episode 15

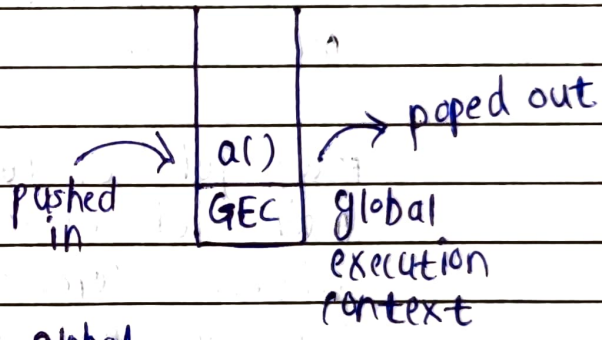
* Asynchronous Javascript and Event loop

* How JS engine executes the code using callstack

Javascript is a synchronous single threaded language. It has one call stack and it can only do one thing at a time and this call stack is present inside the javascript engine. And all the code is executed inside call stack.

example :-

```
① function a() {
②     console.log("a");
③ }
④ a();
⑤ console.log("end");
```



whenever any javascript runs, global execution is created and then global execution context is pushed inside the call stack. Now all our code in global execution context will run line by line. As we move to line ①, we get function definition of a() function and now 'a' will be allocated memory and function will be stored. Now when we reach at line ④, there is function invokation, so in case of function invokation then again execution context is created for function 'a' and again this execution context is pushed inside call stack. Now code of function 'a' will be executed

line by line, that means it will print "a" into the console, and now there is nothing more inside 'a' function so it pops out from call stack.
Now 'a's execution context is deleted.
Now code control moves to line ⑤ and it prints "end" into the console and now again there is nothing more to execute so our global execution context also pops out from call stack.

So, this is how javascript engine executes our code.

Main job of call stack →
main job of call stack is to execute whatever comes inside it.
It does not wait for anything.
whatever comes inside it, it quickly executes that.

[Time, Tide and ~~Javascript waits for None~~]

* How does javascript perform async tasks

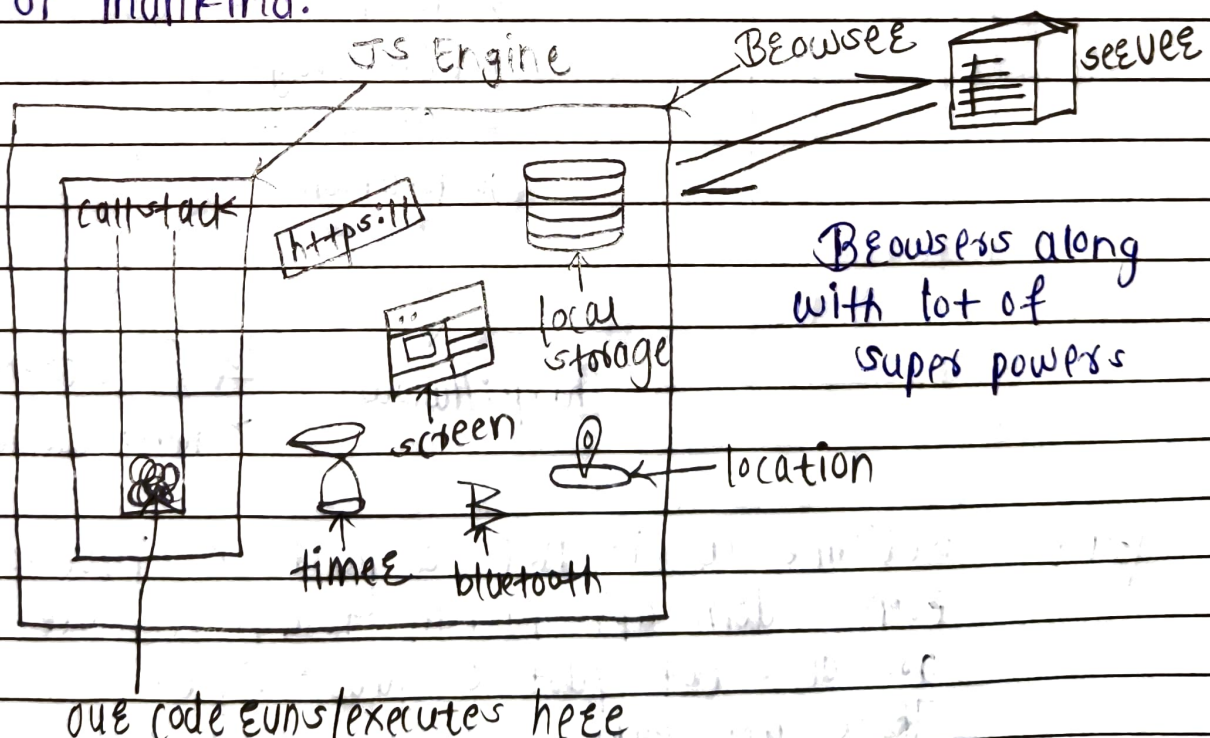
What if we need to wait for something
what if we have a program or a script which needs to execute or run after 5 seconds
can we do that ?
No, we cannot do that

