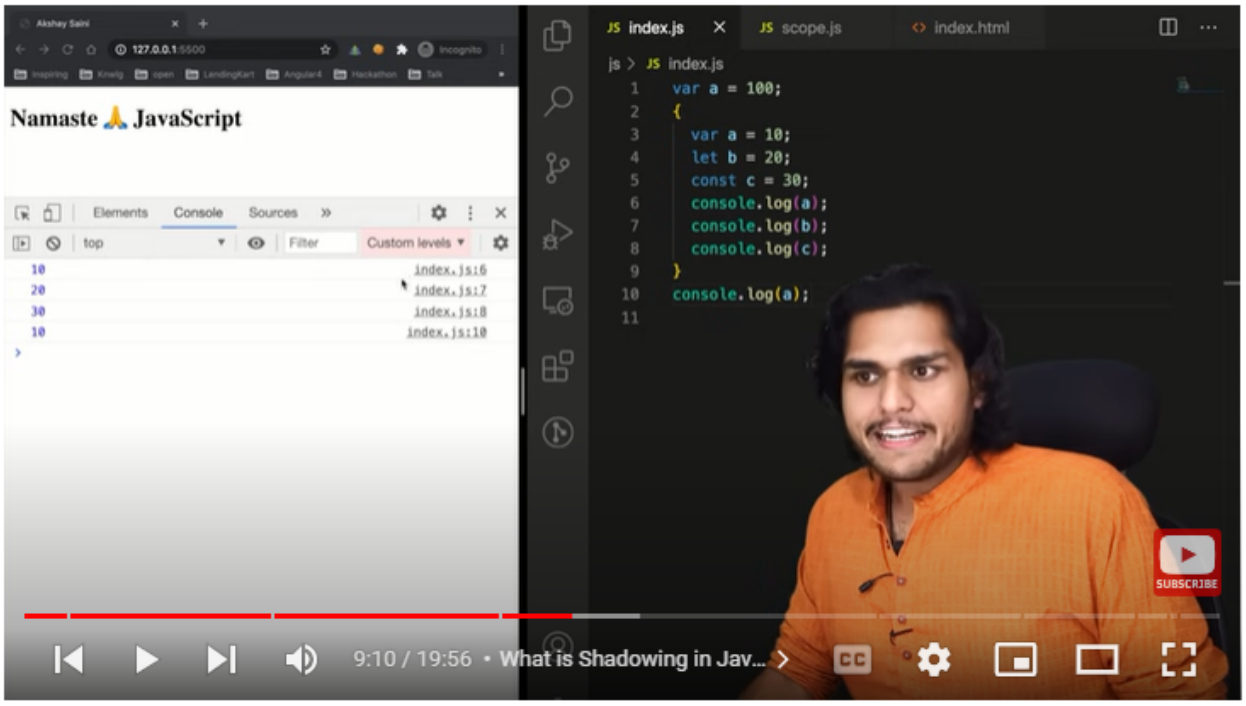
-
- We cannot access the variables declared with let and const outside the block in which they are declared but incase of variable declared with var inside the block, we can access them outside the block as well
- Hence, let & const are called **BLOCK SCOPE.**

-


Namaste JavaScript
Elements
Console
Sources
top
Filter
Custom levels
index.js:5
index.js:6
index.js:7
index.js:9
Uncaught ReferenceError: b is not defined
index.js:10
index.js
scope.js
index.html
var a = 10;
let b = 20;
const c = 30;
console.log(a);
console.log(b);
console.log(c);
SUBSCRIBE
7:10 / 19:56 • What is a Block Scope an...

3. What is shadowing in javascript ?

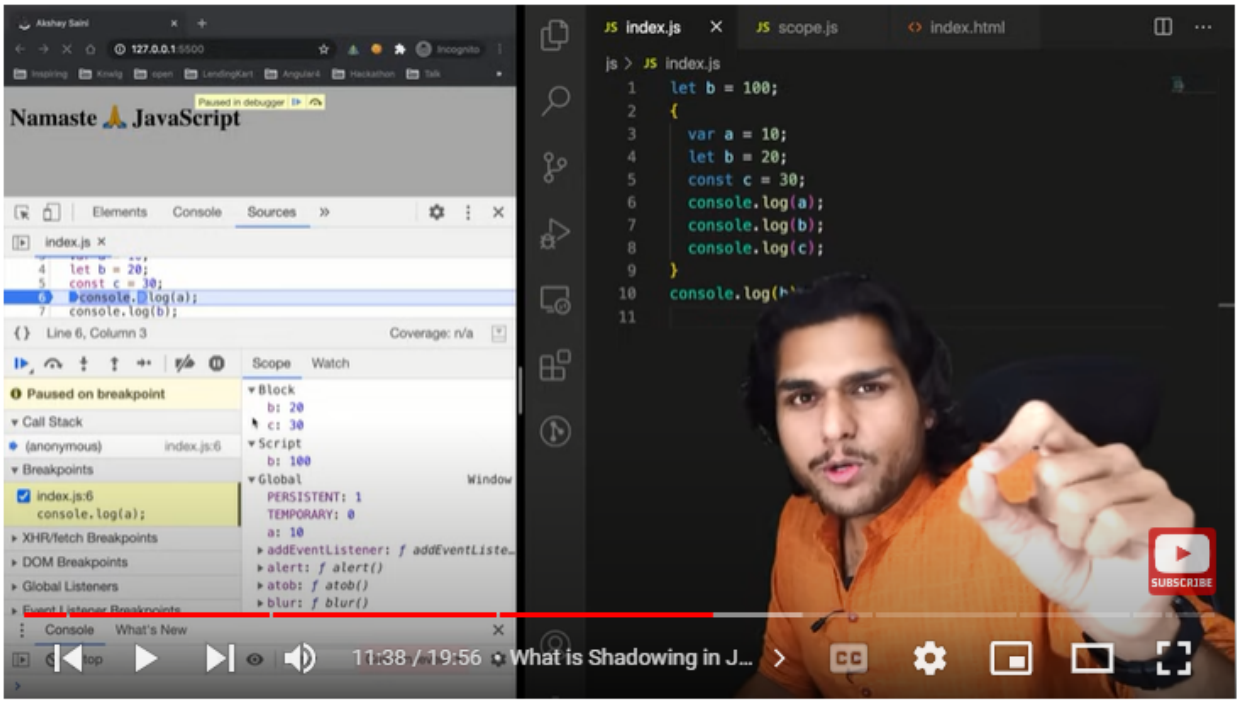
- If we have the same named variable inside and outside the block then the variable which is inside the block will shadow the variable with the same name which is outside the block. It means if we try to access that variable inside the block then the variable which is inside the block will be accessed and the variable which is outside the block but having the same name will not be accessed.



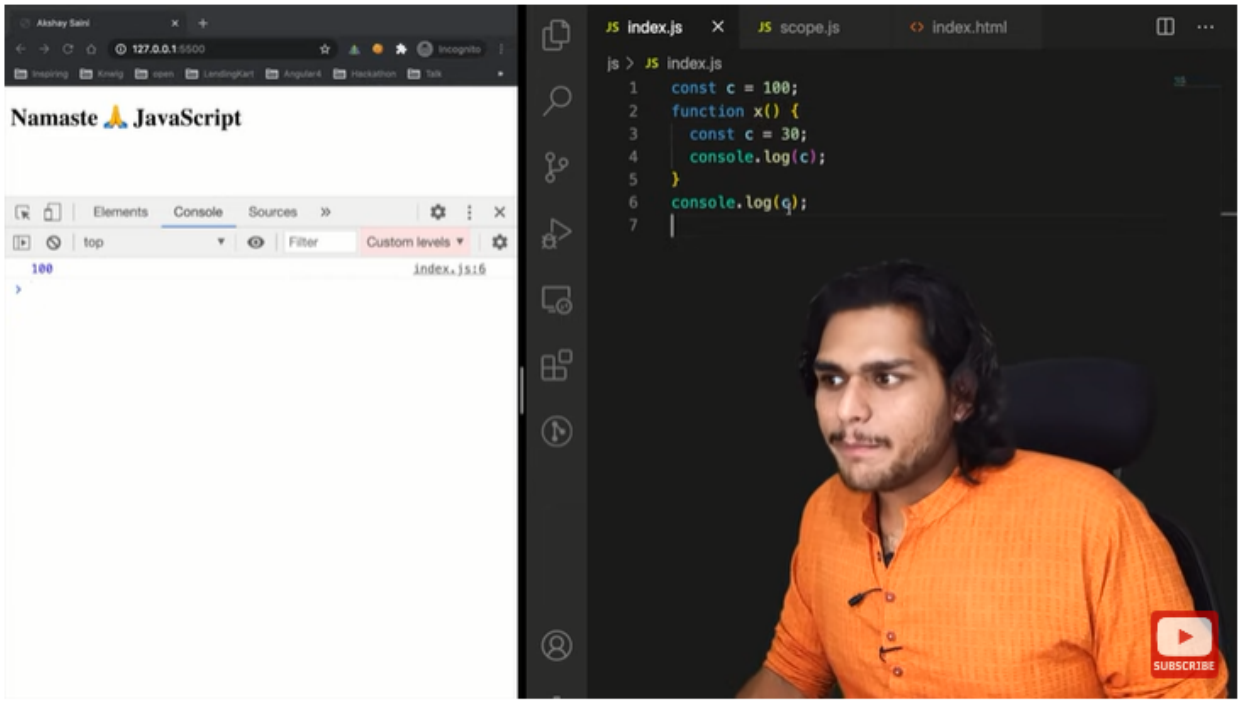
-
- As shown in the above example that if we try to print **variable a** which is declared with **var** inside and outside the block than we will get 10 as the output because variable **a** which is declared at line 1 will be shadowed by the variable **a** which redeclared inside the block and it also modifies the original variable as both of them will point to the same location in the memory. It happens only in the case of var declaration and incase of let and const declaration the behavior is different when they are shadowed(check video for revision if confused)
- If we declare a variable with same name but one inside the block and other outside the block with let or const declaration and if we want to access them inside and outside the block than if we try to console.log it inside the block than the variable which is inside the block will be accessed and if we try to access it outside the block than the variable which is outside the block will be accessed. In the case of let and const declaration shadowing happens inside the block but it does not modify the variable which is declared outside the

block because they both are allocated memory at different locations. Variable which is declared inside the block will be in the block scope and the variable which is having the same name but declared outside the block will be inside some other scope.

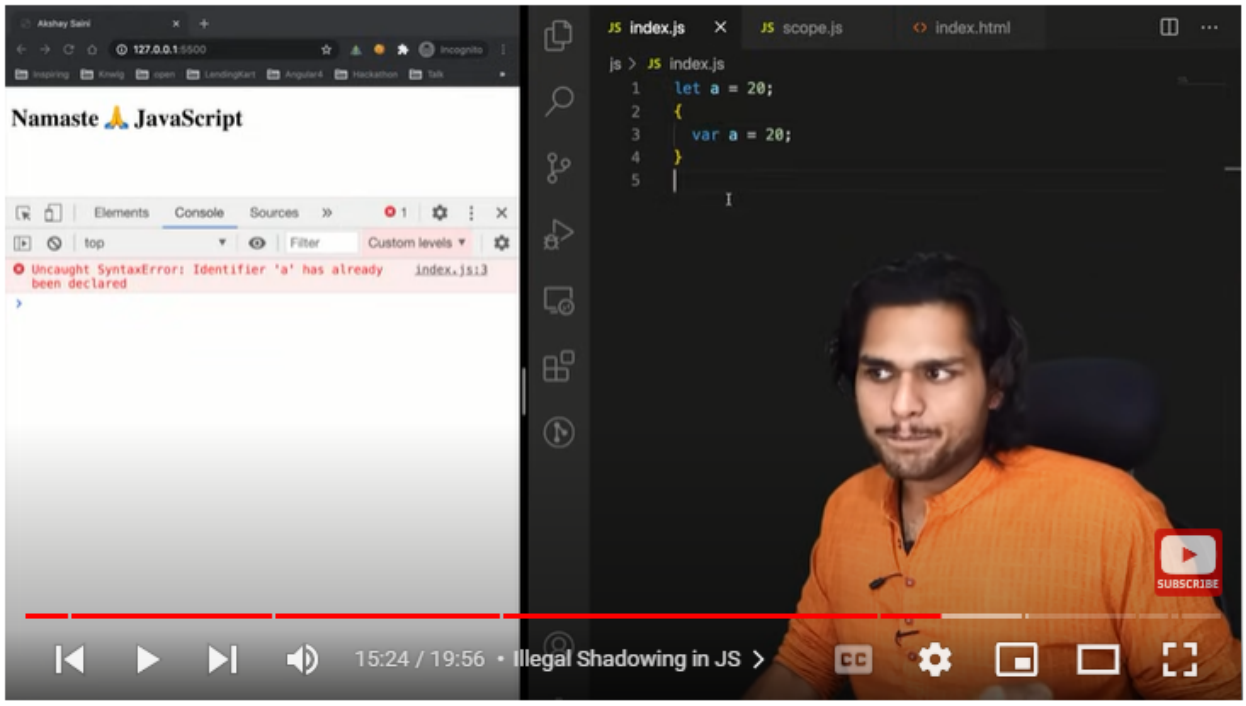
- See the variable b in the below code example.
-



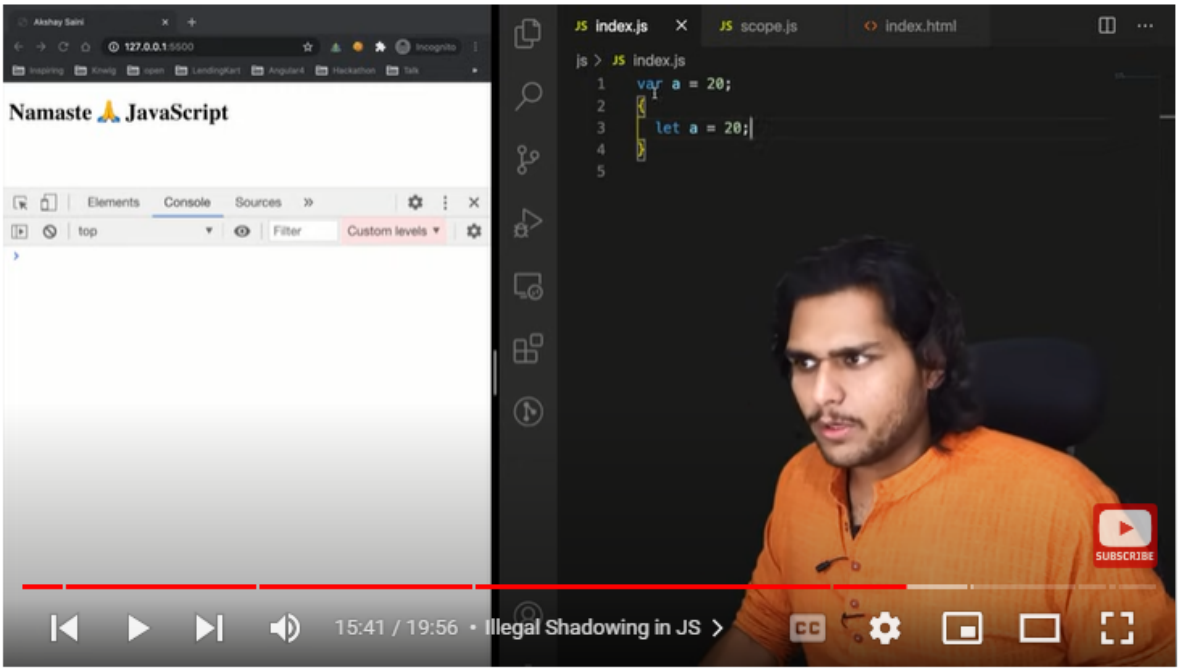
NOTE :- shadowing is not only the concept of block it behaves in similar fashion in functions as well. Look at the below code example.



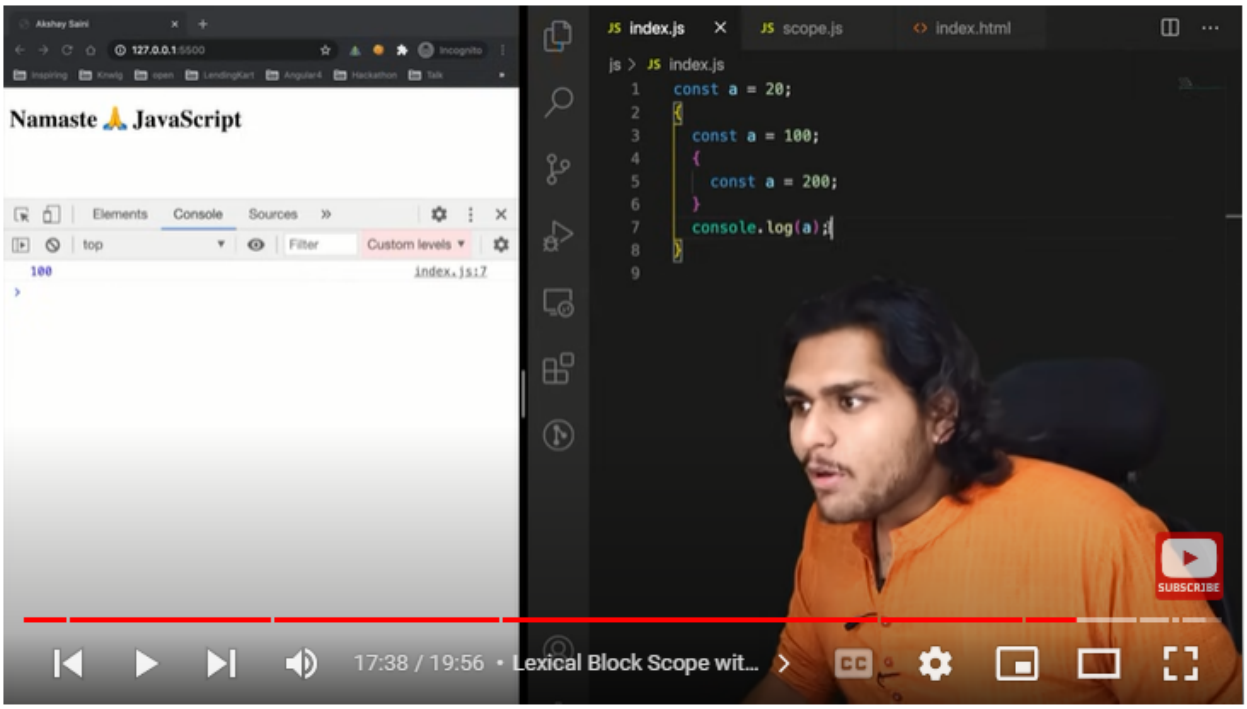
4. **What is illegal shadowing ?**
    - If we want to shadow a variable declared with let using var inside the block then we cannot do it and it is known as illegal shadowing**.**
    - As variables declared with var are function scoped and if they are declared inside a block than it will cross its boundary i.e it will be accessible outside its scope as well means we can access var a = 20; outside the block but we all know that variable named a is already declared in the outer scope so it will throw an error that ‘a’ has already been declared.
    - But instead of putting var a =20 inside a block if we put it inside a function than it will not throw an error as variable declared with var are function scoped and it will then not cross its boundary.
    - **We can shadow let using let**



5. **Can we shadow a variable declared with var using a let inside the block ?**
    - **Yes we can do this because variables declared with let are block scoped and they can be accessed inside the block only and if we have declared some variable with the same name outside the block than it will not throw any error as variable declared with let and const are block scoped and can be accessed inside the block only if they are declared inside the block. See the below code**



6. **Block scope also follows lexical scope and it also follows lexical scope chain pattern.**

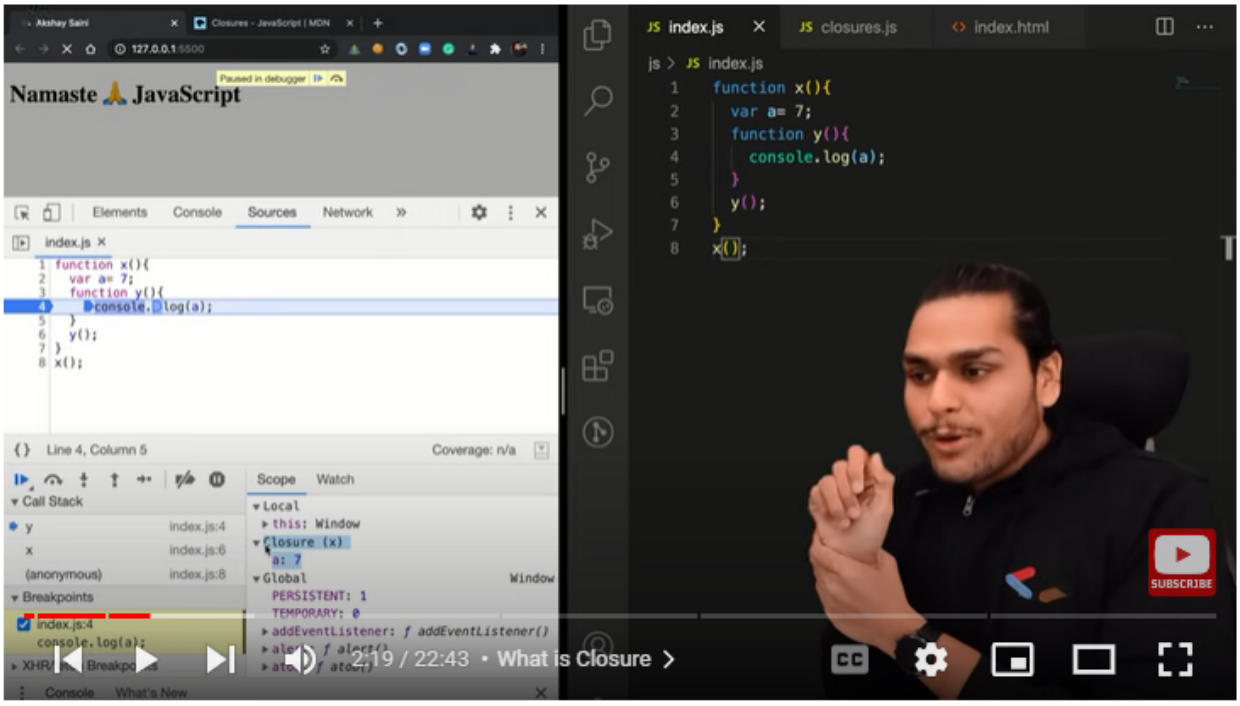


-

7. **All the scope rules for normal function and the arrow functions are the same.**

Questions in Episode :- 10 Closures in JS 🔥

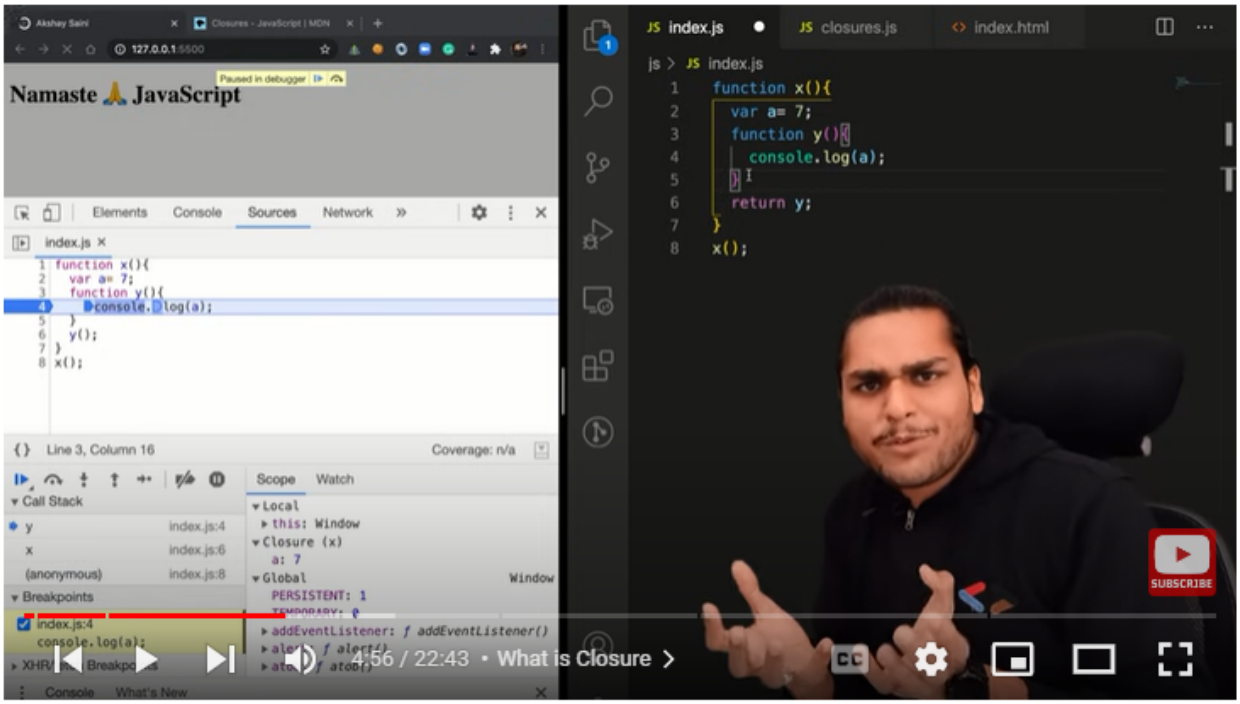
1. **What is a closure ?**
    - **A function bound together with its lexical environment is known as Closure.**
    - **A function bundled together with its lexical environment or A function bundled together with its lexical scope forms a closure.**
    - **A *closure* is the combination of a function and the lexical environment within which that function was declared**
    - **In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time**
    - **Check the link for detailed explanation of closure :- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures**



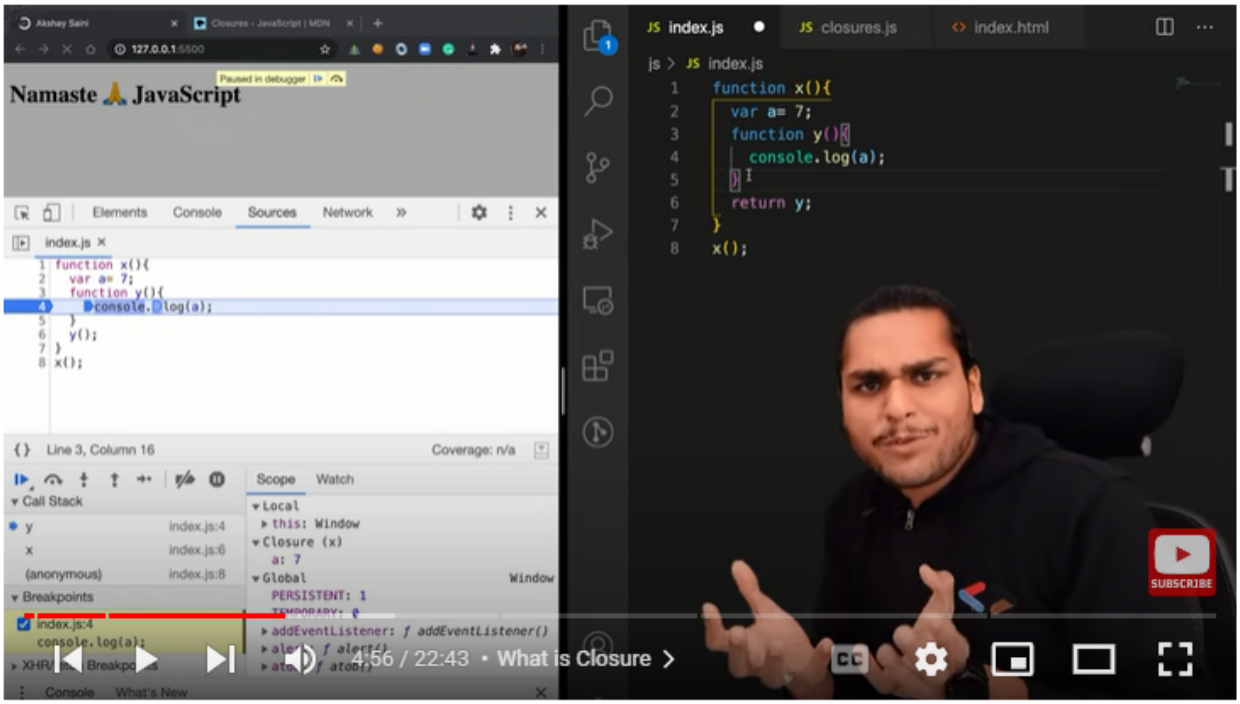
-
- The function y has access to the scope of function y because function y forms a closure with its lexical environment.

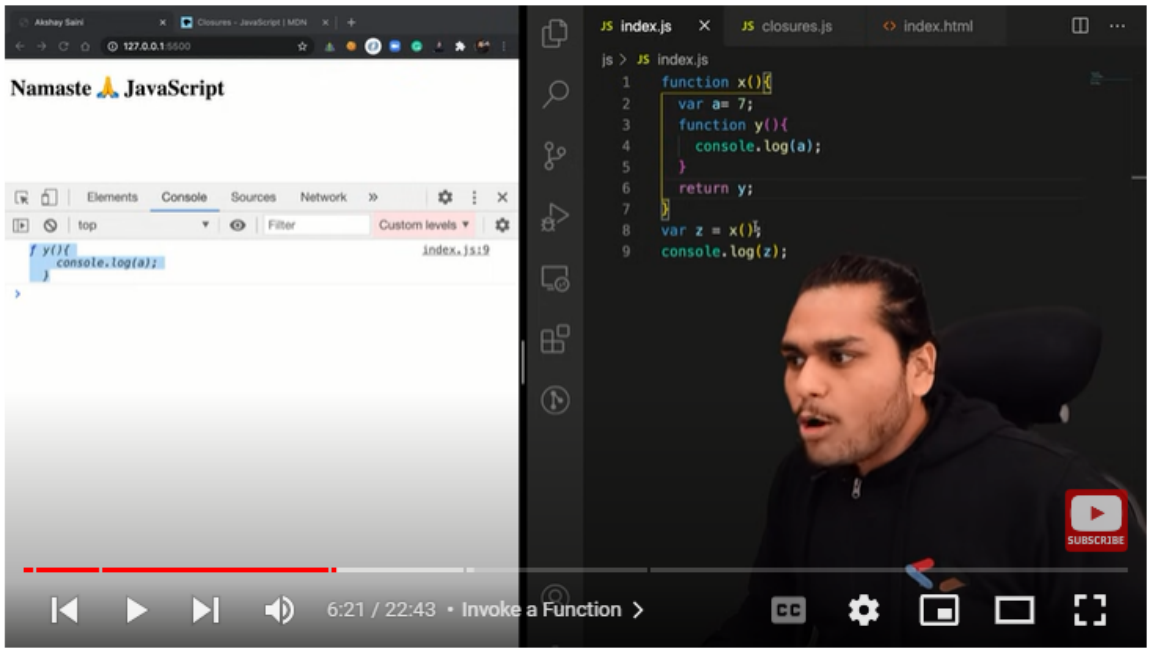
Note :- Functions are very beautiful in javascript. We can assign functions to a variable. We can also pass a function as a parameter to other functions, most programming languages will not allow this but javascript but this is the beauty of

javascript and how beautiful functions are in javascripts. We can even return a function from other functions.

