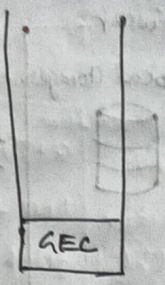
Asynchronous JS & Event Loop:

How JS Engine executes the code using call stack?

→ call stack is present inside JS engine and all the code of JS is executed inside call stack.

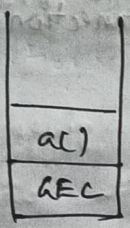



```
function a(){
    console.log('a');
}
a();
console.log('End');
```

→ In the GEC, whole code runs line by line

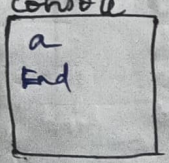
→ As we move on to first line, there is function definition of a and a will be allocated memory and function is stored.

→ In case of a function invocation, an EC is crated to run the code inside the function.



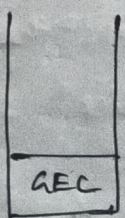
→ code of function a executed line by line and prints a to the console.

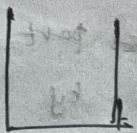
→ There is nothing more to execute inside a(), therefore it gets popped off.



→ Now control moves to last line and prints 'End'

→ There is nothing to execute GEC also pops off.



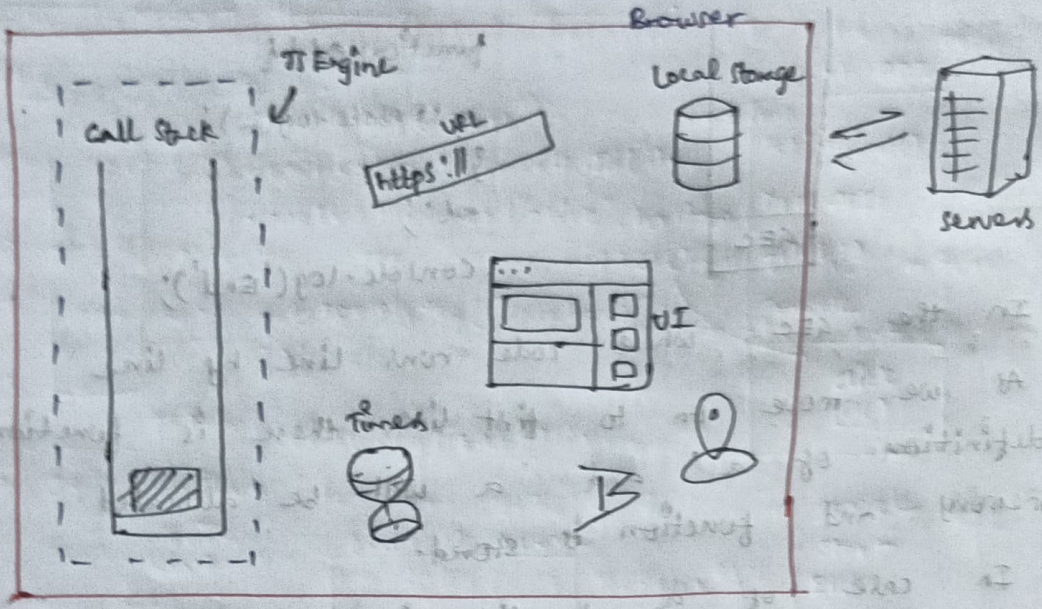


→ Empty call stack

Main job of the call stack:

