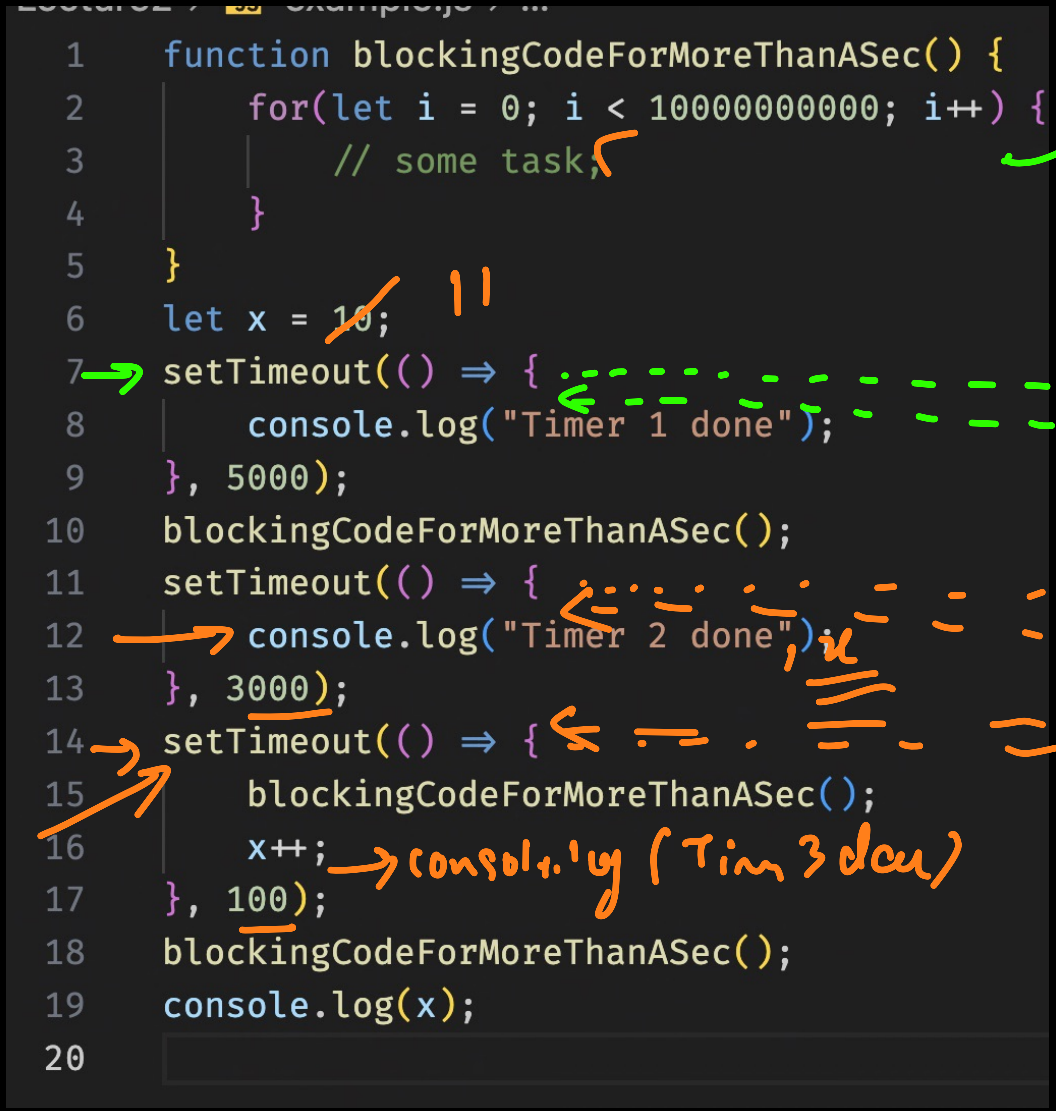
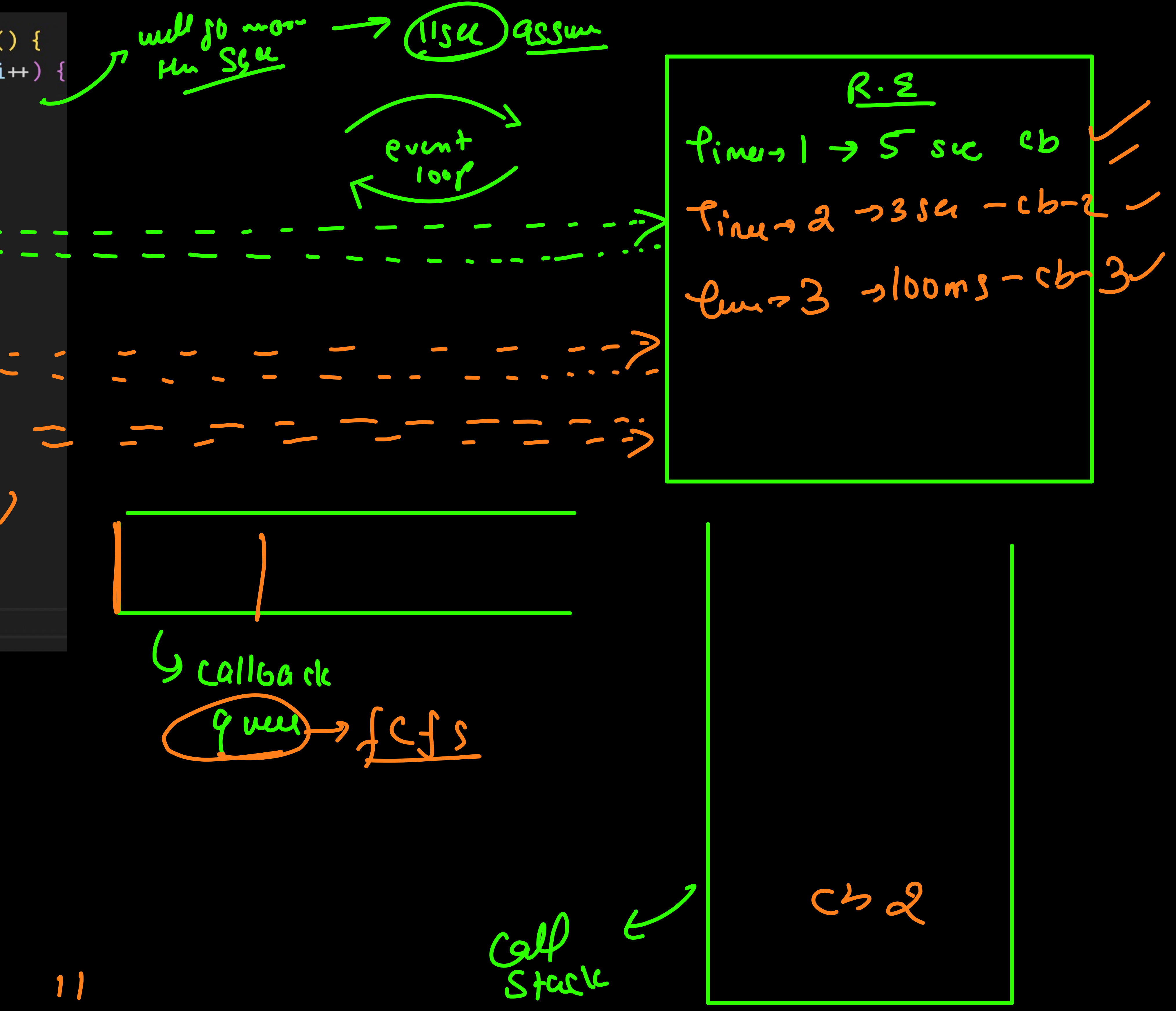
→ example to revise

