Async-2 : Promises

Agenda →

1) Callbacks
2) Promises
3) Async Programming
4) Chaining of promises
5) Async task in Concurrent order

→ CALLBACKS :

$$\begin{tikzcd}
\text{TASK1 } \checkmark \arrow[r, bend left] & \text{Task 2} \arrow[r, bend right] & \text{Task 3.}
\end{tikzcd}$$

$\boxed{\text{JS}}$ → Functions are objects

Function can be passed as params.

```
function print (cb) {
    - - -
    Console.log (`Print done`)
    - - -
    cb()
}
```

callback

Why ??? → JS

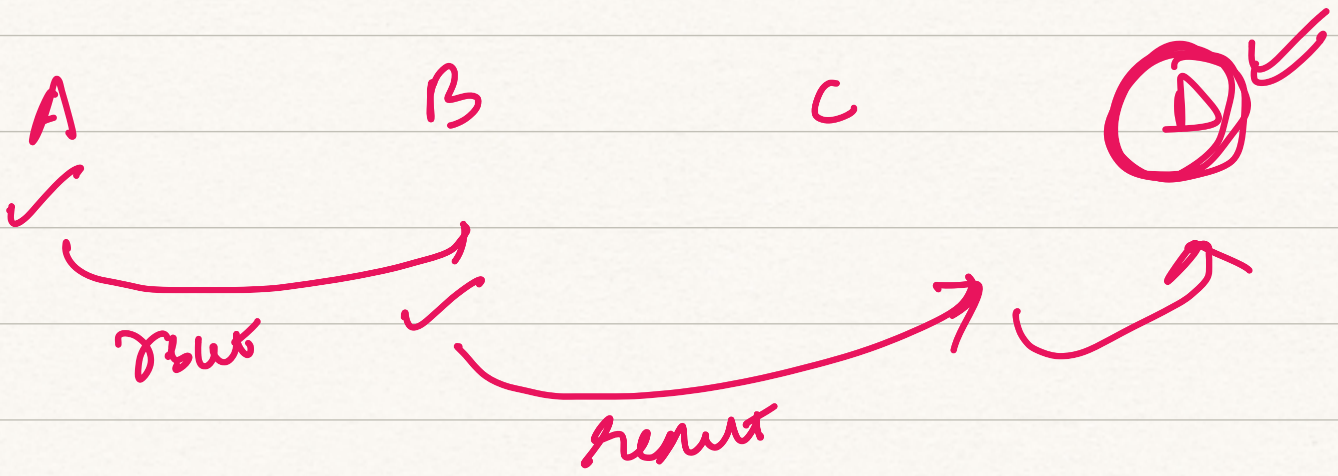
wake
freshenup
ready
drive
reach Office
work

```
function wake ()
    :
    freshenup () {
        :
        ready () {
            :
            drive ()
                :
                - - -
```

Cb → makes sense for " dont run until task is completed.

CALLBACK HELL

→ We need multiple callbacks (nesting callbacks) within a function.



```
getArticles ( )     {
        //result after API call        [user]   ↖result

            getUser.Data (user)     {
                            // [Name] (result)
                            get Address (name)
                                  - - ..
                                      getPinCode ( ) {
                                          ——.
                     }
```

Pyramid of Doom

(Callback Hell)

⇒ PROMISES

{ }

Representing the eventual completion or a failure of given ASYNC operation.

Completed successfully

failed.

Pending

Promise

Resolved (fulfilled)

Rejected (Failed)

SETTLED
Complete

Promise
Settled → This means promise is Completed. Can be — **FULFILLED** or **REJECTED**

States of a promise

1) PENDING (made a promise)
2) RESOLVED (fulfilled)
3) REJECTED (Failed. Can't be fulfilled)
4) SETTLED (Executed)

Promise.

