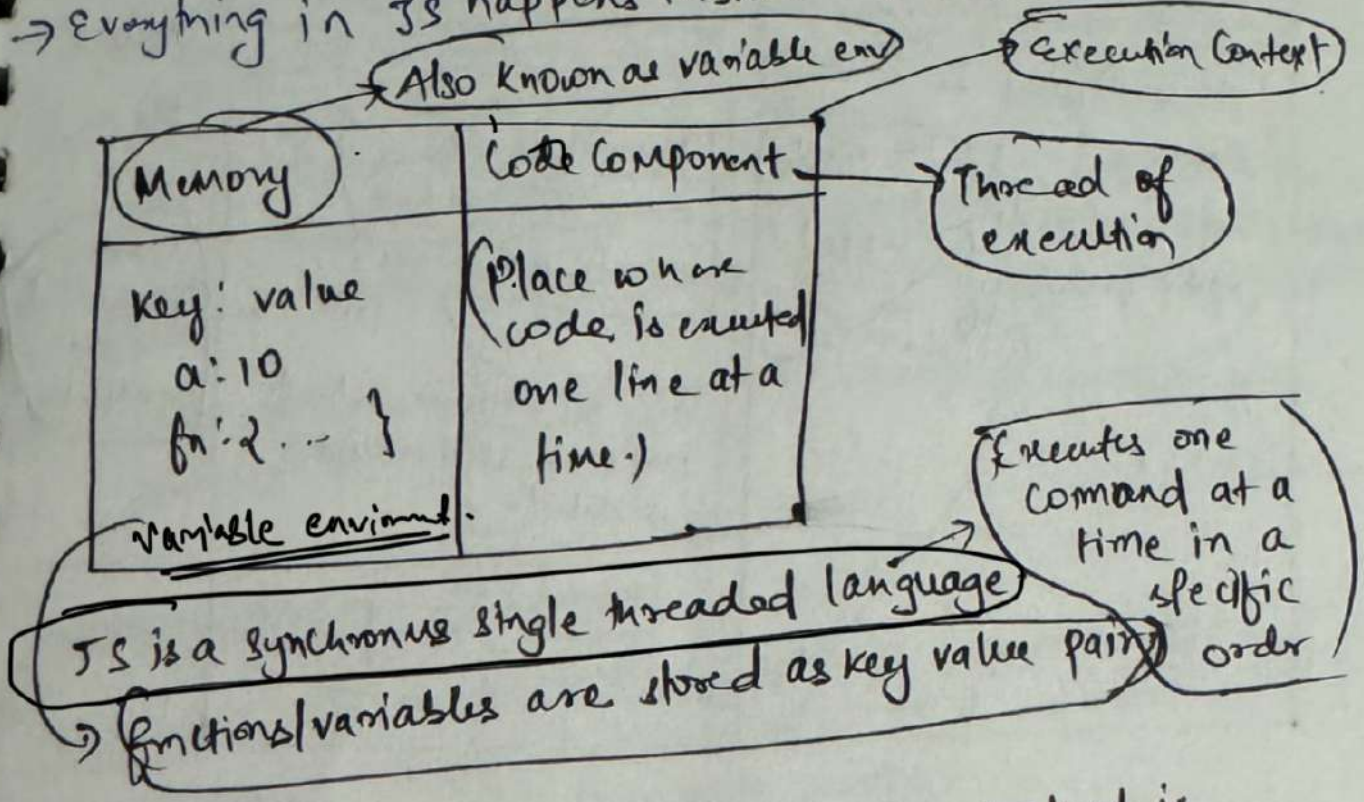
Day 1:

Namaste JavaScript

How JS Works ?

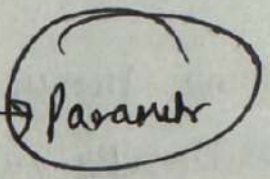
→ Everything in JS happens inside an Execution Context.



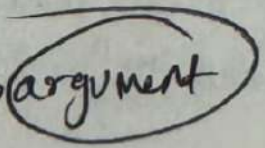
→ When we run a JS code, an execution context is created.

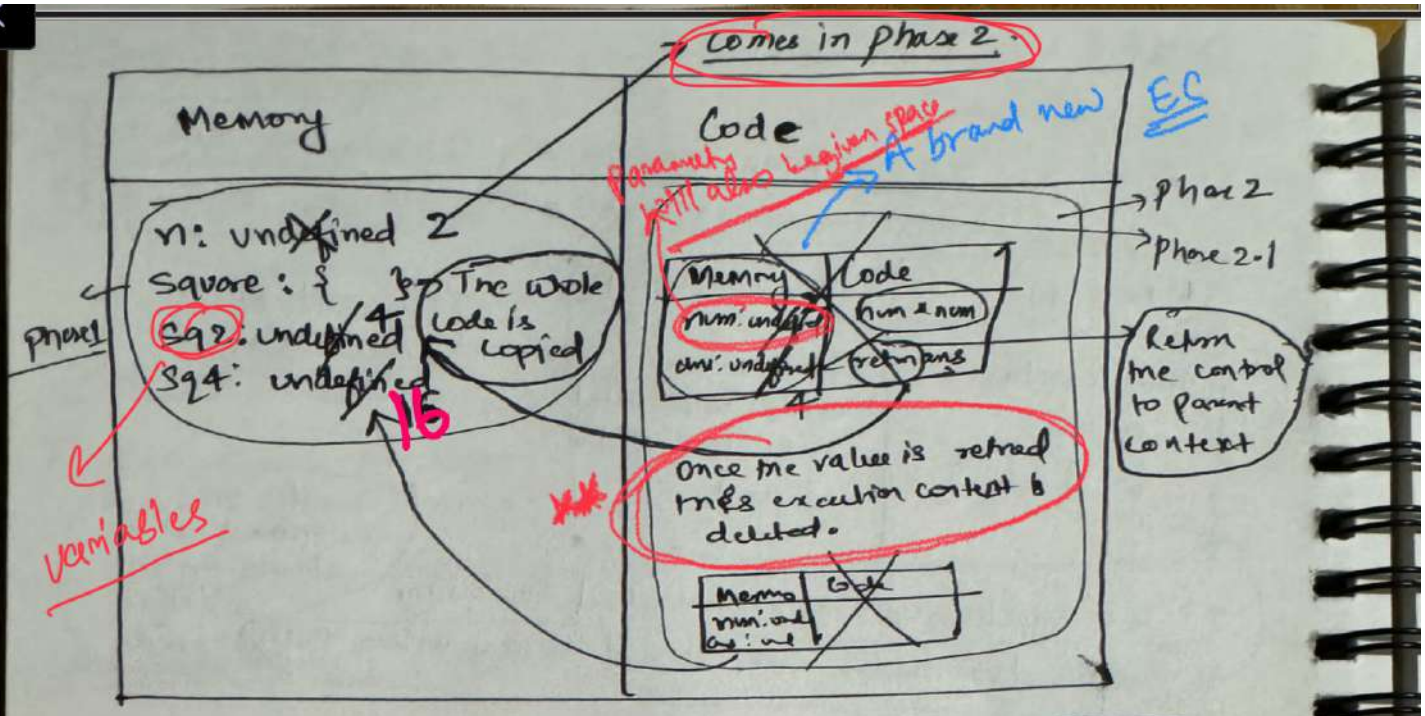
Only go to next line when previous one is executed.

Code:-




```
var n = 2;
function square(num) {
    var ans = num * num;
    return ans
}
var sq2 = square(n);
var sq4 = square(4);
```





1st Phase :- Memory creation phase → JS allocates memory to all variables & functions.

2nd Phase :- Code execution phase.

*** In the 1st phase JS skims through the whole program and allocates variables and functions in the memory block.

* Function is stored as it is

[For variables it stores "undefined" as value and for functions it stores the whole code for that function]

2nd Phase :- Code execution phase

Once again run through whole code line by line.

1st line, var n = 2

so now value of n changes from undefined to 2.

Next it reaches to next line where it encontes a function. It skips it and move to the new line. In that line function invocation is triggered.

```
var sq2= square(n);
```

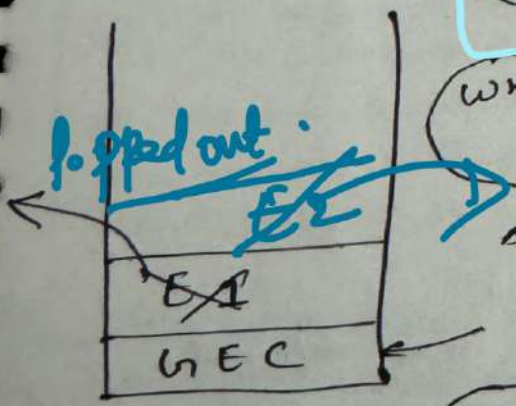
When a function is invoked then a new execution context is created and only that function is taken into consideration.

Once the value is returned then that execution context is deleted.

Important

Call Stack.

(When any JS code is run, this call stack is populated with a Global execution context.)



Whenever a function is invoked which creates a new execution context, it is put inside the stack.

After that function returns anything that execution context is popped out of Call stack.

After the code is done the GEC is popped out of Call stack.

→ Call stack is a Stack.
→ Initialised with a Global execution Context.
→ When a function is invoked, a new execution context is put inside the Stack.

Stack.

3. Hoisting in JS → Phenomenon where we can access function and variables without defining it.

```
var x = 7;
function getName() {
    console.log("Hi");
}
getName();
console.log(x);
```

Output:
Hi
7

What if :-

```
getName()
console.log(x);
var x = 7;
function getName(){
    console.log("Hi")
}
```

→ This is an error in other programming language.

→ In memory allocation phase - it got the value "undefined"

→ Hi
undefined

→ at the time of defining, function was stored fully and then it got executed.

What if :-

```
getName();
console.log(x);
function getName() {
    console.log("Hi");
}
```

→ Hi
x is not defined
(Uncaught Reference Error)

**

(Undefined and not defined are not same.)

Hoisting is a phenomenon in JS where we can access functions & variables without defining it.

```
Var x = 7;
function getName() {
        console.log("Hi");
}
conside.log(getName);
```

Output :

```
f getName(){
    console.log("Hi");
}
```

⇒ Prints the whole function

If instead of logging after defining, we try to log the function before initialising it, we get the same output, i.e, the whole function is logged. In case of variables if we do the same it logs undefined.

The reason is: In execution phase variables store undefined and functions store the whole function.

(Even before the code starts executing, memory is assigned to each variable & function in phase 1.

So that's why x gets "undefined" and the function stores the whole function. → [actual copy of the function]

Concept:

Read this :

So yaad ye rakha naiki Jab V-code diya Rahega, sabse Pehle variables / functions ke liye memory assign hoga, then uske baad dobara code starting se run hoga or values assign / execute hoga.

Phase 2

Phase 1

console.log(x) → undefined

console.log(getName) → function copied

var x = 7 → Ye store hoga

function getName () { → Ye store hoga

}

Not initialised and we try to access it

(not defined vs undefined).

No variable / function is nhi hai code m

x is not present in GEC

It is there in the code but yet not executed

x is present but not been executed yet

```
getName()
console.log(x)
console.log(getName);
var x = 7
var getName = () => {
}
```

output :-

(getName is not a function)

Arrow function makes a function as variable

anonymous function (behaves like a variable)

** So what happens here is, JS stores an arrow function as a variable and not as a function hence it stores undefined inside of it in the memory allocation phase. So when in the code execution phase it sees to invoke the function it gives not a error.

var getName2 = function() { } → It still behaves like a variable.

Functions → Function invocation behind the scene.

```
var x = 1;
a(); b();
console.log(x)
function a() {
    var x = 10;
    console.log(x);
}
function b() {
    var x = 100;
    console.log(x);
}
```

Output :-
10
100
1

A Global execution context is created. It will have 2 components. Memory & Code component.

Memory | Code

Output
10
100
1

Callstack

x: undefined → 1
a: f{ ... }
b: f{ ... }

var x = 1

Memory | Code
x: undefined → 10 | x = 10; console.log(x)

separate variable

Memory | Code
x: undefined → 100 | x = 100

JS will look for this in local memory

** [x in a()'s context is independent of the same in GEC or b()'s context and vice versa.]

(Shortest JS program)

this Keyword

* Empty JS file when executed still creates a global execution context.
** Also creates a window (Check by typing "window" command in console)
↓
object

(Global object + Global Execution + this variable) → functionality given by this keyword

~~Wherever JS~~

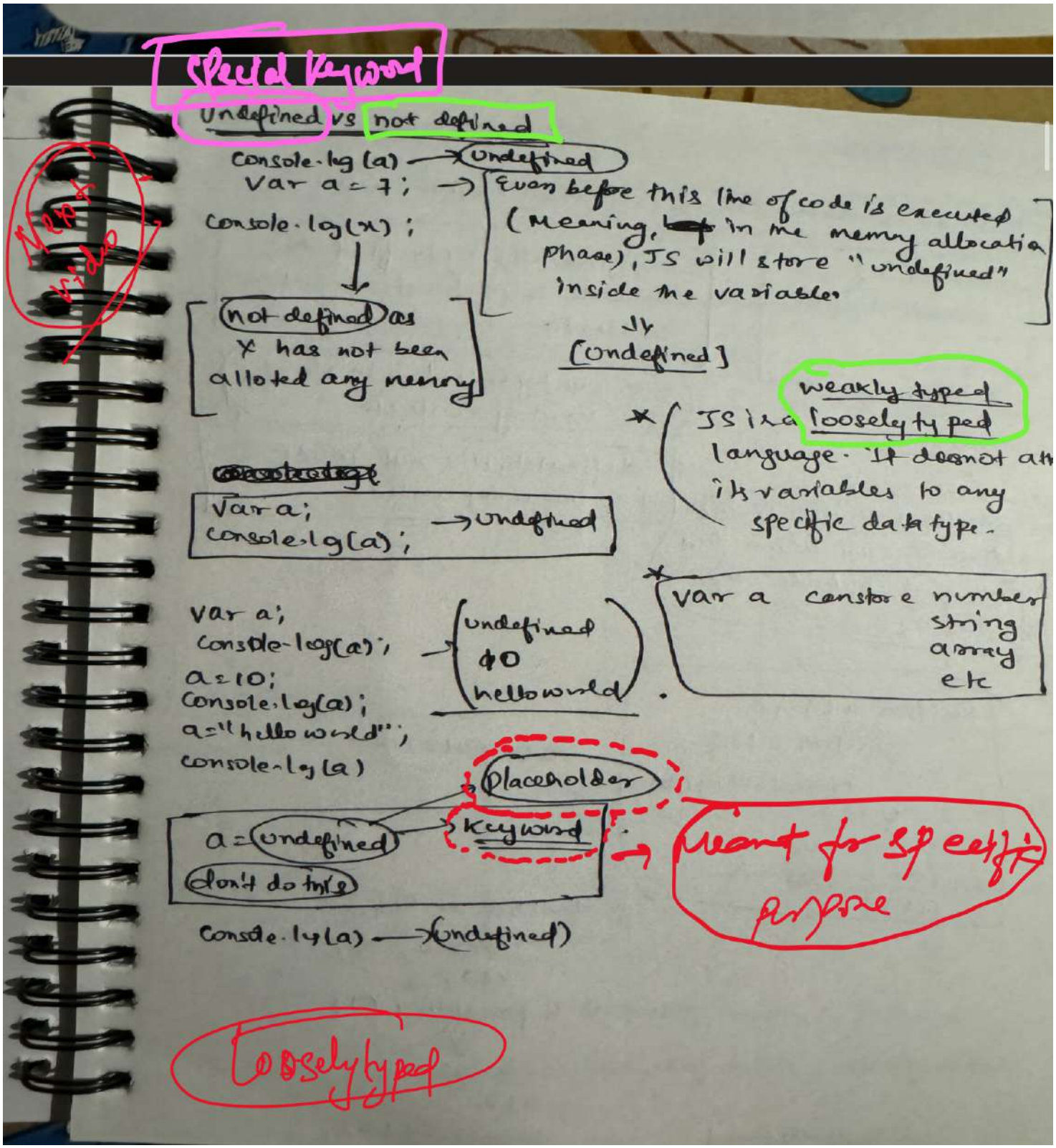
* (Wherever JS is running, it must have a JS engine)
[Chrome - V8 engine]

