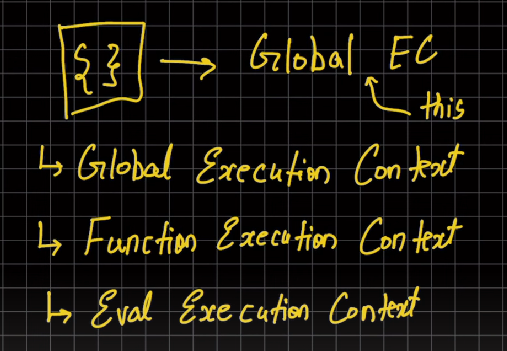
<https://www.freecodecamp.org/news/how-javascript-works-behind-the-scene-javascript-execution-context/>

<https://dev.to/jahid6597/javascript-execution-context-a-deep-dive-4kno>

[JavaScript Execution Context: Behind the Call Stack | by Rabail Zaheer | Medium](https://medium.com/@rabailzaheer/javascript-execution-context-behind-the-call-stack-19f253aad0a4)

[Understanding Execution Context in JavaScript: A Comprehensive Guide | by Sejal | Medium](https://medium.com/@SejalPande1994/understanding-execution-context-in-javascript-a-comprehensive-guide-dfd9231ed197)

**Javascript Execution Context :-** It means whatever file u have made, how JS will run/execute it.

1. Whenever u give any code to JS, firstly **global execution context** is made. It is refered by `this` variable. For browser, `this` is Window object.
2. JS is single threaded
3. 
4. JS run our program in 2 phases –



In **memory creation phase**, memory is allocated to all the variables which we have declared and assigns the initial values for the variables , functions, etc.. For variables value given is 'undefined' and for functions they are set to their function definition

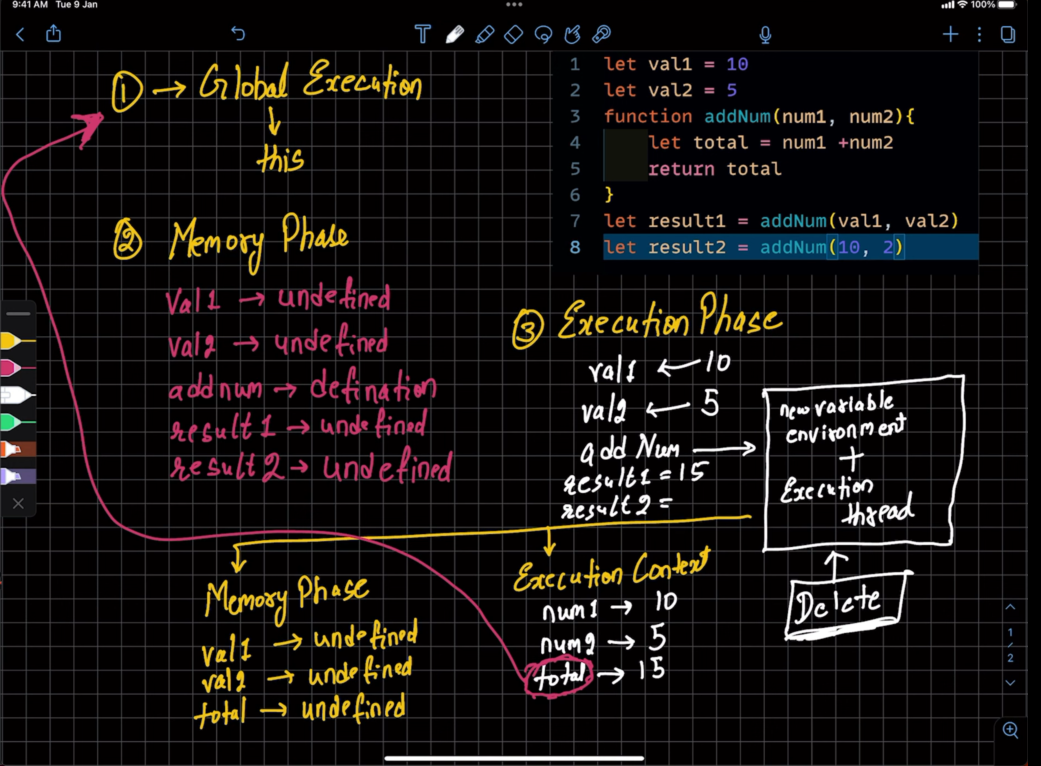
**Execution phase** :-

After the creation phase completed, the execution phase begins where actual values of the variables are set. When a function is called it creates a new 'Function Execution' context which consists of two phases (they works same as explained above but only for the particular function scope):