① CREATE (refer code)

```
let p = new Promise (function (res, rej){
        . ——res( )   //sucess
            rej ( )   //FAILED.
    }
)
```

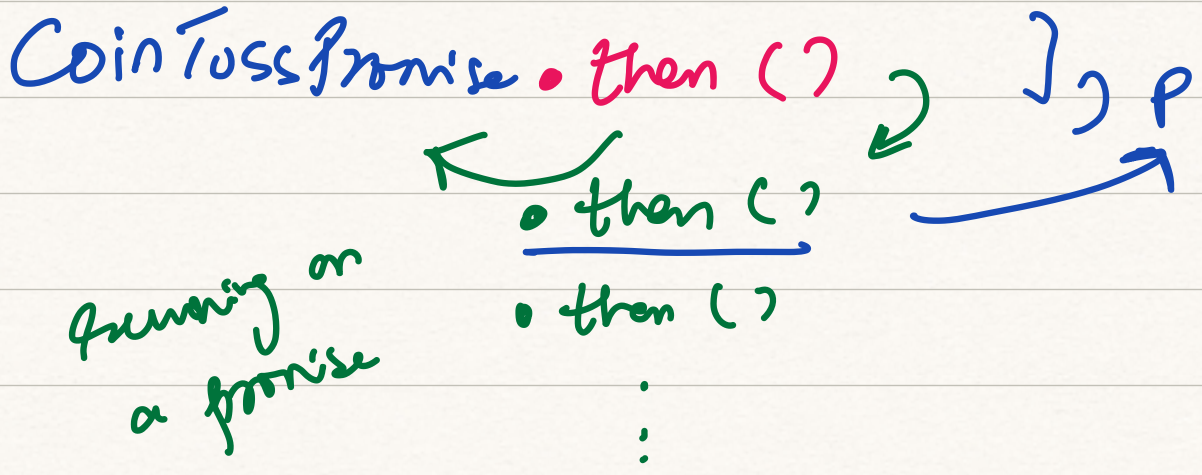
② CONSUME a promise :

To use promise. we attach Callbacks using

- then() → (resolve) [successful] (Callback for success)
- catch() → Handle errors. (rejected).
- finally() ↳ To execute when promise is settled. No matter the outcome of the promise

What does then / Catch return ??

They return a promise



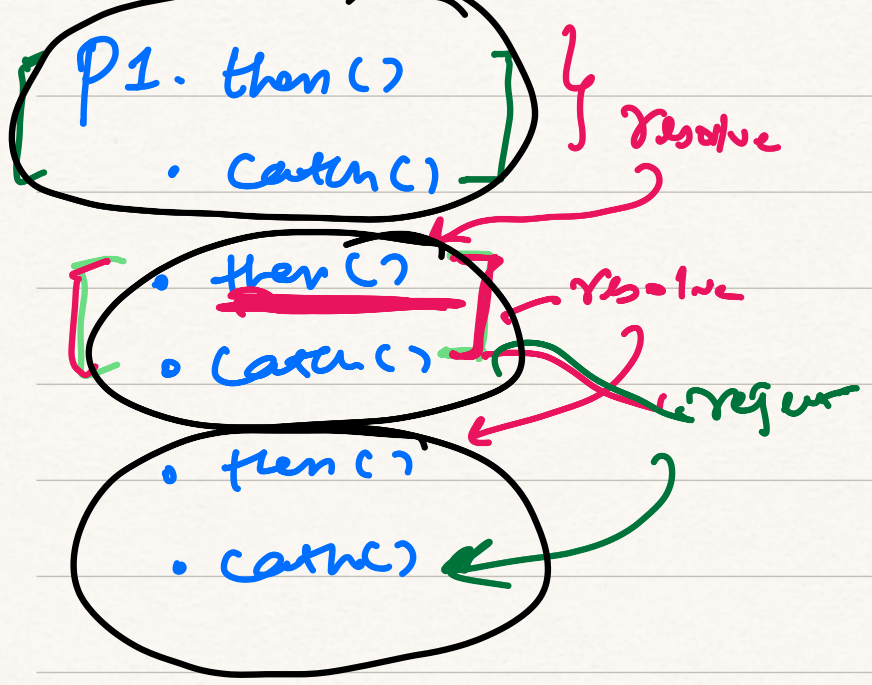
CHAINING OF PROMISES

- .then method returns a new promise, which allows for sequential execution of async operation.

