
java
Script

# Execution context

» everything happens inside it -

variable environment  Thread of execution.

| memory | code |
| --- | --- |
| key value functions | o —<br>o — |

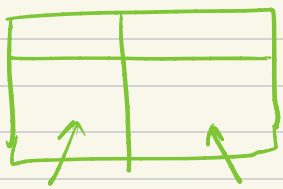
code executed one line at a time

## Javascript is a synchronous, single threadead language

- one comand at a time
- in a specific order

# Execution Context ....

## how javascript code runs?



1st scan memory is allocated

2nd scan, code is executed.

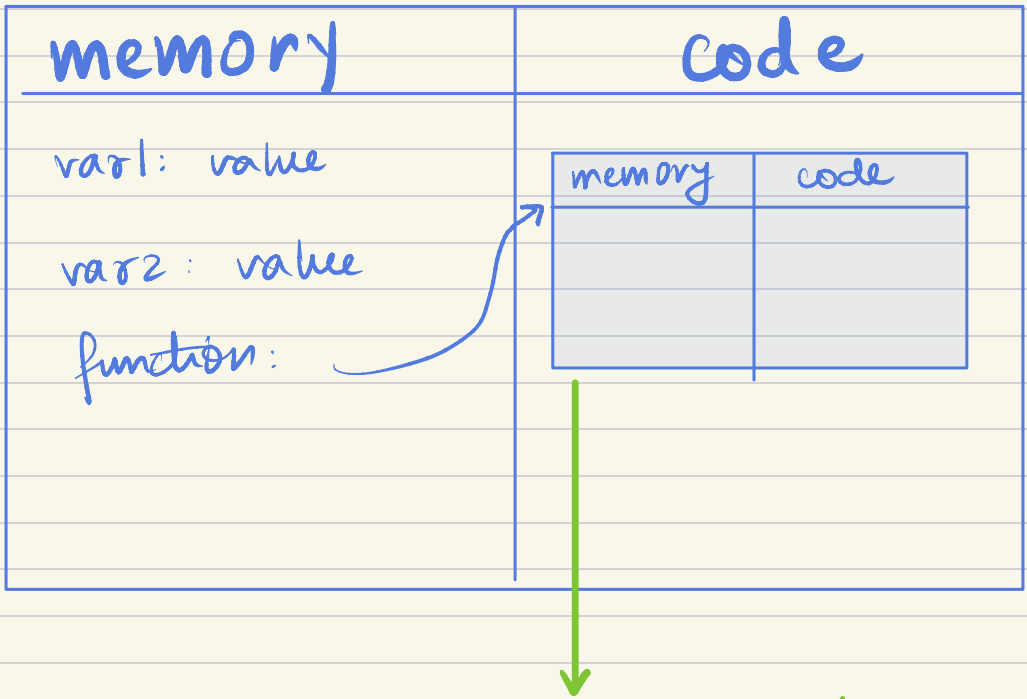
code

2 scans of the code are done.

### 1st scan

| memory | code |
| --- | --- |
| var1: undefined | |
| var2: undefined | |
| fun: <whole code copied> | |
| ⋮ | |

