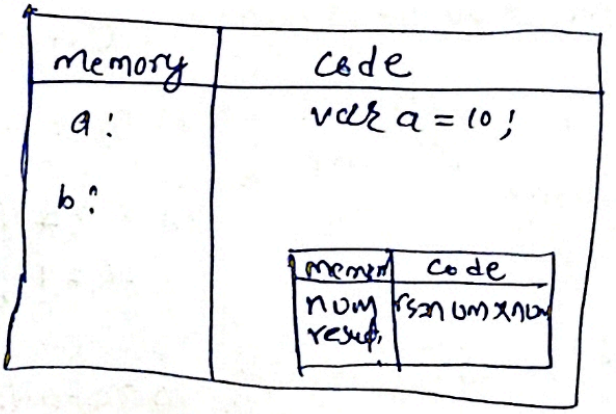
In Java Script Everything happens under the Execution context.

⇨ Execution context has two part.

memory	code
variables & methods	↳ code line 1 ↳ code line 2 ↳ code line 3 : ↳ code line n

Phase 1:- allocate memory ~~to~~ to each variable & function before execute a single line. all the variable will be placed by "undefined" value (hoisting concept)

Phase: 2 Execute the code line by line, whenever compiler encounter a new function invocation it will create a new Execution content in code execution phase



Hoisting:- var, function are hosted into their scope/Globalscope

let & const will get special memory block (script)

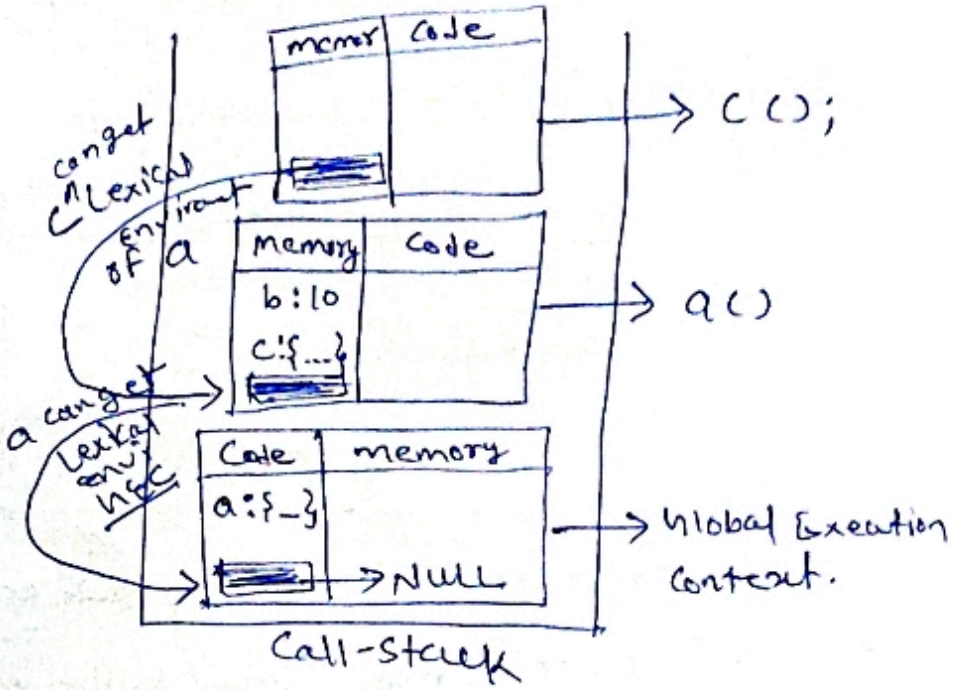
* let & const also get hoisted in "temporal dead zone"

Lexical Environment & scope chaining :-

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

    }
}

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



Lexical Environment: Local memory & it's parent ~~reference~~ Lexical Environment.

There are three type of Errors we encounter mostly.

