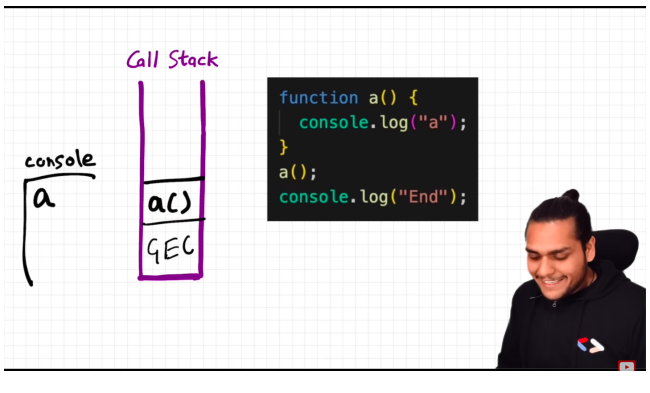
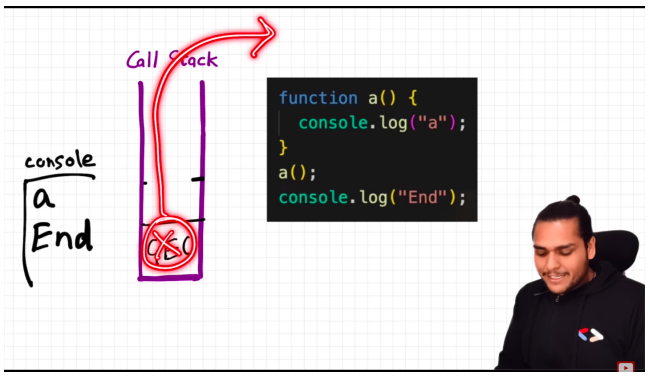
Asynchronous JavaScript and Event Loop

Saturday, 19 August 2023 1:16 AM

→ JS is a synchronous single-threaded language that means everything in JS happens inside a single callstack & everything happens in that callstack only.



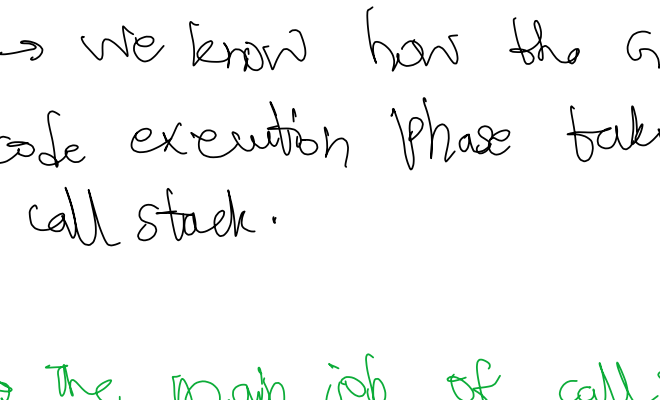


→ We know how the GEC is created & how memory creation & then code execution phase takes place. This all is handled by the call stack.

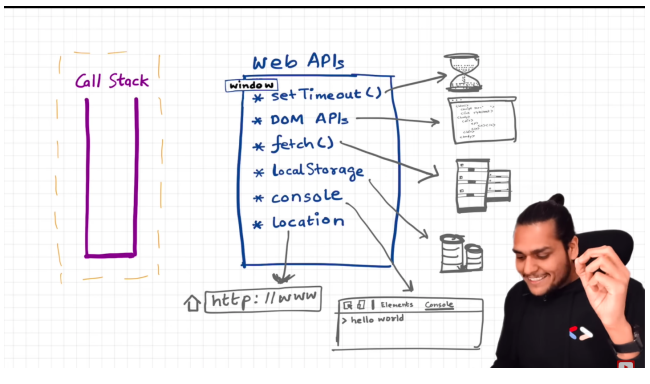
→ The main job of callstack is to execute whatever comes inside it. It does not wait for anything.

→ But what if we need to wait for something (eg: setTimeout)?
ans: The call stack can't wait for any task because it does not have a timer. So if we have to keep a track of time then we need the superpower of a timer.

→ A browser is an amazing creation which has access to various things like localstorage, geolocation, bluetooth, timer, etc.



→ But how does the JS engine gets access to these superpowers?
ans: In order to access the superpowers we have Web APIs.