1. Memory phase

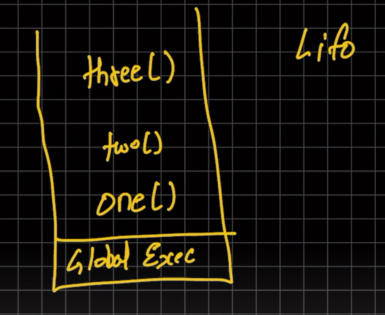
2. Execution phase

The return value of the function is passed to the Parent Execution Context (global in our example)

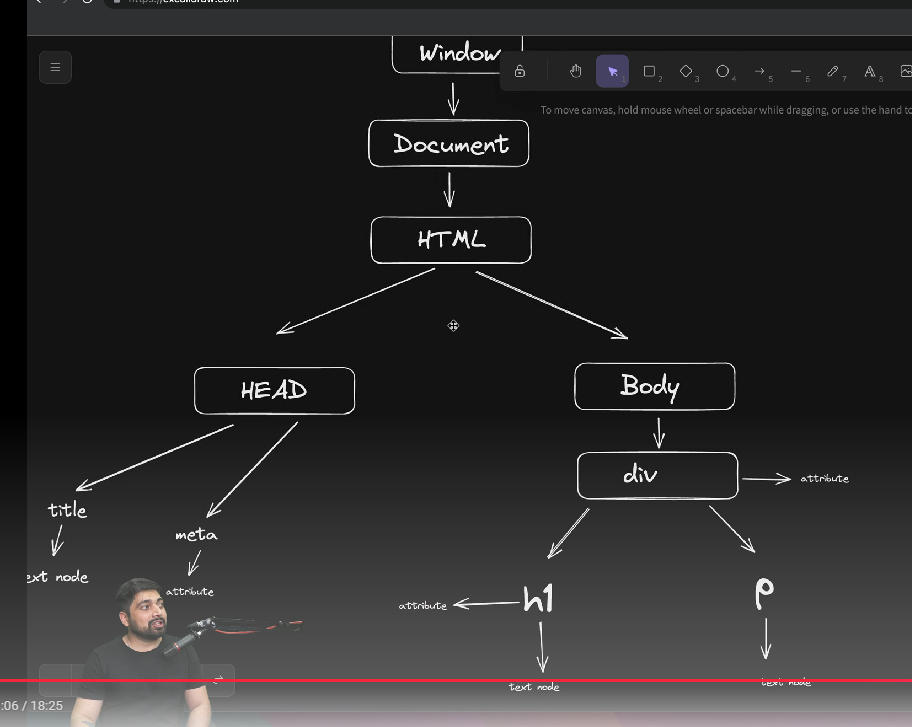


**Call Stack :-**

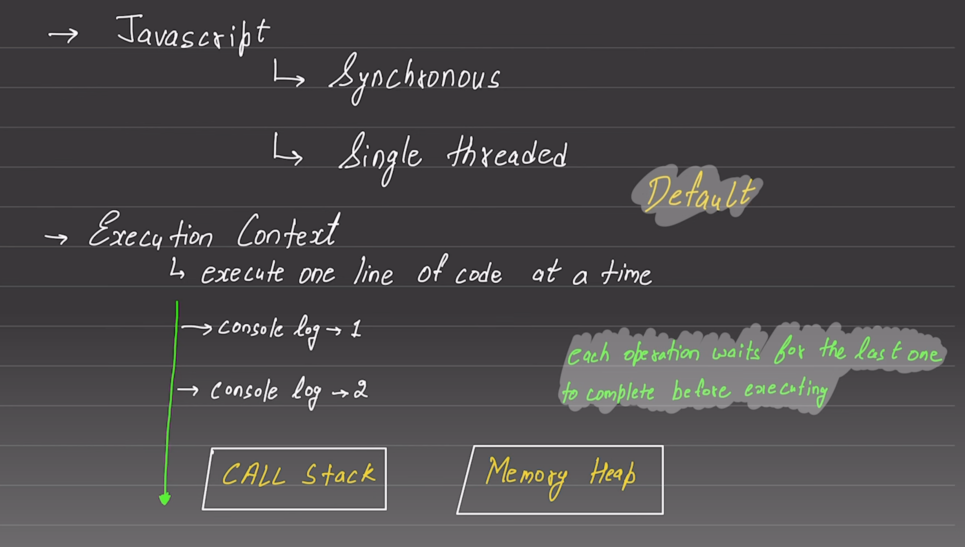
Initially there is Global execution context. Now suppose we call function one() which calls two() which again call three(). So as per LIFO the execution context of three() is on top so it will be executed first and come out of stack first.

