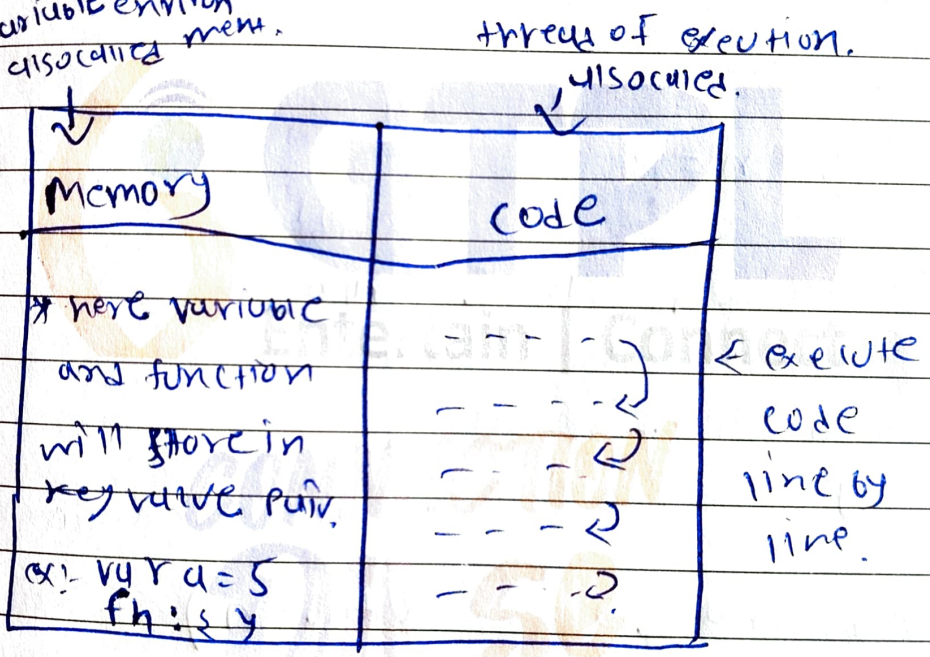
Date: 26/05/25

* Namaste JS

EP-I

* Everything in JS happen inside an execution context.

* Execution context



* JS is a synchronous single threaded language which means code will execute line by line in order.

EP-II

* GEC → will be created using 2 phase.
1. memory creation phase
2. code execution phase.

Date:

① In memory creation phase.

⇒ variable will be Initially assigned as undefined.

⇒ and function will be store as it is

Ex-

```
var n = 2
function square (num) {
    var result = num * num
    return result
}
var square2 = square(n)
var square4 = square(4)
```

(parameter, Argument, function invocation)

* code execution.

memory phase.

var n = undefined.

function square () {...} as it is store.

var square2 = undefined

var square4 = undefined.

② now code execution phase.

now n = 2.

var n = 2 ↙ will allocate memory.

* function square have parameter num.

So it will create another execution context and num will be undefined first.

then result variable is also undefined.

then value will be assign.

num will 2

result will be 2 * 2 = 4.

then return result means here program will end and return the result number 4.

So var square2 = 4

like that wise = 16

square4

* whenever function invoke new jee will create. and when return the value jee will delete.

Date:

* all these process managed by call stack in JS

After execution It will pop up

this will also pop up from call stack after all execution of code.

IMPORTANT NOTES

EP-4 → Hoisting in JS.

* Hoisting in JS means irrespective of the variable and function declaration we can access it even before their declaration.

* Hoisting is the default behaviour of JS where all the variable and function declarations are moved on top.

* examples.

hoisted variable = "Shiv"

console.log (hoisted variable)

var hoisted variable

⇒ the op ⇒ Shiv

*

hello() ← op will be Hello Shiv even before function declare.

function hello() {
console.log ("Hello Shiv")
}

Date:

*. Note:- variable Initialization are not hoisted, only variable declarations are hoisted.

ex:- var x;
console.log (x)
x = 24

→ op - Undefined.

* Arrow function will work as variable ~~defta~~ in JS. It will not same as it is.

⇒. ex:-
hello() ← function Invocation.
const hello = () => {
console.log ("Hello Shiv")
};

⇒ op ⇒ undefined.

*. let and const variable will be in temporal dead zone and will get reference error in JS hoisting

Ep-5

Date:

* How function works in JS & environment variable.

Ex:-

```
var x = 1000

a()
b()
console.log(x)

function a(){
  var x = 10
  console.log(x)
}

function b(){
  var x = 100
  console.log(x)
}
```

OP => 10
100
1000.

=> here first memory creation phase. where variable x will be undefined and then 1000 will allocate.

=> then a() function invoke then another Execution context will created and variable will undefined then 10 will be allocated. and then it will pop up from call stack.
=> 10 will be return to GEC.

=> then function b will be invoked and another Execution context will created. and variable first undefine then value 100 will be assigned then it will return to GEC. -> 100

=> then line 3 console.log(x) so it will first search in inner local environment where already x = 1000 so it will assigned the value.

=> so OP will be 10
100
1000

=> After that GEC also pop up from call Stack.

GTPL Entertain | Connect — Connection Dil Se

Date:

* ep-6.

*. Shortest JS programme and window and this keyword.

