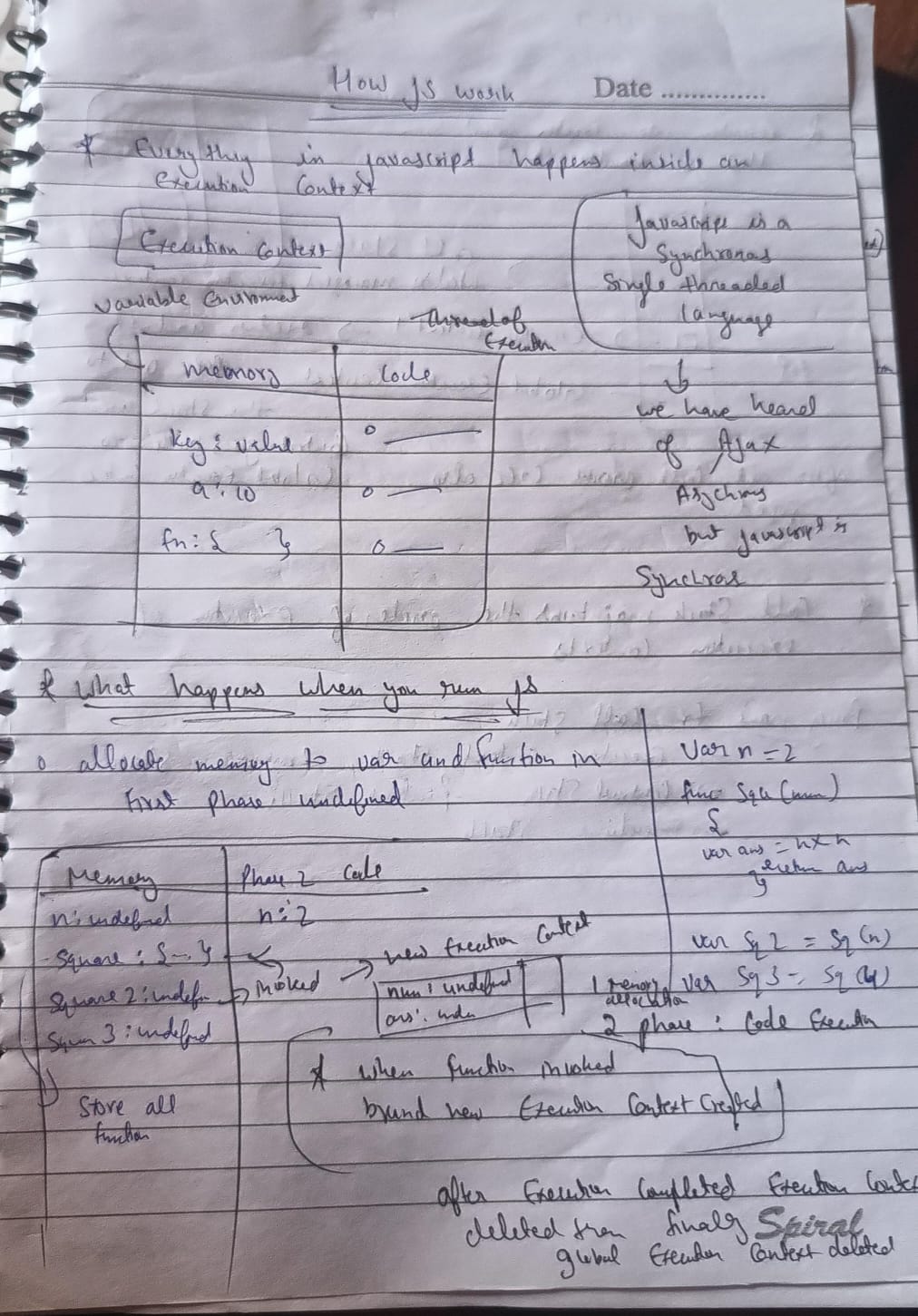
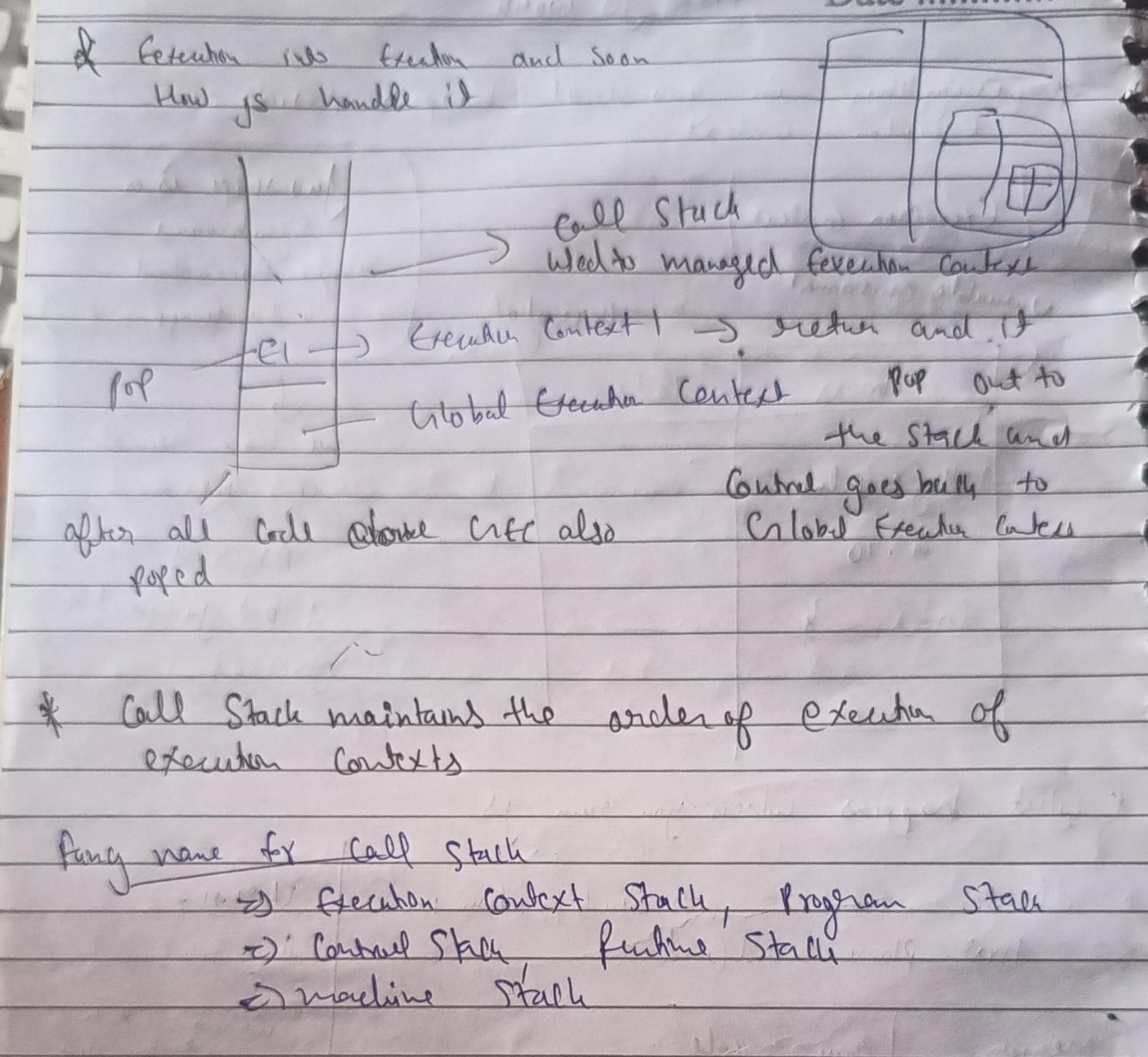
Execution context





# 

# **Hoisting**

Hositing in js is a process in which all the var fun class definitions are declared before execution of the code

1. In JS, before the code is executed, the variables get initialized to undefined.

2. Arrow functions enact as variables and get "undefined" during the memory creation phase while functions actually get run.

3. Hoisting: Mechanism in JS where the variable declarations are moved to the top of the scope before execution. Therefore it is possible to call a function before initializing it.

4. Whenever a JS program is run, a global execution block is created, which comprises of

2: Memory creation and Code execution.

Golden Rules:

1. Variable declarations are scanned and are made undefined

2. Function declarations are scanned and are made available

Undefined means variable has been declared but value is not assigned

Not defined means variable is not declared

IMPORTANT-let const also hoisted

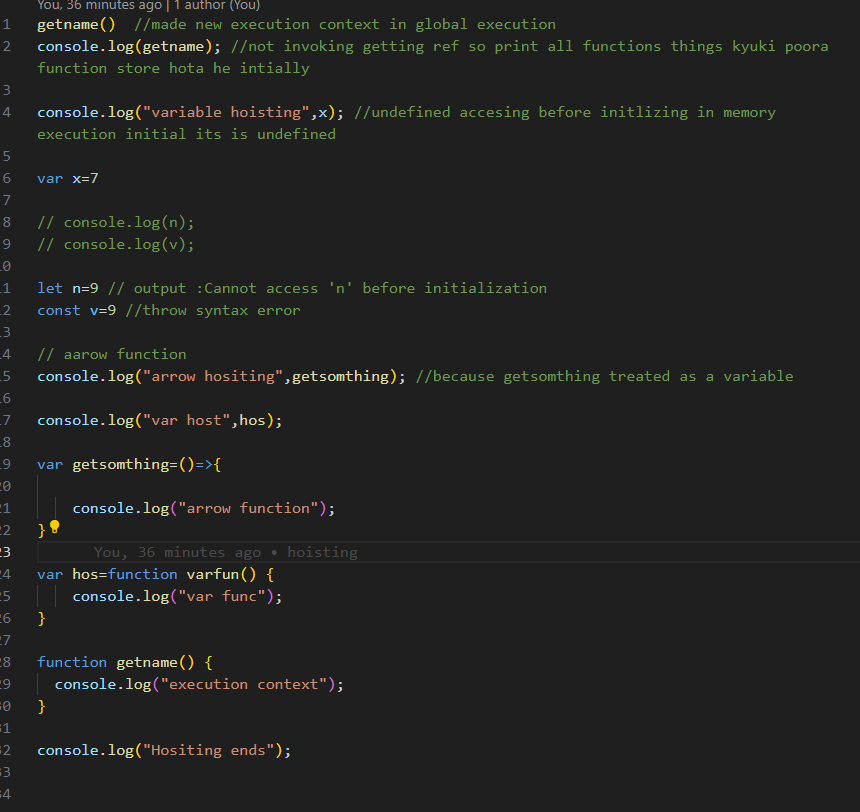
The block of code is aware of the variable, but it cannot be used until it has been declared(initialize).

* Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

* Using a const variable before it is declared, is a syntax error, so the code will simply not run.

code



let & const in JS 🔥Temporal Dead Zone

1. let and const are hoisted but its memory is allocated at other place than window which cannot be accessed before initialisation.

