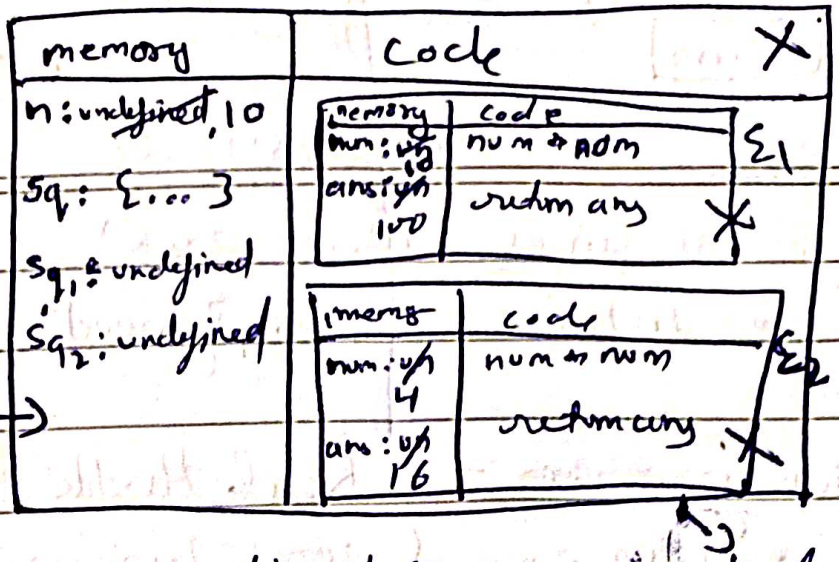
JS: Everything in Js is done in Execution context.
Js is synchronous, single threaded language.

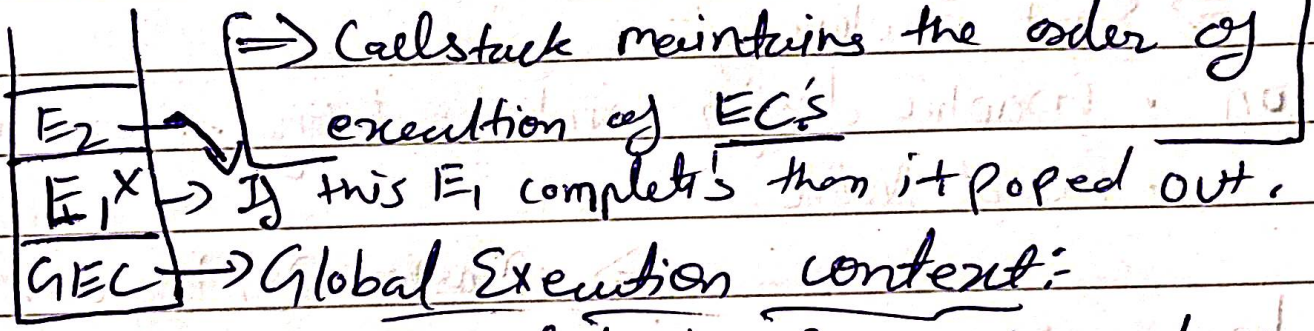
Ex → var n = 10
fn sq(num) {
var ans = num * num
return ans
}
var sq1 = sq(n)
var sq2 = sq(4)

Variable Env →
Global Execution context →



1. In EC first phase is memory creation phase.
2. Second is code execution phase
3. When the EC complete the whole code the it automatically delete ourself.
4. All the EC's are handle in callstack

Ex:-



when all the EC's complete the GEC get poped out, and call stack become empty.

→ call stack has so many names:- 1. Execution context Stack 2. Program stack, 3. Control stack, 4 Runtime Stack, 5. Machine stack.

(A) Hoisting:- This is basically means call the function Before initializing it.

=> Hoisting is not working on variable & arrow based functions.

Ex: var name = () => { } } so here name is behave like a
var name = function() { } } variable, and we get function is not defined error.

Ex: function name() { } → hoisting working on this function initilization.