# 2nd scan global exec context



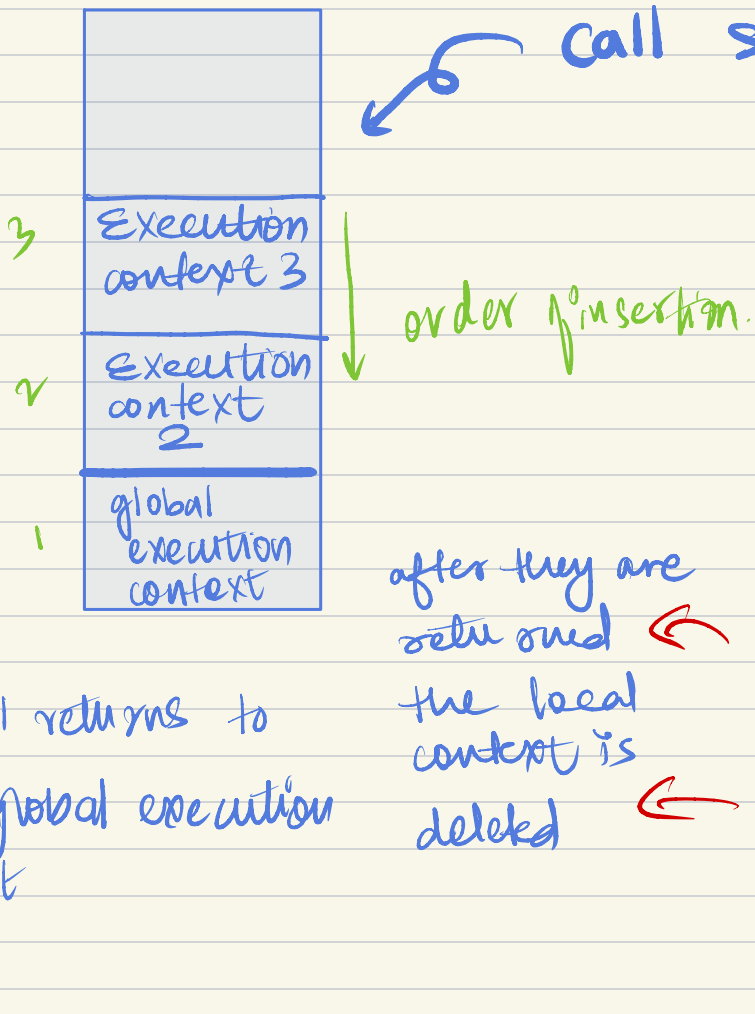

* another execution context is created when a function is called.
* the variables stored in memory
  - parameters
  - arguments
* when the function returns, the execution context of the function is deleted.

(later about closures)

# CALL STACK.

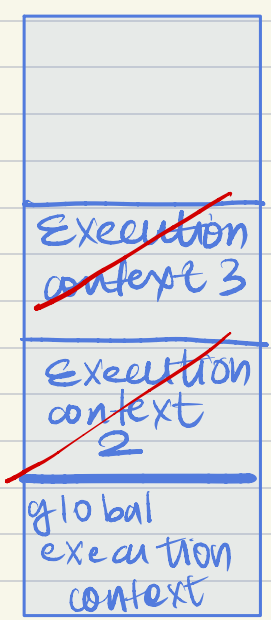
* This creation and deletion of execution environments happens throughout the running of the code.

* to handle this, a call stack is maintained




control returns to the global execution context

after they are returned the local context is deleted



Call stack maintains the **ORDER OF EXECUTION** of execution contexts

# hoisting in js.

you can access variables anywhere in the program, even before you have initialised it:

MDN web docs definition

The process whereby the interpreter appears to move the declaration of func", variable, classes or imports to the top of their scope, prior to execution of the code.

my thoughts: all this is because of the 2 scan process of js, all variables and functions are allocated memory, even before the actually execution takes place.

# Undefined VS not defined.

* before execution of code, all variables have been allocated memory.
  - value allocated to variables
  - ↳ all variables are equal to **UNDEFINED** (place holder keyword)
  - ↳ takes up memory

* if you try to access a variable which is not present in memory → **NOT DEFINED.**
  - ↳ no memory allocated.

# java script is loosly typed

a single variable can hold any type.

dont do, ~~a = undefined~~ not a good programming practice.

can lead to inconsistencies.

# Scopes

→ in order / in hierarchy.

- lexical environment is local memory along with lexical environment

- In execution context, in memory space, you get reference to the lexical environment of the parent.

- all memory has reference to its parent.

- first search your own local memory, then search in the lexical environment of parent

Scope chain

let and const declarations are hoisted.

→ they are in the temporal dead zone.

let a = 10; → cannot access before initialized

var b = 100;

let and const are allocated memory but it is not global memory, its seperate memory

→ temporal dead zone is the time between when the variable is declared and initialised.

let a;


let and const both will be assigned memory in the temp dead zone. → undefined

# Temporal dead zone

time between when variable is hoisted and when the initialized.

also let variables are not attached to window object, as the memory space is different

also cannot redefine a variable which has been defined already with let.

```
let a = 10;

let a = 100;   ✗  SYNTAX error
```

always have all declarations and initializations at the top of the code, to avoid errors.

hoisted

Q. are let and const hoisted

Ans. Explain the concept of temporal dead zone

cannot access let and const before initialization, reference error.

# Block

(compound statement)

- used to group multiple js statements

```
{
    ===   this is a block
}
```

we can group multiple statements in a block where js expects only 1 statement.

```
if (    ) {
    ===
    ===
}
```

# Block Scope : not all vars and functions can be accessed inside the block

let and const are in the block scope. hoisted and undefined in the seperate memory space

Block scope goes away from global scope once its done executing

# Shadowing

a same named variable shadows global variable inside the block.

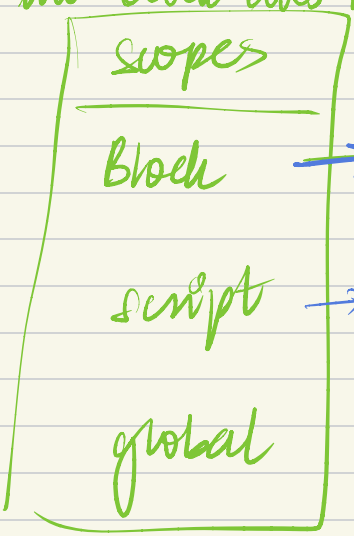
refers to same memory space.

```
var a = 100;        // prints 100
{
  var a = 10;       // prints 10
}
console.log(a)      // prints 10
```

modified original a. in the global memory

but in case of let and const.

it does shadow within a block scope, but outside the block does not modify global scope.