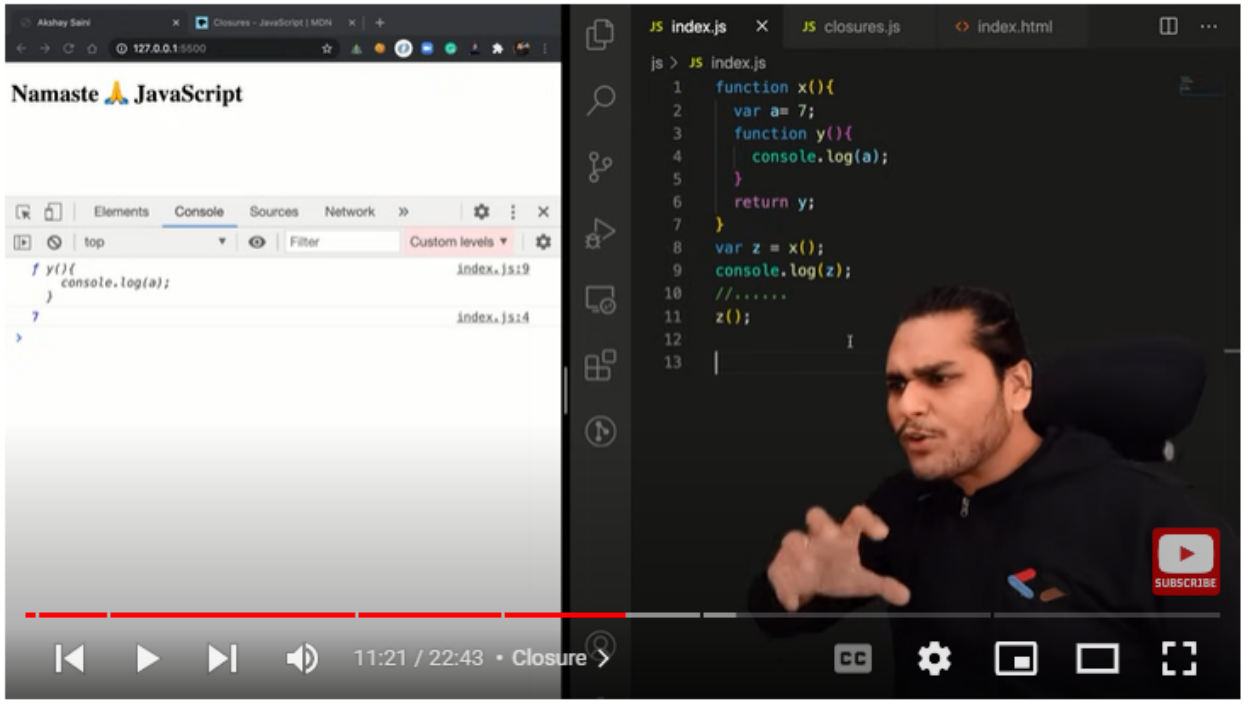


2. **Is returning a function from other functions allowed in javascript ? What happens if we return a function from other functions in javascript ? Explain with reference to closure.**
   - Yes , we can return a function from other functions. See the below code example. In this we have written function y which is inside the function x and when we return the function y which is inside function x, the whole function y will be returned.

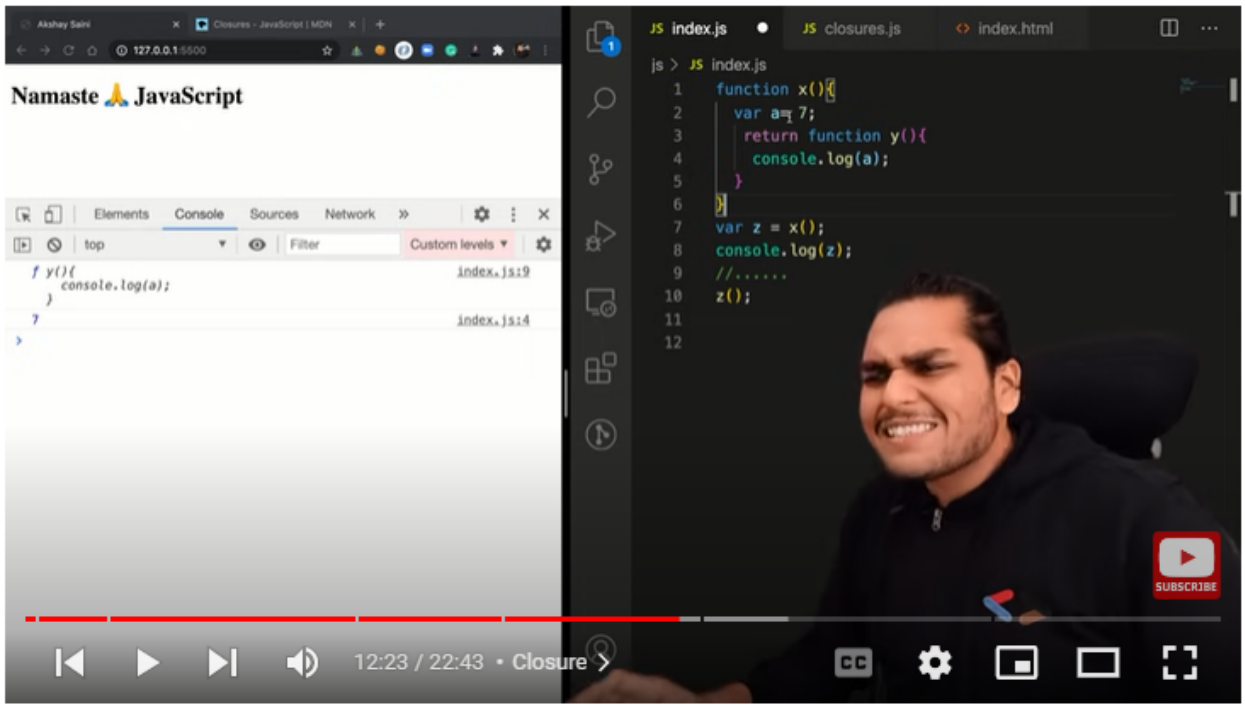
-


-


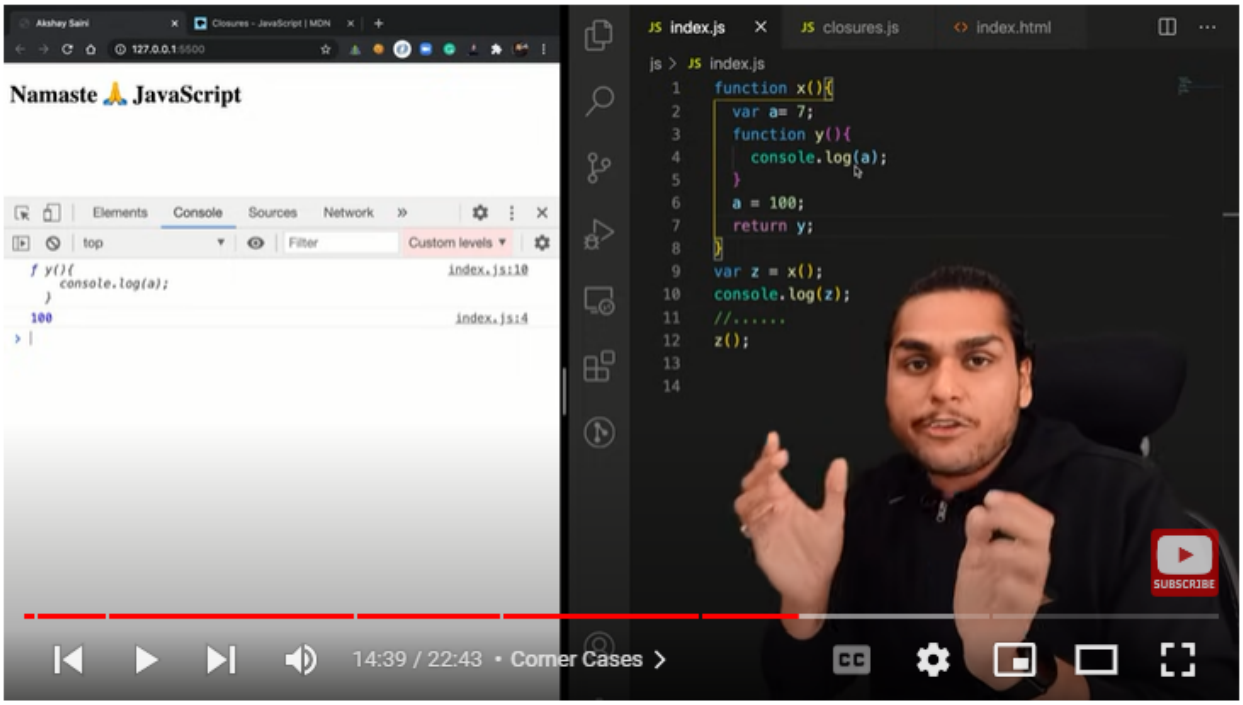
- The return function y value is returned to the line 8 from where it was called and the value can be collected inside a **variable z**, if we log to the console the value of variable z then it will give the entire function y, as we can see in the above picture.
- We have called the function x on line 8 and when a function is called, it creates a brand new execution context and so many things happen, we have already seen right ? and when the function x returns the function y then the function x will vanish and its execution context will no longer be inside the call stack and all its variables and function are completely gone.
- The variable z contains the function y, now here things become complicated because here we have returned a function. We already know what happens when the function y is inside the function x . Here, the function y is returned outside the function x, now how will this behave outside the scope of function x. Now if we try to call z() and function y is stored inside variable z then it will try to find **variable a** but as **a** was inside function x and function x is no longer inside call stack what it will print ? will it print undefined or null or some error ? **To our surprise it will print 7 and here where closure comes into the picture**. Even though function x does not exist but still function y remembers its lexical scope from where it comes from. So when the function y was returned not only just the function code was returned but the closure was returned i.e function along with its lexical environment was returned and it was put inside **z**
-



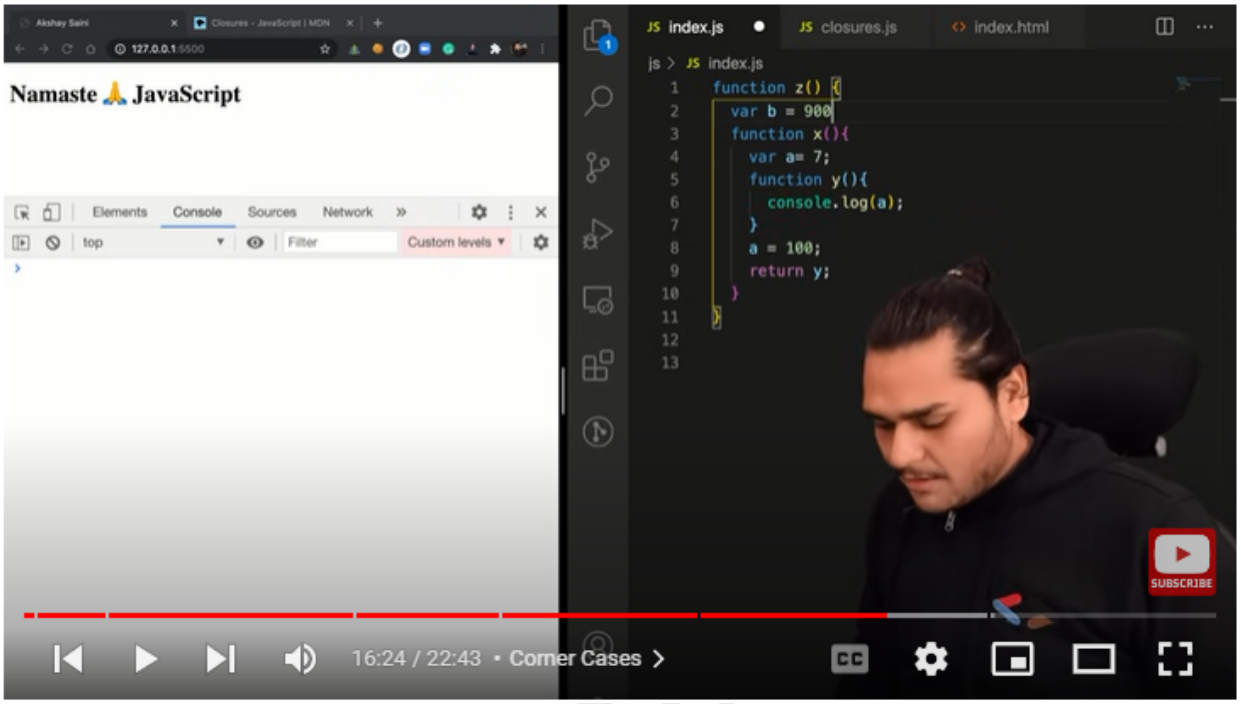
- **Few developers choose to return functions as shown below.**
-



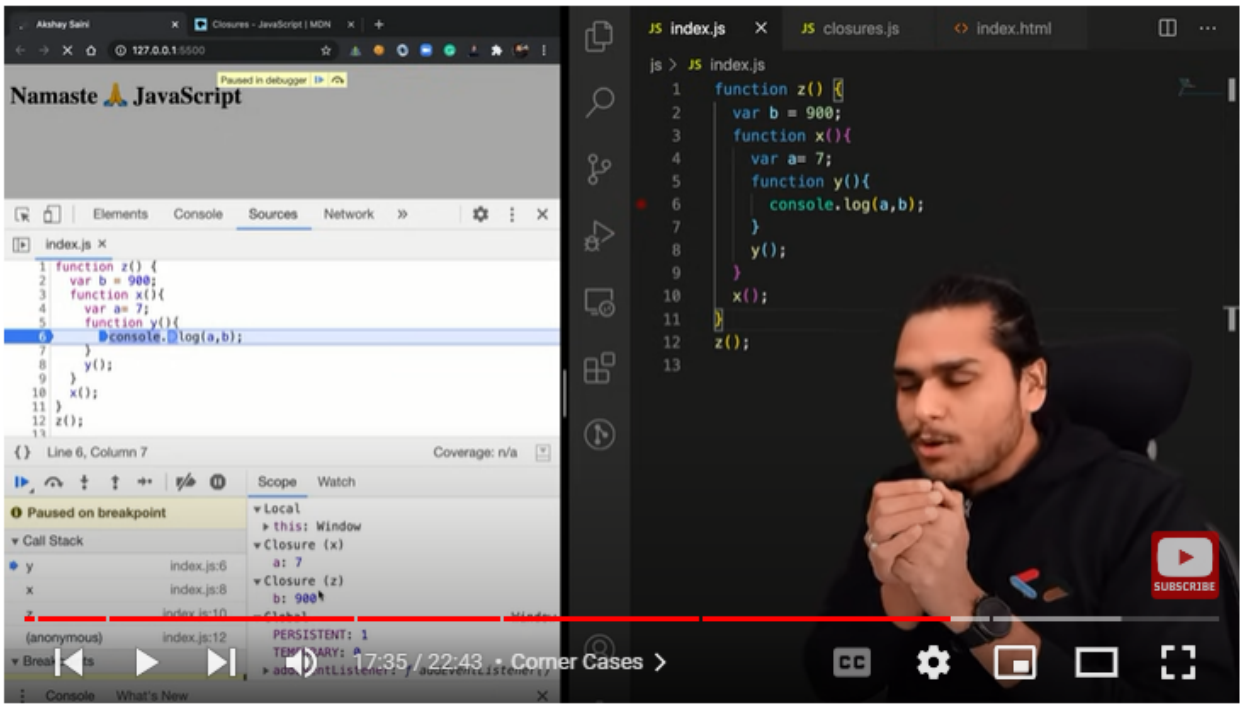
3. **What if we change the value of a = 100 before returning function y. What will it print then ? Will the execution of z() be 100 or 7 ?**
    - **The execution of z() will print 100 because when we return the function y it comes up with its lexical environment and inside that lexical environment the value of var a does not get returned but it is the reference to the var a which will get returned with the lexical environment. Therefore any changes to the var a before the return statement will get reflected when we execute z().**
    - **Therefore in closure the function does not remember the value of the variables, it remembers the reference to the variables i.e the reference to the memory location.**
    -



4. **What happens if we try to access variables from the deepest function ?**

- 

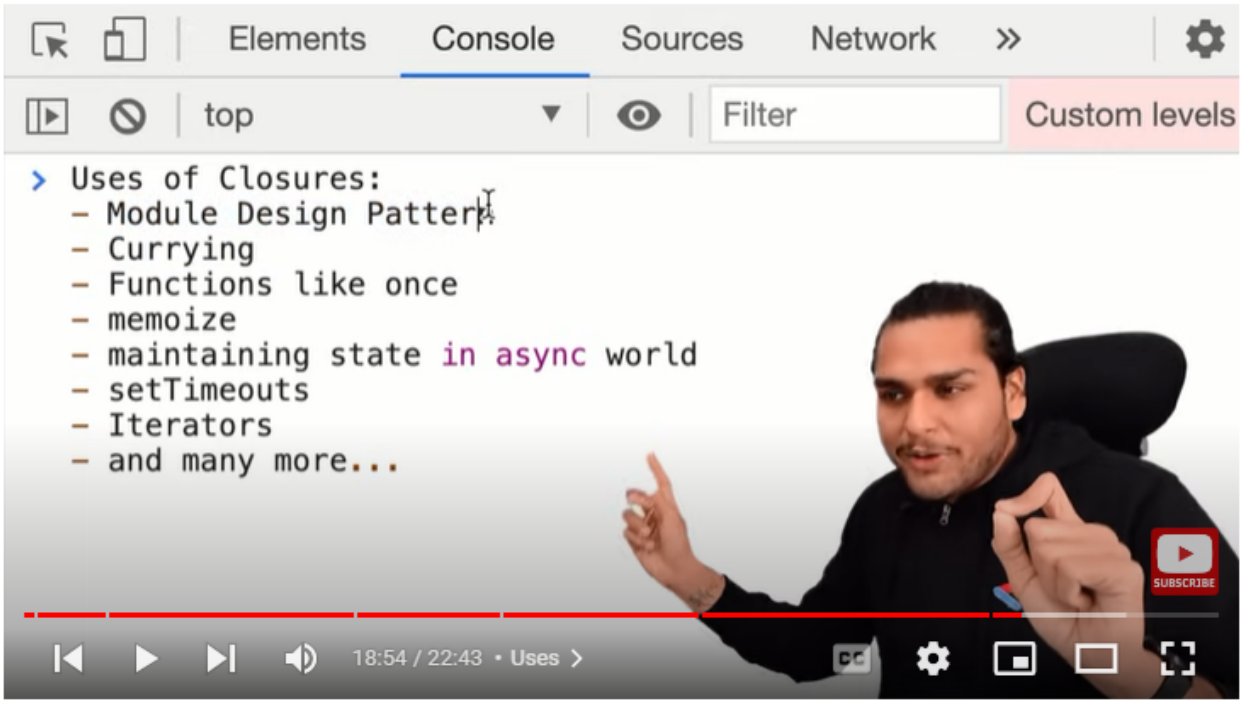
- **The innermost function y will form a closure with its lexical environment i.e with the function x lexical environment and with function z lexical environment. So we can remember the reference to the var a and var b if the function y was returned outside.**

- 

- **Closure makes the function remember its lexical environment even though the function is returned to the outer scope.**

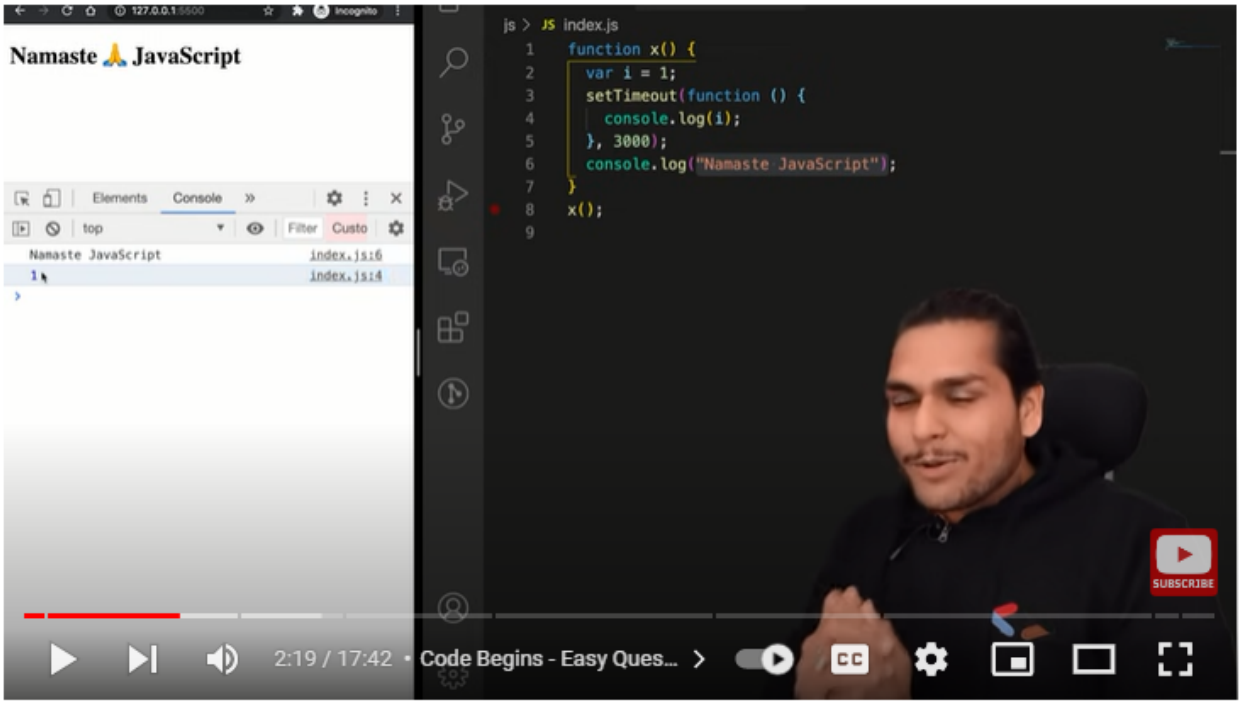
5. **What is the use of closures ?**

-



Episode 11 :- setTimeout + Closures Interview Question 🔥

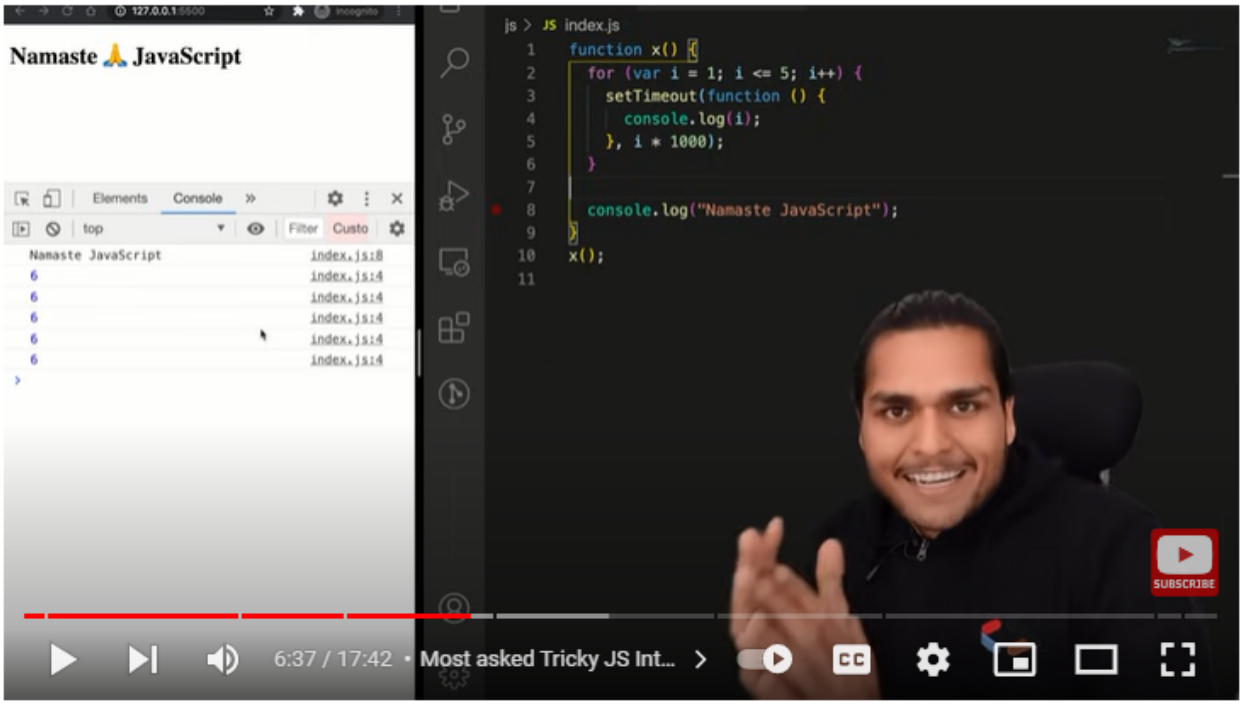
1. **What is the output of the below code ? Explain it**



-
- **It gives the output shown in the above image.**
- **The function inside the setTimeout forms a closure. Therefore, the function will remember the reference to i and form a closure, so wherever the function goes it takes the reference of i with it.**
- **The setTimeout will take the callback function and attach a timer of 3000ms to it and store it at some place. Now, the javascript will not wait for the timer to expire and it continues to the next line and prints “Namaste javascript”. Now, when the timer of 3000ms expires it will again take the function and put it in the call stack and run it and now, console.log(i) will get executed .**

2. **Print 1,2,3,4… after 1s,2s,3s,4s…**

3. **Why the below code behaves the way it is behaving. Give a solution to the fix the problem the below code is facing.**



Note :- Rewatch Episode 11 for revision and if you get any doubts solving this problem

4. **Fix the problem in the above code without using let**

Episode 12 :- Crazy JS interview - Rewatch the video for revision and clearing concepts on closures.

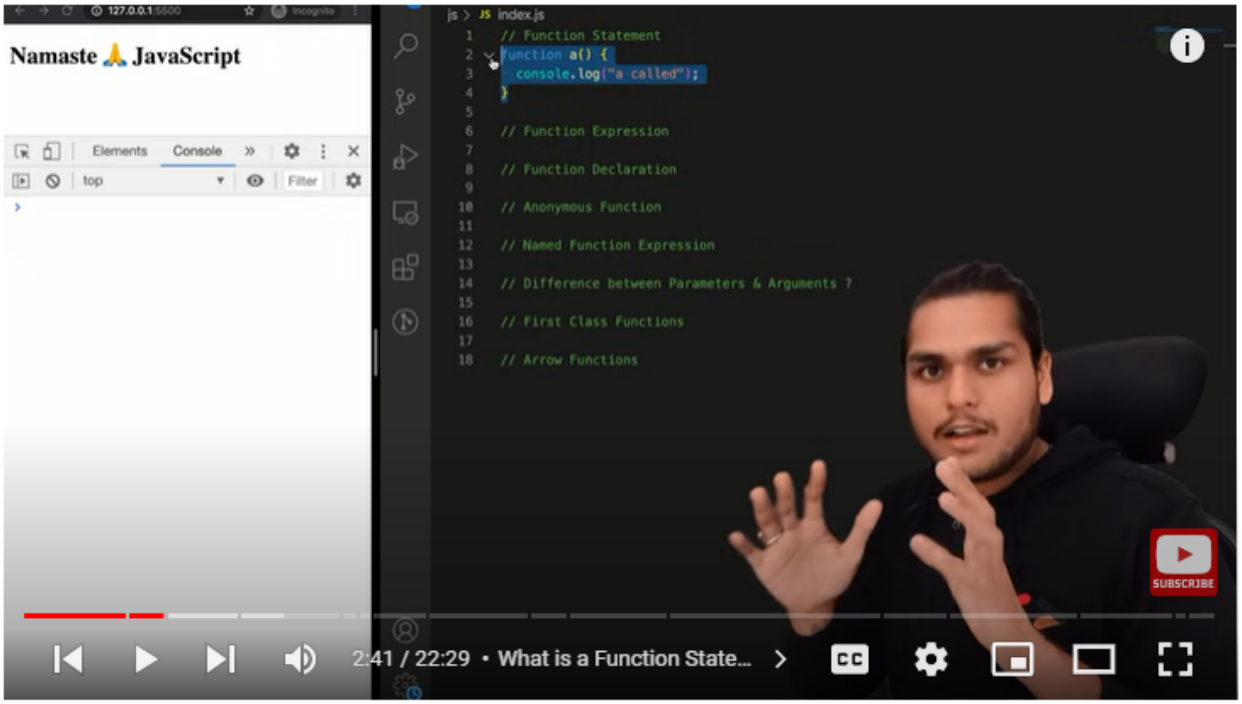
Question in Episode 13: FIRST CLASS FUNCTIONS 🔥ft. Anonymous Functions

1. **What is an anonymous function ?**
    - **A function without any name is an anonymous function.**

2. **What is a function statement ?**

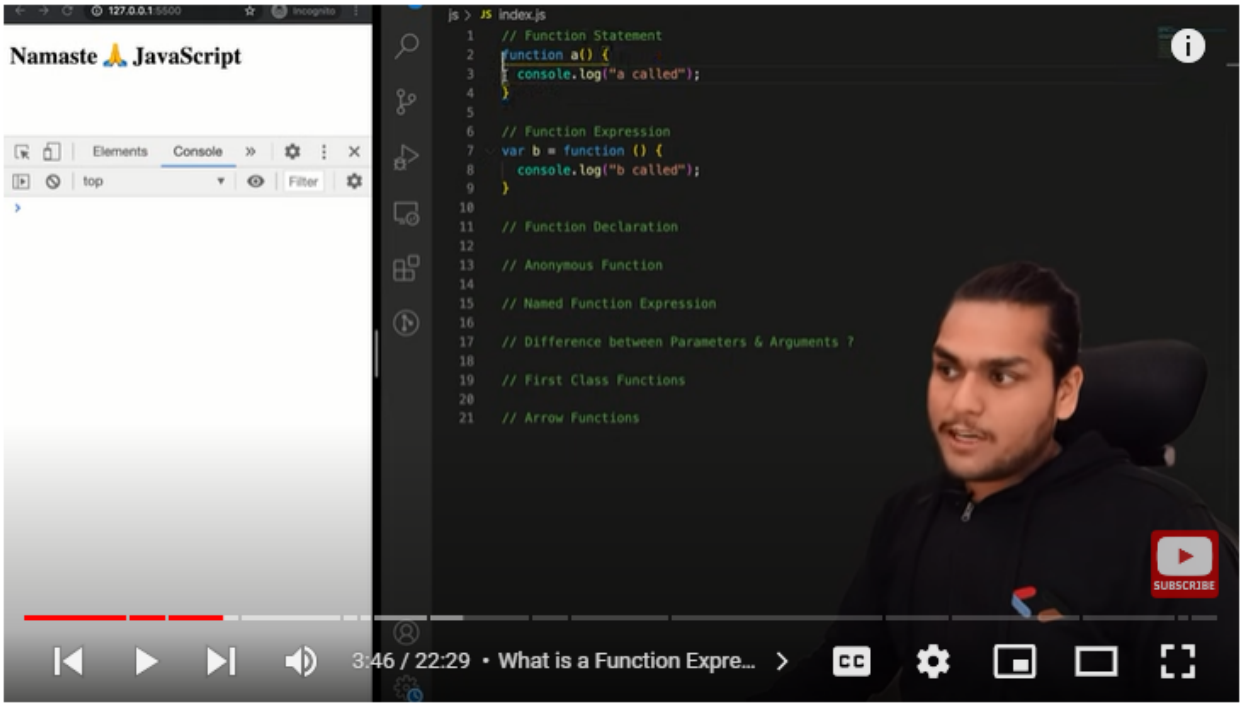
- **The normal way of creating a function is known as a function statement.**

-



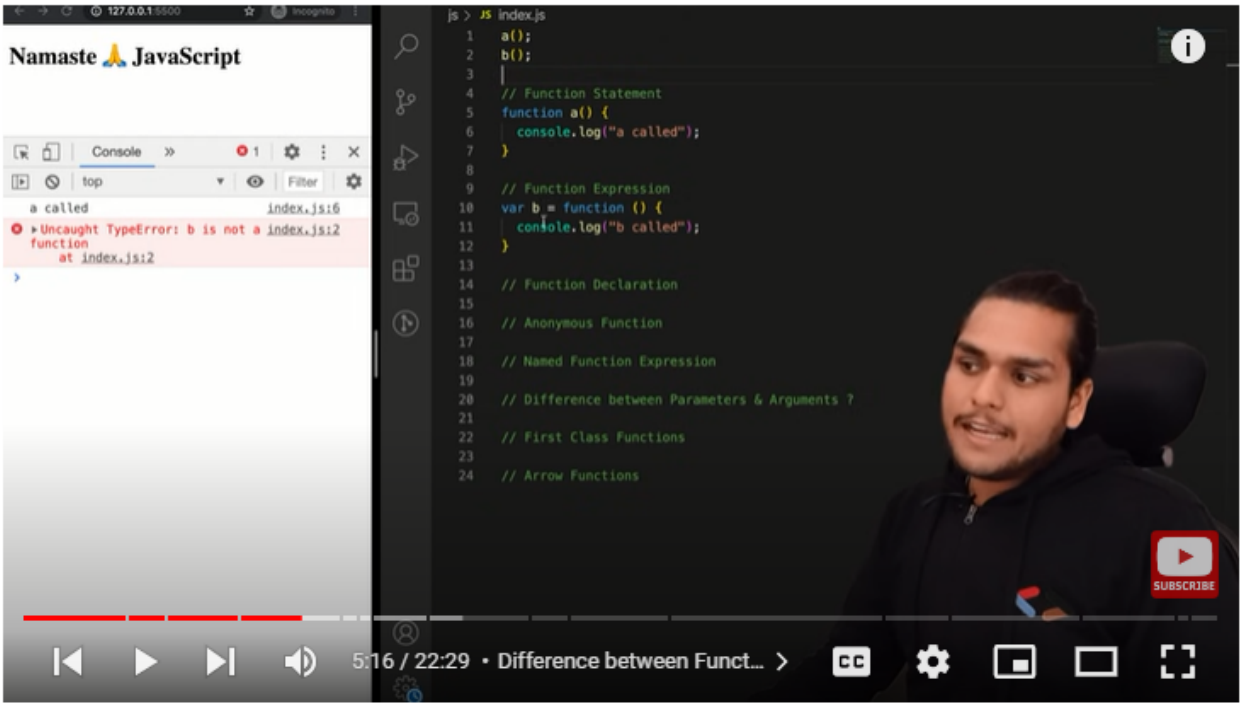
3. **What is a function expression ?**
    - **Functions are very beautiful in javascript, functions are basically the heart of javascript. One of the beautiful features of the function is that we can assign a function to a variable also, in javascript functions act as a value. Therefore, if we assign an anonymous function to a variable then it is called function expression.**