=>. empty is shortest program in JS.

=> If we debug this

=>. there is global execution context created by js engine in browser which are built in method of JS

=>. what is global space :- global space means anything outside of the function are called global space.

=> In function it will call localspace.

=>. this and window are the global space.

=> whatever variable define in global space we can access that.
eg window.that variable name

Date:

or

=> this. variable name
or
variable name.

*. ep-7.

* undefined vs not defined in JS.

* even before program run in JS memory will created.

=> In that the first phase. memory creation. Phase.

=> It will take variable and functions in it.

=> variable at first undefined which means it will allocate memory for that variable in memory but the value haven't assigned yet. we can say that undefined means place holder for future variable value.

Date:

* and not defined means.

=> value never exist in our program and we are try to get that value this will generate reference error.

* do not use undefined manually in JS.

like ex:- vary = undefined.

* JS is loosely type language which means variable can have multiple duty type values It's not fixed duty types.

ex:-

var a = 10

console.log(a) => op => 10.

a = 24

console.log(a) => op => 24.

a = "Shivam"

console.log(a) => op => Shivam.

Date:

* op-8 the scope chain and lexical environment

* what is scope in JS

=> scope means function or variable accessability in our code.

* what is lexical environment.

=> whenever execution context created lexical environment also created. that means a local memory along with lexical environment with it's the parent.

=> that means it can access it's parent variable also.

=> but first it will try to search in local memory if it's not available then it will search in parent lexical memory.

=> the lexical memory of parent also called scope chain.

Date:

ex:-

```
function a() {
  var num = 24

  function b()

  function b() {
    function c() {
      console.log(NUM)
    }
  }
}
```

* again writing this example clearly.

```
function a() {
  var num = 24
  b()
  function b() {
    console.log(num)
  }
}
a()
```

⇒. the op ⇒ 24.

Date:

⇒ In example we see that function b will first try to search NUM variable value in local memory If it's not found then it will get have access of their parent lexical environment. and there num = 24 so it will assigned or take num = 24 from it's parent scope.

*. Eg. let and const.

* let and const variable are hoisted for temporal dead time zone only.

*. let and const variable are not store directly in global memory space instead of they are saved in different memory called script

* so in this different memory space we can not access variable which do not have value in it and this why it will give reference error when we try to access

Date:

* what is temporal dead zone.

=> Since then the let and const variable hoisted means some memory space using undefined allocate and till it's initialize some value this is called temporal dead zone.

* whenever if we try to ~~var~~ access variable in temporal dead zone it will fire Reference error can't access value before it's initialization.

* variable redeclaration in same scope will not work in let and const it will generate syntax error and code will be not run

* ex:- var a = 10
var a = 100
} will work.

but let b = 24
let b = 26
} In same scope will generate syntax error.

Date:

* using let variable we can declare and initilized later.

let a
a = 24
} this will work.

* but In const we have to initilized variable at a time of decldration or else it will generate error.

* ex:- const a
a = 24
-> will not work and generate syntax error., missing Initialiser In const declaration.

const a = 24
-> will work.

* using let we can reassigned different values to our variable.

ex:- let a = 24
a = 25
console.log(a)
=> OP -> 25

Date:

* but using const we can't reassigned value to const variable It will generate a type error.

ex!- const a = 24
a = 25

console.log(a)
↓
or -> type error.

* let and const both have block scope.

* and var has functional scope.

* How to avoid temporal dead zone

=> to avoid temporal dead zone, or unexpected error you should declare and initialize variable at the top of the scope.

* So which variable declaration we should use. (1) (2) (3)
-> Priority -> const -> let -> var.

Date:

* GP-10. Block scope and shadowing in JS.

* what is block in JS and why it is used ?

=> Block & is also called compound statement.

=> Block used to combine multiple statement in one group

=> why we used block using block we can use multiple statement in group using block where JS wants only one statement.

ex!- if (true) { here we used single statement }

=> but if we used to want to use multiple statement in group and want to use in single statements.

* Block SCOPE :- what all the variable and function accesible within a given block called...

Date:

* Ex!-

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

=> op =>
10
100
1000
10
reference error.

* here in above code var a will be assigned in a global scope where let b and const c variable are assigned in different memory space.

=> that's why in scope variable a, b, c all accesible.

Date:

=> but let and const are block scope so outside it will not accessible.

=> and a is already in global space so it's accessible.

=> that's why we are getting this op

* what is shadowing in JS.

Ex!-

```
var a = 10
{
  var a = 20
  let b = 40
  const c = 60
  console.log(a), console.log(b), console.log(c)
}
console.log(a)
console.log(b)
console.log(c)
```

* code explanation -> and op
20
40
60
20 reference error.

Date:

=> as we can see there are total 3 variables using var, let and const. var will be assigned in global memory space even if's in block.

=> let and const will be assigned in a different memory space.

=> var a = 10 first.
out in block code.

a = 20 so new value will be assigned to variable a which is already in global space.

=> so op will be.

20
40
60
20
reference error.

IMPORTANT NOTES

* another example.

```
let a = 24
{
    let a = 36
    var b = 50
    const c = 10
    console.log(a)
        //    (b)
        //    (c)
}
(console.log(a)
    //    (b)
    //    (c)
```

=> op =>
36
50
10
24
50
reference error.