→ what the issues with current async behaviour

→ Resolve issues → **Promises**



```
function blockingCodeForMoreThanASec() {
    for(let i = 0; i < 10000000000; i++) {
        // some task;
    }
}
let x = 10;
setTimeout(() => {
    console.log("Timer 1 done");
}, 5000);
blockingCodeForMoreThanASec();
setTimeout(() => {
    console.log("Timer 2 done");
}, 3000);
setTimeout(() => {
    blockingCodeForMoreThanASec();
    x++;
}, 100);
blockingCodeForMoreThanASec();
console.log(x);
```



printed → 10

Timer1 done

Timer3 done

Timer 2 done 11

every async code we discussed is mainly based out of callbacks

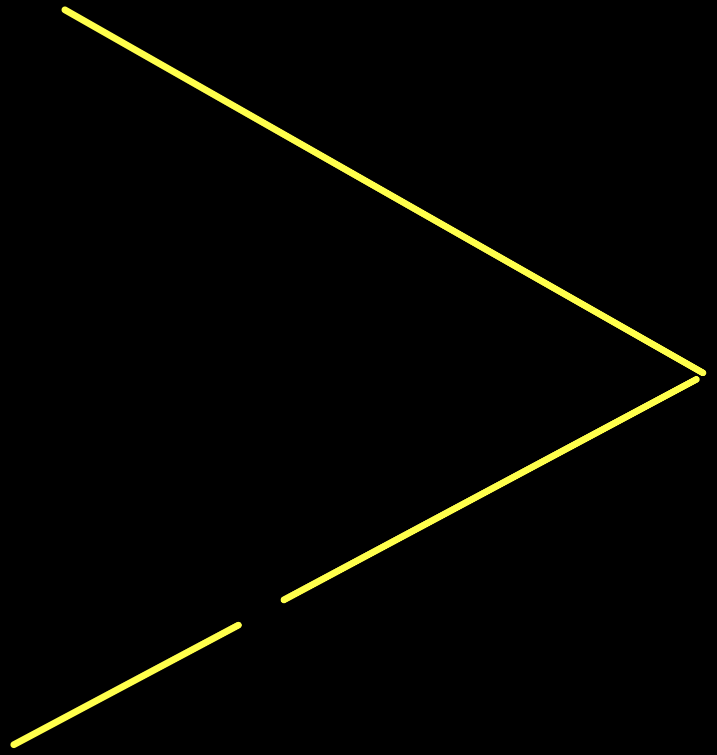
But callbacks can be a bit problematic.

Disadvantages Of Callback Based Codes

#opinion

→ Callback hell → Code readability issue.

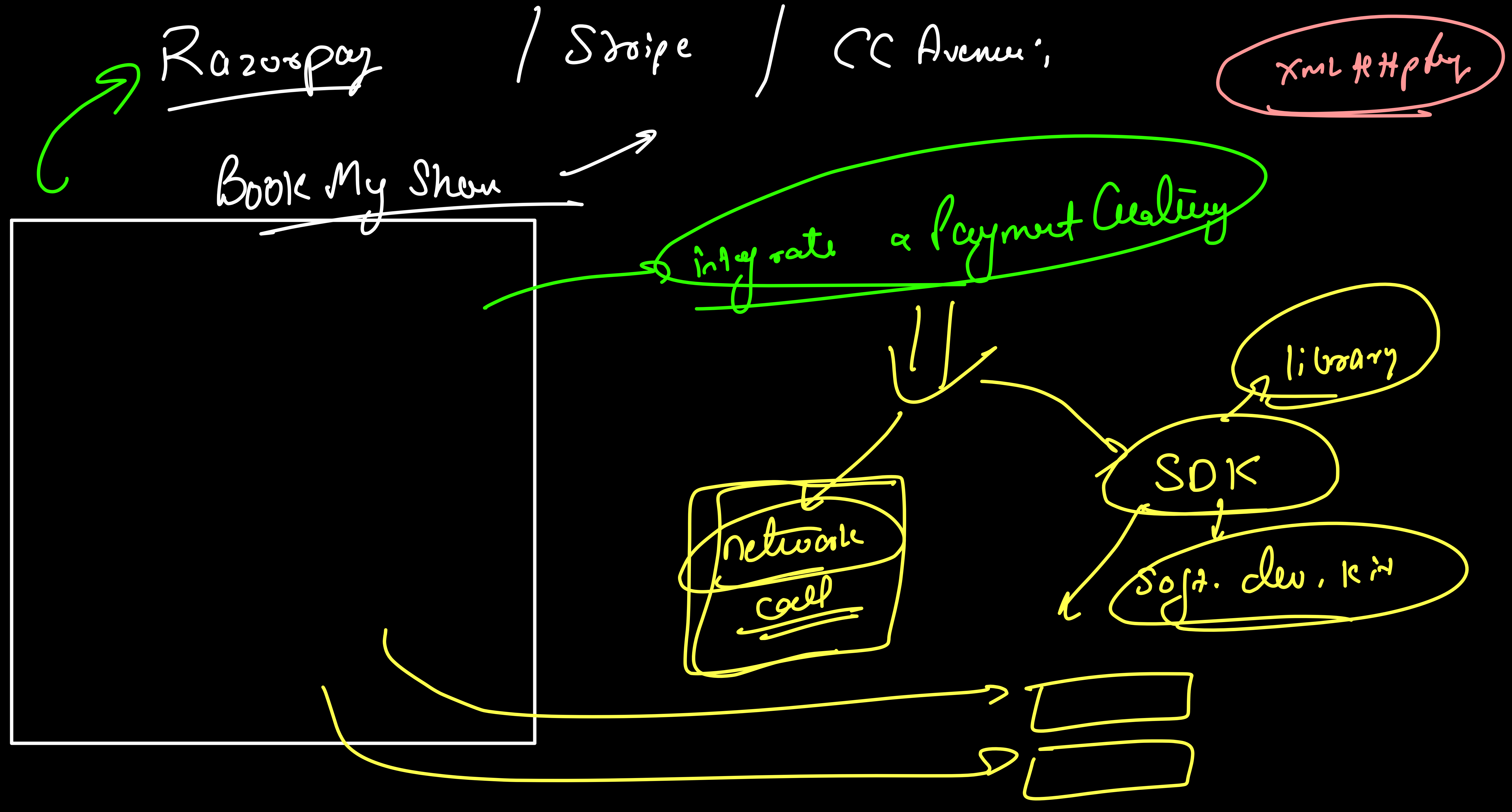
↳ callback insided a call back inside a call back



(IoC)

② Inversion Of Control : you are giving control of your code execution to somebody else.

