· github : divyaranjan10

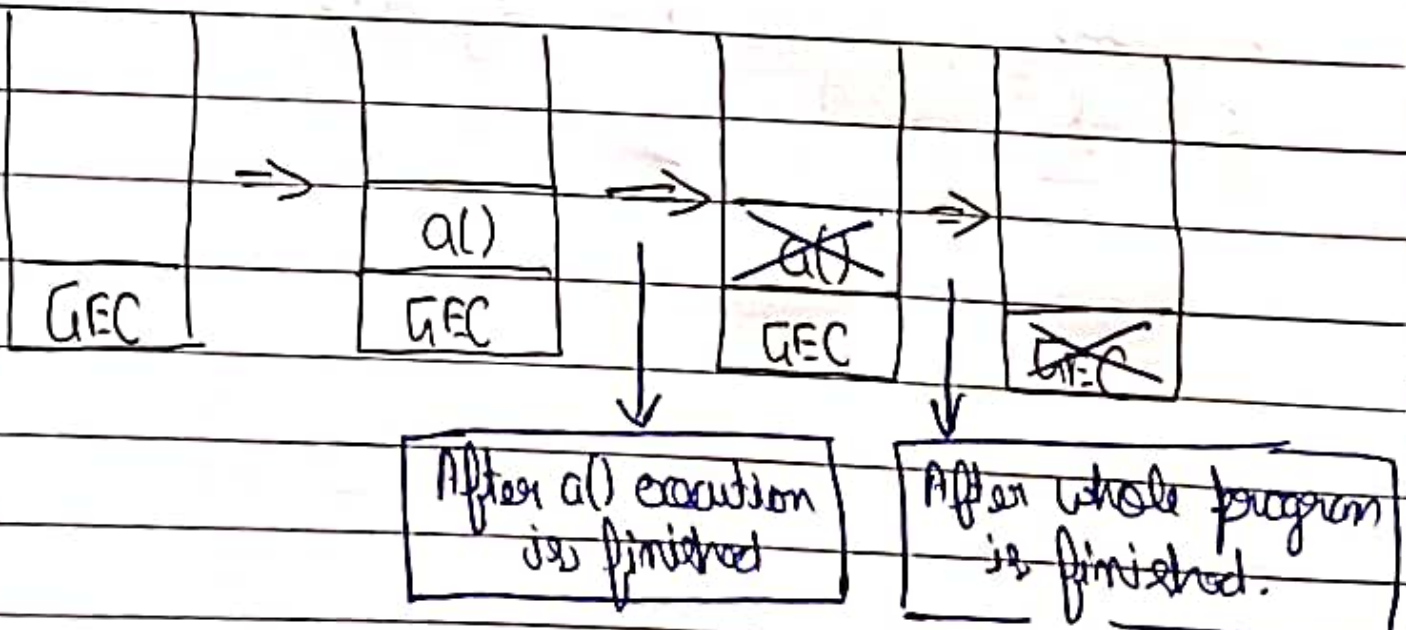
DATE

CHAP 15 : Asynchronous JS & Event loop from Scratch.

" JS is a single-threaded language "
" Whenever we run a JS program, a GEC is created and pushed into the call stack ".

```
function a(){
    console.log("a");
}
a();
console.log("End");
```