Date:

* why because let and const will be stored in another memory space.

=> let a ~~var~~ = 24 will be stored in script. memory.

then block. let a will be stored in block memory

var b will be stored in global space.

and const c will be stored in block memory.

* this shadow will works similar way in function too.

* what is a illegal shadow

```
let a = 20
{
    var a = 40
}
```

global space. vice-versa,
var a = 20
{
let a = 40
}
will work

will generate syntax error

Date:

* legal shadowing.

```
let a = 20
    {
      let a = 30
    }
```

⇒ will work.

* lexical scope also works in block scope

ex:-

```
const a = 10
  {
    const a = 100
    {
      const a = 1000
      console.log(a)
    }
  }
```

⇒ OP => 1000.

If its not available variable a in their scope then it will access parent lexical scope.

Date:

ex:- let a = 20

```
{
  let b = 40
  {
    console.log(b)
  }
}
```

⇒ in above code first it will find in their current scope and then it's not available then it's goes to it's parent scope and b available there and b value 40 so

OP => 40.

* EP-12 closure in JS

* A closure is the combination of a function bundled together (enclosed) with reference to its surrounding state.

⇒ In other words a closure gives you

Date:

access to an outer function's scope from an inner function. In JS it's closures are created every time a function is created, at function creation time.

* ~~In short function~~

ex:-

```
function x () {
    const q = 24
    function y() {
        console.log(q)
    }
    y()
}
x()
```

=> In above program function y will creates a closure of variable x in parent scope.

=>. ~~first~~ function along with lexical environment called. — and still remember the it's lexical scope

=>. Even after pop up from call stack.

ex-

```
function x() {
    var q = 7;
    function y() {
        console.log(q)
    }
    y
    return y;
}

var z = x();
console.log(z)
z().
```

=> op => f y() { console.log(q) }

y

-> 7.

*. function when they are return from another function they still maintain their lexical

=> they still remember where they exist.

Date:

EP-16. functions and their diff cargens and meaning.

* function statement and declaration both are same. how we define our...

ex:- function a() {
    console.log("a")
}

-> function declaration

a() -> function Invocation

* function expression.

let b = function () {
    console.log("this is b")
}

=> here we are assigning or attaching our function to variable b

=> ① function declaration vs function expressions.

Date:

* function declaration will save in memory as it when hoisted and we can access that.

* but in function expression function will be stored as variable so first's it's undefined and if we try to access before it's initialization it will throw error this is the main diff between...

* Anonymous function.

* a function which have no name called Anonymous function you can use this as function expression only or else it will throw error due to ECMA rule.

ex:- const a = () => {
    console.log("a")
}

* Named function expression

ex:- let b = function a() {
    console.log("named")
}

Date:

* Parameter & Argument. (placeholder → parameters)

```
function multiply (num1, num2) {
    return num1 * num2
}

multiply (2, 2)   // 2, 2 ← Argument.
```

* first class function or first class citizen.

=> function can take function as argument and function can return function this is called first class function.

```
function a() {
}

function b (function a) {
    return function a
}
```


Date:

```
function b()   function a() {
               }
```

* ex:- function can return function

```
function c () {
    return function d () {
        console.log("d")
    }
}
```

c() -> op will be -> d

where function d()

=> Arrow function is new shorthand property to define function.

ex:- let a=() => {

```
    console.log("a")
}
```

a() => op -> a.

Date:

* parameter & Argument. (placeholder → parameters)

```
function multiply (num1, num2) {
    return num1 * num2
}
```

multiply (2, 2) ← Argument.

* first class function or first class citizen.

=> function can take function as argument and function can return function this is called first class function.

```
function a() {
}

function b (functiona) {
    return function a
}
```


Date:

```
function b() ← function a() {
}
```

* ex:- function can return function.

```
function c() {
    return function d() {
        console.log('d')
    }
}
```

c() -> op will be -> d

where function d()

=> Arrow function is new shorthand property to define function.

ex:- let a=() => {

```
    console.log("a")
}
```

a() => op -> a.

Date:

* V-16 EP-14.

* callback function.

=> function which take function as an arguments are called callback function.

> why callback function used

without callback function why can't able to asynchronous programming

=> what is asynchronous -> a function which will take some time to execute after execution it will execute

ex:- setTimeout, async-await, API.

> but what is the need for that.

=> JS is synchronous single threaded language which means it will execute code line by line at wait for none.

Date:

=> So in synchronous single threaded language we can use asynchronous programming with the help of callback function.

ex:- setTimeout(() => { console.log("timer"), 2000}

function q (b) {

console.log(4)

b()

}

q(function c() { ← function as argument
return console.log("b")
}) ← function invocation

=> OP => q
b
timer ← after 2 second
timer will pop up in call stack

Date:

* we have never block our execution thread or call stack.

=> long time taking task we have to use our asynchronous programming.

=> what is smartly garbage collector variable and function which are not used in program will not be saved hence we can save our memory

* EP-15 Async JS & EventLoop

=> JS is synchronous single threaded language which means it will execute code line by line inorder and wait for none.