[(this === window) → at the global level] → Very important

* [Global space → Any code in JS which is not inside a function]
***

```
var a = 10;          → Global space.
function b() {       → Global space
    var x = 10;      → space of b
}
console.log(window.a) → 10
```

Console.log (a) → 10
Console.log (x) → not defined (not undefined)
Console.log (this.a) → 10


Special Keyword
Undefined vs not defined
Console.log(a) → undefined
Var a = 7;
Console.log(x);
not defined as x has not been alloted any memory
Even before this line of code is executed (Meaning, in the memory allocation phase), JS will store "undefined" inside the variables
[undefined]
weakly typed
JS is a loosely typed language. It doesnot attach its variables to any specific data type.
Var a;
Console.log(a); → undefined
Var a;
Console.log(a); → undefined, 40, helloworld
a=10;
Console.log(a);
a="helloworld";
Console.log(a)
Var a constore number string array etc
placeholder
keyword
a = undefined
don't do this
Meant for specific purpose
Console.log(a) → undefined
loosely typed

EP7 · Scope chain

→ Scope in JS is related directly to lexical environment

```
function a() {
    console.log(b);
}
var b = 10;
a();
```

→ output = 10

1. JS engine will try to find the variable b in local execution context.

JS will try to find b inside of a's memory context.

definately it's not inside of a's memory context

So somehow b inside the function a can acan the value of b outside the function a.

Children can access parn'ts scope variables

→ output = 10

Important

```
function a() {
    c()
    function c() {
        console.log(b);
    }
}
var b = 10;
a();
```

It can access it too.

```
function a() {
    var b = 10;
    c();
    function c() {
    }
}
a();
console.log(b);
```

not defined ←

~~Scope of b → where can I access a variable or~~

Scope means where we can access a specific variable or function in our code.

Scope of b → where can I access the variable b.

Is b inside the scope of a function → Can I accn b inside a

scope is directly dependent on the lexical environment.

```
function a(){
    var b=10;
    c();
    function c(){
    }
}
a();
console.log(b);
```

GEC

Memory	Code
a: fn{}	

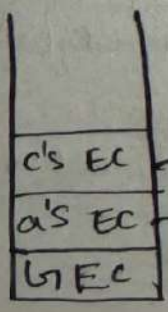
A

Memory	Code
b: 10 undefined c: {}	

C

M	code

Callstack.



* Lexical Env ⇒ local memory + lexical Env of its parent

hierarchy/sequence

* c function is lexically inside "a" function.

* "a" function is lexically inside the main function.

Whenever a context is created a lexical env is also created and we get a reference to the lexical env of its Parent.

LOGIC :- [Every execution context stores the reference to its lexical parent context in the memory block.]

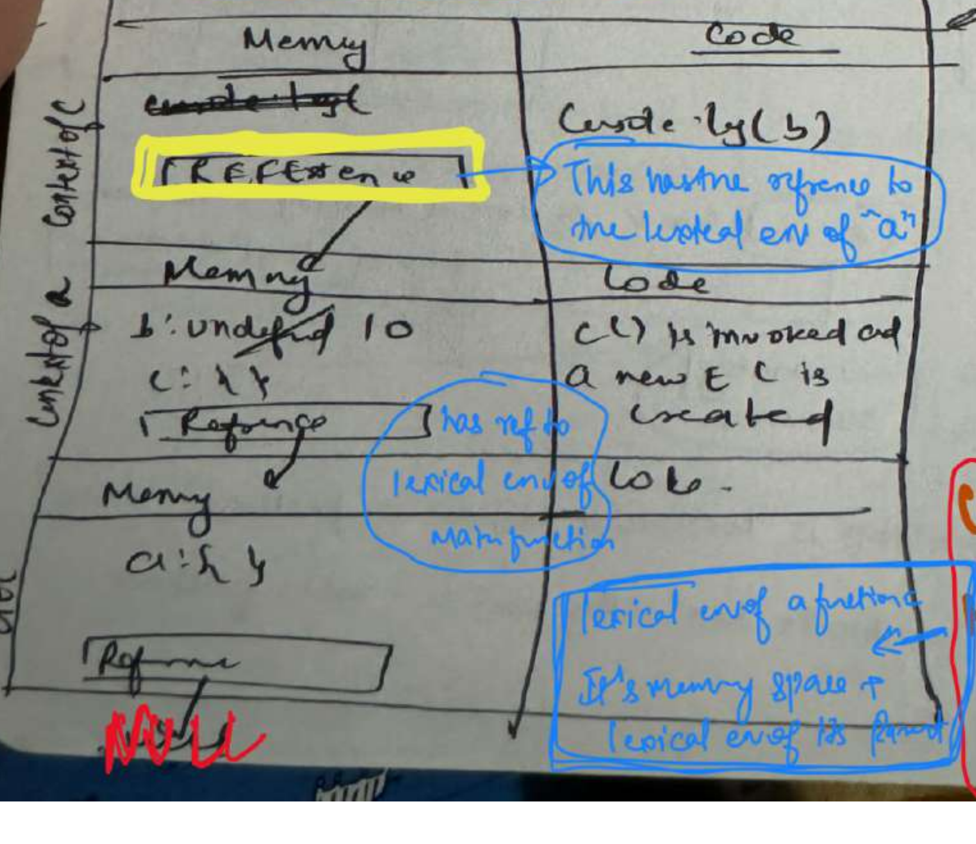
lexical parent of c is a

lexical parent of a is GEC

[lexical enviorment of a is its memory space and its parent's lexical enviorment]

GEC's lexical pant refence is NULL.

Call stack



Code

```
function a() {
    var b = 10;
    c()
    function c() {
      console.log(b);
    }
}
a();
console.log(b);
```

lexical env of a = Memory space + lexical env of Main function

Logic:- How it happens is, if inside the scope of c we want to print it's value. If b is not present inside c, then it will look for it in it's lexical parent which is a. It searches for b in lexical env of a and finds that b is 10. So now it prints the value 10.

Suppose if b is not in lexical enviornment of a, so it will reference to lexical enviornment of a's parent which is global execution context o

And similarly it reaches to NULL if b was not present in global also.

* This way is called scope chain → Nice Explanation ✓

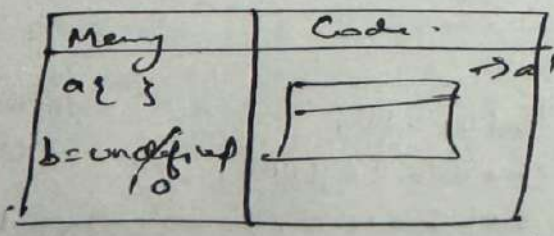
So we can say that (created when a EL is created)

** lexical env = local memory + lexical env of parent

Scope in JS is directly related to lexical enviornment.

```
function a() {
    console.log(b);
}
var b = 10;
a();
```

→ 10




```
function a() {
    b();
    function c() {
        console.log(b);
    }
}
var b = 10;
a()
```

→ 10

8) Temporal Deadzone ?
Are let & const declaration hoisted ?
Syntax / Refference / Type Error.

Hoisting is a phenomenon where we can access variables & functions even before defining it.

→ let & const declarations are hoisted but

They are in the temporal deadzone for the time being.

↳ phenomenon in JS where we can access a variable / function without defining it.

```
console.log(b); ⟶ undefined (Not an error)
var b=100;
```

We can access b even before we had defined it (var can be hoisted)

```
console.log(a) → Cannot access a before initialisation.
console.log(b) → undefined
let a=10;
var b=100;
```

I can only access after initialisation

[JS allocated memory to a]

```
let a=10;
console.log(a);
var b=100; ⟶ Stored in global space and the value is undefined.
```

Stored in script and the value is undefined.

In case of let and const are stored in different memory space and we cannot access them

The var was stored in global memory space but 'a' was stored in script memory space and we cannot access that memory space before putting values in them. So it is a case of hoisting ⇒ as undefined was stored initially

Temporal deadzone :- (Time since when this let variable was hoisted and till it is initialised with some value.)

So initialised hone undefined nojaaega but jab tak value nahi milega tab tak wo TDZ mein rahega

```
console.log(a);
let a=10;
var b=100;
```

Whenever we try to access a variable in temporal deadzone it gives a reference error ⇒ as it will not be

Script
a : undefined
So until it is not initialized with any value, it will be in temporal deadzone

Important inside of global object.

{ we can't do window.a
but we can do window.b }

let is stricter than var.

```
let a=10;
let a=100;
```
→ syntax error :- (Identifier 'a' has already been declared.)

Not a single line of code is run.

```
let a=10;
var a=100;
```

Const → Even more strict than let

```
let a=10;
const b=1000;
a=10;
console.log(a);
```

Script
a: undefined
b: undefined

```
let a;
const b=1000;
a=10;
console.log(a)
```

→ fine
10

We cannot do something like

```
Const b;
b=1000;
```

Syntax error:
Missing initializer in const declaration

(It expects initialisation at that point itself.)

```
let a=1000;
const b=10000;
b=1;
a=10;
console.log(a)
```

→ Type error as it is a const variable and we can not change its value.

```
let a=1000;
let a=100;
```

Syntax Error:
'a' has already been declared.

```
Const b;
```

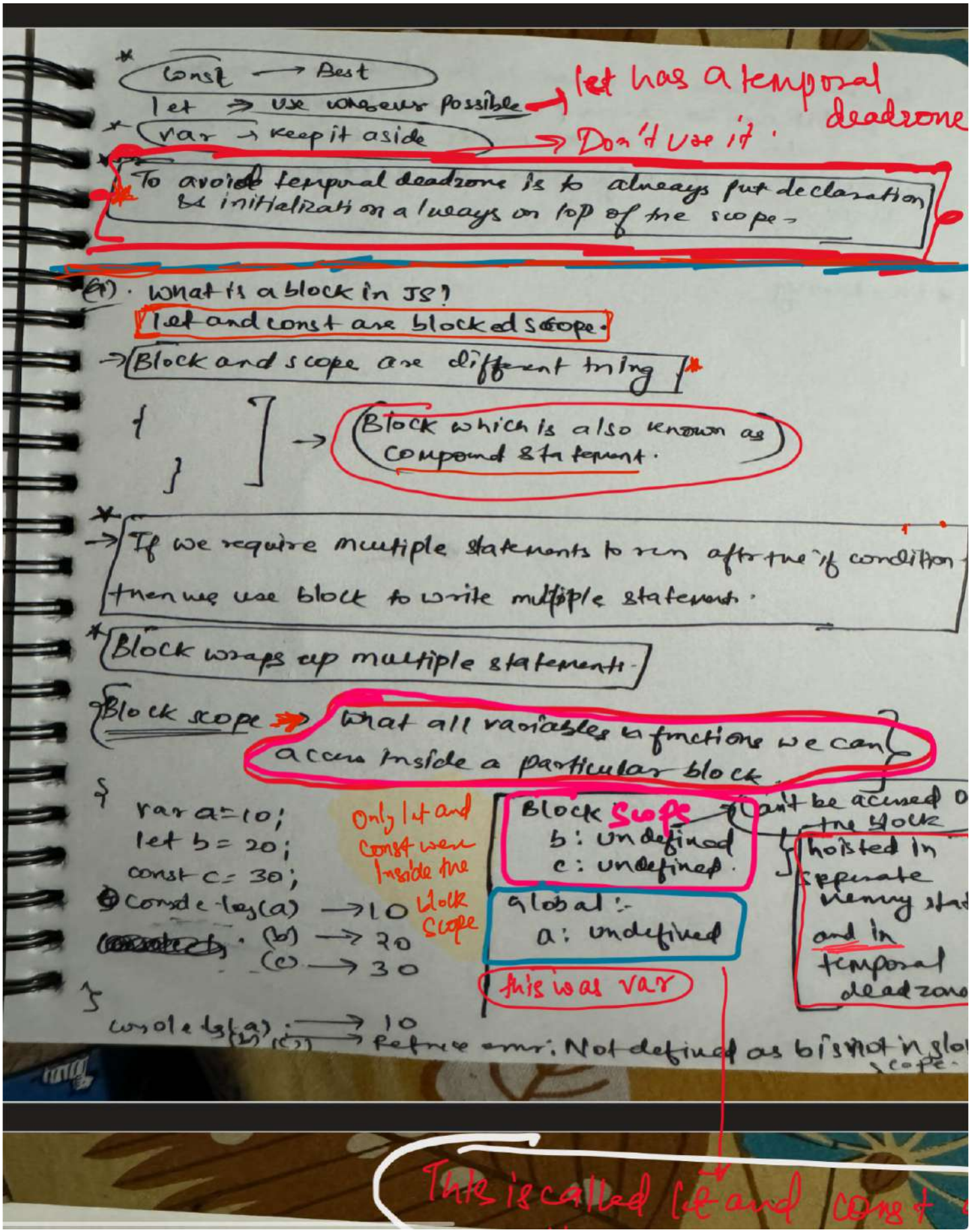
→ Missing initializer in const declaration

Refrance Error: Tries to find a variable inside a memory space and cannot access it

```
Const c.log(a)
let a=100;
console.log(a)
```

→ Refrance Error
as a is in temporal deadzone.