2. Temporal Dead Zone exists until variable is declared and assigned a value.

3. window.variable OR this.variable will not give value of variable defined using let or const.

4. We cannot redeclare the same variable with let/const(even with using var the second time).

5. const variable declaration and initialisation must be done on the same line.

6. There are three types of error: [1] referenceError {given where variable does not have memory allocation} [2] typeError {given when we change type that is not supposed to be changed} [3] syntaxError {when proper syntax(way of writing a statement) is not used}.

7. Use const wherever possible followed by let, Use var as little as possible(only if you have to). It helps avoid error.

8. Initialising variables at the top is good idea, helps shrinks TDZ to zero.

/////////////////////////////////////////////////////////////////////////////////////////////

**How functions work in JS ❤️ & Variable Environment**

**1. We learnt how functions work in JS.**

**2. At first a global execution context is created, which consists of Memory and code and has 2 phases: Memory allocation phase and code execution phase.**

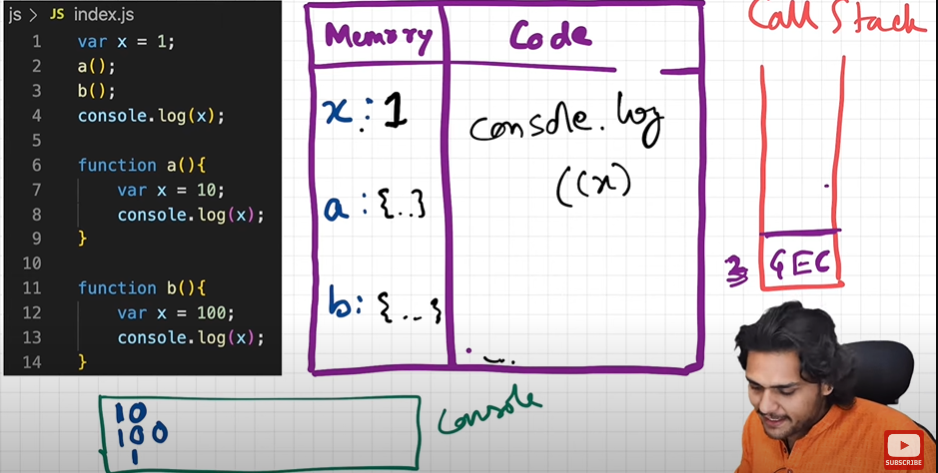
**3. In the first phase, the variables are assigned "undefined" while functions have their own code.**

**4. Whenever there is a function declaration in the code, a separate local execution context gets created having its own phases and is pushed into the call stack.**

**5. Once the function ends, the ExecutionContext is removed from the call stack.**

**6. When the program ends, even the global EC is pulled(pop) out of the call stack.**

**When log (x ) its find in local memory of particular execution context first then global**

****

**------------------------------------------------------------------------**

**Shortest js program (empty js file)**

• window object is created by the JS engines of the respective browsers when global execution context is created.

• whenever an execution context is created a "this" variable is also created.

• at the global level "this" points to the global object( window object in case of browsers).

• anything that is not inside a function is the "global space".

• whenever we create any variables or functions in the "global space", they get attached to the global object( window object in case of browsers).

so to access the variables/function defined in the global space ,

we can use any of the below:

console.log(window.a);

console.log(a);

console.log(this.a) //at the global space level, where this points to the window object

------------------------------------------------------------------------------------------------------------------

undefined and not defined

1.Undefined is a Special Placeholder which is used to reserve memory for the variables in the memory creation phase. Even before a single line of code is executed JS engine assigns undefined to the variables.

2.Not Defined means if we try to console or access any variable or fun which is not declared in the code then we get Not Defined error.

3. JS is a loosely typed language or weakly typed language means it does not attaches its variables to specific data types like in C++ and java.

4.Remember undefined !== not defined.

The Scope Chain, 🔥Scope & Lexical Environment

Scope – the current context or environment in which all variable are declared

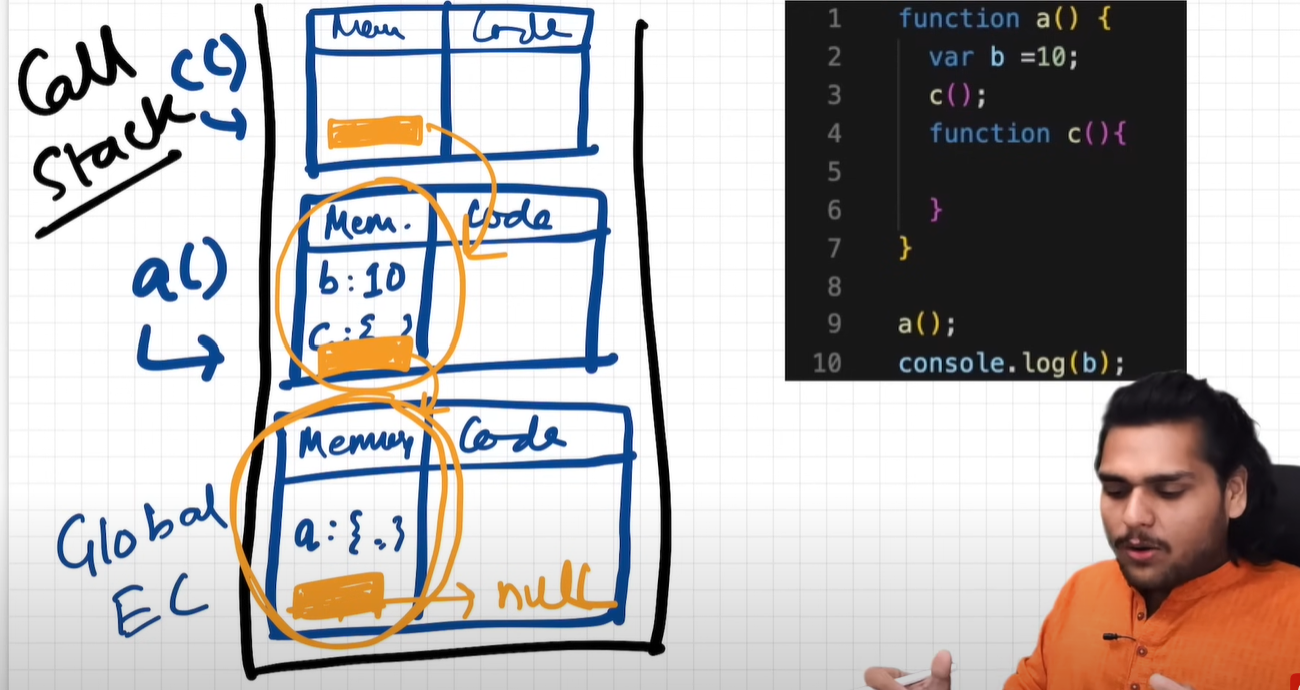
1. Scope of a variable is directly dependent on the lexical environment.

2. Whenever an execution context is created, a lexical environment is created. Lexical environment is the local memory along with the lexical environment of its parent. Lexical as a term means in hierarchy or in sequence.

3. Having the reference of parent's lexical environment means, the child or the local function can access all the variables and functions defined in the memory space of its lexical parent.

4. The JS engine first searches for a variable in the current local memory space, if its not found here it searches for the variable in the lexical environment of its parent, and if its still not found, then it searches that variable in the subsequent lexical environments, and the sequence goes on until the variable is found in some lexical environment or the lexical environment becomes NULL.

5. The mechanism of searching variables in the subsequent lexical environments is known as Scope Chain. If a variable is not found anywhere, then we say that the variable is not present in the scope chain.



//////////////////////////////////////////////////

Q) What is block in JavaScript?

> multiple js statements formed in a group enclosed in brackets and it forms a block

Q) What is need of a block/Grouping?

Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if, else, loop, function etc.

> JavaScript sometimes expect to run a single statement to run, but we need to run commands with multiple statements which is only possible by block

eg. on 4:14

write a simple function:

// even empty script is perfectly valid js script, what about empty brackets!!

{

var a = 10;

let b = 20;

const c =30;

}

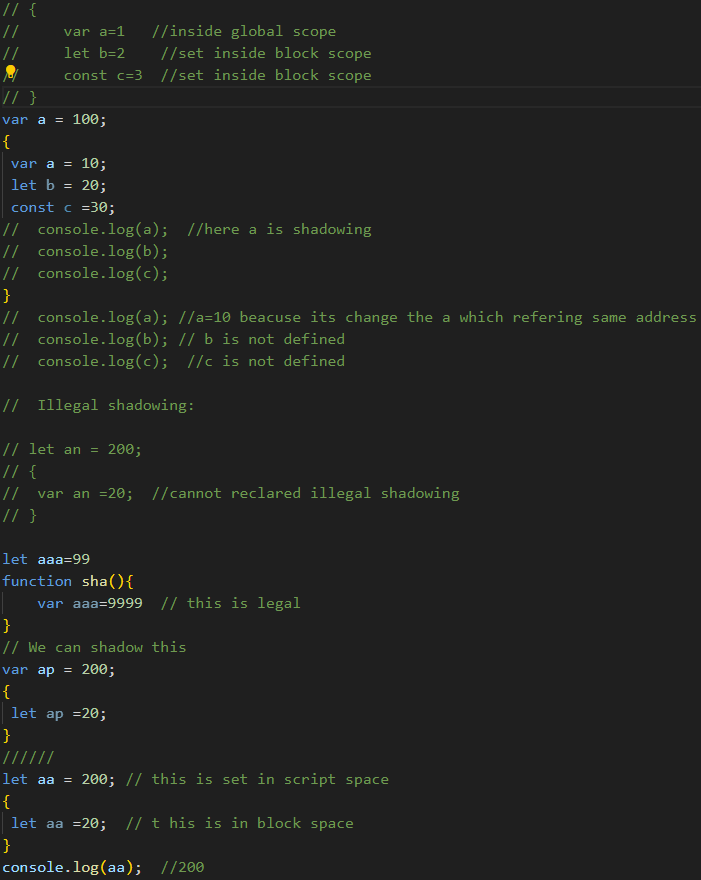
When a js script get hoisted (a Global Execution Context) gets created 'var' listed towards 'Global environment' and other variables 'let' and 'const' declarations go to the 'Block environment'

This become especially important when deciding the scope of a particular variable, since b and c are located in 'Block environment' and for a as we know exists in 'Global environment' any statement out of the "Block" can access 'a' ie. ' Variable in Global environment' and other are not!

so when we understand the extent of Global and local environment variables and their 'Scopes' == Environment that forms the lexical hierarchy of 'Scopes' and 'Scopes' have Levels like 'Scope inside scope'

Shadowing :- Providing same name to the variable as of those variable which are present in outer scope.