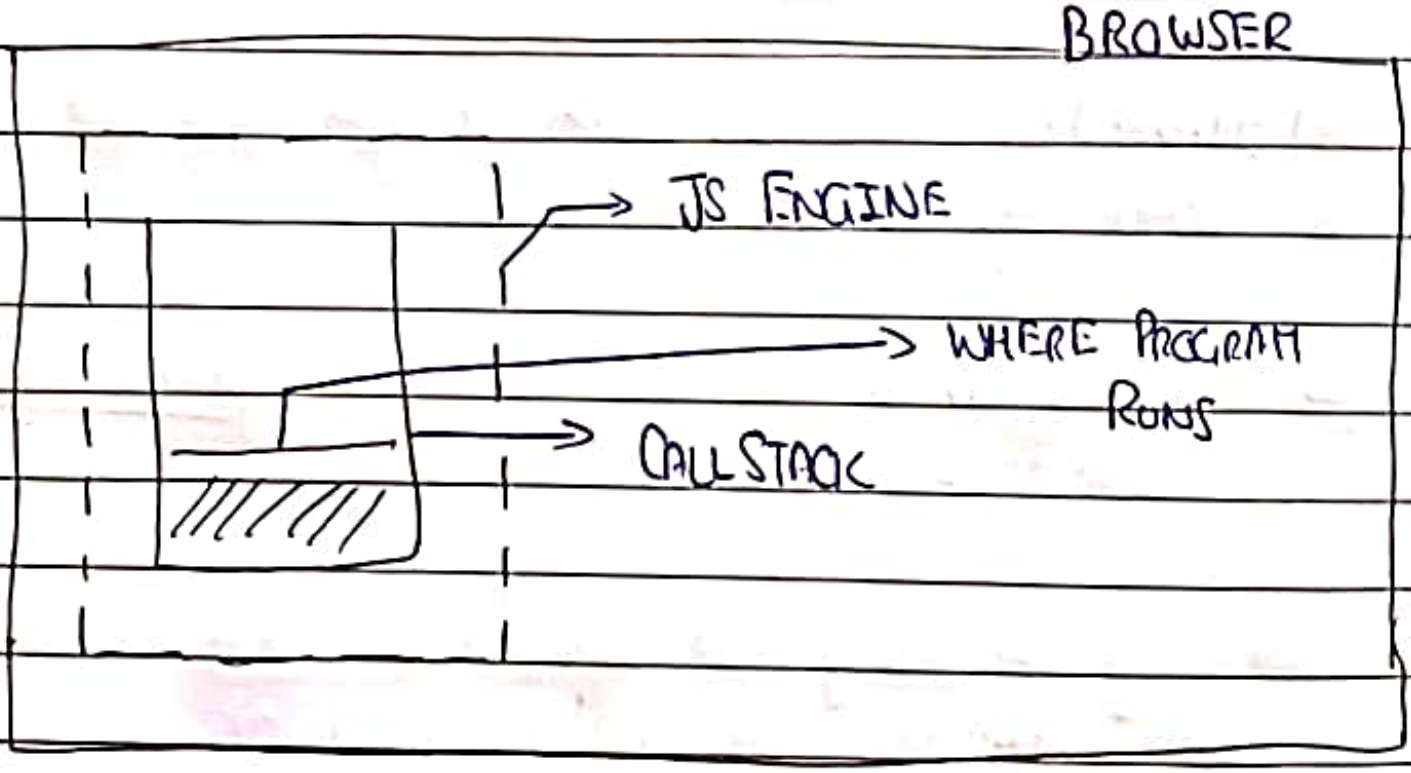
· Console
a
End



DATE

"Main job of the call stack is to execute everything that comes inside it. It doesn't wait for anything, it just quickly executes what comes inside it."

- Behind the scenes in Browser:



- Web APIs: Web APIs are the powers which are given by the browser to the JS engine to use it.

DATE

- Web APIs : setTimeOut()
  DOM APIs
  fetch()
  local Storage
  console
  location

" setTimeOut(), DOM APIs and console are not a part of JS "

" We can use these web APIs using a global object called window ".

** We may or may not use this word "window" in our program, as window is a global object and it is present in the global scope, we can write it without using this keyword. So "window.setTimeOut()" or "setTimeOut()" both are same thing. **

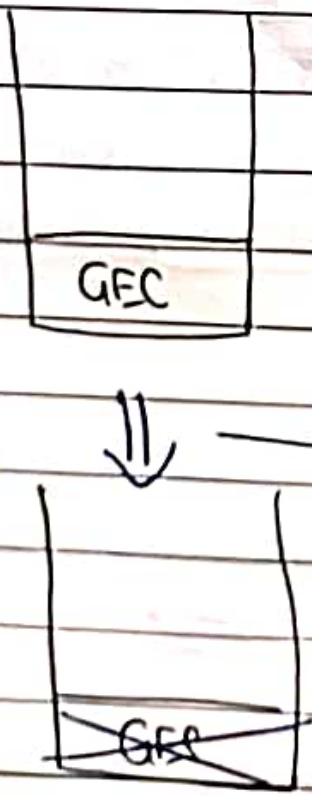
DATE

- How setTimeout works behind the scenes in Browsers:

```
console.log("Start"); (15.1)
setTimeout(function cb(){ (15.2)
    console.log("Callback");
}, 5000);
console.log("End"); (15.3)
```

Web APIs
- setTimeOut()
- DOM APIs
- fetch()
- console

- Call Stack
- Console
  Start
  End



⇒ (15.1) and (15.3) use the web API "console" and consoles "Start" and "End" from the program.