because whatever comes inside the call stack then it automatically quickly executes that, it cannot wait for 5 seconds because it cannot have timers
[callstack does not have a timers]

suppose if we have to keep the track of timers and we have to execute some piece of code after certain delay, here we need some extra super power
so let's see how to get that super power of timers

* Behind the scenes in browsers

Browsers is most remarkable creation in history of mankind.

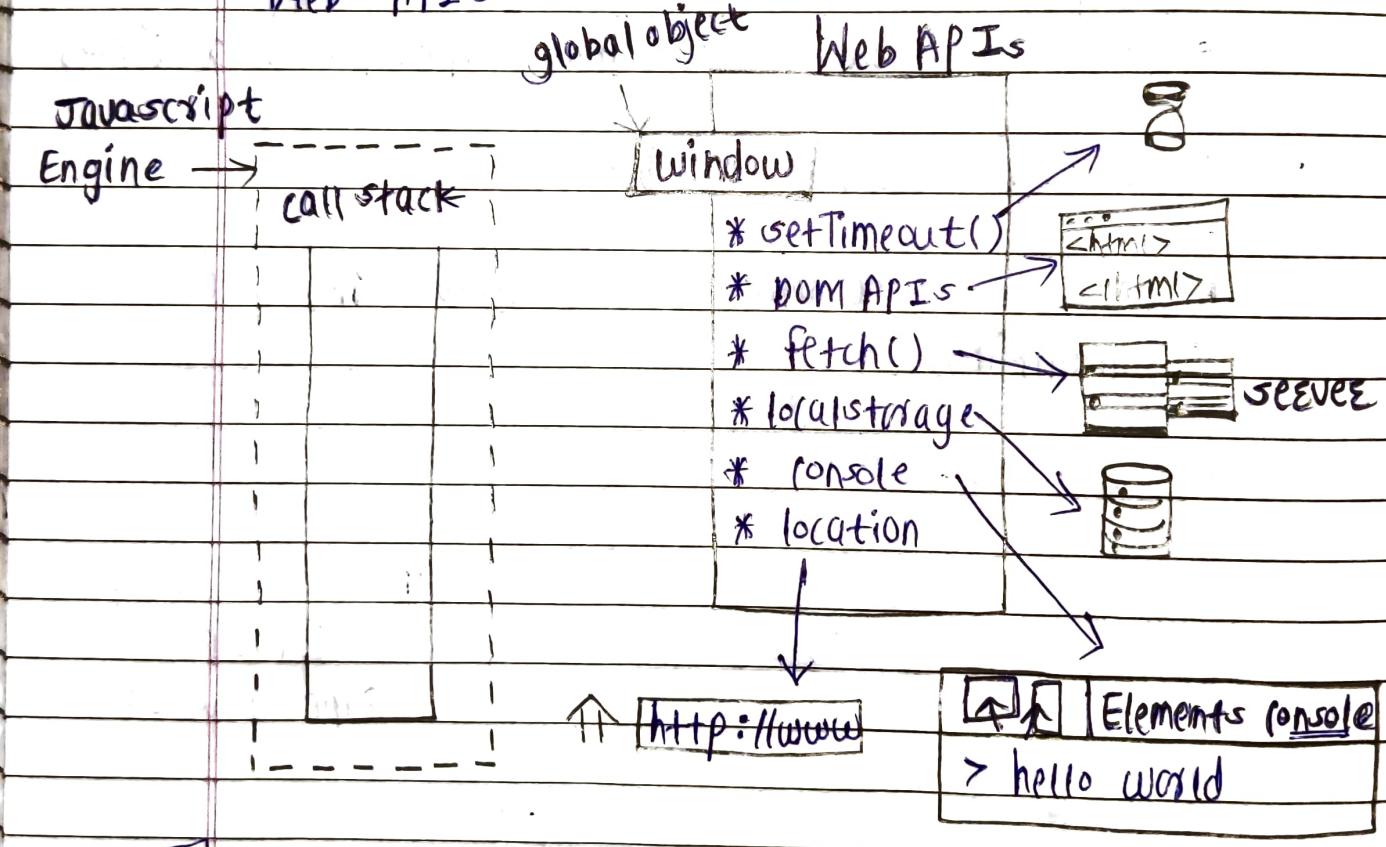


Now we need to access to timers, url, page which is rendered, local storage, bluetooth, etc

we need to access all these super powers so for this, we need to have a connection. javascript engine needs some way to access these super powers
so let us see how we can do that

* Web APIs in javascript

So to access all these super powers, we need Web APIs