→ not defined refrance Error.


const → Best
let ⇒ use whenever possible
var → keep it aside
let has a temporal deadzone
Don't use it.
To avoid temporal deadzone is to always put declaration & initialization always on top of the scope.
(Q). What is a block in JS?
let and const are blocked scoped.
Block and scope are different thing
{ }
Block which is also known as compound statement.
If we require multiple statements to run after the if condition then we use block to write multiple statement.
Block wraps up multiple statements.
Block scope ⇒ what all variables & fnctions we can access inside a particular block.
var a=10;
let b=20;
const c=30;
console.log(a) → 10
(b) → 20
(c) → 30
Only let and const were inside the block scope
Block scope
b: undefined
c: undefined.
global:-
a: undefined
this was var
can't be accessed o the block
hoisted in separate memory sta and in temporal dead zon
console.log(a) → 10
(b) (c) → Refrence error: Not defined as b is not in glo scope.
This is called let and const

This is called let and const
block scoped

So, when the code is run, b and c is in block scope environment and it can be accessed inside the block.
We can also access the 'a' inside the block.
But outside we can only access 'a' as it is in global scope.

* shadowing

```
{
  var a = 10
  let b = 20
  const c = 30
  console.log(a)
  (b)
  (c)
}

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

10
20
30
10
not def
(Refer
erro

Hoisting means using variables even before it was declared.

2022 February

(Shadowing in JS) :-

Saturday 12
Day (043/322)

They were pointing to same memory space in GEC

It will shadow initial initialization

```
var a = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(a);
}
console.log(a);
```

output:
10
10

→ Shadowed and modified the value of (a).

```
let b = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;

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

→ Script (BAV)
→ Global Scope (BAV)
→ Block Scope (BAV)

→ 10
20
30
100

Reason:

Scope
- Block
  - b: 20
  - c: 30
- Script
  - b: 100
- Global
  - a: 10

Sunday 13

Shadowing is for block only

Here it shadowed the initial initialization

This will get the original value of b

similar with const.

MAR
APR
MAY
JUN

February 2022

Mo	Tu	We	Th	Fr	Sa	Su
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28						

March 2022

Mo	Tu	We	Th	Fr	Sa	Su
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Work to do

However we can shadow var with let

Illegal shadowing

Cannot shadow let using var

February 2022

14 Monday

Shadowing behaves in a similar way in functions as well.

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

→ Identifier a has already been declared.

Illegal shadowing

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

→ (legal shadowing)

Episode 10

& [(Closures) in JavaScript].

```
const c = 100;
function x() {
  const c = 30;
  console.log(c);
}
```

→ 30 (shadow).

We judge ourselves by what we feel capable of doing, while others judge us by what we have already done.

2022 **CLOSURES** . Class-10:

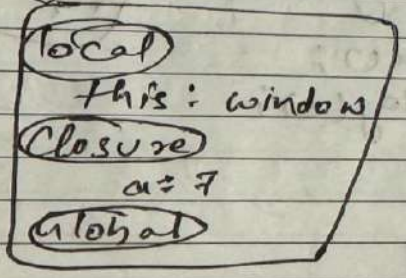
February

Tuesday 15
Day (046/319)

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

→ debugger at this line

* (This is what closure is)



→ Output: 7.

This is Closure

Function [redacted] bind together with its lexical [redacted] environment

* Function y was bind with the variable of x and it forms a closure.

* function with its lexical enviornment

MAR APR MAY JUN

Wk	February						2022
	Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27
	28						

Wk	March						2022
	Mo	Tu	We	Th	Fr	Sa	Su
10		1	2	3	4	5	6
11	7	8	9	10	11	12	13
12	14	15	16	17	18	19	20
13	21	22	23	24	25	26	27
14	28	29	30	31			

Work to do

February 2022

16 Wednesday
Day (047/318)

Module Design Pattern
Currying in JS
function like once
memoize
maintaining state in async world
Set Timeout
Operators

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

a:100

Output :-

```
f y() {
  console.log(a);
}
```

z ⟶ Because JS has remembered its closure when z is called. 100

So here, when z was consoled log, the existence of x was removed from call stack at it was executed but still due to closure, JS remembered the reference of a and hence when invoked printed value of a.

{ Here existence of x was not in call stack }

Closure is function bound together with its lexical environment.

One should always play fairly when one has the winning cards.

Work to do

2022

February

(Practical).

Thursday
Day (048/317)
17

If we do not write any code in JS we can still acces the "window" command.

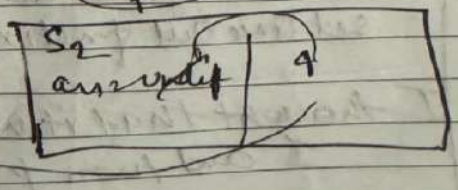
```
var n = 2;
function square(num) {
    var ans = num * num;
    return ans;
}
var s2 = square(2);
var s4 = square(4);
console.log(s2);
console.log(s4);
```

(Global execution context):-

Step 1

n = undefined 2
square : f( ) { }
s2 : undefined
s4 : undefined

Step 2



Work to do

Saturday 19
Day (050/315)

Closure :- This function forms a closure.

↓

```
function() { console.log(i)};
```

It remembers the reference to 'i' and it forms a closure. Takes the reference of i along with it.

* (SetTimeOut takes the callback function and stores it and attaches a timer to it and the JS proceeds to next line and logs and once the timer expires it takes the function and put it in callstack and proceed.)

Sunday 20

Print 1, 2, 3, 4, 5 after each seconds in JS.

{ 1 after 1 second
2 after 2 second
3 after 3 second }

4 after 4 seconds.

February 2022

18 Friday
Day (049/316)

(Day 7) :

* (setTimeOut + Closures) :

* JS waits for none

```
function x() {
    var i = 1;
    setTimeOut(function() {
        console.log(i);
    }, 1000);
    console.log("Hello");
}
x();
```

Hello

(output :- Prints 1 after 1 second)

* ** → If we add a log statement after the setTimeOut function then what will happen!

I thought that it will wait for 3 seconds, print 1 and then print hello, but I am wrong.

It prints hello, waits 3 seconds & then prints 1.

Kindness is a language the dump can speak and deaf can hear and understand.

How??

21 Monday
Day (052/313)

Wrong code :-

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

→ Output:

Hello
6
6
6
6
6

Because :- SetTimeout takes the function stores it somewhere and it remembers the reference of i. (not the value of i)

When loop runs first time, it does this, so in every time it remembers same reference of i (Note :- Not value but reference) as they have same environment.

Knowledge is the eye of desire and can become the pilot of the Soul.

Work to do

2022

February

Tuesday 22
Day (053/312)

~~Episode 11~~ :-

~~(Closure and setTimeOut)~~

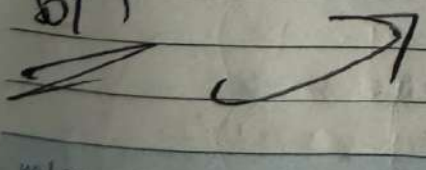
JS doesn't wait, it runs again & again. So when it comes to print hello and the settimeout timer expires it was too late then and i's value has already reached to 6. Since it has remembered the ~~value~~ reference of i, it prints 6, five times after each second.

(Use let as in each iteration, the "let i" will create a new scope which will refer to individual i.

[let are block scoped] → Means a new copy of i

[A new closure is created in each iteration]

```
function x(){
    for(var i=0; i<=5; i++){
        function close(x){
            setTimeOut(function(){
                console.log(x);
            }, x*1000);
        }
        close(i);
    }
}
x();
```



February

Date:- 13th Sept 2024

2022

23 Wednesday
Day (054/311)

So even if this function is created somewhere from where its parent it will remember the reference of defined variable & functions.

(Episode-12). (Interview of JS).

(1) Closure:- [A function along with the reference to its outer environment forms a closure.]

→ function + lexical scope bundled together

Function has access to its own environment & its parent

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

it has access to its outer environment

Calling inner function

If we return this function and call it somewhere else in code then also it still remembers its closure.

Q:- Even if we make var as let it will still form a closure and will work in same way.

The best part of our knowledge is that which teachers us where knowledge leaves off and ignorance begins.

Work to do

Thursday 24
Day (055/310)

```
function Outer(b){
    function inner(){
        console.log(a,b);
    }
    let a=10;
    return inner;
}
var close=outer("HelloWorld");
close();
```

→ (10, HelloWorld)

* In this case nothing changes as it will again form a closure and it will access b as well it is a part of outer env of inner function.

```
→ function outest(){
    var c=20;
    function outer(b){
        function inner(){
            console.log(a,b,c);
        }
        let a=10;
        return inner;
    }return outer;
}
var close=outer("hello");
close;
```

Inner function will still form closure with c

(10, hello, 20)

Work to do

February 2022

25 Friday
Day (056/309)

(Global variable with conflicting name)
(No change)

(Advantages of closure) → Most beautiful part of javascript

(a) Module Pattern →
(b) function currying → Currying
(c) High order function → Memoise, once

→ Helps us in data hiding & encapsulation.

(Data hiding & Encapsulation).

eg:- function counter() {
  var counter = 0; → This variable is hidden
  function incrementCounter() {
    counter++;
  }
}
console.log(counter); → Can't access this counter variable. Hence we hid the data.

This will create a new closure and start from 0 value again.

So if we didn't return and try to access the counter, we can't

var counter = counter()
counter1()
counter1()
var count2 = counter()
counter2()

Work to do

Labour is a pleasure in itself.

2022 February

Saturday 26
Day (057 / 308)

* More closures leads to more use of memory and they are not garbage collected.

As it will store the reference of data and hence the memory is used.

First class functions) → (New episode) → (Higher order functions)

(Date:- 6/7/2022)

(First class function) :-

→ function statement vs Expression vs declaration.

Garbage collector help us to free storage eg- unused variables.

→ function Statement

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

Sunday 27

→ function expression

function acts like a value

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

→ If we call function "a" before creating the function (hoisting), 'a' will work but b will not.

Work to do

```
function a() {
    var x = 20;
    return function b() {
        console.log(x)
    }
}
```

Relation between garbage collector & closure

February 2022

Mo	Tu	We	Th	Fr	Sa	Su
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28						

March 2022

Mo	Tu	We	Th	Fr	Sa	Su
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

February 2022

28 Monday
Day (059/306)

* Function declaration a Ka function Statement

[Anonymous function] :- Doesnot have its own identity.

function () { }

Invalid syntax
syntax error :- function statements require a function name.

** used at a place where it is used as a value

Anonymous function

var c = function () {
    Console.log("Hello");
}

Const c = () => {
    console.log("Hello")
}

*[Named Function Expression] :-

Same as Function Expression with a name.

var d = function xyz() { }

Real leaders are ordinary people with extraordinary determinations.

March '22 — Monthly Planner

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

→ Uncaught reference: xyz is not defined.

(We can access xyz inside xyz function) as it is inside its lexical environment.)

* Difference between Parameters & arguments

function ( ) → Parameters

local variables inside the function

→ [Parameters].

```
var b = function (p1, p2) {
    console.log("b called");
}
b(1,2);
```

→ [argument]

2022

March

Tuesday 01
Day (060/305)

* [First class functions]

```
var b = function (param1) {
        console.log (param1);
};
b (function () {  });
```

The ability to use functions as values & pass them to other function as argument also returned as a function.

* (Functions are 1st class citizens)

Arrow function came as ES6.

(New Episode).

* [Callbacks functions in JS ft. Event listeners]

(Callback) :- Take a function and pass it into another function, this function which is passed into another function is known as callback function.

Work to do

March 2022

02 Wednesday
Day (061/304)

Q: What is a callback functions in JS :-

→ Functions are first class citizens in JS → Take a function and pass it into another function.

The function which we pass into another function is known as callback function.

It gives us access to whole asynchronous world in a single threaded synchronous language

↓

Only do 1 thing at a time in a specific order.

```
function x(y) {
}
x(function y() {
})
```

Callback function
Work to do

This function y can be now called by function x anywhere in the code hence known as callback function.

What you cannot enforce, do not command.

2022

2022 March

Thursday 03
Day (062/303)

SetTimeOut takes a callback function as ~~[illegible]~~ a argument.

```
setTimeOut (function() {
        console.log("time")
},5000);
function x(y) {
    console.log("x"); y();
}
x(function y() {
    console.log("y");
});
```

y
y
time(after 5 seconds).

SetTimeOut will take the function and store it in a separate space ↔ attach a time to it.

1 callstack only → 1 thread
→ If any operation blocks the call stack it is known as blocking the main thread.

We should always try to use async operations for things which takes time.

Important

↓

Time Jada time lage ga waha

Work to do

March 2022

04 Friday
Day (063/302)

document.getElementById("clickMe")
.addEventListener("click", function(){
console.log("Button Clicked");
});

Callback function

So when we reach document line then JS will get the element "clickMe" attach an event listener to it which is a click event. So this event when triggered will provoke a callback function.

7) Closure demo with event listener).

Do not use global variable but use closures to encapsulate the data.

You will never be a leader unless you first learn to follow and be led.

Work to do

Saturday 05
Day (064/301)

```
function attachEventListeners() {
    let count = 0;
    document.getElementById("clickme").
    addEventListener("click", function xyz() {
        console.log("Button clicked!", ++count);
    });
}
attachEventListeners();
```

(callback function creating a closure).