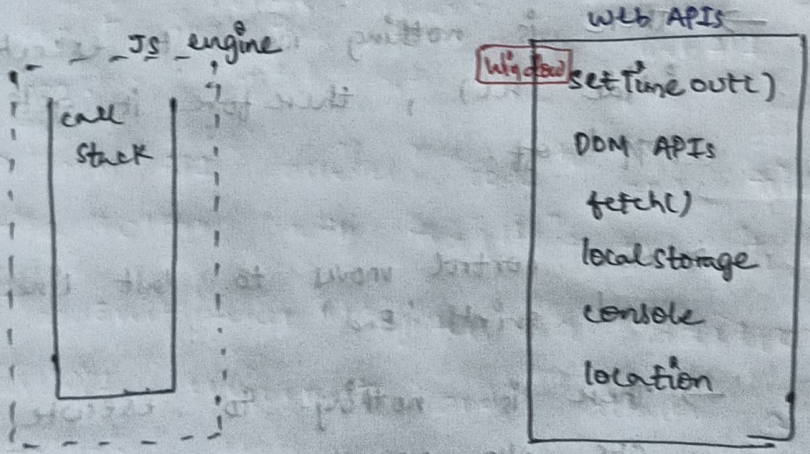
→ Main job of the call stack is to execute whatever comes inside it. It does not wait for anything.

Behind the scenes in Browser:



→ JS code needs access to all these features provided by the browser. JS engine needs a connection here, for that we need Web APIs.

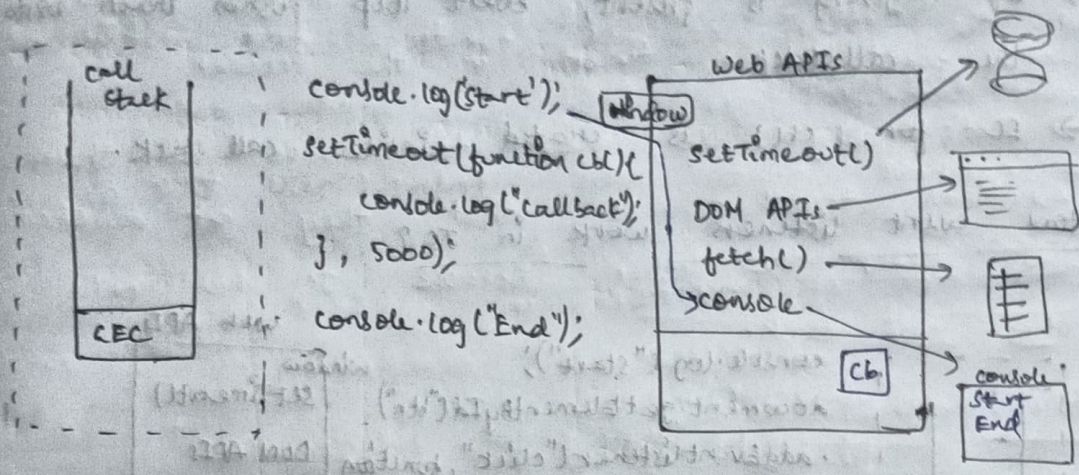
Web APIs in JS:



→ These Web APIs are not part of JS, they are the features provided by browser to JS

engine this access is provided by window object.

setTimeout in the Browser:



→ 3rd last line of code calls setTimeout API which gives access to timer.

→ This API registers a callback and timer start 5000 ms

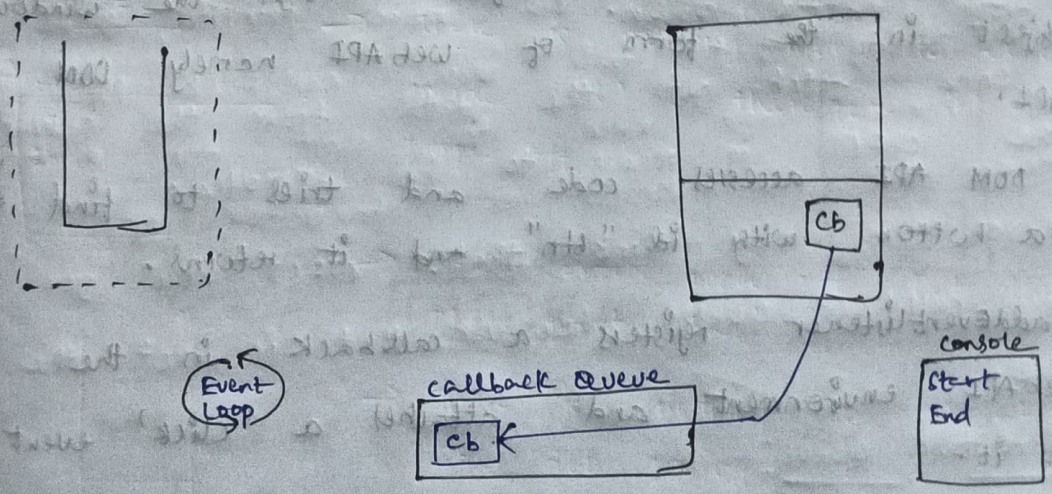
→ Then code moves to next line.