# let shadowing example

```
let a = 5
{
  let a = 10;
  console.log(a); // 10
}
console.log(a) // 5
```

# Summary of shadowing

| var | let / const |
|---|---|
| • shadows within block scope. | • shadows within block scope |
| • after block over, global value modified with block's value. | • after block over, global value is NOT modified |
| • because memory space is same : global | • because memory space is different not in global memory scope. |

# illegal shadowing.

let cannot be shadowed with var

```
let a = 20;

{
    var a = 200; // illegal, error
}
```

let a = 20;

{
    let a = 200; // perfectly fine
}

* var can be shadowed by let

# CLOSURES.

↳ meaning: a function bound together with its lexical environment.

- function has access to parent's lexical scope.

↓ you can return a function from a function.

If you save this value in a variable, the function will exist even outside its lexical scope.

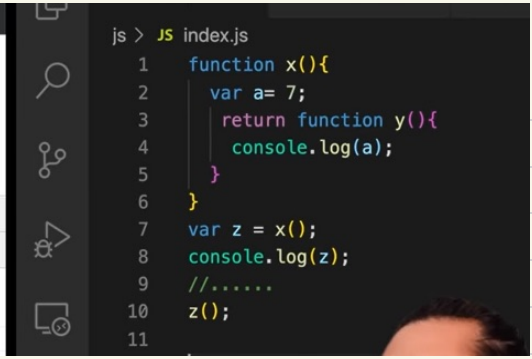
when functions are returned, and its execution context is deleted, it still maintains its lexical scope.

closure is returned

## Q: CLOSURE?

function along with its LEXICAL SCOPE bundled together forms a closure

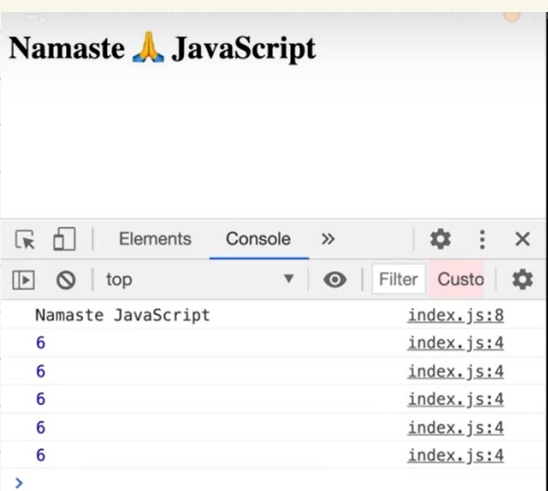
* function remembers something

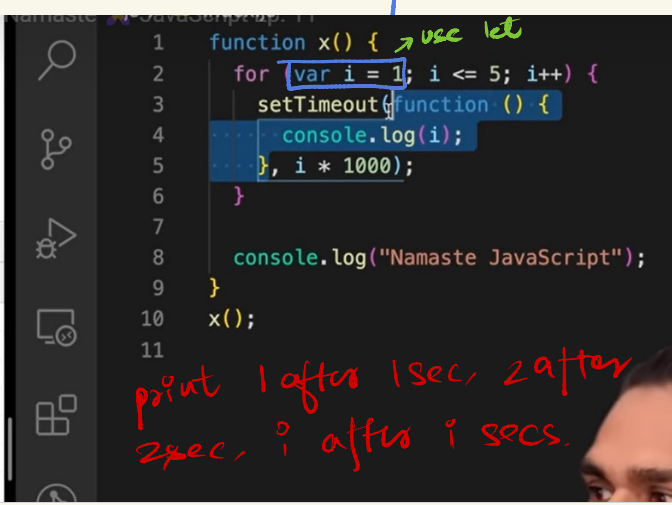




# Set time out

when set time is reached, it will NOT wait even and there.

Time, tide and Java script waits for none.




the i is referring to the same spot in memory.

the loop does not stop, i keeps incrementing.

but will not wait for setTimeout -

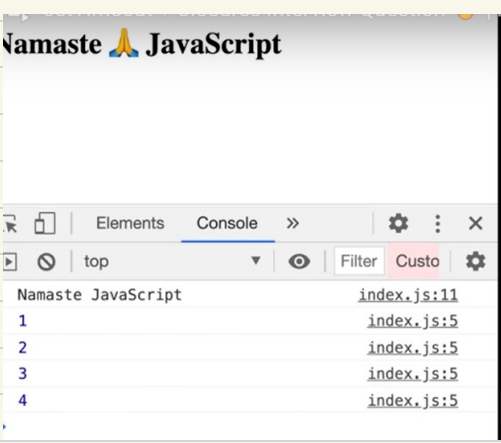
Instead of var, use let, → let has a block scope
creates a new copy every time

every time the function is called, copy of i is new

if don't have to write let, and use only var.