So in block " var a = 10;" influences the value within the block hence console.log(a); >> 10 and outside of the block 'Variable in Global environment' influences value of a hence console.log(a); >> 100



as 'var' declaration goes to 'Global environment' and sets in Memory context, it cannot be set using 'Block environment' value Hence: Uncaught SyntaxError: Identifier 'a' has already been declared

-------------------------------------------------------------------------------------------------------

**Closures**

**Function bundled with its lexical environment is known as a closure.**

**Whenever function is returned, even if its vanished in execution context but still it remembers the reference it was pointing to. Its not just that function alone it returns but the entire closure and that's where it becomes interesting !**

**A closure gives access to all the variables of it's parent function even after the that parent function has returned or executed. The function keeps a refernce to it's outer scope which preserves the scope chain throughout the time**

**­­**

**Uses of closures**

**-module design pattern**

**-currying**

**- functions like once**

**-maintaining state in async world**

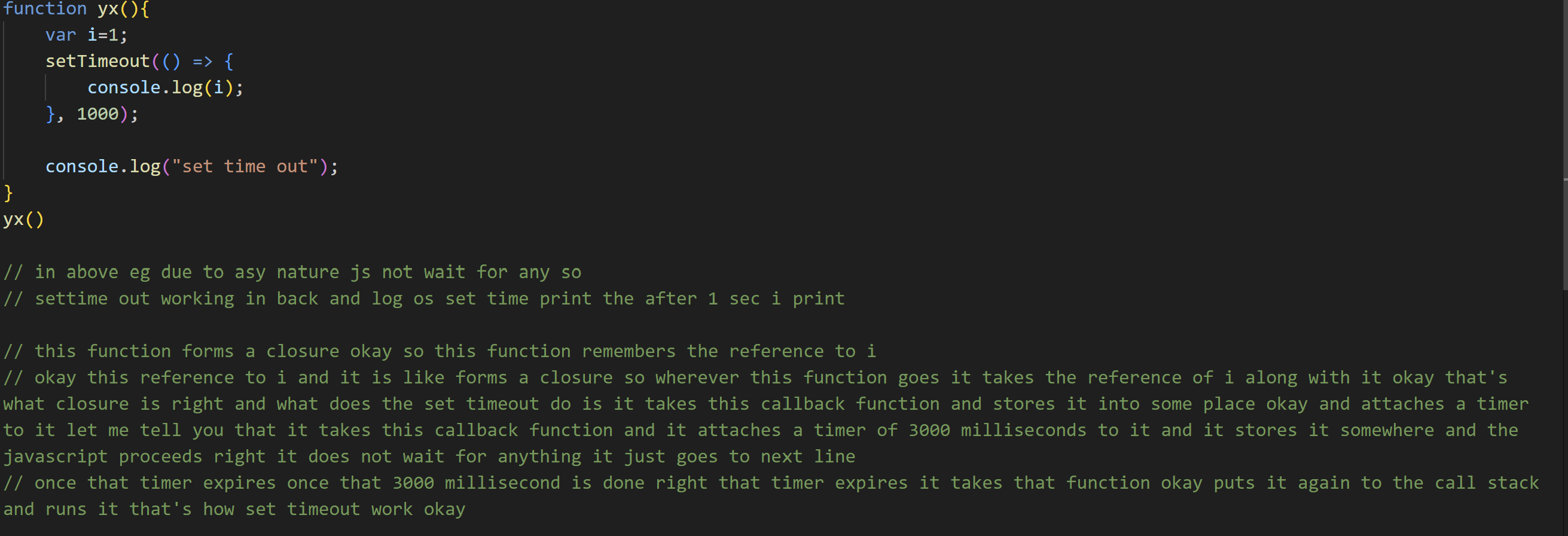
**-memoize**

**-settimeouts**

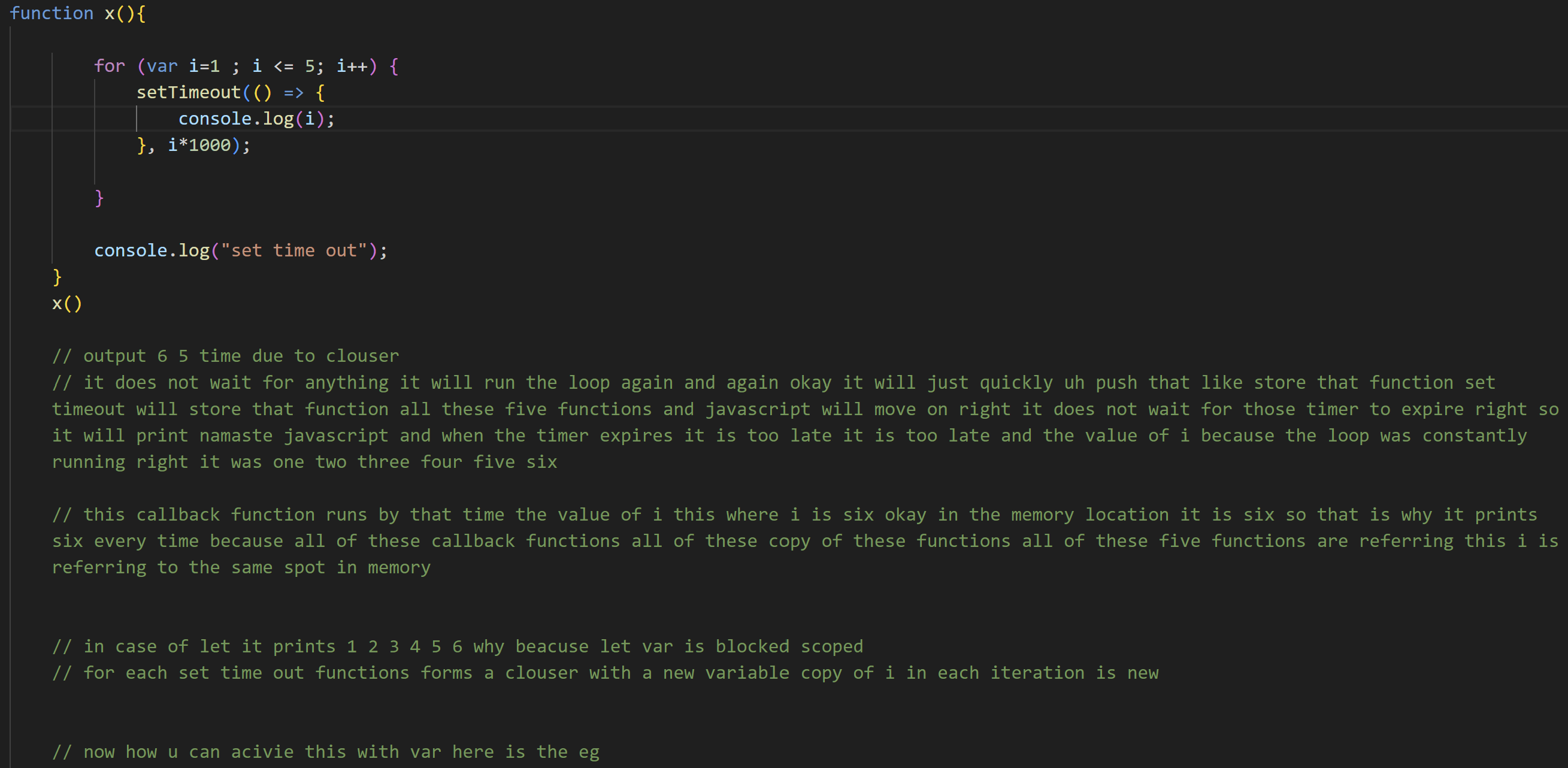
**-iterators and may more**

Set timeout + cloures

1. setTimeout stores the function in a different place and attached a timer to it, when the timer is finished it rejoins the call stack and executed.

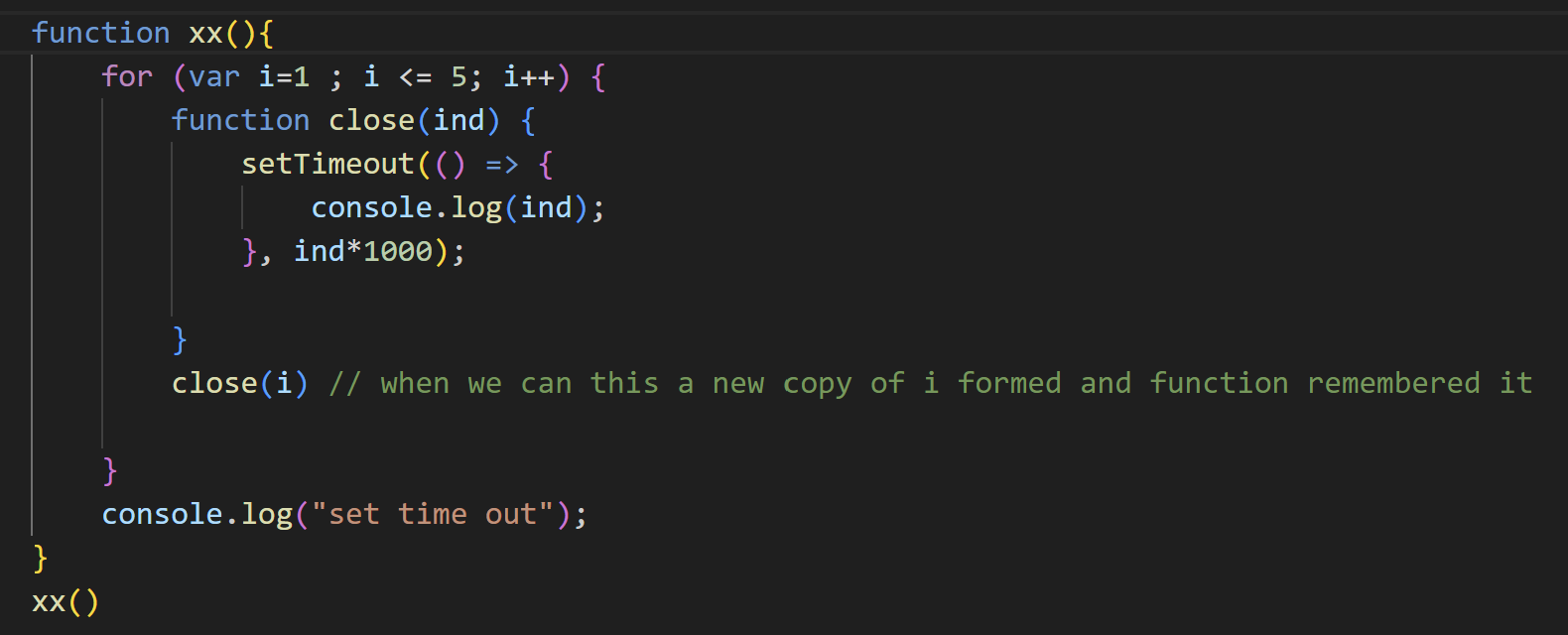


2. Without closure the var reference gives the latest value as it does not retain the original value but rather has the reference so any update in value after timeout will be shown.



3. If we use let/const because they have block scope, every time a new copy of variable is attached, thus this can be done without closure

// In below case, close is getting called for each value of i. And for each close() called a new execution context is getting created and in execution phase of that context new callback function context is created with lexical environment, which is reffering to i with which close() was called as close(i) was immediate parent of that callback function. Above cycle goes on and on until for loop ends.