Imp setTimeout is not a part of javascript even dom apis, local storage, console are also not part of javascript.
These are super powers which browsers have, they all are part of browser

Page No.
Date

IMP Browsers gives javascript engine facility to use all the super powers of browsers through a keyword known as "window".

so if you want setTimeout inside your code then you have to do window.setTimeout and if you do this then now we will get access of times.

so this is how we get access to all super powers.

but if we write our code, then we don't use window.setTimeout, we just write setTimeout because it is global object, setTimeout is present in global object [window is] or at global scope so we access it without keyword "window"

window.setTimeout — same thing — setTimeout

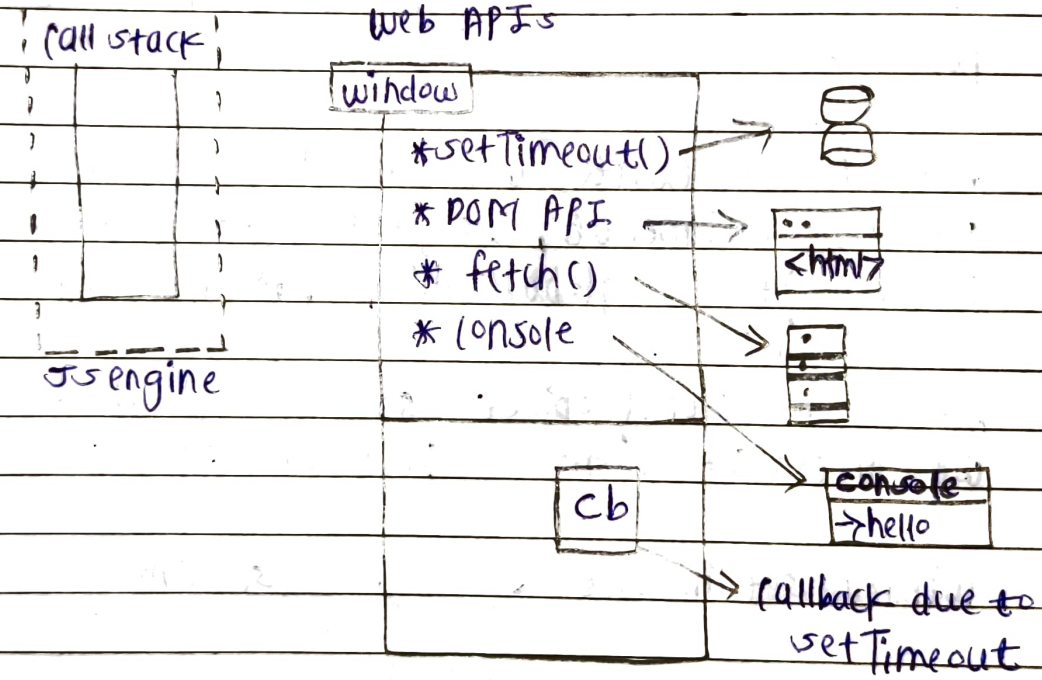
so this browsers wraps up all these super powers apis into global object (i.e. window) and gives access to our call stack. [of window object]