⇨ Syntax Error :- violation of JS syntax, no code will execute in such case.
⇩
Compile time Error!

```
const a;
a = 10;
```

⇨ Type Error :- While trying to re-initialize the const variable.

```
const abc = 10;
abc = 20; // Reference Error, type Error.
```

⇨ Reference Error :- While trying to access the variable which is not there in global memory.

```
console.log(x); // without declaring x.
```

⇨ Block scope & shadowing :-

① **What is Block :-** Inside two {...} whatever we write is called block. or write multiple statement into a block. called block/Compond statment.

② **What is block scope :-** what all variables ~~& functions~~ under a block.

③ **shadowing :-** same type of variable in globel scope & local is called shadowing

```
var a = 100;
{
    var a = 10;
    let b = 100;
    const c = 1000;
    console.log(a, b, c);//10, 100, 1000
}
console.log(a);// 10
```

a is shadowing here but in case of let & const it will allocate memory in seprate scope.

Illegal Shadowing :-

```
let a = 10;
{
    var a = 100;// Error
}
```

It's called Illegal shadowing.

Closures :- function along with it's lexical scope bundled together called as closure.

```
function x(){
    var a = 10;
    function y(){
        console.log(a);
    }
    return y;
}
var z = x();
console.log(z);
//.....
console.log z();
```

Here z remeber it lexical scope and print value of "a" = 10;

Uses of closures:-

- → module design pattern
- → currying
- → function like once.
- → memoize
- → maintaining state in async word.
- → setTimeouts
- → Iterators.
- → many more.....

⇨ SetTimeout + closures Interview Questions :-

→ Javascript wait for none.

```
function x(){
    for(var i=1; i<=5; i++){
        setTimeout(function(){
            console.log(i);
        }, i*1000);
    }
    console.log("Namaste Javascript");
}
x();
```

use let to fix the issue.

↓ fix with var only

```
function x(){
    for(var i=1; i<=5; i++){
    function close(x){
        setTimeout(()=>{
            console.log(x);
        }, x*1000);
    }
    close(i);
    }
}
```

⇨ Closure with more Examples:- A function along with outer parent Environment Access form a closure, In other word closure is a combination of a function and It's Lexical scope is called closure.

⇨ What is anonymouse functions?
→ without name declared function called anonymouse fn

⇨ what is first class function?
~~when~~ A function can be passed as argument & It can be return from another function & a function can be assigend to another variable ~~I R know as first class function.~~
this ability is called First class function. OR first class citizens.

function statment / declaration

```
function xyz(){
---
}
```

function Expression

```
var b = function xyz(){
---
}
```

Higher order function

⇨ H.O.F is a fn ~~which~~ that recives a fun as an argument or return the function as output
some example:-
Array.prototype.map
" " .filter
" " .reduce

→ H.O.F. that takes atleast one first class fn as an argument or return one atleast.

First class function → that are treated like an object (or assignable to a variable)

```
var b = function xyz(){
---
}
//here b is first class function
```