(B) The way of finding the variable in all available Execution context and switching b/w the lexical environment of context's upto Global context is called scope chain.

(C) Lexical Env:- This is nothing but the, storing the local memory + reference of in the memory of the other (previous) Context/s is called lexical Env.
(parent)

(d) Block scope:- If we want to use multiple statement then we use Block scope { ---}. Scope basically means the all the variable's those we can access inside this block is called block scope.

(e) Clouscers:- clouser is basically a function that binds together with there lexical environment.

OR

function along with its lexical scope (env) bundled together forms a clousers. It is also used for data hiding

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

(output)

```
fn y() {
  console.log(a)
}
```

Module design pattern
Currying
function like once
memoize
maintaining state in async world
setTimeouts
Iterators

⇒ Interview questions :-

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

5 5 5 5 5 5

⇒ Now you have achieve output [0,1,2,3,4] with or without let :-

```
fn x() {
  for (let i=0; i<=5; i++) {
    setTimout(fn() {
      console.log(i)
    }, i*1000)
  }
}
x();
```

0 1 2 3 4

why ⇒ The reason is everytime when the loop iterate's let creates its ~~value~~

new instaces.

Now we have to create a new instance with Var :-

```
fn x() {
  for (var i=0; i<=5; i++) {
    fn z(a) {
      setTimout (fn() {
        console.log (a)
      }, a*1000)
    }
    z(i)
  }
  x()
```

0
1
2
3
4

In this case everytime var i creates a new instance.

Garbage collector :- Garbage collector is basically a program in the browser which freeze out the memory, mainly unused variable's.

(f) functions :-

(1) function statement :- → fn a() {
aka function decleration :- → console.log ("a")
}
a()

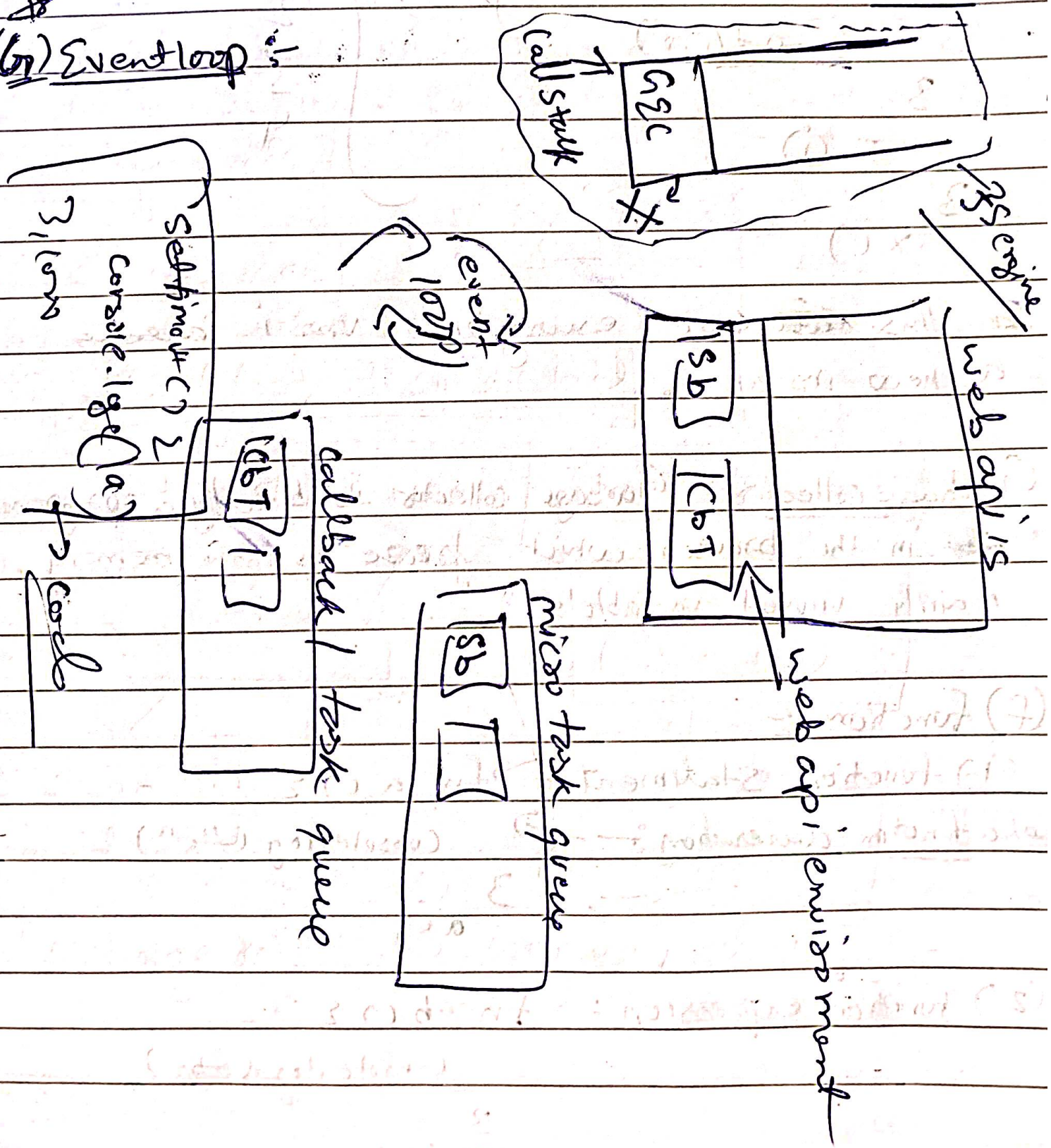
(2) function expression :- fn b() {
console.log ("b")
}
b()