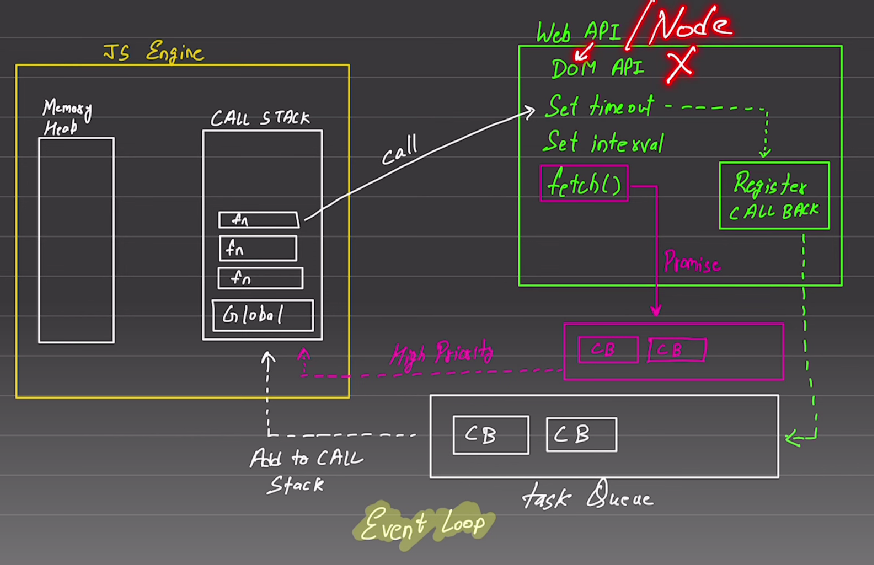


**DOM Manipulation**

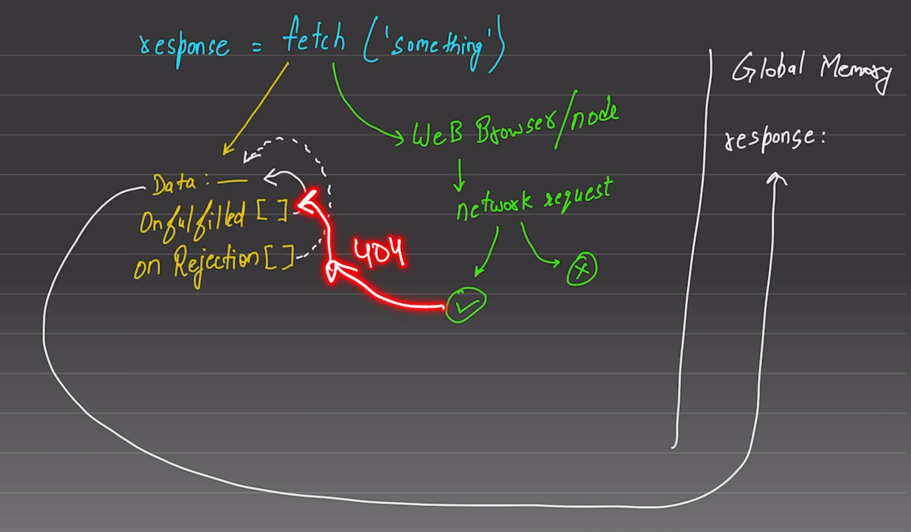


**ASYNC JAVASCRIPT**



* If u see only javascript engine that is slow becoz its single thread, but it is not used alone. It is available in runtine environment like browser or nodejs
* 
* Our program itself cannot read a file, to read file we have to give context to our kernel. Then kernel goes and read the data and again execution is given to our program. Till then ur program cannot do anything.
* 
* Web API is in browser or we use environment like Node
* Task queue is responsible to make javascript fast and asynchronous
* Suppose ur function calls setTimeout(), that call is transferred to web api. All these information about async task is registered inside “Register call back”
* Whenever the event is triggered or timeout happen, then “Register call back” will add the call back immediately inside the Task Queue.
* Task Queue will add all ur call backs inside the call stack. Since it is added on the top of call stack so it is immediately executed also.
* Suppose u log(1) then setTimeout(0, log(2)) then log(3)

Its output will be 1, 3, then 2 becoz although it is 0 second timeout but still call made to Web API then Register callback add it to taskQueue. While all these steps happen 3 gets printed.

* For fetch(), high priority queue was made.
* So we can say that complete JS runtime allow asynchronous but default JS engine is synchronous
* 
* When we use fetch() it starts working in 2 parts
* One part goes to handle browser/node api requests and other part goes to reserve space in memory

Onfullfilled [] array is for resolution of promise, onRejection[] for rejection of promise

We cannot push values in these array

If network request is made and we get some response then it goes in OnFullfilled array, otherwise if req is not made it goes in OnRejection array

* Value of data is reserved in memory its initial value is empty/undefined

OnFulfilled[] and OnRejection[] has function which is responsible to fulfill the data, then this data will fulfill the reponse which we get from fetch()

* 