→ All this while timer is running, Nothing to execute now, GEC popped out of stack.

→ Now cb() will be executed, we now need this cb() inside call stack,

Eventloop and the callback Queue!

→ cb() cannot directly go into the callstack.



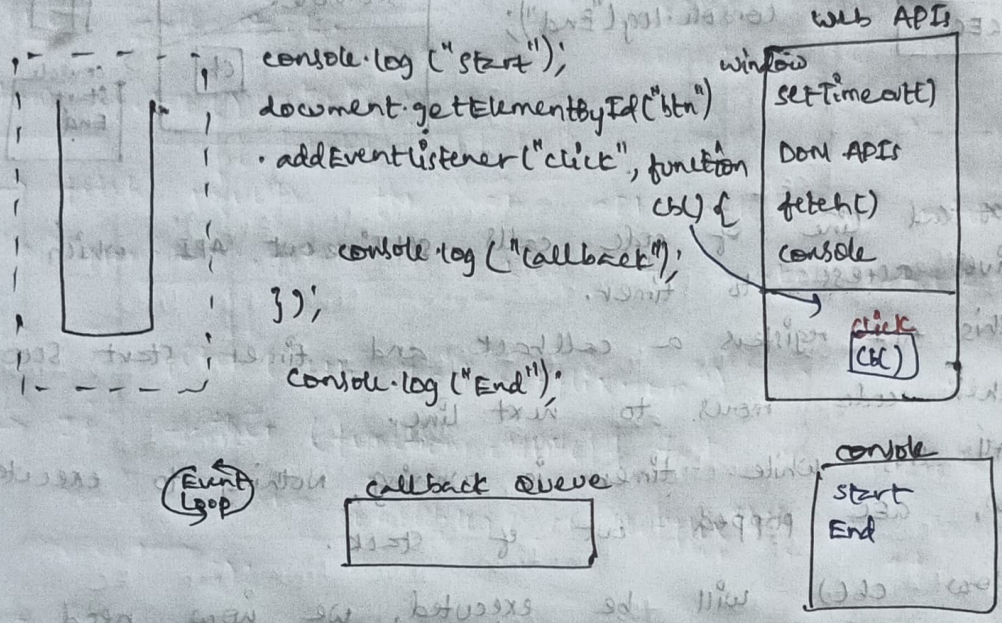
→ When the timer expires, the cb() is pushed inside callback Queue.

→ Event loop acts as a gatekeeper and checks if anything is present inside callback queue.

→ If anything is present, event loop pushes them onto the callstack.

→ EC for cb() is created inside call stack.

How event listeners work in JS?



→ As usual, a GEC is created inside callstack and pushed to callstack.

→ In 2nd line, addEventListener is a feature given by the browser to JS engine through the window object in the form of Web API namely DOM API.

→ DOM API accesses code and tries to find a button with id "btn" and it returns.

→ addEventListener registers a callback in the Web API environment and attaches a 'click' event to it.

→ JS moves to next line and prints "End".

→ Now there is nothing to execute and GEC pops out of the stack.

→ But event handler will stay in the Web API environment until and unless we explicitly removes it or close the browser.

→ If any user clicks on the button, cb() goes to callback queue.