-



4. **Difference between function statement and function expression**

- **The major difference between function statement and function expression is hoisting.**
- **If we call the function created using function statement and function created using function expression before they are declared in the program then the function created using function expression will throw an error because of hoisting.**



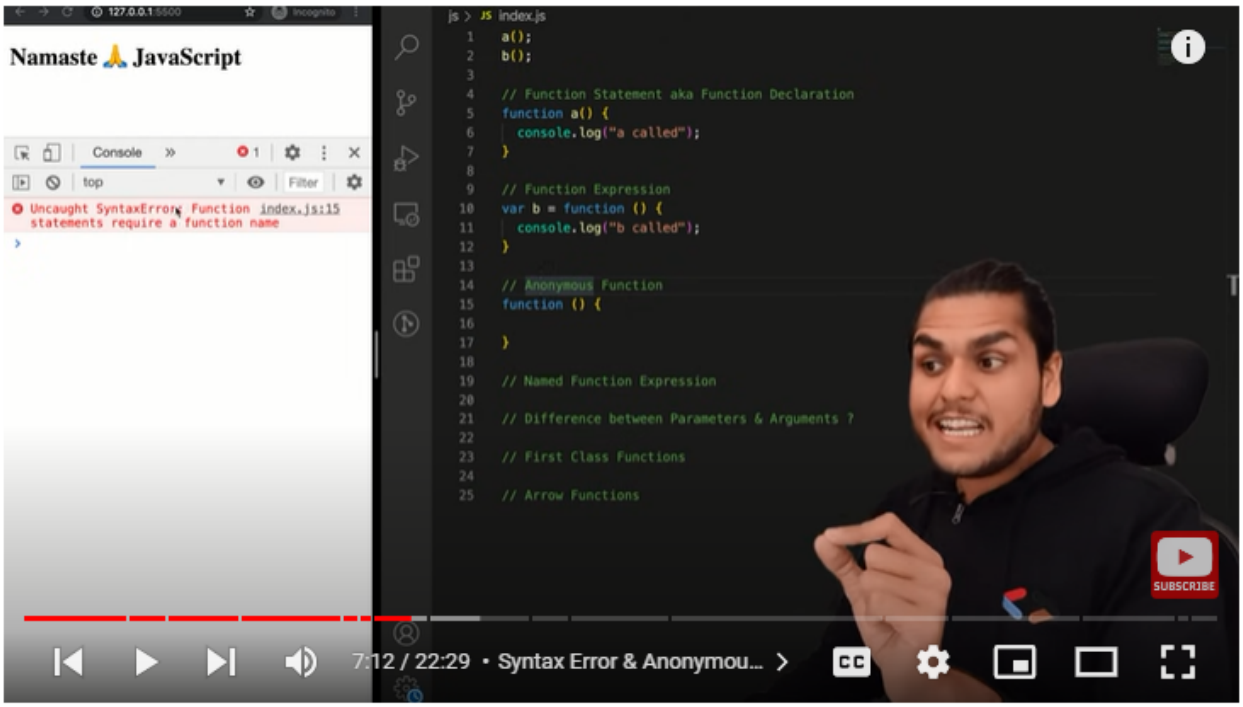
-

- **Incase of the function statement during the memory creation phase the whole code of the function is store inside the memory and incase of the function expression, it will be treated as variable and "undefined" is stored inside the memory allocated to the variable during the memory creation phase and if we try to call function expression before the declaration than it will throw an error.**
- **So, this is the basic difference between function expression and function statement.**

5. **What is a function declaration ?**

- **Function declaration is the same as the function statement. Function statement or Function declaration these both are the same thing.**

6. **What is an Anonymous function ?**

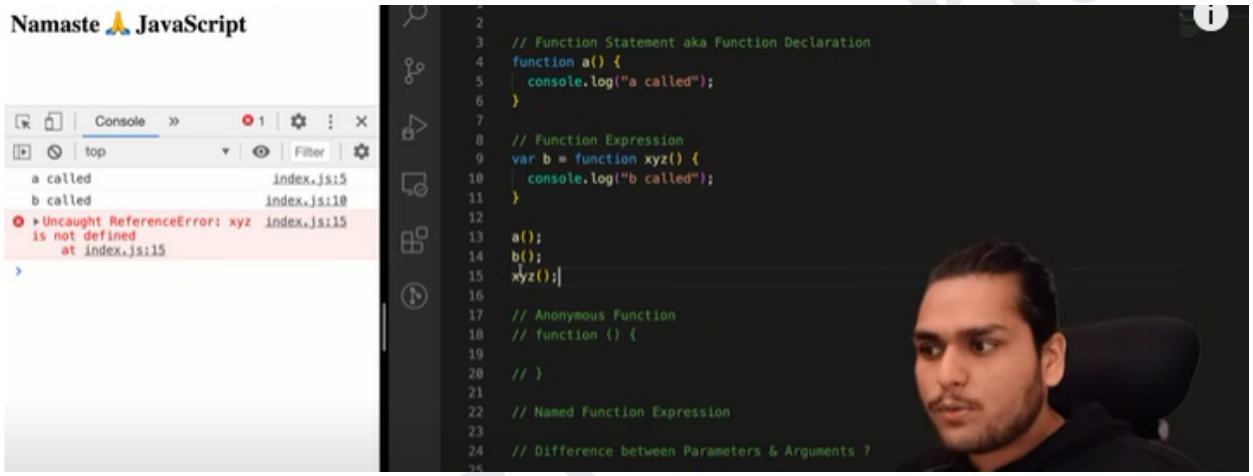
- **An anonymous function is a function without any name. It does not have its own identity means if we create an anonymous function as shown in the image below then it will give a syntaxError.**
- **An anonymous function looks like a function statement but without any name but according to the ecmascript specification a function statement should have a name. Therefore, it is an invalid syntax.**
- 

- **The error message clearly says that the function statement requires a function name, so if we can't use anonymous functions like this then what is the use case of the anonymous function ?**
- **The anonymous functions are used in a place where functions are used as values. Therefore, we can use anonymous function to assign it to some variable, just like we can assign any other value to a variable we can also assign an anonymous function to a variable.**
-

7. **What is a named function expression ?**
   - **If we assign a function with a name to a variable then it is known as a named function expression.**

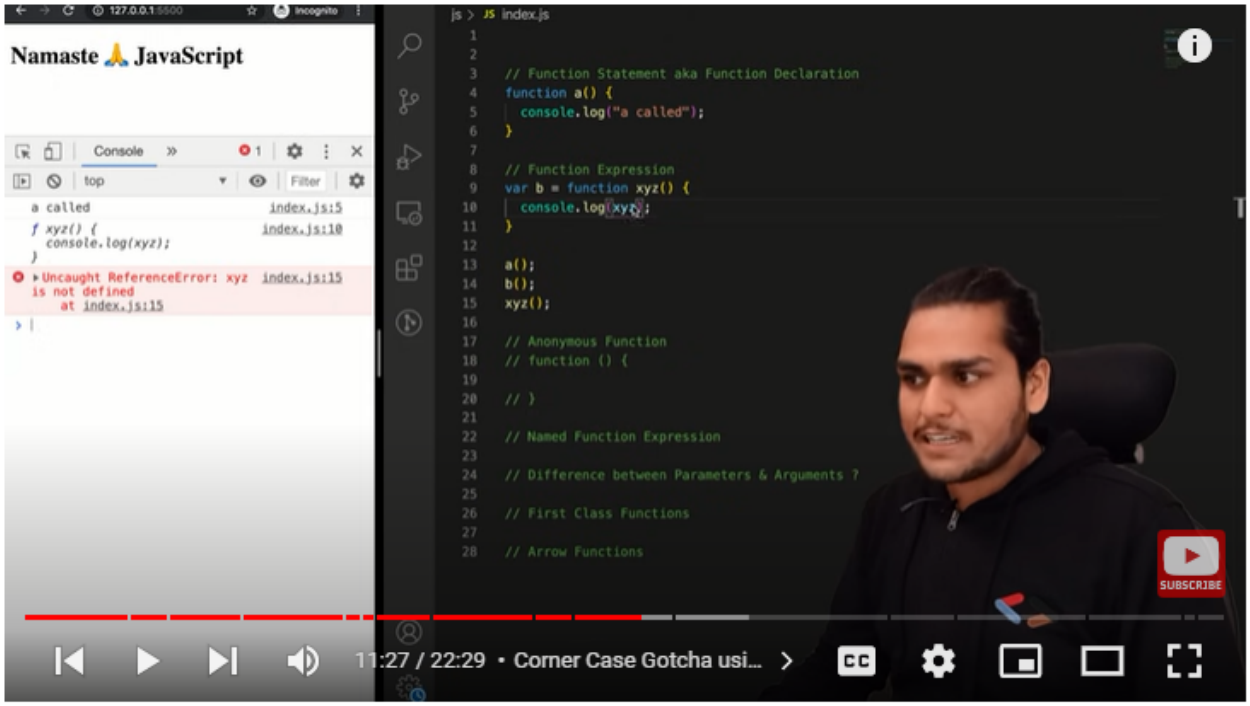
```
// Named function expression
var a = function xyz(){
    console.log("Namaste javascript");
}
a();
```

   - **Here, the named function expression has two names one is a and the other is xyz, now if we call this named function expression with a(); than it works fine but if we call it with xyz() than it will give an error.**



   - **If we use the function xyz as a value, so in this case the function xyz will not be created inside the outer scope but it is created as the local variable. Therefore, if we want to access function xyz inside xyz, we can access it but in the outer scope it will throw an error.**

-



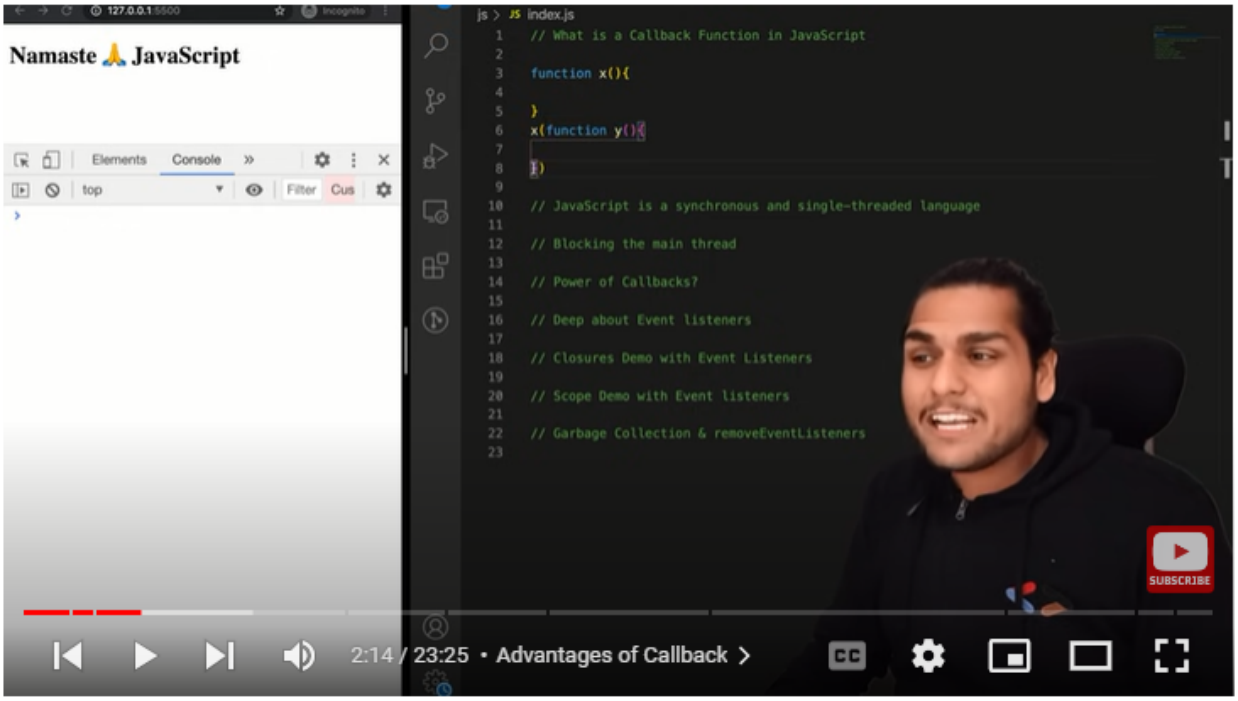
8. **Difference between parameters and arguments.**
   - **The values which we pass when we call the function are known as the arguments and the identifiers or labels that we pass during the function declaration in which the arguments are stored are known as the function parameters**

9. **What are first class functions ?**
   - **The ability of functions to be used as value is known as first class function. Functions in javascript can be assigned to some variables, they can be passed as arguments to some other function and even they can be returned from a function.**
   - **Functions used as first class citizens are the same thing that the functions are first class functions in javascript.**

Question in Episode 14 :- Callback Functions in JS ft. Event Listeners 🔥

1. **What is a callback function in javascript ?**
    - **We know that functions are first class citizens in javascript. Therefore, we can pass a function into another function. So, the function passed into another function is known as a callback function. Callback functions are so powerful in javascript it gives access to the whole asynchronous world in a synchronous single threaded language. Javascript is a synchronous single threaded language but due to callback we can do async things in javascript as well.**
2. **Why it is known as a callback function ?**

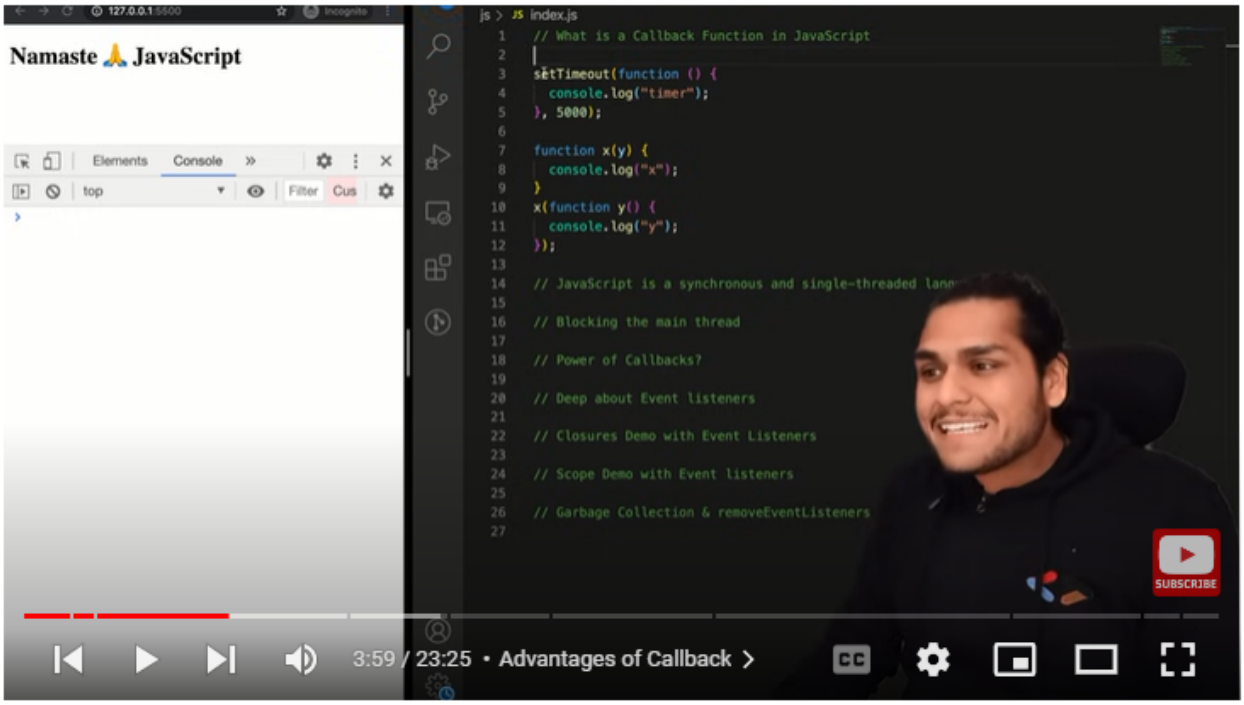
- **Let's take an example**
- **We have created a function x and when we call the function x, we are passing another function y into it. The function y is known as the callback function.**
-



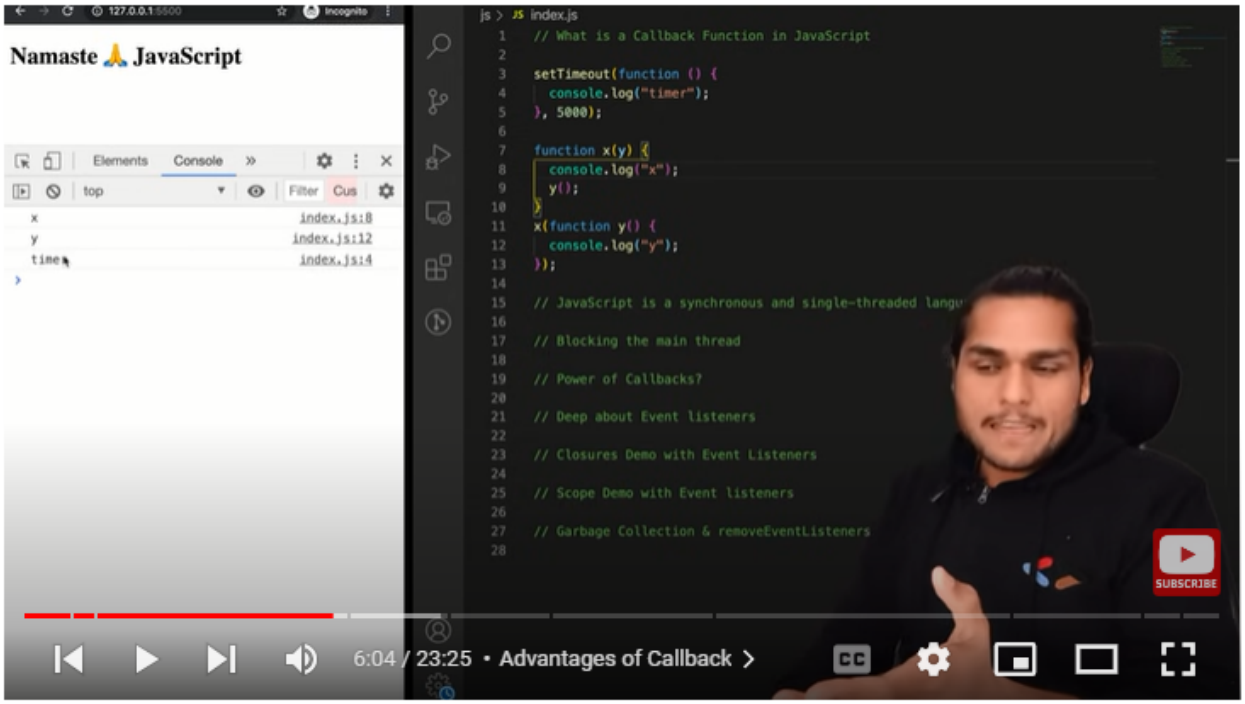
- **It is known as a callback function because it is called back sometimes later in the code. Here, we have passed the function y into function x, so now it's the responsibility of function x to call the function y sometimes later in the code.**

3. **How callback functions are used in asynchronous tasks ?**

- **Let's take an example of setTimeout**
- **setTimeout takes a callback function as the first parameter and to the second parameter it takes time in milliseconds. So, in setTimeout the callback function is called after the timer attached to the functions expires**
- **The function which we have passed to the setTimeout as the first parameter is the callback function and it is called after sometimes in the program. It is called back after some time which we have passed as the second parameter. The timer which we have passed as the second parameter is in the milliseconds.**



- **What output does this above program give ?**
- **As we know that javascript is synchronous single threaded that means code is executed one line at a time and in a specific order**
- **The first thing that will happen is registering the setTimeout, so setTimeout will take the function and it will store at separate space and it will attach a timer of 5000ms to it. Now, javascript will not wait for 5000ms to expire and execute the callback function given to the setTimeout instead javascript will move to the next line and continue executing the code on the next lines. This is how callback functions gives the power of asynchronicity as it does not wait for 5000 ms to expire.**

- **Whatever is needed to be done after 5000ms will be passed as the callback function inside the setTimeout so that it can be later executed in the program.**
- **Now, javascript moves to the next line of code and it will call the function x which is taking function y as the call back function and x will be printed on the console.**
- 

- **Now, inside the function x , the call back function y is called and then y is printed to the console. Now, after 5000ms expires the call back function passed as the first parameter of setTimeout will be called and whatever is inside that call back function will be printed to the console. So this how call back functions help in doing asynchronous operation in javascript.**
- **Demo in the browser is important (timestamp :- 6:00 to 8:25).**

4. **Blocking the main thread**
    - **Javascript has one call stack and we can call it as a main thread so whatever is executed inside the page is executed through the call stack only. So whatever function we executed in the above question function x , function y and the callback function will be executed through callback only. So if any operations block this call stack is known as blocking the main thread.**
    - **Suppose a function x has a very big operation which takes almost 20 to 30 seconds to complete executing that function so during that 20 to 30 seconds javascript will not be able to execute any other function because it has just one main thread or call stack, so everything in the code will be blocked. Hence, we should never block our main thread. Therefore, we should use async operation for the things which takes time like we have used setTimeout in the example given in the 3rd question**
    - **So, if javascript doesn't have the first class function i.e if there are no callback functions in javascript then we will not be able to do the async operations.**
    - **So, with the help of this webAPI setTimeout and callback functions we are able to achieve asynchronous operations in javascript.**

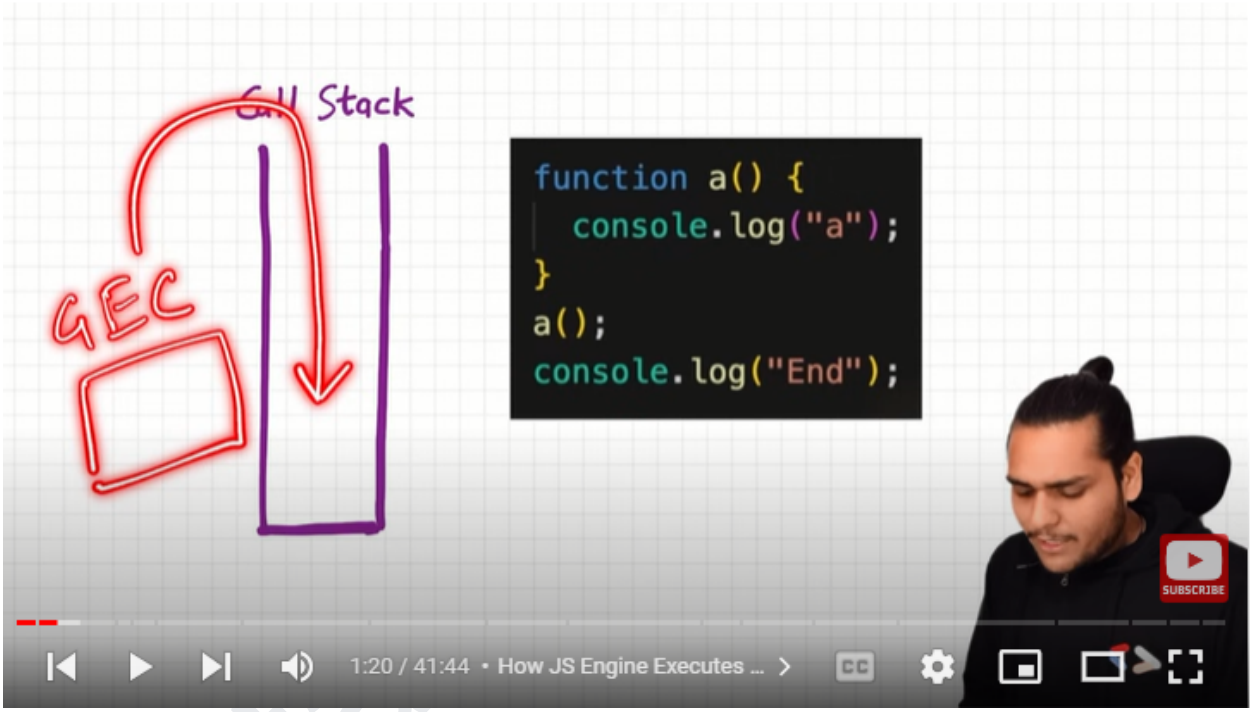
5. **Event listeners in javascript and closures along with event listeners.**
- **Watch timestamp (10:18 to 19:25) for revision.**
- **Explain callback with event listener**
- **Explain closures with event listeners**

6. **Garbage collection and removeEventListeners**
    - **Revision from video**
    - **Why do we need to remove event listeners ?**

Questions in Episode 15 :- Asynchronous JavaScript & EVENT LOOP from scratch 🔥

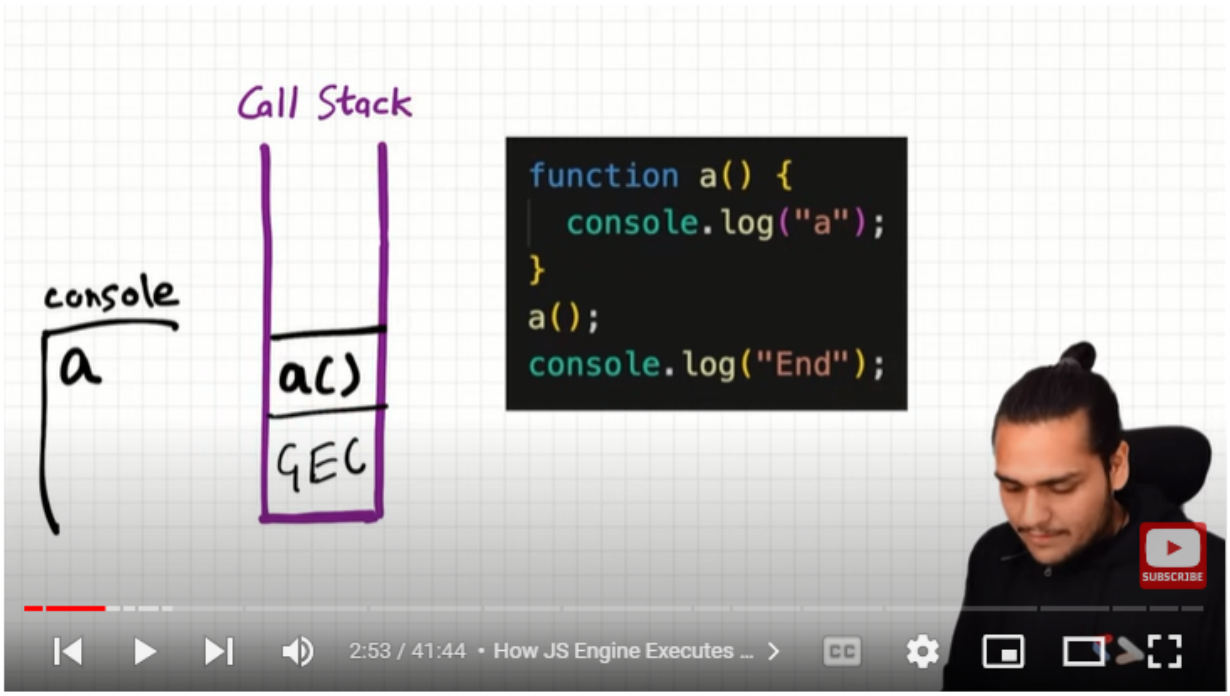
1. **Javascript is a synchronous single threaded language. It has one call stack and it can do one thing at a time. The call stack is present inside the javascript engine and all the code of javascript is executed inside the call stack.**

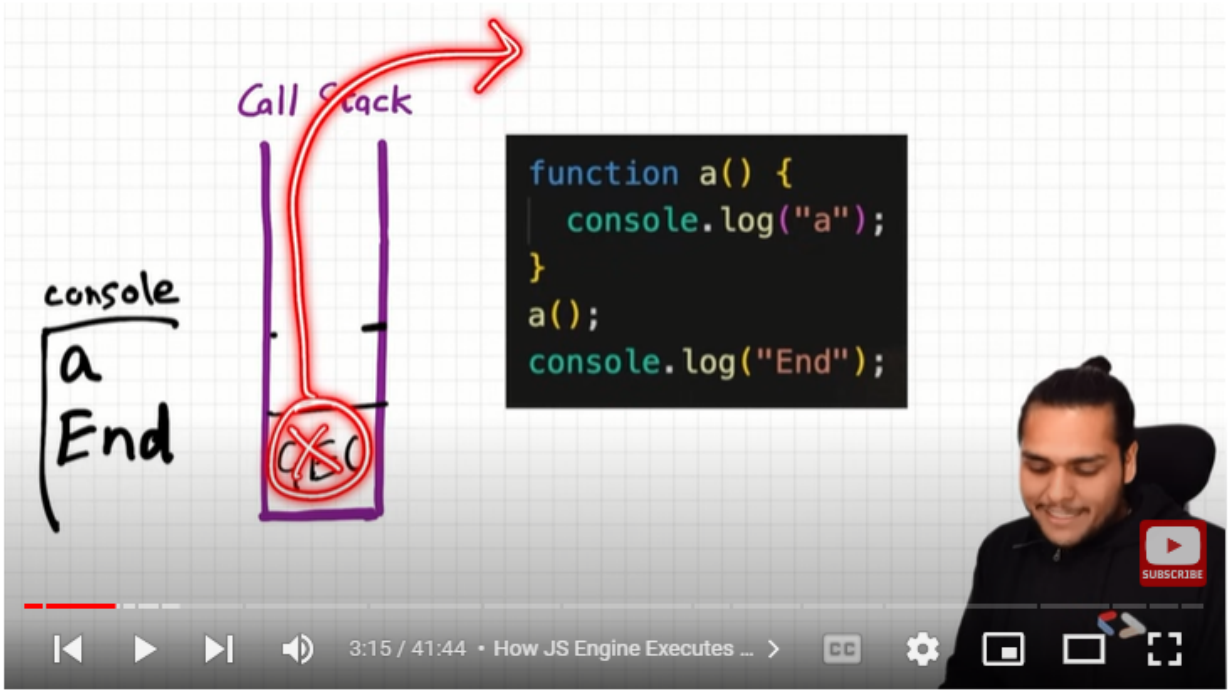
2. **Whenever a program is run in javascript a global execution context is created and it is pushed into the call stack.**



- **Now, the whole code will run line by line inside the global execution context. For a function call also an execution context is created and pushed into the call stack and the code inside that function will be run line by line in the execution context.**
- **After the function is executed its execution context is popped from the stack and after the last line in the above program is executed , the global execution context is also popped from the stack.**

-



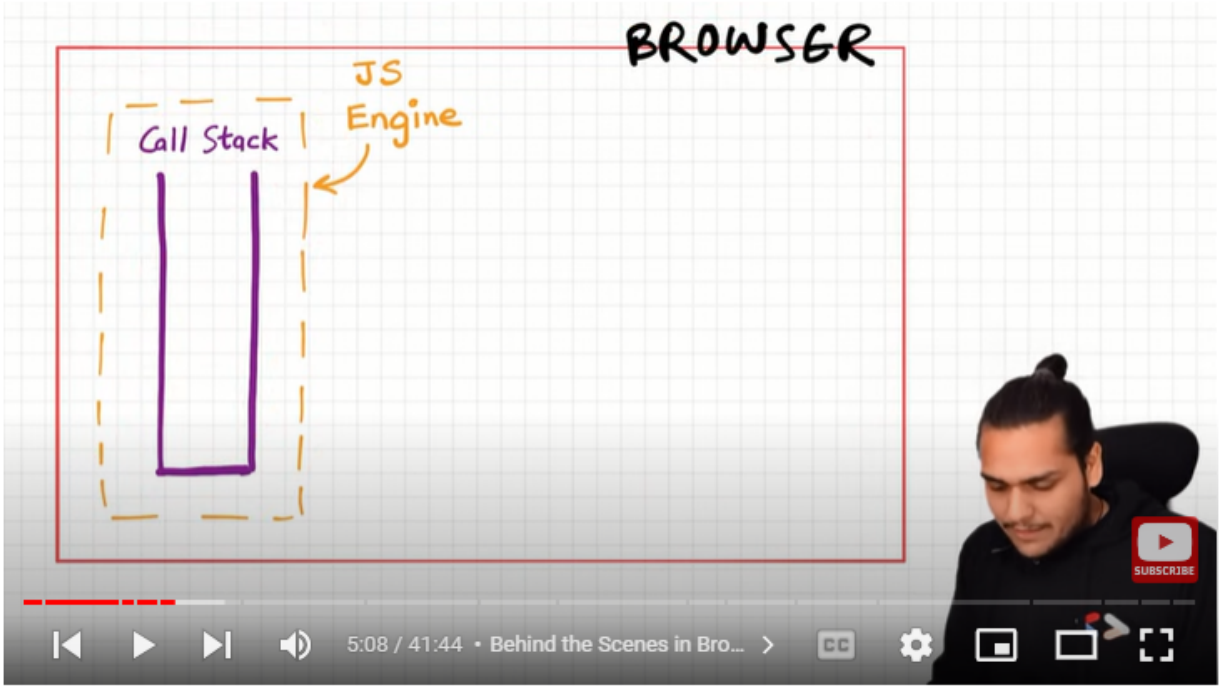


So, this is how the javascript engine executes the entire javascript program

Note :- The main job of the call stack is to execute whatever comes inside it. It does not wait for anything, it just quickly executes whatever is given to it.

