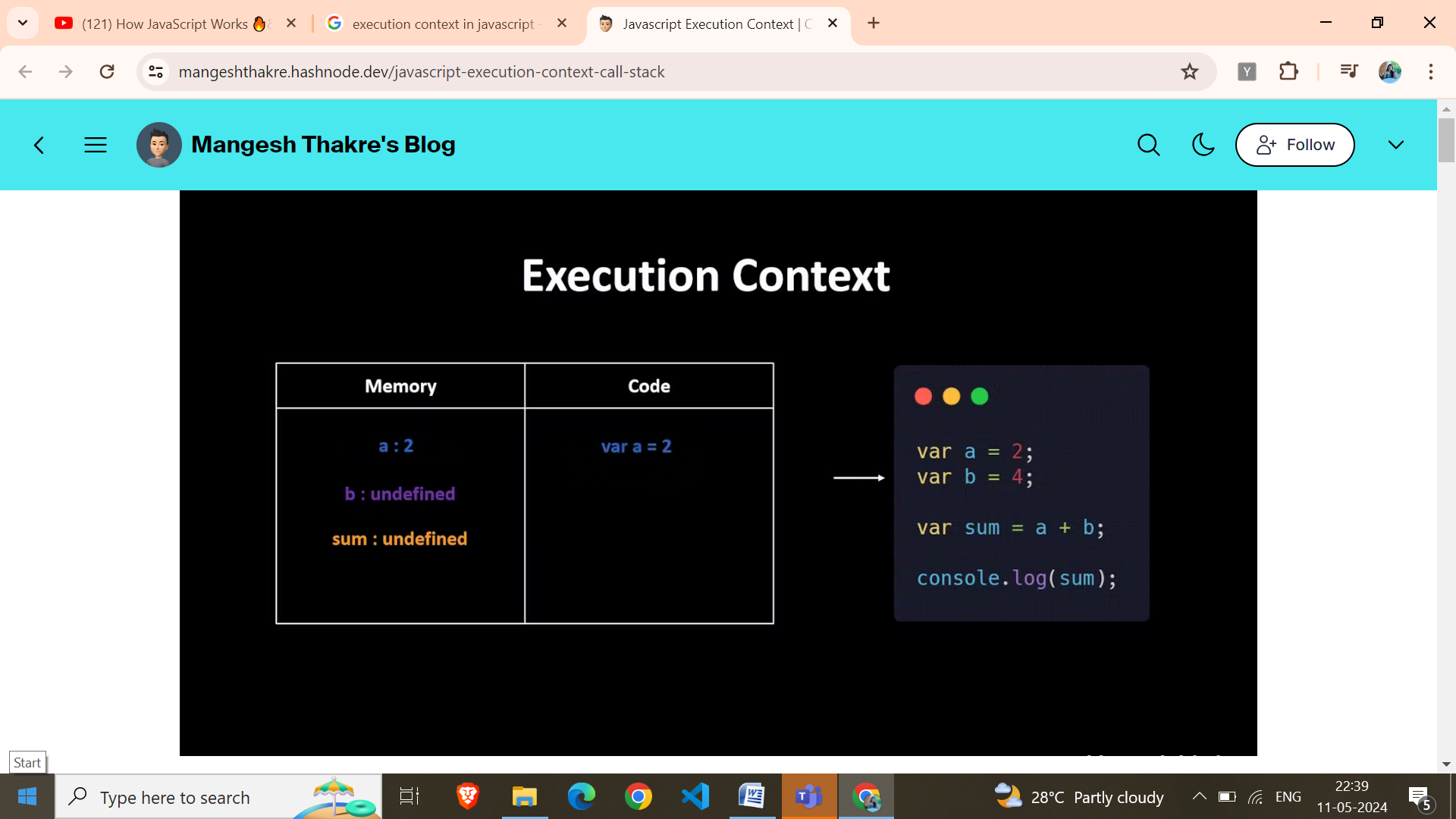
1. Everything in JS happens inside an “Execution Context”

[See how it works](https://www.google.com/url?sa=i&url=https%3A%2F%2Fmangeshthakre.hashnode.dev%2Fjavascript-execution-context-call-stack&psig=AOvVaw2l22_EXykYdpLMM9j2U1_F&ust=1715533644028000&source=images&cd=vfe&opi=89978449&ved=0CBEQjRxqFwoTCMiLp4WLhoYDFQAAAAAdAAAAABAJ)



Thread Of Execution

Variable Environment

Phase 1 – Memory Creation Phase – JS will allocate memory to all the variables and functions.