Razorpay / Stripe / CC Avenue;



SDK of razorpay

callback based

RazorPay Checkout

```
RazorpayCheckout({ ..... },
  () => {

  })
```

```
sign
SDK
Razorpay Checkout (credentials, checkout Cb ) {
    verifies (credentials);
    eckout Cb ();  ← deduct money
  → checkout Cb ();
}
```

OK !! → How to solve it ??

$\boxed{\text{PROMISES}}$

Cb

Promise ✗

Promises

It will be a long road before we understand how Promises solve IoC.

→ What is a Promise in JS ??

- ↳ It is a special JS object.
- ↳ It is a part of native JS lang. (i.e. it is a feature of JS not the Runtime).
- ↳ Promises are considered Readability enhancers as well. (it is slighty more readable than cb)

↳ Just like cb, Promises can also be used with sync or async code.

↳ Promises are also considered placeholders for future tasks.

To understand Promises →

② ├── how to create a Promise ?

① ├── how to use a Promise ?

Assume for some time, you don't need to care about how promise is created.

How to Consume a Promise ??

Properties of promise object :

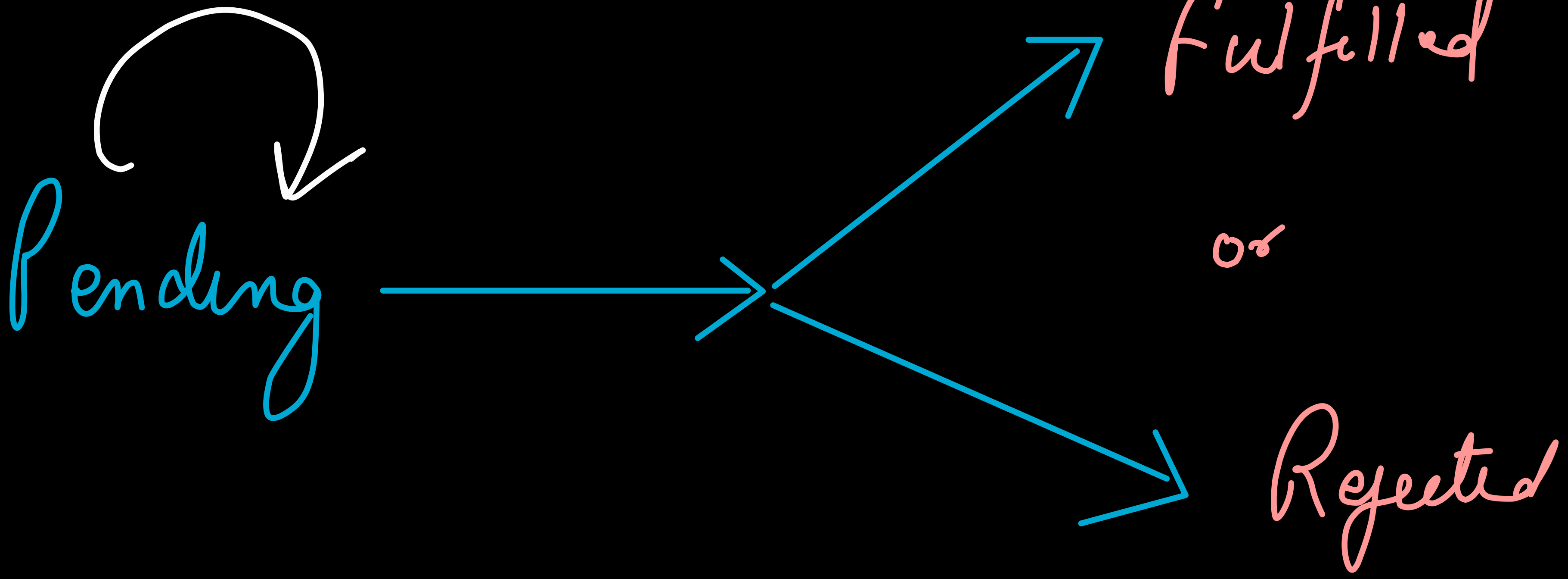
Note: there are other func^n also apart from these properties

→
- status → every promise object can have one of the 3 status: Pending, Fulfilled, Rejected
- value
- onfullfillment
- onRejection

State of promise :

↳ Possible states : Pending , Rejected , fulfilled

↳ the moment we create the promise immediately the Status is "Pending" always.



from pending you can go to fulfilled or rejected Status.

once the promise is either fulfilled OR rejected state, the state cannot change again

→ When will the status change & to what it will change is programmed when promise is <u>created</u>. Consumer of promise donot decide when & how state changes.

→ A promise can be in forever pending state <u>also</u>.

Value of a promise : $\longrightarrow$ PromiseResult

↳ initially when Promise is created, the state is pending & the value property is undefined.

↳ when the state of a promise changes to fulfilled or rejected, then the value property MIGHT-CHANGE.

↳ Value of a promise cannot change without state change. i.e. if promise pending forever, value will be undefined forever.

Q:- if the value of a promise has changed once, can it change again ?? **NO.**

onfulfullment : [f1, f2, f3]

↳ It is an array.

↳ It holds all the funcⁿ which we want to execute once promise state goes from PENDING to FULFILLED. It has nothing to do with rejection state.

↳ Who writes these funcⁿ & registers them in the array?

↳ the consumer of the promise writes the methods & manually register / store them in this array.

↳ the array remains empty until or unless you register / store the first func^n. That means, state of state change doesnot control, when the array is empty.

→ when the func^n's stored in this array ^when it will be executed is controlled by state change.

→ How to register func^n ??
↳ we will discuss in sometime.

onRejected: $\rightarrow$ [f1, f2, f3]

↳ It is everything you learned in onFulfillment just the state change we target is

PENDING $\rightarrow$ Rejected.

note → all the func^n in the onFulfillment & onRejected array takes one & only one argument during their execution.

viz, the value property of the promise object.

how to register functions in the onfulfillment & onRejected array ??

→ On a promise object, we have access to a .then method.

