**Event Loop**

* As we know that javascript is synchronous single threaded language and each line of code in js is executed one after the other.
* These executions are handled using callstack.
* callstack work in a manner of stack data structure LIFO.
* By default Global execution context is added in callstack and then each functions are pushed after it.
* Consider below example:

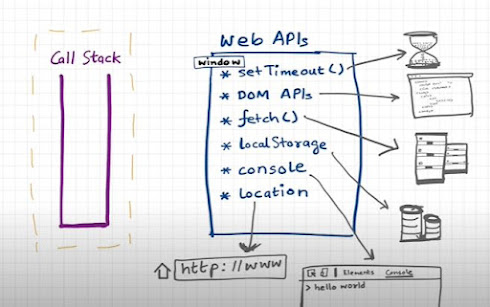
A picture containing text

Description automatically generated

* In the above example Javascript first created a Global Execution Environment and than pushed it as a first entry in the callstack.
* After global execution environment function a in pushed and accordingly once the function is executed it is moved out of the call stack and finally console.log of GEC is executed

In the above example, we have the have callstack is used internally to executed synschronous js, then how does js works asynchronously.

To understand this, lets go through below terms which are more frequently used in js.



* In the above example, the web API's which we most frequently use is not a part of javascript instead it is the browser which communicate with these web api.
* These web api's are provided through window object in javascript.
* Javascript contains the window object globally.

Diagram

Description automatically generated

* In the above example, we have a setTimeout method which will delay execution of its callback method by 5 sec.
* As you usual GEC is added in the call stack, then its first console statement is executed.
* As soon as it sees a setTimeout, browser starts the timer and as Js dont wait for its execution it prints the second console statement and GEC is removed from the call stack.
* Once the timer ends, the callback function of setTimeout is moved in the **callback queue.**
* **It is the event loop which act as a gatekeeper between callstack and callback queue and it pushed the callback method into callstack accordingly.**

Text

Description automatically generated

* In the above example, we are using another **window object document.**
* Here callback of listener waits for the click event from user and the moment user clicks it, it moves the callback to callback queue.
* Event Loop again waits for the call stack to get empty as it continuously monitors callback queue it picks up and moves it to callstack again for execution.

Example: Fetch Api - Microtask queue

Text

Description automatically generated

* Unlike setTimeout or document object, fetch API works different.
* In the above example, once the global execution is executing, fetch api waits for an api to return its result at the same time setTimeout timer start.
* Once the API returns result its callback is stored in a special area Microtask queue rather than callback queue.
* Microtask queue is given higher precedence than callback queue.
* If callstack is empty and if there are pending callbacks to be executed from both the queue, like fetch api callback in microtask and setTimeout callback in callbackqueue than microtask callbacks are pushed first into callstack and once all micro task queue executes then callbacl queue callbacks are pushed into callstack.

Few points to remember -

* All the promise API's and Mutation Observer are stored in microtask queue
* In above all examples, callback queue is known as Task Queue.
* There is a possibility of Starvation of access to callback queue if all tha callbacks in microtask queue contains a sub tasks which is again stored in microtask queue then there may be a scenario where callback queue will never get access to execute.