Phase 2 – Code Execution Phase – execution of code takes place.

. JS maintains a call stack for function calls.

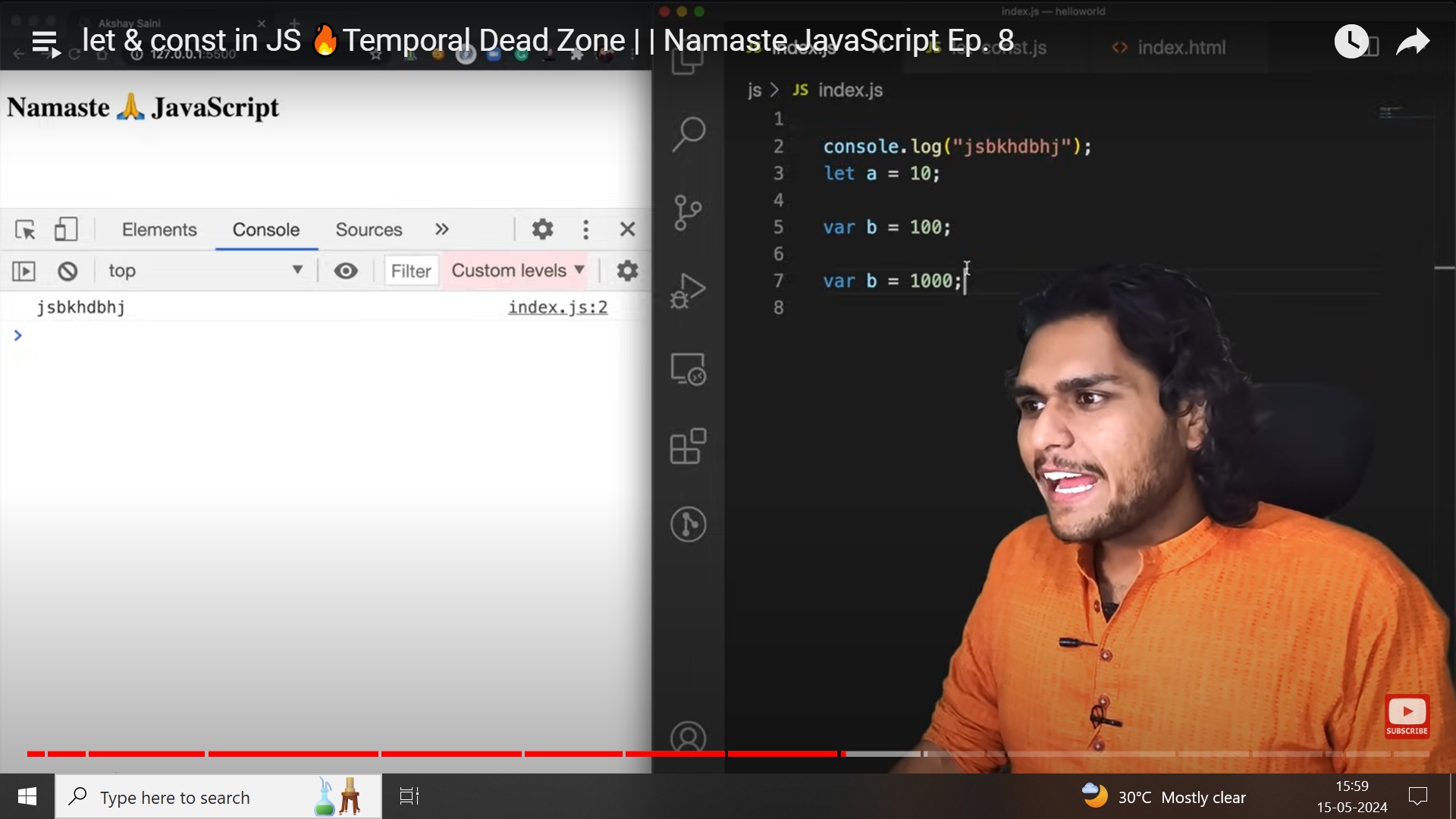
1. Hoisting

* We can access variables even before initializing(because during memory allocation phase the variables is considered with the value undefined). Always keep concept of execution context in mind.
* If the variable is not defined anywhere in the code, then it will show an error
* If we do console.log(functionname)=>it will print the whole body, since during memory execution the whole function is stored instead of just name
* But if we use arrow function, it acts like a variable name. Therefore if we do functioname() in the start of the code. It will throw an error. Telling it is not a function
* 1. Variable declarations are scanned and are made undefined 2. Function declarations are scanned and are made available
* 3. this/ window refers to the variables at global space(not inside any function).
* JS is a loosely typed language.()