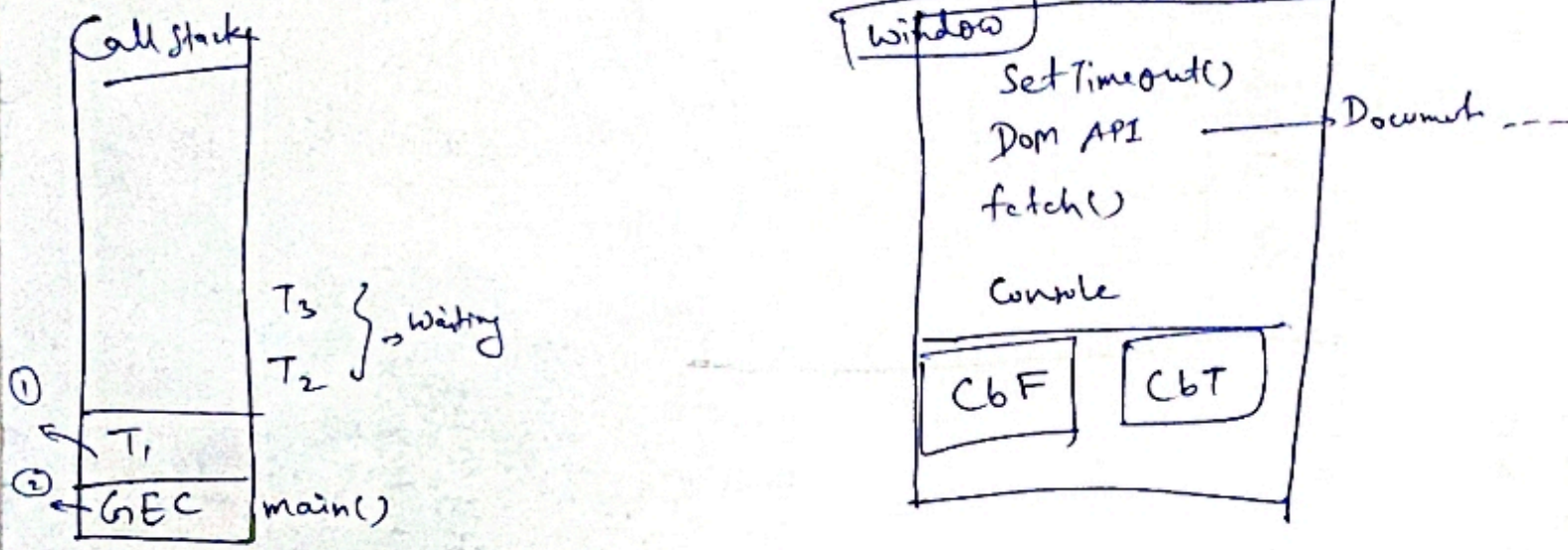
~~Callback~~ :-

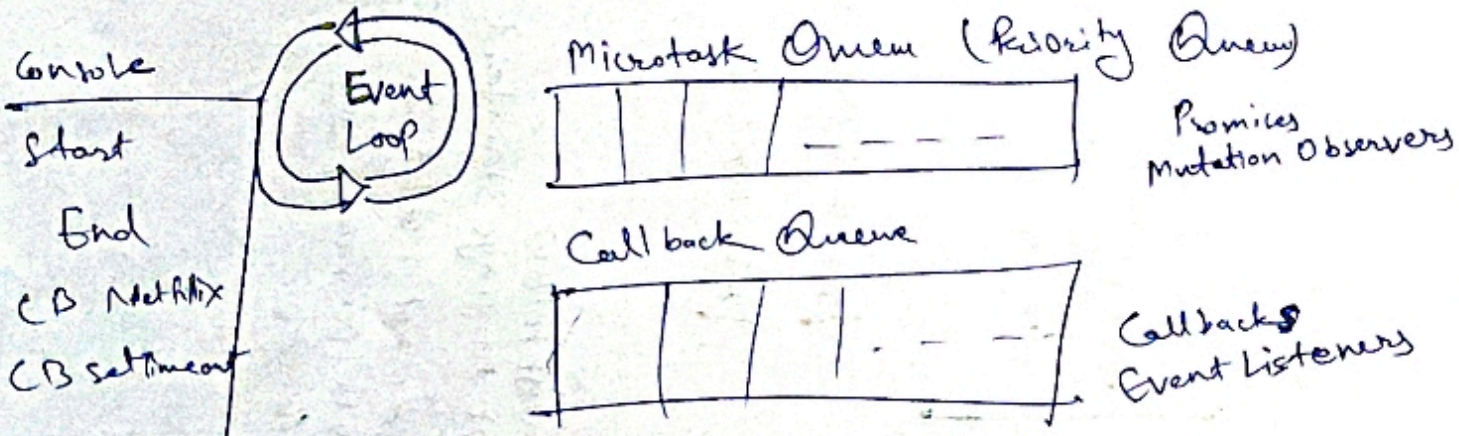
Event Loop :-

Code →

T1
T2
T3

Web APIs / XHR





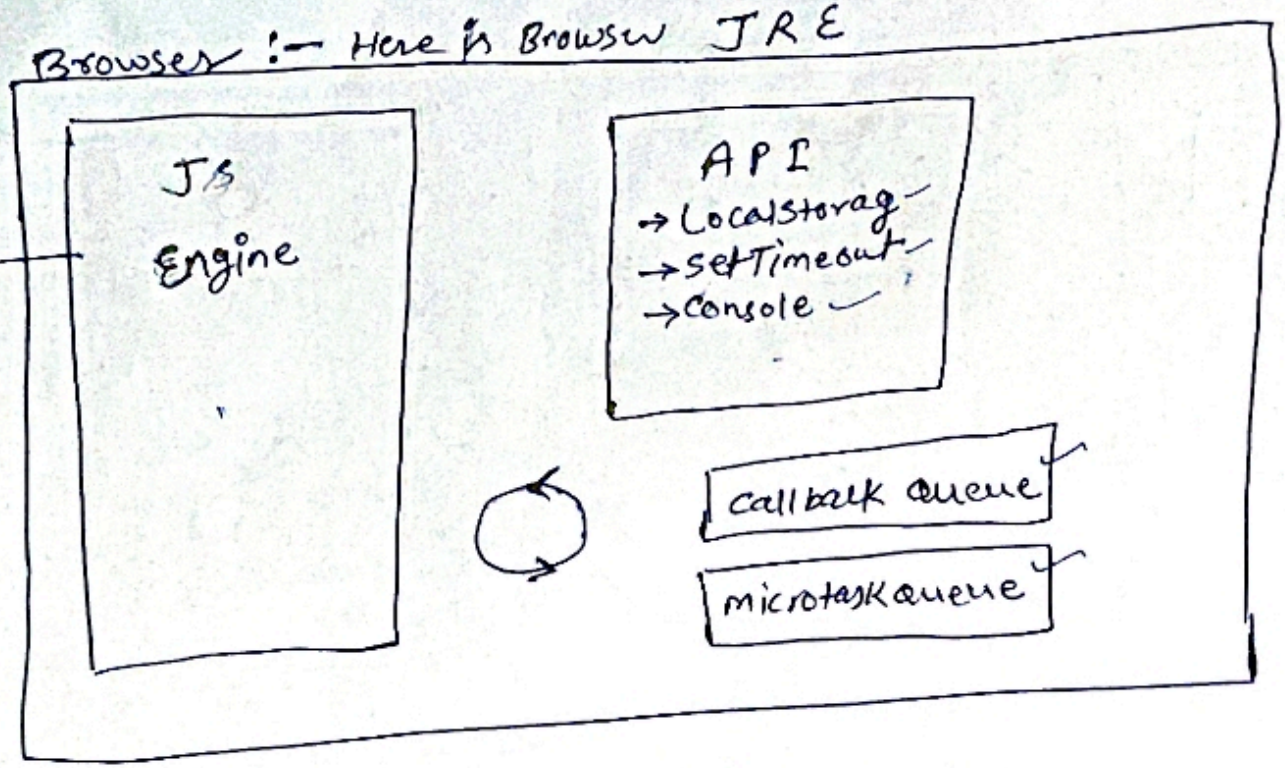
Starvation

Definition → ~~whenever~~ Since JS is a single-threaded Non-blocking language, everything happens inside Call stack (line by line). (Execution Context)

Once an asynchronous task happens inside Callstack, it refers to Web APIs and it gets removed from Call stack (GEC follows after that). Once Web APIs returns the results it will go inside Callback Queue. Now there comes Event loop, which keeps checking the Call stack and Callback Queue to execute the task. (Frequency of Event loop → 16.6 ms)

```
Console.log ("Start");
SetTimeout ( function cbT() {
 Console.log ("CB SetTimeOut");
}, 5000);
fetch ("https://api.netflix.com")
.then (function cbF(){
 Console.log ("CB Netflix");
});
Console.log ("End");
```