Now we can use all these super powers by using window. or directly we can use it.

* How setTimeout works behind the scenes in browsers

example :-

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



when javascript starts executing, global execution context (GEC) is created and pushed inside call stack.
Now code will run line by line.
so at line ① we need to use web api called "console"
so now "start" prints inside console.

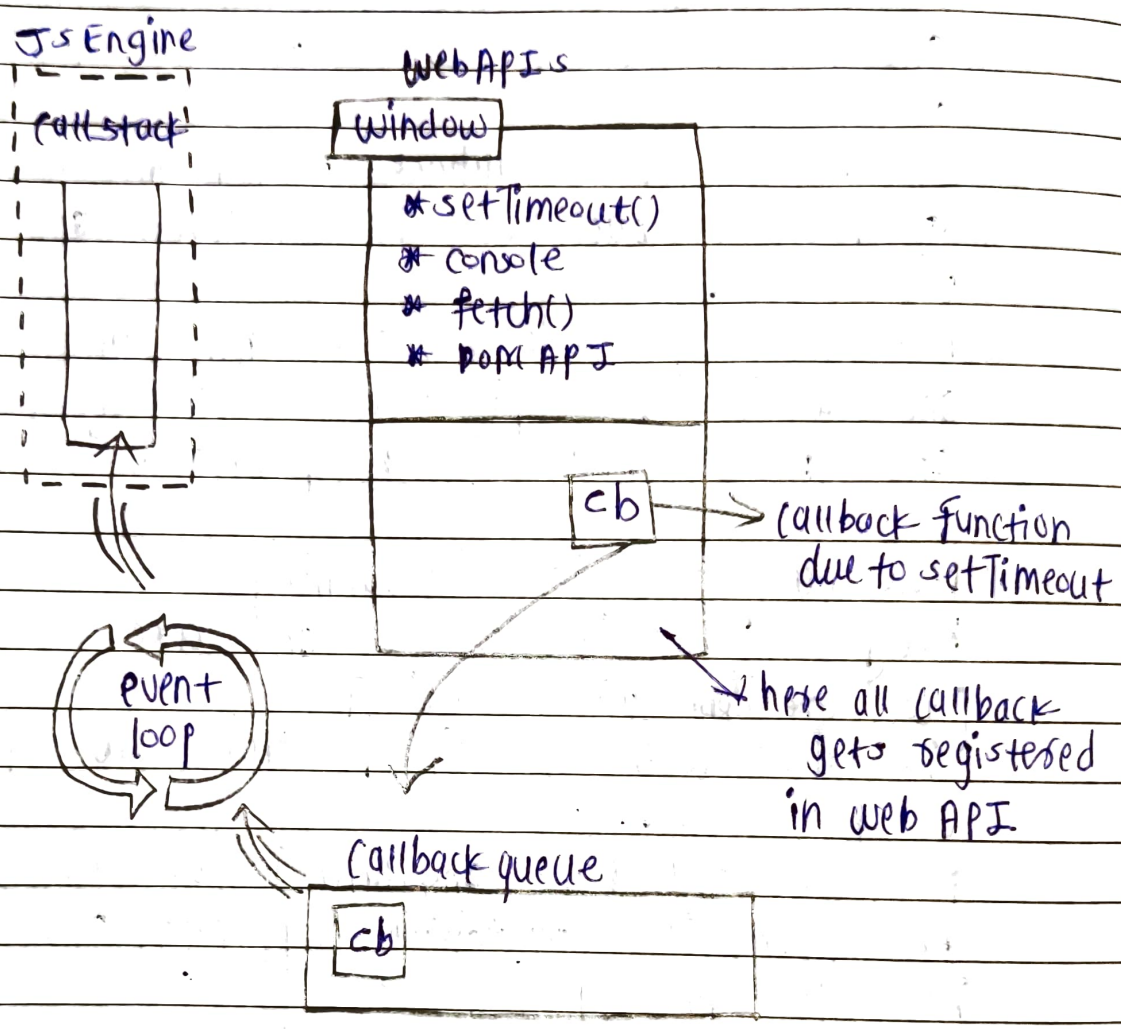
Page No.
Date

Now we move to line ⑫, here setTimeout will go into Web API and will call setTimeout and now we get access of times feature of browsers.
so now it takes callback function and some delay so now callback will be registered and timer will start.
Now javascript code moves to next line ⑤ and again web api of console will be called and now "end" will be printed inside console.

Now our timer is still running but our javascript code [we are done with executing] is done with executing. Now GEC pops out of the stack.
Now we don't have anything in our call stack.
And meanwhile all this is happening, our timer is still running
As soon as our timer expires, now our callback function need to be executed and because we know, all code in javascript is executed inside call stack, so we somehow need our callback function inside our call stack.

so now event loop and callback queue comes into picture

* Event loop and callback queue in JS



As soon as timer expires, callback function needs to go in call stack, but it cannot directly go into the call stack then how does it goes into call stack

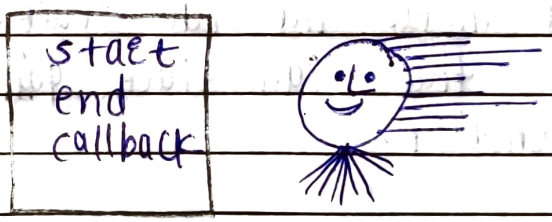
It will go through callbackqueue.
When timer expires, callback function is put inside callback queue

Page No.
Date

Now the job of the event loop is to check callback queue and put callback functions from callback queue inside our call stack.