→ After JS program is finished, GEC is also popped out of the call stack.

DATE

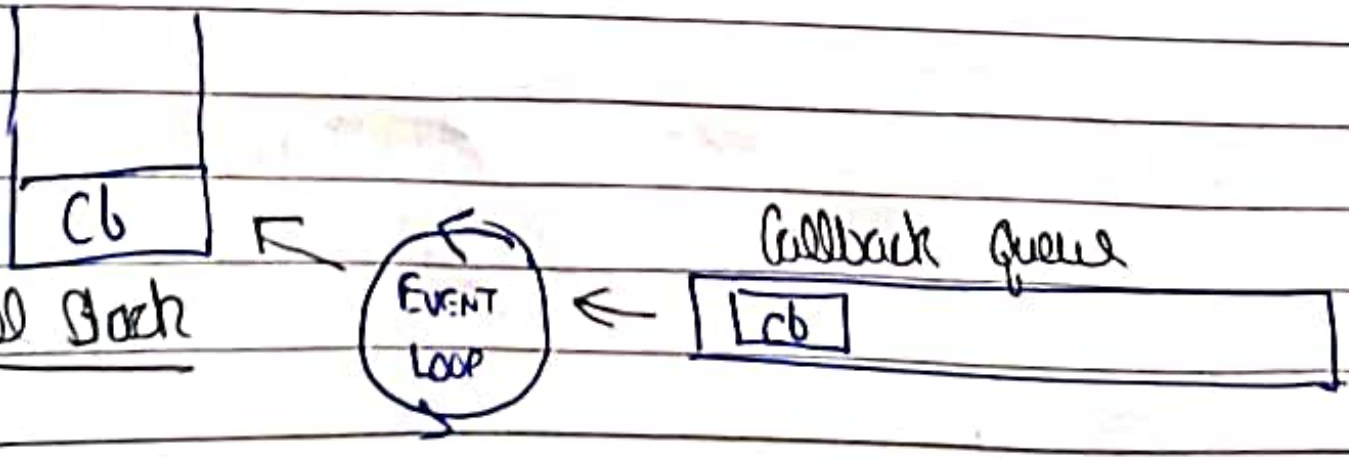
(15.3) It takes the callback f^n and stores it in a separate space and attaches a timer to it, when timer expires, the function executes. Also, we cannot put 'cb' function directly into the call stack. We need a 'call back' queue' and 'event loop'.

* After timer finishes, the 'cb' function gets pushed in the 'callback queue'. *

- Callback Queue

| cb |

=> Then event loop comes into picture, it acts like a gatekeeper and checks the 'callback queue' if it has something and then pushes it into the call stack.



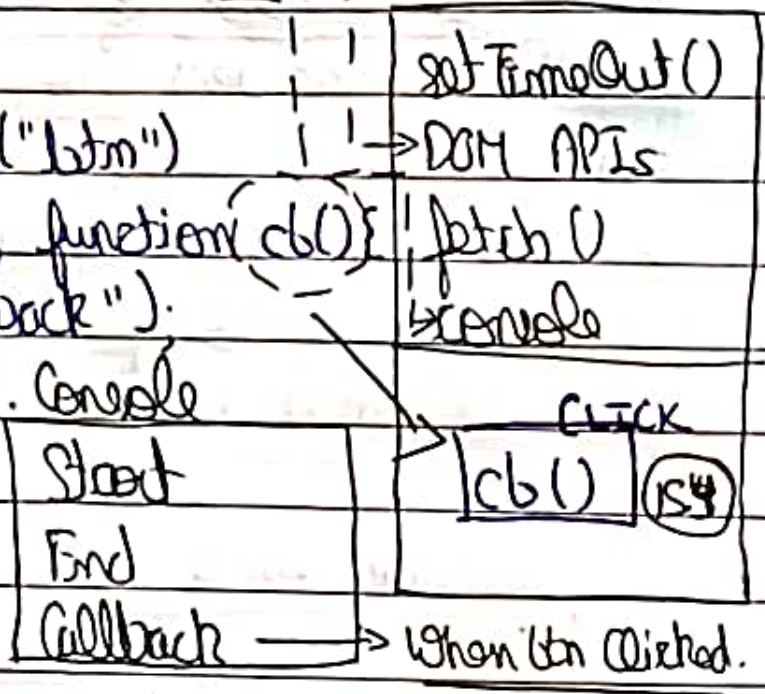
DATE

- After cb executed, then console: And 'cb' gets popped out the call stack.