4. Scope Chain – If a variable is not found in the scope, it checks for it in its parent scope and so on. If not found completely then throws an error that it is undefined.

5. let and const datatypes are also hoisted but they are in the temporal deadzone(is the time since when the let variable was hoisted till it was initialized with some values). i.e., they are also allocated memory but is is not stored in the global space but in a separate memory space and we cannot access this memory space before we have put some value in it.

🡪 var is stored in the global space and can be acces using window object but let cannot be accessed from window object.



Re-declaration of any var variable is allowed but not for let. It will throw an error.

6. Block is also known as compound statement(it helps to keep together multiple statements toghether). let and const are block scoped but var is in global scope.

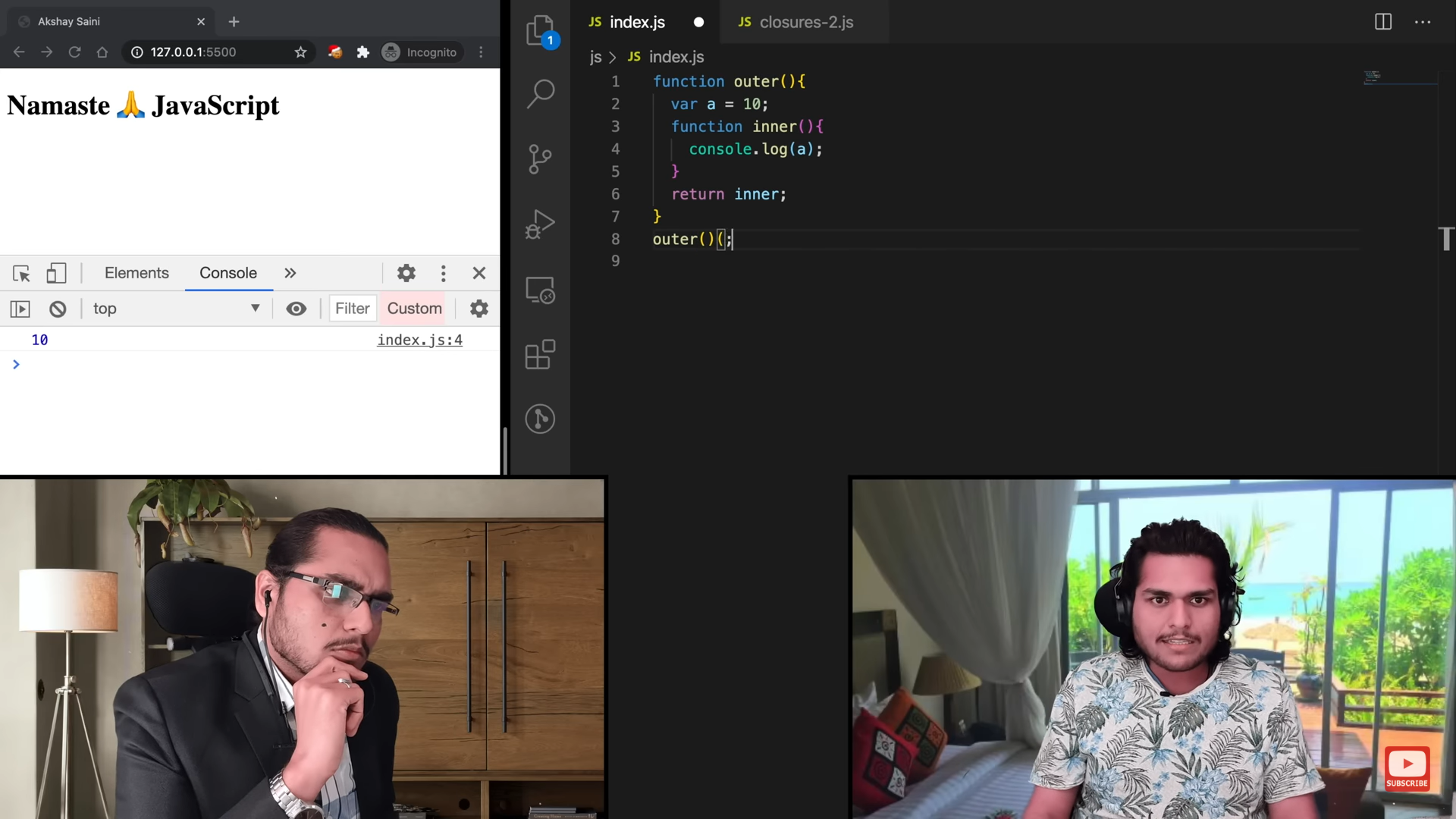
7. Functions along with its lexical scope are **closure.** Whenever a function returns a function, it is returned with its lexical scope.