$$\begin{tikzcd}
\text{✓ Clean my room ✓} \arrow[r] & \text{Chocolate} \\
\text{✓ Collect garbage ✓} \arrow[ru] & \\
\text{✓ do H.W. ✓} \arrow[ruu] &
\end{tikzcd}$$

In Chaining, if any one of promises fails, other 'then' are skipped & control goes to 'catch' block.

p.then()	P	F	P
.then()	F	Skipped	P
.then()	Skipped	Skipped	F
.catch()	✓	✓	✓



Catch('Hi')

SUCCESS

EVENT LOOP (with promises)

✓ Call Stack
✓ Callback Queue
✓ Web | Node Apis
✓ Event Loop

→ Micro Task
→ Macro Task | Callback Q.

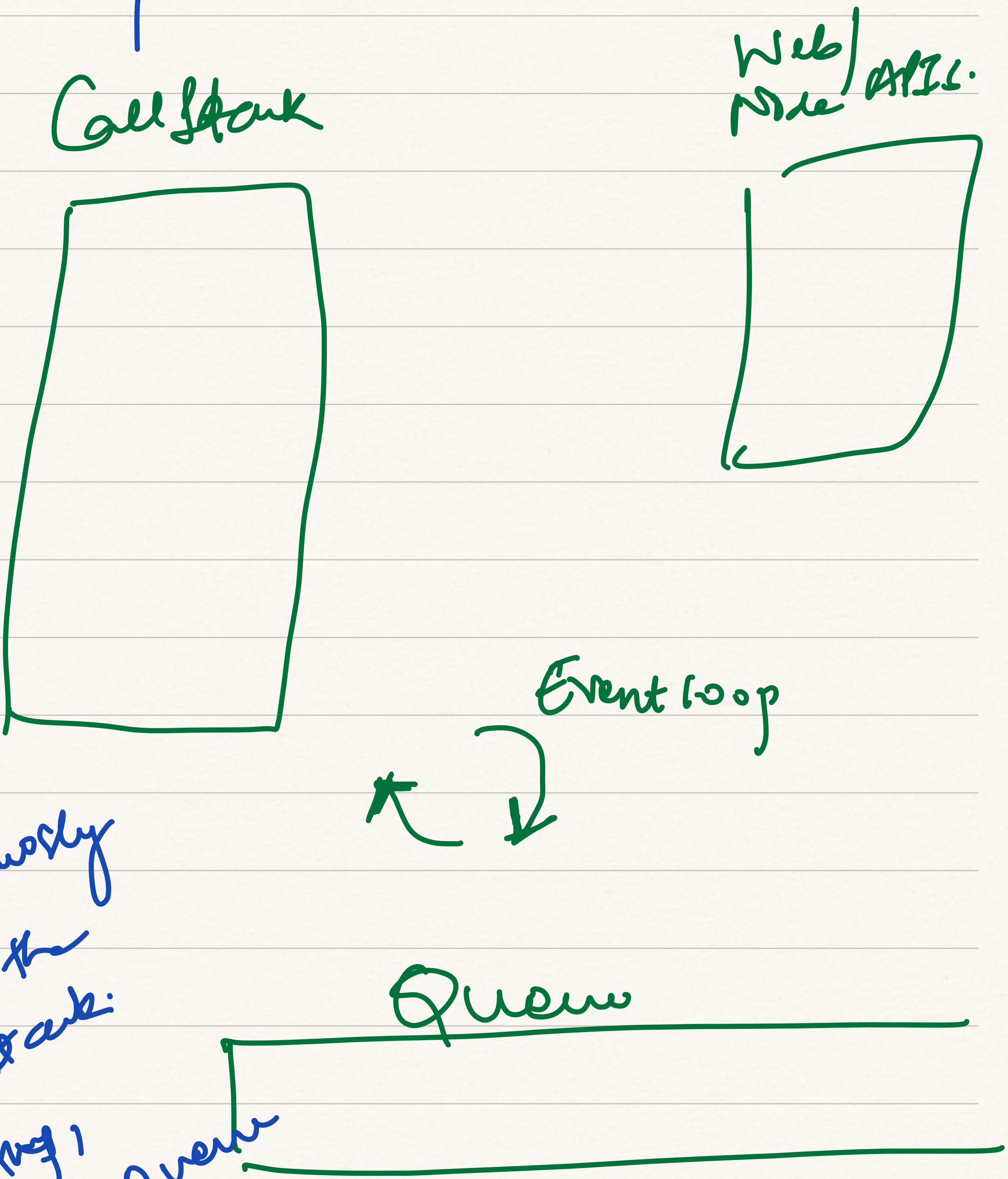
1 Console.log('Start')