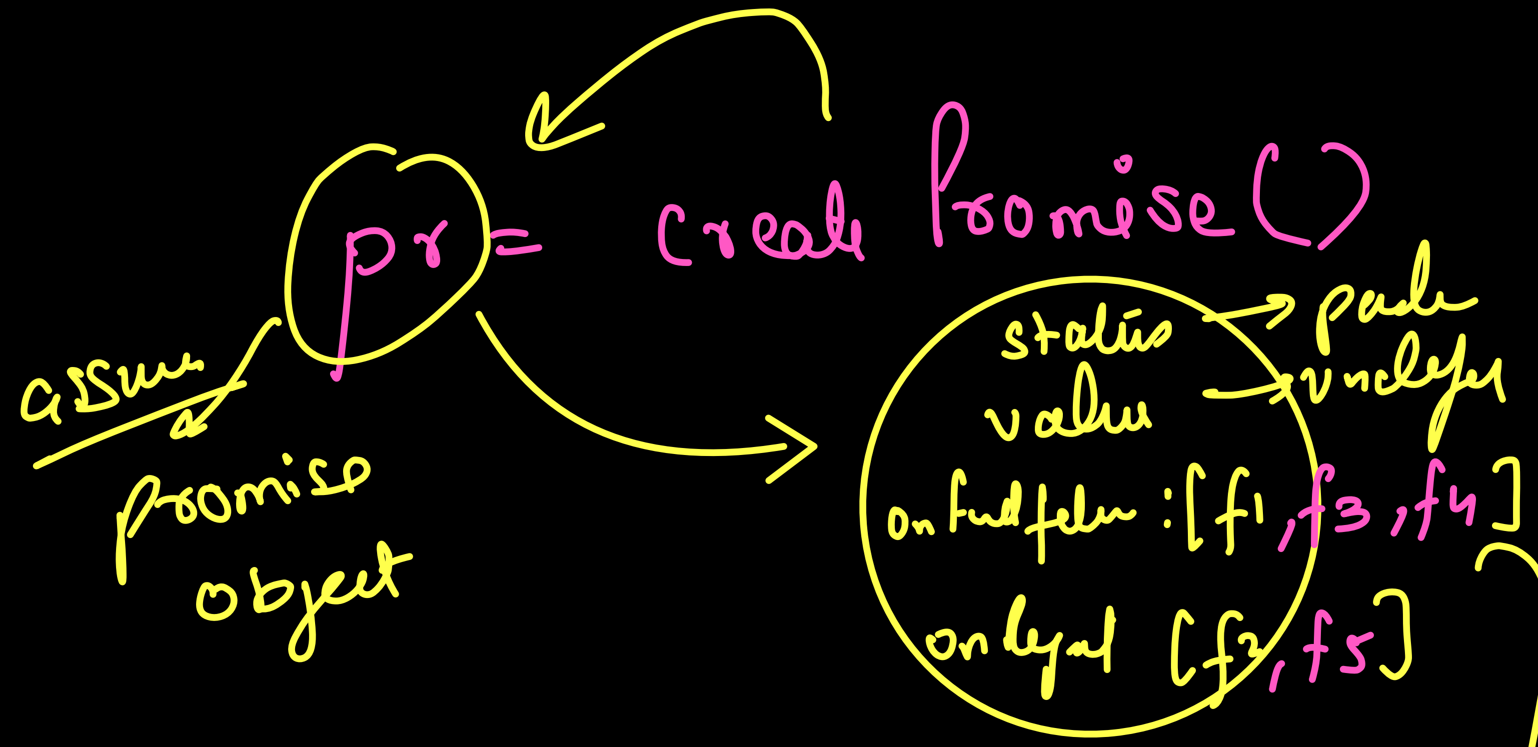
→ .then (success cb, failure cb) optional

this a HoF

because it takes func^n as arg

.then (success cb)

function myfun (—)



pr. then (f1, f2)

pr. then (f3)

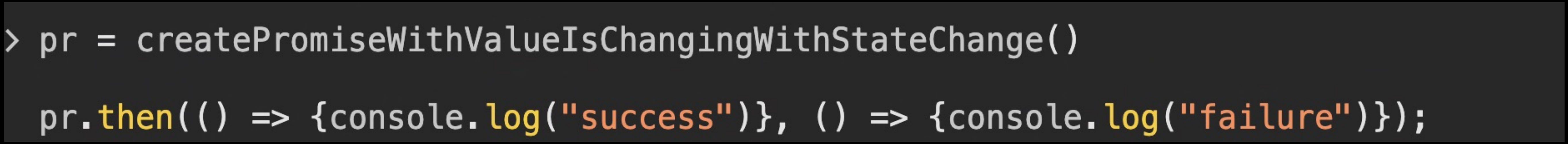
pr. then (f4, f5)

→ multiple times we can call .then on a promise object.

→ the moment we call .then on a promise, the first cb is stored in the onfulfillment array & second is stored in the onRejected array.

Only registration happens while .then, not execution of these func^n

Let's take a code example to see, and validate that .then is not responsible for execution.



```
> pr = createPromiseWithValueIsChangingWithStateChange()

  pr.then(() => {console.log("success")}, () => {console.log("failure")});
```

2

this code donot executes the callback, only registers them.

Execution is not in control of consumer.

Q: if .then is a func^n, what is the return value of it ??

→ .then actually returns a brand new promise object again.

```
pr = createPromise();
```

promise object.

```
promise2 = pr. then (f1, f2)
```

this line return one more promise.