Garbage Collection and removeEventListeners

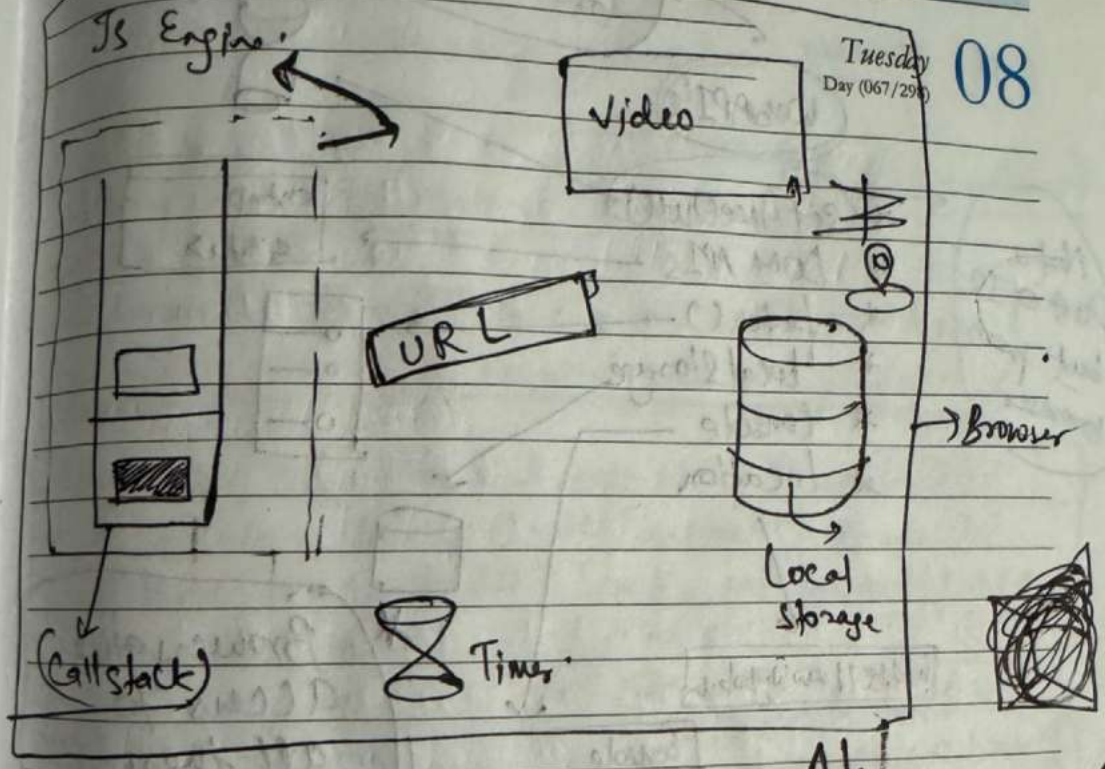
EventListeners are heavy (It takes memory) Even when it is not in call stack.

→ It forms a closure.

Sunday 06

2022 March

Tuesday 08
Day (067/298)



while running the code
In callstack, we need access to eg Bluetooth, GPS, local storage.

To access all these we need WEB API

Work to do

Wk	March						2022
	Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27

Wk	April						2022
	Mo	Tu	We	Th	Fr	Sa	Su
					1	2	3
	4	5	6	7	8	9	10
	11	12	13	14	15	16	17
	18	19	20	21	22	23	24
	25	26	27	28	29	30	

March 2022

09 Wednesday
Day (068/297)

superpowers which browsers have

WebAPI's

* setTimeOut()
* DOM APIs
* fetch()
* localStorage
* console
* location

Not a part of JS but from browser

<html>
</html>

https://www.google.com

console
>helloworld

Browsers give access to all these superpowers inside JS engine to callstack.

setTimeOut() is not a part of JS

DOM API
fetch
localStorage
console.log

Work to do

I would not exchange my leisure hours for all the wealth in the world.

2022 March

Thursday 10
Day (069/296)

We get access of these powers inside callstack because of global object.

Global Object is keyword "window". window.setTimeout()

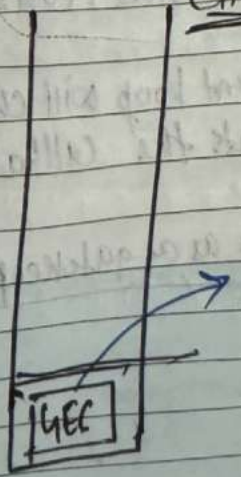
* The browsers give JS engine to facility all these powers through a keyword known as window.

* Global scope Mein present hote hain ye sb API's to hume window.setTimeout() karen ki jaroorat nahi padti. Simply setTimeout() access kar sakte hain.

```
console.log("start");
setTimeout(function cb() {
    console.log("Callback");
}, 5000);
console.log("End");
```

registers this callback and attaches a time of 5000 with it and JS moves to next line.

Callstack.



Work to do

→ Start
End
Callback

March 2022

11 Friday
Day (070/295)

* Meanwhile the callstack is empty the timer to the callback function is still running and as soon as it is expired the function is pushed into callstack for execution.

(Event loop) · (Callback Queue)

* The callback function cannot directly go inside the callstack when the timer expires.

→ A callback queue is used.

→ When the timer expires this callback function is put inside callback queue.

→ Event loop will check the callback queue and puts the callback function in the callstack.

(Acts as a gatekeeper).

We all live under the same sky, but we don't all have the same horizon.

Work to do

2022

March

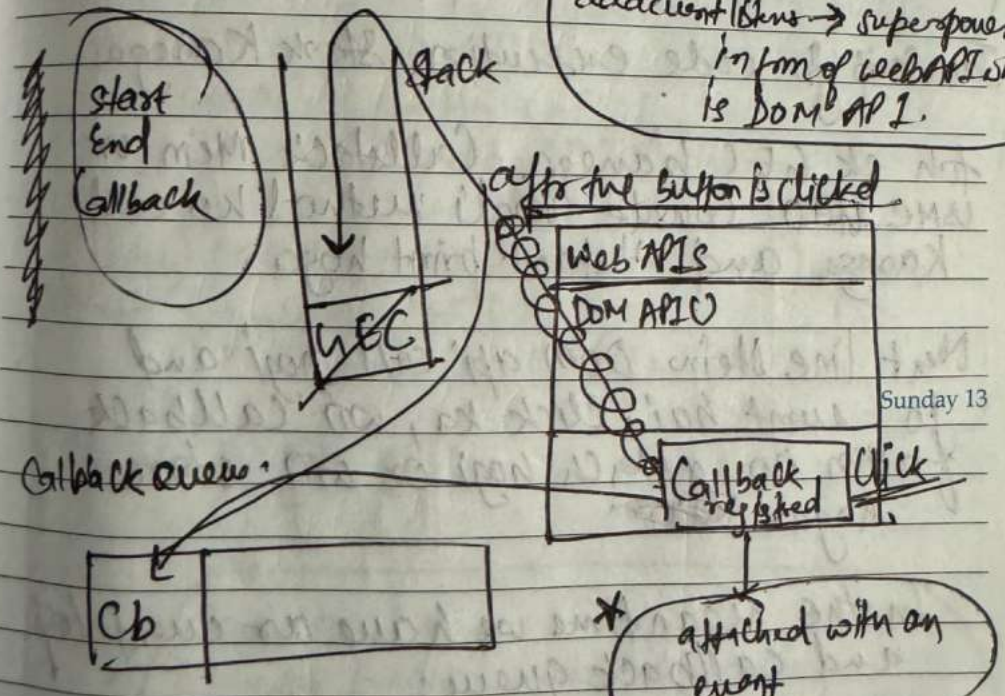
Saturday 12
Day (071/294)

How event listners work :-

```
console.log("start");
document.getElementById("btn")
  .addEventListener("click", function cb() {
      console.log("Callback");
  });
console.log("end");
```

registers a callback on event

addEventListener → superpower in form of webAPI which is DOM API.



attached with an event

When we click the button the callback function is provoked.

Sunday 13

March 2022

14 Monday
Day (073/292)

(Event loop) :- * (Continously monitor call stack & Callback queue.)

(Poora Method) :-

JavaScript jo code execution Stack karega toh ek GEC banega Callstack Mein or usME woo console wali method ko call karega and 1 start print hoga.

Next line Mein DOM api call hogi and jo event hai, click ka, woo callback function ko attach hogi or woo store hoga.

In the meantime we have our event loop and callback queue.

Agar hum click karte hain to woo callback function callback queue Mein aaega.

Life is really very simple, but men insist on making it complicated

2022

March

Tuesday 15

GEC call stack se bahar jab ho jayega or event loop jab check karega ki queue me koi function to wo usko stack me bhej dega for execution.

(fetch Works) ?.

```
console.log("Start");
setTimeOut(function CbT(){
    console.log("CB timeout");
}, 5000);
fetch("https://api.netflix.com").then(function CbF(){
    console.log("CB netflix");
});
console.log("End");
```

→ promise

Goes and request an API.
Fetch function returns a promise.
We have to pass a callback function which will be executed once this promise is resolved.

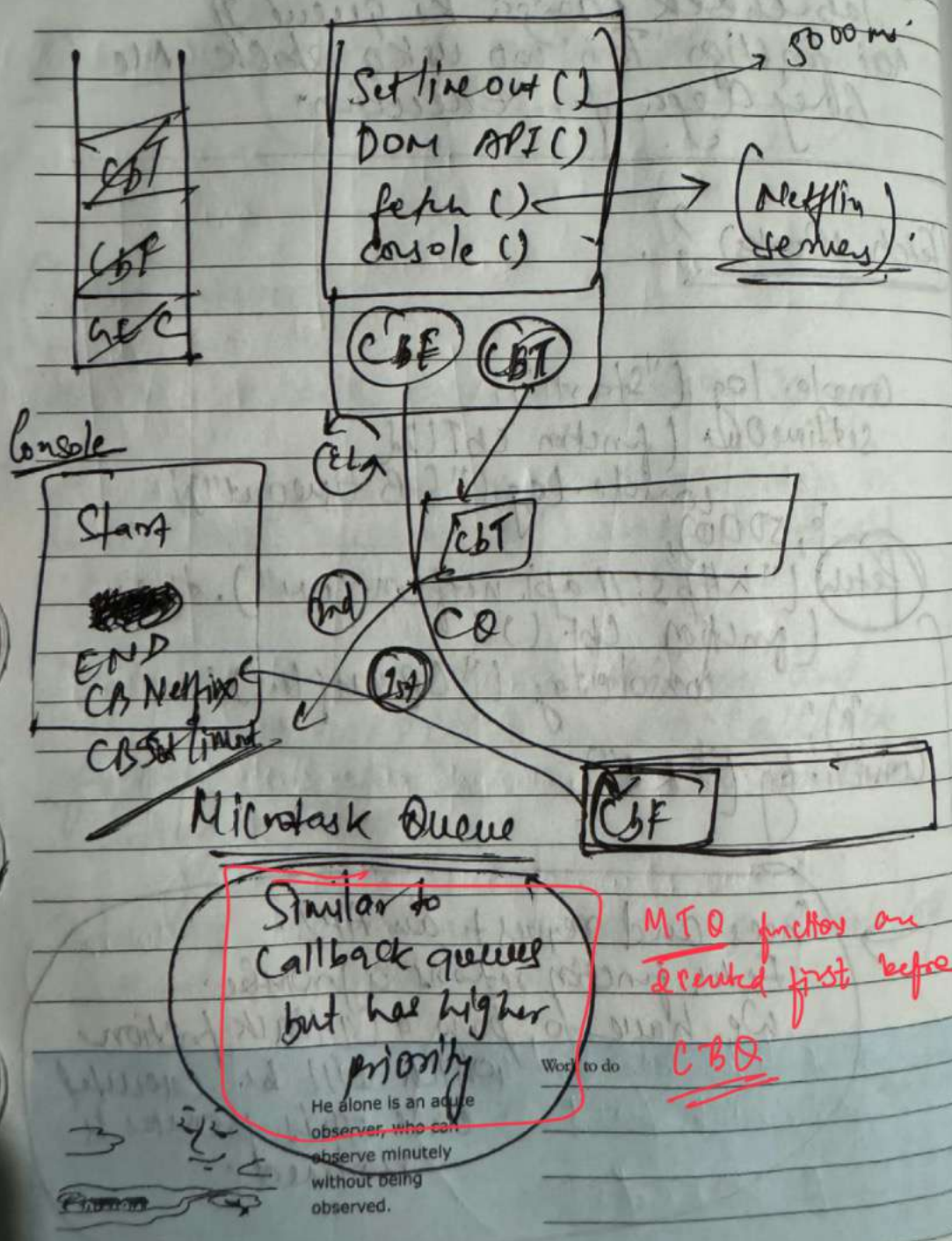
APIs

Mixed bhaji

JUN

March 2022

16 Wednesday
Day (075/290)



2022 March

Thursday 17
Day (076/289)

* (Microtask Queue) :

All callback functions which comes through promises will go inside MTQ.

(Promises and mutation observer) → Keeps on checking whether there is some mutation in the DOM tree or not.

(Given more Priority).

If there are three ~~microtasks~~ callbacks in microtask queue and only 1 in callback Queue. The event loop will give prefence to MTQ only.

~~Starvation~~

(Starvation of Callback queue).

Work to do

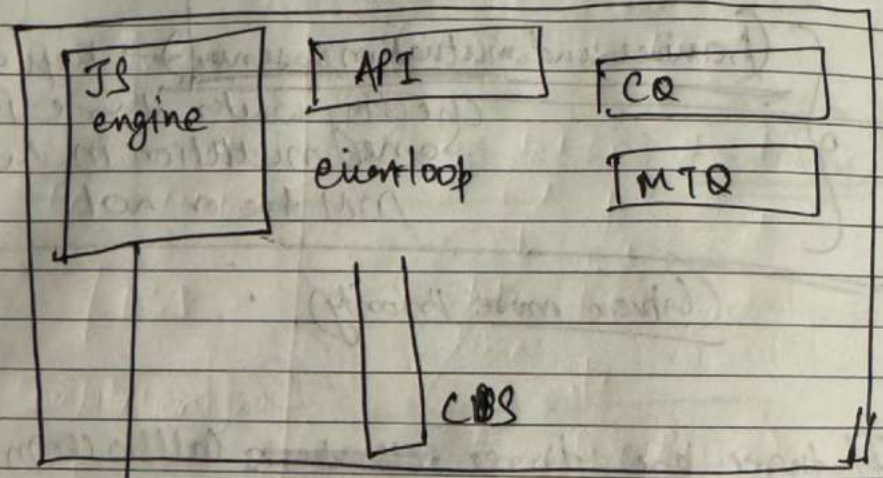
March 2022

18 Friday
Day (077/288)

(New Episode) :-

(JS Runtime Environment)

(JS Engine) :-



(heart of JS runtime environment)

NodeJS is an opensource JS runtime environment

can run this piece of code outside the browser.

It has done me good to be somewhat parched by the heat and drenched by the rain of life.

Work to do

Saturday 19
Day (078/287)

for eg:- If we want to run some piece of code inside a water cooler. We only need to have a JS runtime environment in that cooler.

There are a lot of JS engine's available -
All browers have a JS engine.

Microsoft edge JS engine → Chakra
firefox → SpiderMonkey
Chrome → V8

* { Used in node JS as well as deno JS }

Sunday 20

(First JS engine :- Creator of JS itself :→ SpiderMonkey)

(JS engine is not a Machine) : (It is a normal program written in C++.)