In earlier case, only x() was there which was getting called only once and then callback function was called 6 times but all of them were having their parent as x() and hence all were pointing to same address(reference) of i.

Deep dive into Functions

Anonymous func- function without a name

Arguments are passed to function&

Parameters are received in/by function

**1. What is Function Statement ?**

A. A normal function that we create using Naming convention. & By this we can do the Hoisting.

For Ex - function xyz(){

console.log("Function Statement");

}

2. What is Function Expression ?

A. When we assign a function into a variable that is Function Expression. & We can not do Hoisting by this becz it acts like variable.

For Ex - var a = function(){

console.log("Function Expression");

}

**3. What is Anonymous Function ?**

A. A Function without the name is known as Anonymous Function. & It is used in a place where function are treated as value.

For Ex - function(){

}

4. What is Named Function Expression ?

A. A function with a name is known as Named Function Expression.

For Ex - var a = function xyx(){

console.log("Names Function Expression");

}

5. Difference b/w Parameters and Arguments ?

A. When we creating a function & put some variabels in this ( ) that is our Parameters.

For Ex - function ab( param1, param2 ){

console.log("

}

& When we call this function & pass a variabel in this ( ) that is our Arguments

For Ex - ab( 4, 5 );

6. What is First Class Function Or First class citizens?

A. The Ability of use function as value,

\* Can be passed as an Argument,

\* Can be executed inside a closured function &

\* Can be taken as return form.

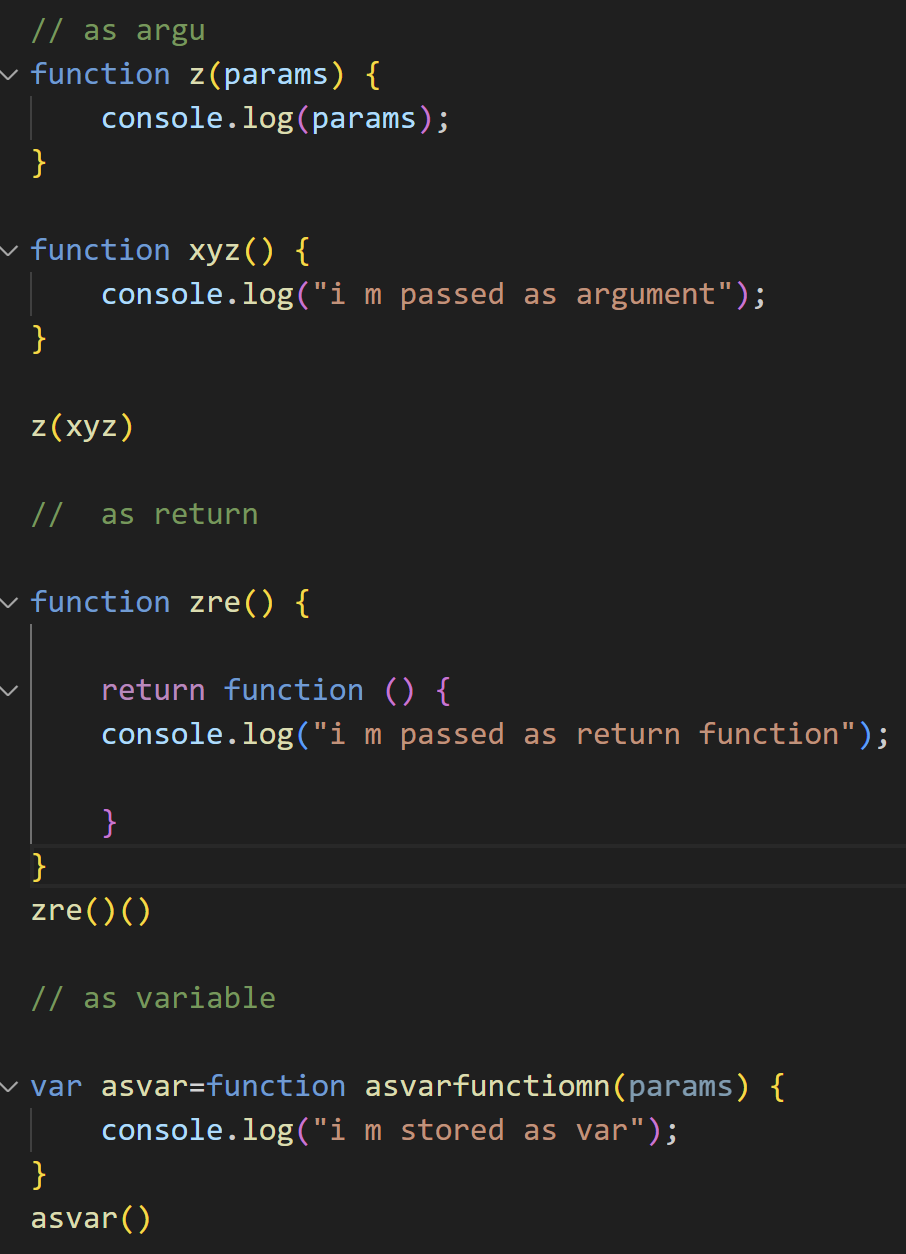
For Ex - var b = function(param){

return function xyz(){ / function return

console.log(" F C F ");

}

}



7. Function are heart of JS. They are called first class citizens or first class functions because they have the ability to be stored in the variables, passed as parameters and arguments. They can also be returned in the function.

--------------------------------------------------------------------------------------------

**Callback function**

**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.**

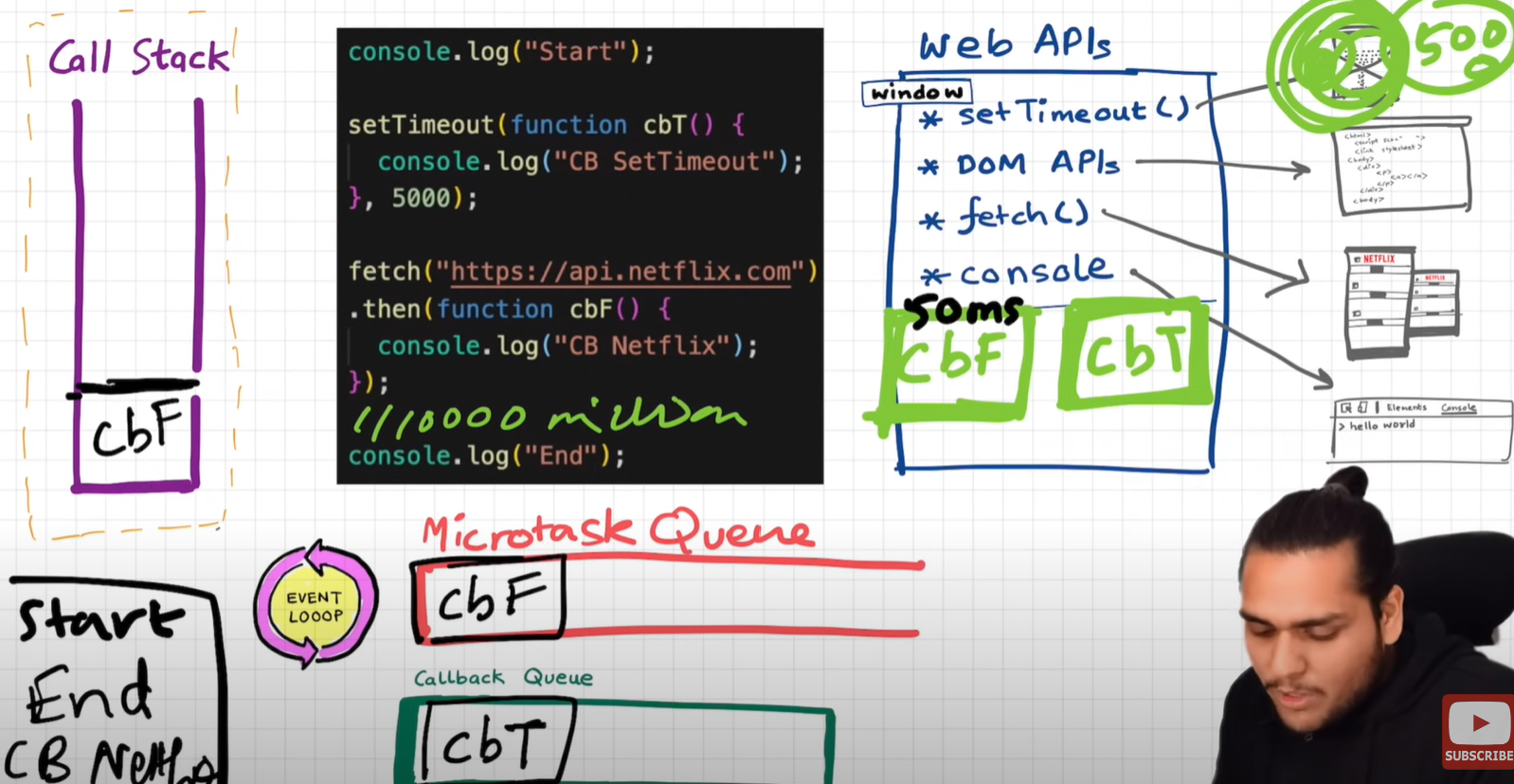
**3. Event listeners can also invoke closures with scope.**

**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.**

**Todo: watch clouser interview video and notes**

**-----------------------------------------------------------------------------**

**Event Loop**

****

**Start end cb Netflix cd settimeout**

**1. Browser has superpowers that are lent to JS engine to execute some tasks, these superpowers include web API's such as console, location, DOM API, setTimeout, fetch, local storage.**

**2. Callback functions and event handlers are first stored in Web API environment and then transferred to callback queue.**

**3. Promises and mutation observer are stored in API environment and then transferred to microtask queue.**

**4. Event loop continuously observes call stack and when it is empty it transfers task to call stack.**

**5. Micro task is given priority over callback tasks.**

**6. Too many micro tasks generated can cause Starvation (nit giving time to callback tasks to execute).**

The job of event loop is it continuous watch the call stack to check weather it is empty as soon as it founds empty call stack it popped the micro task call back from micro task queue and pushed to call stack for execution. once the micro task queue gets empty then it starts popped call back from the macro task queue and pushed the call back to the call stack for further execution.

------------------------------

The microtask queue holds tasks that are also ready to be executed but has a higher priority than the callback queue. Microtasks are usually scheduled by JavaScript promises, mutation observers, and other similar mechanisms.

Here's how they work together:

When an asynchronous operation is encountered, such as a setTimeout or a Promise, the callback associated with that operation is sent to the callback queue after the specified time or when the Promise settles.

When the call stack is empty (no functions being executed), the event loop takes the first task from the microtask queue and pushes it onto the call stack.

If the microtask queue is empty, the event loop looks at the callback queue and pushes the first task onto the call stack.