The main difference b/w them is hoisting.

(3) Anonymous functions :- Anonymous function's are used when the function used as values
fn() {}

(4) first class function :- Passing one function to another
Callbacks :- → function as an argument.
or
Ability to be used like values

(5) Eventloop :-



Browser web APIs :- Settimeout, console, fetch(), Dom API, (global) window → location, localstorage,

Ex of callback (settimeout)

So, basically call stack is used for code and create the execution context according to the present code.

Now first line get execute we get (start) in the console, Now when JS engine moves to the second line they got web api which set timout, Now this goes into the web api envisoment, Now cbT() fn waits for set time expire so that it moves to the callback queue and same for fetch because fetch is also a web api but fetch is move to micro task queue.

```
console.log (start)
setTimout (fn cbT() {
    console.log (a)
}, 1000)
fetch ("____").then(fn
    sb() {console.log (b)
});
console.log (end)
```

output :-

start.
End.
fetch data (b)
(a)

why → because every promise or function obscever is handled by micro task queue and micro task queue is the first priority of event loop.

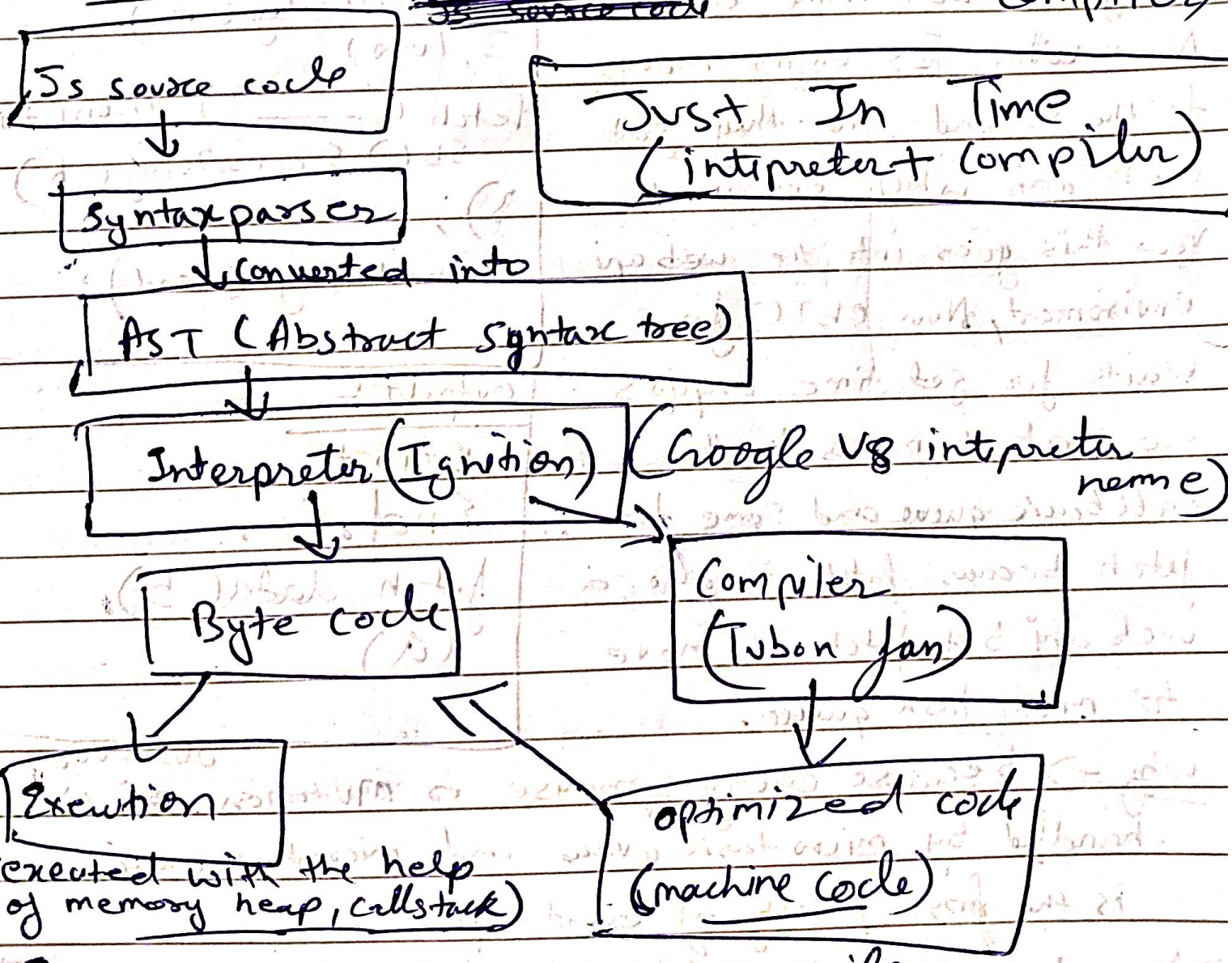
→ Now both the callback (timout, fetch) is Now moved to the queue's, but the last is remaining in the CSEC, which console.log (end) so end get print.

Now event loop is waiting for call stack to get empty so that they can push the micro task queue function (sb) to the callstack and then it get executed, this is same for callback / task queue function (CbT), event loop waits for call stack to get empty so that it can push the function / task in the call stack, that's how event loop works.

(H) Js engine :- V8 (Js is JIT compiler)

Js source code

Just In Time (interpreter + compiler)



Garbage collector → Orinoco, oilpan.
use mark & sweep algo

Optimization technique :- copy elision, inline caching inlining,

(I) HOC → A function which takes an another function as an argument or returns a function from it. (map, filter, reduce etc)

(J) Inversion of control :- when we use callback's, and create callback hell, then we loss control on the code.

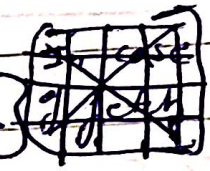
Reason ⇒ Because we pass one function to another fn to another fn give the control to other function.

(K) Promise :- Promise is an object representing the eventual completion or failure of an async operations.

