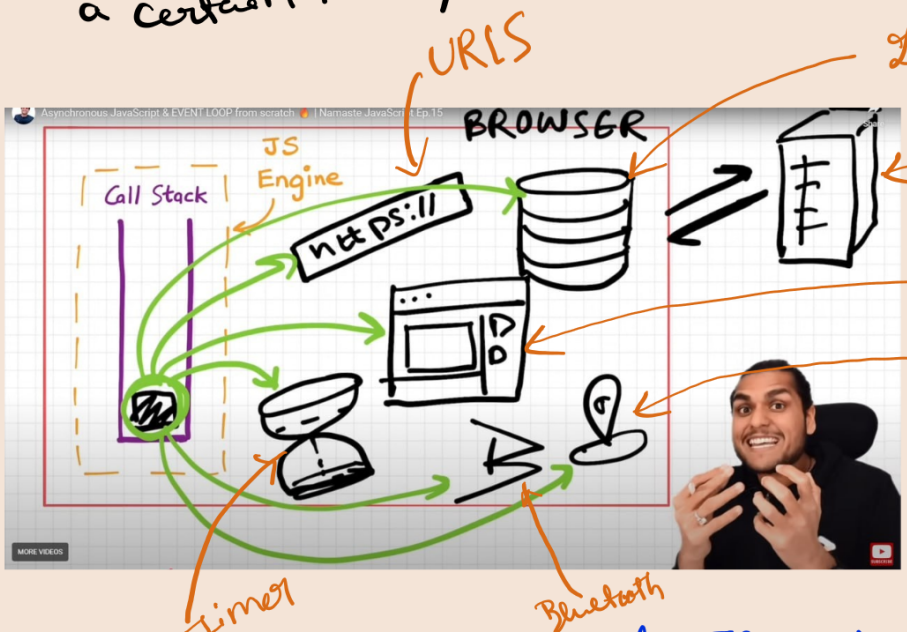
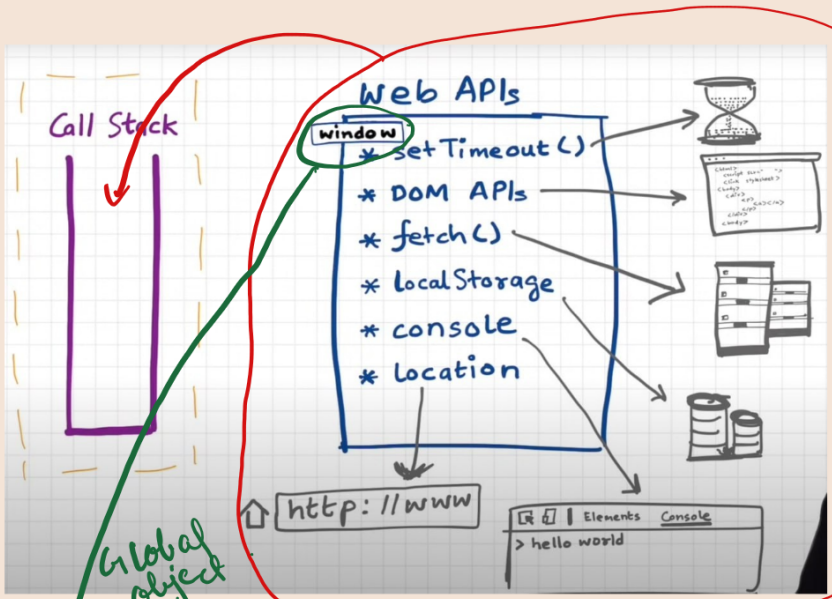
- If anything comes inside call stack, it quickly executes.

So, if want to keep a track of time and execute a piece of code after a certain time, we need some extra superpowers.