1. **Can we execute a piece of code or a script after a few seconds inside the call stack ?**
    - **No, we cannot execute a piece of code or a script after a few seconds inside the call stack because the call stack executes quickly whatever comes inside it and it cannot wait.**
    - **If we give a script or a code to the call stack and ask it to execute after a few seconds then it is not possible in the call stack because the call stack executes quickly whatever is given to it. Because call stack does not have timer**
    - **So, if we have to execute a piece of code after a few seconds then we need some extra super powers, the superpowers of timers**
2. **How do we get the superpowers of timer ?**

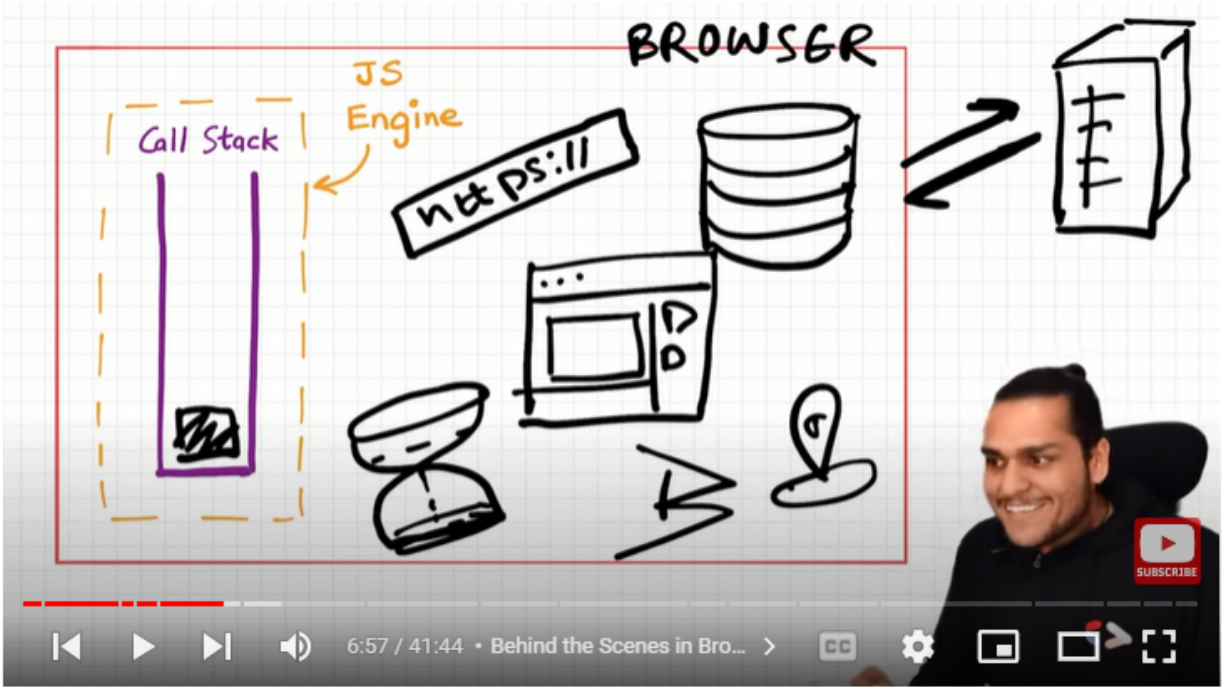
- **The call stack is inside the javascript engine and the javascript engine is inside the big red box which is the browser.**



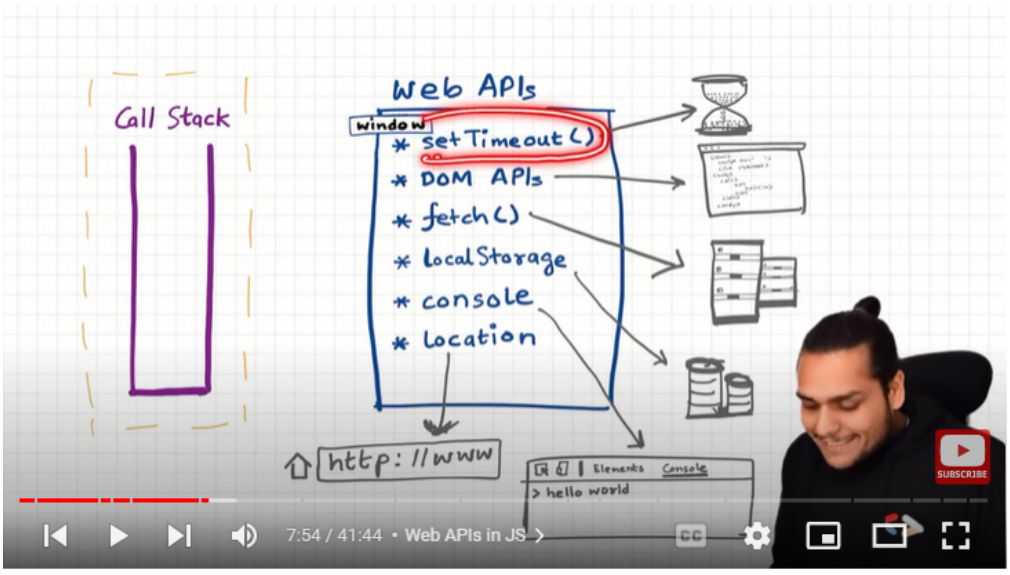
- **And inside the call stack our javascript code executes**

- **The browser has the javascript engine inside it, local storage, timer, it also has a url, the browser has the superpower of communicating to the external world like it**

can communicate to the netflix server to display the data on the website. Browsers also have access to the bluetooth, geolocation and various other things.

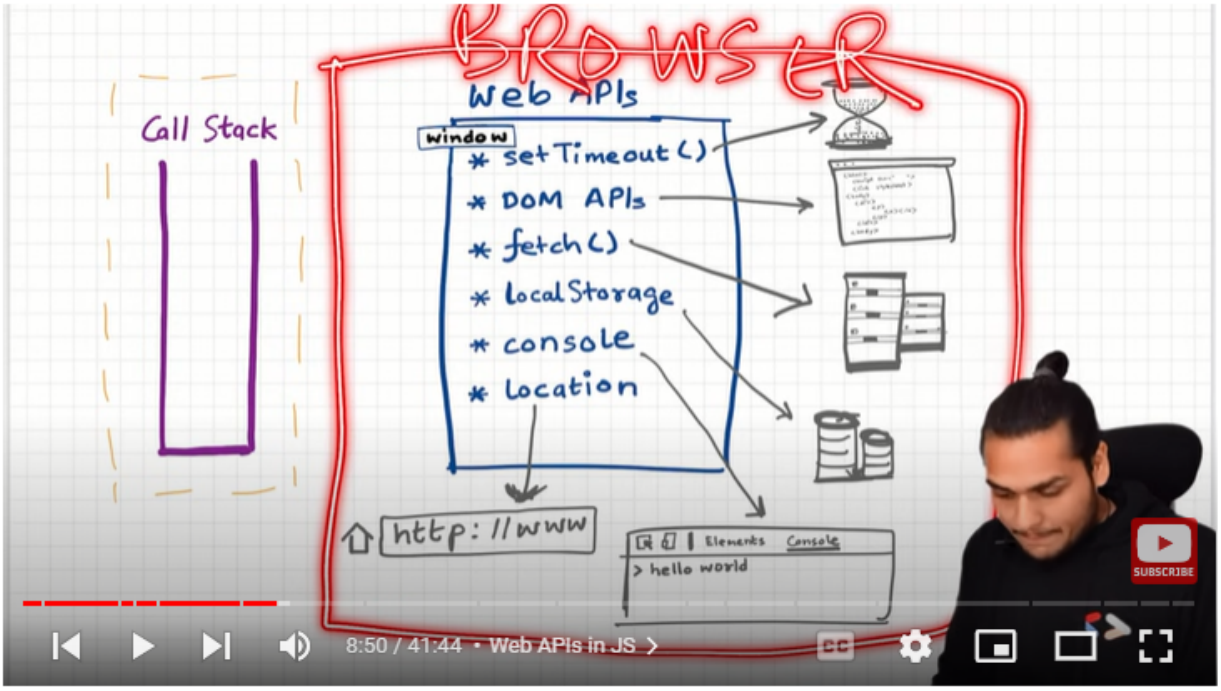


-
- **Therefore, the browser is the most remarkable creation in the history of mankind 😁 .**
- **Suppose, we need access to these superpowers in the code running inside the call stack. So, the javascript engine needs some way to access these powers. Let's see how we can do it ?**
- **To access all these superpowers we need web API's**



-

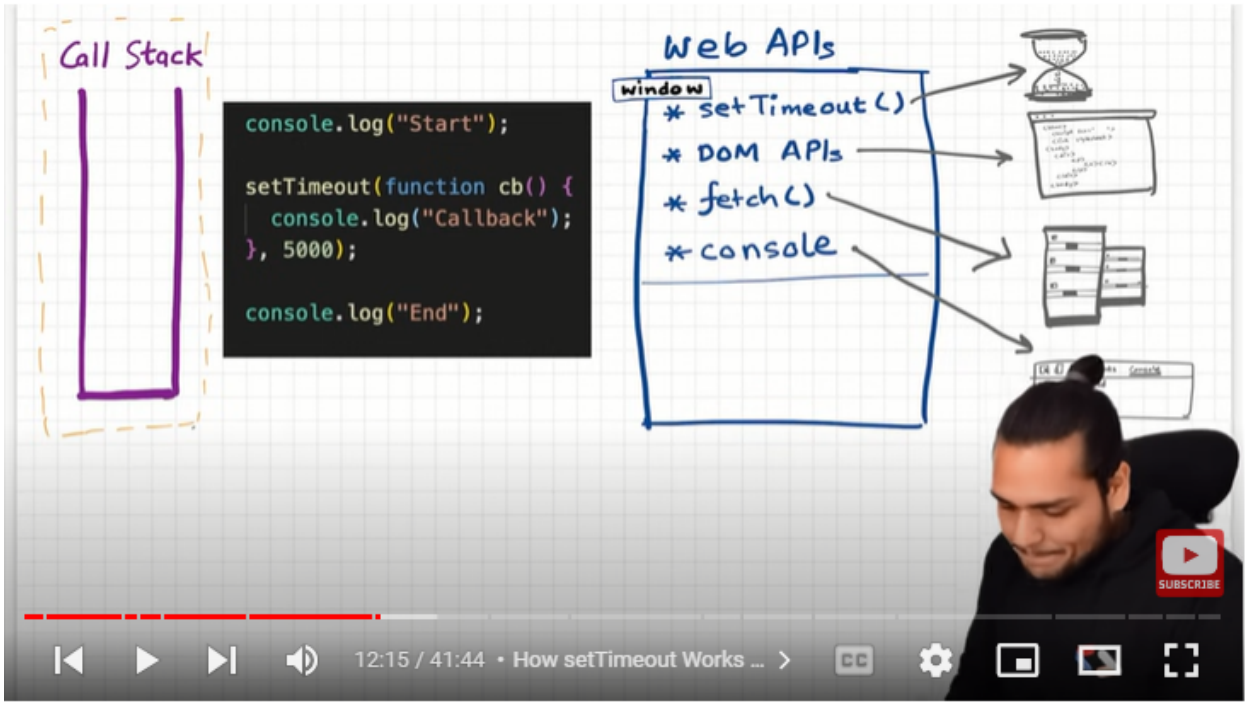
- **All these setTimeout, fetch(), localstorage, location, console and DOM APIs are not part of javascript**
- **All these web APIs are part of the browser**



- **Browser gives access to all these web APIs inside the call stack which is inside the javascript engine through the global object which is window incase of the browser**
- **Suppose if we want to access setTimeout inside our code which is executing inside the call stack then we need to do something like window.setTimeout(). If we do window.localStorage then it will give access to the local storage. This is how we can access web APIs inside the javascript code.**
- **But as window object is the global object we can access these web APIs inside our javascript code without the window keyword as well. Example :- we can just write setTimeout() instead of writing window.setTimeout().**
- **The browser wraps up all these web APIs into a single object known as the window object and gives the access of the window object to the call stack or to the javascript engine.**

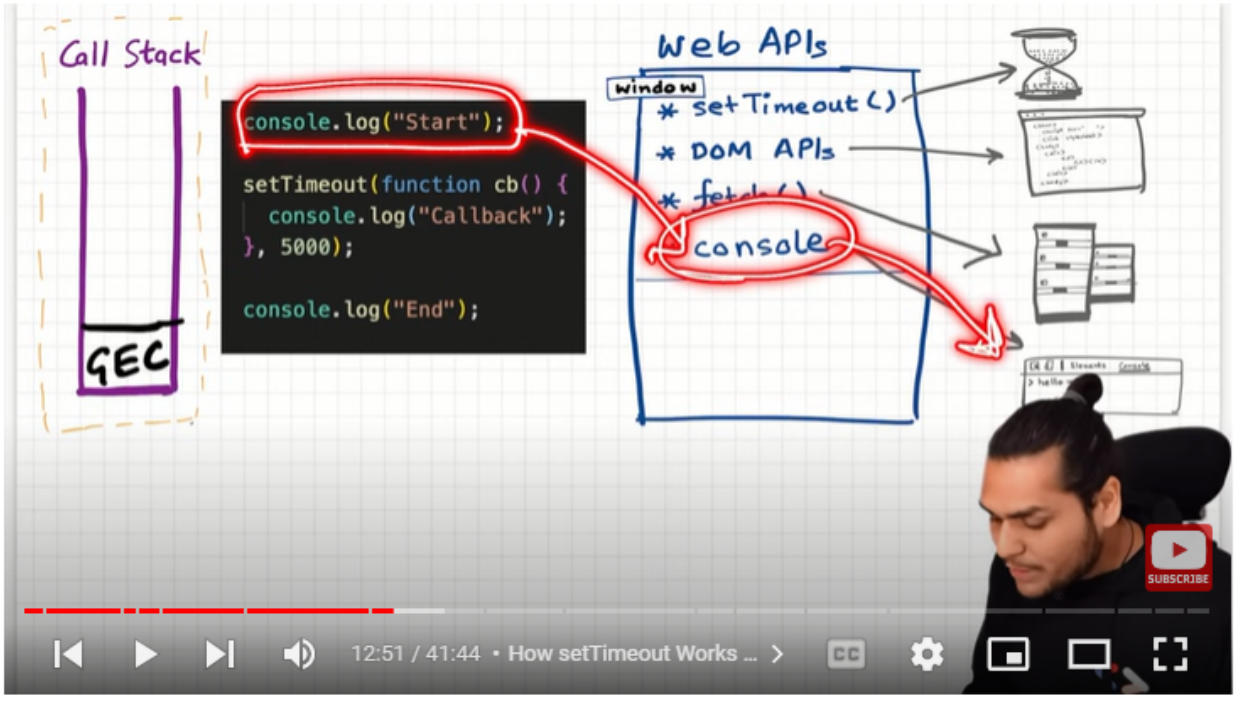
3. **Explain how the below javascript code executes behind the scenes ?**

-

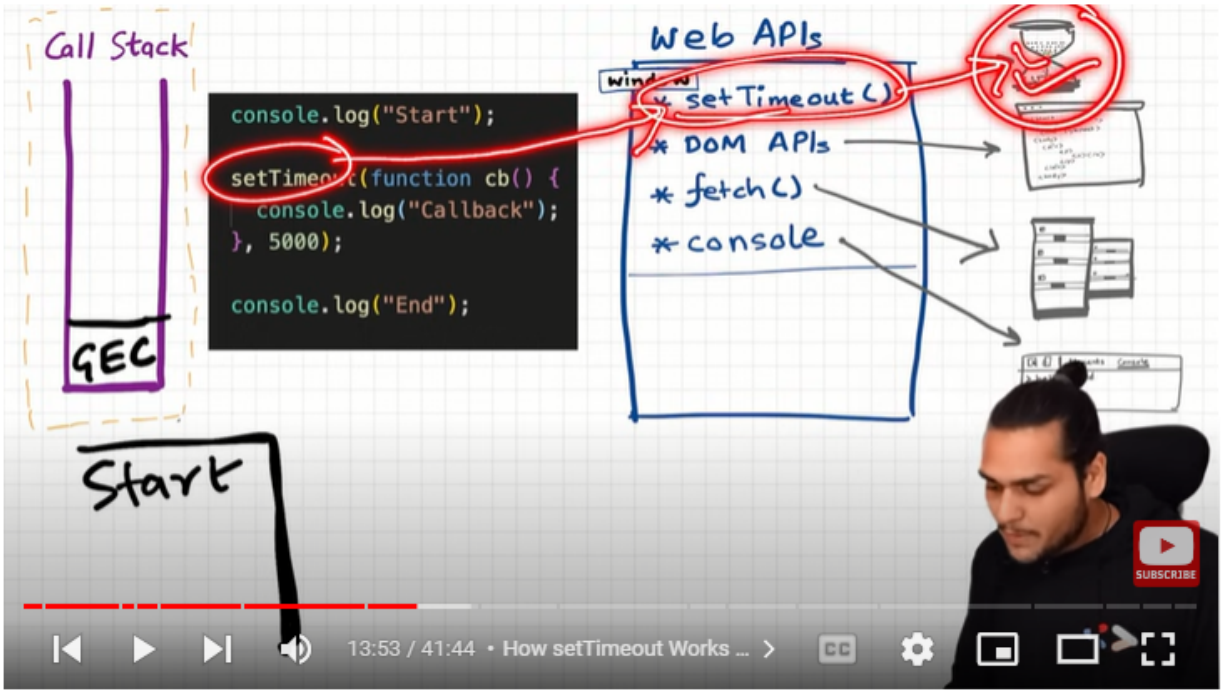


- **Whenever a javascript program is executed a global execution context is created and pushed to the call stack. Now the whole code will run line by line.**
- **At line 1 there is console.log("start"), so it will call the web API and then web API internally logs "start" to the console.**

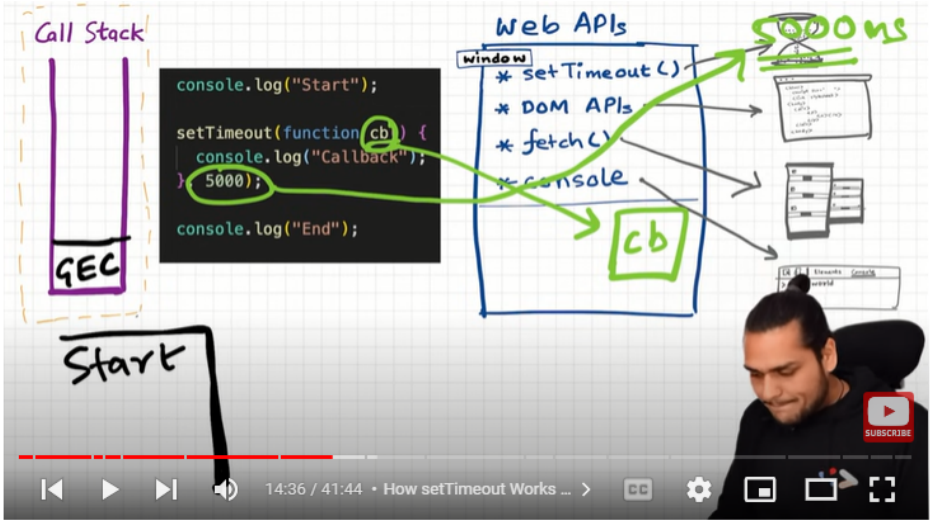
-



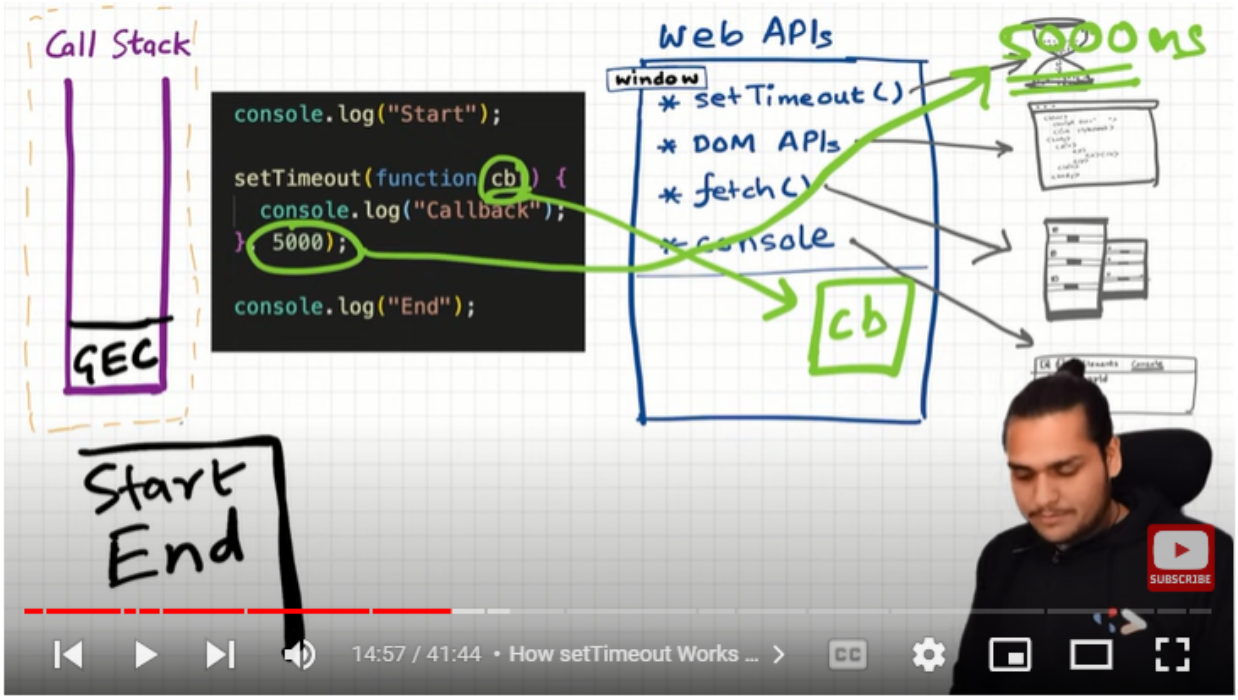
- **We are able to do console.log() because of the web API console and this web API is plugged to the javascript code through the window object.**
- **Now the execution will go to the next line. On this line it will call the setTimeout web API which will give access to the timer superpower of the browser**



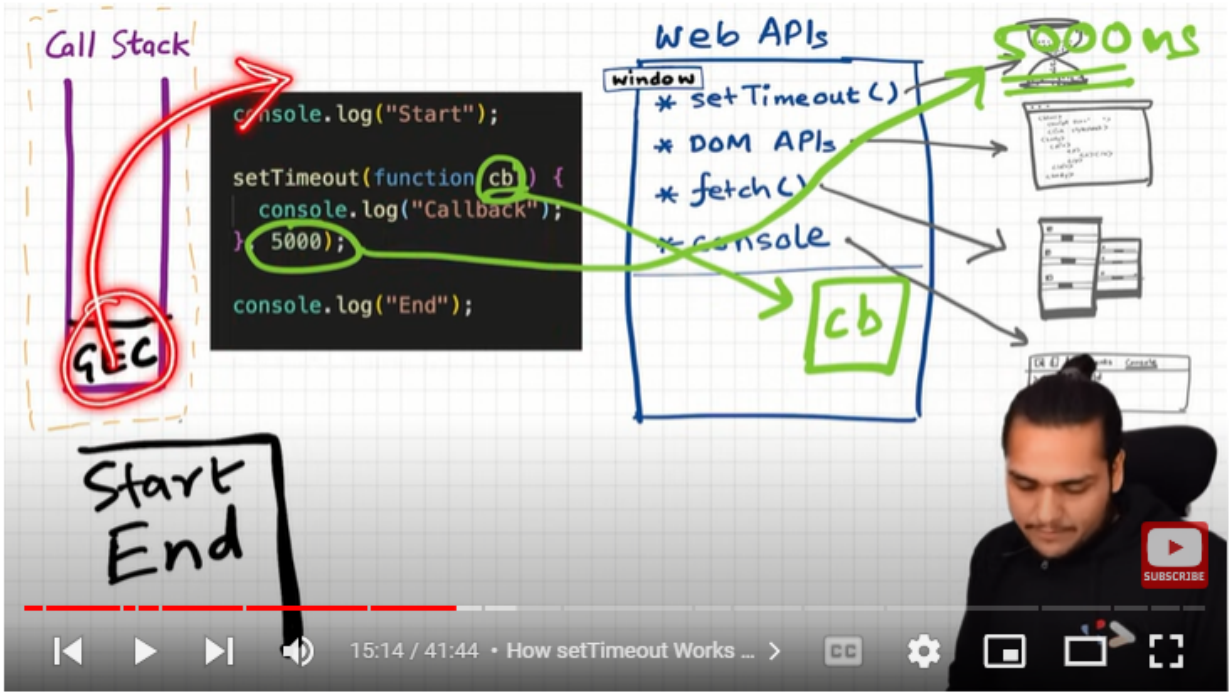
- **setTimeout takes a callback function and some delay of milliseconds. When we pass a callback function to the setTimeout, it will register the callback at some place inside the web API and as we have passed some delay so it will also set the timer accordingly.**



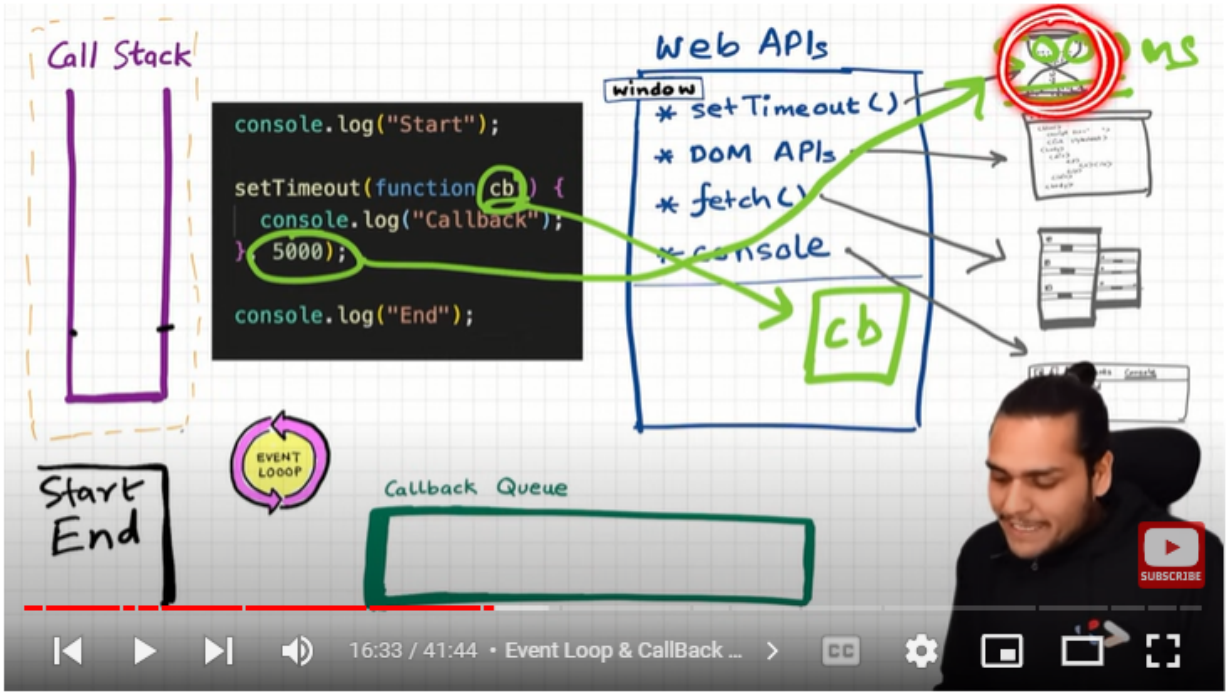
- **Now, the javascript code moves to the next line and again it will call the console web API and log to the console “end”.**



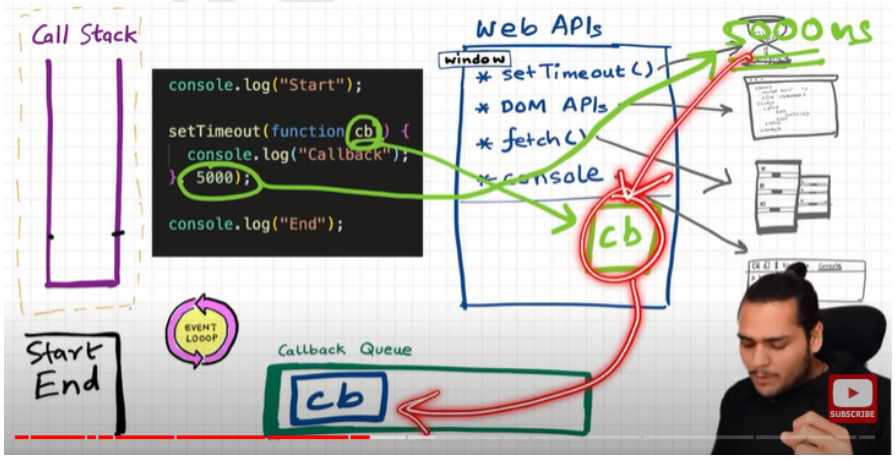
- **Now, the timer is counting the 5000ms and our code is done executing. Hence, the global execution context will be popped out of the call stack.**



- **Now, as soon as the timer expires we have to execute the callback function and in order to execute the callback function we have to push it to the call stack. Therefore, we need to somehow put the registersd callback function to the call stack.**
- **Now, comes into the picture the event loop and the call back queue**