* key = give a new copy of i, to closure every time

Closure is the answer




basically, when set time out reached, its saved in some other memory not global, and the normal execution of the code keeps running, then later on, set time out code runs.

# Closure is the most beautiful part of javascript

wuhuu :)

# Mock javascript interview ....

akshay saini 0p.

Q. what is closure

- a function with its lexical scope bundled together.

- each function has access to its outer lexical environment.

- a function remembers its outer lexical environment, even if it was executed someplace else.

eg:

```
function outer () {
    var a = 10;
    function inner () {
        console.log(a);
    }
    return inner;
}
outer()();
```

if there is a parameter here, it is also in the outer scope and still can be accessed.

this calls the inner function.

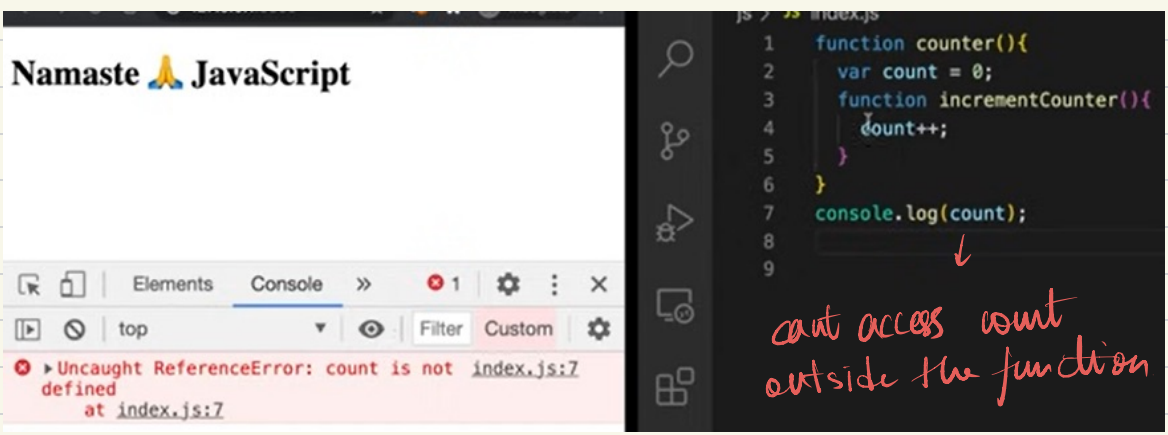
even if the outer function is nested within another function, access to the outer most lexical environment still exists.

# Advantages of closures

1. function currying
2. module pattern
3. used in higher order functions
4. data hiding called incasulation

## data hiding and encapsulation

other functions cannot access a variable




If counter is called again, it creates an independant copy, and does not touch, any previously called ones.

So count is a local variable and it is safe from outer environment

# Disadvantages of Closure

1. over consumption of memory
   - garbage collection does not happen for closure variables until expiration
   - accumulation of memory

garbage collector → function that frees memory for unreachable variables

& Relation between garbage collector and closure.

when a closure is formed, the access of variables local within a function still remains, even after function expires.

some modern browsers finds out of variables are unreachable, smartly collects the unused etc variables.

example some variable which is not being utilised.

# First Class functions.

it is a programming concept!

function statement / function statement

normal function

eg:
```
function a() {
    console.log("a called")
}

a(); // call a now
```

note: this can be hoisted as usual. i.e, you can call a, before function is defined

## function expression

```
var b = function(){
    console.log("b called");
};
b(); // call b;
```

note: b cannot be hoisted usually as b is treated as a normal variable.

## anonymous function

↳ function without a name

# Named function expression

name

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

3

note: you can call b(); but you cannot call xyz. as xyz has local scope, does not exist in outer memory. outside var b.

## first class functions / first class citizens

instead of passing arguments, you can pass functions in arguments

→ pass in anonymous function

functions treated as values.

simply, first class functions means, js treats functions as normal variables.

The ability of functions to be used as values and can be passed as arguments to another functions and be returned from a functions

↓ First class functions.