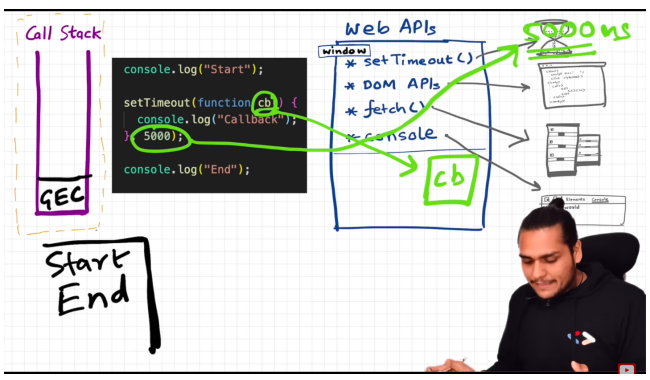
Note: Web APIs are not a part of JavaScript. These are basically superpowers that the browser provides. The browser only gives access to these superpowers to JS engine.

→ We get these superpowers' access inside the call stack through the global object.

→ window is the global object. So browser gives the JS engine access to these superpowers through the keyword window.

Eg: If we do window.localstorage inside our JS code, we get access to the localstorage.

Note: We know that we can use setTimeout or localStorage without the window keyword because these superpowers or web APIs are present in the global scope so they can be accessed without the window keyword.

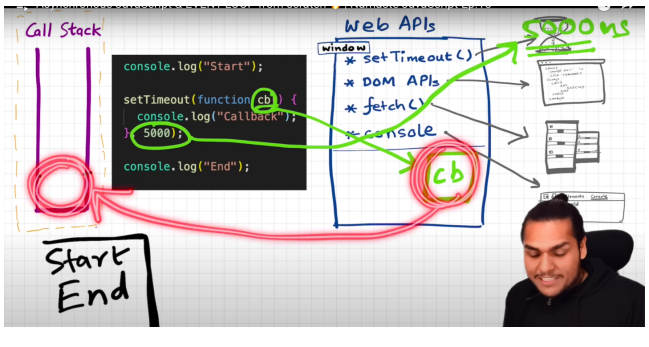


→ In above, first a GEC is created & pushed onto the call-stack, during code execution phase when JS encounters 'console.log' it accesses the web API & prints on the console.

→ Now, when JS engine encounters setTimeout, it accesses the setTimeout from web API and registers the callback fn & also attaches a timer to it.

→ Now, the JS engine does not wait for setTimeout & instead goes onto execute the next line of code and prints 'end'.

→ After this GEC is popped off the call stack.

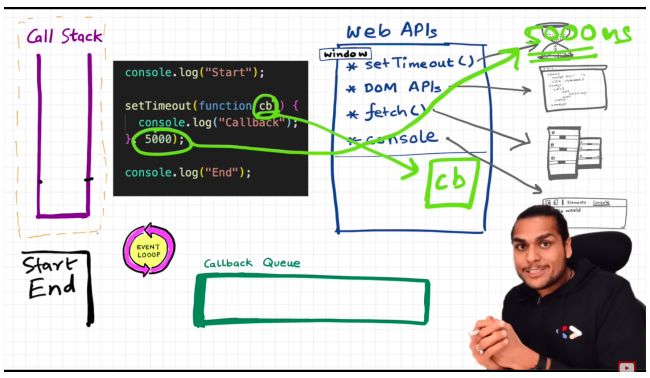


→ Now we know that after 5secs, we somehow need the callback fn inside the call stack. When we will get it inside

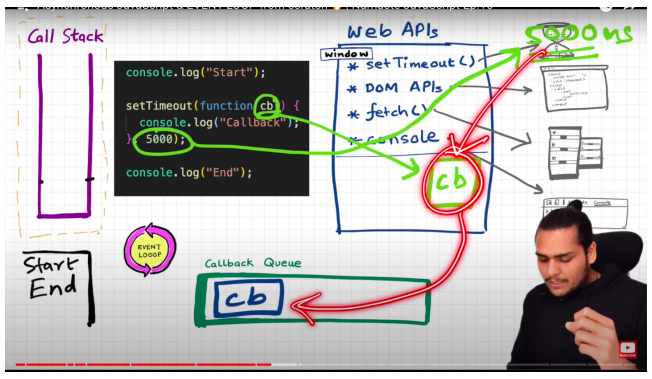
the call stack, the call stack will immediately execute it.

→ So how does the callback fⁿ actually go to the call stack?

ans.



Callback fⁿ goes to the call stack after 5secs by passing through the callback queue.