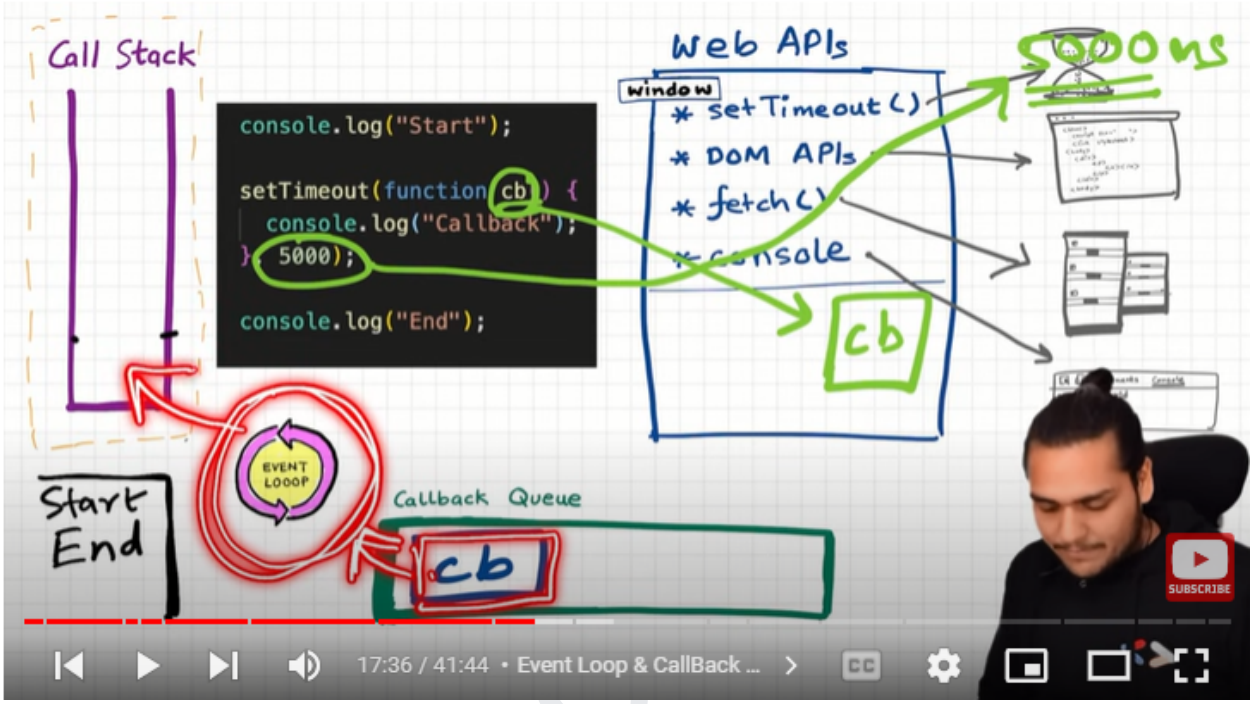


-
- **As soon as the timer expires, the callback function needs to go to the call stack for execution but it cannot go directly into the call stack. Here, comes the role of the callback queue. The callback function goes to the call stack through the callback queue**
- **When the timer expires the callback function is first put to the callback queue**

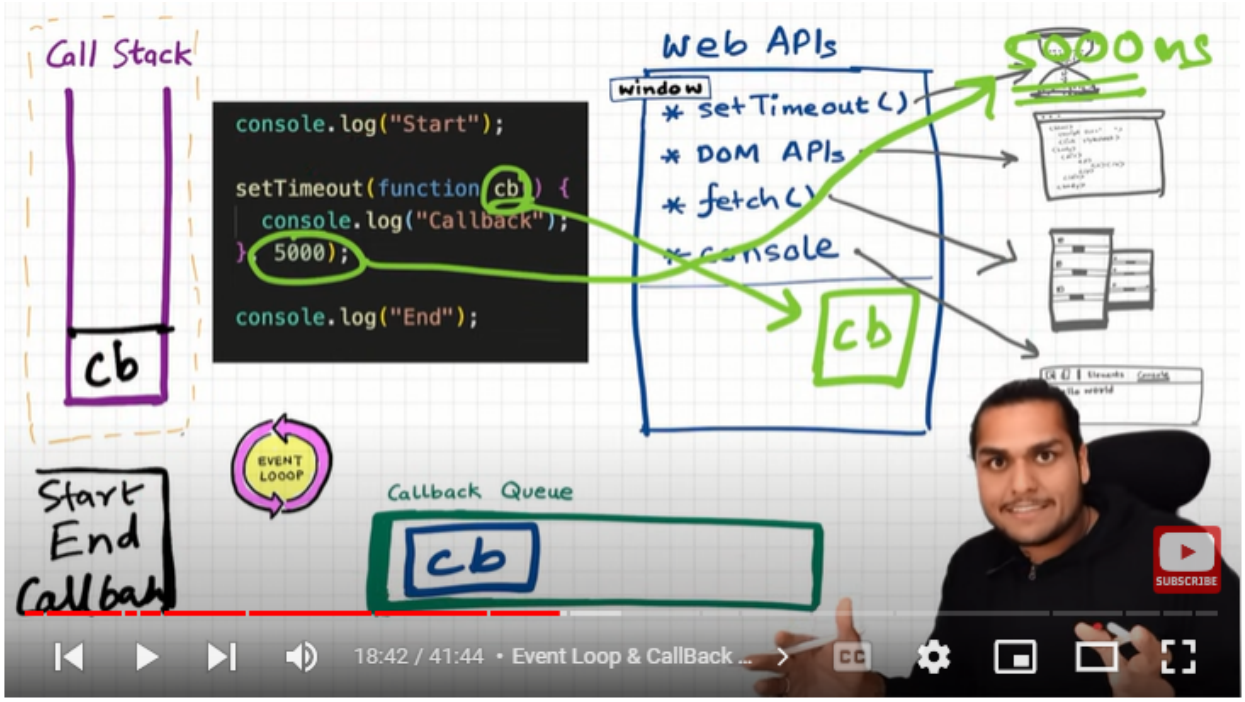


-

- **The job of the event loop is to monitor the callback queue. It checks whether there is something inside the callback queue, if there is something inside the callback queue then the event loop will put that call back function to the call stack for execution.**



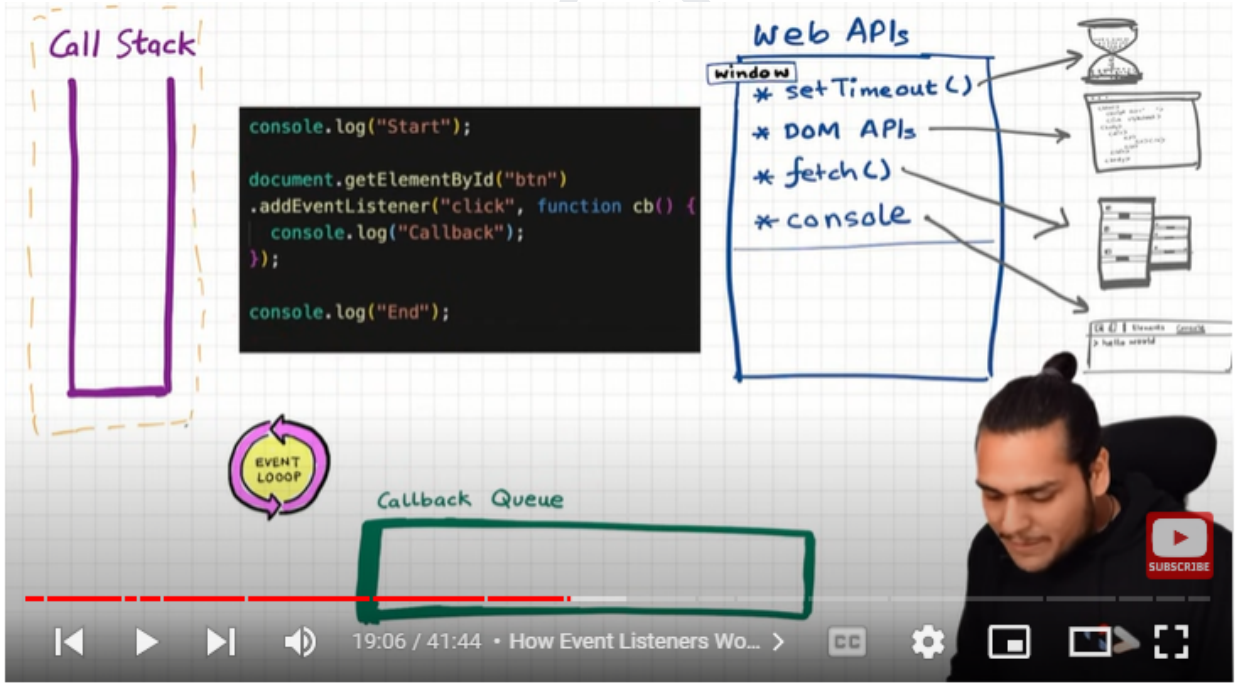
- **Now, as the callback function is inside the call stack. Hence, it will again create an execution context and then start executing the code inside the callback function line by line. After it completes the execution, the execution context will be popped out of the stack.**



-

- **This is how the whole javascript program executes behind the scenes.**

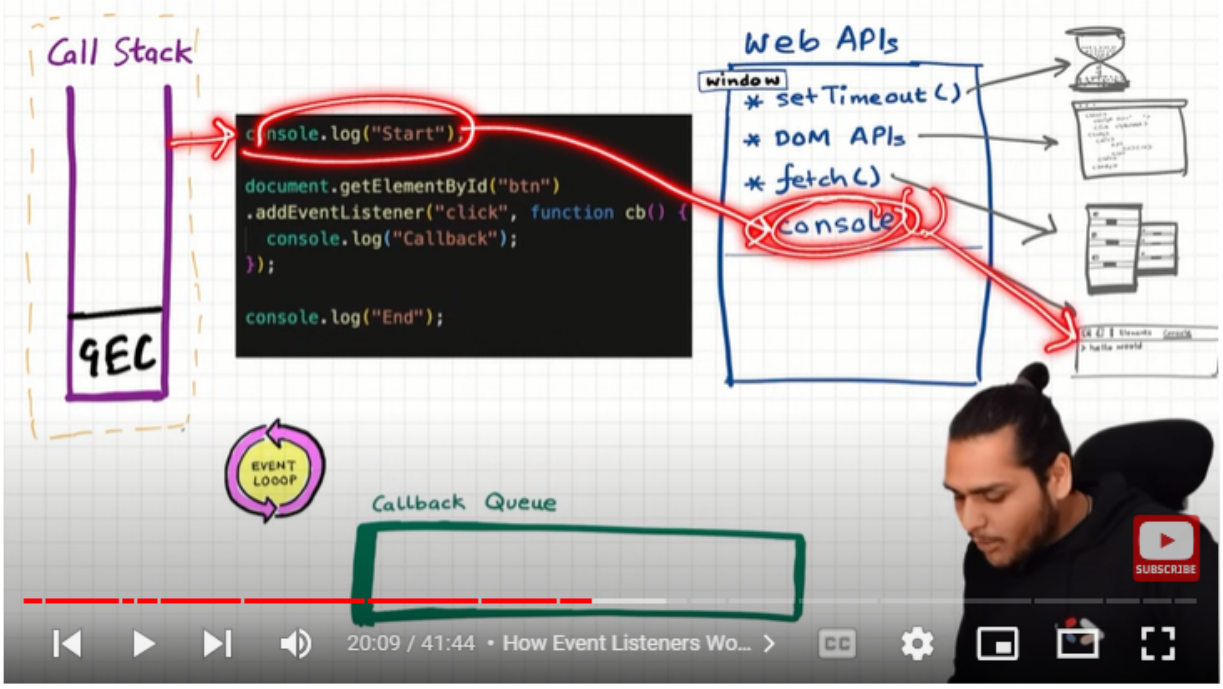
4. **How is the below code executed behind the scenes ?**

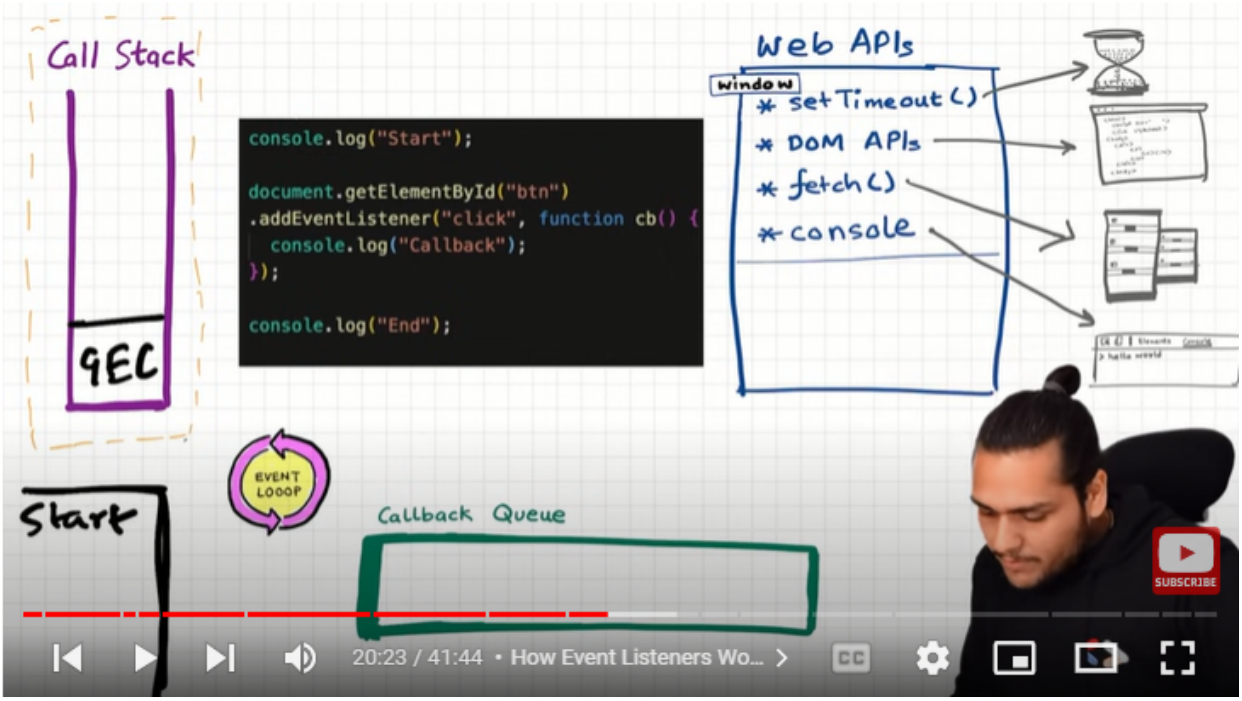


-

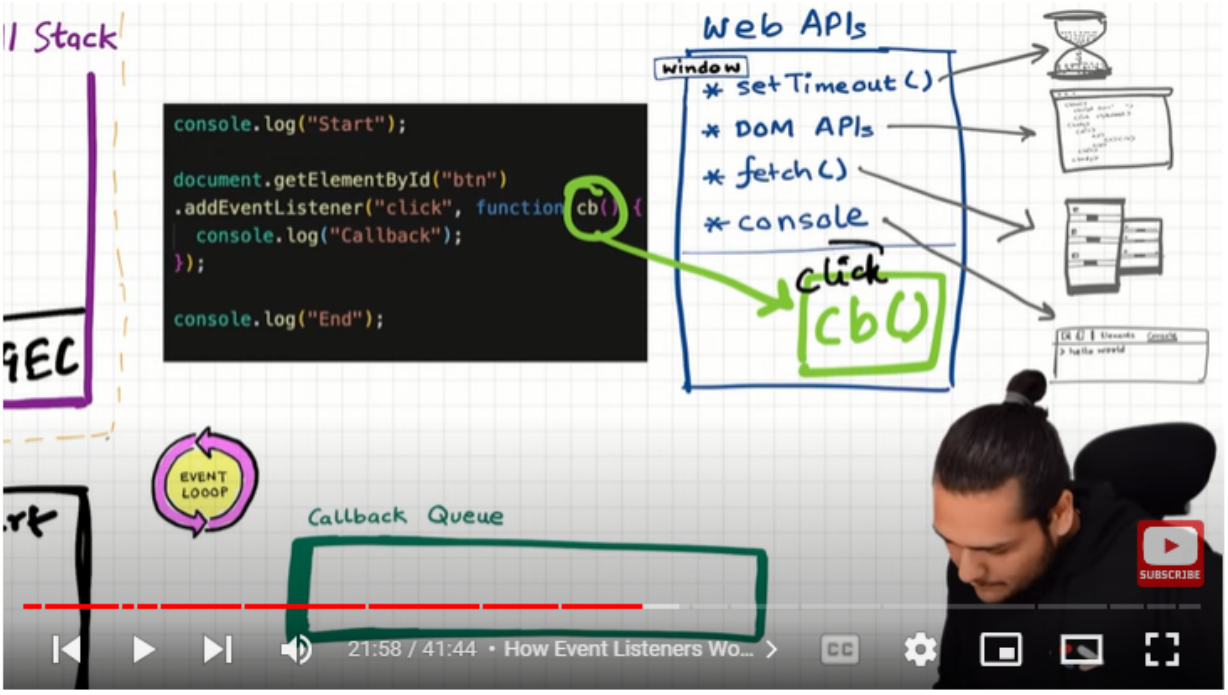
- **Again, when the javascript code is executed, the global execution context is created and pushed to the call stack and javascript code starts executing line**

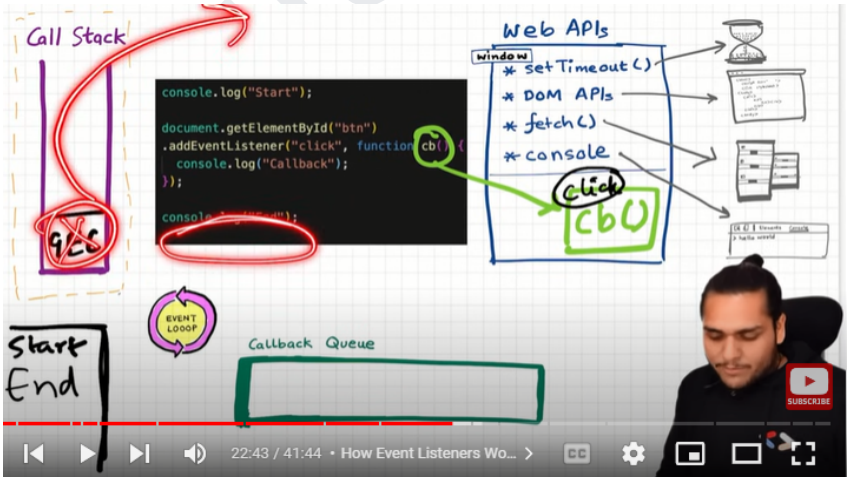
by line. At line 1 it encounters the console.log(“start”), so it will call the console web API and then it will log “start” to the console.

- 

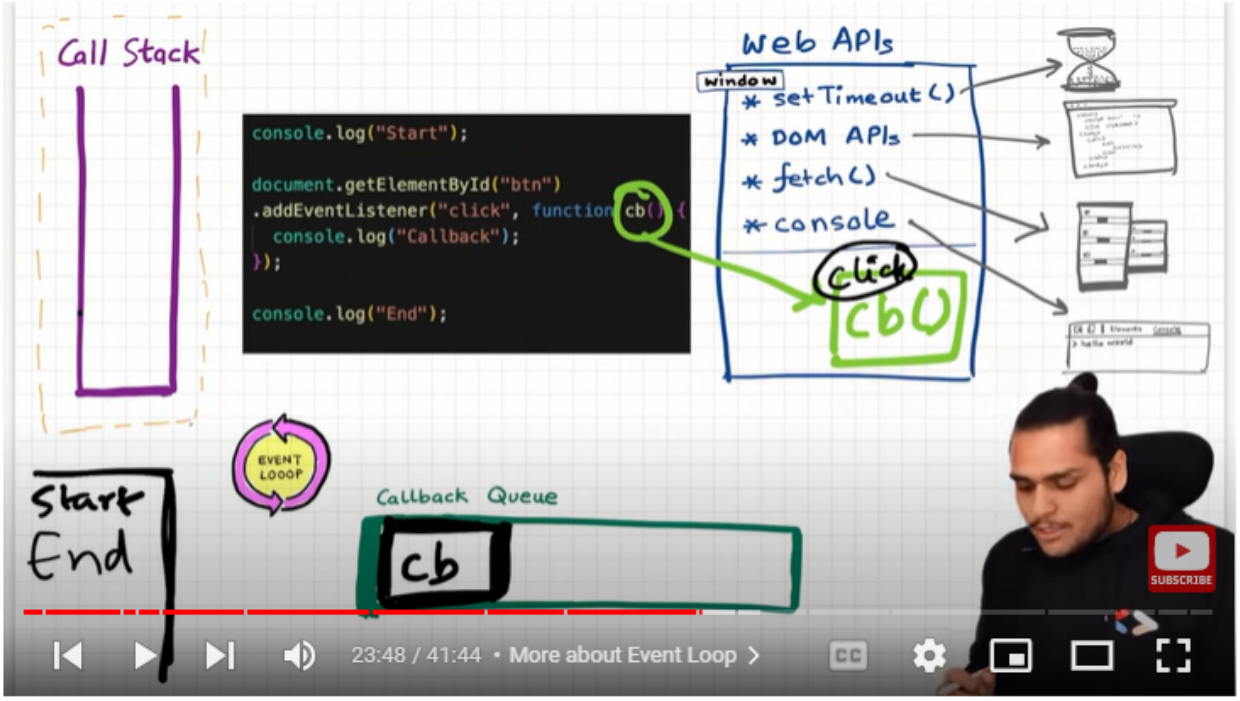
- 

- Now, the execution will go to the next line and it will encounter addEventlistener on this line.

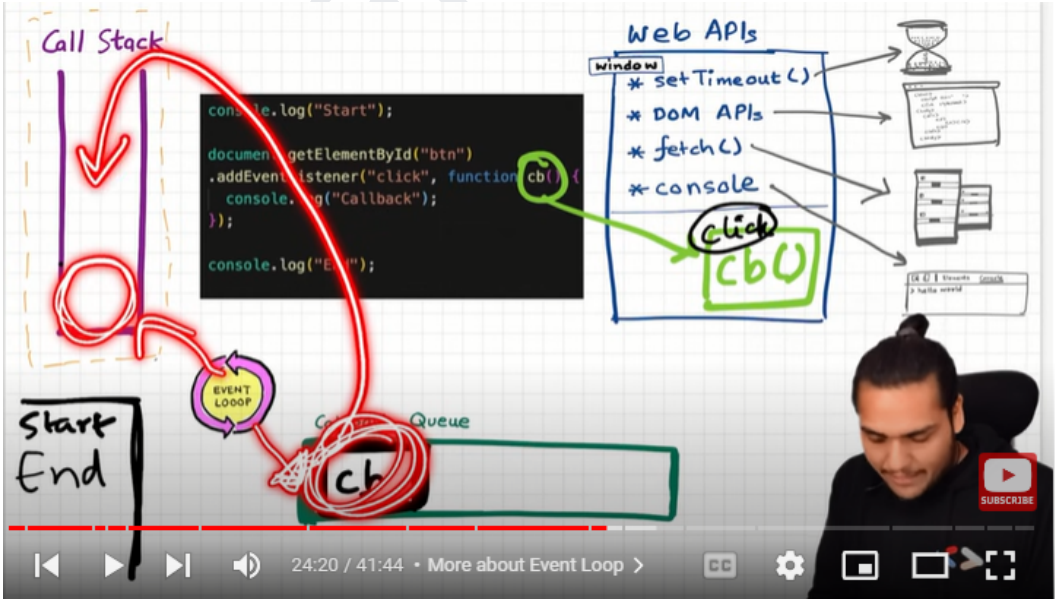
- **addEventListener is another super power given by the browser to the javascript engine through the global window object in form of web API which is the DOM APIs**
- **This addEventListener registers a callback function to the web APIs environment and attaches a click event to it. This is known as registering the event.**
- 

- **After registering the call back function to the web APIs environment, javascript will move to the next line and execute the next line of code and it will log “end” to the console. After that we have nothing much to execute so the global execution context will be popped out of the stack**
- 

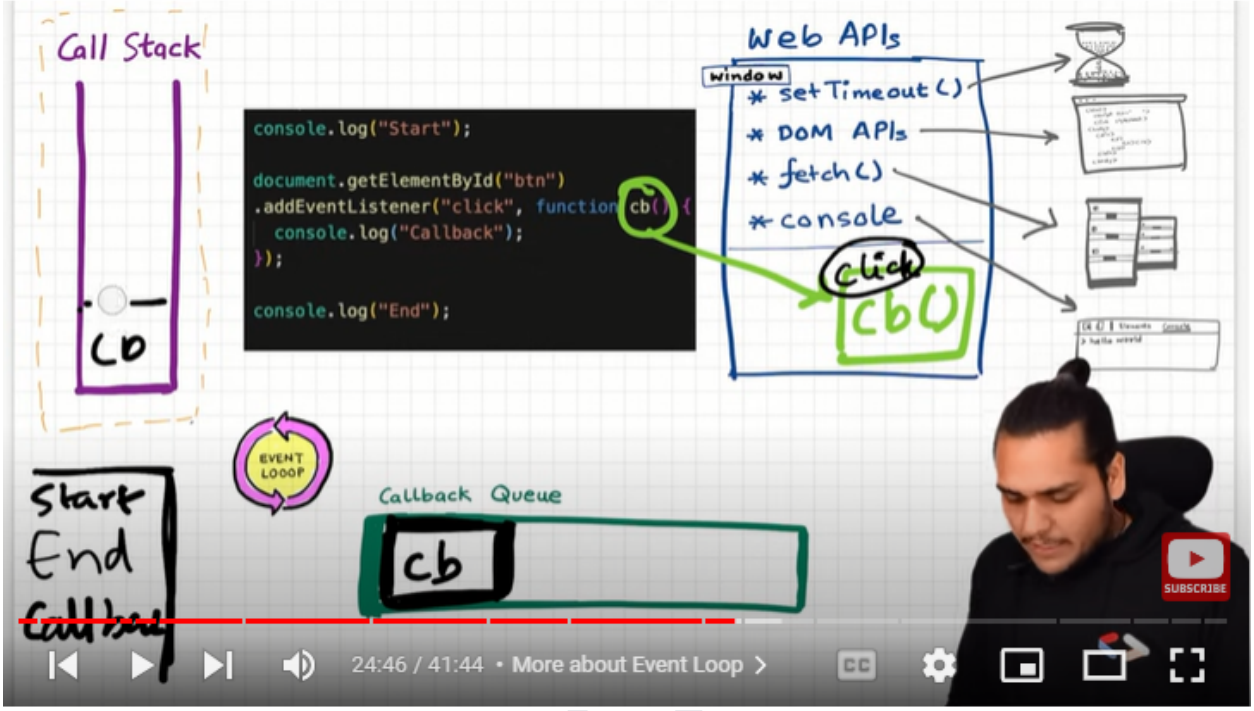
- **The eventhandler will stay inside the web APIs environment until and unless we explicitly remove the event listener or close the browser.**
- **When the user click on the button, the call back function is then pushed inside the call back queue and waits for its turn to be executed.**
-



- **Event loop has just one job of continuously monitoring the call stack and the call back queue and if the event loop sees that the call stack is empty and there is also a call back function inside the call back queue then it will quickly push that callback function to the call stack.**
-



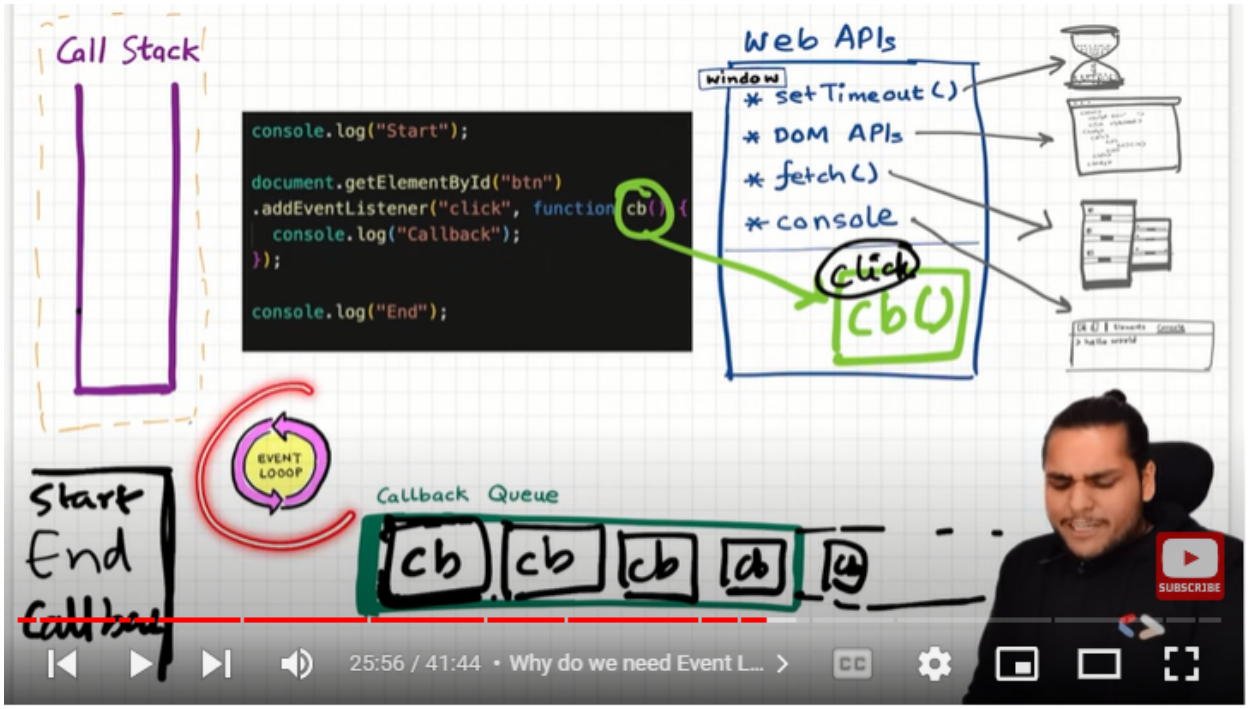
- **The call back function will be quickly executed and “callback” will be printed to the console. After this the execution context of the callback function is pushed out of the stack.**



-
- **This is how the javascript program is executed behind the scenes.**

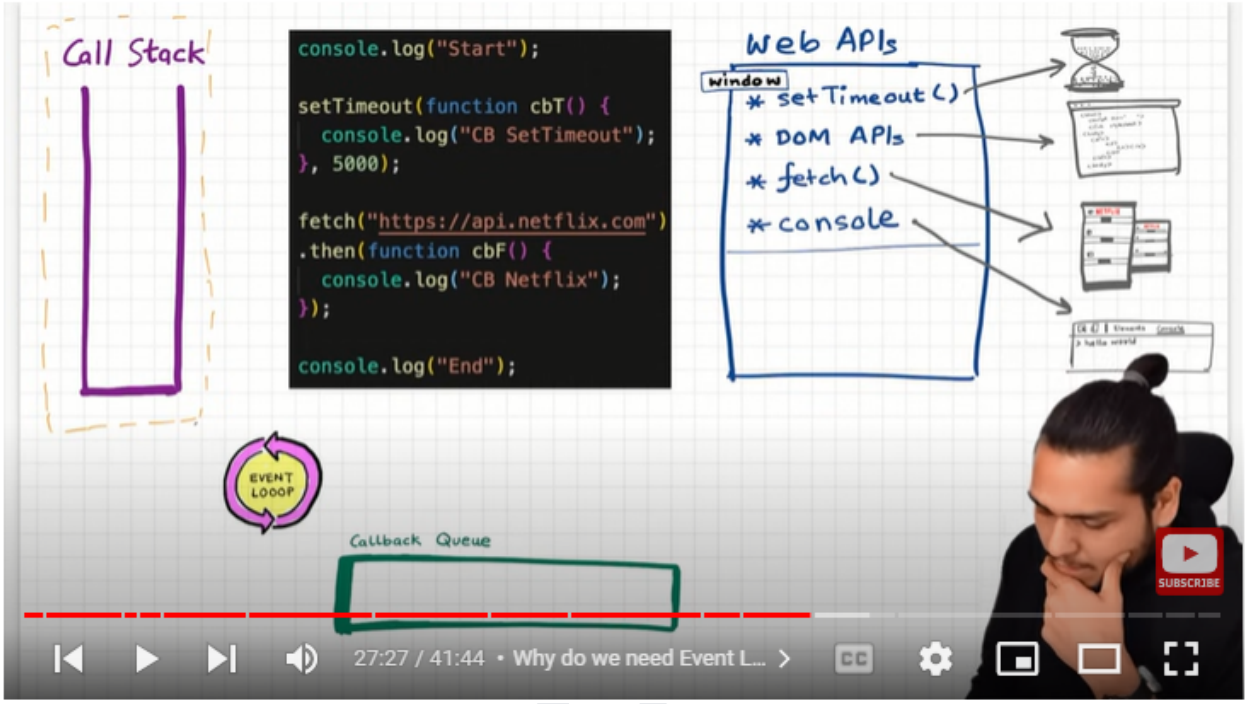
5. **Why do we need the callback queue ? we could have picked up the registered callback function from the web APIs environment and pushed it directly to the call stack whenever the call stack is empty.**

   - **We need a call back queue because lets say users click on the button several times in that case there will be more than one callback function pushed inside the call back queue.**