This process repeats, allowing JavaScript to handle asynchronous operations without blocking the main thread.(call stack which is single)

------------------------------

1. When does the event loop actually start? - Event loop, as the name suggests, is a single-thread, loop that is `almost infinite`. It's always running and doing its job. ️

2. Are only asynchronous web API callbacks are registered in the web API environment? - YES, the synchronous callback functions like what we pass inside map, filter, and reduce aren't registered in the Web API environment. It's just those async callback functions that go through all this.

3. Does the web API environment stores only the callback function and pushes the same callback to queue/microtask queue? - Yes, the callback functions are stored, and a reference is scheduled in the queues. Moreover, in the case of event listeners(for example click handlers), the original callbacks stay in the web API environment forever, that's why it's advised to explicitly remove the listeners when not in use so that the garbage collector does its job.

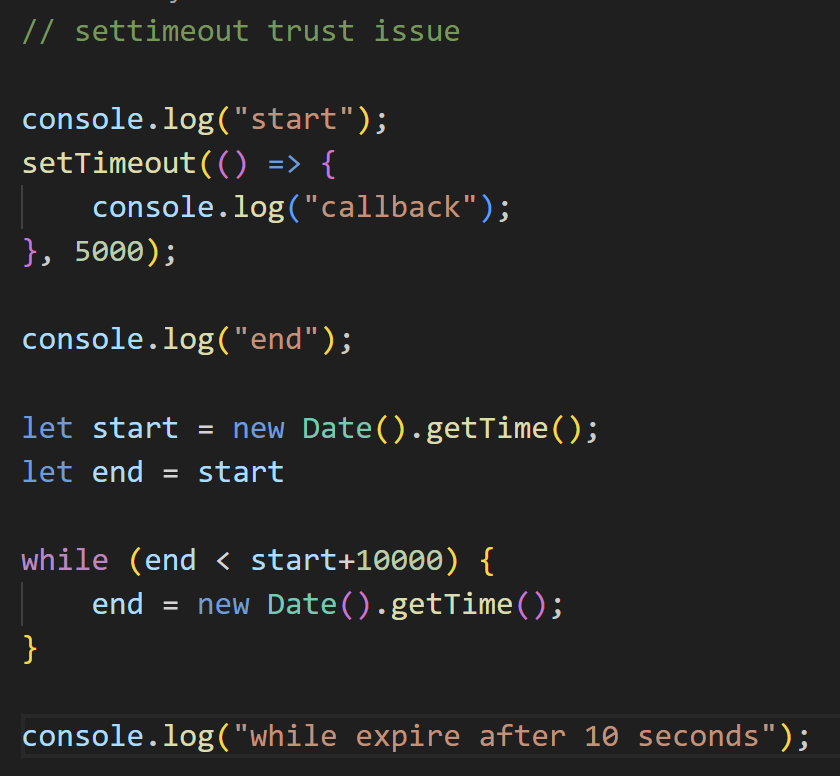
4. How does it matter if we delay for setTimeout would be 0ms. Then callback will move to queue without any wait?

No, there are trust issues with setTimeout() . The callback function needs to wait until the Call Stack is empty. So the 0 ms callback might have to wait for 100ms also if the stack is busy.

**Trust Isuues with set time out**

**1. The setTimeout function stores it in the callback queue which is executed only after call stack is empty, even if setTimeout is set to 0ms.**

**2. setTimeout ensures that minimum it will take the time mentioned because it may be paused due to call stack not empty.**



Doubt : why does js has only one call stack

Js syncronus single threaded language that the beauty because of one call stack all code run there so because of this js is kind of intpreated language which makes its fast even inside the browser not wait for compile like other

**JS Engine google v8 engine**

**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(just in time) compilation, it interprets while it optimises code as much as it can on run time.**

**7. Execution and Compilation are done together.**

**8. Execution has Garbage collector and other optimisation such as inlining, copy elusion, inline caching etc.**

**Basic idea about Mark & Sweep Algo:**

**It comprise of 2 phases -**

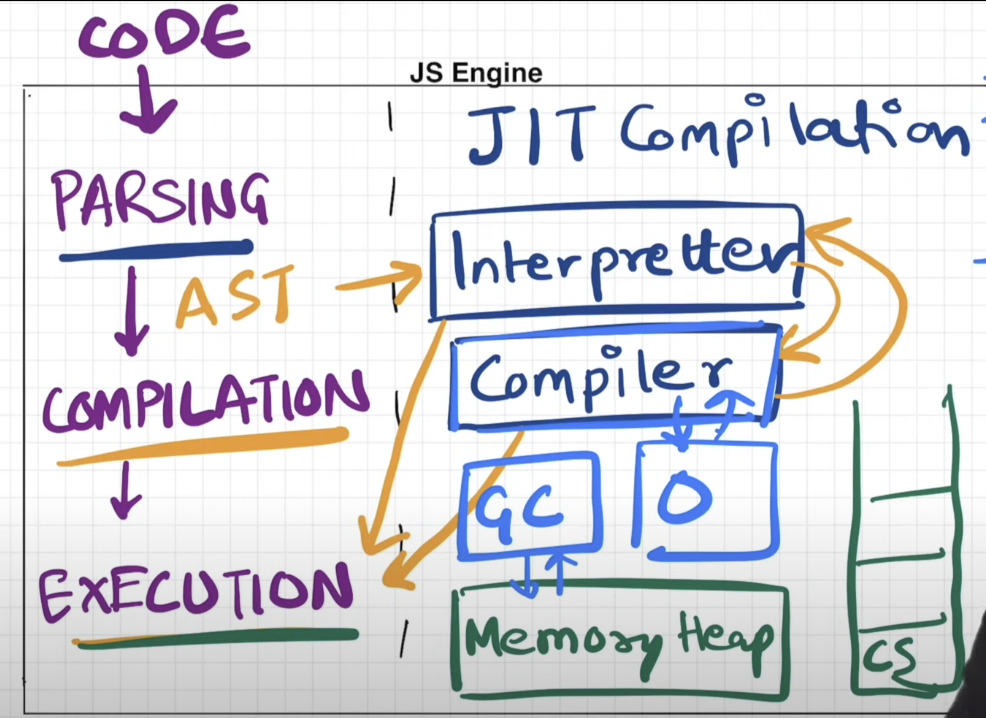
**1. Mark phase**

**2. Sweep phase**

**All objects are marked as 0 initially (at creation) and in mark phase the objects that will be accessible are marked as 1 (reachable) by a DFS graph traversal.**

**During sweep phase, the objects marked with 0 are removed from heap memory. and also all reachable objects are again initialized with 0 (made unreachable) because the algorithm will run again.**

**So, it's basically tracing garbage collector concept. :)**

****

**Code is broken down into tokens**

**Syntax parser generate ast (tree like structure)**

**Then ast pass to compilation phase**

**Compiliation and execution hand in hand**

**Jit -byte code -> execution phase -call stack**

**Memory heap is a plce where all variables , functions assigned memory**

**Gc-garbage collector**

**Js is behave as inteprated lang and compile lang depends on js engine**

**Nows day new browsers uses both called just in time compilation**

**--------------------------------------------------------------------------------**

**Higher oder functions**

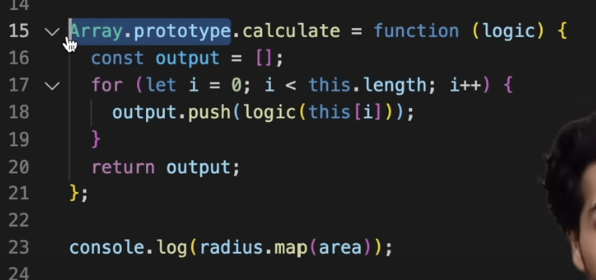
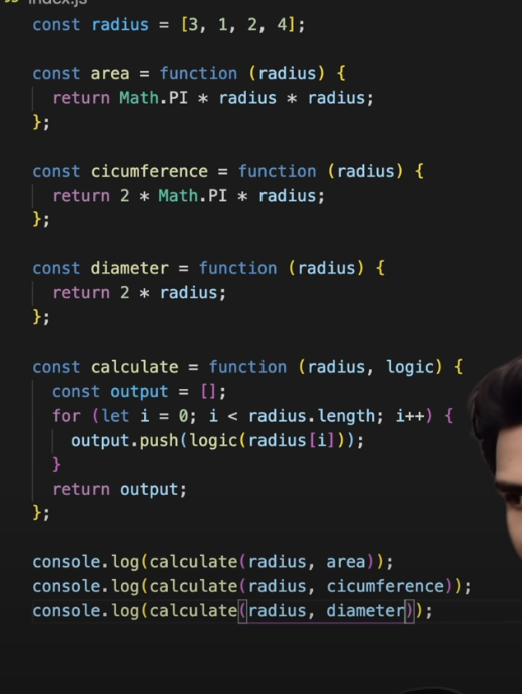
**1. Follow DRY(Don't Repeat Yourself) principle while coding.**

**2. Use function to stop writing repeating line of codes.**

**3. Function that takes another function as argument(callback function) is known as Higher order functions.**

**4. It is this ability that function can be stored, passed and returned, they are called first class citizens.**

**5. If we use Array.property.function-name. This function is accessible to any array in your code.**

****

**For someone who got confused:**

**First-class functions are JavaScript functions that can behave like variables. They can also be passed as arguments to higher-order functions.**