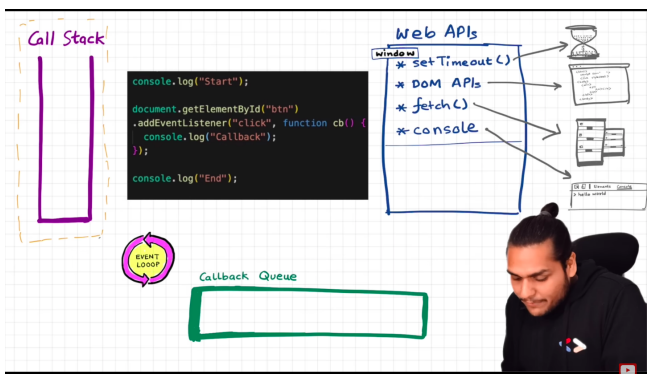


So after the timer expires, the callback fⁿ goes to the callback queue.

→ The job of the event loop is to check the callback queue & put the callback fⁿ inside the callback queue to the callstack.

→ So the event loop creates an execution context for the callback fⁿ & pushes it inside the callstack for execution.

→ Another example:



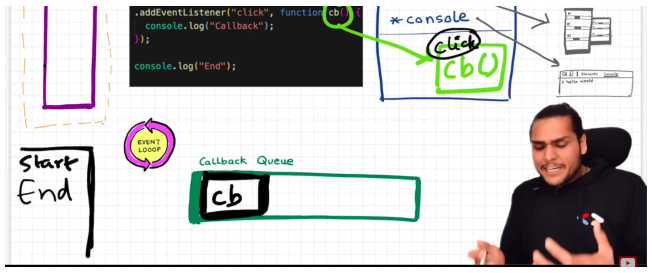
→ Now we know that the GEC will be created & pushed onto the callstack and 'start' will be printed.

→ When JS engine encounters `document.` it knows it has to access the Web API and a callback fⁿ is registered with the event of click attached to it.

→ Now, since JS engine doesn't wait for anyone, so it goes onto to execute the next line & 'end' is printed.

→ Now, the callback fⁿ sits inside the callback queue waiting for the event of click to take place.

→ This callback fⁿ will stay in the callback queue until we explicitly remove the event listener or close the browser.

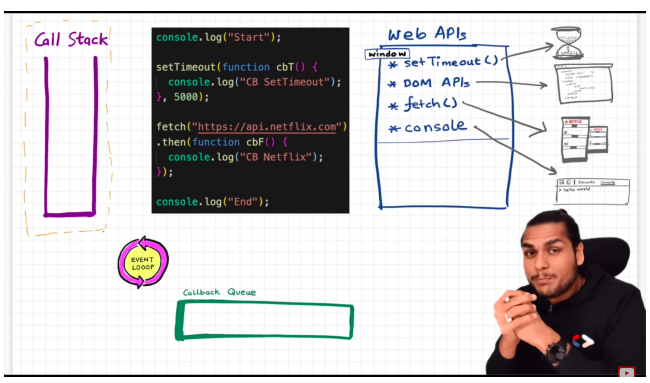


Note: The job of event loop is to continuously monitor the call stack & the callback queue. When the event loop sees that the call stack is empty & the callback queue has a callback fn waiting to be executed, then it pushes the callback fn inside the call stack by making its execution context. When this happens the callback function vanishes from the callback queue.
And after the callback fn finishes executing in the call stack, it is popped off from there as well.

Q: Why do we need a callback queue?

ans: We need callback queue because, for example if the user clicks on a button 5-6 times then the callback fn gets pushed to the callback queue 5-6 times waiting to be executed.
So, the event loop will continuously monitor the callback queue & the call stack, so if the call stack is empty & there is some fn inside the callback queue waiting to be executed, event loop creates an EC of that fn & pushes it to the call stack.
So, slowly the event loop removes the callback fn's from the callback queue & slowly executes the fn's inside the call stack.
In an actual web app there are many event listeners & setTimeouts, so we need a queue to manage them. That's why we need callback queue.

→ Another example:



→ fetch() fn requests an API call & returns a promise.
→ We need a callback fn which will be executed once the promise is resolved i.e when we get data from the url.

→ In above, the same things happen, first a GEC is created &

'start' is printed.

→ Then, cbT() is registered with a timer & setTimeout accesses the Web API.

→ JS engine does not wait & goes on executing the next line.