2 setTimeout(function(){

3 log.('Hi')

```
}, 2000)

4 log('end')
```



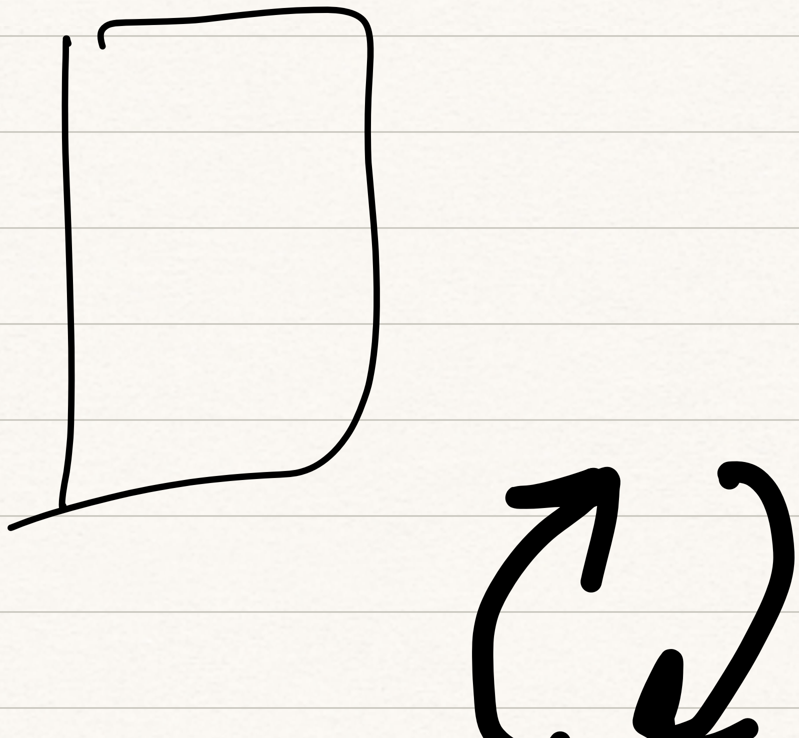
EL continuously checks for the Queue & Stack.

If stack empty, queues task from Queue to stack for execution.

Event Loop → mechanism that allows node / JS engines to perform **non-blocking** I/O tasks despite JS being **single threaded**