[event loop acts like gatekeepers and it checks whether we have something inside callback queue and if we have something then it pushes inside our call stack and now call stack quickly executes callback function but how
again it creates an execution context and it runs line by line, when it sees console.log then again it calls web api and now "callback" gets printed inside our console.
and then again execution context will be deleted.

final output



This is how whole thing works.

* Another example you can try :-

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

* More about Event Loop

to

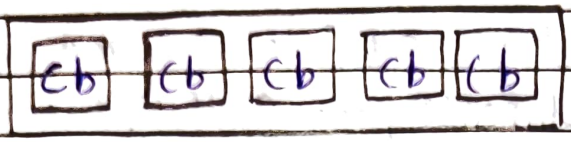
It's only job is ↑ continously monitor the call stack as well as the callback queue. so if call stack is empty and event loop sees that, there is also a function waiting to be executed inside callback queue then event loop takes function from callback queue and pushes it inside the call stack.
And then call stack quickly executes that function.

[when event loop picks any function from callback queue then that function gets vanished from callback queue]

* Why do we need callback queue

suppose user clicks on the button seven to eight times continously, in that case callback function will be pushed inside callback

queue seven to eight times.



callback queue

Now we will have all these callback functions waiting to be executed. in a queue.
And now with help of event loop one-one callback function will be pushed inside call stack.

so, generally in real-life application, we often see that there are a lot of event listeners, lot of timers, so that is why we need to queue all callback functions together in callback queue so that they can get a chance one after the other because javascript only have one call stack.