* ex:-> function a() {

}

console.log("a");

}

-> a()

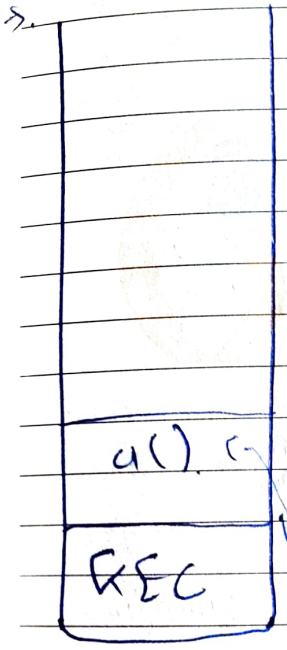
console.log("End")

Date:

* now now this program will execute

=> JS has 1 call stack where all the program runs in sec

=>



global execution context will be created it has 2 phase 1. memory creation 2. code execution.

1. phase memory creation -> function a will store as it is

then line 5 there is a function a invocation so new execution context will creat

call stack

=> in a there a console.log("a") so it will print a in console after that it will pop up from call stack.

=> then in gec line 6 there is a

Date:

console.log "End" so it will log the end in console.

⇒ After that GEC will pop up from call stack and our program finished there.

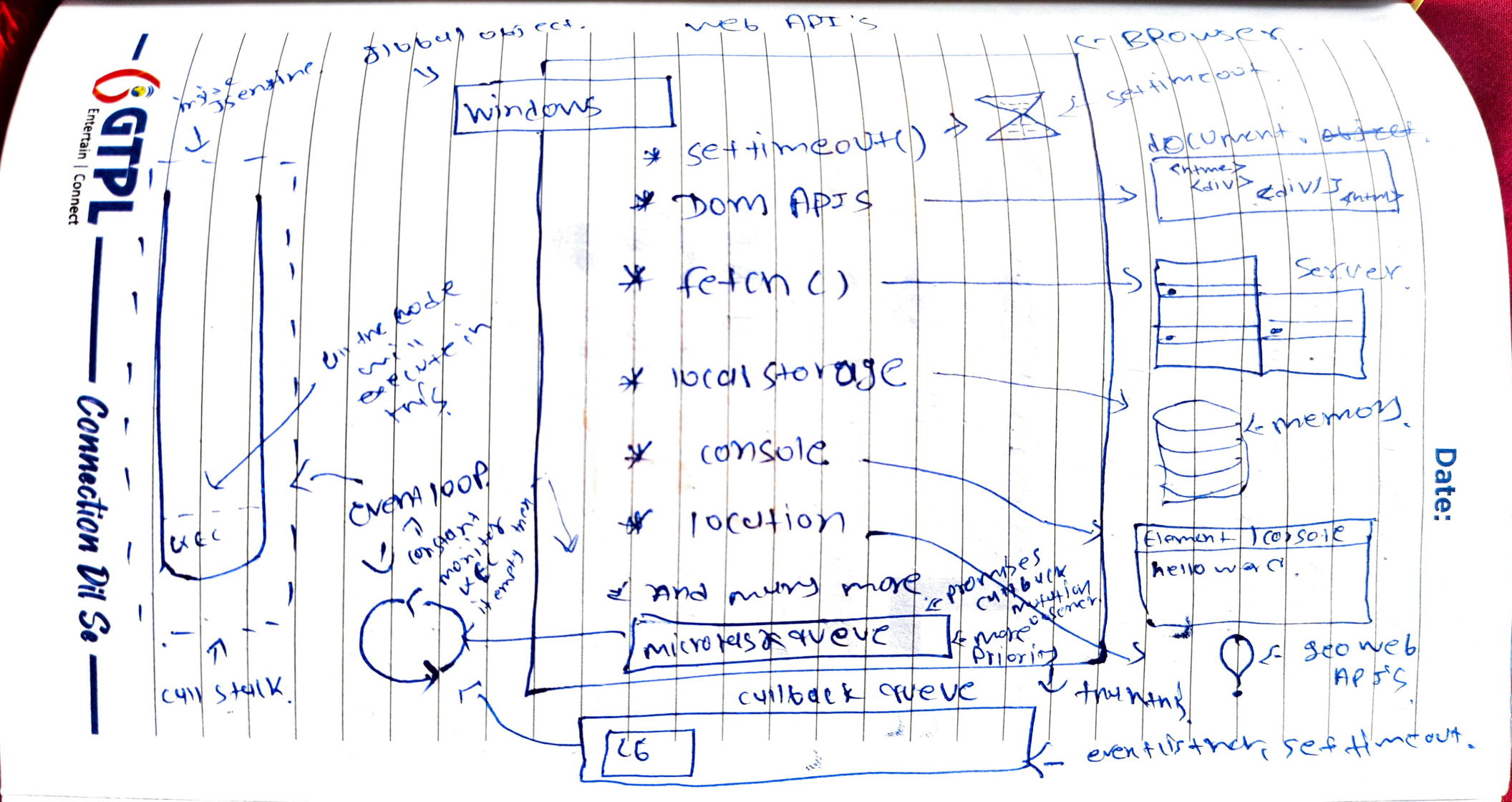
⇒ Call stack the main job to execute whatever inside it.

⇒ But what if we need to wait for something in program.