Lexical Scope?-First the code tries to find a variable in the local scope if not found it searches it in the parent scope and so on.

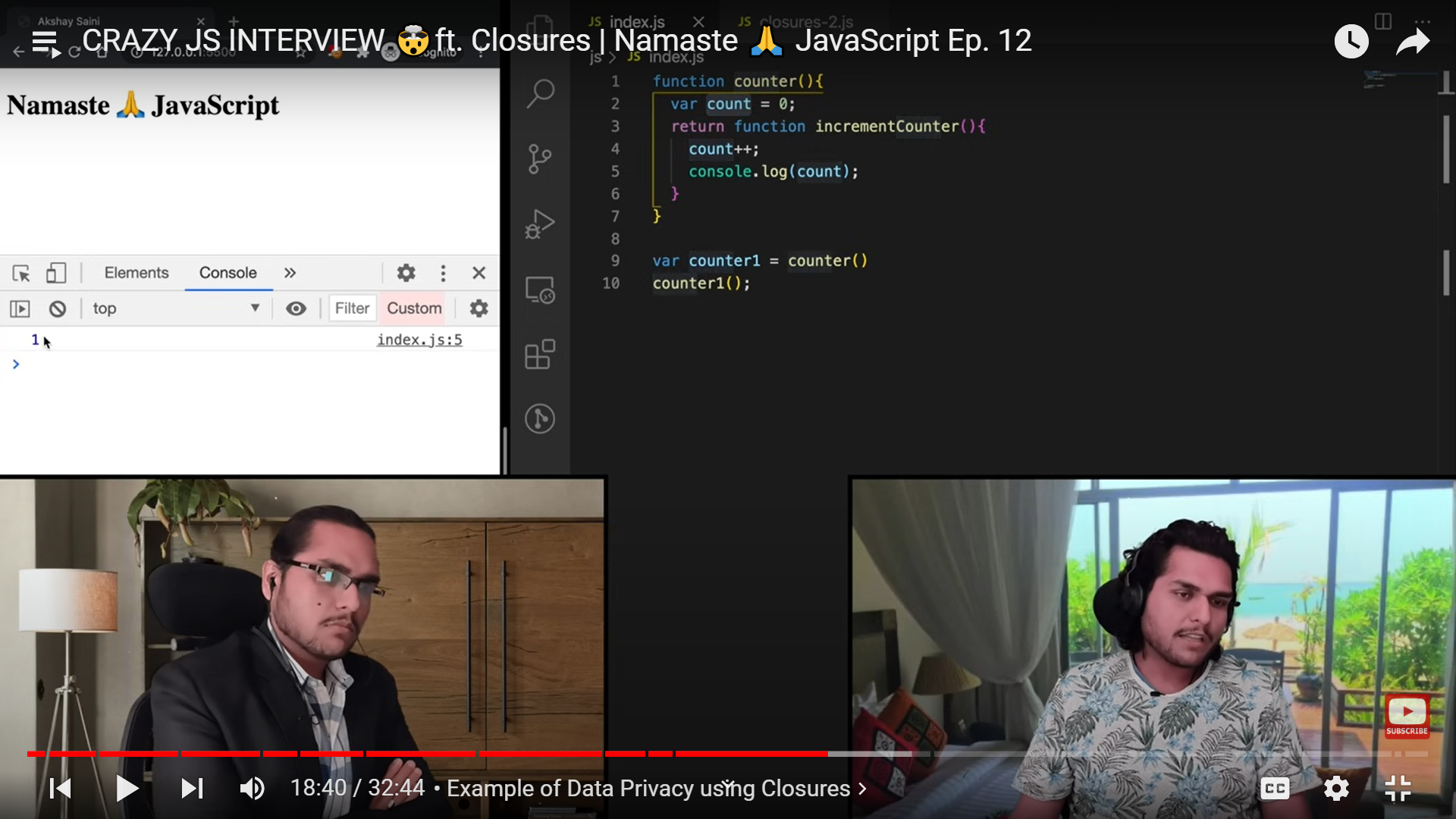


This will print 7, even we have not called x(). But z will store the function