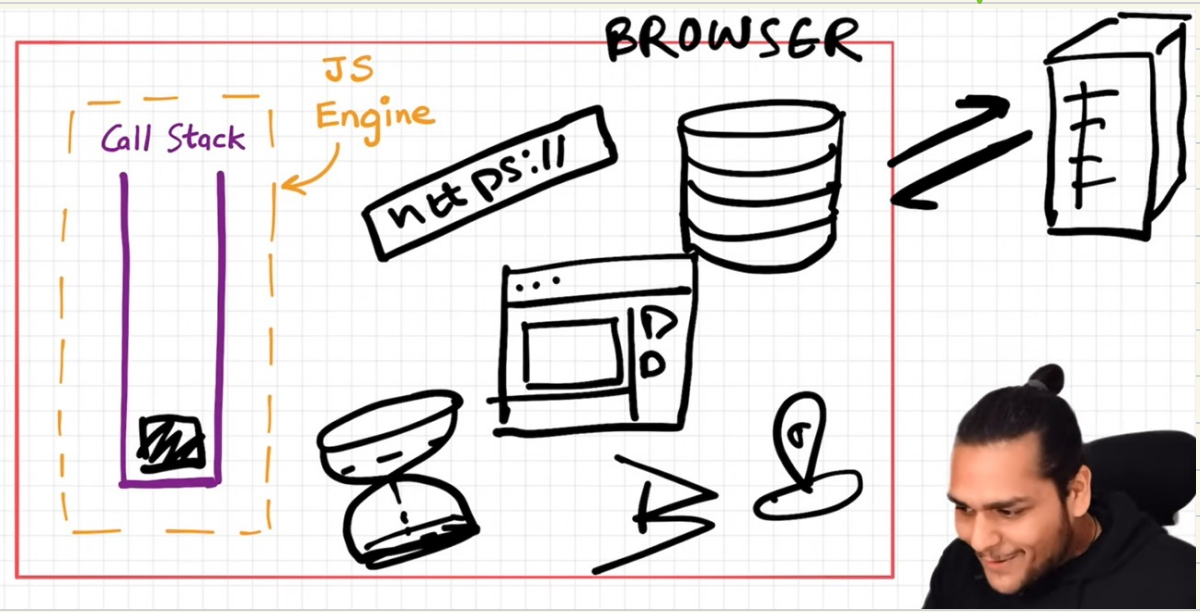
# Call back func".

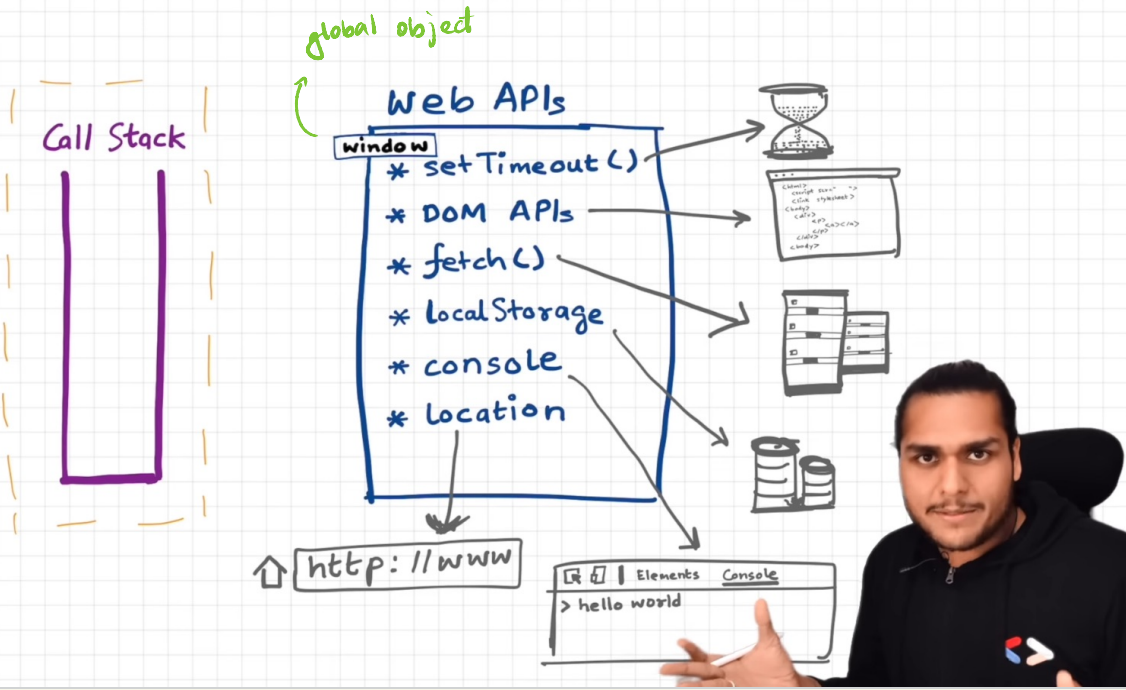
functions passed to other functions.