What to execute next

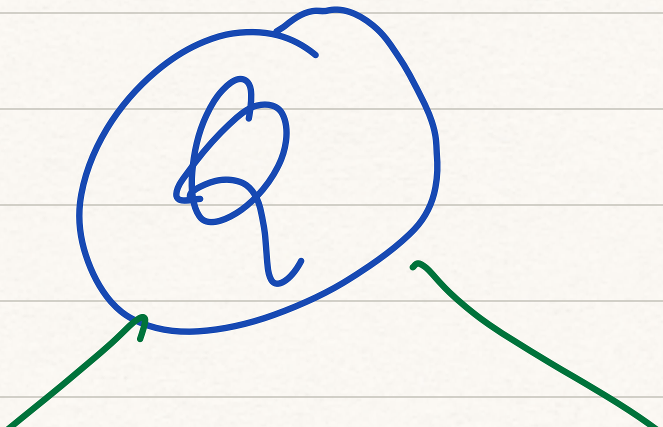
Call Stack



Queue

FIFO

First In First Out



VIP ↓ Microtask Queue

Promises Callbacks

Normal ↓ Macrotask Q

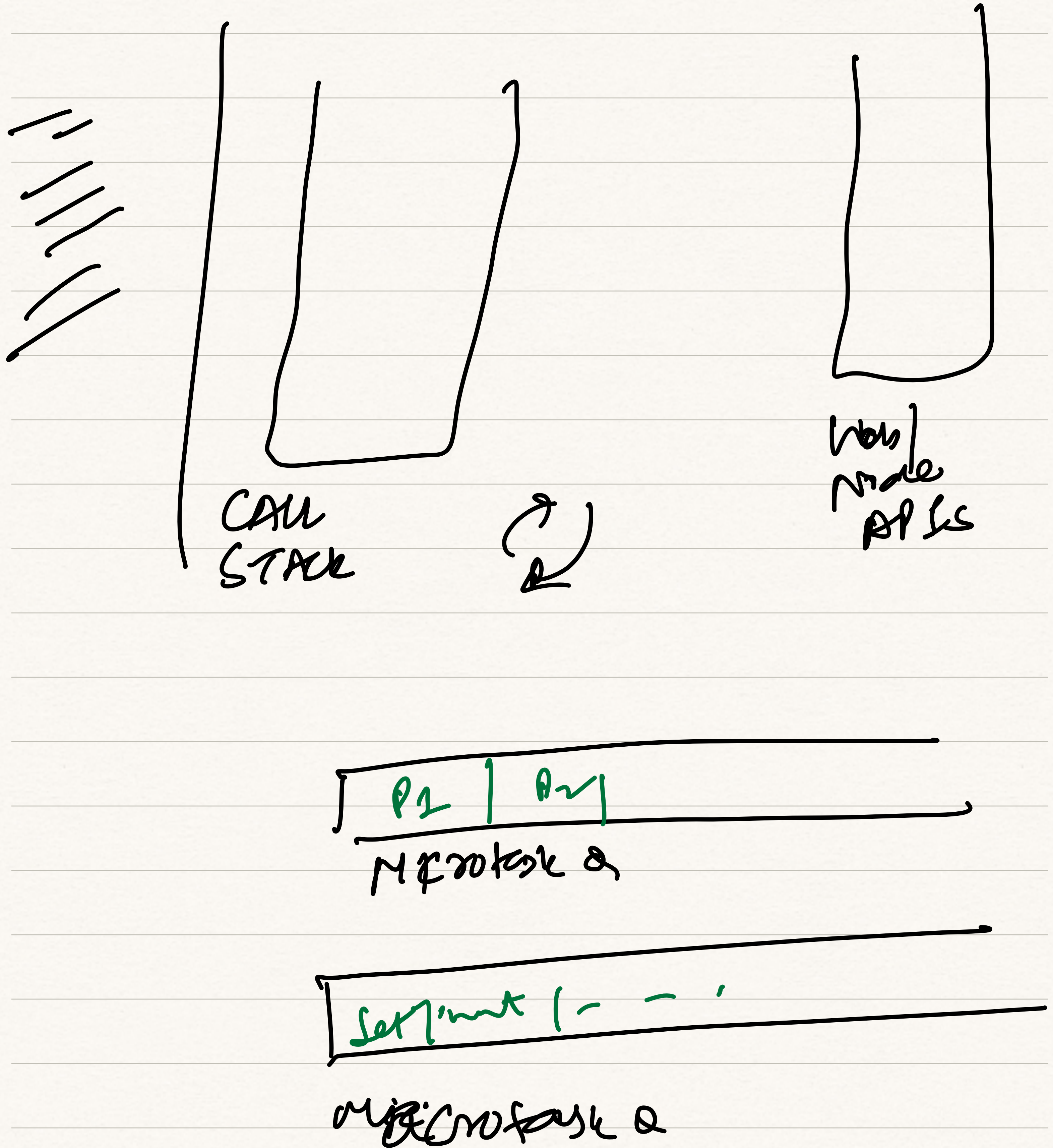
setInterval
setTimeout

mouseOver
I/O callbacks

- Event loop gives higher priority to task in microtask Queue over tasks in

[illegible]

macrotask queue



(Callback Q)

(Job Q) .