as well as its lexical scope, i.e., closure.

[Video on closure Example](https://youtu.be/eBTBG4nda2A?si=kL2cNcmGu9-KY2i3)



Data Hiding using closures



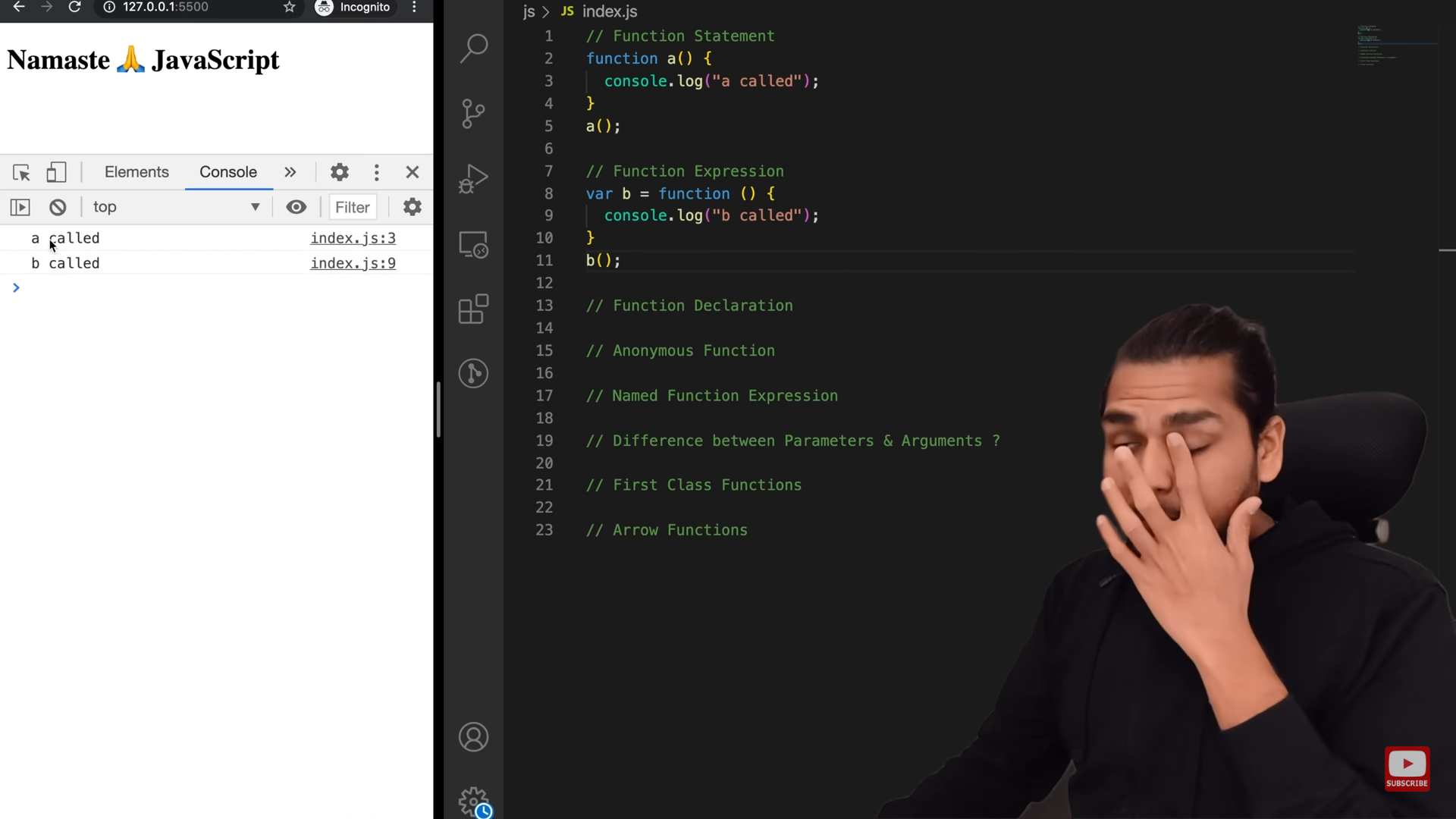
8. Functions without a name are anonymous function.