⇒ wait if our program need to execute after 5 second or any time given period

⇒ Can we done? In JS but in JS is synronous single thread language which means it will wait for none and execute the code so now can we do waiting, and asynchronous task in JS language.

⇒ To achieve this we need timer.


GTPL
Entertain | Connect
Connection Dil Se
Date:
Global object
Web API's
Browser
Windows
my JS engine
JS engine
call stack
exec
Event loop
constantly monitor CQ if empty then
callback queue
CQ
microtask queue
setTimeout()
DOM APIs
fetch()
localStorage
console
location
and many more...
promises
callback
mutation observer
more priority
setTimeout
document.
<h1>Time</h1>

Server
memory
Element
Console
hello world
geo web APIs
eventListener, setTimeout.

Date:

```
1  console.log("start");
2
3  setTimeout(function cb() {
4      console.log("callback");
5  }, 5000);
6
7  console.log("End");
```

⇒ How this program will execute.

⇒ first line it will log start in console.

⇒ the line 3 there is setTimout which means it will access browser setTimeout API and assign this cb function to it and timer will also start 5000

⇒ then line 7 console.log("End") it will log the end in console

⇒ After some time our timer will expire and our program is ready to execute but it will not direct pop-up in call stack and execute. instead it will go in callback queue

IMPORTANT NOTES

first then event loop will check callback queue if there is cb function ready to execute. So it will load in call stack that function. if call stack empty

⇒ then again execution context will be created. and function cb has console.log "timer" so it will print in our console

⇒ So this is how program will execute in asynchronus way.

⇒ So OP ⇒ Start
End
callback.

* what is event loop and what will eventloop do.

⇒ eventloop continously monitor the call stack empty or not. if its empty then it will first prioritize micro task queue

Date:

and then callback queue task so it will schedule the task and one after another execute in call stack.

* what is micro task queue in which function will execute in it

⇒

⇒ microtask queue have more priority then the callback queue.

⇒ a callback function which are returned from promise and mutation observer this function will scheduled in microtask queue

⇒ a function from settimeout and eventlister will be queued in callback queue.

⇒ now eventloop will check if call stack is empty then first it will execute micro task and after that callback queue task in call stack this is how it work.

Date:

* what is starvation problem in ~~an~~ in JS.

⇒ starvation means microtask queue program execute but what if microtask has more and deep Microtask and I can't back task queue is ready to execute but the microtask is more priority it may be not run due to microtask program is still running this is called

* EP-16 JS engine.

⇒ To run JS code, any where we need javascript Runtime environment ↑ It's JS engine

* What is JRE It's capable to run JS code and also use browser WebAPI and also run in our local machine outside of the browser through node.JS

* there lot of JS engine available in

Date:

diff - diff browser.

like:- mozila has spider monkey JS engine chrome has V8 in browser and node.JS and deno.

⇒ we can make our own JS engine through following ECMA Script standard.

⇒ the first JS engine by the creator of JS Brenden eich which name was spider monkey JS engine.

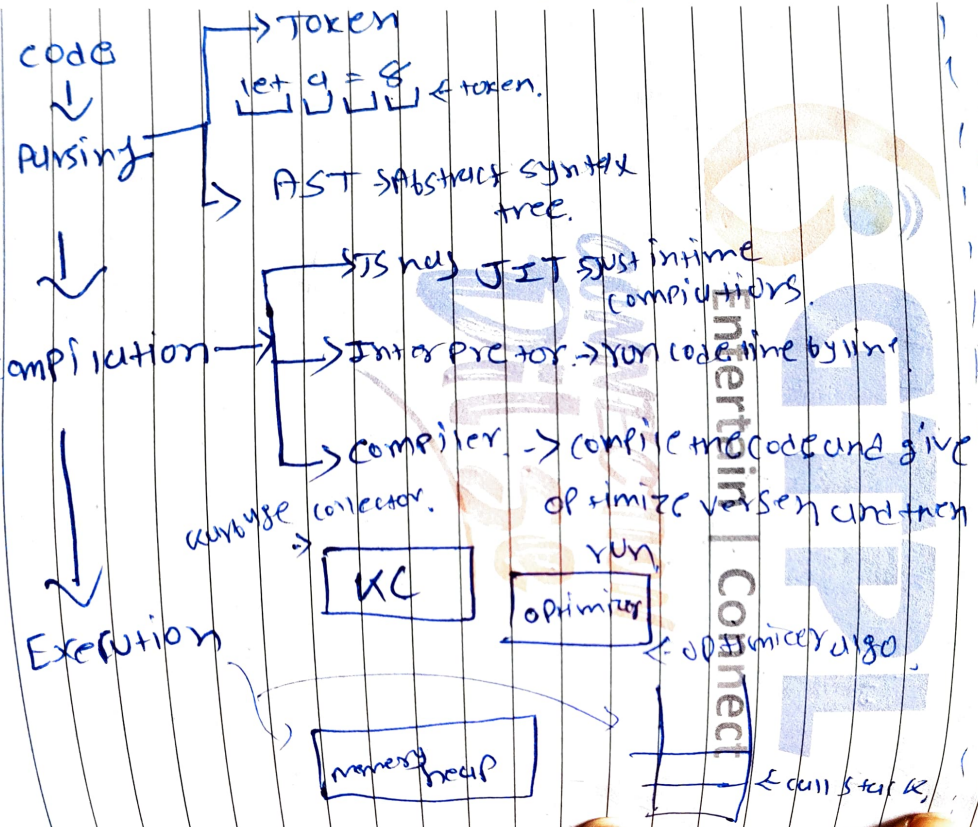
⇒ JS engine Architecture.