callback func" is a function that is passed to another func"

# Asynchronous js + Event loops

+ everything goes in call stack, and it gets executed.
+ call stack does not have a timer
+ we need web APIs to use a lot of functionalities

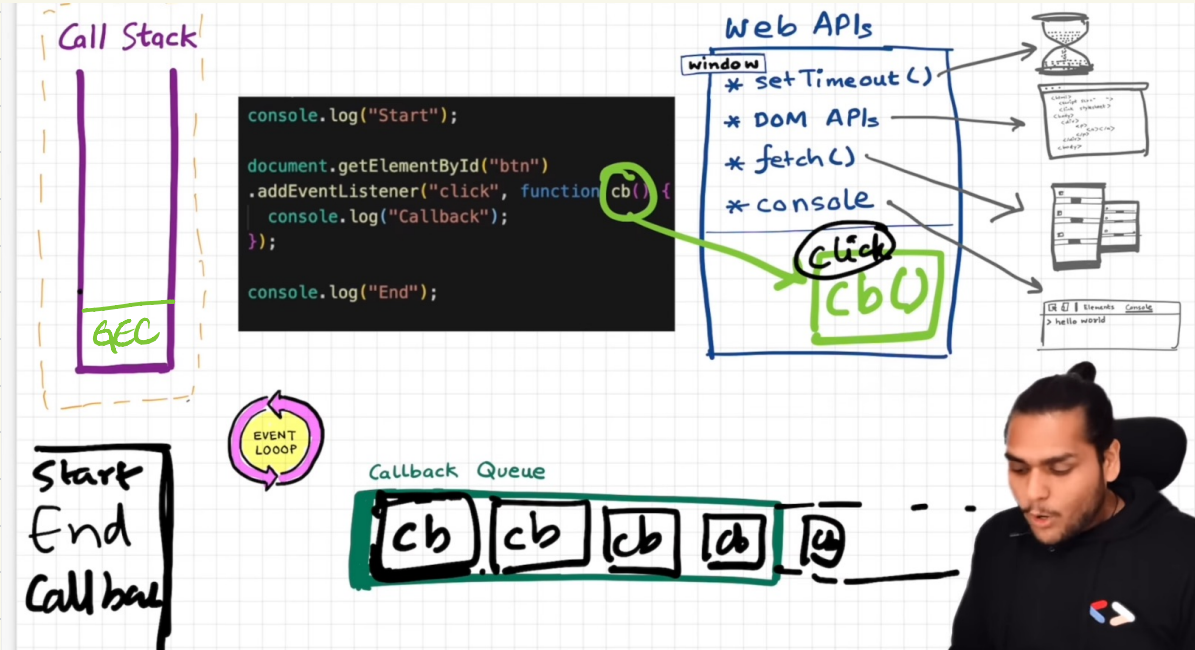





use window (global object) to access the web APIs

all web APIs are present in the global scope.

# how does the callback stuff work in context of browser?

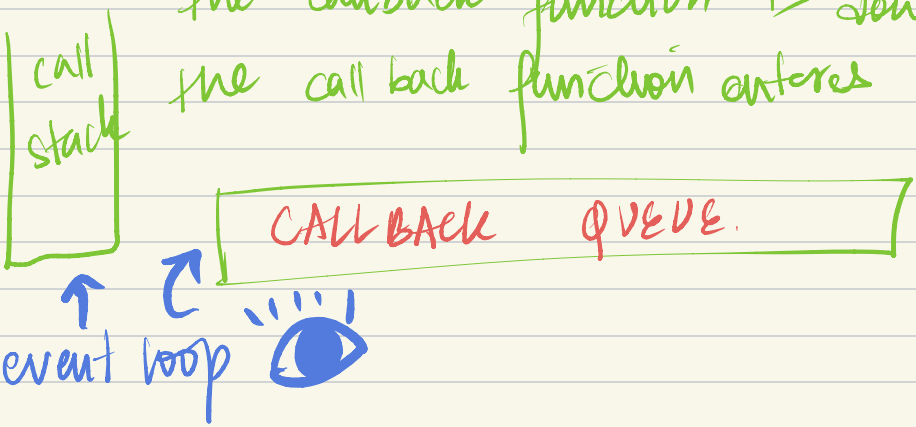



## how things work (Web API's → setTimeout, DOM API's)

1. The code is executed line by line
2. As soon as code starts executing, a global execution context is created
3. If a callback function is encountered, it has a separate context (not in call stack) and it gets stored there.

4. meanwhile the stuff in GEC keeps executing.