Difference between function statement and function declaration is hoisting.

Function declaration and statement are same

Named Function declaration:

var b= function xyz() {…}



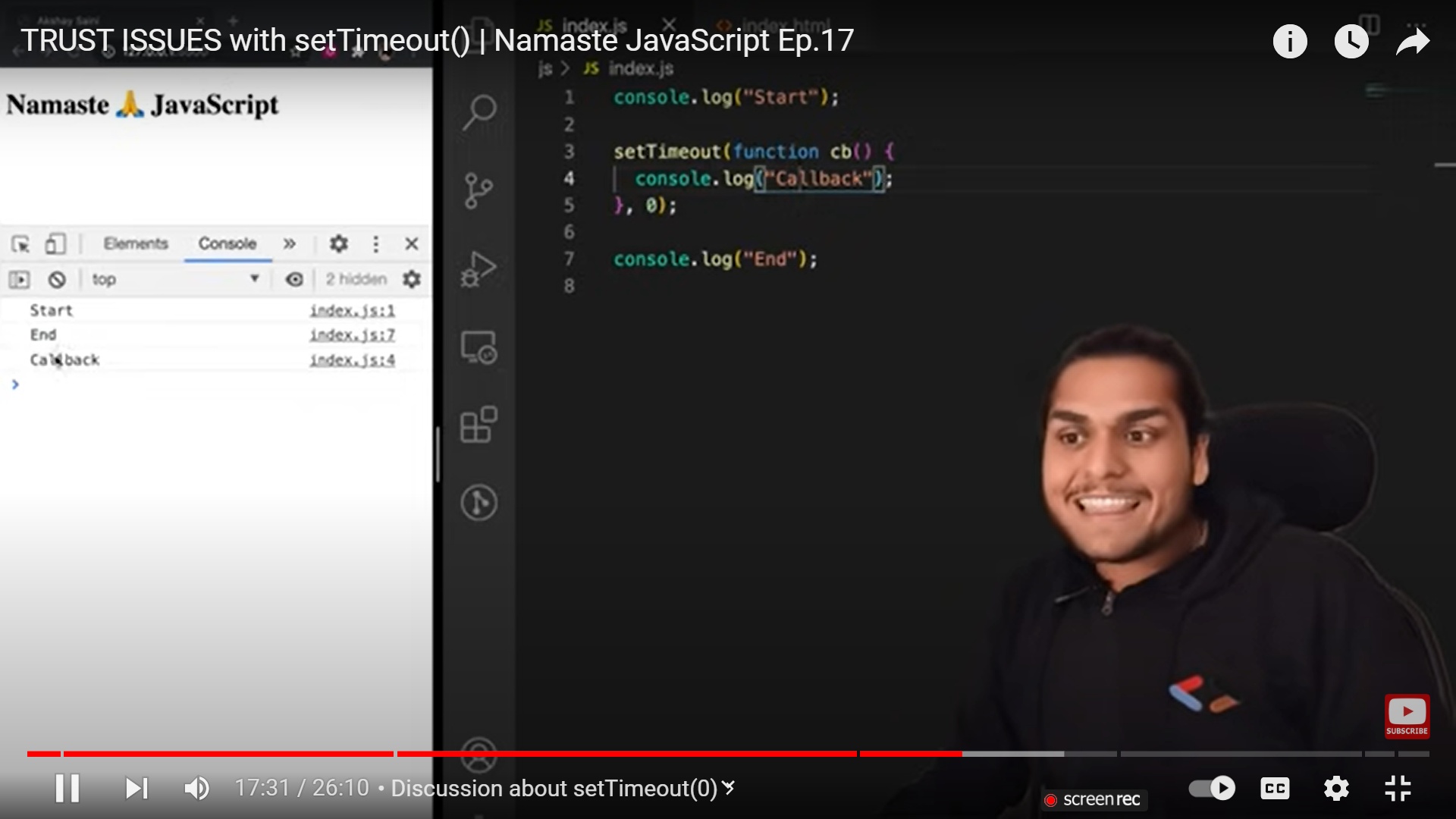
9**. First Class functions –** The ability to use functions as value, i.e., to pass function as arguments, to return functions from a function is known as first class functions