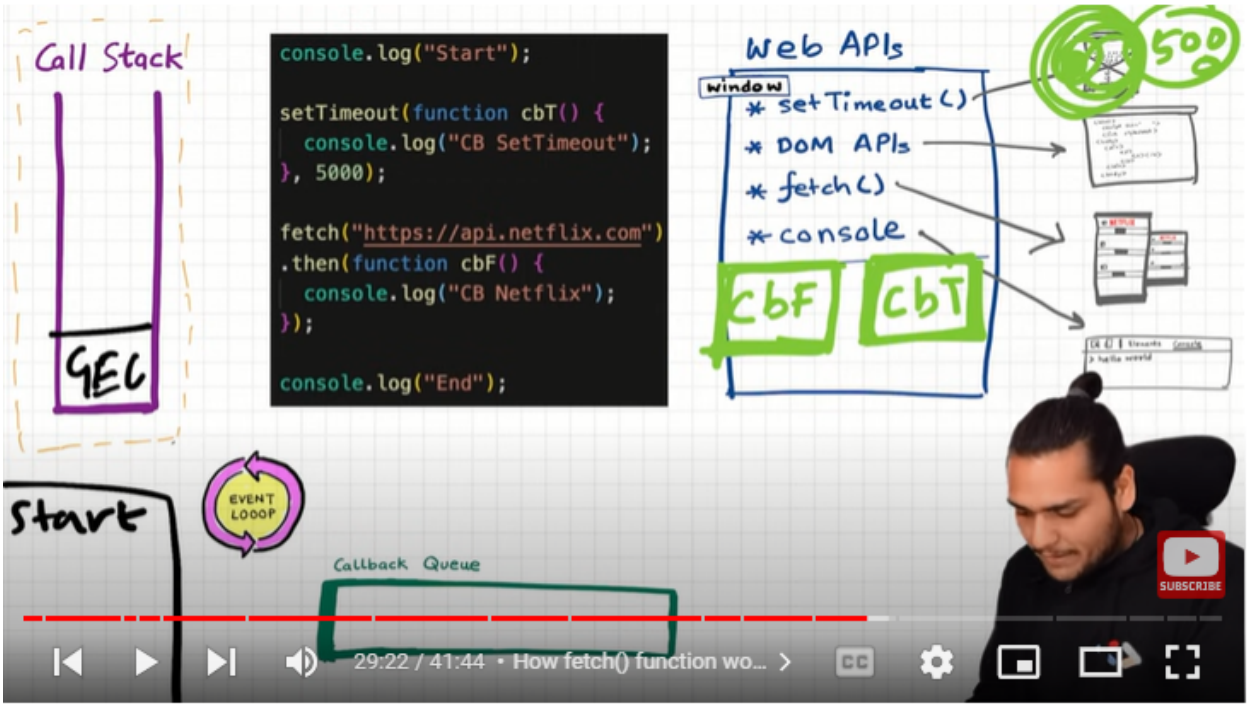
- 

- **Now, the event loop will check whether the call stack is empty and there is some callback function inside the callback queue to be executed, if there are some call back function inside the callback queue, it will then put every call back function one after the another inside the call stack and this is how each and every callback function is executed one by one.**
- **In real life applications, there are a lot of callback functions registered inside the webAPI environment and we need to execute them one by one whenever it is needed, so we need to put them in the callback queue, so that they will be pushed inside the call stack and can be executed one after the other.**

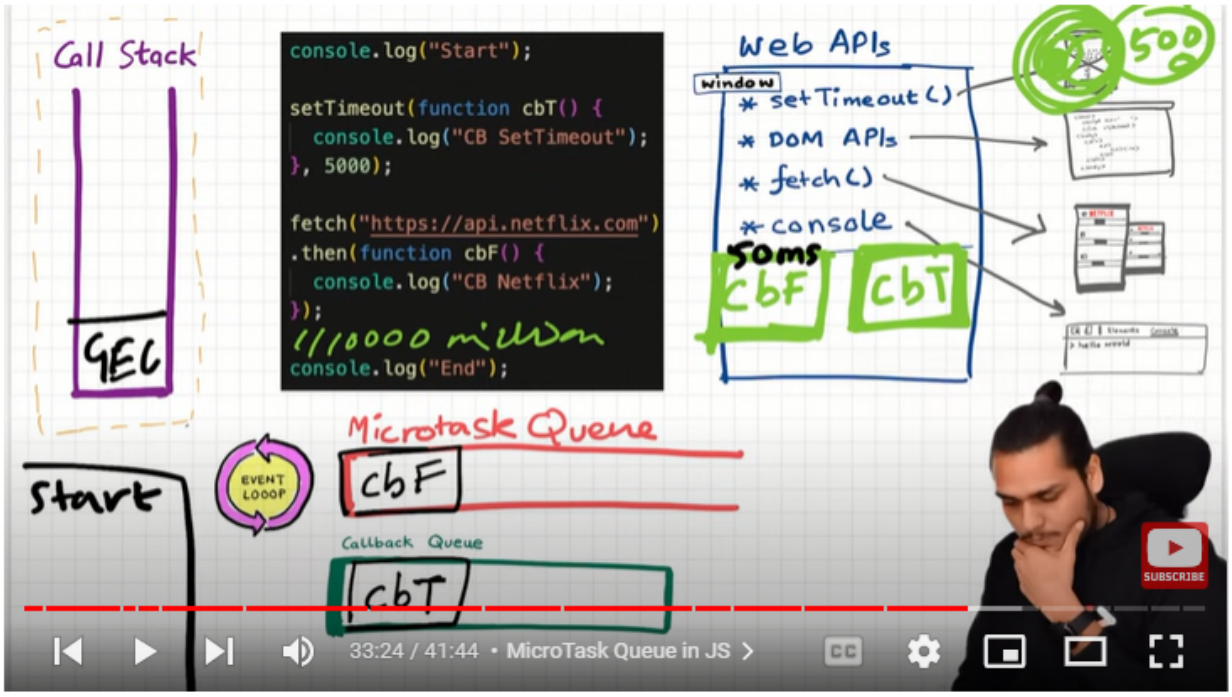
6. **How does the fetch web API work and how does it works different from setTimeout, console or DOM APIs ?**



-
- **In this example we have setTimeout as well as fetch function**
- **When the program starts executing a global execution context is created and it is pushed to the call stack and javascript starts executing the code line by line.**
- **After executing the line 1 it will print “start” to the console**
- **Now, javascript moves to the next line and now setTimeout will register a callback function inside the web APIs environment and also 5000ms timer starts**
- **Now, the Javascript engine moves to the next line. On this line there is a fetch() which is a web API that is used to make network calls. fetch () also register a callback function to the web API environment**

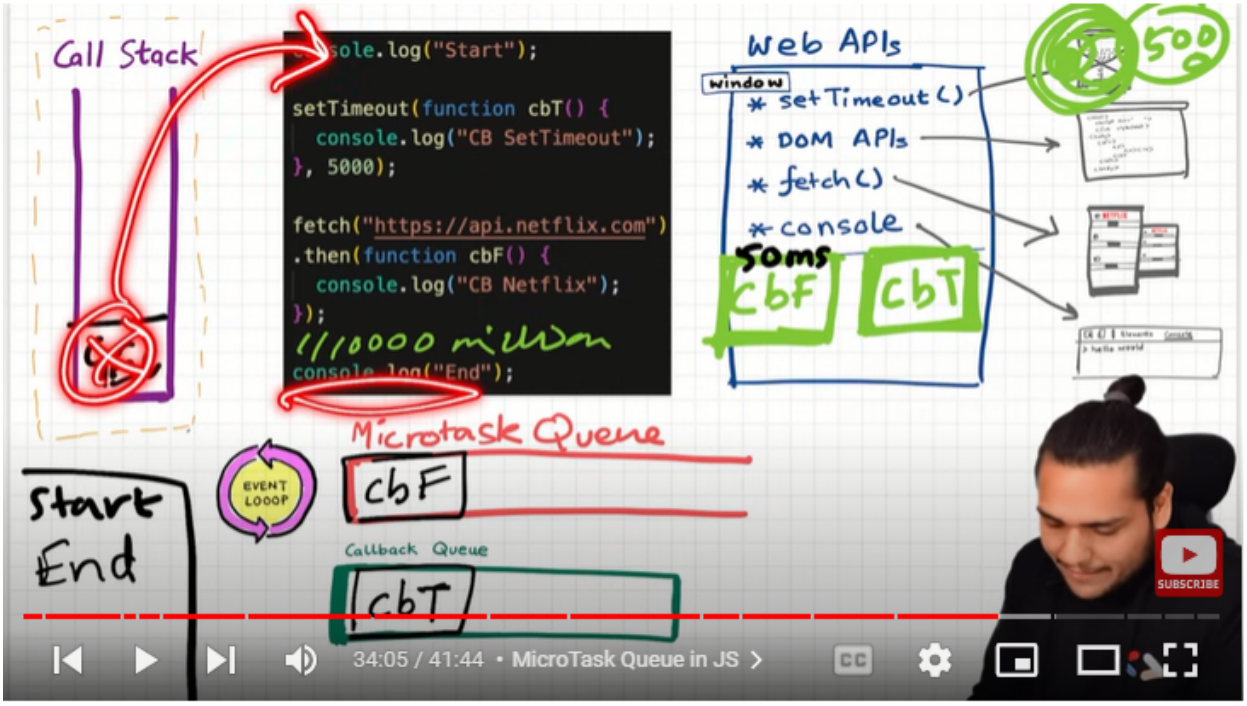


- The CBT call back function is waiting for the timer to expire and the CBF call back function is waiting for the data to be returned from the netflix server. fetch() makes a network call to the netflix server and this netflix server will return back the data to the fetch and once the data is received, now the callback function which is registered by the fetch inside the web API environment is ready to be executed. Now, you might think that the call back function will move to the callback queue, right ? But it is not the case. Instead it will move to something called Microtask queue which is similar to the callback queue but it is having higher priority. So whatever functions will move inside the microtask queue are executed first and the call back function which will move to the call back queue will be executed later.
- Which callback functions will move to the Microtask queue ?
- The callback functions in case of the promises or the network call will move to the microtask queue.
- Let say there are millions of lines of code after the fetch API call, so javascript is busy in executing these lines of code and meanwhile the timer expires, as soon as the timer expires the call back function registered by the setTimeout will move to the call back queue.



-

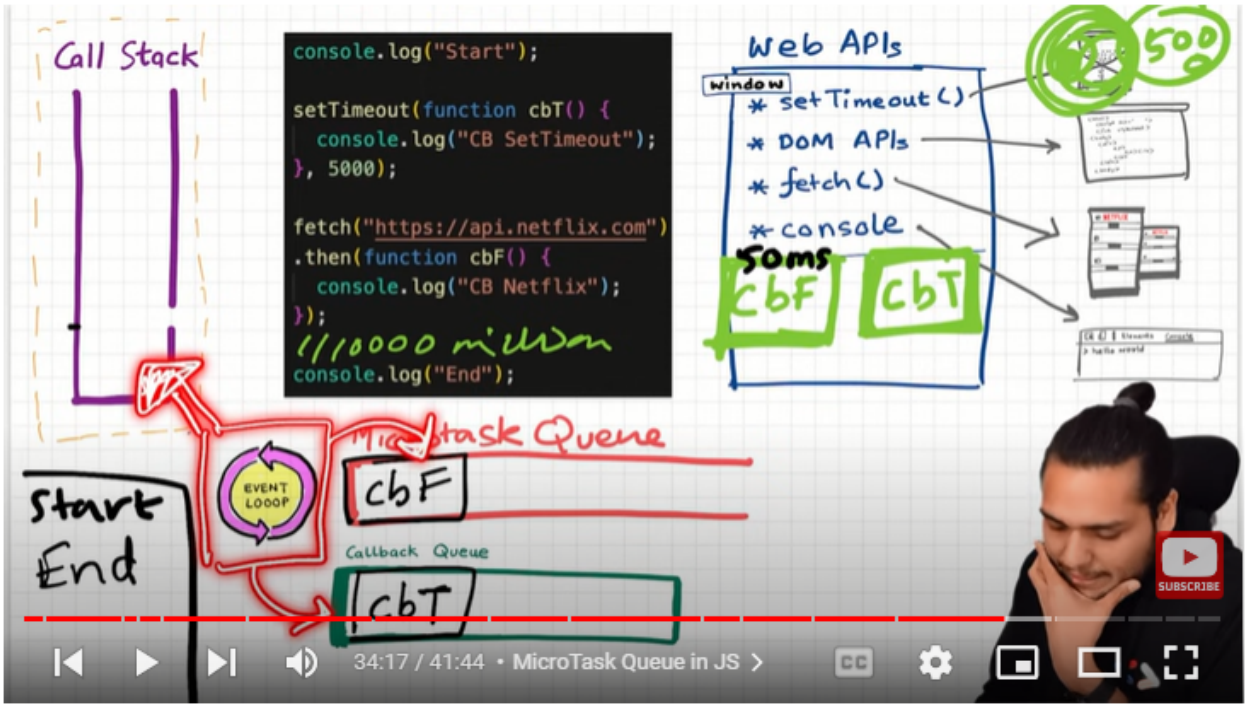
- **Let's say millions of lines of code are executed and the javascript engine reaches to the last line and prints “end” to the console. Now, there is nothing much to execute. Hence, the global execution context will be popped out of the stack.**



-

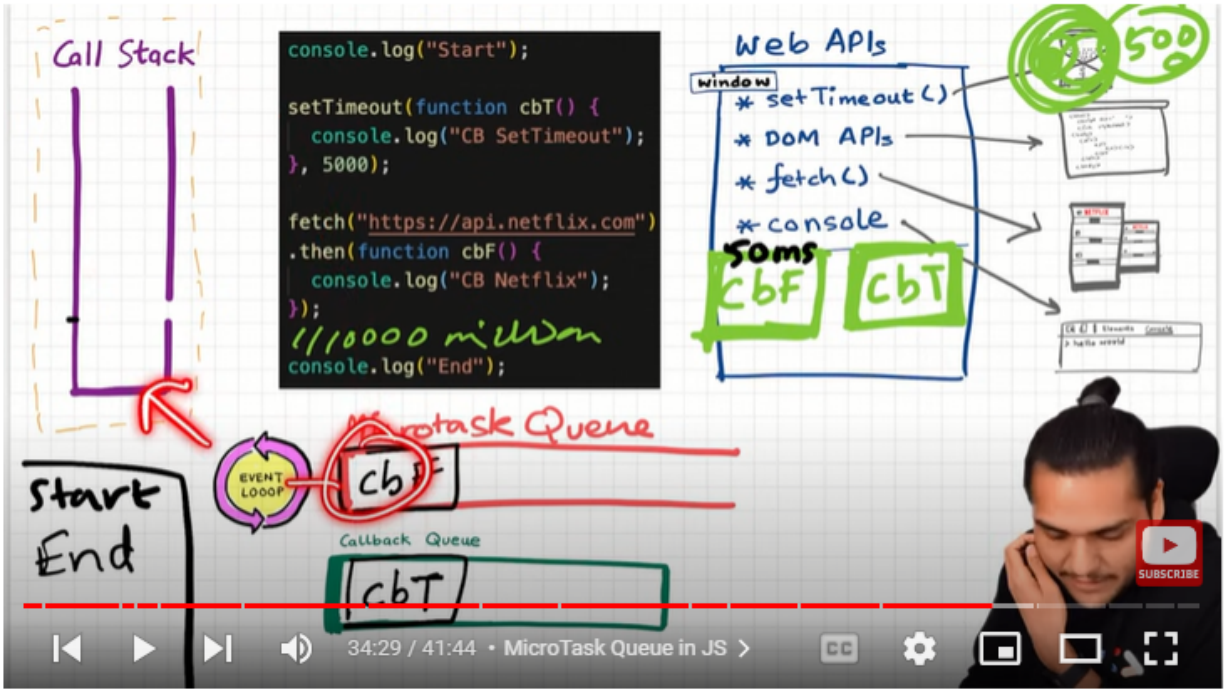
- **The event loop is continuously monitoring the call stack, the microtask queue and the call back queue. If the event loop see that the call stack is empty and**

there are call back functions inside the Microtask queue and the callback queue then it will push those call back functions to the call stack for execution.



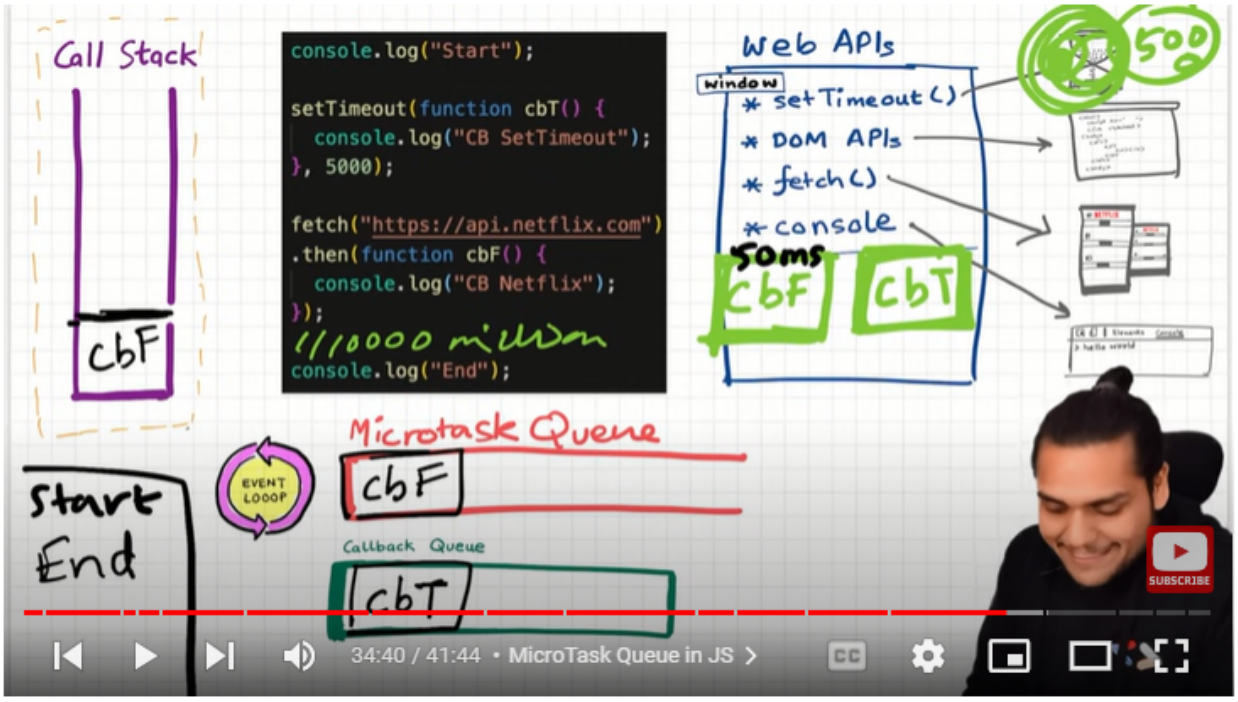
-

- **The event loop continuously monitors the call stack, the Microtask queue and the callback queue and it finds that the call stack is empty and there are callback functions inside the Microtask queue and the callback queue.**
- **We know that the priority of the Microtask queue is high, therefore the event loop first moves the CBF call back function to the call stack for the execution.**



-

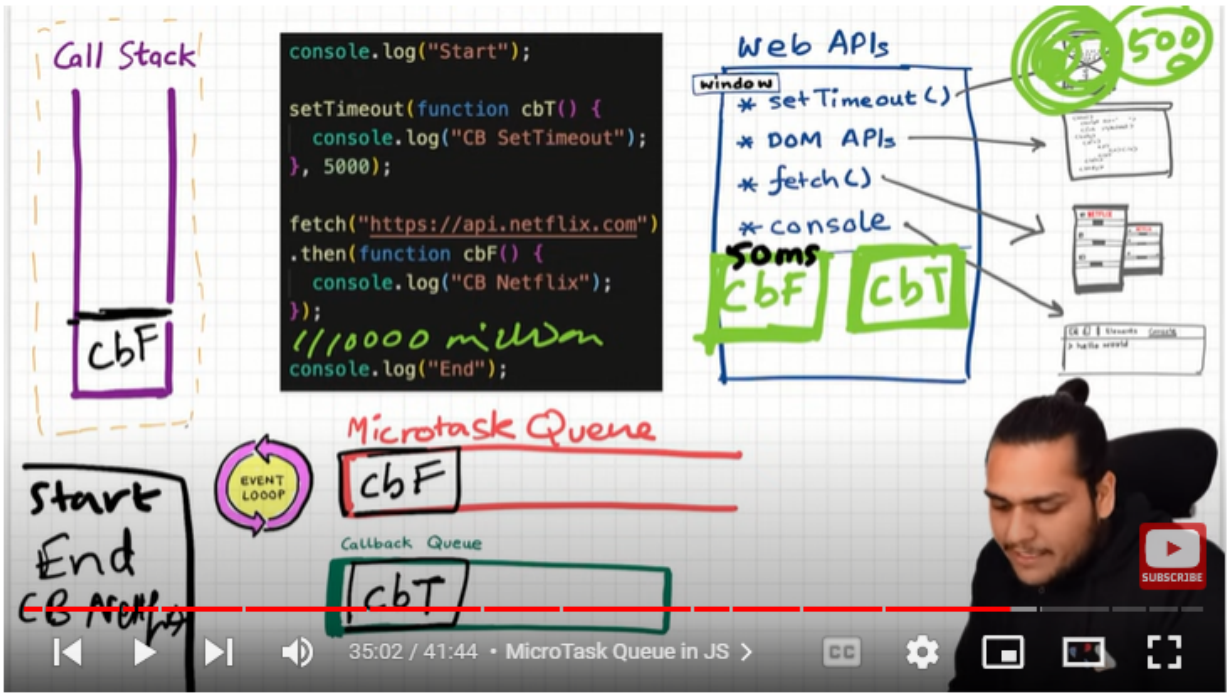
- **Now, again the CBF function moves to the call stack and then the code inside this function will start executing line by line.**



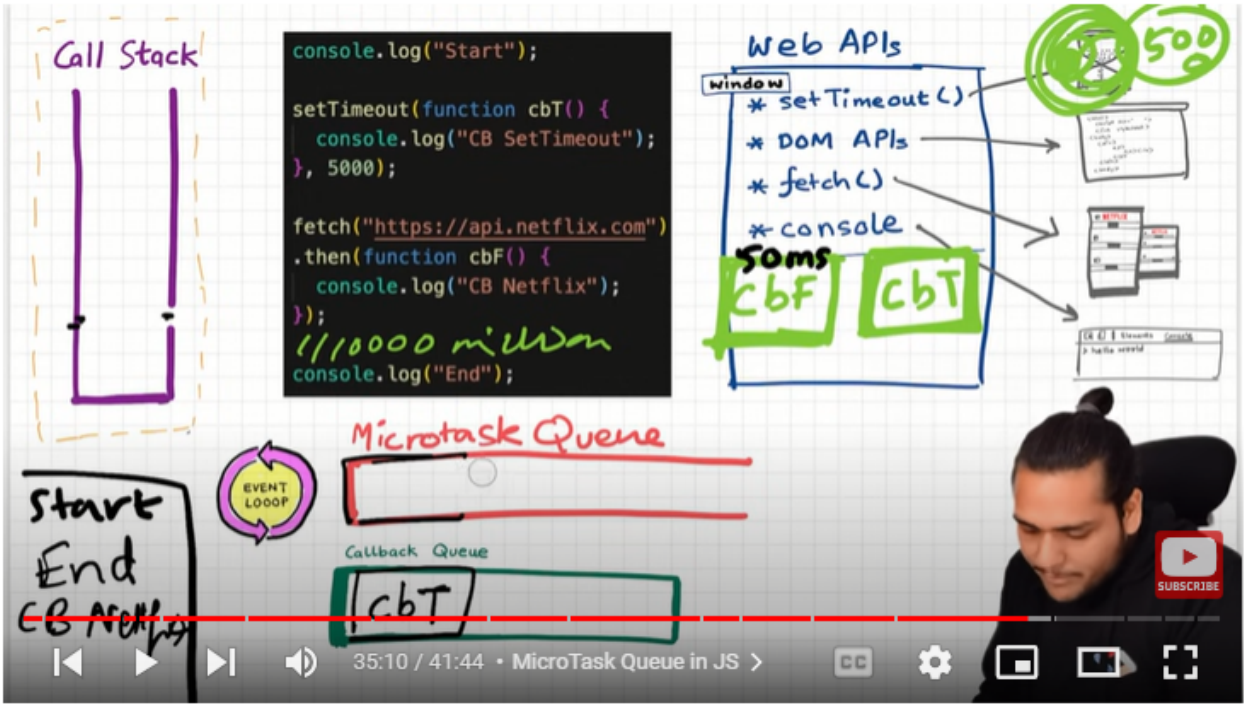
-

- **Finally after executing the code inside the CBF call back function It will be popped out of the call stack and CB Netflix will be printed to the console.**

-



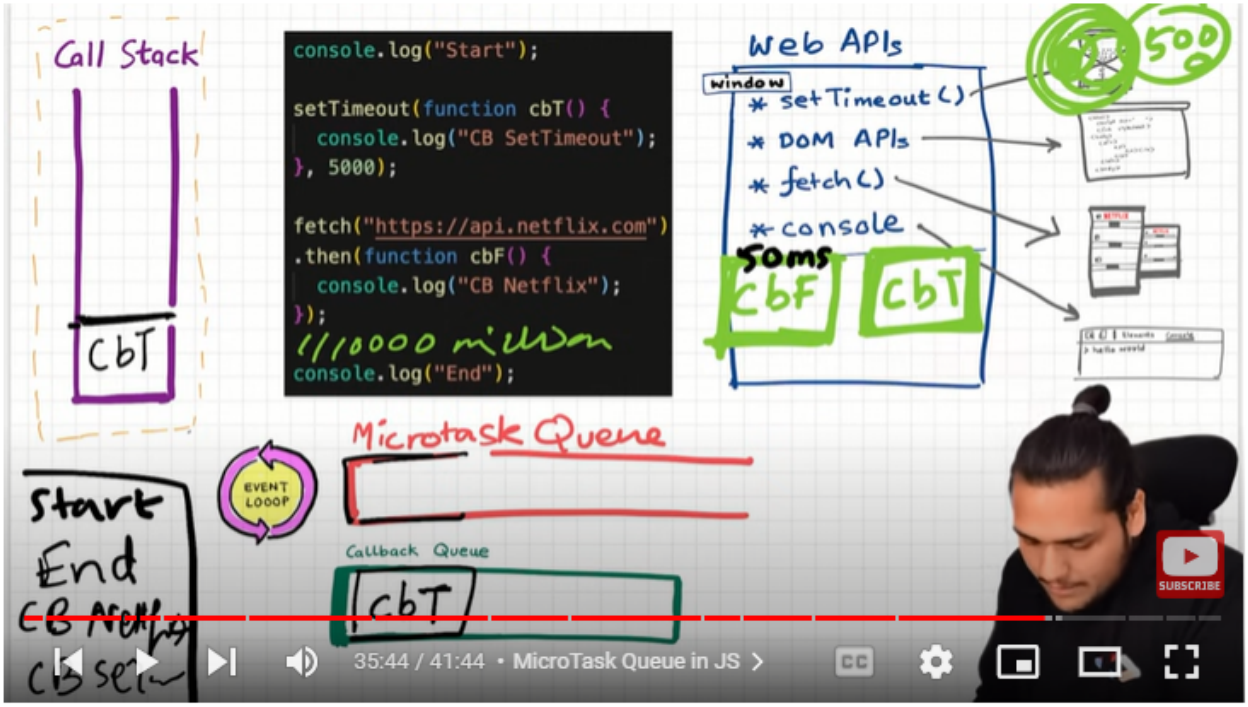
-



-

- **Here, we can see that the execution context of the CBF function will be popped out of the stack once it is executed completely and also the Microtask queue will be empty**
- **Now event loop is continuously monitoring, so it will see that there is CBT call back function inside the callback queue and also the call stack is empty. So it**

will push the execution context of the CBT call back function to the call stack and it will execute it and print “CB setTimeout” to the console



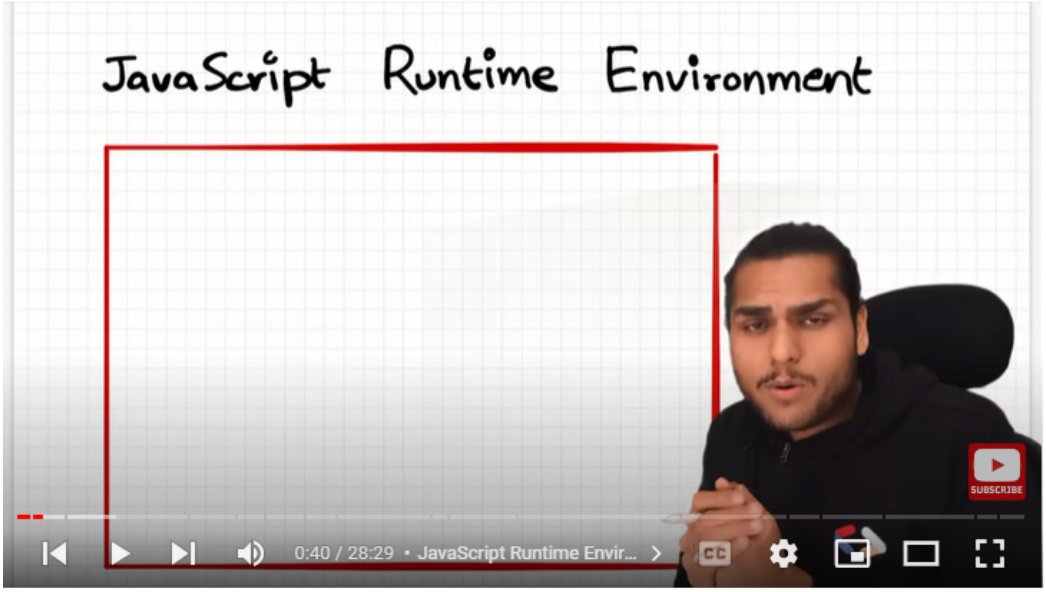
-
- After executing CBT. It will be popped out of the call stack and the entire program is executed
- This is how the program is executed behind the scenes.

7. Microtask queue is given priority so what can come inside the microtask queue ?
   - All the callback function which comes through promises will go to the microtask queue and the callback function which comes through the mutation observer will go to the microtask queue.
   - The rest call back functions which come from the setTimeout, DOM API etc will go to the callback queue.
   - callback queue is also known as the Task queue.