(Promises are immutable)

⇒ Promise gives 3 things → Prototype → Promise
Promise State → Pending
Promise Result → Data
initially undefined

⇒ 3 main states :-
⇒ Pending, fulfilled, Reject.



{while chaining we have to return the promise from the promise.}

⇒ Promise chaining → Promise chaining is a syntax that allow you to chain together multiple async operations takes in a specific order.

⇒ If we want to get rid of promise chaining, we can use async/await.

⇒ Async/await always return promise and await is always use inside the async function.

⇒ Promise Api's :- 1. Promise.all() , 2. Promise.allSettled()
3. Promise.race() 4. Promise.any() .

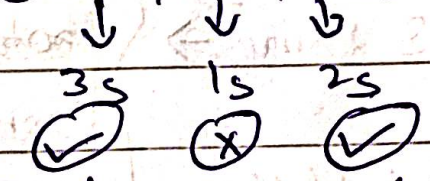
1. Promise.all ⇒ It handles multiple promises or multiple Api's. It takes iterables (array) of promises.

[⇒ promise.all ([P1, P2])] ⇒ It always wait for all of them to finish.

⇒ In case of fulfilled state it return's an array of promises.
⇒ But in case of reject promise.all get reject and throw the error of rejected promise.

2. Promise.allsettled :- This is same as promise.all in case of fulfilled, but in case of error it waits for promises to get settled.

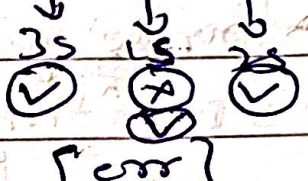
Ex promise.allsettled ([P1, P2, P3])



Now it returns :- [val1, err, val2]

Promise.all()

([P1, P2, P3])

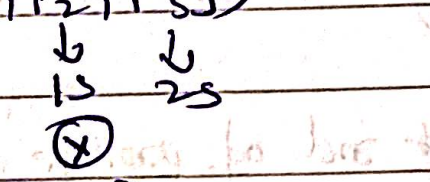


[err]

[val1, val2, val3]

3. Promise.race () :- It basically returned the first settled promise weather it is fulfilled or rejected

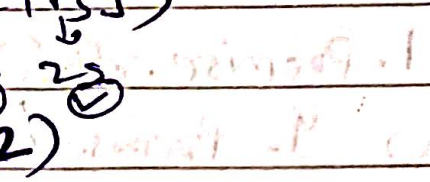
Ex promise.race ([P1, P2, P3])



returns → (err)

4. Promise.any :- It basically returned the first a success promise settled

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



returns ⇒ (val2)

In case all promises get rejected and throw errors, then the error is Aggregate error, basically means the array of all the errors [err1, err2, err3].

Hoisting main pts :- when you declare the variable using the var keyword, it undergoes a process called "hoisting". During hoisting, variable declaration moved to the top of their containing scope but not their assignments, so the variable 'a' is hoisted to the top of its scope, but its value is not assigned until the line, where you set it to 5.

```
clg (a)
clg (b)
var a = 5;
var b = 5
```

→ Declaration var a;

→ assignment a=5

L) Call, Apply, Bind ⇒ Call is a function that helps you change the context of the invoking function. In other words, it helps you replace the value of "this" inside a function with whatever value you want.

(or)

call method allow an object to use the method (function) of another object.

Apply ⇒ only difference in syntax.

Bind ⇒ Gives the copy of the function.

const P1 = {

fname → Appu

lname → Gupta → hometown, country

fullname → () => {

return this.fname + " " + this.lname + " " hometown + " " + country

2

const P2 = {

fname = Appu

lname → guptaa

3

call ⇒ P1.fullname.call(P2) → {Appu gupta} output

with parameter ⇒ P1.fullname.call(P2, "Alwar", "India")

output → (Appu gupta Alwar India)

Apply ⇒ P1.fullname.apply(P2, ["Alwar", "India"])

same

Bind ⇒ Bind returns a function so that you can you anywhere.

const result = P1.fullname.Bind(P2, "Alwar", "India")

ctg (result()) → same output

(m) ⇒ Arrow function this always takes a values from its parent scope (fn) this value.