March 2022

Wk	Mo	Tu	We	Th	Fr	Sa	Su
		1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27
	28	29	30	31			

April 2022

Wk	Mo	Tu	We	Th	Fr	Sa	Su
					1	2	3
14	4	5	6	7	8	9	10
15	11	12	13	14	15	16	17
16	18	19	20	21	22	23	24
17	25	26	27	28	29	30	
18							

Work to do

March 2022

21 Monday
Day (080/285)

* Architecture (What happens) (JS engine)

Code → In JS
↓
PARSING
↓
AST
↓
Compilation
↓
Execution

(Broken into tokens)

eg- let a = 7
↓ ↓ ↓
1st token, 2nd token, 3rd

↓

(Syntax Parser)

↓

Convert into AST (Abstract Syntax tree)

eg- const bestJSCourse = "Namaste JS";

↓

AST : {
type: "Program",
"start": 0,
"end": 42,
"body": [
{ type: ...

Nothing would be done at all if a man waited will be could do it so well that no one could find fault with it.

2022

* AST

* (JS

* Interpr

* Compila

Fas

More

fur th in

But no

Wk	March	
	Mo	Tu
10		1
11	7	8
12	14	15
13	21	22
14	28	29

(astexplorer.net)

2022

astexplorer.net

March

* AST passed to Compilation phase.

Tuesday 22
Day (081/284)

* (Just in time compilation)

* Interpreter :- Takes code and execute code line by line.
* Compiler :- Whole code is compiler even before executing. It then forms optimized version of this code and then it is executed.

(Fast executed) → JS can behave as both interpreted aswell as compiled language.

(More efficiency)

(Depends upon the JS engine in which it is executed.)

Further if JS engine it was supposed to be an interpreted language.

But now interpreter + compiler.

W6	March						2022
	Mo	Tu	We	Th	Fr	Sa	Su
09		1	2	3	4	5	6
11	7	8	9	10	11	12	13
12	14	15	16	17	18	19	20
13	21	22	23	24	25	26	27
14	28	29	30	31			

W1	April						2022
	Mo	Tu	We	Th	Fr	Sa	Su
14					1	2	3
15	4	5	6	7	8	9	10
16	11	12	13	14	15	16	17
17	18	19	20	21	22	23	24
18	25	26	27	28	29	30	

Work to do

March 2022

23 Wednesday
Day (082/283)

Inlining
Copy elision
Inline Caching

JS engines can use an interpreter along with a compiler.

So,
Compilation and execution go hand in hand.

AST → Interpreter → Byte Code → Execution Step

(compiler) → To optimize the code as much as it can on the runtime.

*
Uses 2 components of JS engine
(a) Memory heap → All function & variables are assigned memory.
(b) Call stack.

(Garbage collector).
(Uses mark and sweep algorithm).

As I grow to understand life less and less. I learn to love it more and more.

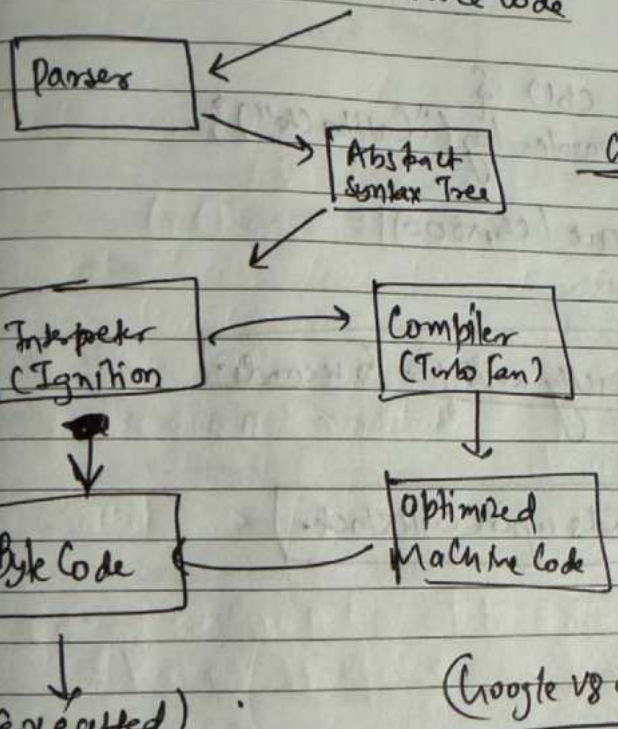
Work to do

2022

(V8 has Interpreter named as Ignition.
Turbofan is the compiler for V8)

2022 March

Thursday 24
Day (083/282)



Garbage Collector : Orinoco.

(Executed).

(Google V8 engine blog).

JS → functional
→ object oriented

it
time.

ARS

JUN

Work to do

Wk	March						2022
	Mo	Tu	We	Th	Fr	Sa	Su
10		1	2	3	4	5	6
11	7	8	9	10	11	12	13
12	14	15	16	17	18	19	20
13	21	22	23	24	25	26	27
14	28	29	30	31			

Wk	April						2022
	Mo	Tu	We	Th	Fr	Sa	Su
14					1	2	3
15	4	5	6	7	8	9	10
16	11	12	13	14	15	16	17
17	18	19	20	21	22	23	24
18	25	26	27	28	29	30	

March 2022

25 Friday
Day (084/281)

(Trust issues) with setTimeout()

doesnot exactly wait for 5 seconds

```
function cb() {
    console.log("Callback");
}
setTimeout(cb, 5000);
```

Can take 6 seconds, depends upon call stack

* doesnot exactly wait for 5 seconds.

* It depends upon callstack.

Still it will wait for the whole program to execute. 10 sec

```
console.log("Start")
setTimeout(function cb(){
    console.log("Callback")
}, 5000);
console.log("End")
```

{Takes 5 seconds. and has expired now}

#1 million line of code (Takes 10 seconds to execute)

Work to do

A liar will not be belived, even when he speaks the truth.

(18) (Rs.)

2022 (Season-2 Episode-1) March

Saturday 26
Day (085/280)

* [Callback hell]
  - (Good Part)
  - (Bad Part)

[Sychronous single threaded language] → JS
(Do one thing at a time.
It has one call stack and execute one thing at a time.)

Using callback we can make JS code asynchronous

```
Console.log("Hello");
setTimeout(function() {
    console.log("JS");}, 5000);
console.log("Season 2");
```

Const cart. [" ", " ", " "] → array of items

Sunday 27

api.createOrder()
api.proceedToPayment() → (In order to payment API to work it has to depend upon the create Order API to complete it's task.)

Work to do

March 2022

28 Monday
Day (087/278)

To callback wala code Kaisa hoga':-

- api.CreateOrder(cart, function() {
  api.proceedToPayment() })

Now it's the responsibility of createOrder API to call the proceedToPayment function whenever required.

Now what if we need to call another function only after the Payment api is called?

Then we need to pass that function (which is showOrderSummary as a parameter to payment API call as a callback.

Life's more than a magazine in which we flip the pages and enjoy the pictures.

Work to do

2022

March

Tuesday 29
Day (088/277)

```
api.createOrder(cart, function() {
    api.PTP(function() {
        api.SOS()
    })
})
```

[That's how we used callback for asynchronous tasks.]

[A large codebase and so many function calls dependent upon each other. We end up falling in a callback hell.]

Pyramid of doom

*[Inversion of (control) → (problem).]

[Loose the control of code while using callbacks.]

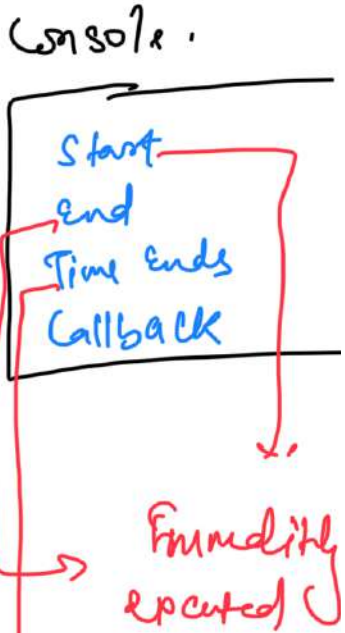
Whenever we have this callback function and we pass it to some other function we are giving the control of our callback function to that function.

It is very risky.

⇒ what if Cocah Ede has bugs and cannot call passed to payment

The trust issue with setTimeOut.

```
console.log("start")
setTimeout(function cb(){
    console.log("Callback"); }, 5000);
console.log("End");
let startDate = new Date().getTime();
let endDate = startDate;
while(endDate < startDate + 10000){
    endDate = new Date().getTime();
}
console.log("Time Ends")
```



Took 10 seconds to excute, in the mean time, the settimeout expired and was waiting in CBQ as main thread was blocked due to the while looping for 10 seconds.

if we put setTimeOut time to 0ms,

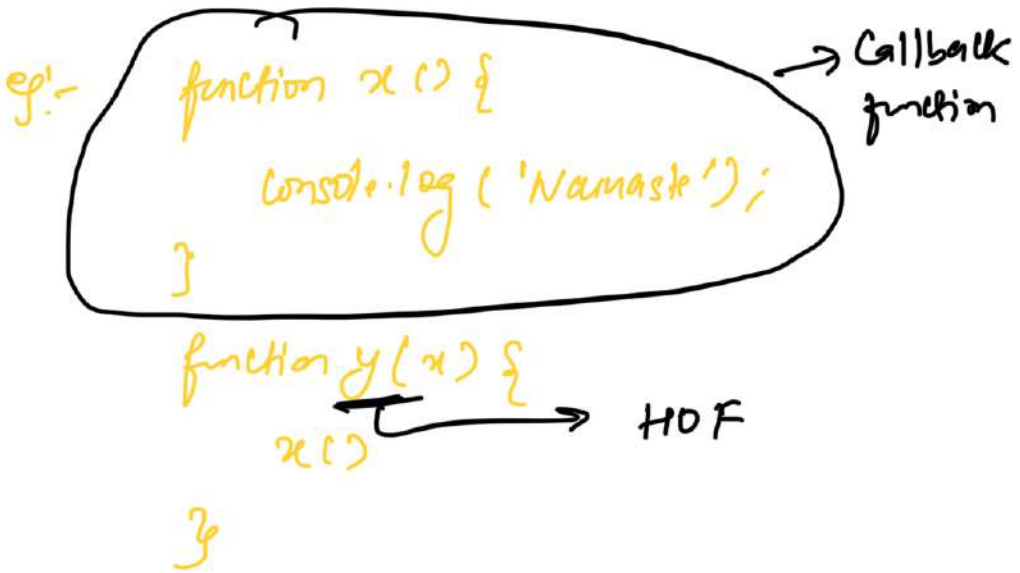
EL will still be registered in the queue and hence excuted later.

Setting up the environment

1) Download vs code
2) Chrome
3) Folder → index.html
   index.js
4) Live Server

Functional Programming → Episode -18

Function which takes another function as an argument or returns function from it is called H.O.F.



Modular code / Functional way

DRY Principle : Don't repeat yourself.

→ [Create functions of different requirements and make the main function generic.]

Abstract the logic out and pass it down to main function (made generics.

Think or make logic according to functions.

→ Every function will have its own functionality.
→ Main function will not have the basinus logic.

Map is a higher order function.

So,

```
const calculate = function (arr, logic) {
    ....
    ....
};
```

↓ similar to map

If we want to use calculate as radius-map, we want to do radius.calculate, Just do.

```
Array.prototype.calculate = function ... { }
```

So now you can attach calculate to any array.

Very Important

Map polyfill →

```
Array.prototype.calculate = function (logic) {
    const output = [];
    for (let i = 0; i < this.length; i++) {
        output.push (this[i])
    }
    return output;
}
```

now we can use calculate with any array as method.

→ radius.calculate (logic)

HOF → A function which takes another function as input or return a function from itself.

CF → A function passed into a HOF is called callback function.

Episode - 19.

Map → Transform each and every value of the array and get a new array out of it.

```
const arr = [5, 1, 3, 2, 6]
→ //double : [10, 2, 6, 4, 12] → double the original value.
```

```
function double(x) {
    return x*2
}

const output = arr.map(double)
```

→ Run double function on each element of the array.

{ map is hence a HOF }

```
const output = arr.map((x) => {
    return x.toString(2)
})
```

forEach vs map vs for

Filter function → used to filter.

```
const arr = [5, 1, 3, 2, 6]

// filter odd values

function isOdd(x) {
    return x%2;
}
```

function will make it filter with what logic put inside.

const output = arr.filter(isOdd)

Reduce function → doesnot reduce anything → used at a place when we have to take all the elements of the array and come up with a single value out of them.

const arr = [5, 1, 3, 2, 6]

```
function findSum(arr){
  let sum = 0
  for(let i=0; i<arr.length; i++){
    sum = sum+arr[i];
  }
  return sum
}
```

For eg:- Find the sum of the array

Largest value

Minimum value

accumulator, current

```
const output = arr.reduce(function(acc, current){
    acc = acc+curr
    return acc
},0)
```

0 → initial value of the accumulator.

Reduce iterates the function over each element of the array, current represents the value of the current element being iterated, acc is the accumulator, the result which we have to get

→

```
function findMax(arr){
let max = 0;
for(let i=0; i<arr.length; i++){
    if (arr[i]>max){
      max= arr[i]}
    }
  return max
}
```

⇒

```
const output= arr.reduce (function(acc, curr){
    if (curr>acc){
        acc = curr}
    return acc
}, 0)
```

↓ initial value

```
const users= [
  {firstname : "a", lastname: "b", age: 26},
  {firstname : "a", lastname: "b", age: 75},
  {firstname : "a", lastname: "b", age: 30},
  {firstname : "a", lastname: "b", age: 26}  ]
```

// find out the list of [illegible]

→ const output = users.map(x => x.firstName + x.lastName)

// How many users have same age, let say 10.

→ // {26: 2, 75: 1, 50: 1} [one object with diff unique values]

```
const output = users.reduce(function(acc, curr) {
    if (acc[curr.age]) {
        acc[curr.age] += 1
    }
    else { acc[curr.age] = 1 }
    return acc
}, {})
```

This will be empty object as initial value as our output will look like {26: 2, 75: 1, 50: 1}

[Reduce can be used at multiple places. Think it of more often.]

filter first name of all people whose age is less than 30.

```
let output = users.filter((user) => user.age < 30);
```

last output: `users.filter((usr) => usr.age < 30).map((x) => x.firstName)`

This is called chaining

age < 30 & first name using reduce.

```
const output: users.reduce((acc, usr) => {
    if(usr.age > 30){
        acc.push(usr.firstName)
    }
    return acc
}, [])
```

Polyfill of map, filter and reduce → Important.

Season 2 → episode - 2

→ Promises represent objects which represents eventual completion or rejection of an asynchronous event.

Promises are used to handle async operations in javascript.

const cart = ["shoes", "pants", "kurta"]

createOrder (cart) ; → // give an order ID

proceedToPayment (orderID) ; → // proceed to payment

} asynchronous and dependent on each other.