diff than pr

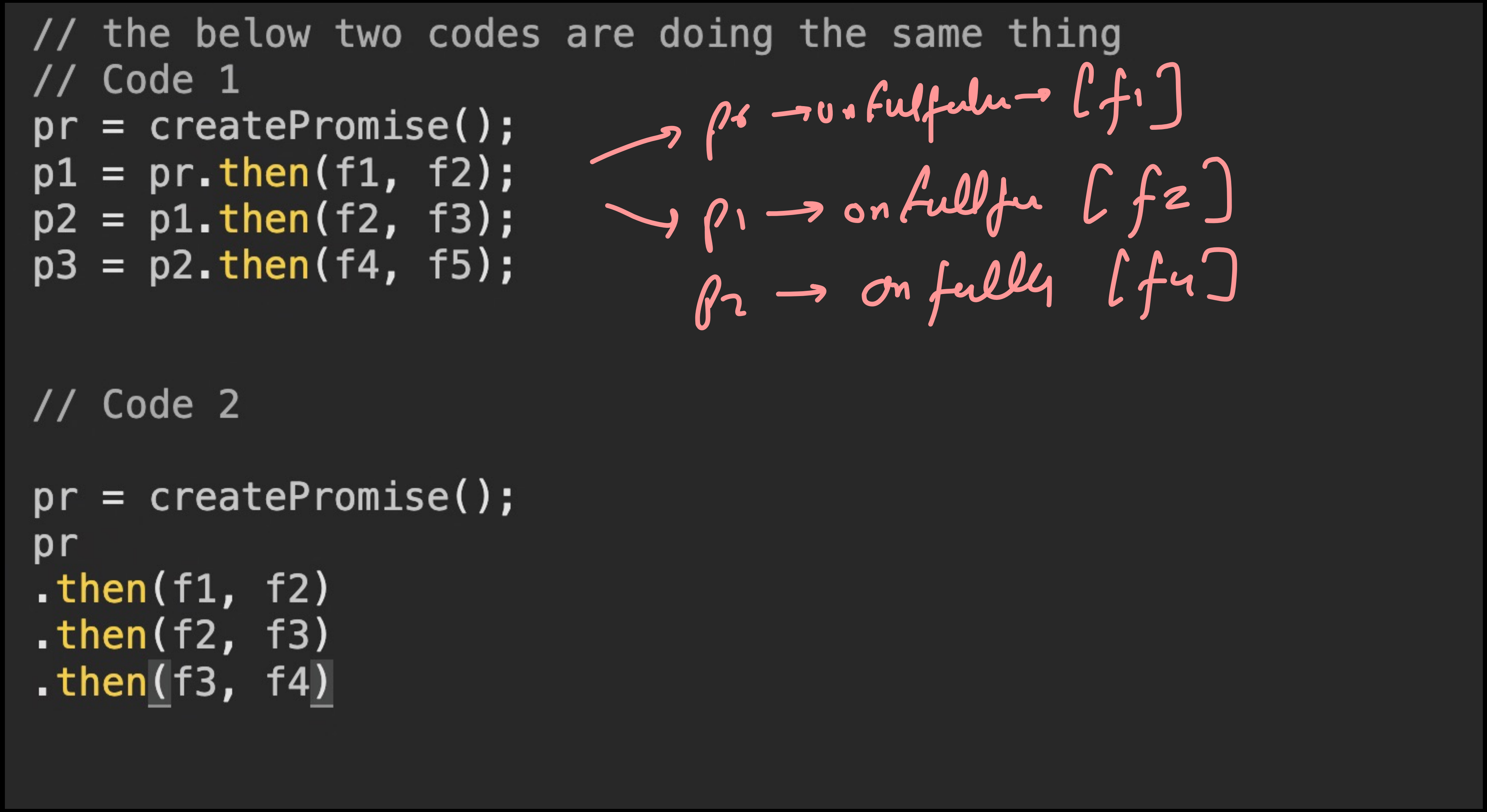
```
pr = createPromise()

pr.then(f1, f2)
  .then(f3, f4)
  .then(f5, f6)
```

registration of f1, f2 on pr

registers f3, f4 on the promise return in the prev line

.then chaining

↓
is applying registration of f1, f6 on the promise value in prev line.



```
// the below two codes are doing the same thing
// Code 1
pr = createPromise();
p1 = pr.then(f1, f2);
p2 = p1.then(f2, f3);
p3 = p2.then(f4, f5);


// Code 2

pr = createPromise();
pr
.then(f1, f2)
.then(f2, f3)
.then(f3, f4)
```

pr → onFulfillment → [f1]

p1 → onFullfill [f2]

p2 → onFullfill [f4]

and above two code are diff from this below code.

```
pr = createPromise();
pr.then(f1, f2);
pr.then(f2, f3);
pr.then(f4, f5);
```

pr. then (f1 , f2)

① → f1 is registerd in onFulfillment of pr
② f2 → is registerd in onRejected of pr
③ this line returns a brand new promise obj independent of ~~pr.~~
④ whenever f1 or f2 will be executed then value property of pr is passed as an arg.

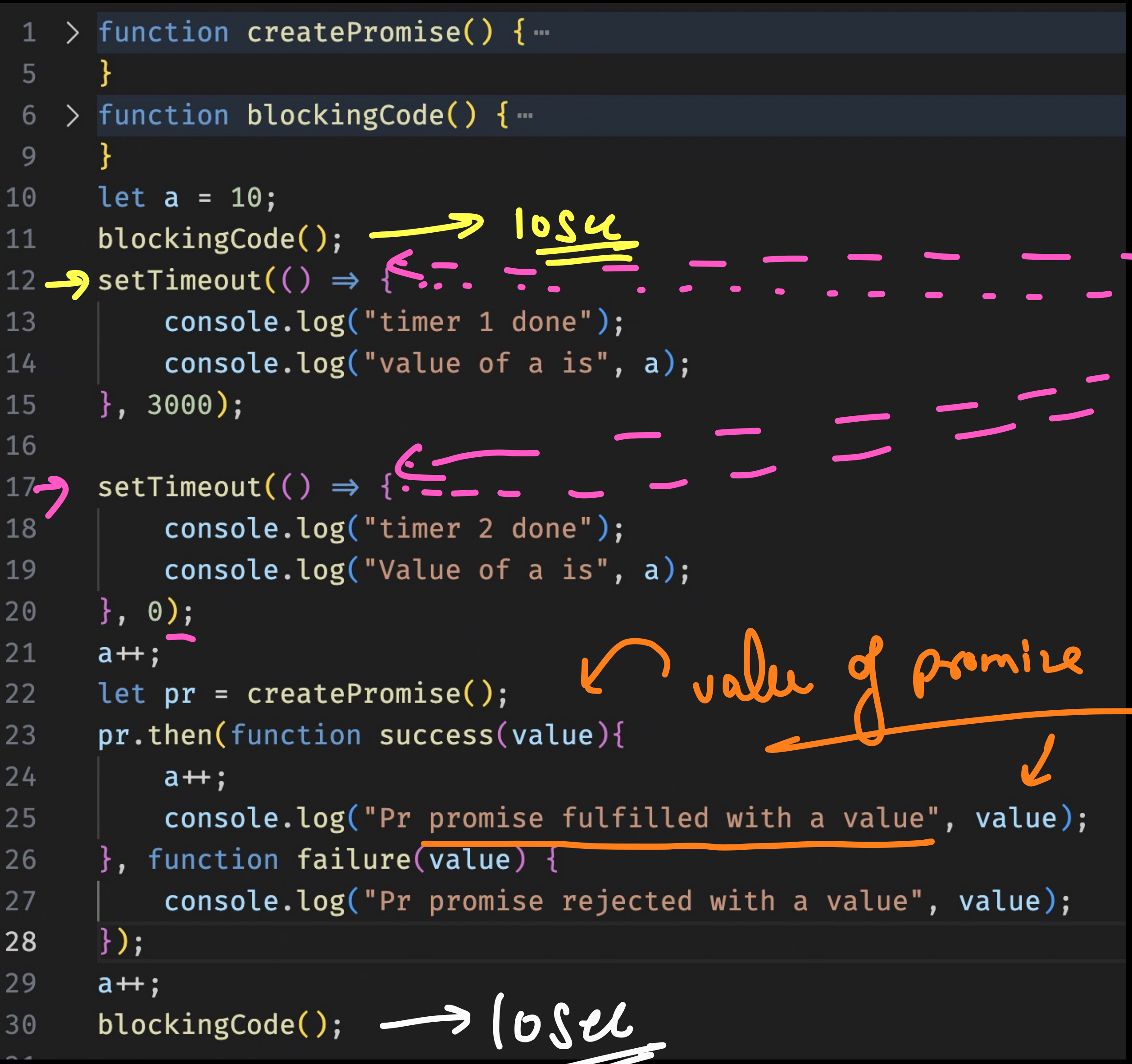
how execution of promise happens.

