Episode 17

Trust Issues with setTimeout()

setTimeout does not gurantee that the callback method will be called exactly after given set of time, it may take more time to execute that callback function. This all depends on the call stack.

example :-

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

First GEC will be created and pushed into the call stack.
Now code will run line by line.
Now "start" will print in console
Now code moves to line ②
Now setTimeout will be registered into the web APIs and timer of 5000 will start
Now, javascript does not wait for anyone so now line ⑤ will be executed and "End" will be printed into console.

Now consider that we have 1 million line of code below line ⑤ which takes lot of time to execute.
suppose these all lines takes 10 seconds to run.

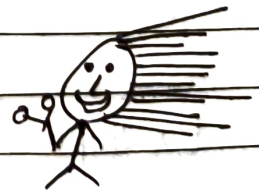
In this case our GEC is still busy in executing all these lines of code.
But now our timer of setTimeout callback function expires after 5 seconds
Now our timer expires and now callback function cb() will be in callback queue and is ready to execute
But our callstack is still busy in executing code, i.e. our callstack is not empty yet.

Event loop is constantly checking whether call stack is empty or not because when it will be empty then and then only code or functions which are inside callback queue will be executed.

But in our scenario, GEC will not be moved out from callstack before 10 seconds.

Now we have big question here, when will callback function cb() get a chance to execute as it is waiting in callback queue.

Answer is after 10 seconds



Why it happened so,
though setTimeout was just for 5 seconds, but it will wait for GEC to move out of call stack, and then execute.

imp And this is also called as concurrency model in javascript.

So after 10 seconds, global execution context will be moved out of callstack.
So, now our callstack is empty
Now event loop will pick up callback method cb()
and pushed into callstack.
And now our cb() function will be executed
and "callback" will be printed inside console.

But when we wrote this code, we actually expected it to run after 5 seconds.
So this is why setTimeout has trust issues.

We should not block our main thread, that means, we should not block our callstack
we should not have something in our program which is blocking the main thread.
because if callstack is not empty, then it cannot process any other event.

* Example of blocking the main thread.

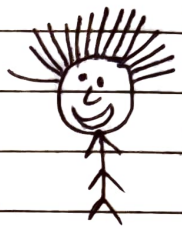
```
console.log("start");
setTimeout(function cb() {
    console.log("callback");
}, 5000);
console.log("End");

let startDate = new Date().getTime();
let endDate = startstart;

while (endDate < startDate + 10000) {
    endDate = new Date().getTime();
}
console.log("While expires");
```

output

start
End
While expires
callback

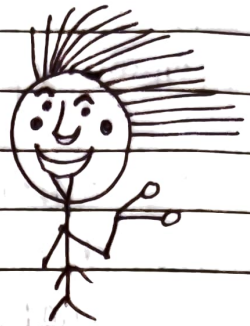


will take 10 sec to expires

first of all "start" will be printed
then callback function cb() will be registered into the Web API environment, and timer of 5000 second will be started.
Now "End" will be printed
then flow goes into while loop which takes 10 seconds to run.
Now this while loop kind of block our main thread.
And After 10 seconds "While expires" will be printed.
Meanwhile all these things are happening, our timer of setTimeout of 5 seconds has expired long back.

but callback function cb() was waiting in callback queue to execute.

Start
End
[after 10 seconds]



while expires
callback

setTimeout does not gursantee that it will take exactly 5 seconds but it gursantee that it will take at least 5 seconds to execute.

* What happens with setTimeout(0)

example :-

```
console.log("start");
setTimeout(function cb() {
    console.log("callback");
}, 0);
console.log("End");
```

⇓⇓

Start
End
callback

Even if timer is 0 seconds, function cb() has to go throw all the process behind the scenes.

This will not be very helpful, but these are use cases as well.

Again

JAVASCRIPT is synchronous single threaded language.

But

we can do async things in javascript as well.