* How fetch() function works

we have seen how setTimeout and DOM API (document.getElementBy.... addEventListeners) works, but we have not seen how fetch works and fetch does not work like setTimeout and DOM API, it works differently

Example :-

```
① console.log("start");
② setTimeout(function cbT(){
③     console.log("CB setTimeout");
④ }, 5000);
⑤ fetch("https://api.netflix.com")
⑥ .then(function cbF() {
⑦     console.log("CB Netflix");
⑧ });
⑨ console.log("end");
```

many line hose

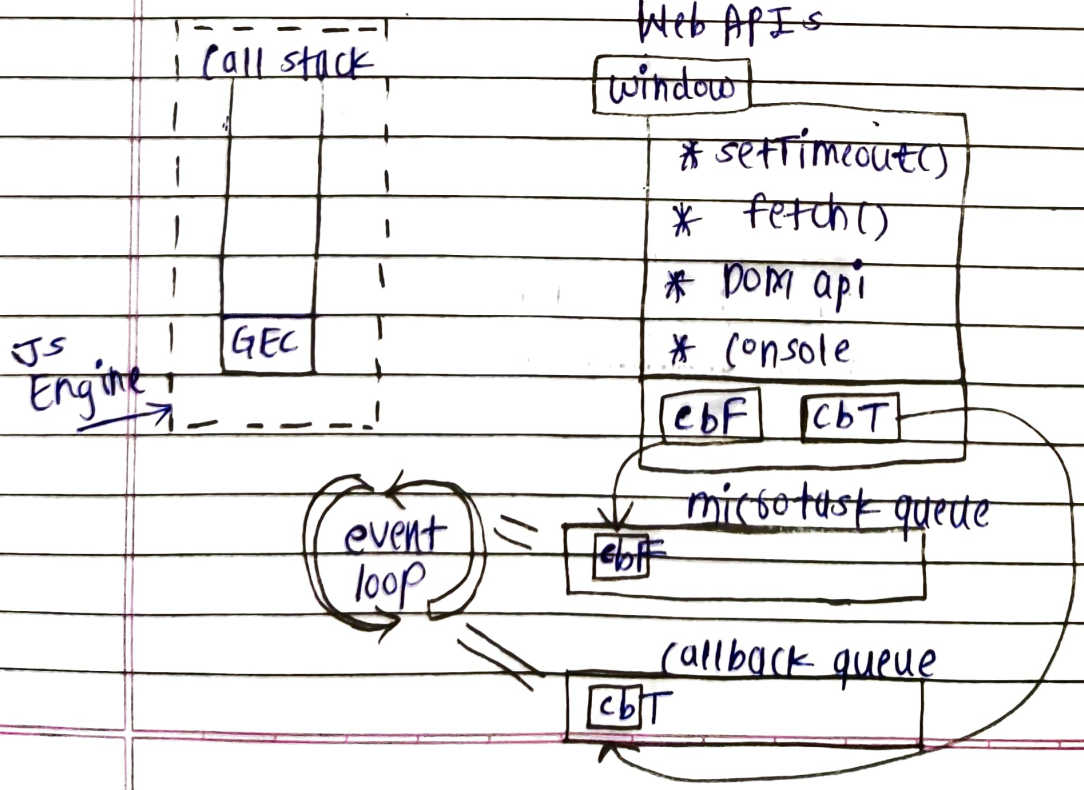
output

start

end

CB Netflix

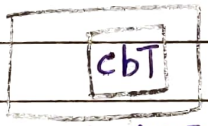
CB setTimeout



so now this time we have setTimeout as well as fetch function.

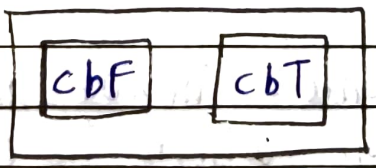
[fetch basically goes and requests an api call, fetch function returns a promise and we have to pass a callback function which will be executed once promise is resolves.]

first GEC is created and pushed inside the call stack, code is executed line by line and do a console.log
Now "start" will be printed inside console.
Now setTimeout will register a callback function in web API and times of 5000 will start



web API environment

Now we move to line ⑤
so Now we have fetch function, and fetch is again a web api which makes network call and also registers callback function into the web api's environment.



web api environment

cbT function is waiting for the times to expire so that it will be pushed into callback queue.