→ with cb, an cb coming from R.E. used to wait in the callback queue.

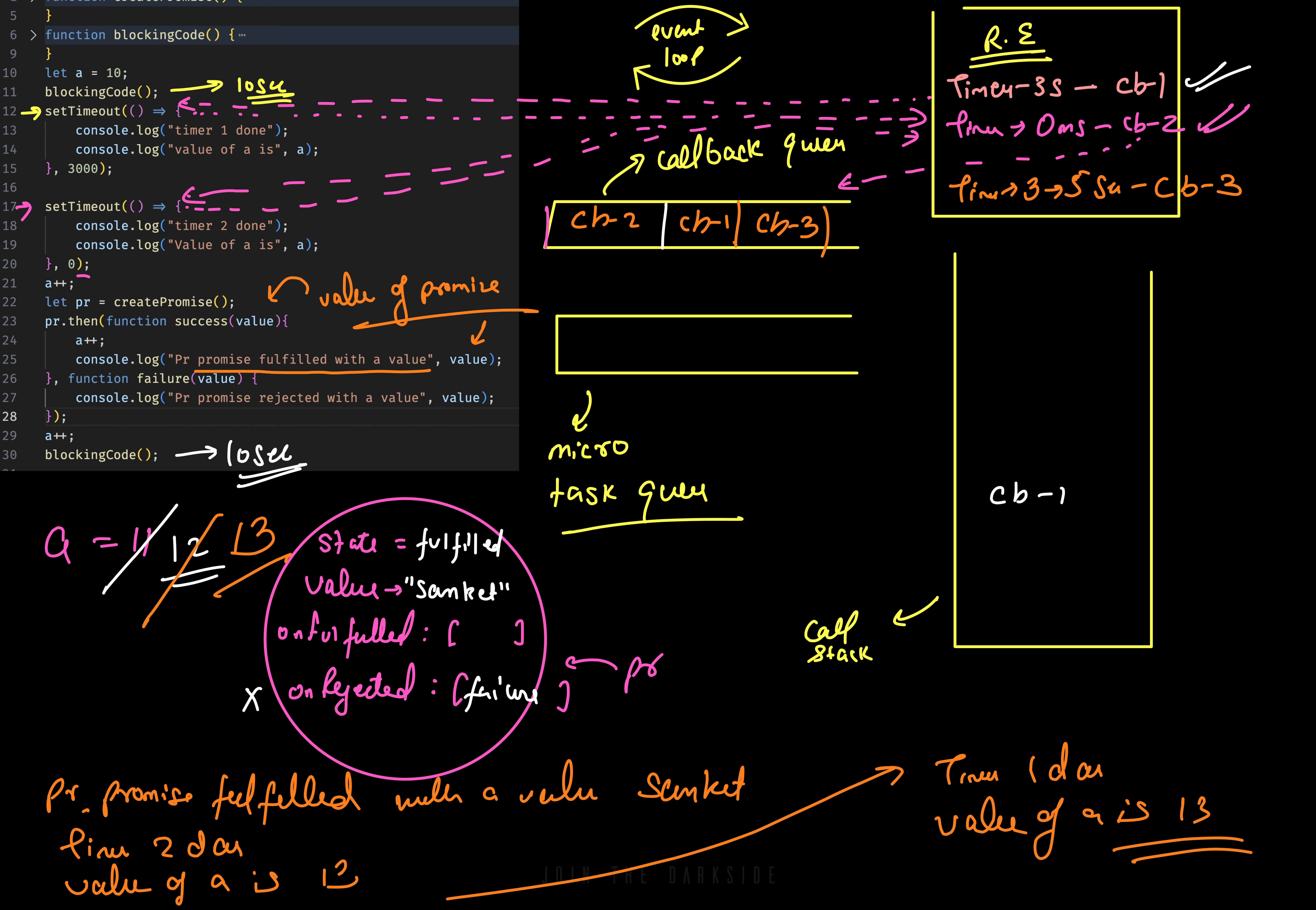
→ Apart from callback queue, we have a microtask queue also.

internall
↓
setTimeout

this create promise method create a promise object which after 5 sec goes from pending → fulfilled state - while value going from undefined → "Sanket"



```
function createPromise() {…
}
function blockingCode() {…
}
let a = 10;
blockingCode();
setTimeout(() => {
    console.log("timer 1 done");
    console.log("value of a is", a);
}, 3000);

setTimeout(() => {
    console.log("timer 2 done");
    console.log("Value of a is", a);
}, 0);
a++;
let pr = createPromise();
pr.then(function success(value){
    a++;
    console.log("Pr promise fulfilled with a value", value);
}, function failure(value) {
    console.log("Pr promise rejected with a value", value);
});
a++;
blockingCode();
```