* JS Runtime environment is not a machine

⇒ It's also a program which will execute our code. (low+anjuage)

⇒ different browser have diff JS engine to run the code.

JS engine.



google V8 engine is the fastest JS engine.

→ It has mark & sweep gc algorithm.

→ what is gc, it will remove unused function and variable in memory.

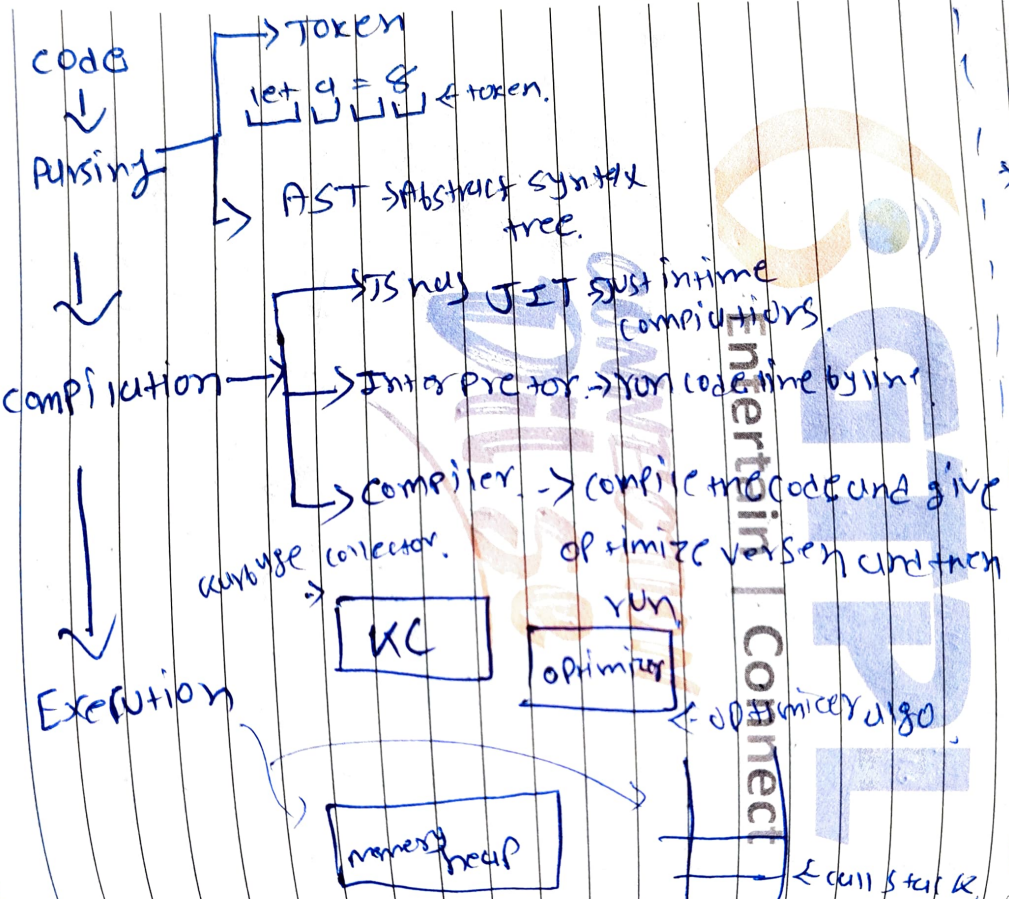
→ what is optimize algo.

→ there are lot's of optimize algo. to optimize the code.

→ inlining, copy elision

→ inline caching.

JS engine.



google V8 engine is the fastest JS engine.

→ It has mark & sweep gc algorithm.

→ what is gc; will remove unused function and variable in memory

→ what is optimize algo.

→ there are lot's of optimize algo. to optimize the code.

→ inlining, copy elision

→ inline caching.

Date:

Date:

JS engine execution.

1. code will parse in token in small small piece then AST will be created. abstract syntax tree for that token.

2. then it will go in compilation phase.

3. in program language there are two type of compilation.
   (1) interpreter
   (2) compiler.

(1) interpreter :- will execute code line by line without bother so that will below code and all.

→ It's simple execute code line by line. that's why it's faster.

(2) compiler :- compiler compile the code and make it's more efficient and optimize version. → it's slow compare to interpreter but efficient.

(3) JS use JIT → just in time compiler

Date:

> In short both compiler and interpreter to run the code as much as fast.

> google v8 engine interpreter name is ignition and compiler name is turbo fan.

*. concurrency model in JS, settimeout trust issue. cp - 17.

*. setTimeOut not guarenty that your code will execute after given time expiration.

> It's assure it can run that code only after timer time.

*ex:-

```
console.log('1')

function cb() {
    console.log("callback")
}

setTimeout(cb, 5000)
console.log('end')
```

Date:

=>OP -> 4
end
callback.