⇒ Callback Mein kaise Karte the →

```
createOrder (cart, function () {
    proceedToPayment (orderId);
});
```

→ leads to callback hell & inversion of control.

[We think that createOrder will callback proceedToPayment function. We can't blindly trust this function to callback our fn.]

⇒ The promise way.

const promise = createOrder (cart);

↳ async operation and it will take some time.

// { data : }

Whatever it returned

Initially it will be empty and when response is returned from createOrder, it will fill data.

Now we attach a callback to this promise object.

```
=> promise.then(function(orderId) {
        proceedToPayment(orderId)
    })
```

This will be automatically called once data is filled with orderdata.

Here we are attaching a cb to promise but in previous method we passed the cb

(This pTp callback will only be called once the promise is returned from the server.

[fetch() → api function provided by browser to call external servers.

It returns a promise.

State of a promise → pending → fulfilled

result → stores the data returned by promise.

```
const api = "....."
const user = fetch(api)
console.log(user)

user.then(function(data){
    console.log(data);
```

Immutable

Resolved just once.

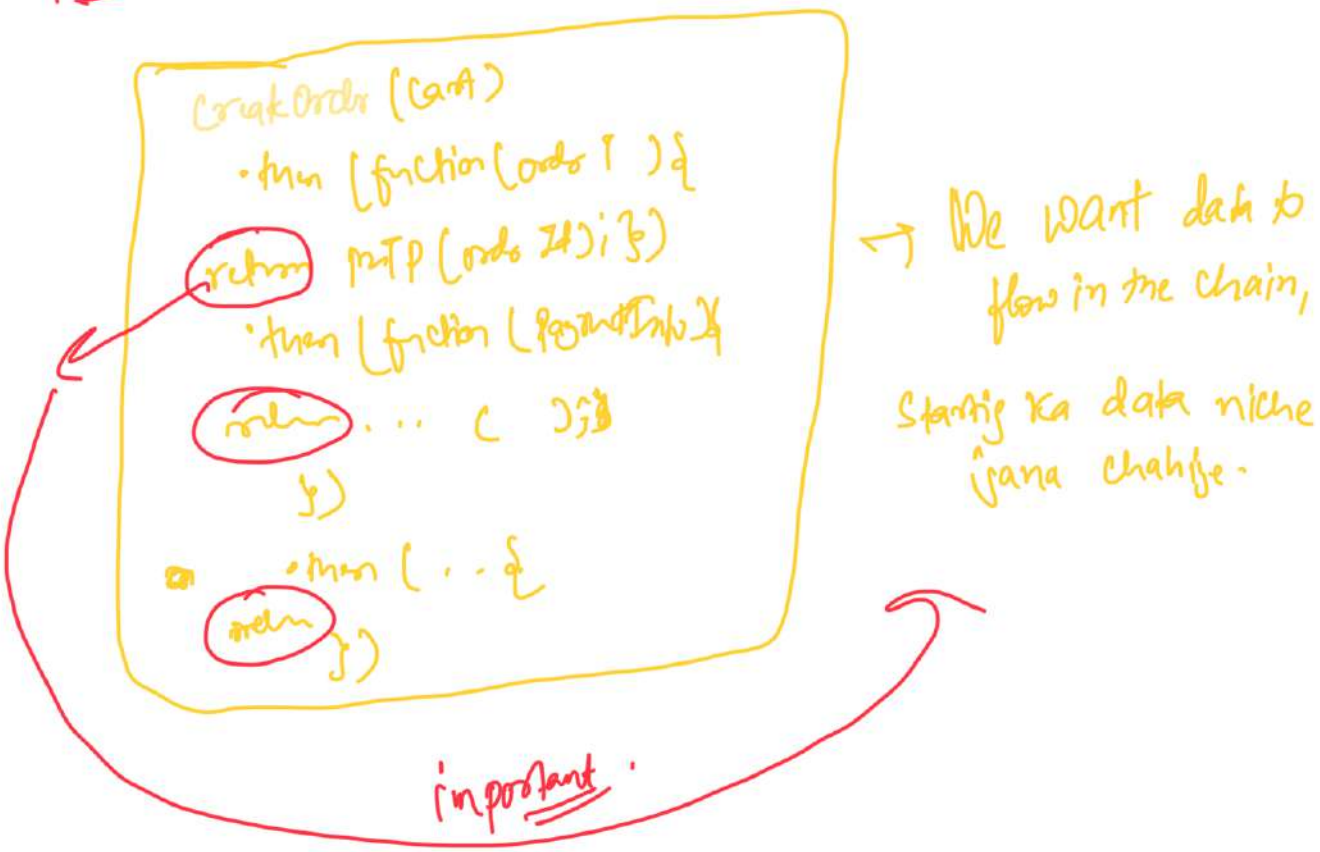
});

Interview Questions

What is promise:-

An object representing eventual completion or failure of an asynchronous event.

Promise Chaining



→ We want data to flow in the chain,

Starting ka data niche jana chahiye.

****

what is a promise → promise is an object used for representing eventual completion or failure of an async events.

why promises → To get out of callback hell and inversion of control.

How we can consume a promise?

Season-2, episode-3

How to create a promise

So yahan pe, Jab createOrder Call hua, wo simulate hua api call jaise jo hme 5 second baad response dega. Us beech, jb hum promise log kiye to pending state dikhaya and 5 second baad response mila.

```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart);
console.log(promise)
promise.then(function () {
    console.log(orderId
    proceedToPayment() })

function createOrder(cart) {
    const pr = new promise(function (resolve, reject) {
        if (!validateCart(cart)) {
            const err = new Error("Cart is not valid");
            reject(err);
        }
        const orderId = DBCall.getOrderId    // 12345
        if (orderId) {
            setTimeOut(function() {
                resolve(orderId);
            }, 5000)
        })

function validateCart() {
    return true
```

Iska matlab JS provide only 2 things can happen.

function

→ The output will be

→ promise <pending>

→ after sec → 12345 → order Id.

ek baar or :-

hamaare paas cart tha :-

cart = ["A", "B", "C"]

Iss cart se hum createOrder api call karenge.

To hume promise return karaaega.

```
const promise = createOrder(cart)
```

Jab ye promise resolve ya reject hoga To hum ek dusra api call proceedToPayment karenge.

→ same hona chahiye jo return ho raha hai

```
promise.then(function(orderId) {
    console.log(orderId)
    proceedToPayment(orderId) })
```

AB apna promise banate hain

```
function createOrder(cart) {
    const pr = new promise(function(resolve, reject) {
        if (!validateCart(cart)) {
```

Ye JO
reject karega.

```
const err = new Error( Cart is blank )
reject (err)
}
const ordoId = 12345 → Suppose DB se aaya
if (ordoId) {
```

Timeout deke
isko actual API
bana Rahe.

```
setTimeOut (function () {
    resolve (ordoId) }, 5000)
}
})
```

```
const validateCart (cart) {
return true }
```

Agar hume reject mila promise se toh,

```
promise. then (function() {
    console.log (ordoId)
    pOT (ordoId) })
```

```
. catch ( function (err) {
    console log (err. message)
}
```

[gracefully handling the error]

[and showing the message.]

promise chaining

```
createOrder(cart)
  .then(function(orderId){
      console.log(orderId)
      return orderId
  })
  .then(function(){
      return proceedToPayment(orderId);
  })
  .then(function(paymentInfo){
      console.log(paymentInfo)
  })
  .catch...
```

Yahan pe bass hume log karna tha, yahan pe ?TP wo call kaun karega.

Chaining

What is wrong here?

We didn't return the previous promise.

"If there is a big promise chain and we get error in any part of the chain, then the catch will handle anything inside the chain.

agar sirf hume particular chain tak hi error pata karna hai, to hum wo catch block ko us level ke then tak leke chale jaenge.

// Create Order [Create a promise]

// proceedToSignIn
//showOrderSummary
//updateWalletBalance

Chain, handle error.

S02, E04 **Async Await**

→ Async → keyword used before a function to make it async.

⇒ asyncfunction getData() {
return "Namaste";
}

→ how it is different?

→ This always returns a promise.

Always return a promise

*** If we don't return a promise. Then this function will automatically wrap this value inside a promise and return it.

promise { <fulfilled>: 'Namaste' }

const data = getData()

↓

will contain a promise returned by getData

data.then(res ⇒ console.log(res));

→ Namaste

Let's return a promise.

```
const p = new promise [function (resolve, reject) {
        resolve ("promise Resolved value")
}
```

Consoled this value

↓
This is returning promise.

```
async function getData() {
    return p;
}
```

⇒ This is promise, not a value so it will not be wrapped, but simply the promise will be returned

```
const dataPromise = getData()
dataPromise.then((res) => console.log(res));
```

⇒ Promise Resolved Value

Using await with async?

⇒ Async and await combo is used to handle promises.

How was it before?

```
const p = new Promise ((resolve, reject) => {
        resolve ("Promise Resolved Value");
});
```

→ Not async function

```
function getData() {
    p.then((res) => console.log(res)) }.

getData();
```

Console → Promise Resolved Value.

How do we handle this thing async await

```
const p = new Promise ((resolve, reject) => {
    resolve ("promise Resolved Value");
});

async function handlePromise () {
    const val = await p;
}
console.log(val) → promise Resolved Value
handlePromise()
```

This is how we handle promise in async await.

await → keyword only used inside an async function.
↳ in front of promise.

Difference between both methods

Normal promise

→

```
const p = new Promise ((resolve, reject) => {
    setTimeOut ( () => {
        resolve ("promise Resolved Value");
    }, 10,000), });
```

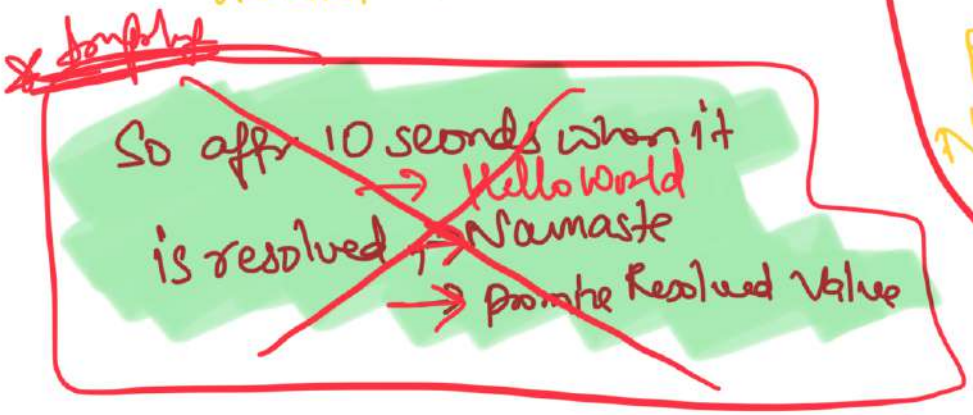
```
function getData() {
    p.then((res) => console.log(res))
    console.log("Namaste")
}
getData();
```

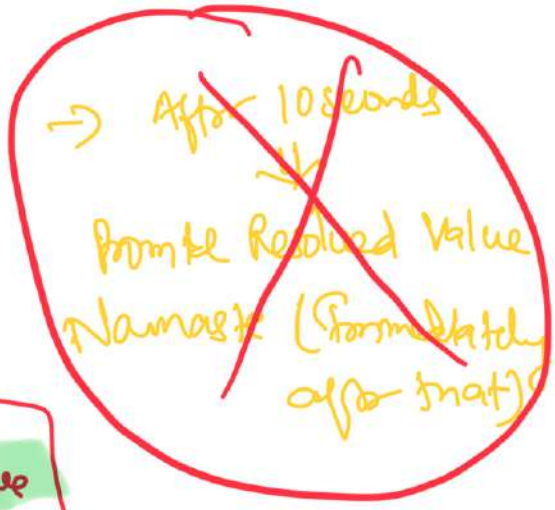
Console > Namaste
Promise Resolved Value (After 10 seconds)

Here what is happening is that since the promise took 10 seconds to resolve it will hold that function callback for 10 seconds and print Namaste first and when the timer expires it will display the resolved text. This was very confusing to developers.

How it work in async await -> Yes you guessed it right. JS engine will wait for it to be resolved first.

```
const p = new Promise((resolve, reject) => {
  setTimeOut(() => {
    resolve("Promise Resolved Value")}, 10000);
});

async function handlePromise() {
  console.log("Hello World")
  const val = await p;
  console.log("Namaste")
  console.lg(val);
}
handlePromise()
```





This is also wrong.

Conclusion:- It will only wait at the await line and will make the code after await wait, anything before that will be instantly executed.

So above example:-

Hello World

waits 10 second

IMPORTANT =>

Namaste JavaScript + Promise Resolved Value.

It still executes the log first and then the promise log.

(Basics of JS is still there)

What if we have multiple await inside the async function ?

```
asyn function handlePromise () {
    console.log ("Hello World")
    const val1 = await p
    console.log ("Namaste JS")
    console.log (val1)
    const val2 = await p
    console.log ("Namaste JS 2")
    console.log (val2)
}
handlePromise ();
```

-> More complicated code :-

```
12  const p1 = new Promise((resolve, reject) => {
13    setTimeout(() => {
14      resolve("Promise Resolved Value!!");
15    }, 10000);
16  });
```

```
17
18  const p2 = new Promise((resolve, reject) => {
19    setTimeout(() => {
20      resolve("Promise Resolved Value!!");
21    }, 5000);
22  });
23
24
25  // await can only be used inside an async function
26  async function handlePromise() {
27    console.log("Hello World!!");
28    // JS Engine was waiting for promise to reoslved
29    const val = await p1;
30    console.log("Namaste JavaScript");
31    console.log(val);
32
33    const val2 = await p2;
34    console.log("Namaste JavaScript 2");
35    console.log(val2);
36  }
37  handlePromise();
38
```

Hello World

10 second timer starts 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

P2 timer expired but still not logged

Namaste JavaScript
promise Value Resolved
Namaste Javascript 2 → This was already resolved at 5 seconds mark.
promise Value Resolved

printed together

Agar ab P1 ko 5 second kar dein and P2 ko

10 second [illegible]

Hello World

Timestart for 5 sec → 1, 2, 3, 4, 5 ↓ P1 resolves ← P1 resolves

Namaste JS

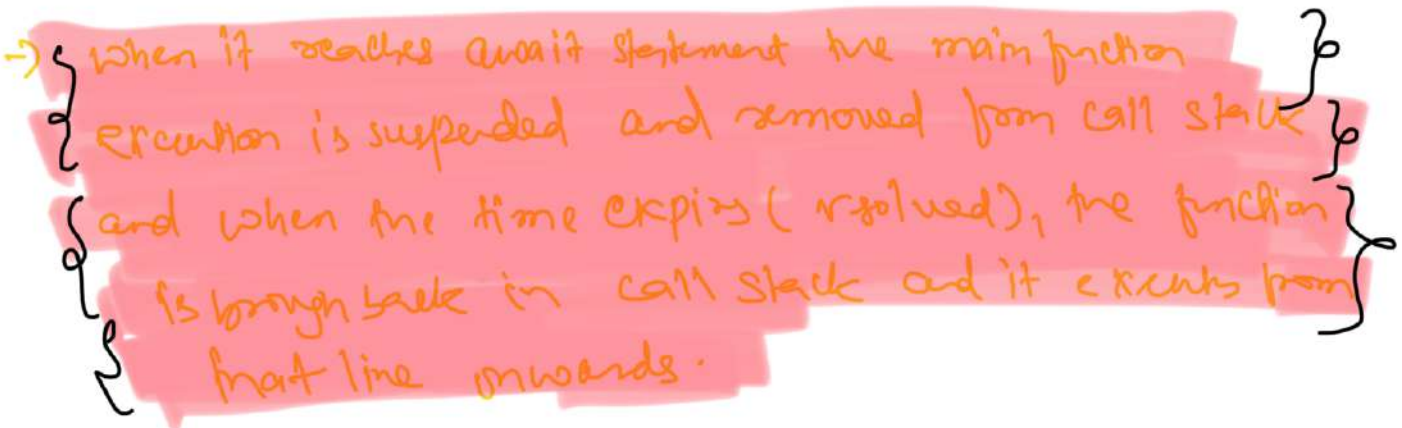
promise Resolved

Time continued for 5 more second (Total 10 se, 6, 7, 8, 9, 10) ↓ P2 resolves ← P2 resolves

Namaste JS 2

promise Resolved

How does it work behind the scene.