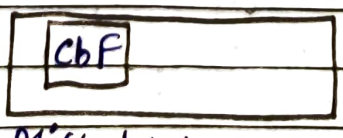
And cbF function is waiting for the data to be returned from server of netflix.
[fetch will make network call to netflix server]

Now netflix server gives back data and now cbF function is ready to be executed.
suppose netflix servers are fast and it returns quickly the data
Now guess where cbF function will go
in callback queue ← No
then where

* MicroTask queue in Js

Just like callback queue we also have something called microtask queue.
both are same but microtask queue has higher priority than callback queue.
i.e. functions inside microtask queue will be executed first and functions inside callback queue will be executed later.

Now what comes inside microtask queue
cbF function which is callback function in case of promises or in case of network calls will go inside microtask queue



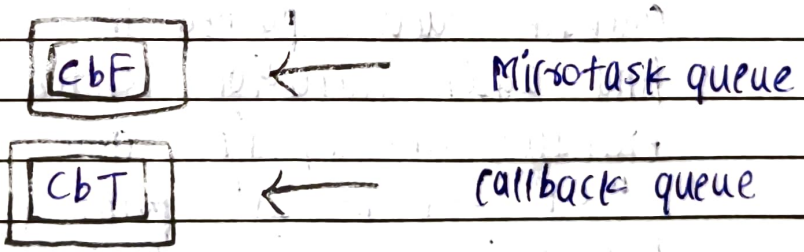
Microtask queue

Page No.
Date

Now we have got response from netflix servers and cbF function is now in microtask queue.
Now event loop will check whether call stack is empty or not.
And it's not empty because we are not done with our code, still there are many lines below line ⑧
There are millions of line below, line ⑧ and it takes some time to execute these lines.
MeanWhile we are running and executing these lines, now our times also expires

[browsers is doing lot of things, understand carefully]

As times is expired, now our function cbT wants to be executed and it now comes inside callback queue.



Now our functions cbF and cbT, both are ready to executed but our code is still running as we have millions of line.
Now suppose all millions of line finished executing, so "end" will be printed in console.
And now there is nothing to execute
Now GEC will be poped up from call stack.
Now our call stack is empty

Page No.
Date

Now event loop finds that our call stack is empty and we have some functions waiting to be executed which are in microtask queue and callback queue.

Now because microtask queue has high priority so cbF function will be pushed inside our call stack.
then again execution context will be created and code will run line by line and now "CB Netflix" gets printed into the console.
Now again execution context will be poped up from call stack.
Now again event loop finds that call stack is empty
so now the function cbT which is inside callback queue pushed in call stack.
And again execution context is created, code executes line by line and
"CB setTimeout" prints inside console
And again execution context will be poped out.

This is how this whole thing Works.

```
:Start
end
CB Netflix
CB setTimeout
```

Page No.
Date

* What are MicroTasks in Javascript

What can come inside microtask queue ?
all the callback functions, comes through promises will go inside microtask queue.
There is something called mutation observee.
Mutation observee basically keeps on checking whether there is some mutation on dom tree or not.
If there is some mutation on dom tree then it can execute some callback function.

So two things, callback functions which comes through promises and mutation observee goes inside microtask queue.

And all other callback functions which comes from setTimeout or dom api like event listners goes inside callback queue.

[callback queues are also called as task queues]

Microtask queue is given high priority.

suppose there are three microtask pending inside microtask queue and we have only one task inside callback queue
So Now event loop will only opportunity to callback queue task, once all the tasks from

microtask queue is completed.

imp * starvation of functions in callback queue :-

just because event loop gives chance first to microtask queue before any of the callback queue task

suppose, microtask creates a new microtask in itself and again this task creates another microtask and so on ....
so now the task which in callback queue will never get a chance to get executed because microtask has more priority.
And this is called as starvation of task inside callback queue.

[this is possibility]

[all topics/things which we see in this episode is power of Browser]

Next episode, power of JS engine.