- The call stack is inside JS engine.
- JS engine is inside the Browser
- Browser has all the superpowers



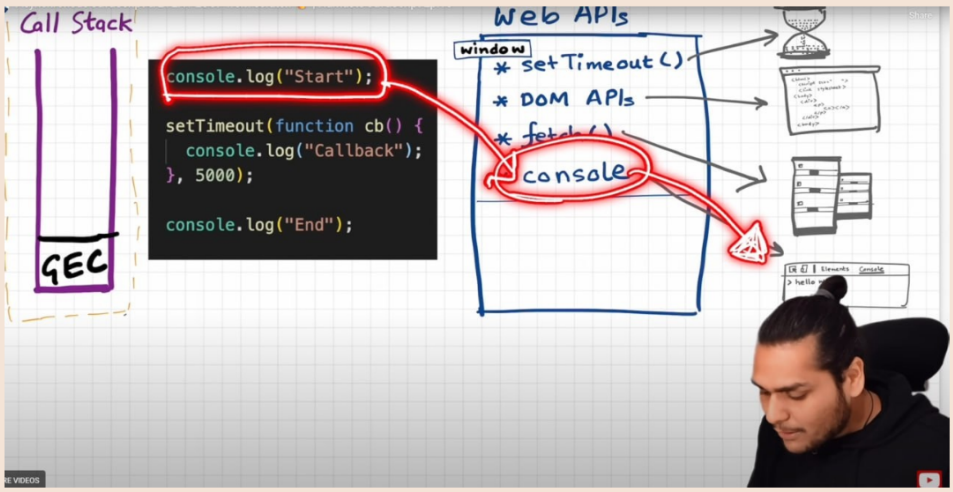

- To access all these superpowers of browser, we need Web APIs.
- Browser give JS engine all the superpower through a keyword called window.
  - Suppose we want to access the Web API setTimeout(), then we write window.setTimeout()

But because window is a global object and setTimeout() is present in global scope, we can use it without the word window and can simply write

window.setTimeout

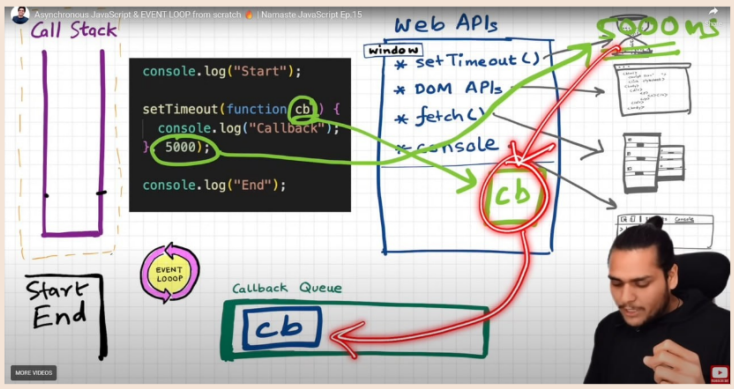
Example:-

```
① console.log("Start")
  setTimeout(function cb(){
    console.log("Callback"); 5000);
  console.log("End");
```




① This code will run line by line,
console.log("Start") it calls web API console and this internally makes a call to modify or log something on console.
Console is plugged through window to our code.

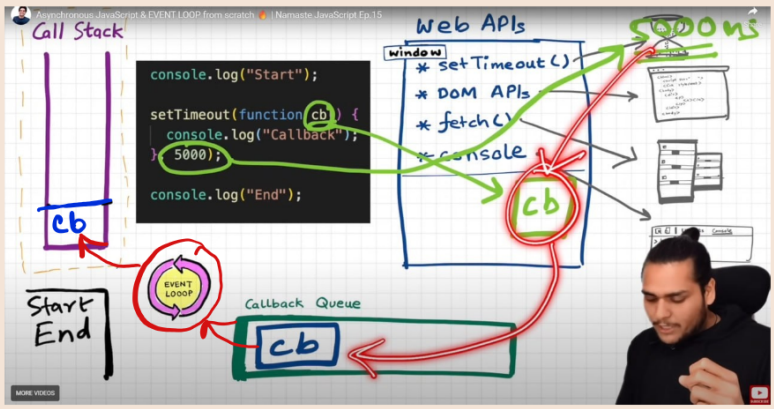
② setTimeout will call the web API setTimeout() and this gives us access to timer feature of the browser.
When we pass this callback function to setTimeout, it registers a callback and because we pass the delay, it also starts the timer of 5000 ms inside the timer.



③ console.log("End") again calls console and it logs End in the console.

④ After all our code is done executing GEC is popped off the call stack.

⑤ After 5000 ms expires, callback need to go into the call stack to execute. But it can't directly go into the call stack.