Real world example.

async function handlePromise ( ) {

const data = await fetch(API_URL); → It gives a promise so put await

await jsonValue = await data.json()

{This is also a promise}

So same, the handle promise in call stack will be suspended here and once it is resolved, if we have other statements, it will executed from by putting handlepromise back inside the call stack.

console.log(jsonValue)

}

handlePromise()

Error Handling

We don't have a .catch inside of asyncawait.

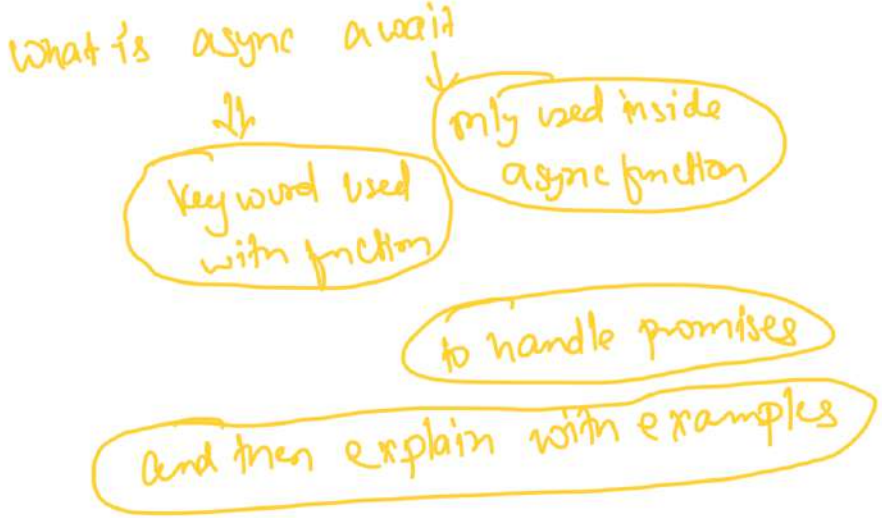
```
async function handlepromise() {
  try {
  const data = await fetch(API_URL);

  await jsonValue = await data.json()
```

{This is also a promise}

```
console.log(jsonValue)
} catch (err) { console.log(err) }
}

handlePromise()
```

So it will be handled by the catch block.

Interview Question



⇒ What should we use?

Asyn await is just a syntactic sugar.
Behind the scene JS is using .then promise only

Better ? → Both are good

Last episode

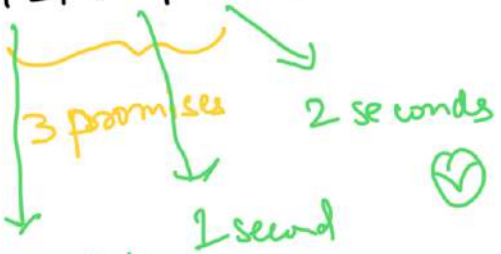
Promise APIs

(1) Promise.all( ) ← [If we need to call multiple parallel api calls.]

Fail fast

It takes an array of promises.

Promise.all([P1, P2, P3]) → It will make 3 parallel api calls.



[val1, val2, val3]

All were success api call

Output of promise.all() will be an array with the result of all three promises.

→ How much time will it take?

After 3 second we will get the output array. It will wait for all of them to finish.

What if any promise gets rejected?

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

↓ ↓ ↓ 2secs

3secs 1sec

↓

Rejected

Important:

As soon as any of these promise gets rejected, promise.all will throw an error.

output will be ERROR (same error recived by P2)

Immediately, as soon as error happened, it will return the error. In this case P2 got error after 1 second so, the error output will be received after 1 second. It will not even wait for other promises to resolve.

[THIS IS KIND OF ALL OR NONE]

(It will not wait for other promises)

What if the 2second promise got rejected, then 1second promise will be resolved.

(2) promise.allSettled( ) → takes an array of promises →

[P1, P2, P3]

↓ ↓ ↓

3s 1s 2s

if all was resolved,

after 3s → [val1, val2, val3]

if one was rejected (P2 was suppose ...)
after 1 sec.

* It will wait for all promises to settle. If doesnot matter if it is success or failure.

output: [val1, Error2, val3]

(3) Promise.race() ← Takes an array of promises as argument.

[P1, P2, P3]
↓ ↓ ↓
3s 1s 2sec

As soon as first promise is resolved, it will give me the result of first settled promise.

Suppose → P1, P2, P3
3s 5s 2s
↓
[resolved after 2second first so value returned will be value3.]

What if first promise was rejected?
So lets take example if P3 fails
error will be thrown.

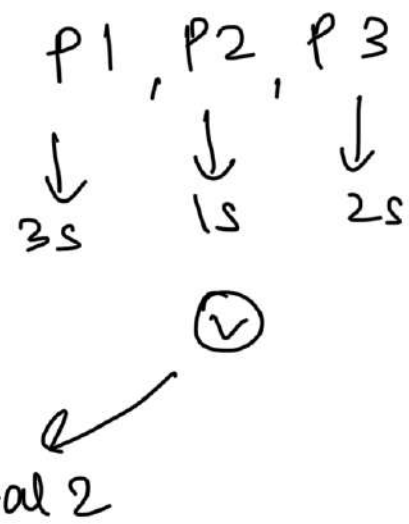
It will return result of first settled promise. It will

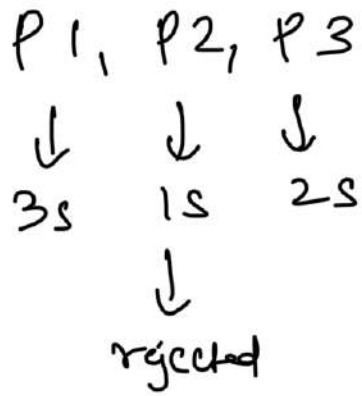
not wait for other promise to be resolved.

(4) Promise.any( ) → Takes an array of promises

([P1, P2, P3])

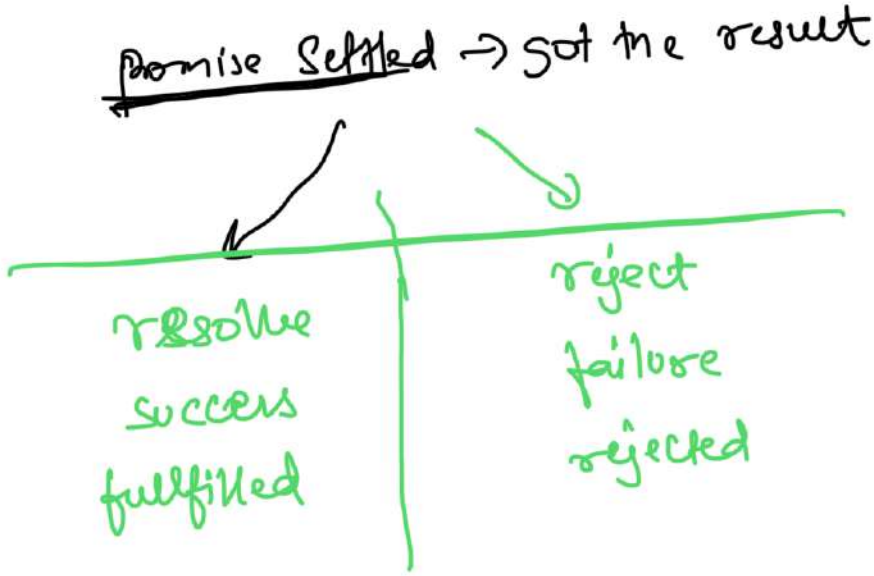
It is very much similar to race. Whenever first promise gets resolved. It will wait for the first promise to get succesfull.