Start
End
Callback

- How Event listeners work in JS:

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



* The process is almost similar like in the previous example *

=> In the Web APIs environment, it puts the callback function 'cb' and attaches 'click' event to it. (5.4)

(5.4) the callback 'cb' stays there until we remove the event listener.

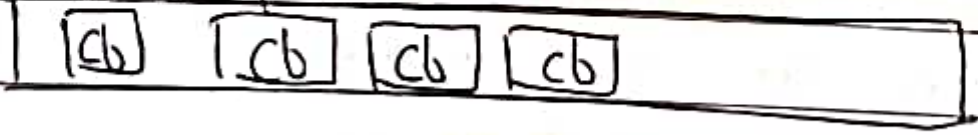
DATE

** Event loop job is to continously monitor the 'callback queue' and 'event loop'.

- Why we need callback queue?

⇒ So, generally in a real life application, there must be a lot of 'event listeners' and 'callback' functions needed to be executed, then there might be a long queue form then we execute all the callback functions, that is why we need to put them in the callback queue so that it can execute one after another

- Callback queue



- How fetch() function works?

```
console.log("start");
setTimeout(function cbT(){
    console.log("CB timer");
},5000);
```

DATE Web APIs

```
fetch ("https://api.netflix.com")
.then (function cbF() {
    console.log ("CB Netflix");
});
setTimeOut
Dom APIs
fetch()
console
```

```
console.log ("End");
```

50ms

cbF | cbT

* So, this fetch() function requests an API call and returns a promise. *

=> And when promise is resolved, it execute the function.

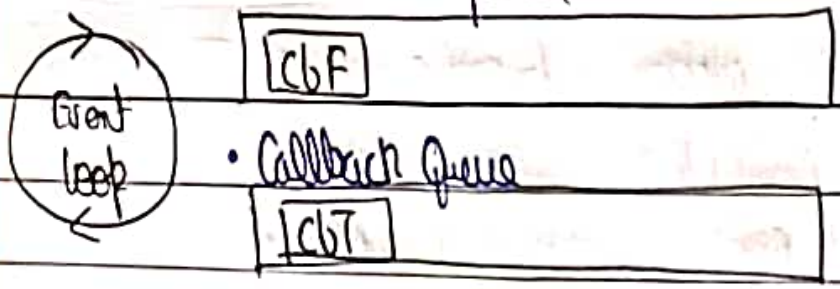
- Unlike the callback function 'cbT' in the set Timeout web API, the callback function in the fetch, i.e 'cbF' doesn't go in the callback queue but the Microtask queue.

- Microtask queue priority is greater than callback queue.

Microtask Queue >>> Callback Queue

DATE

- Microtask Queue



=> So, this event loop continously checks and gives priority to the microtask queue when callstack is empty.

=> Therefore output will be:

- Console:

Start
End
CB Netflix
CB Timer

** All the callback functions which comes through promises and mutation observer (change in DOM tree) goes into the microtask queue. **

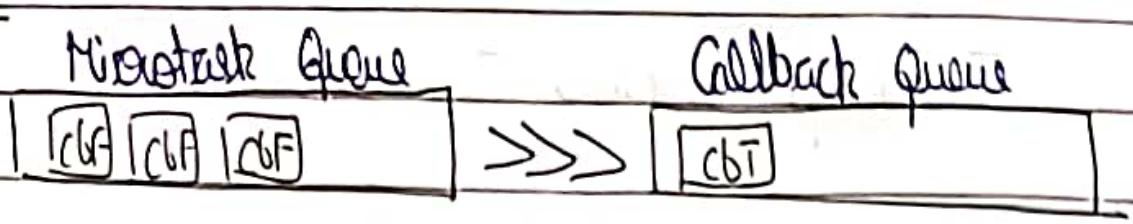
* Important Note *

=> Suppose, microtask queue has 3 callback functions to be executed and callback queue has only one, still event loop will give chance to callback queue once all the 3 tasks in microtask

DATE

queue gets executed.

Ex:



- Starvation of functions in Callback Queue :

- Microtask Queue

- Callback Queue

⇒ Suppose, the callback fn in microtask has another microtask and that microtask has one more microtask in itself. Therefore, the 'callback fn' in the 'callback queue' has to wait for long to be executed as microtask queue has higher priority, and this leads to the starvation of functions in Callback queue.