Javascript Runtime Environment



JS Engine is heart of JRE

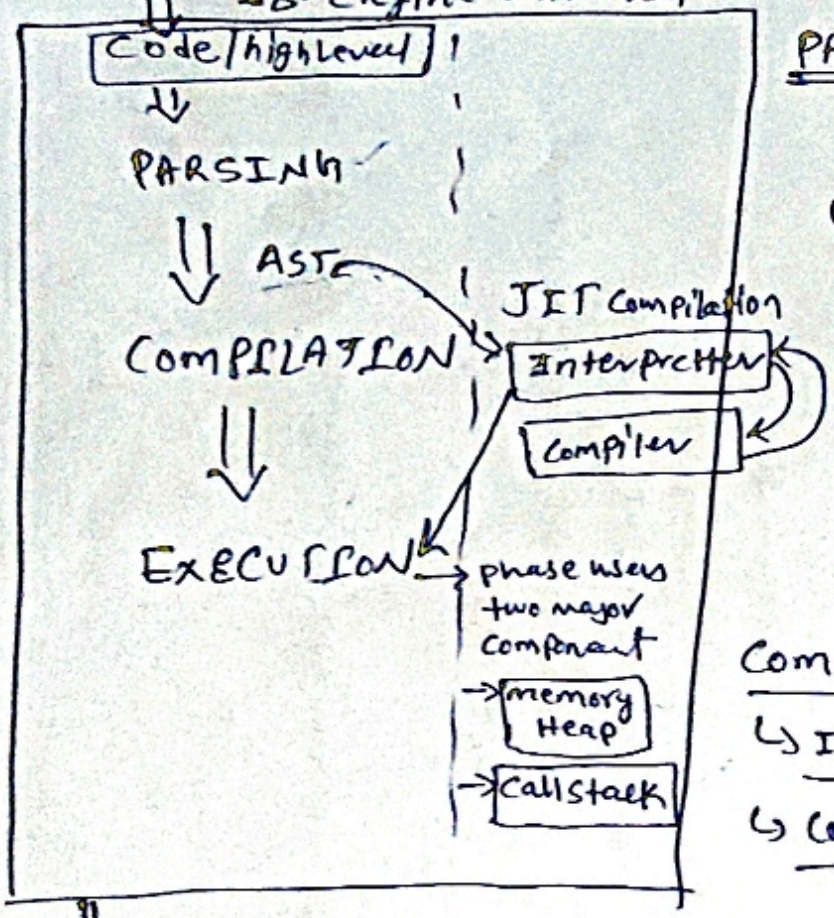
Nodejs
↳ also has there JRE

Javascript Engine
→ chakra → microsft
→ spidermonkey → mozilla
→ V8 → chorme

→ JS Engine must follow ECMAScript language standard.

→ JS Engine is not a hardware it's an peace of code/program. which executed by machine/browser.

JS-Engine Internal architecture

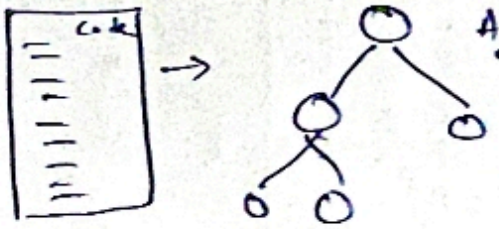


JS Engine also has
G.C → using mark & sweep Algo

PARSing :- (i) Code will be convert into tokens.

Let a = 7; ← tokens

(ii) **Syntax Parser :-** Code will convert into AST (Abstract syntax tree)



Check :- astexplorer.net

COMPILATION :- JIT Compilation :-

↳ **Interpretter :-** takes your code and execute line by line

↳ **Compiler :-** It's Compiles your code and generates an optimized code for you.

⇨ Javascript can be both Interpretter + Compiled Language which we say JIT (Morden browser & JS Engine)

* most Powerful & Efficient JS engine till now is

Google - V8

↳ Ignition (Interpretor)

↳ Turbofan (compiler)

↳ Orinoco (G.C) → using Mark & Sweep Algo

Debouncing :- Debouncing In javascript is a pratice used to improve browser performance. debouncing is used to ensure that time consuming tasks, do not fair so often, that it stalls the performance of the page.

The debouning() function forces a function to wait a certain ammount of time before runing agin. The function is buid to limit the onumber of time a function is called.