**Higher-order functions are functions that return a function or take in a function as an argument.**

**Map ,filter and reduce**

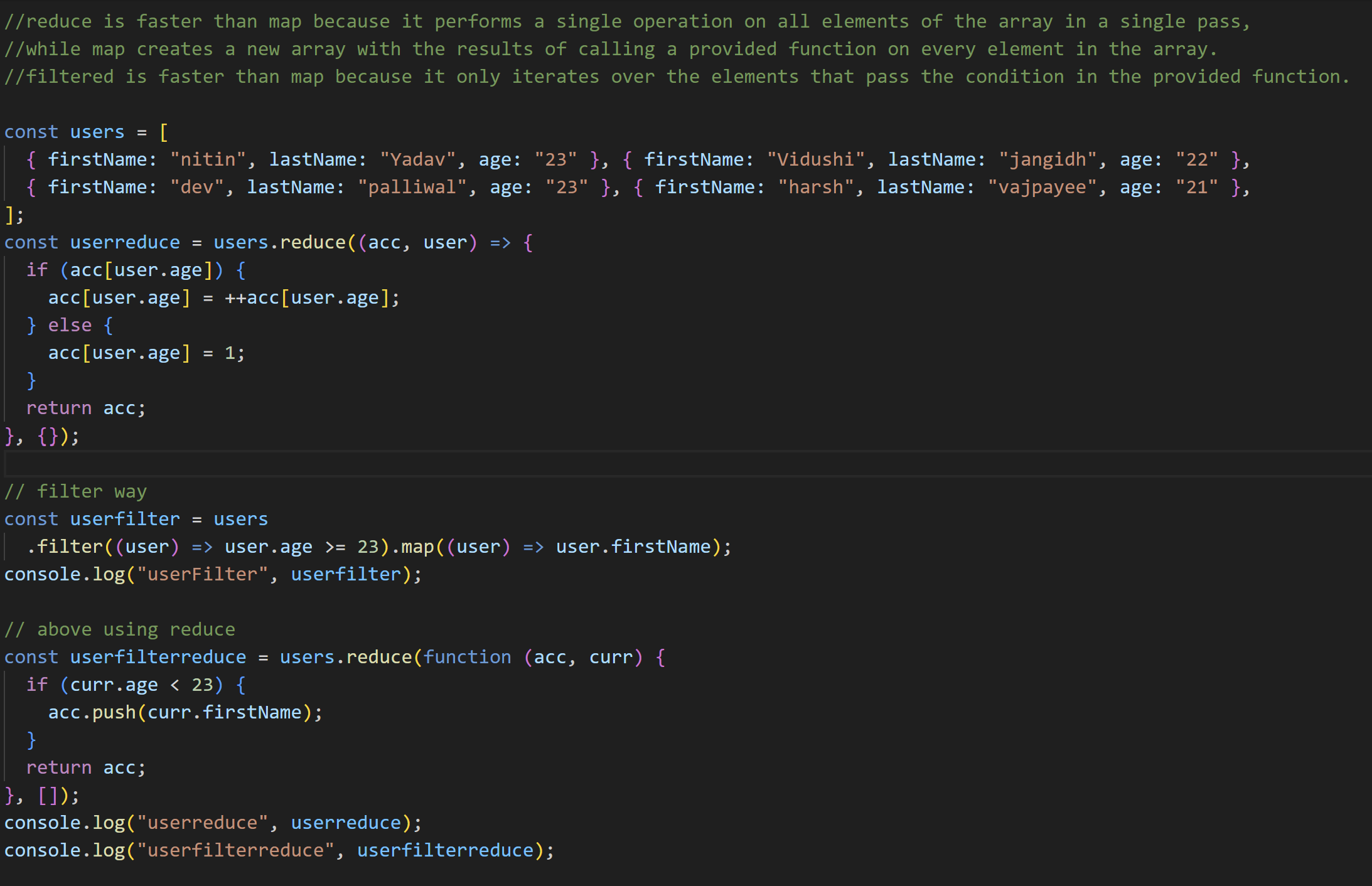
**1. map method is used when we want transformation of whole array.**

**2. filter is used when we want to filter the array to obtain required value.**

**3. reduce is used when we want to reduce the array to single value eg (max, min, avg, sum, difference etc).**

**4. reduce passes two arguments one function(which includes accumulator and initial value as argument itself) and another initial value of accumulator.**

**reduce(),filter,map is a non-mutating method, meaning that it does not change the original array. Instead, it returns a new value.**

****

**CALLBACK HELL**

**Pros:**

**JS is a single-thread language and is synchronous by nature, but with callbacks, we can make async calls.**

**Cons:**

**1) Callback Hell: Callbacks are useful when we want to perfrom some extra functionality with our already existing function for example passing an error handling callback to our already created function, BUT when callbacks within themselves start taking in other functions as callbacks then that mess that you are left with is known as the Callback Hell leading to unreadable code, hence unmaintanable code.**

**2) Inversion Of Control: when we pass a function to other function as a callback we are giving the called function the control of whether to even call it or not or maybe call it in a wrong context. For example a success callback is called when an error occours inside a called function (maybe due to human error while writing the code for called function), this type of giving up of control over our functions is known as inversion of control.**

**While using callback we face 2 issues which are as follows:**

**1. Callback Hell : When more than 1 APIs depend on each other to get call so then we pass the callback function inside callback function so it created nested callback function this makes our code less maintainable and readable this is callback hell. It also know as "Pyramid of Doom".**

**2. Inversion of Control: When we pass a callback function into another function then the execution of callback function is depend on that function so in this way we loose the control over our code this is know as Inversion of Control.**

**Promises**

**1. Before promise we used to depend on callback functions which would result in 1.) Callback Hell (Pyramid of doom) | 2.) Inversion of control**

**2. Inversion of control is overcome by using promise.**

**2.1) A promise is an object that represents eventual completion/failure of an asynchronous operation.**

**2.2) A promise has 3 states: pending | fulfilled | rejected.**

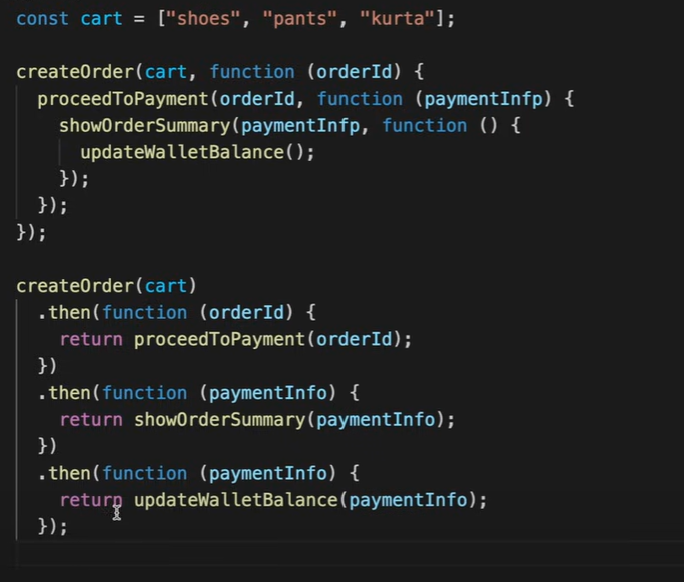
**2.3) As soon as promise is fulfilled/rejected => It updates the empty object which is assigned undefined in pending state.**

**2.4) A promise resolves only once and it is immutable.**

**2.5) Using .then() we can control when we call the cb(callback) function.**

**3. To avoid callback hell (Pyramid of doom) => We use promise chaining. This way our code expands vertically instead of horizontally. Chaining is done using '.then()'**

**4. 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()**

**//instead of passing function we attached the handlers or call back functions**

**How to create promise**

1. Promise can be created using a new Promise() constructor function.

const pr = new Promise(function(resolve, reject) {})

2. This constructor function takes a callback function as argument.

3. The callback function has 2 arguments named 'resolve' and 'reject'. Resolve and reject are the keywords provided by JS.

4. We can only resolve or reject a promise. Nothing else can be done.

5. An error can also be created using new Error('error message').

const err = new Error('Cart is not valid!');

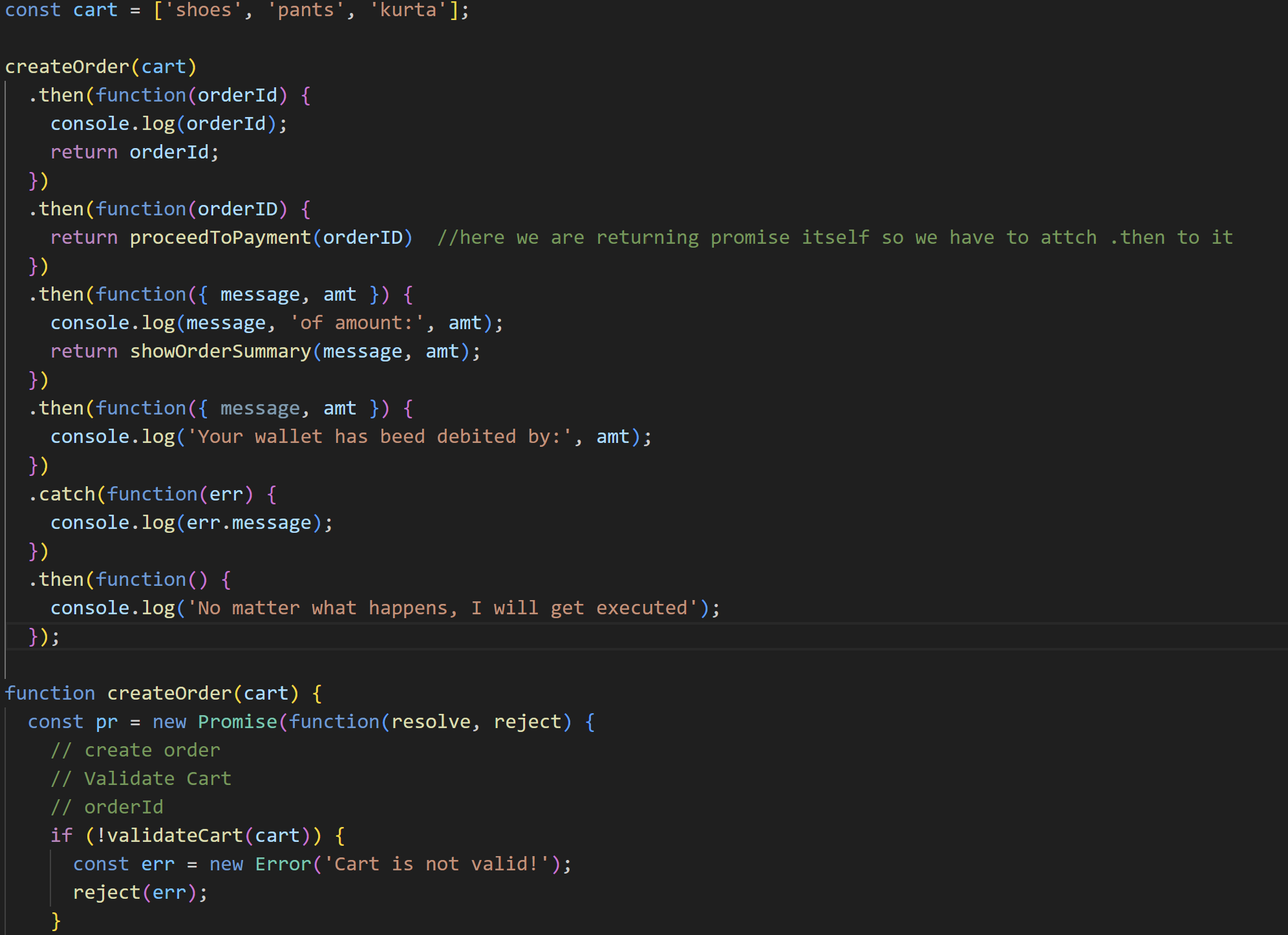
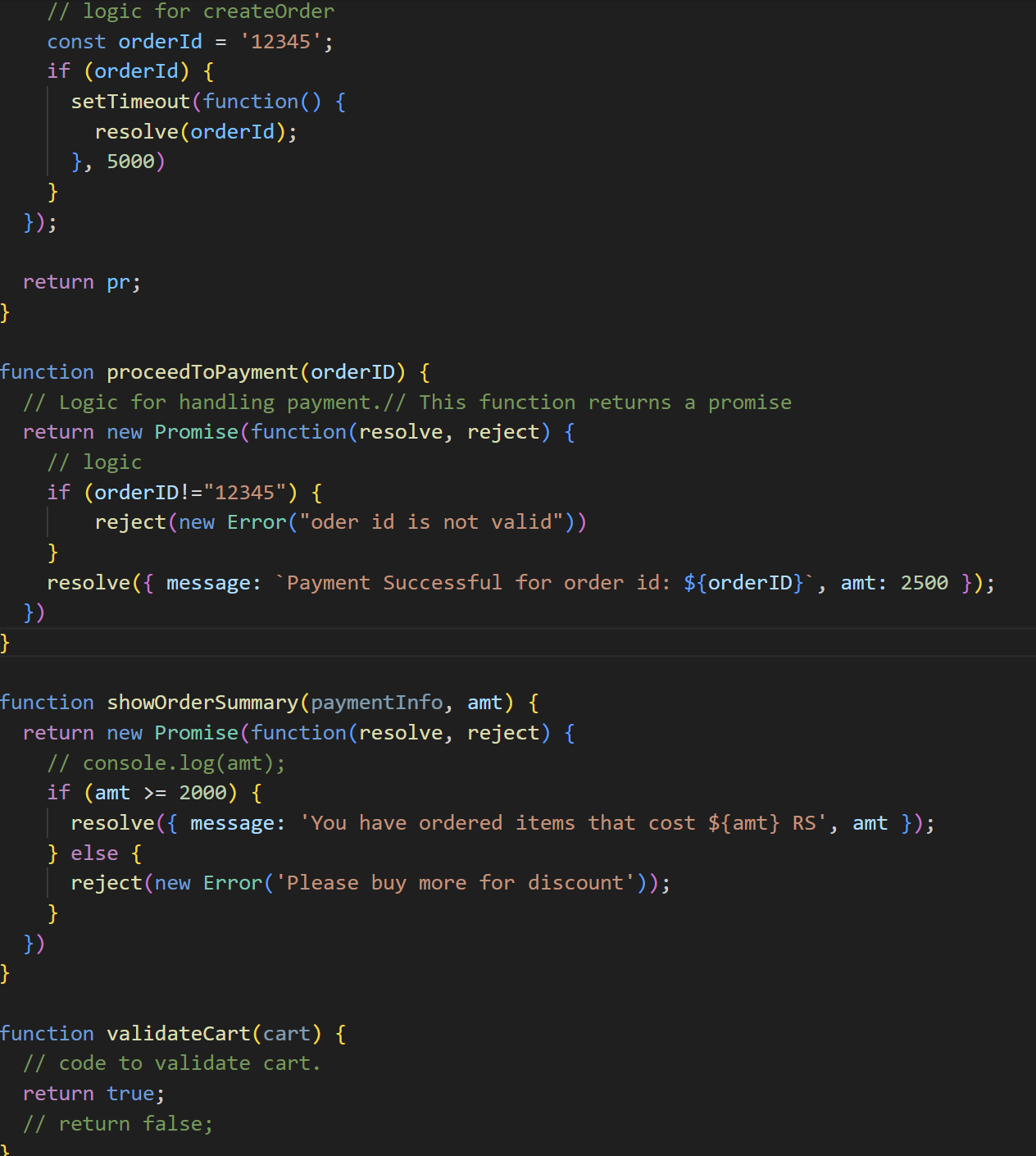
6. There is also .catch() which is used to attach a failure callback function that handles any error that pops up during the execution of promise chain.