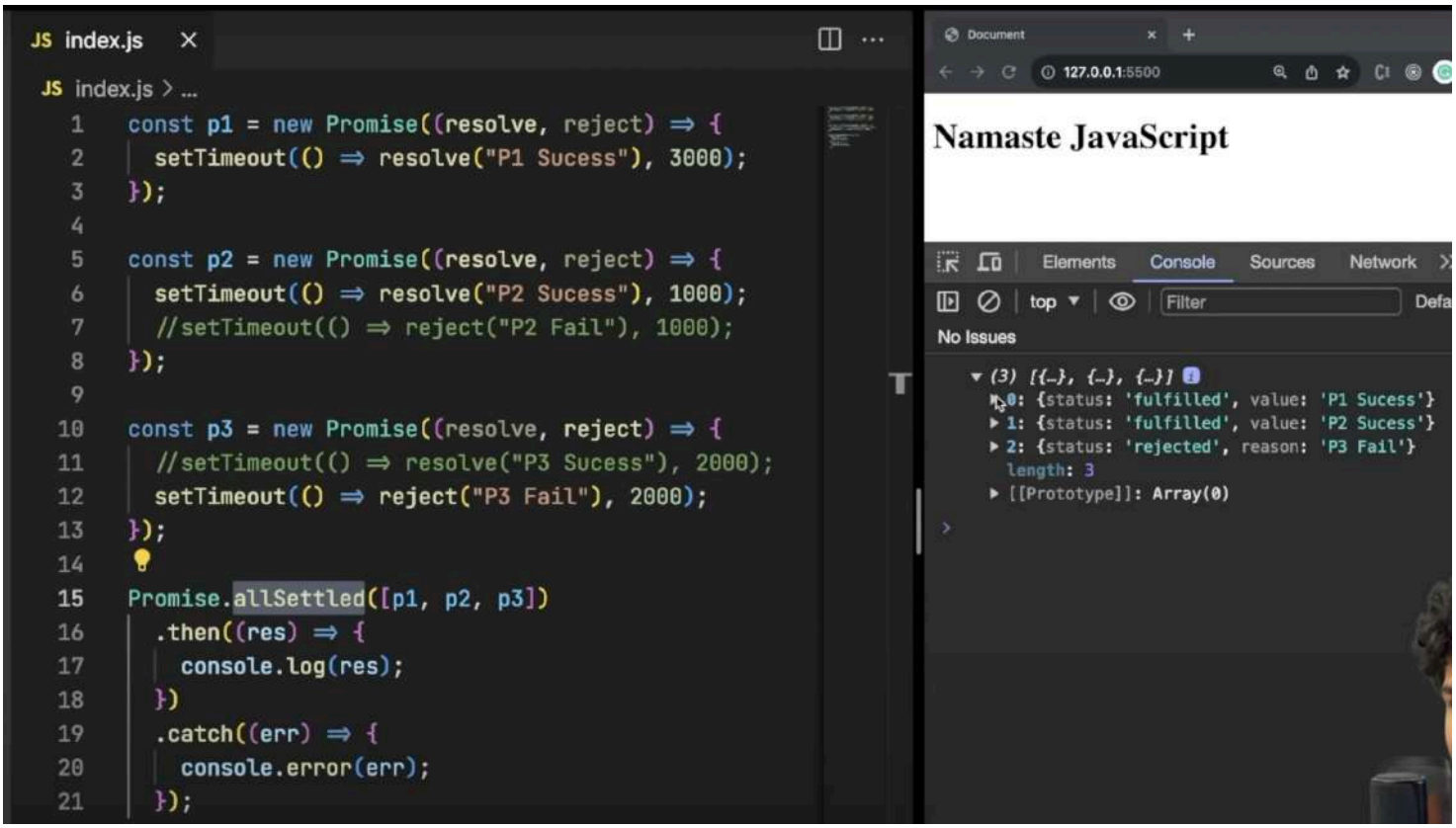


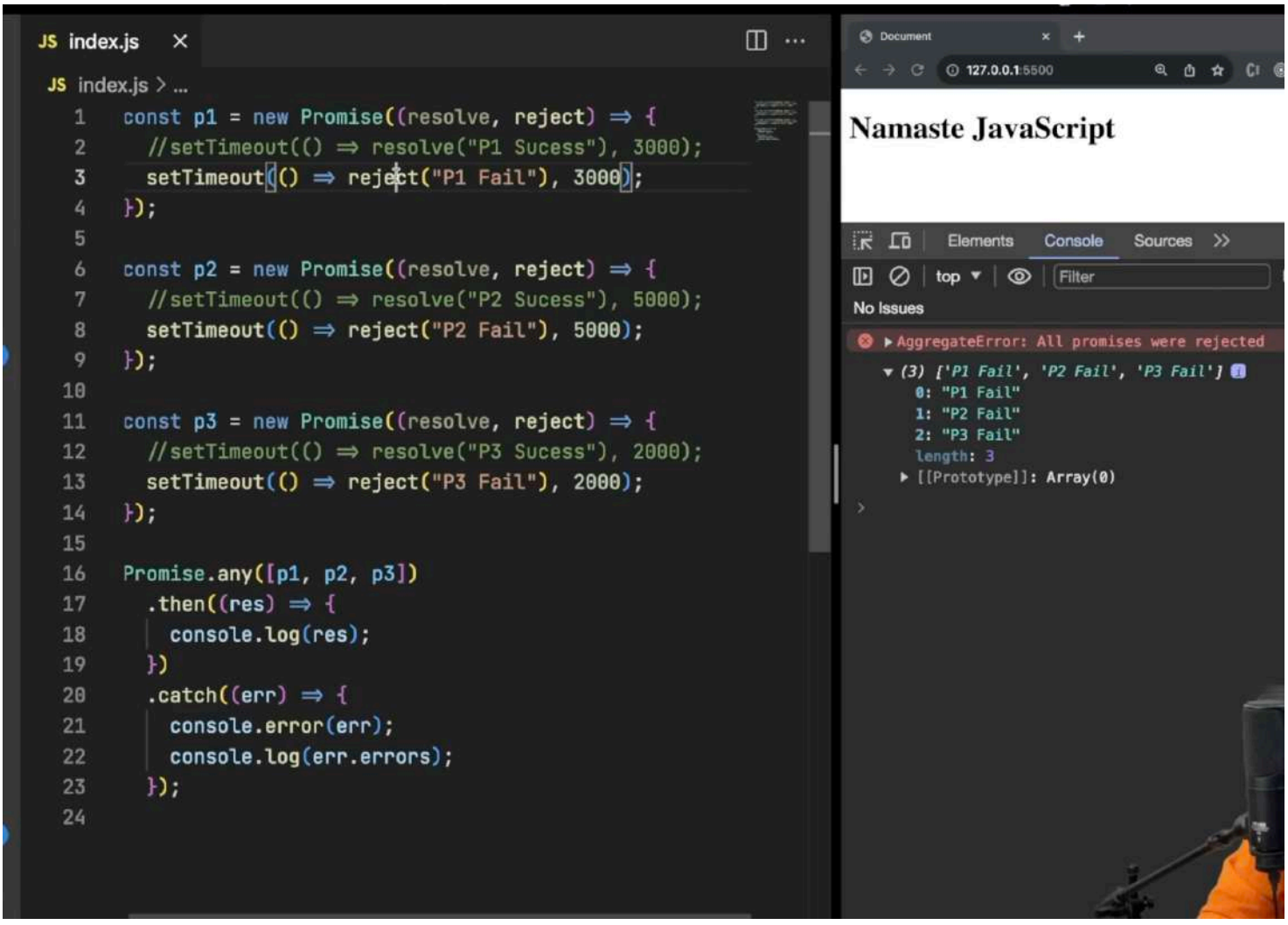
So it will wait for the first promise to resolve even if something was failed earlier.

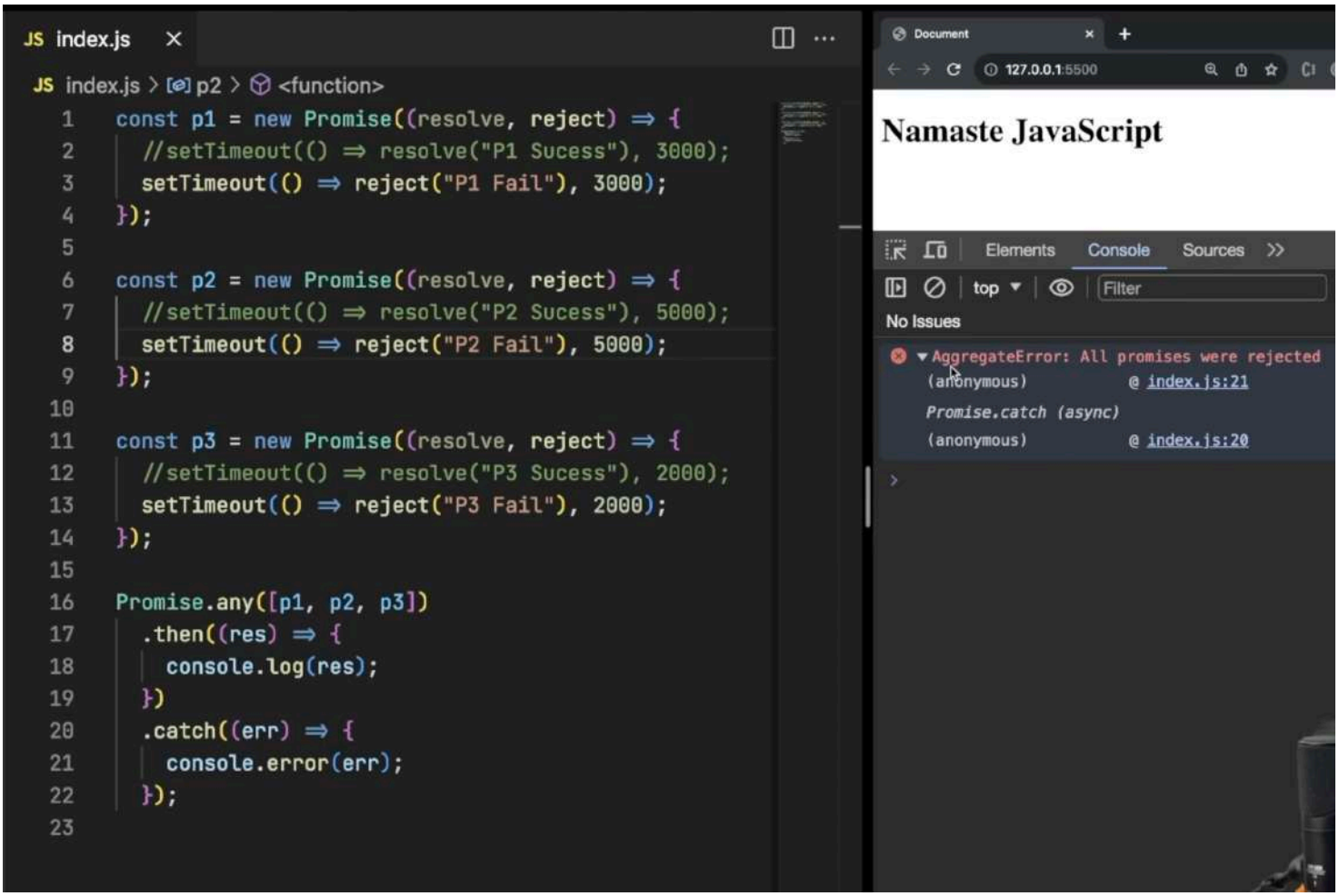
what if everything fails?
→ returned result will be aggregate error which will be an array of all the errors.
[error 1, error 2, error 3]

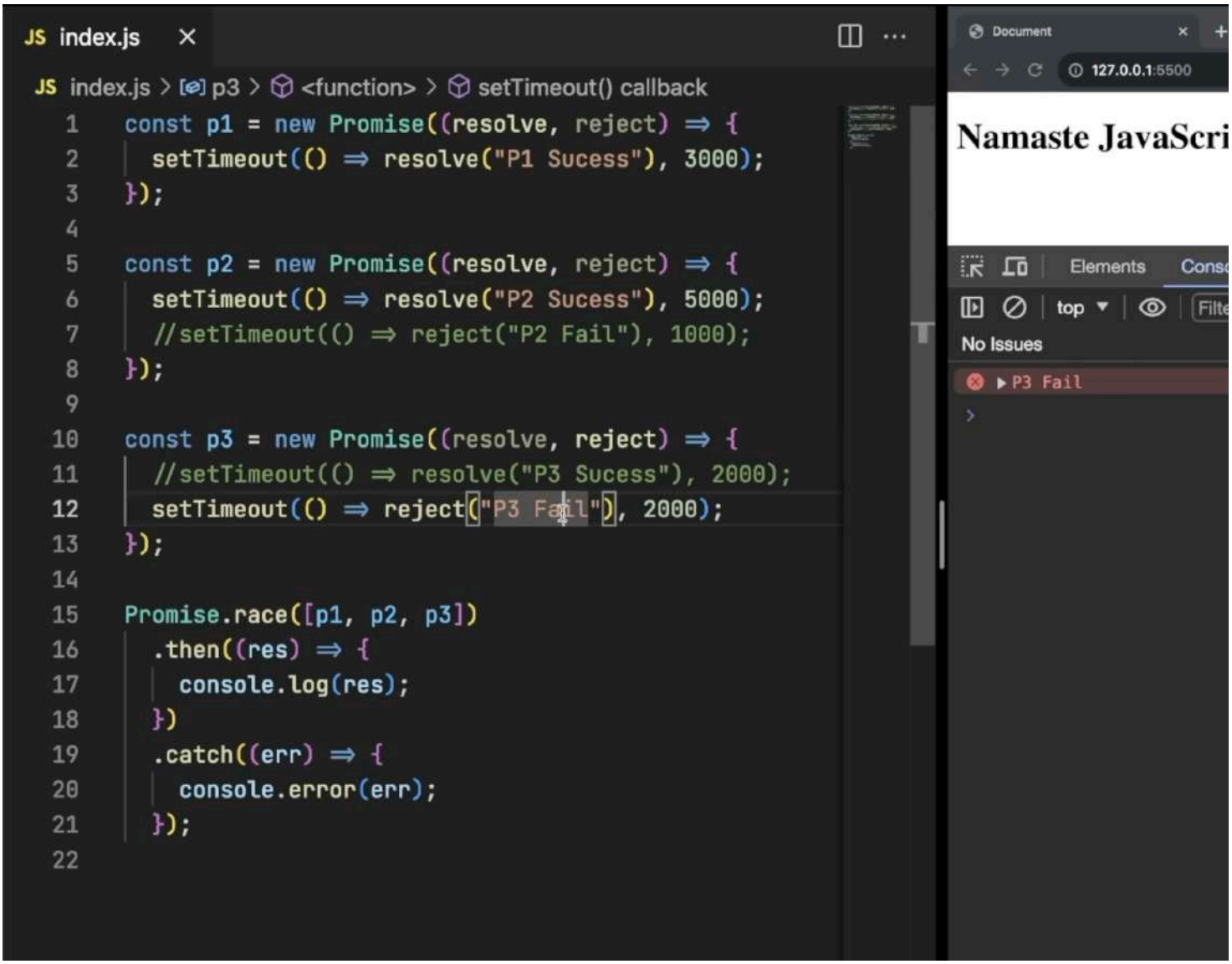

Promise Settled → Got the result
resolve
success
fullfilled
reject
failure
rejected

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P1 Sucess"), 3000);
});

const p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("P2 Sucess"), 1000);
  //setTimeout(() => reject("P2 Fail"), 1000);
});

const p3 = new Promise((resolve, reject) => {
//setTimeout(() => resolve("P3 Sucess"), 2000);
  setTimeout(() => reject("P3 Fail"), 2000);
});

Promise.all([p1, p2, p3])
  .then((res) => {
    console.log(res);
  })
  .catch((err) => {
    console.error(err);
  });
```


index.js
const p1 = new Promise((resolve, reject) => {
setTimeout(() => resolve("P1 Sucess"), 3000);
});
const p2 = new Promise((resolve, reject) => {
setTimeout(() => resolve("P2 Sucess"), 1000);
//setTimeout(() => reject("P2 Fail"), 1000);
});
const p3 = new Promise((resolve, reject) => {
//setTimeout(() => resolve("P3 Sucess"), 2000);
setTimeout(() => reject("P3 Fail"), 2000);
});
Promise.allSettled([p1, p2, p3])
.then((res) => {
console.log(res);
})
.catch((err) => {
console.error(err);
});
Document
127.0.0.1:5500
Namaste JavaScript
Elements
Console
Sources
Network
top
Filter
No Issues
(3) [{…}, {…}, {…}]
0: {status: 'fulfilled', value: 'P1 Sucess'}
1: {status: 'fulfilled', value: 'P2 Sucess'}
2: {status: 'rejected', reason: 'P3 Fail'}
length: 3
[[Prototype]]: Array(0)


index.js
const p1 = new Promise((resolve, reject) => {
//setTimeout(() => resolve("P1 Sucess"), 3000);
setTimeout(() => reject("P1 Fail"), 3000);
});
const p2 = new Promise((resolve, reject) => {
//setTimeout(() => resolve("P2 Sucess"), 5000);
setTimeout(() => reject("P2 Fail"), 5000);
});
const p3 = new Promise((resolve, reject) => {
//setTimeout(() => resolve("P3 Sucess"), 2000);
setTimeout(() => reject("P3 Fail"), 2000);
});
Promise.any([p1, p2, p3])
.then((res) => {
console.log(res);
})
.catch((err) => {
console.error(err);
console.log(err.errors);
});
Document
127.0.0.1:5500
Namaste JavaScript
Elements
Console
Sources
top
Filter
No Issues
AggregateError: All promises were rejected
(3) ['P1 Fail', 'P2 Fail', 'P3 Fail']
0: "P1 Fail"
1: "P2 Fail"
2: "P3 Fail"
length: 3
[[Prototype]]: Array(0)


index.js
index.js > p2 > <function>
1 const p1 = new Promise((resolve, reject) => {
2 //setTimeout(() => resolve("P1 Sucess"), 3000);
3 setTimeout(() => reject("P1 Fail"), 3000);
4 });
5
6 const p2 = new Promise((resolve, reject) => {
7 //setTimeout(() => resolve("P2 Sucess"), 5000);
8 setTimeout(() => reject("P2 Fail"), 5000);
9 });
10
11 const p3 = new Promise((resolve, reject) => {
12 //setTimeout(() => resolve("P3 Sucess"), 2000);
13 setTimeout(() => reject("P3 Fail"), 2000);
14 });
15
16 Promise.any([p1, p2, p3])
17 .then((res) => {
18 console.log(res);
19 })
20 .catch((err) => {
21 console.error(err);
22 });
23
Document
127.0.0.1:5500
Namaste JavaScript
Elements
Console
Sources
top
Filter
No Issues
AggregateError: All promises were rejected
(anonymous) @ index.js:21
Promise.catch (async)
(anonymous) @ index.js:20


index.js
index.js > p3 > <function> > setTimeout() callback
1 const p1 = new Promise((resolve, reject) => {
2 setTimeout(() => resolve("P1 Sucess"), 3000);
3 });
4
5 const p2 = new Promise((resolve, reject) => {
6 setTimeout(() => resolve("P2 Sucess"), 5000);
7 //setTimeout(() => reject("P2 Fail"), 1000);
8 });
9
10 const p3 = new Promise((resolve, reject) => {
11 //setTimeout(() => resolve("P3 Sucess"), 2000);
12 setTimeout(() => reject("P3 Fail"), 2000);
13 });
14
15 Promise.race([p1, p2, p3])
16 .then((res) => {
17 console.log(res);
18 })
19 .catch((err) => {
20 console.error(err);
21 });
22
Document
127.0.0.1:5500
Namaste JavaScri
Elements
top
No Issues
P3 Fail