Html

```
<Input type="Text" onkeyup="getdata" />
```

→ if we replece getdata → betterfunction() It will call after 300 ms.

javascript

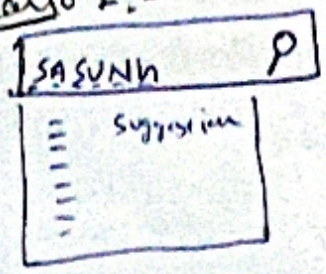
```
let count=0;
function getdata(){
    console.log("getdata called", count++);
}

const debounce = function(fn,delay){
    let timer;
    return function(){
        let context=this;
            args = arguments;
        clearTimeout(timer);
        timer = setTimeout( () =>{
            fn.apply(context, arguments);
        }, delay);
    };
}

const betterfunction = debounce(getdata, 300);
```

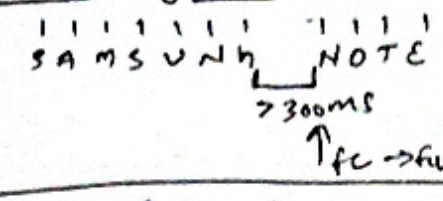
⇨ Debouncing & Throttling :- Optimized the performance of a web application.

Senario 1 :-

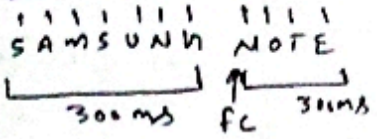


onkeyup="getResult"

⇨ Debouncing case :-



⇨ Throttling case :-

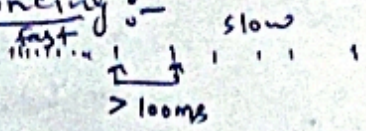


In Throttling function call will happen for each "keyup" event. It will wait for 300 ms then make a call.

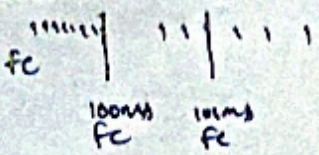
* In e-Commerce search senario → debouncing make more sense.

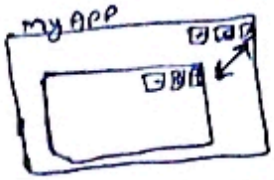
Senario 2 :- How often a user is resizing our application.

Debouncing :-



Throttling :-





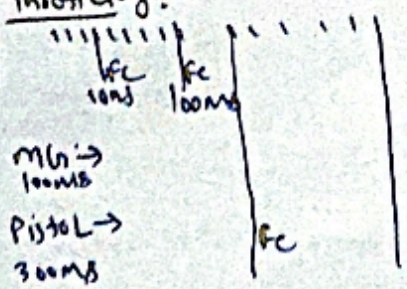
addEvent("resize", () => {})

Senario-3 :-

onclick="saveData"

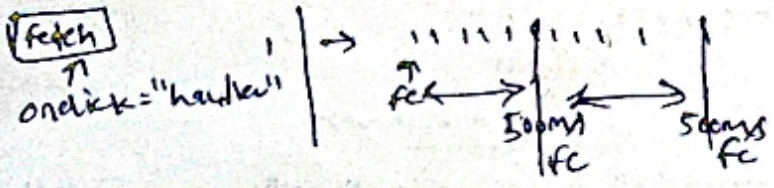
debouncing :-

Throttling :-



Shotting game is best senario for Throttling

Throttling :→ used for optimizing performance of an application OR limit the rate of the function/API call for a certain amount of time.




```
Const Expensive = () => {
    console.log("Expensive");
}

window.addEventListener("resize", Expensive);

Const betterExpensive =
    throttle(expensive, limit);
```

```
Const throttle = (func, limit) => {
    Flag = true;
    return function () {
        if(flag){ // flag to limit OR throt
            func();
            flag = false;
            setTimeout(() => {
                flag = true;
            }, limit);
        }
```

⇨ sum(1)(2)(3)(4)..... (n)();// Amazon UI/frontend question

```
let sum = function(a){
    return function(b){
        if(b){
            return sum(a+b);
        }else{
            return a;
        }
    }
};
console.log(sum(1)(2)(3)(4)());// 10
```

This is a recursion function calling problem. here we are calling sum(a+b) => until I get 'b' as undefined.