→ JS engine encounters fetch() and accesses the Web API which makes the network call. So cbF() is registered.

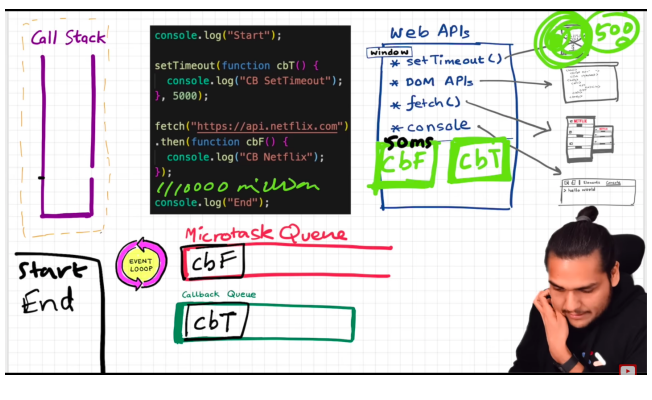
→ JS does not wait & goes on executing the next line and 'end' is printed.

→ Now, cbT() has 5000ms timer attached to it, let's assume that the data is returned from Netflix API in 50ms.

→ So, the promise is resolved in 50ms & the callback fn must go to the callback queue first instead of cbT() since it has a timer of 5000ms.

But this is not the case.

→ So what actually happens?



→ In JS engine, there is another queue called the Microtask Queue which has higher priority than callback queue

→ So, cbF() goes into Microtask Queue.

→ After that cbT() timer also expires & it goes to the callback queue.

→ Let's assume that after fetch() fn there are millions of lines of code, so event loop keeps on monitoring when the callstack becomes empty, in order to push the callback fn.

→ Finally when 'end' is printed, since microtask queue has higher priority so event loop pushes the EC of cbF() inside the callstack & cb Netflix is printed.

→ After that EC of cbF() is popped off the call stack & event loop pushes EC of cbT() inside the callstack & cb setTimeout is printed & cbT() 's EC is also popped off the call stack.

→ This is why fetch() works differently from other web APIs.

Note: All the callback f's which come through promises will go to the microtask queue. Also, the callback f's which come through the mutation observer will go through the microtask queue while all the other callback f's go through the callback queue.

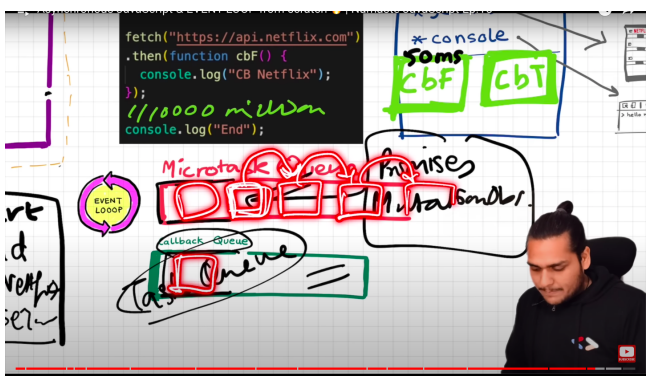
Q: What is a mutation observer?

ans. MutationObserver observes the changes made in the DOM tree.

Note: Callback queue is also known as task queue

Note: The event loop only executes callback f's from the callback queue once all the callback f's inside the microtask queue have been executed.

→ Starvation of f's in callback queue:



→ Let's assume if the callback f's inside the microtask queue keep on creating more tasks in the microtask queue, then event loop won't be able to execute callback f's of callback queue, therefore there will be starvation of f's in callback queue..

Q: When does the event loop actually start?

ans. Event loop is a single thread loop that is almost infinite. Its always running & doing its job.

Q: Are only asynchronous web API callback registered in the web API environment?

ans. Yes, the synchronous callback function that we pass inside map, filter and reduce aren't registered in the web API environment

Q: Does the web API environment store only the callback f^n & push the same callback to callback queue/ microtask queue?

ans. Yes, the callback f^n's are stored and a reference is scheduled in the queues. Moreover, in case of event listeners, the original callback f^n's stay in web API environment forever. That's why it is

advised to explicitly remove the event listeners when not in use so that the garbage collector does its job.

Q: What if the delay for setTimeout is set for 0ms? Then will the callback move to the callback queue without any wait?

ans: The callback fn needs to wait until the callstack is empty. So the 0ms callback might have to wait for 100ms if the call stack is busy.