Pr promise fulfilled with a value Sanket
timer 2 done
value of a is 12

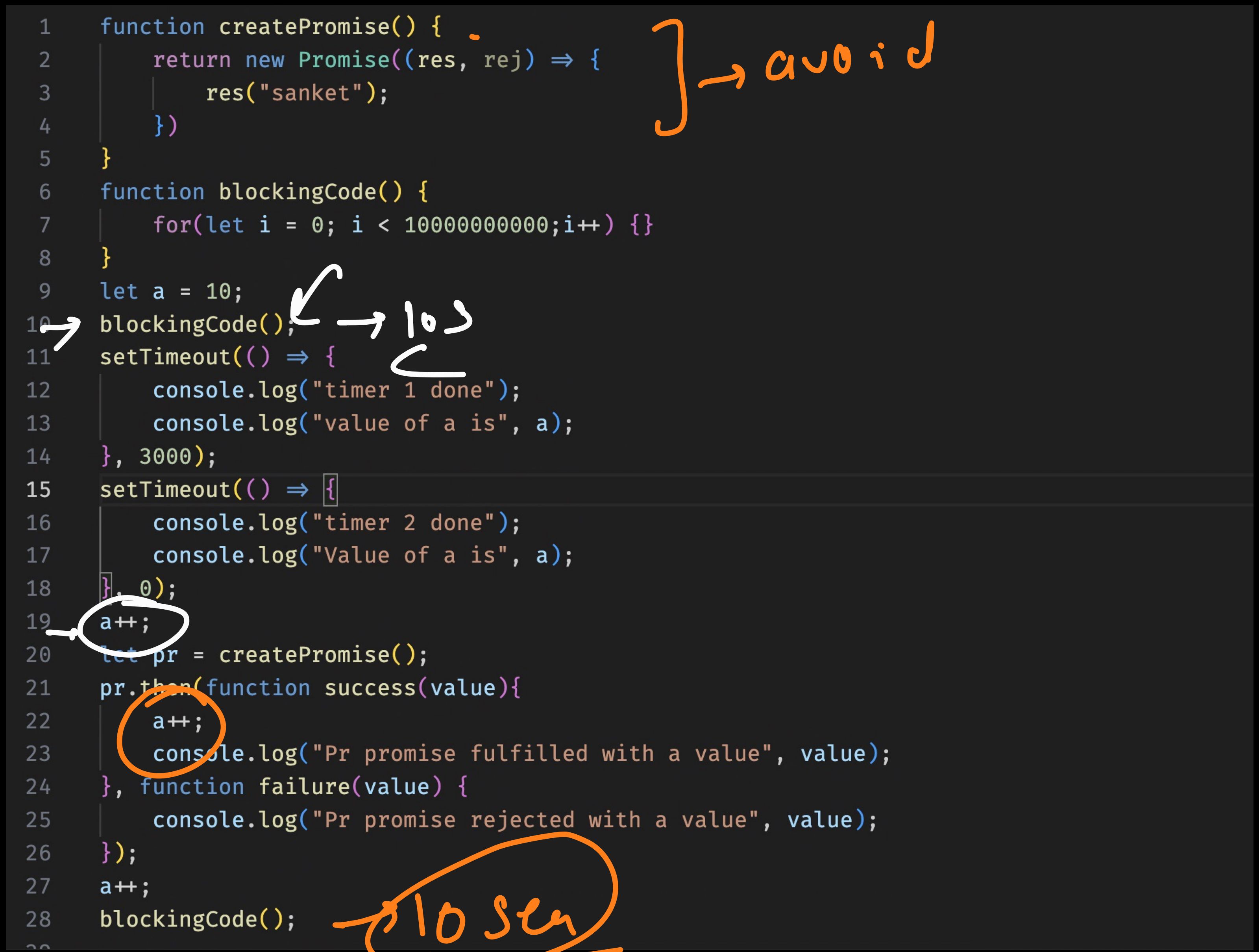
Timer 1 done
value of a is 13

① The moment promise state goes to fulfilled, ignore onReject & vice-a-versa

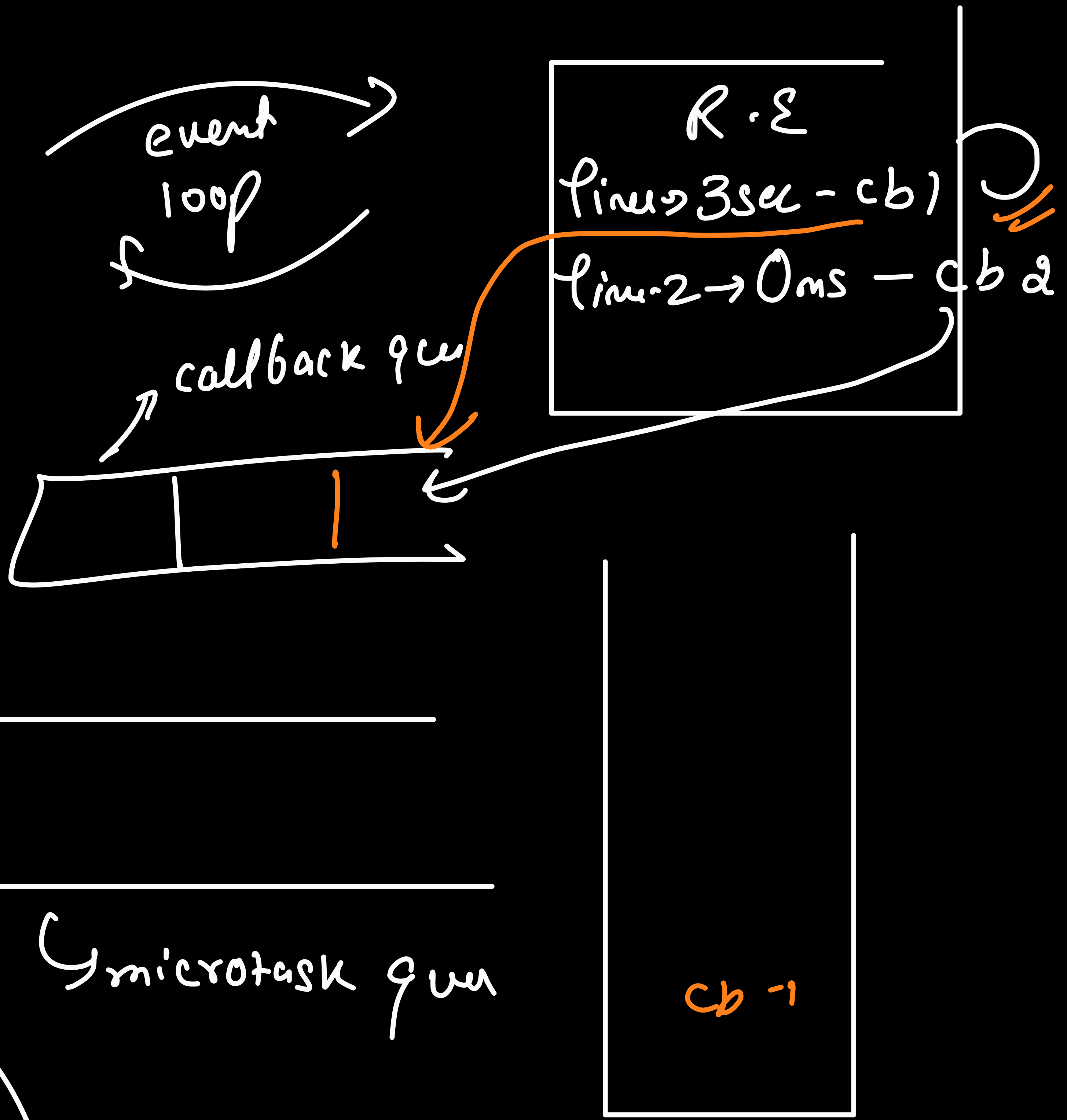
② whatever are the func^n present in the onfulfillment array are one by one pushed in the MICROTASK Queue.