⑥ When the timer expires, the callback func is put into the callback queue.
The job of event loop is to check the call back queue and also see if call stack is empty then it puts the function from Callback queue to call stack.
This creates an execution context in call stack.

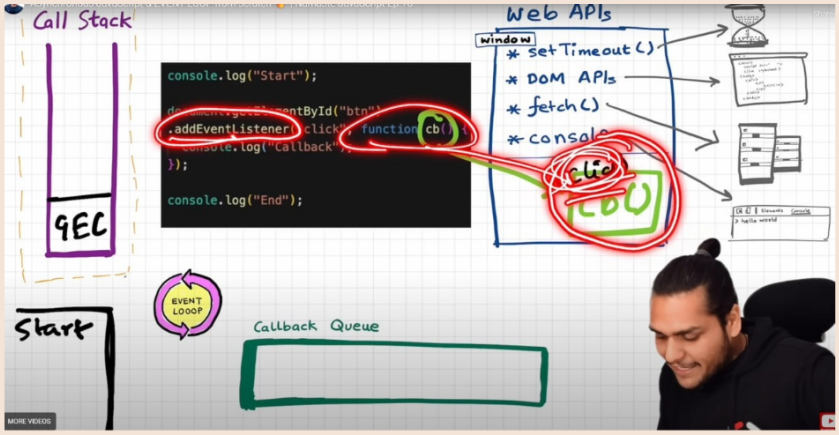
② Console.log("Start")
document.getElementById("btn")
.addEventListner("click", function cb(){
console.log("Callback");
});
Console.log("End");

① GEC is created pushed inside call stack, code is executed line by line

② Whenever we do document., it calls DOM APIs which in turn fetches something from the DOM (Document Object Model) → it is like HTML source code.

③ So whenever we call DOM API, it access the HTML code and tries to find a button with some ID and returns it.

④ addEventListner → registors a callback with an event to it, here the event is click.




⑤ It then goes on executes next line, when the whole code got run, GEC is popped out of call stack.

⑥ So the registered callback method inside the web APIs environment just sits over there and waits for click.

⑦ When the user clicks on the button, the callback method is pushed inside callback queue.

⑦ event loop – It has only one job to continuously monitor the call stack and Callback Queue. If the call stack is empty and event loop sees there's also a function in the Call stack waiting to be executed. It takes the function and pushes inside call stack. Then it gets quickly executed.

Q) Why do we need a callback queue? The event loop could have directly pushed it into call stack.

So if the user clicks the button multiple times. Callback function will get pushed inside the callback queue multiple times waiting to be executed.

Event loop then takes each callback function and pushes them into call stack. Slowly it executes all the callback functions one by one.

Also if there is lot of timers, lots of eventListners, it gets executed in the same way.

Example ③ fetch()

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

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

**fetch()** – It is a built-in JS function used to make HTTP requests and get data from a server. It returns a Promise, which lets us handle the response asynchronously.

# fetch in JS is browser's built in way of saying: "Hey server give me the data, but don't freeze the whole page while I wait"

# It is a function for making HTTP requests (GET, POST, etc.) and it returns a Promise. That means we don't get the data instantly, we get a shiny "promise" that says "I'll bring the result when it's ready".

⇒ we pass the call back function, which gets executed when the promise is resolved

**Steps:-**

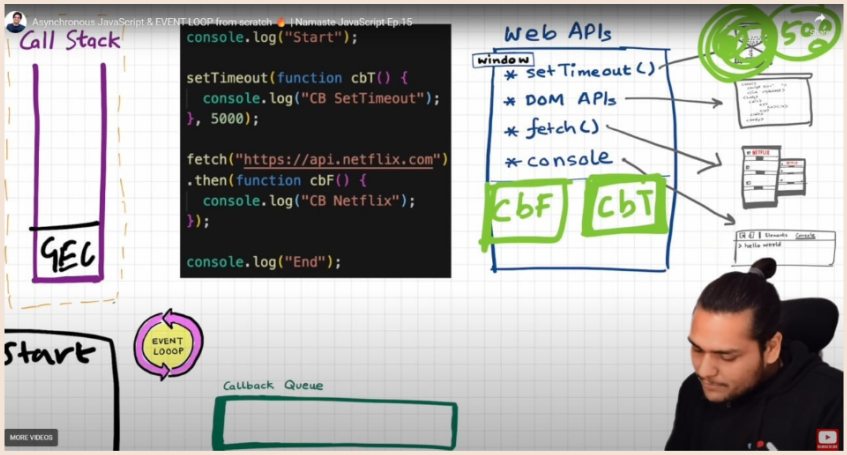
① GEC is created pushed inside call stack, code is executed line by line

② setTimeout will call the web API setTimeout() and this gives us access to timer feature of the browser.