7. .catch only handles error of .then() that are present above it. If there is any .then() below it, catch will not handle any error for that, also that ,then will get executed no matter what.

8. It can be useful in a way if we want to catch error for a particular portion of a chain. We can have multiple catch based on requirement and then a general catch at the end.

9. Always remember to return a value in the promise chain for the next .then to use

10. If it returns a value => It will be used as an argument in next function. If it is a promise then the next .then in the promise chain is attached to the promise returned by the current callback function.

f

Promise APIs || all, allSettled, race, any

promise:

[p1,p2,p3] all three take array of promises

Promise.all – fail fast

* Return a array with value of all the results
* Make api call paraller but wait all api to finish before returning result

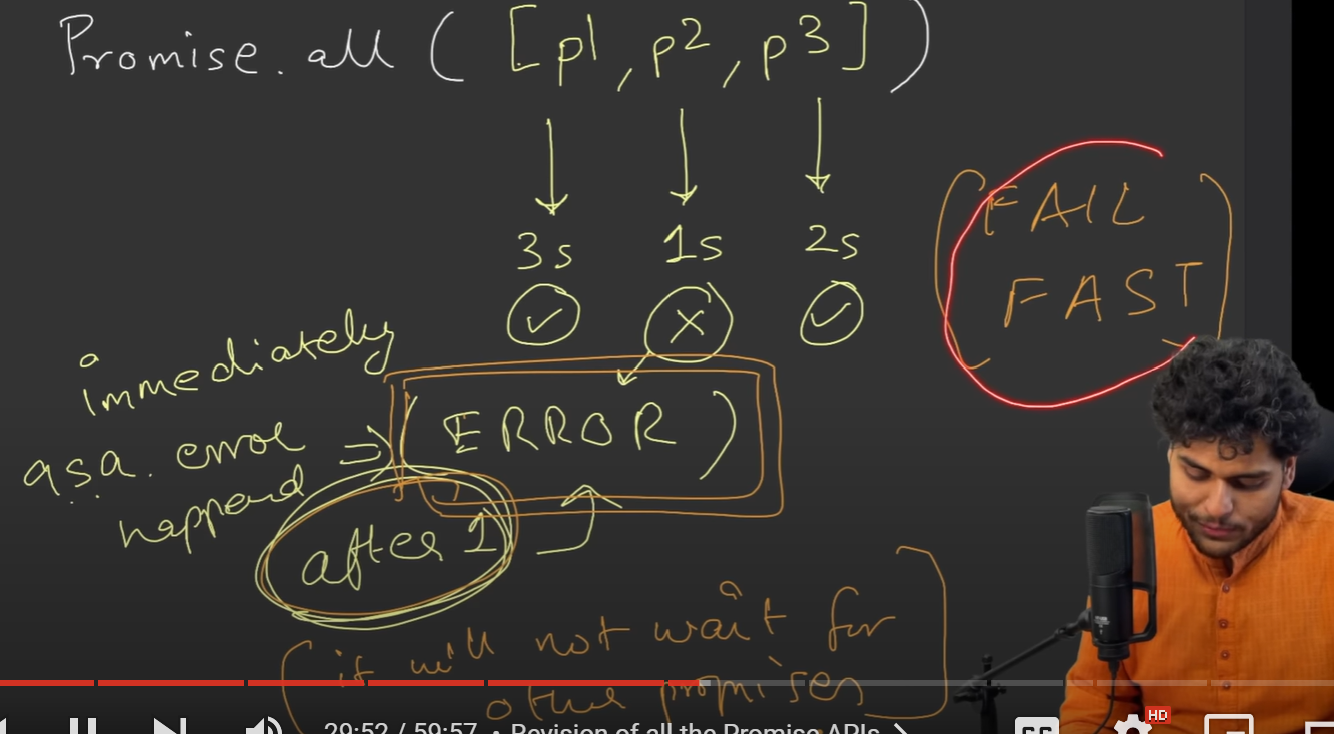
What if any one rejected

-as soon one of the promise failed rejected promise.all throughs an error

It will not wait for other promise

-immediately as soon as error happened it throw an error (OR or none)

-what happen to other settled success promise -whne a promise is created u cant cancle a promises in between so these promise is eventually accepts or rejects depends on their fate but promise.all rejects ans nay of thr promise rejected

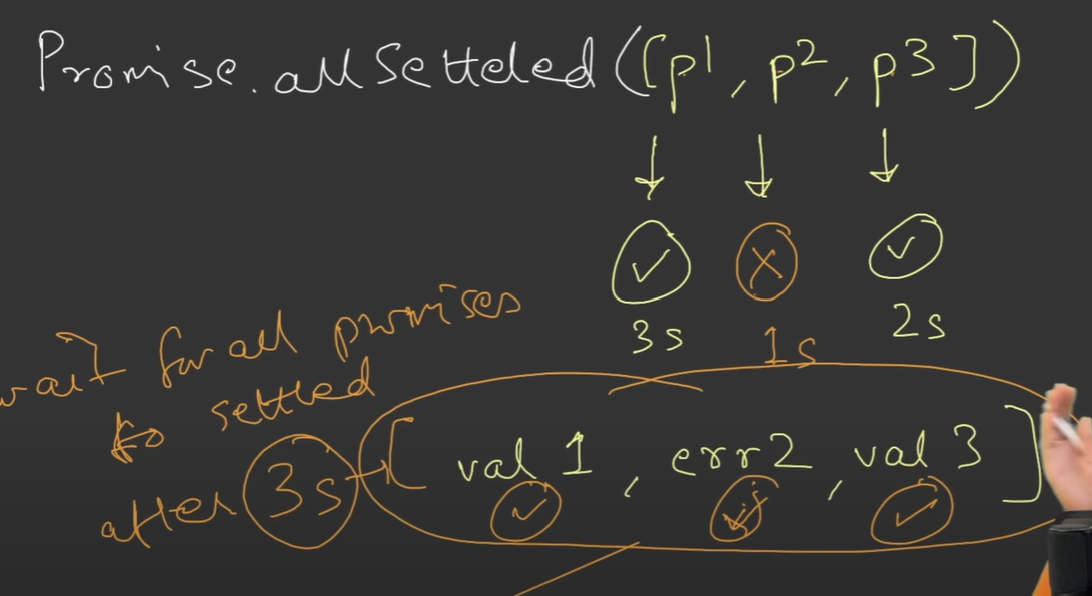


Promise.allsettled - it waits

-all sucess- same result as all

If one of the fails

* It will wait for other promises to settle before throwing error (array return)