10. 1. Function that is passed on as argument to another function is called **callback function**.   
2. setTimeout helps turn JS which is single threaded and synchronous into asynchronous.   
3. Event listeners can also invoke closures with scope.   
4. Event listeners consume a lot of memory which can potentially slow down the website therefore it is good practice to remove if it is not used.

11. fetch requests an api call from the url mentioned. It returns a promise and we need to pass a callback function when promise is returned.

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

13. setTimeOut function’s body gets into the callback queue and once the timer expires and the body gets in the main call stack when the call stack becomes empty.



14. A function that takes a function as an argument or returns a function from it is known as **higher order function.**

**15. Reduce** is used when we need one value out of the whole array. Ex- largest, sum of array

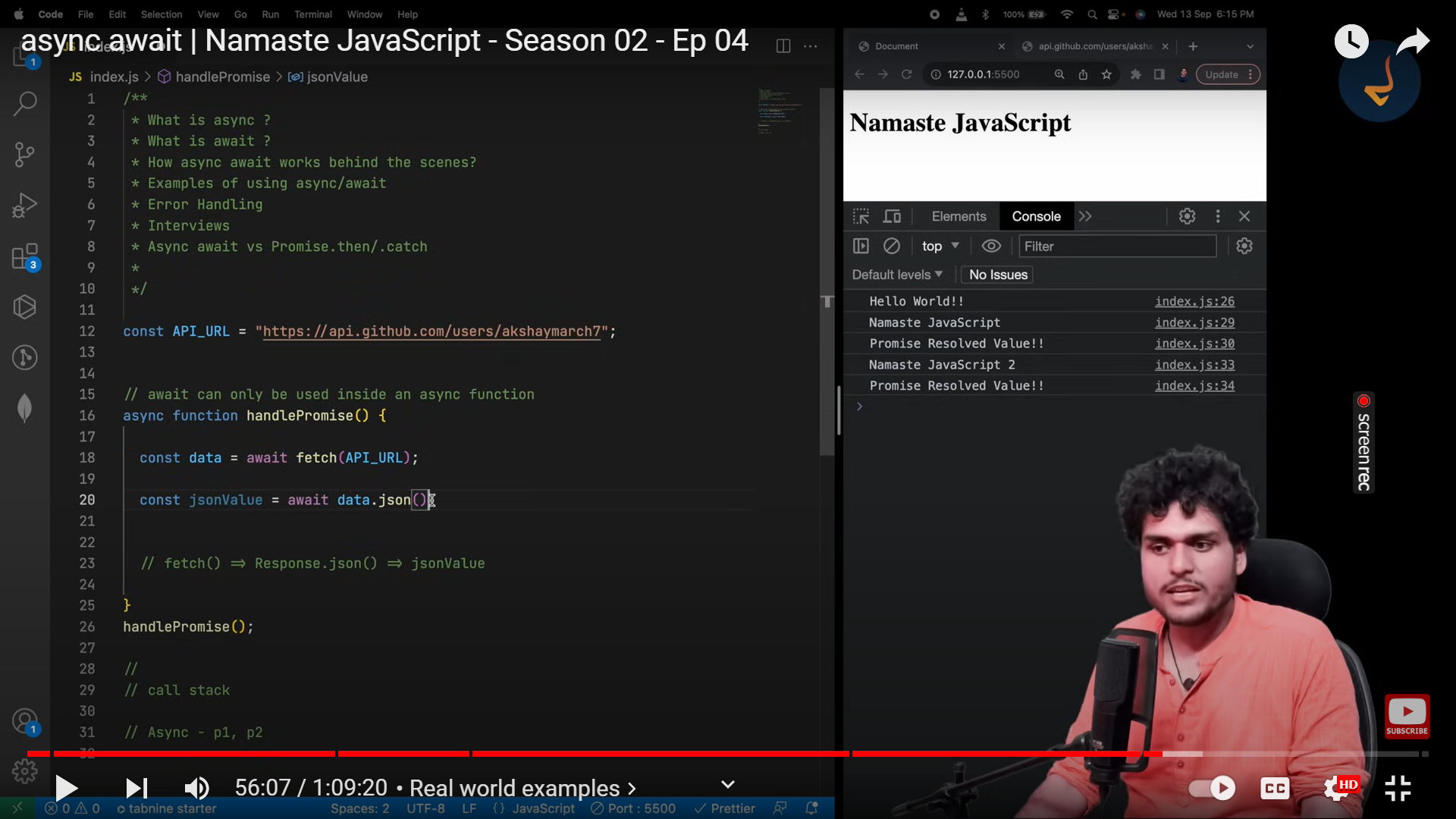
16. **Callbacks –**

* Can lead to callback hells.(To prevent this we use Promise Chaining)
* Inversion of control – When we lose control over our code when using callbacks

17. **Promise –**

* A promise is an object that represents eventual completion/failure of an asynchronous operation.
* A promise has 3 states: pending | fulfilled | rejected. As soon as promise is fulfilled/rejected => It updates the empty object which is assigned undefined in pending state.
* A promise resolves only once and it is immutable
* Using .then() we can control when we call the cb(callback) function.
* A very common mistake that developers do is not returning a value during chaining of promises. Always remember to return a value. This returned value will be used by the next .then()

18. The fetch() method returns a Promise that resolves to a Response object. **Fetch** function is a promise that gives response. This response is again a promise. We need to convert this response to json using .json().