8. What is Starvation of Functions in Callback Queue ?

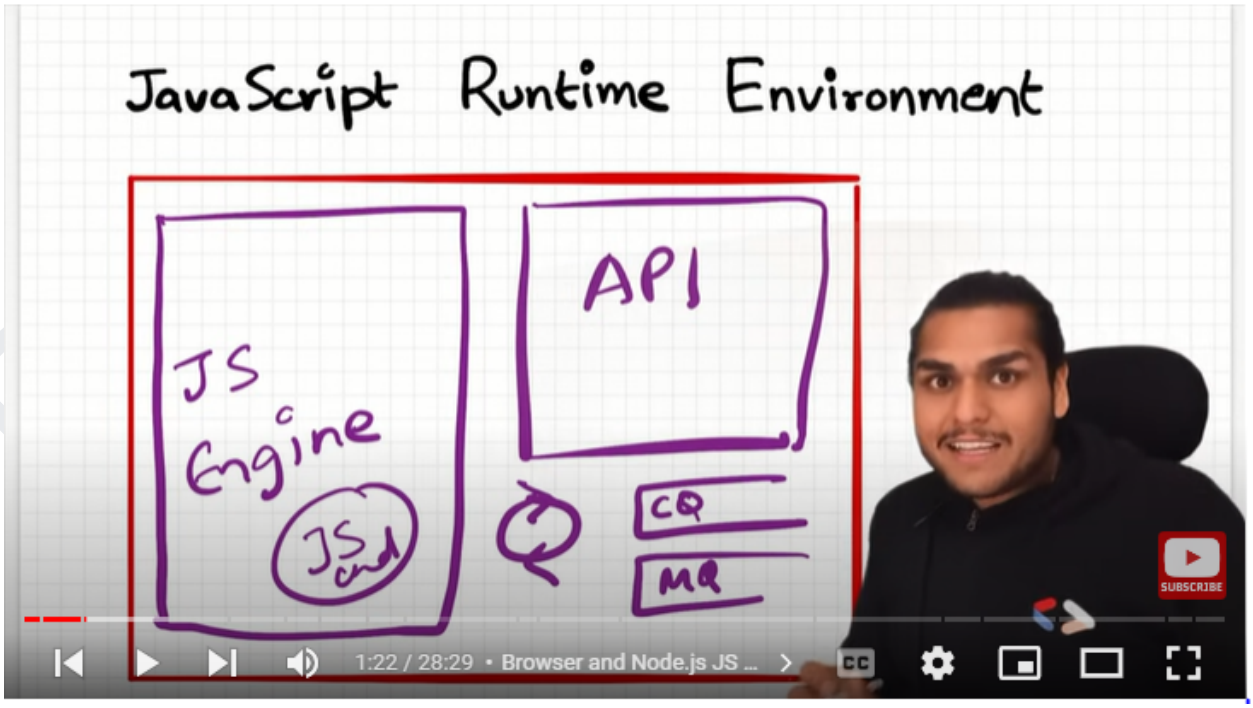
Episode 16 : - JS Engine EXPOSED 🔥 Google's V8 Architecture

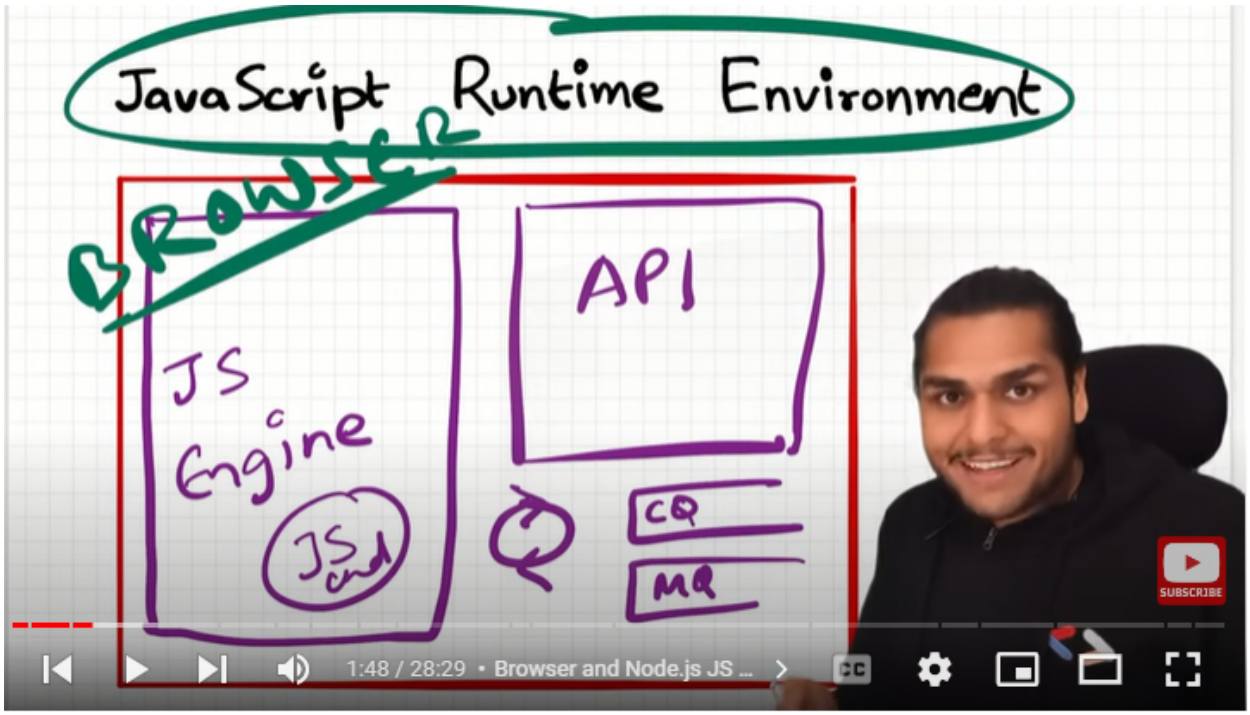
1. **Javascript can run inside robots, Browsers, servers, smartwatch and many other things. This is all possible because of the javascript runtime environment. We can imagine a javascript runtime environment as the big container which has all the things required to run javascript.**



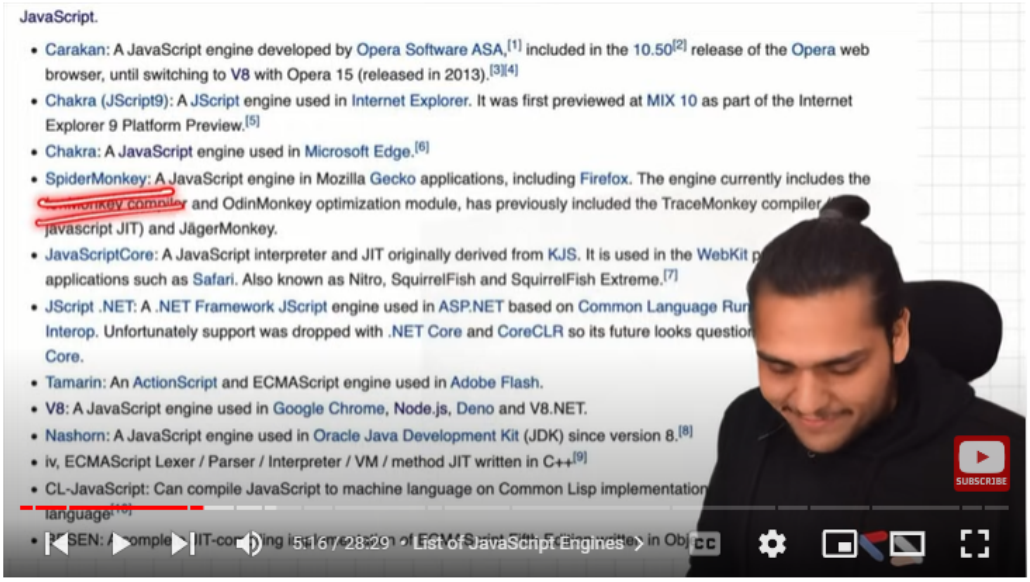
- **Javascript Runtime environment contains javascript engine, event loop, WEB APIs, callback queue, Microtask queue**

-

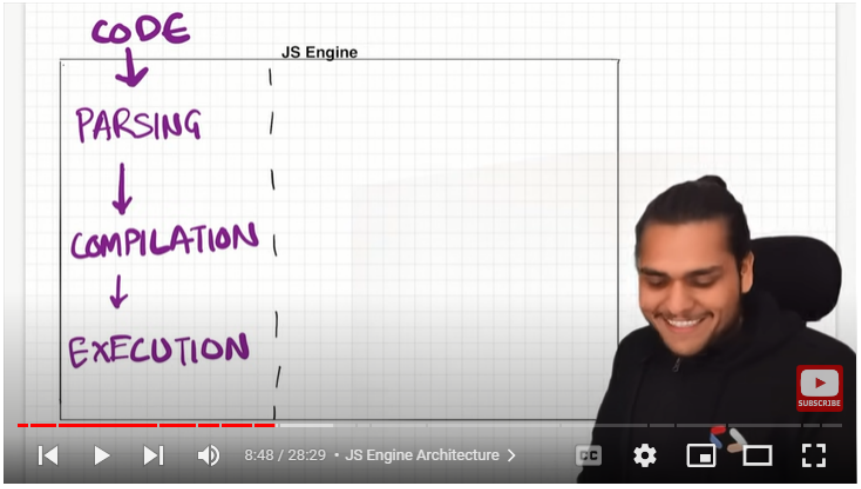


- **Javascript runtime environment is not possible without the javascript engine, so basically javascript engine is the heart of javascript runtime environment.**
- **Browsers can execute the javascript code because it has the javascript runtime environment. Every browsers has javascript runtime environment.**
- 

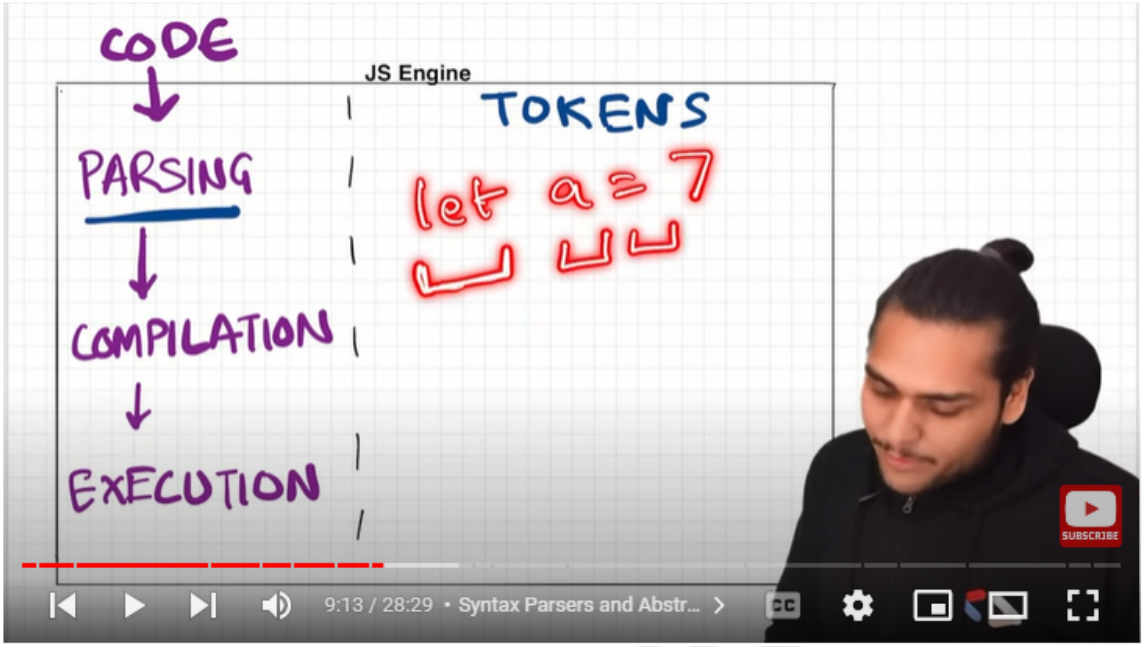
- **NODE js also contains a javascript runtime environment. Therefore, it can run javascript code outside the browser**
- **Let's say we want to run the javascript code inside the water cooler. All we need is the javascript runtime environment inside the water cooler to run the javascript program inside the water cooler.**
- **Therefore, because of the javascript runtime environment, javascript is able to run inside a lot of devices.**
- **The APIs inside the javascript runtime environment gives us access to the superpowers which we can use inside our javascript code.**
- **These APIs can be different or same in NODE js and browsers. For example, inside the javascript runtime environment of the browser we have localStorage API but incase of the NODE js runtime environment it can be different.**
- **The setTimeout web API can be found in both NODE js as well as browser javascript runtime environments. But the implementation of the setTimeout inside them can be different.**

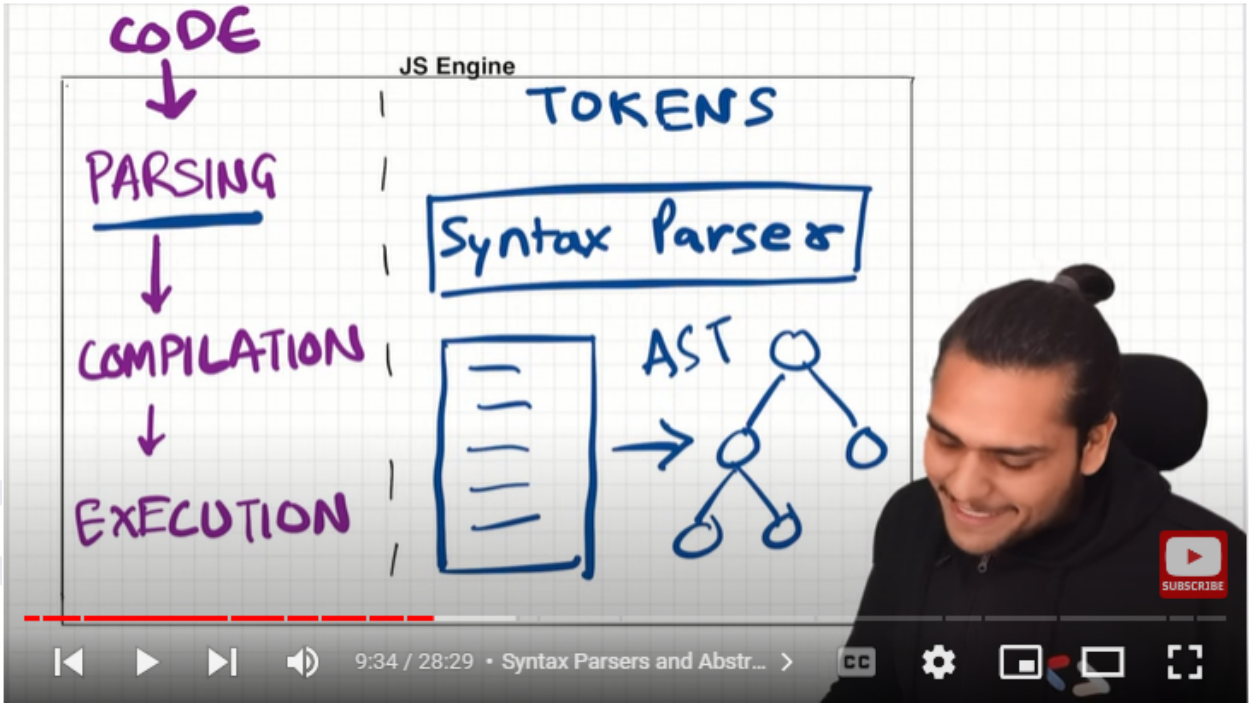
- **There are different javascript engines inside different browsers.**
- 

- **The first JavaScript engine was created by Brendan Eich in 1995 for the Netscape Navigator web browser**
- **Javascript engine is not a Machine. It is just like normal code which is written in low level languages. For example :- Google v8 engine is written inside c++.**

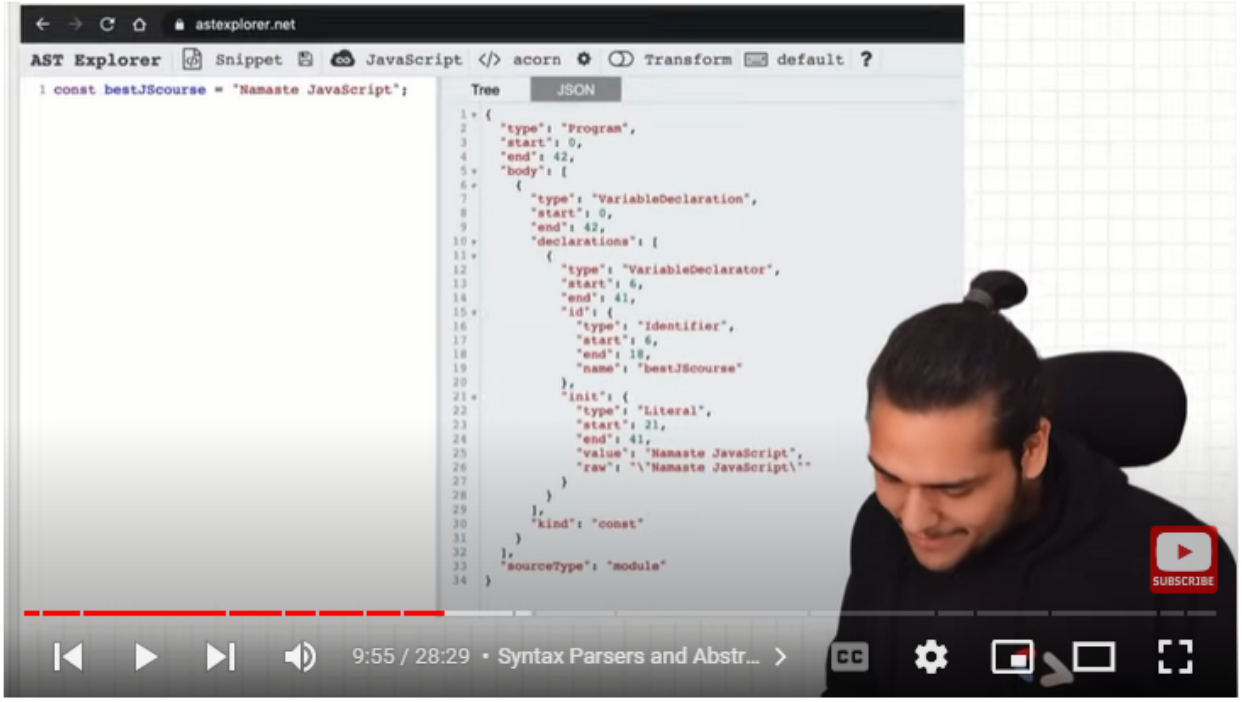
2. **Javascript engine architecture.**
- **Javascript engine takes javascript code as the input. This code goes through three major steps inside the javascript engine 1). Parsing 2). Compilation 3). Execution.**
- 

- **Let's see each step one by one and observe how it executes the code.**

- **During the 1st step, the code is broken down to different tokens**
- 

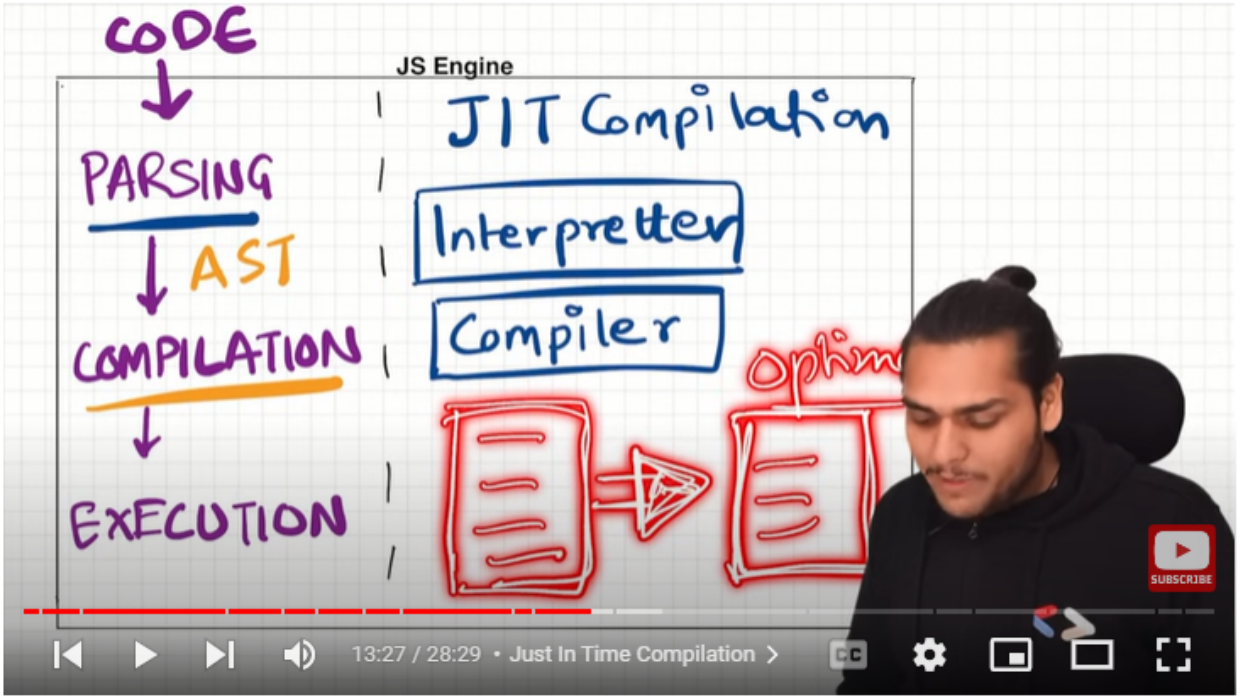
- **Now, there is something called syntax parser which converts code into AST (Abstract syntax tree).**
- 

- **This is how an Abstract syntax tree looks like for one line of code. Imagine the AST for hundreds of lines of code 😁.**



-
- **Now the AST generated is passed to the compilation phase.**

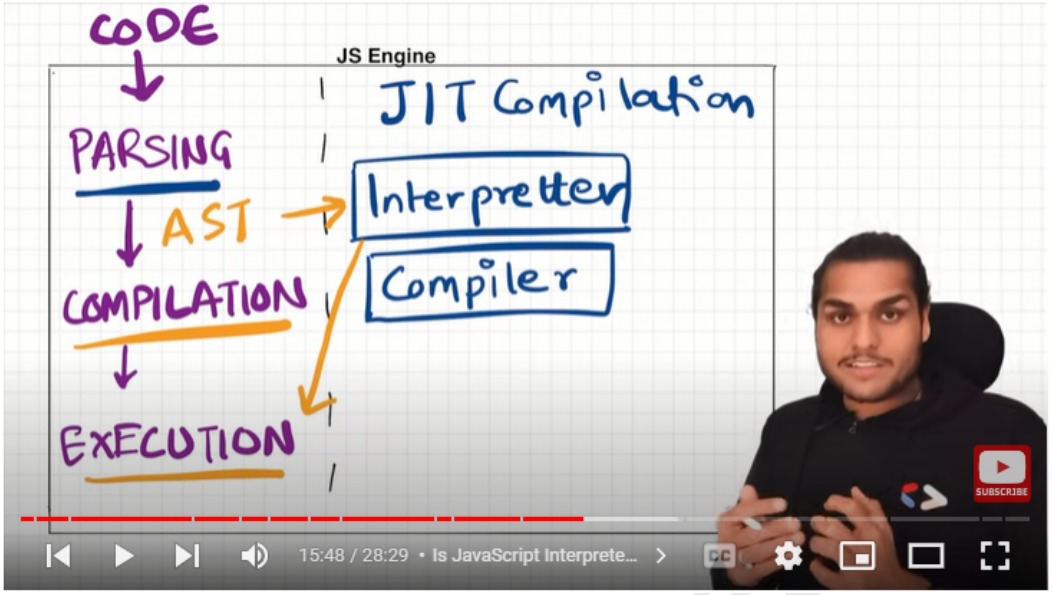
- **2nd step :- compilation stage**
- **Javascript has something known as JIT(just in time compilation)**
- **Before understanding JIT, let's understand interpreter and compiler**
- **Interpetter :- It takes the code and starts executing the code line by line. It is used by many languages.**
- **Compiler :- Many languages use a compiler to compile the code. Incase of the compiler the whole code is compiled before executing. So the code is compiled and a new code is generated which is the optimized version of the code and then it is executed. The optimized code runs faster and it has a lot of performance improvement.**



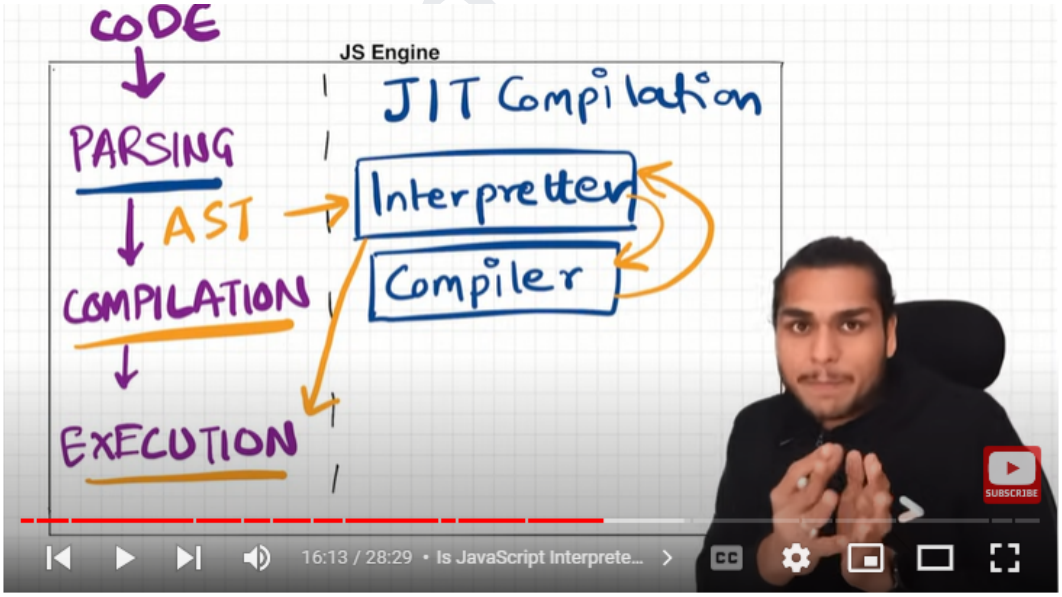
-
- **There are pros as well as cons with both interpreted and compiled languages**

3. **Is javascript interpreted or compiled language ?**
- **Javascript can behave as interpreted as well as compiled language. Everything is dependent on the JS engine**
- **When javascript was created by brendon eich it was interpreted language. The javascript engine he wrote used an interpreter to execute the code because javascript used to run majorly on browsers and browsers cannot wait for compiling the code before executing it. So, it was an interpreted language**
- **But in today's world as we use modern browsers and modern javascript engines, the javascript engine uses both compiler and interpreter.**
- **So, at present it depends on the JS engine whether it uses just in time compilation or it purely uses an interpreter.**
- **Just in time compilation :- In JIT, we use both compiler and interpreter to execute the code.**
- **In modern javascript engine compilation and interpretation go hand in hand**

- **After parsing we got the AST, now this AST will go to the interpreter. Now this interpreter converts our high level code to bytecode and the code moves to the execution phase.**

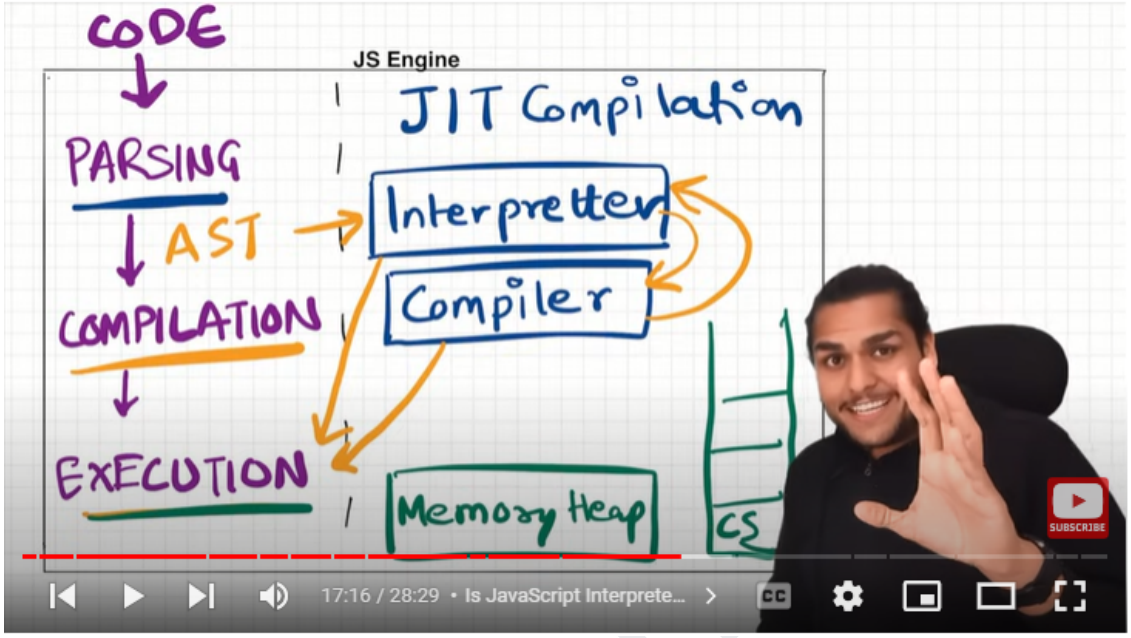


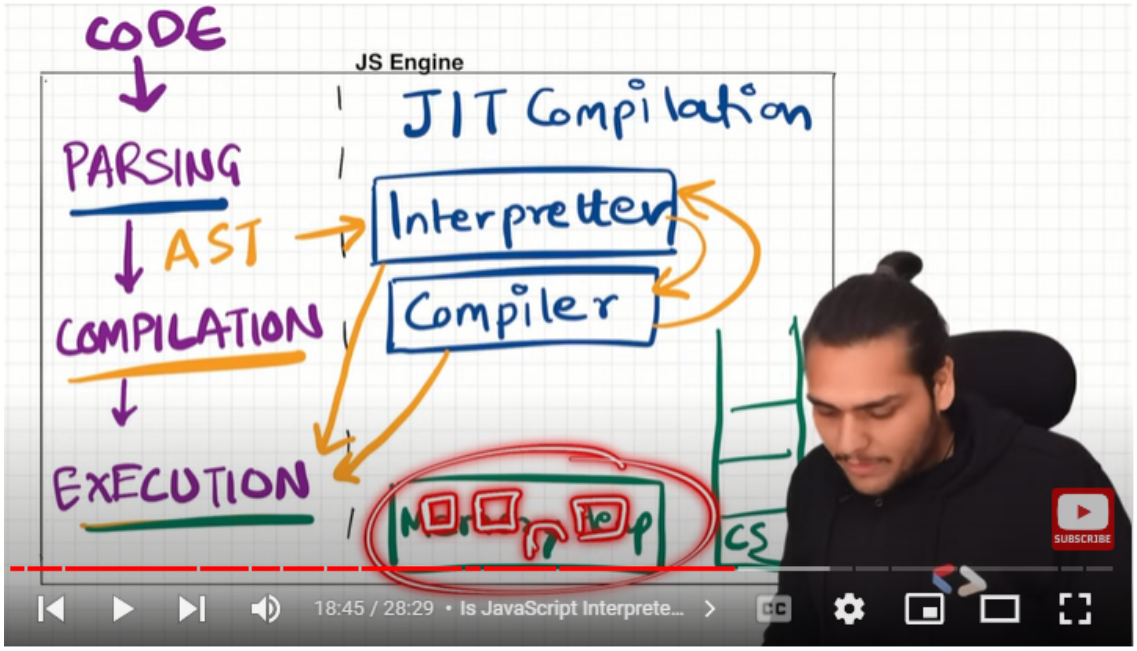
- **While the interpreter is converting the code to bytecode. It takes the help of the compiler to optimize the code. So, the compiler basically talks to the interpreter and while the interpreter is interpreting the code line by line, the compiler tries to optimize the code as much as it can, on the runtime. Therefore, it is known as just in time(Time) compilation.**

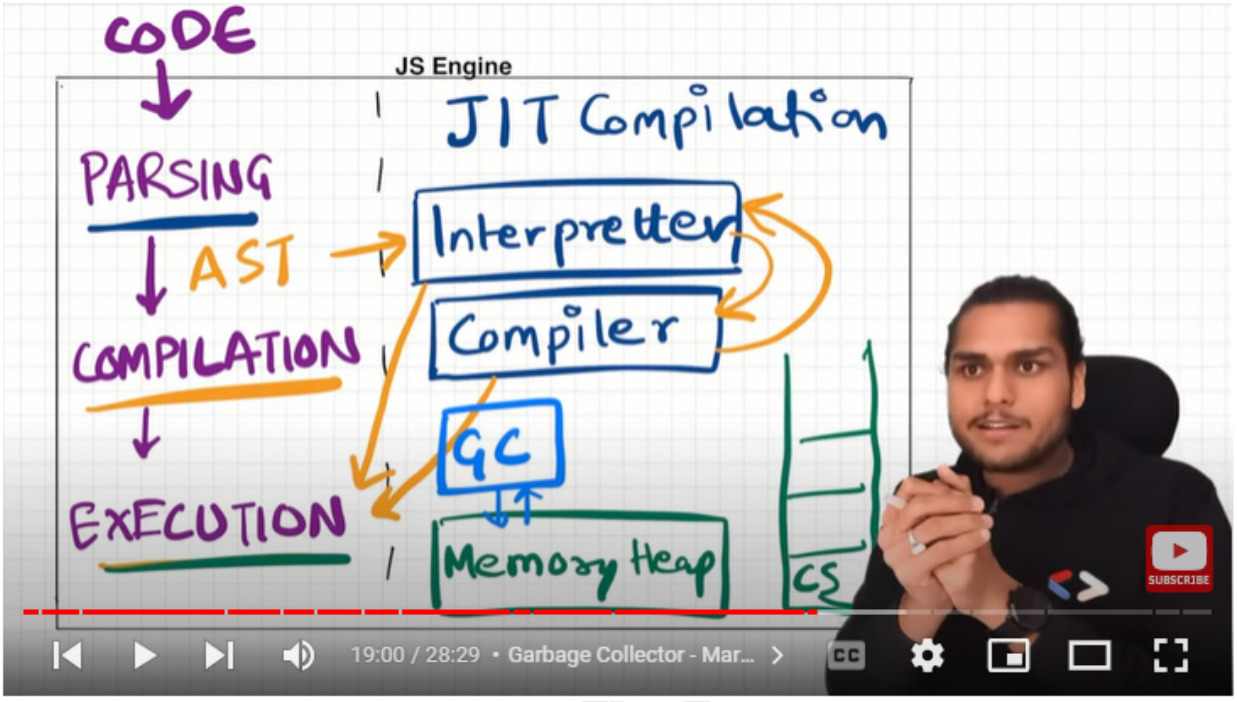


- **In some javascript engine there is Ahead of time compilation(AOT), in that case the compiler takes piece of code which is going to be executed**

later and tries to optimize as much as it can and it also produces the bytecode which then goes to the execution phase.