⇒ Normal function this represent the global / window.

Interview Question :-

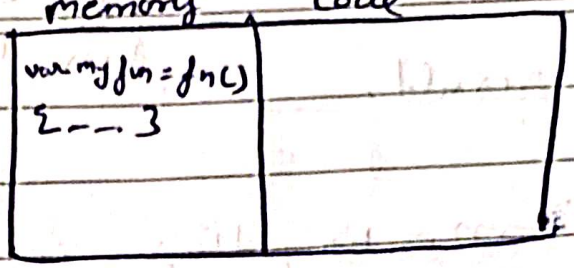
```
myFun()

var myFun() = fn() {
    clg(first)
}
myFun()

fn myFun() {
    clg(second)
}
myFun()
```

[second, first, first] why?

when the function is calling back and forth then always remember Global Execution Context



Now initally when the EC is created the memory phase starts, then var myFun is declared as undefined after that we created a function with same name so the variable myFun is now assigned to a function so that why it console the value of that function then goes to the variable based function.

(2)

```
const obj = {
    height : 10
}
clg(obj.height) → 10
delete obj.height
clg(obj.height) → undefined
```

```
const obj = Object.create({
    height : 10
})
clg(obj.height) → 10
delete obj.height
clg(obj.height) → 10
```

Now in this object.create the object and when we assign the value height to, it assigned in prototype inside the object, that why delete would'nt work.

JS is a interpreted language before V8 engine, but after V8 engine JS is JIT (JIT).

⇒ Interpeter → it means, interpeter change the code to the machine code (byte code), and then (line by line) give the result.

⇒ Compiler → it means, this compile the whole code and change to machine code and then gives the result.

⇒ But in compiler the error comes in compile time, but in interpeter the error comes in runtime.

⇒ Compiler is fast because it changes the whole code in once in respect to intepeter.

⇒ But Now JS in JIT, it basically means the combination of both interpeter & compiler.

⇒ REPL → Read Eval Print Loop.

Ways to convert String to Numbers

1. clg(+'10') → NAN
2. clg(Number('10')) → NAN
3. clg(parseInt('10')) → 10
   clg(parseInt('a10')) → NAN

✱ Strings Methods & properties :-

⇒ Properties :-

1. Length

⇒ Method (without argument) :-

1. toUppercase() , 2. toLowercase()
3. trim() , 4. trimStart() , 5. trimEnd().

⇒ Method (with argument) :- const a = ('Hello world')

1. Includes () ⇒ a.includes('Hello') ⇒ true/false
2. indexof() ⇒ a.indexof('e') ⇒ 2 / -1
   - 2 ↙ index of available char
   - -1 ↘ if not present
3. replace () ⇒ a.replace('Hello', 'Hi')
   - 'Hello' ↙ Jisko replace kona h
   - 'Hi' ↘ Jisse replace kona h.

   output ('Hi world') ⇒ (only change first match char)

4. replaceAll () ⇒ change all the matching char.
5. Concat() ⇒ Add multiple strings.

   const a = 'Hi'

   b = 'Hello'

   a.concat(b) → HiHello

   a.concat(' ', b) → Hi Hello

6. Padstart() ⇒ Mask Number / Hide Number

   const a = 1234

   a.padstart(16, '*') ⇒ ************1234

   ↓ This value is for, total number of output (final string length)

7. padEnd() => Same as padStart, but do at end.

a.padEnd(16,'A') => 1234 A A A A A A A A A A A A

8. charAT() => a = "Hello"

a.charAT(3) -> l

9. split() => a = 'Apurv Gupta'

a.split() => ~~['Apurv', 'Gupta']~~ ['Apurv Gupta']

a.split(' ') => ['Apurv', 'Gupta']

a.split('u') => ~~['Ap', 'rv', 'G', 'pta']~~ X

['Ap', 'rv G', 'pta']