19. **Promise** can be created in this way

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

        if(!valideCart(cart))

            {

                const err=new Error('Validating failed')

                reject(err)

            }

        const orderId = 1234

        if(orderId)

            {

                resolve(orderId)

            }

    })

    return pr

to handle errors in promises 🡪

createOrder(cart)

.then((orderId)=>{

    console.log(orderId)

    return proceedToPayment(orderId)

})

.then(function(paymentInfo){

    console.log(paymentInfo)

    // return orderSummary(paymentInfo)

})

.catch((err)=>{

    console.log(err.message)

})

* P1🡪 2s P2🡪1s P3🡪 3s
* Promise.all([p1,p2,p3]) 🡪 promise.all takes an array of promises and returns an array of their responses. If any of them throw an error then it throws an error. Throws an error as soon as any of the promise throws error or waits for all of them to complete.
* Promise.allSettled([p1,p2,p3]) 🡪 returns list of responses of the successful promises. It doesnot throw an error if some of them gets rejected. It waits for all the results then shows the ans.
* Promise.race([p1,p2,p3]) 🡪 As soon as the first promise is settled(success/ failure), it will give the value of the winner. According to the example P2 will be winner.
* Promise.any([P1,P2,P3]) 🡪 Itt will wait for the first promise to get successful and give the response. If all the promises get failed then it will give array of all the errors.

20. **Async await – used to handle promises**

* Async functions always returns a promise. If not returning promise. The value automatically gets converted to promise.
* Await can only be used inside async function.
* Why use this? 🡪 JS engine doesnot wait to to get a response from a promise and run the rest of the code. But in async await, if we have used await the lines after that will be executed until it gets a response from the promise
* Use try catch whenever using await.