When we pass this callback function to setTimeout, it registers a callback and because we pass the delay, it also starts the timer of 5000 ms inside the timer.

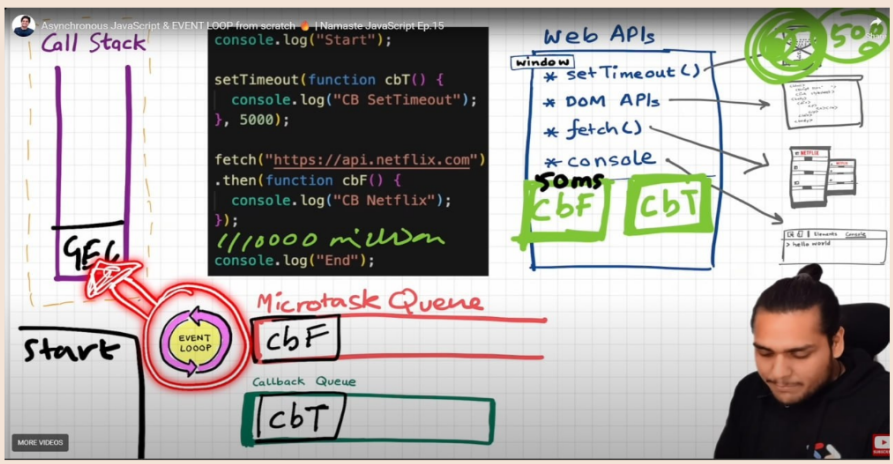
③ JS engine moves to the next line, we have fetch function, it also registers callback function in Web APIs environment.

④ Now we have two functions sitting in the web APIs environment CBF and CBT. CBT is waiting for the timer to expire. CBF is waiting for the data to be returned from netflix servers!




⑤ Both are waiting to be executed. Now which one to execute first?

Just like callback queue we have microtask queue. It is similar to callback queue but it has higher priority.




Event loop will continuously check when call stack is empty, if there are any functions in microtask queue and callback queue or not. Since microtask queue is of higher priority, functions inside are pushed first in the call stack and gets executed.

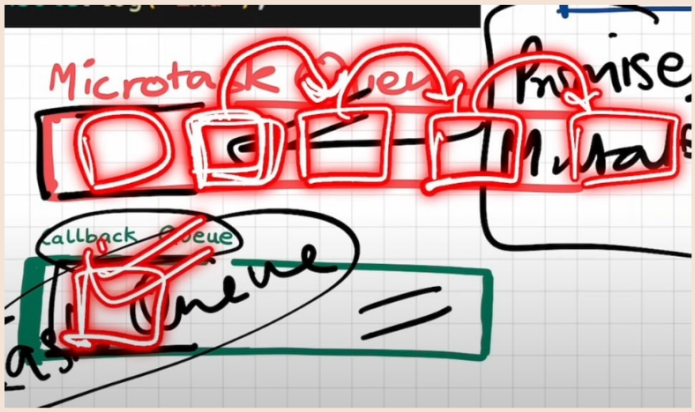
Q) What can come inside Microtask queues?

Ans – Callback functions which comes through promises and mutation observer goes inside the microtask queue.

All other callback functions goes inside microtask queue.

# Callback queue / Task Queue / Event queue.

Starvation – Suppose the microtask creates a new microtask in itself, which then creates another microtask and this continues. The task in callback queue will then never get a chance to execute.
Since microtask queue is of higher priority and needs to be executed first, there might be a possibility that callback function inside callback queue doesn't get executed for a long time.




Promises – A Promise in Javascript is an object that represents the eventual result of an asynchronous operation.
Think of it as a "contract" – it promises to either:

- resolve – succeed, give us data, or
- reject – fail, throw an error.

**Mutation Observers** - It is a built-in JS object that watches DOM, (the webpage's structure) and notifies us when something changes - like a spy sitting in the corner taking notes.