Promise.race – as soon as first settled promise

Val of first settled promise

-it will return of first settle promise weather it is success or failure



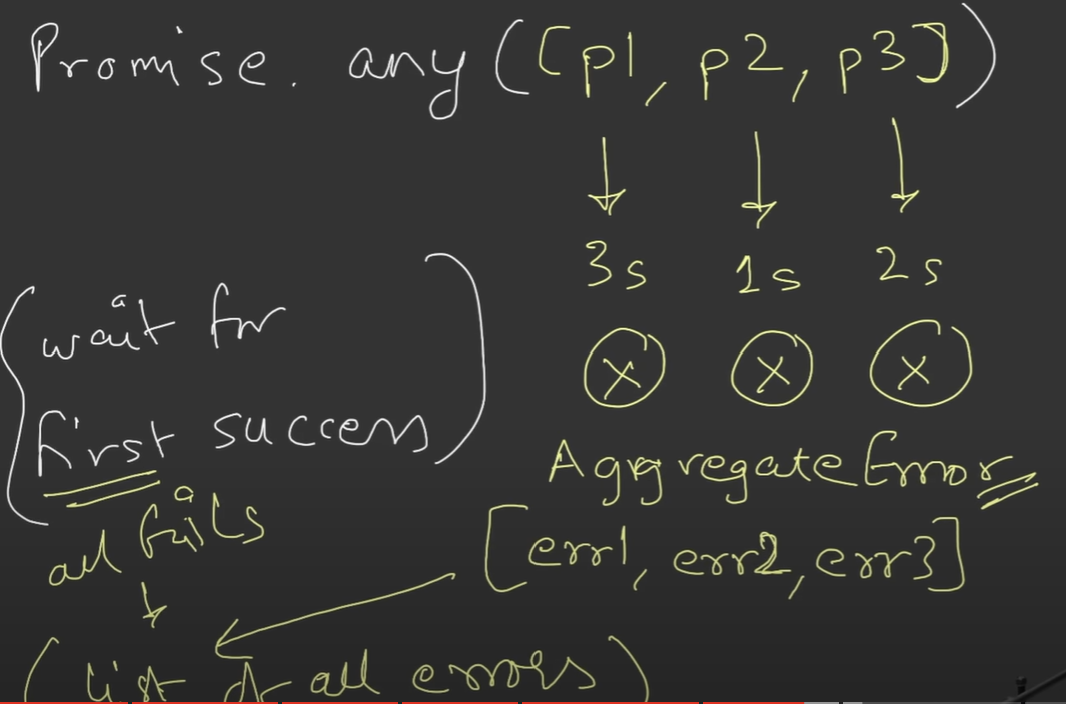
Promise.any- wait for first sucess promise

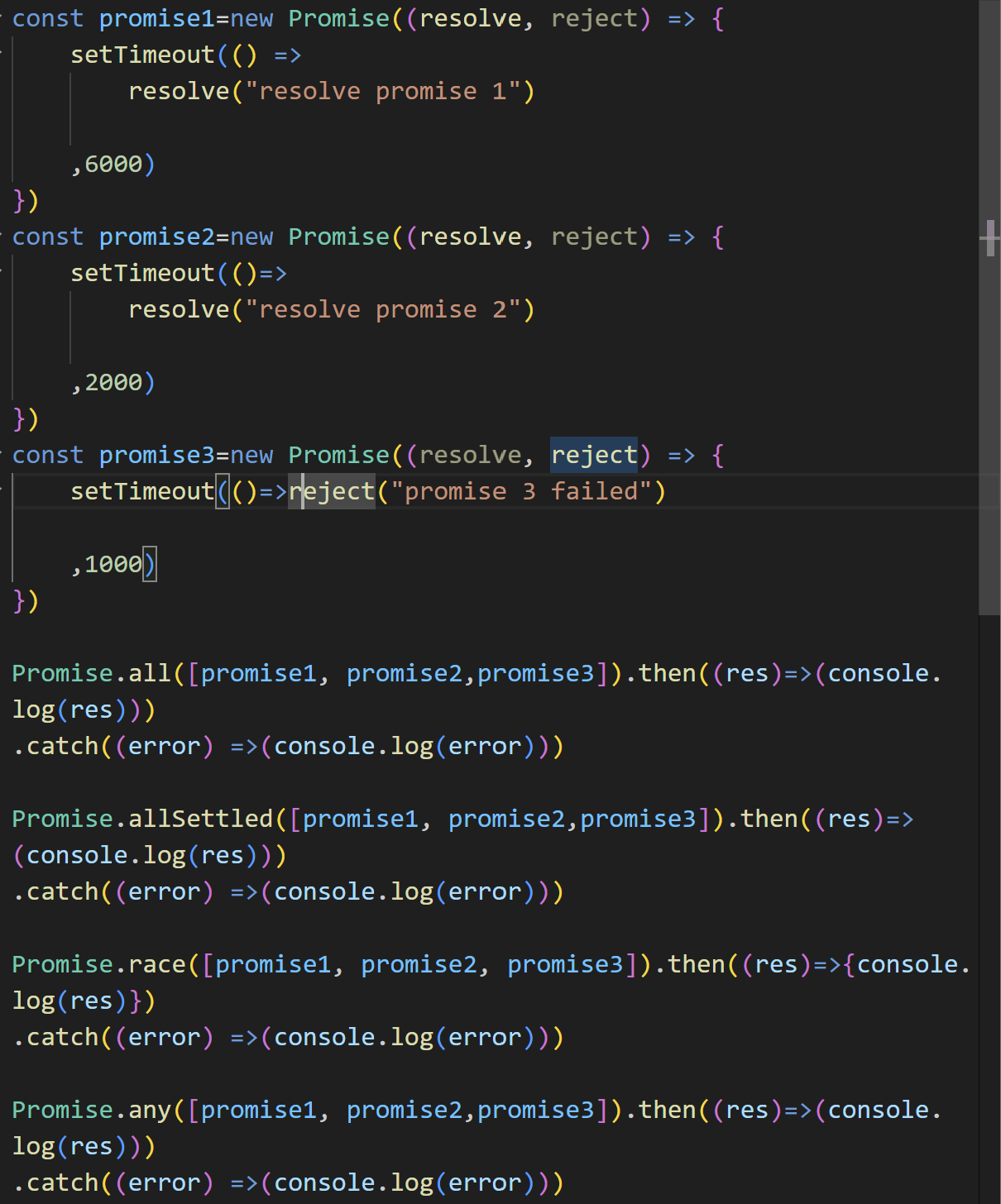
Seekinging the first sucess

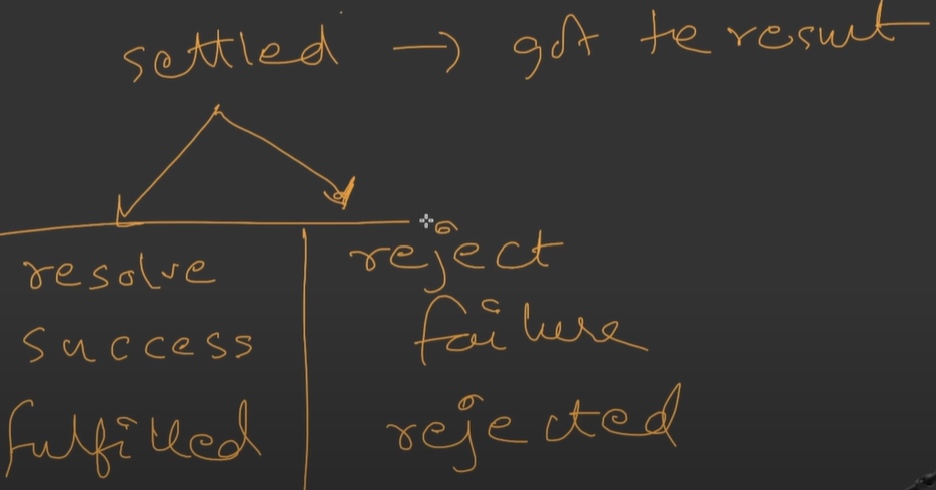
If it finds rejected promise it igrine and go to other promises

What all fails:

Return result will we aggregate error -array of all the 3 errors







**Asyn await**

**What is async**

* **Async is a keyword i.e used before a function to create a async function**

**How this is different from a normal function--- this(asyn) is always retun a promise(if value return then asyc wrap this in promise first then return)­**

**Using await with async**

* **Async and await combo used to handle promise**

**How we handle promises before**

async function asyncfun(promise) {

  return promise;

}

const data = asyncfun("promise full filled");

data.then((data) => console.log(data));

const promise\_creation = new Promise((resolve, reject) => {

    setTimeout(() => {

        resolve("promise fullfilled succesfullly");

    }, 3000);

});

//  how we handle promise before

function getpromisestate() {

  promise\_creation.then((res) => console.log(res));

}

// getpromisestate();

// new way using await async

// await can only used inside an async function

// and await write in front of promise in an async function

async function getpromisestate\_await() {

  try {

    const res = await promise\_creation;

    console.log(res);

  } catch (error) {

    console.log(error);

  }

}

// try and catch use to handel errors

// why this async await is special

// --handling promise in older and new ways --

function getpromisestate2() {

    promise\_creation.then((res) => console.log(res));

    console.log("i print first then promise full filled log");

  }

  // in this log above is print first because js engine is not wait to promise to full filled

  // so it print first then promise fullfilled log

//   getpromisestate2();

// now what happen in async function

async function getpromisestate\_await2() {

    console.log("i print first then promise fullfilled log");

    // js engine wait to promise to fullfilled  first

    const res = await promise\_creation;

    console.log(res);

    console.log("i m not print first then promise fullfilled log because due to await it wait till the promise to full filled");

  }

// getpromisestate\_await2()

// when await write more then once what happenng in async function

program wait till the promise to fullfilled

const pc1 = new Promise((resolve, reject) => {

    setTimeout(() => {

        resolve("promise  1 fullfilled succesfullly");

    }, 5000);

});

const pc2 = new Promise((resolve, reject) => {

    setTimeout(() => {

        resolve("promise 2 fullfilled succesfullly");

    }, 3000);

});

async function promisec2() {

    // js engine wait to promise to fullfilled  first

    const res1 = await pc1;

    console.log(res1 +"p1");console.log("promise 1");

   const res2 = await pc2;

   console.log(res2 +"p2");console.log("promise 2");

  }

  promisec2()

  //for p1 -3000 sec  and p2 -5000 sec in this p1 print after 3 sec and after 5 s it print p2

  // but in p1-5000 p2 -1000 in this first it wait to full filled the p1 and after 5000 its print both p1 and p2

  // beacuse js is single threaded language

  // when js engine find await  its suspend the async function execution for that time (so main thread in not block) and wait till the p1 to be resolved and once this resolved then only it move a ahead and after 5 sec it come back to the call stack at that point it suspend

  // in oppsite eg in this by the time p1 resolve p2 already resolve but function  is suspend till the promise of p1 to resolved after p1 resolve it print and then it comes to promise 2 it see p2 already resolve then it print all the pending

  // impt to notice js engine is not waiting but the function suspended and comes again in call stack after fullfill the promise

/// real world eg of async await

// how fetch works with

// fetch gives promise fetch return the response from the server

//fetch()=>response object(response.json) this again a promise => json value

async function fetchfun() {

    try {

       await fetch("http://api.gitthub.com/users/nickop21")

        // const response = await data.json();

       .then((response) => response.json())

        .then((res) => {

          console.log("response:", res);

        });

    } catch (error) {

        console.log("somthing wents wrong",error);

    }

}

// way to handle error in async await

// old method

fetchfun().catch((error) => {console.log("old way to handle error",error);});

// async await is just a syntactic sugar over promises behind the seen js uses normal promises

// in async await we don’t have to deal with callbacks we don’t have to deal with promise channing

Ques). for those who thinks if code reaches await of p1 suspends function execution and after 5 sec it should go to await p2 the timer for p2 should start and should take 10 sec there so overall function execution should be 15sec?

Ans). the timer ticking doesn't start at await, instead it started in the beginning of the code only where promise was declared.

If the declaration would have been like inside async p1 declare then p1 await after that p2 declared follwed by p2 await then function would have taken 15 secs