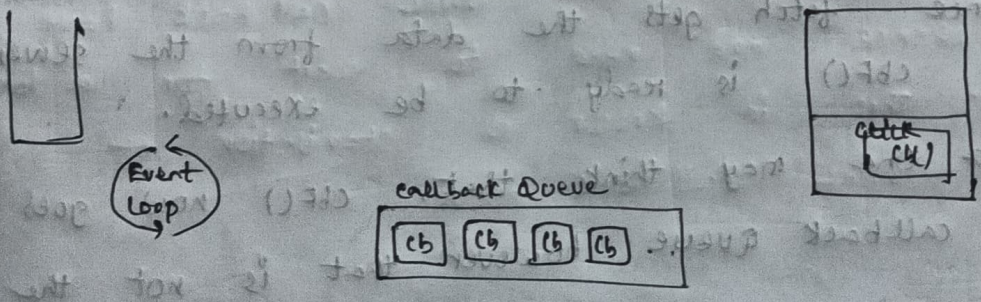
Event Loop:

→ Its only job is to continuosly monitor call stack and callback queue.

→ If event loop notices call stack is empty and any functions are in call back queue, then event loop takes that function and pushes onto the stack. Then the cb function executed inside the stack.

Why do we need this callback queue?

→ Event loop could have directly taken up the cb() from web API environment and pushed inside stack. But this is not the case. Why?



→ Suppose if a user clicks a button multiple times, in that case cb() pushed inside queue multiple times waiting to be executed.

→ one by one, these are picked up by event loop.

→ In practical situations, there will be number of event listeners, timers in the browser

→ That's why we need to queue all these functions together so that they get a chance one after the other because JS has only one call stack.

How fetch() works!

→ fetch requests for an api call and returns a promise. fetch is used to make network calls.

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

→ cbF() is registered as a callback in webAPI environment.

→ cbF is waiting for data to be returned from netflix servers.

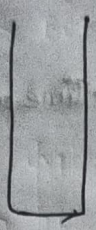
→ once fetch gets the data from the servers the cbF() is ready to be executed.

→ One may think that cbF() now goes to callback queue however that is not the case.

Micro task queue!

→ It is similar to callback queue but

microtask queue has higher priority.

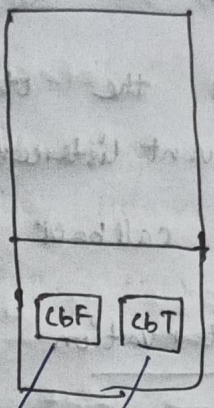


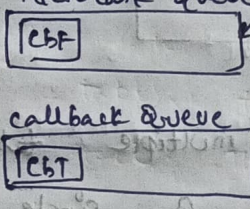
```
console.log("start")

setTimeout(function cbT(){
    console.log("CB Timeout");
}, 5000);

fetch("https://api.netflix.com")
.then(function cbF(){
    console.log("CbF Netflix");
});

console.log("End")
```





→ whatever functions in Microtask queue are executed first and then callback queue.

→ CbF goes inside M.T queue and event loop checks for call stack to empty and takes functions from queues.

What are Micro Tasks?

→ All the callback functions that come from promises goes inside micro task queue.

→ There is a Mutation Observer which checks for any mutation in the DOM tree. If any mutation is observed, it executes some callback functions.

→ So these two (promises & Mutation observers)

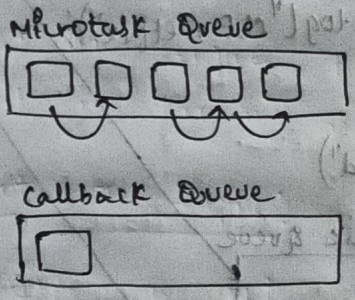
goes inside microtask queue.

↳

All the other callback functions from setTimeout, event listeners goes inside callback queue.

↳ callback queue also known as Task Queue.

Starvation of functions in callback Queue:



→ If there are multiple tasks in microtask queue and only a single task in callback queue,

→ In the MT queue, if a task creates a new task again, there will be no chance for tasks inside callback queue to execute.

→ Because MT queue has more priority, there is possibility that tasks inside call back have no chance and need to wait for a long time. This is known as starvation of tasks inside call back queue.