But what will happen if our code has more line of code to execute and It may take time to execute 20 sec.

=> during this time our set timeout timer expires and scheduled in callback queue to execution callstack.

=> but our callstack is not empty due to current code running.

=> this problem called Concurrency model in JS.

ex:- console.log("start")

function cb(){
console.log("callback")
}

setTimeout(cb, 5000)

=> Suppose million line of code here

Date:

=) op -> Start
million line of code of
callback.

Q:->

```
console.log("start")

function (b){
  console.log("callback")
}
console.log("Interval")

setTimeout(cb, 0)

console.log("End")
```

op->. Start
Interval
End
callback.

=) here callback in the end even after
time is 0 sec but after sec executed
in callstack. It will come from callback
queue suddenly we are getting output.

Date:

*. ep-18. Highe order functions.

⇒. a function which takes function as an argument or return another function called the highe-order functions.

or!-->

```
function a() {
    console.log("a")
}

function b(a) {
    a()
}
```

⇒ in above example function b is is a higher order function and function a is callback function

⇒ we should write our code in modular and reuse.

⇒ we should follow a DRY principle. do not repeat your self.

Date:

EP-19. map, filter & reduce

1. map

=> map is a higher order function which will map ~~on~~ ~~on~~ ~~elem~~ each and every element of an array and return the new array.

=> ex:- suppose const arr = [2, 5, 4, 8]

Now we want to double value of this array element then (or) ?

```
cons double = (x) => {
    return x*2
};

const output = arr.map(double)
                  ↓ higher order function   ↓ callback function
console.log(output);
```

op -> [4, 10, 8, 16]

We can write this also.

```
const output = arr.map((x) => x*2)
```

Date:

⇒ In short map a higher order function will run our own logic function for each and every element of given array and transfer the value accordingly to it and will return new array with the transformable value.

4. filter.

⇒ filter is also a higher order function which will filter out the value in given array based on our logic and return the new array.

⇒ ex:- suppose we want even number values from this array

```
const NUM = [1, 2, 3, 4, 5, 6]

const output = NUM.filter((x) => x % 2 == 0)

Console.log(OUTPUT)
```

OP => [2, 4, 6]

Date:

* Reduce.

- Reduce is also higher order function which will iterate through elements of an array and return only one value.

=> we can use it if we want sum of the given array.
=> want to find max in given array.

=> it takes two arguments
(1) callback function, (2) initial value.

Q:- find y max in given array.

```
const num = [2, 4, 6, 8, 10]

const findMax = (x) => {
    const max = 0.

    for (let i=0, i < x.length, i++) {

        if (x[i] > max) {
            max = x[i]
        }
    return max.
```

Date:

console.log("max", findmax(num));

-> op -> 10.

* Now using reduce function.

(callback function)

const output = num.Reduce(function(acc, curr){

if (curr > acc){

acc = curr

}

return acc

}, 0); <- here initial value.

=> console.log("max", output)

op -> 10.

* here current means current value and accumulator is the

=>. curr = x[i] and acc = max

Date:

* another example

* sum of the number.

const num = [2, 4, 6, 2]

* using function.

⇒ function findsum (num) {

let sum = 0

for (let i = 0; i < num.length; i++) {

sum = num[i] + sum

}

return sum

}

console.log ("sum", findsum (num)

⇒ op -> 14.

Date:

* Now using reduce

const num = [2, 4, 6, 2].

=> const OUTPUT = num.Reduce(function(acc, curr){

acc = acc + curr

return acc

}, 0)

↑ second argument initial value of Accumulator

=> console.log(OUTPUT)

OP -> 14.

IMPORTANT NOTES

7/06/25

* Namaste JavaScript S2

Callback Hell EP.1

- Callback is very useful asynronous programming using callback we can use asynronous programming in JS.

- Callback hell one issue with callback

- when our program grow horizontaly instead vertical

- what it means ⇒ when our program depend nested callback i.e one callback function depend other and other callback function depend on other callback this program called callback hell also called pyramid of doom.

ex:- function createOrder(cart, cb) {

cb(cart) {

}

}

Date:

* other problem we faced

* inversion of control.

* when we attaching callback function to some function.

Ex:- function createOrder(cart, cb) {

cb() {

}

}

=> in above code we bind trust that createOrder will execute our code at some time.

=> but what if createOrder function have some bug and will never execute so we bind trust that our function execute but it will not in this scenario this is called inversion of control problem with callback function.

Date:

* ep-9. promises.

* As you showed earlier there is problem with callback hell and inversion of control in async program.

⇒ to overcome this problem we can use promises.

* what is promises ⇒ promise is object which represents the eventual completion or reject of async Program.

⇒ Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

⇒, as you know in inversion of control problem we need no control on our cb function we are passing our cb function in it.

⇒ but using promise we are not passing our cb function we are attaching to it according to our

Date:

result it's presented that it will execute and once only.

ex:-

here we are passing cart & cb.

function createOrder(cart, proceedToPayment) {

    proceedToPayment() {

        ....

    }

}

* now using promises

const promise = createOrder(cart)

promise object. state → pending, fulfilled, reject.

promise.then(proceedToPayment() {

})

here we are attaching cb function.

* it will run only after promise fulfilled.