5. Once the timer or whatever is holding the callback function is done, the call back function enters a




6. An event loop keeps monitoring the call stack AND as soon as callstack is empty

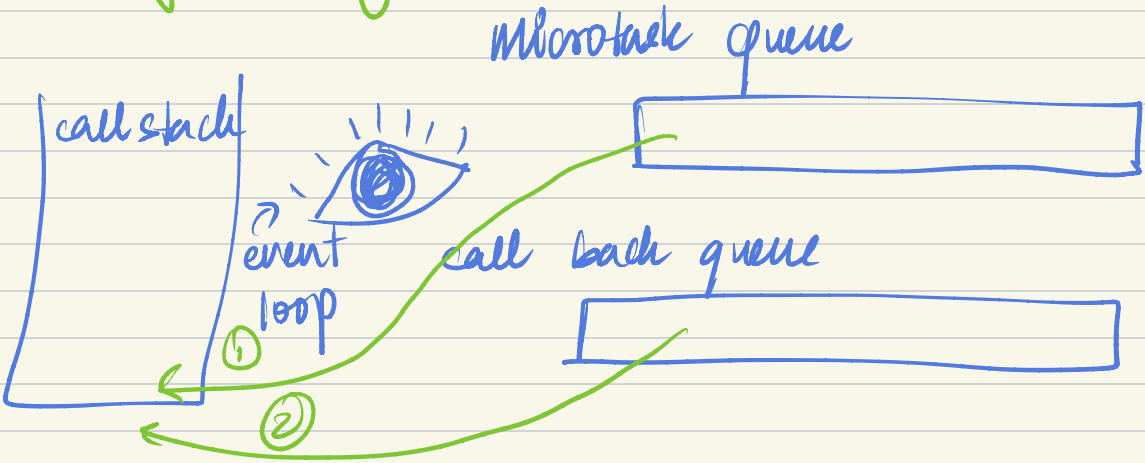
(the main GEC code finishes executing)

it pops the call back function from callback queue and pushes it into the call stack

# fetch()

↳ this waits for some input from https://
hence its has the web API scope and does
not exist in the global execution context.

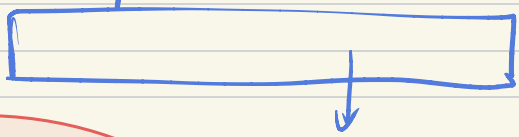
some high priority tasks go to the




first priority given to micro task queue
and then functions in call back queue.

all tasks in microtask queue should be
executed and the call back tasks get
executed.

Microtask queue / task queue.



high priority

Promises and mutation observer

Starvation of the Call Back Queue.

# JIT compilation

↓

both

interpretter

+

compiler

memory heap
all variables and functions assigned memory here.

* garbage collector
* optimization

Parsing
↓
compilation
↓
execution

javascript :

procedural
functional
OOP lang

# Set time Out issues

does not guarantee that the timer expires exactly after the specified time.

The reason :

- Until the global execution context is completely empty, The setTimeout() has to wait in the call back queue even if its timer has expired.
- Once GEC is empty setTimeout can enter the GEC and start executing.

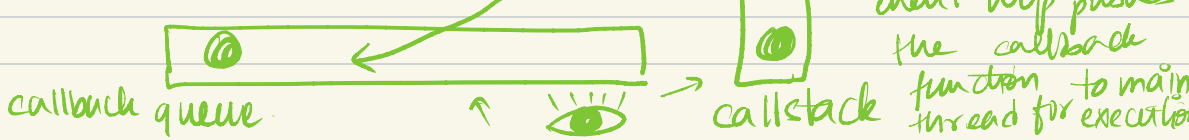
i.e the timer in setTimeout() determines when the call back function enters the call back queue, does not determine when the call back function gets executed.

```
js > JS index.js
1   console.log("Start");
2
3   setTimeout(function cb() {
4       console.log("Callback");
5   }, 0);
6
7   console.log("End");
8
```

output

Start
End
Callback

once end is printed then callback fun^n goes into the callback queue then End is printed and later GEC gets finished then event loop pushes the callback function to main thread for execution