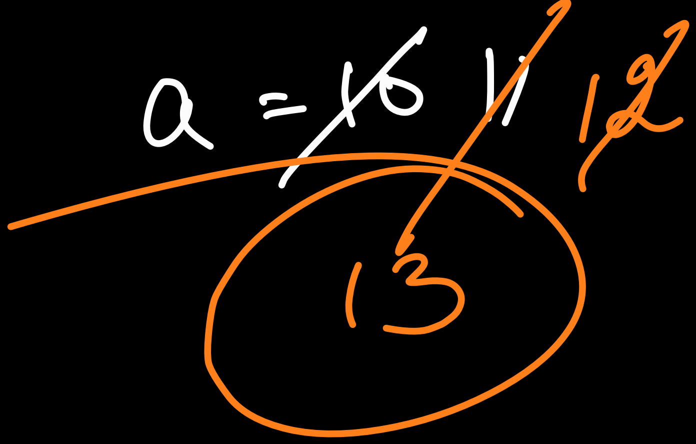
③ Event loop if stucks in a situation when it has to decide b/w MTQ & CBQ, it uses MTQ.

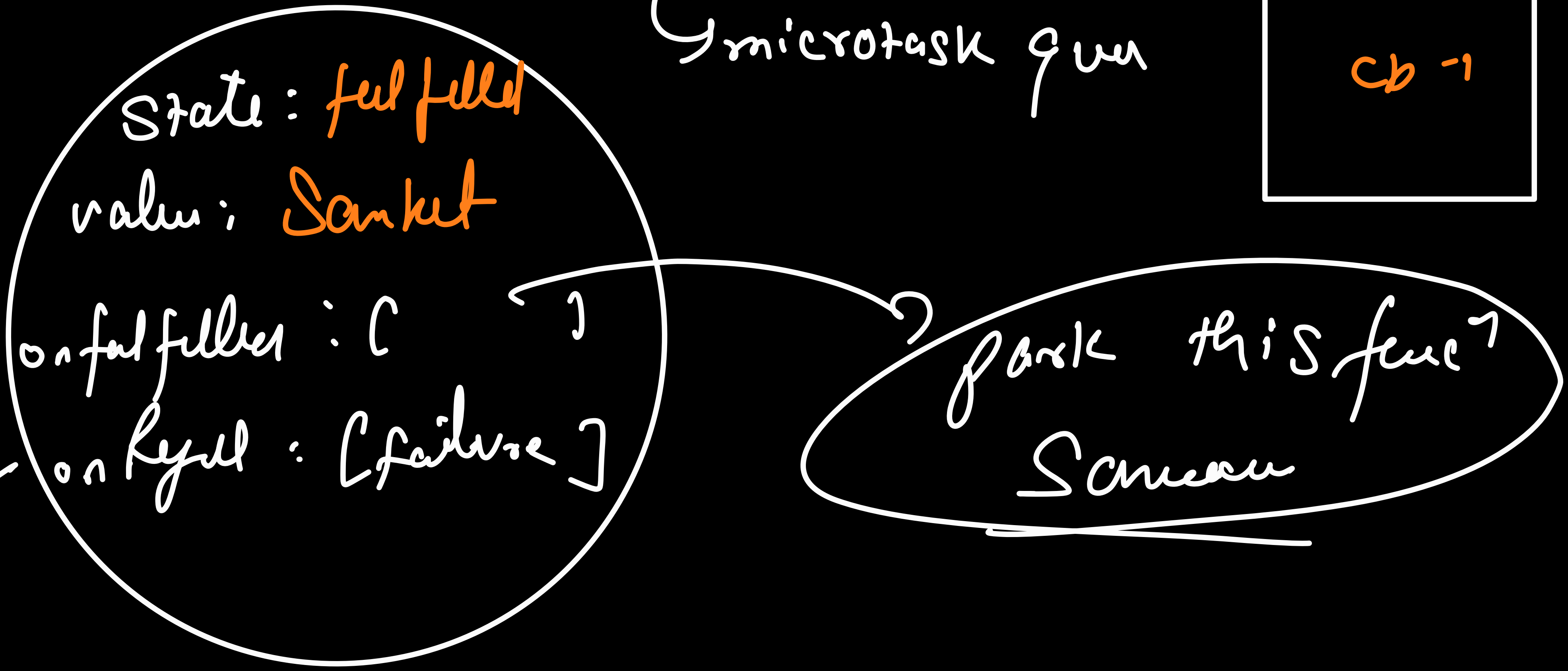
Create Promise → return us a promise which almost immediately gets resolved pending → fulfilled with value Scaler

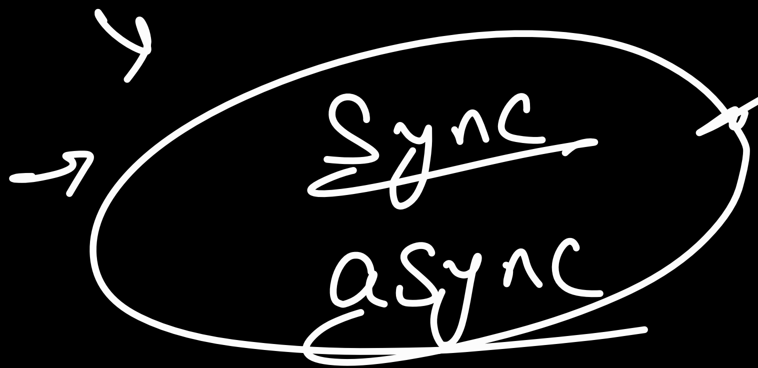


```
function createPromise() {
    return new Promise((res, rej) => {
        res("sanket");
    })
}
function blockingCode() {
    for(let i = 0; i < 10000000000;i++) {}
}
let a = 10;
blockingCode();
setTimeout(() => {
    console.log("timer 1 done");
    console.log("value of a is", a);
}, 3000);
setTimeout(() => {
    console.log("timer 2 done");
    console.log("Value of a is", a);
}, 0);
a++;
let pr = createPromise();
pr.then(function success(value){
    a++;
    console.log("Pr promise fulfilled with a value", value);
}, function failure(value) {
    console.log("Pr promise rejected with a value", value);
});
a++;
blockingCode();
```









Priority Of MTQ $>>$ Priority of CB queue

$$\begin{tikzcd}
{} \arrow[r] & \text{Event loop} \arrow[d, bend right] & & \\
& \text{Main thread} \arrow[r] & \text{call stack} \arrow[d, "\text{if empty}"] & \\
& & \text{Microtask queue} \arrow[d, "\text{if empty}"] & \\
& & \underline{\text{Callback queue}} &
\end{tikzcd}$$

→ why promise behave like R.E

Remember IoC ??

JOIN THE DARKSIDE