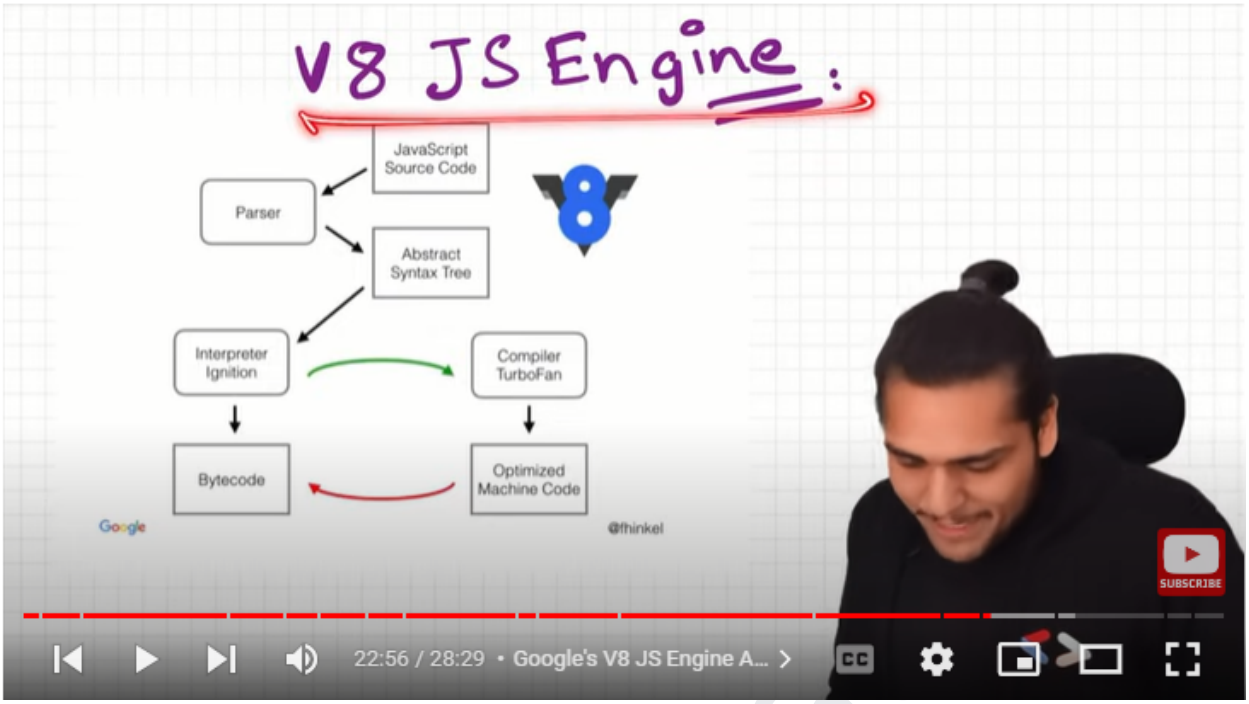
- The execution is not possible without two major components of javascript engine 1) Memory Heap 2) Call stack
- 

- Call stack is the same thing which we have studies earlier
- Memory Heap is the place where all the memory is stored. It is constantly in sync with the call stack, garbage collector and a lot of other things. Memory heap is the place where all the variables and functions are assigned memory
- 

- **Garbage collector is the thing which is used to free memory space whenever possible.**
- 

- **Garbage collector basically collects all the garbage (a function not being used or clearing the setTimeout) and sweeps it. Garbage collector uses Mark and sweep algorithms.**
- **H.W :- Read about MARK and SWEEP algorithm, Inlining, copy elision, inline caching.**

4. **Google's v8 engine.**
    - **The V8 engine has an interpreter which is known as ignition.**
    - **It also has an optimizing compiler named turbo fan.**
    - **The v8 engine architecture looks like**

- 

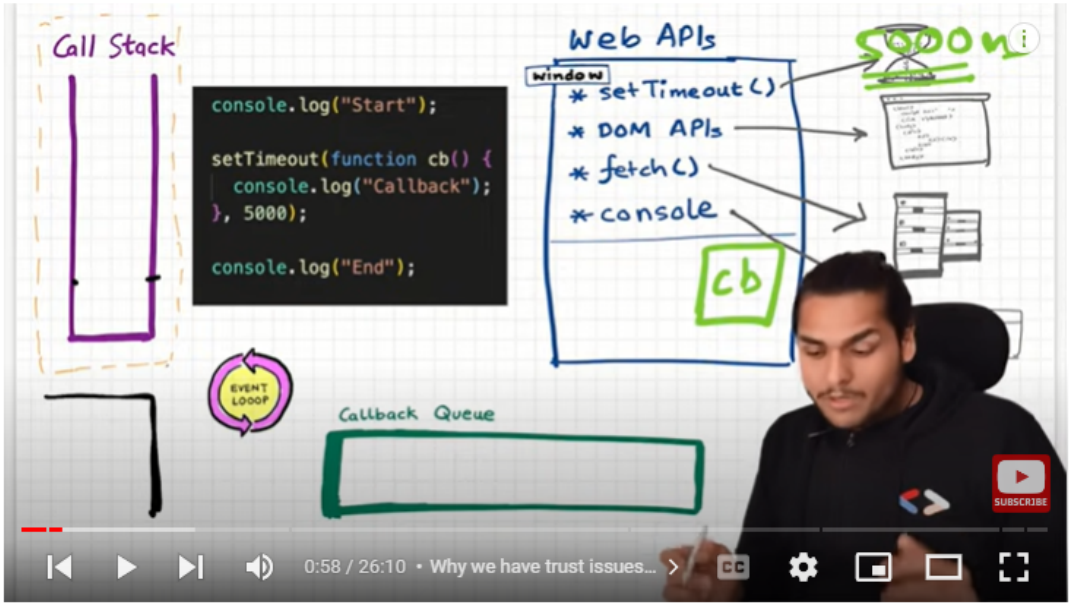
- **V8 engine has a garbage collector known as orinoco which uses mark and sweep algorithm.**

Question in Episode 17 :- TRUST ISSUES with setTimeout()

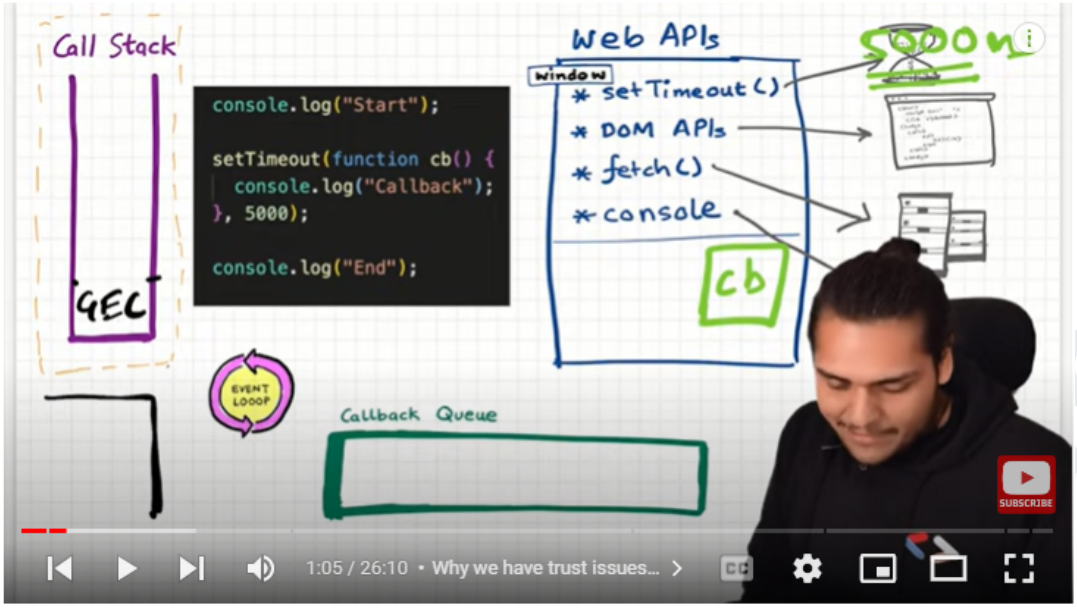
- **setTimeout has trust issues. A setTimeout of 5000 ms delay does not always exactly wait for the 5000 ms.**
- **setTimeout does not guarantee that the callback function is called exactly after 5000 ms. It might take 6000 ms or even 10 seconds. It all depends on the call stack.**

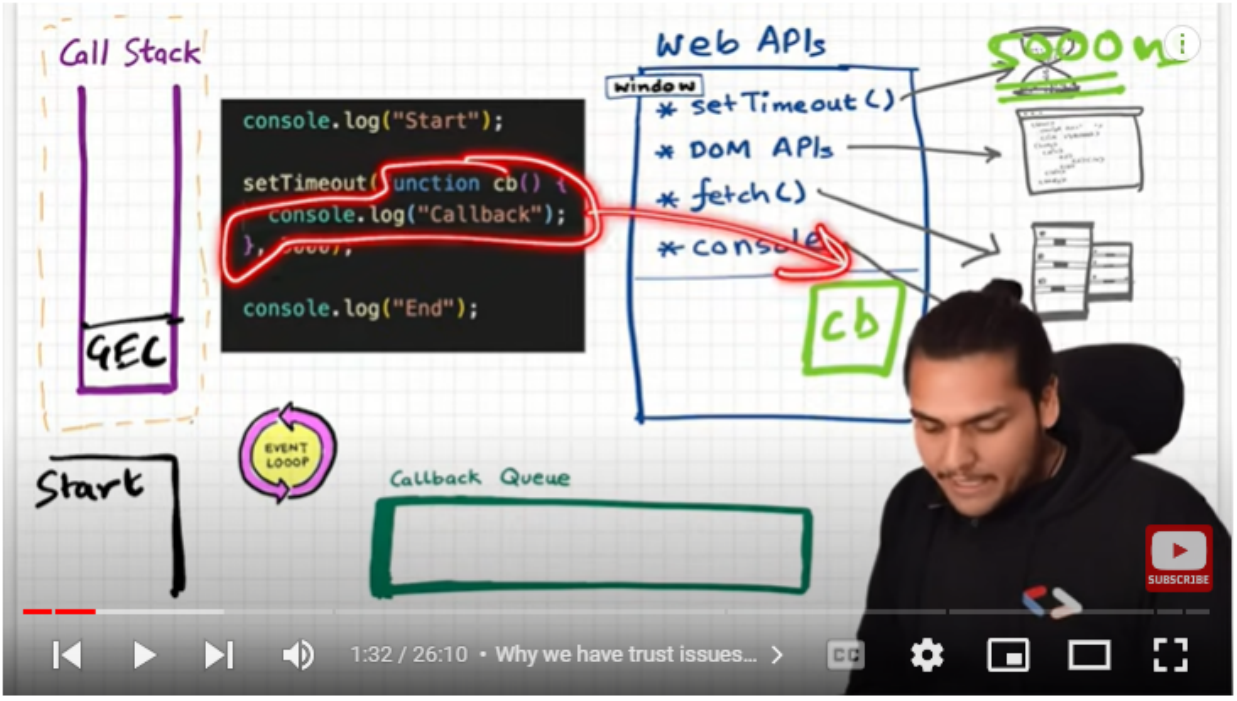
1. **Why do we have such trust issues ?**

- **Let's take an example**



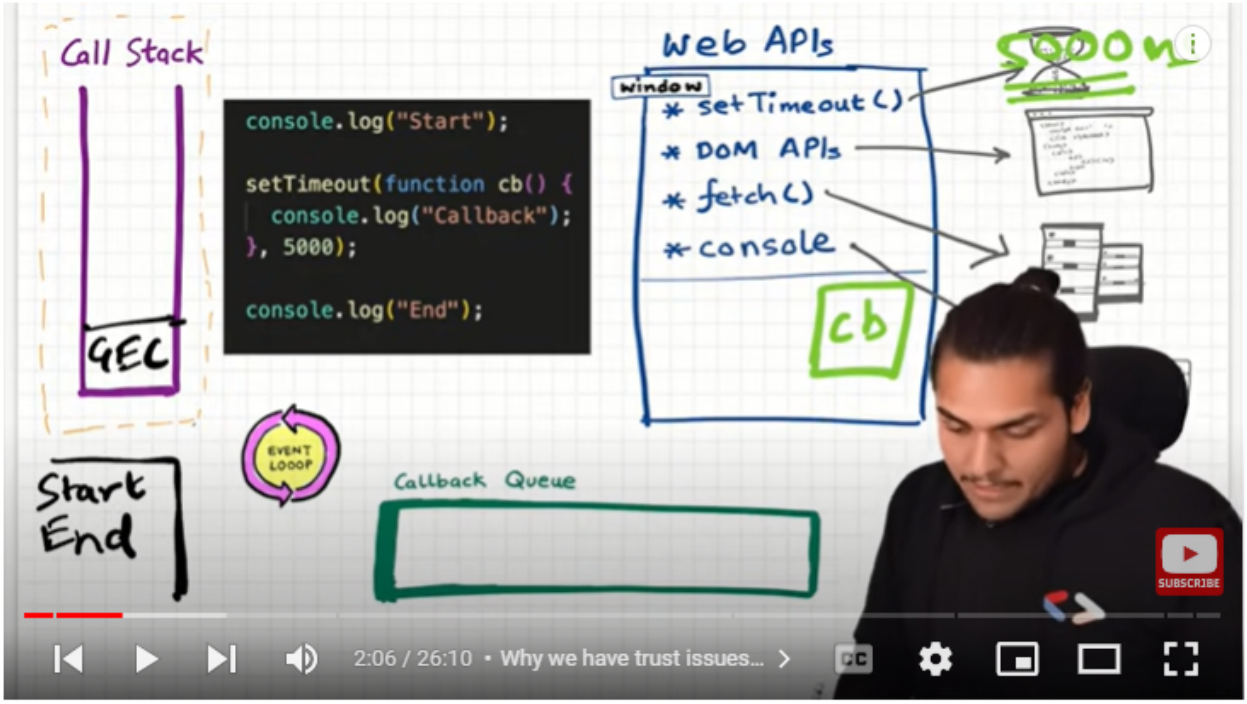
- **We know that when a javascript code is executed a global execution context is created and it is pushed inside the call stack.**

- 

- **Now, the javascript engine will run the code line by line and it will see console.log(“start”) on the first line than “start” will be printed on the console.**
- **Now the code moves to the next line and it will see the setTimeout and it will register a call back method inside the web API environment and also starts a timer of 5000ms.**
- 

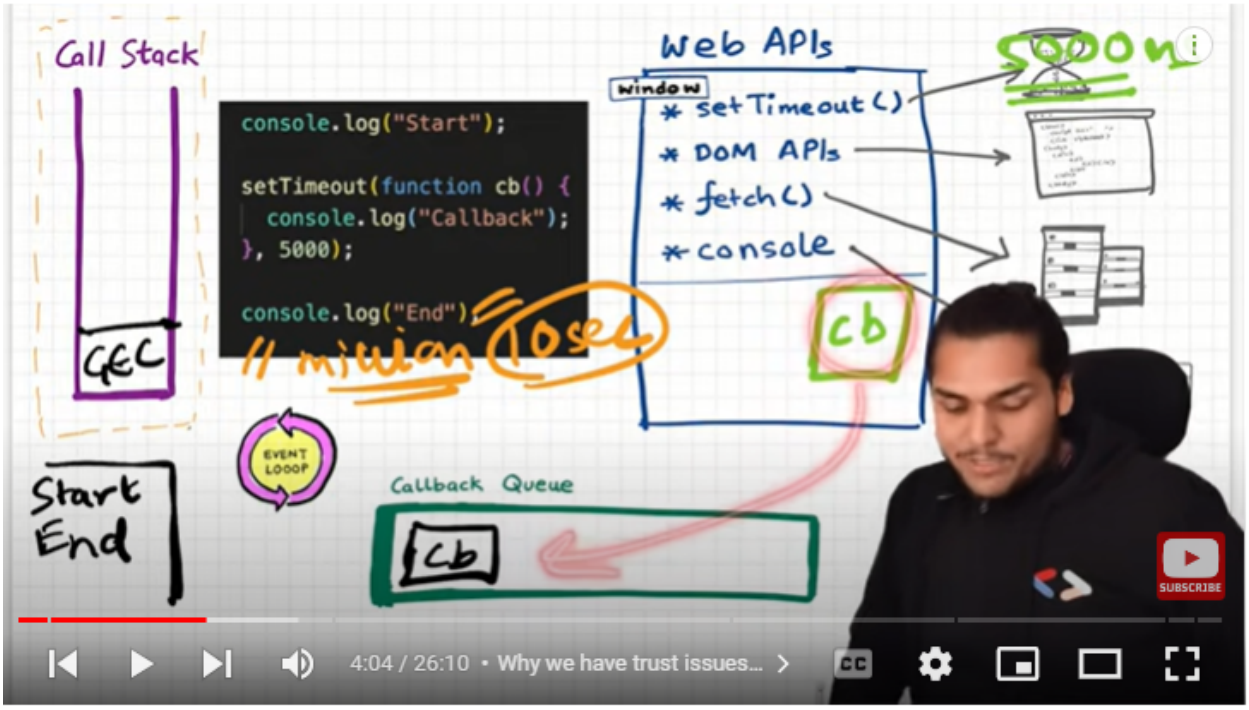
- **The callback function will wait for its turn to be executed once the timer expires.**

- **Javascript waits for no one so after registering the call back function it will move to the next line and now “end” is printed on the console.**

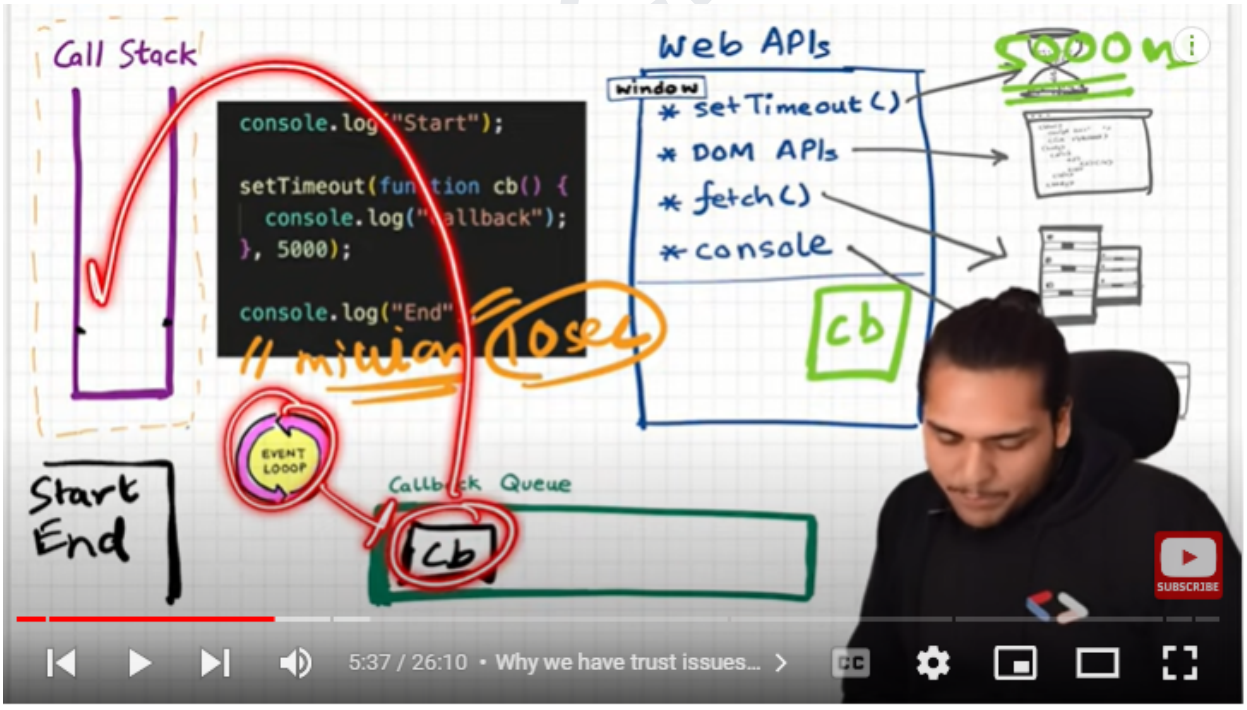


- **Let's suppose after the console.log(“end”), we have millions of lines of code written which takes a lot of time to run. Let’s say these millions of lines of code needs 10 seconds to be executed**
- **Now, the global execution context will not be popped out of the stack before it executes the millions of lines of code. So the global execution context will be popped out of the stack after 10 seconds only.**
- **While the global execution context is executing the millions of lines of codes, the timer attached to the setTimeout callback function expires as it was running for just 5000ms and now the call back function moves to the call back queue.**
- **The event loop is constantly checking whether call stack is empty or not, if it is empty than only it can take the call back queue function and put inside the call stack but in our case the global execution context does not move out of call stack before 10 seconds and the timer attached to the callback function expires in 5 seconds and the callback function which was registered inside the web API environment moves to the call back queue. As the call stack will not get empty before 10 seconds the event loop will be able to put the call back function inside the call back queue only after 10 seconds.**

- **So, after 10 seconds the global execution context will be put out of the call stack and now the event loop can put the call back function inside the call stack for execution.**

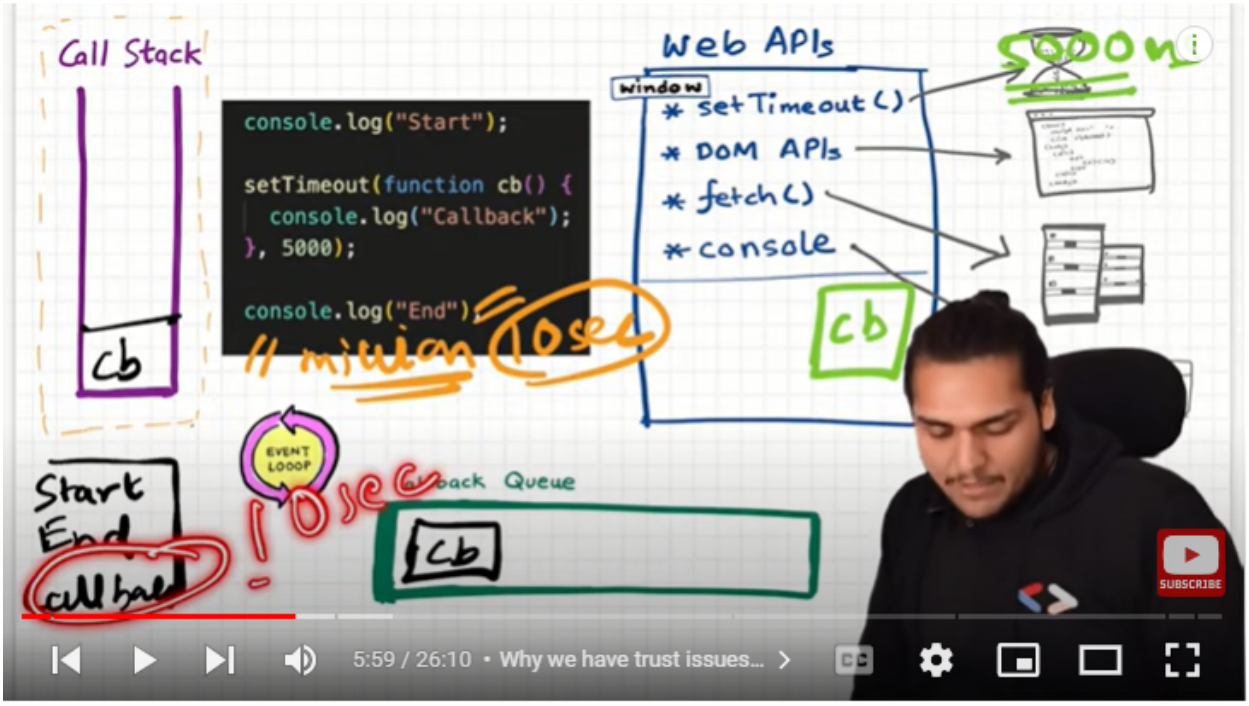


-



-

- **Now, this call back method will get a chance to execute and “callback” is printed to the console after 10 seconds**

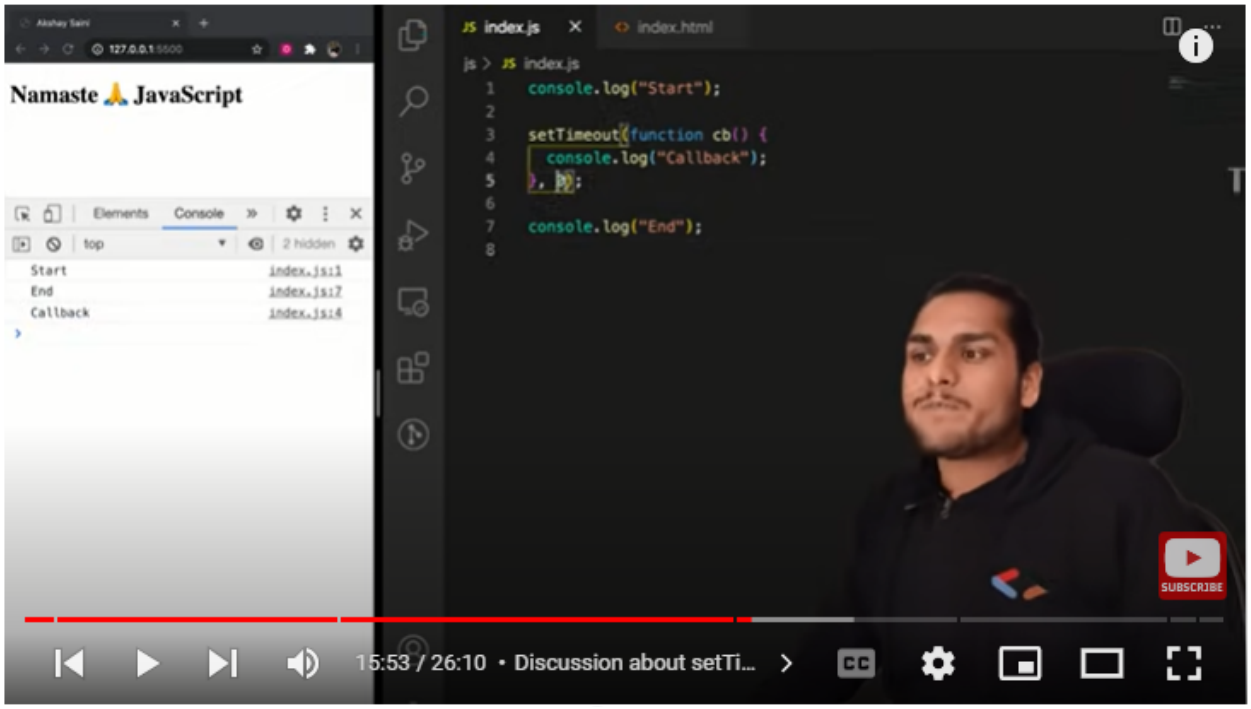
- 

- **But when we wrote the code we expected the call back function to get executed after 5 seconds but it executed after 10 seconds. So that is why setTimeout has trust issues.**
- **This is also known as the javascript concurrency model.**
- **Javascript is a synchronous single threaded language and if we want to do asynchronous operation inside the javascript than we need this concurrency model.**

2. **Why do a lot of people say that don't block your main thread ?**
   - **We should not block the call stack or we should not something in our program which is blocking the main thread because if the call stack is not empty it cannot execute any other event.**

3. **Code demonstration time stamp :- 6:52 to 15:35 for revision**

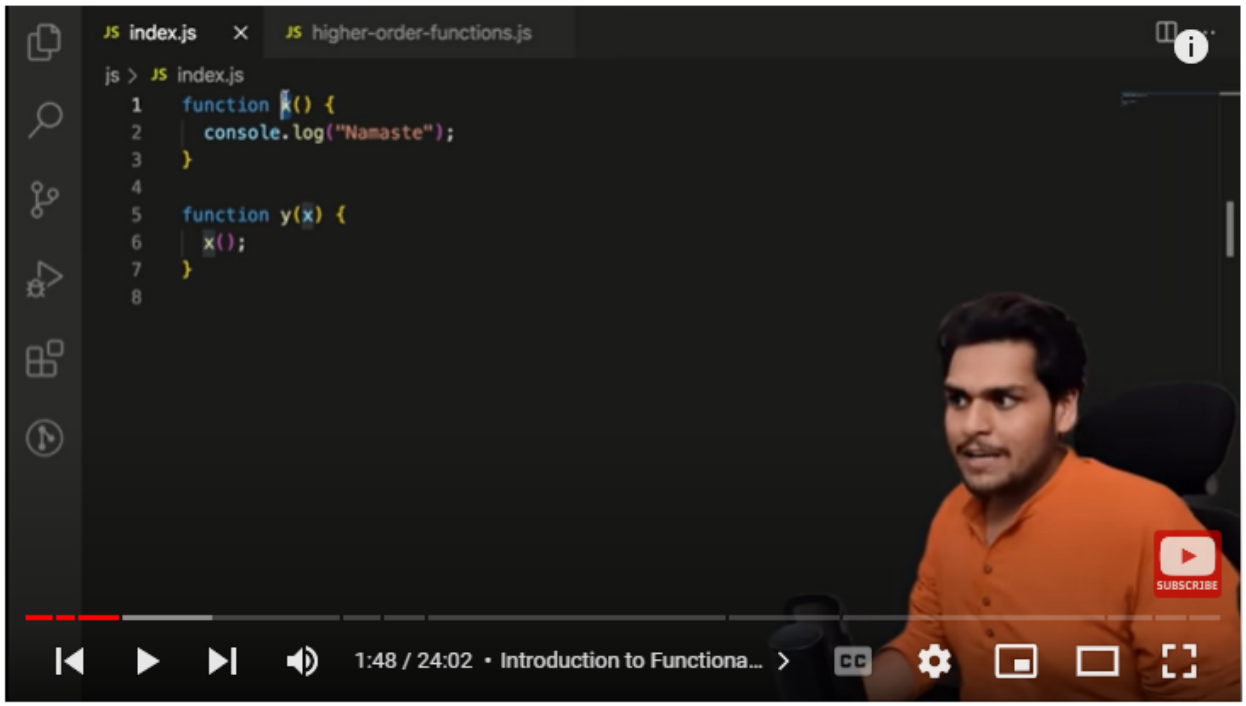
4. **What happens if we set the timer inside the setTimeout equals to zero ?**



- First “start” will be printed to the console then javascript moves to the next line. In the next line it sees the setTimeout therefore it will register a callback function in the web API environment and attach a timer of 0 seconds to it.
- Now, it will move to the next line and print “end” to the console
- Now, the global execution context will be popped out of the call stack. Meanwhile the timer expires and the call back function will move to the call back queue. Now the event loop is continuously monitoring the call stack and as the global execution context is popped out of the stack, the event loop finds that the call stack is empty, and now the event loop will move the call back function to the call stack so that the call back function can be executed.
- Now the call back function will move to the call stack and it will be executed and callback will be printed to the console.

Question in Episode 18 :- Higher-Order Functions ft. Functional Programming

1. **What is a higher order function ?**
   - A function which takes another function as an argument or returns another function from it is known as the higher order function.
- **Let's take an example**



-
- **In the above example function y() is the higher order function as it takes function x() as an argument and function x() is the call back function.**
- **Notes :- In this lecture I have learnt how we can optimize the code and also learn how we can implement our own map function. For revision watch video again according to the timestamp**

Question in episode 19 :- map, filter & reduce 🙏 Namaste JavaScript Ep. 19 🔥

1. **Map function ?**
    - **It is a higher order function which is used to transform an array**
    - **To transform each and every element inside an array and to get a new array of these transformed elements**
    - **Examples 👍**
    -