Event loop

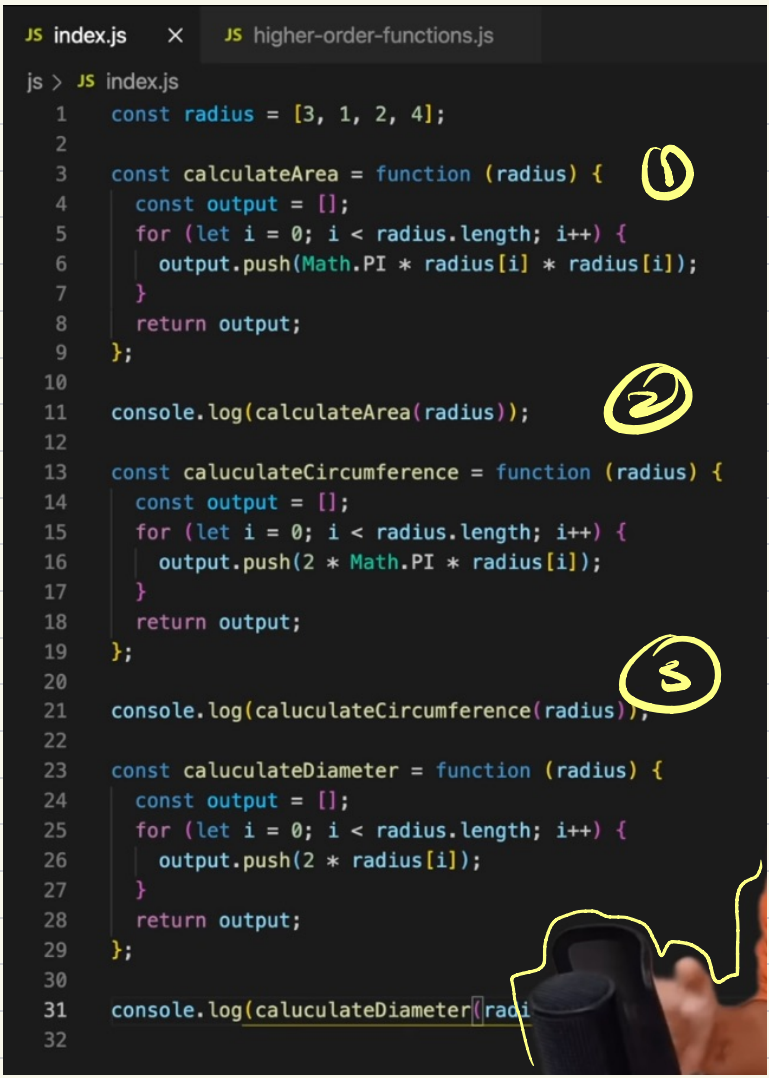
# higher order functions

a function that takes another function as an arg
OR
returns another function

You should write → • Modular • Reusable Code

DRY principle

dont repeat yourself.




↓Repetitive code

Extract logic.

# Modular Code.

index.js ● higher-order-functions.js

js > index.js

```
1   const radius = [3, 1, 2, 4];
2
3   const area = function (radius) {
4     return Math.PI * radius * radius;
5   };
6
7   const cicumference = function (radius) {
8     return 2 * Math.PI * radius;
9   };
10
11  const calculate = function (radius, logic) {
12    const output = [];
13    for (let i = 0; i < radius.length; i++) {
14      output.push(logic(radius[i]));
15    }
16    return output;
17  };
18
19  console.log(calculate(radius, area));
20  console.log(calculate(radius, area));
21
```

← extract logic

generic function that calculates any logic given to it.

map is a common higher order function.

the calculate function is exactly like map

Array.prototype.calculate

↳ this function becomes available to all arrays in your code

# Map

arr.map( ... ) ← send a function that you want to perform

↓

it will do it on all elements of the array

# filter

the values in an array. for specific requirements you use this function.

* find odd values in array

arr.filter(function_here)

↳ add a odd function logic

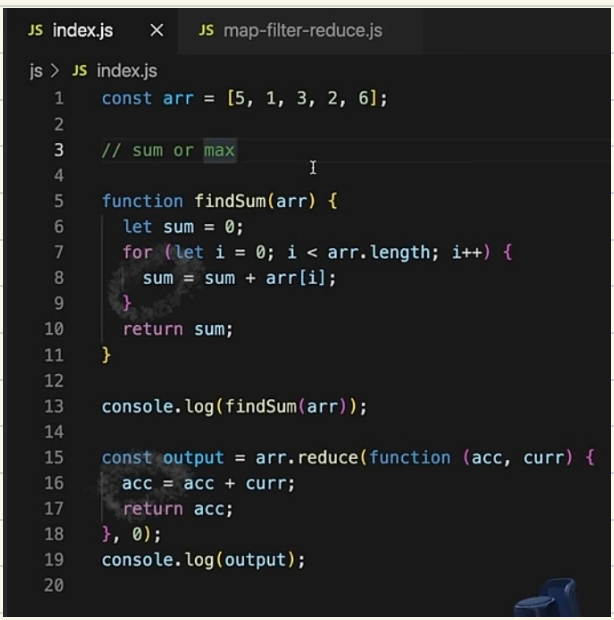
filter will iterate through the array and check if it matches the require mark

Me

# Reduce

does not reduce anything

when you have to take all elements of the array and get a single value from them.




acc ↔ sum

arr[i] ↔ curr.