Date:

as you know our ~~for~~ using call back nested we can face call back hell problem.

=> but we can overcome this problem using promise chaining.

ex:-

const order = createOrder()

↑

suppose Promise object

order.then(createBill).then(Show summary).then(Proceed to Payment).then(success).

Using promise chaining we can avoid call back hell problem.

=> Promise give control to developer to handle the program we can control over cb function, which to run or not.

=> promises given guaranty and trust ~~transaction~~ our whole transaction.

Date:

* It's immutable.

* EP-3, Creating promise, chaining & Error handling.

⇒ creating promise.

~~const creating promise = new Promise(function~~

const promise = new Promise(function(resolve, reject){

↑
resolve, reject function
by JS.

});

↑ producer code.

⇒ consuming code.

promise.then(createOrder(){}).then(orderSummary(){}). ← this is called promise chaining.

⇒ We can handle our error by .catch in promises.

Date:

S2 ES * Promise API

=>. Promise.all() => It is used to handle multiple promises together, eg:- multiple API call

=> It will take array of promises as an input

Example. -> Success case.

=>. Promise.all([P1, P2, P3])

times it takes -> P1: 3s, P2: 1s, P3: 2s

OP -> after 3s [val1, val2, val3]

wait for all of them to finish.

=> in above example promise.all will take multiple promise as argument in array.

=> then it will run all promise together and wait for all of them to finish.

=> after 3s will return result of Promises in new array.

Date:

*. What if any of promise is rejected.

⇒. Ex:-

promise.all ([P1, P2, P3])

↓ ↓ ↓

3S 1S 2S

↓

(X) ← suppose failed.

⇒ So op ⇒ will be immediately (ERROR) it will return Error only It will not wait for other promises to fulfilled

So in short in promise.all if all promised success it will return the success result if any of fail it will return Error.

⇒ now how is the scenario what if i want result from successfull promise weather one or any other promise failed but still i want my successfull promise data we can use do them using.

Date:

* promise.allSettled()

⇒ promise.allSettled([P1, P2, P3])

P1 ↓ 3s ✓
P2 ↓ 1s ✗
P3 ↓ 2s ✓

⇒ It will wait for all the promises to settled then it will return the result of the array either success or failer.

so op → [value1, Error, value3]
(After 3s)

. promise.race()

⇒ It's race whatever Promise settled first it will return the value of that settled promise. not an array.

Ex:- promise.race([P1, P2, P3])

P1 ↓ 3s
P2 ↓ 2s
P3 ↓ 1s

⇒ op will be P3 value ⇒ value 3
(not an array)

Date:

* what if some promise failed.

=> so it will return the error if it's take less than time from others.

=> in short promise.race() will return the result of first promise settled weather it's succese or fail, which promise taking shortest time to complete.

=> promise.any([P1, P2, P3])

=> It will wait for first success and it will return that result value. and ignore the failure.

* example.

Promise.any([P1, P2, P3])

	P1	P2	P3
time →	3s	1s	2s
status →	✓	✗	✓

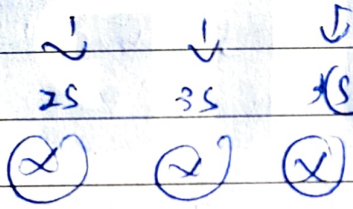
=> so it will return the P3 value because it's

Date:

P2 failed then it will ignore P2 and after that P3 success earlier then it will return P3 value.

⇒ but what if after all promise failed.

promise.any([P1, P2, P3])



→ the O/P will be aggregated error.

→ [err1, err2, err3]

* ~~Promise.all example.~~

Date:

* EP-4. async - await.

=> what is async.

=> async is a keyword given by JS to make async function.

=> await is also keyword given by JS await can be used in any async function.

=> async - await used to handle promises in synchronous way.

=> it's always return promise

=> eg:-

```
const promise = new promise (resolve, reject) => {
setTimeout (() => {
        resolve ("Promise Resolved")
    }, 1000);
function getdata () {
promise.then ((res) => console.log (res));
console.log ("It will Print")
}
getdata()
```

Date:

=> OP => It will print -> immediately and after 10 sec -> promise resolved.

* because js wait for none. It will print immediately and wait for none.

* now using async - await this program.

```
const promise = new promise((resolve, reject) =>
    setTimeout(() => resolve("promise resolved"), 10000)

function getdata() {
    [const data = await promise
    (console.log("It will wait")
    (console.log(data).
}
getdata()
```

OP => Promise resolved after 10 sec and also It will wait will log in console.

Date:

but JS is wait for none then now it's possible.

⇒ now now this program will execute.

⇒ variable promise will be undefined first then.

⇒ function getData will save as it is.

⇒ then getData invocation new EC will be created. in call stack. at that first data variable will be undefined after that there is promise so it will register promise and it's also check there is await key word.

⇒ so it will wait to promise fullfilled. in technical term JS will not wait It's suspended the getData() function in call stack till promise resolved.

⇒ if promise resolved it will pop-up again in callstack

Date:

and OP will => It will wait Promise resolved -> both after 10sec.

=> this is now async await works.

=> basically async await is syntactically sugar of Promise.then method.

=> We can handle error in async await using try catch block.