JavaScript

"Everything in JavaScript happens inside an Executing Content"

"JavaScript is a synchronous single threaded language.

Execution Content:

Execution content is like a box which have two components:

Memory	Code
It is the place where variables with key : value pair and function are stored. It is also called variable environment.	One line of code is executed at a time. It is also called thread of execution.

Synchronous Single Threaded means?

It means that javascript executed one line of code at time and synchronous means after executing one line of code it goes to the another line.

How JavaScript Code Executed ?

JavaScript code runs into steps -

1- Memory Execution phase
2. Code Execution phase

```
var n = 2;
function square(no) {
  var ans = no * no;
  return ans;
}
var square2 = square(n);
var square4 = square(4);
```

Memory	Code
n = undefined function {...} square2 = undefined square4 = undefined	

Code square2

Memory	Code
no = Undefined ans = Undefined 3.4	=> 2

Memory	Code
no = Undefined ans = Undefined no = 2 ans = 16	

=> Firstly every variable has undefined in memory execution
=> When function call whole new execution content created and after completion it will deleted.

=>

Memory	Code
n = 2 function {..} square2 = 4 square 4 = 16	

→ After Code Execution

After complete execution Execution context is deleted

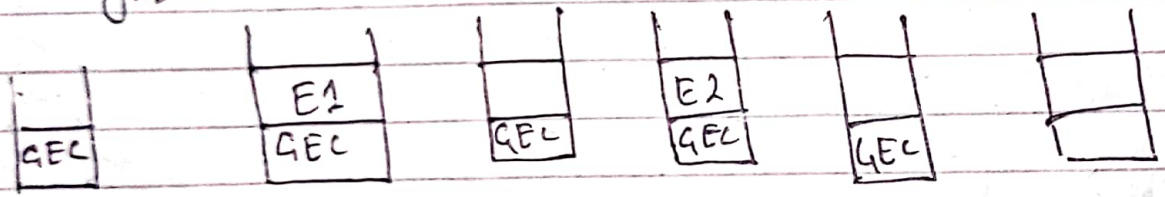
Every execution content is done in stack - this is called call stack -

-> Firstly we have a Global Execution content - for write code -

-> If any function is invoked we push ↙ execution content into the call stack -

-> After completing we push out to stack (pop operation)

-> After whole code execution we pop out the global execution content and call stack is empty -



E1 -> square 2 ();
E2 -> square 4 ();

0. Call stack also know as Execution content stack
1- Program stack
2- control stack
3- Runtime stack
4- machine stack.

Hoisting in Javascript:

```
1-  getName();
2-  console.log(x);
3-  var x = 3;
4-  function getName()
      { console.log("Hello world");
      }
```

if we comment the line no 3 then:
output will be:

output,

```
Hello world
undefined
```

```
Hello World
x is not
defined.
```

Defination: Hoisting is the phenomina by which you can access the variables & function even before its initialized it. and put some value in it-

```
var x = 3;
var getName = () => { ... }
var x = function () { ... }
```

=> In all cases the JS consider it as the variable and if you try to access it will give you undefined. This is because in Memory Execution phase all variable home value of undefined.

Name

function _() { ... } only in this syntax JS understand as function memory allocation phase.

Shortest Javascript Program:

An empty JS code is the shortest program - when we haven't code anything and just attach file with .html file and open and go to console and type Window => we see a bunch of code -
this => " " " "

Window: window is a global object which is created along with global Execution context - It is created by JS engine -

this === window

whenever a Execution context is created (this) is created along it even for the functional (Global Execution context) and global level this points to global object and window that is in care of browser -

Global space:

```
var x = 10;          ——>  global space
function a() {       ——>  Global space
    var y = 10;      ——>  not a global space
}
```

Simple words Any thing which is not inside a function called global space

Undefined: Undefined is just a placeholder of value - When Memory Execution phase working it gives all variable the value of undefined -

Not Defined: Not defined is anything that doesn't take any memory - but undefined has some memory -

Scope: It means where you can access a specific variable or function in our code

Lexical Environment:

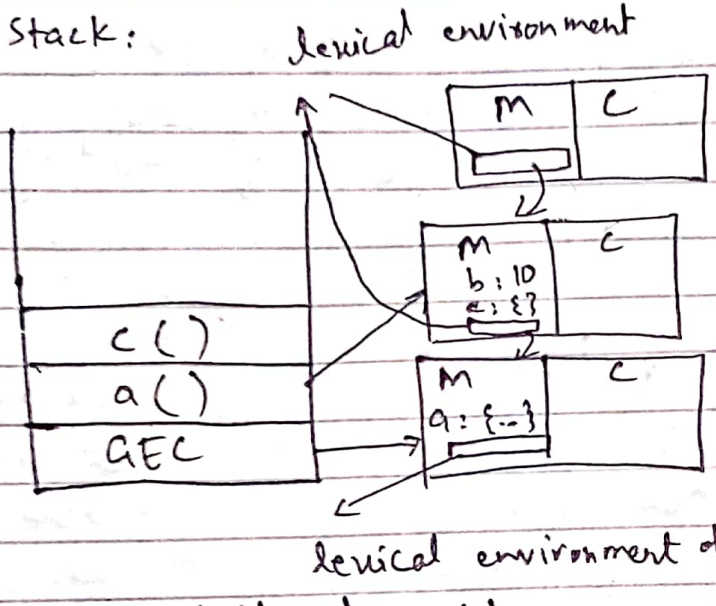
Whenever an execution content is created a lexical environment is also created so lexical environment is local memory along with its lexical environment of its parent -

C is lexical sitting inside the function a-

OR Simple: Lexical means a sequence / Hierarcy -

In code: Where a code is physical present inside the code -

Call Stack:




```
function a() {
  var b = 10;
  c();
  function c() {
  }
}
a();
console.log(b);
```

c() is child of a()
a() is child of GEC() ∴ As we it follows the hexachiy
GEC have parent is null:

How Lexical Environment used:

Let suppose in above c has log(b) and suppose b is not inside a it is defined in global scope - The JS when try to execute console.log(b) it find where b lies which local memory. Fiistly it goes it lexically parent and try to find b - if it is not then goes to the global - if it here it show output.

and if it is not there than it goes to the parent of GEC which null than it simple say the b is not defined. This way of finding called the scope chain.

Topic (Let & Const)

Let and const come in ES6.
Let and const are hoisted just like var but not that way that var is. when i try to access a it gives me Reference Error. when we like let it place variable in separate memory or simple allocate separate memory not in GEC (Global Execution Context). Let also got undefined but on separate space.

Code:
```
console.log(a);
let a = 10;
console.log(b);
var b = 5;
```
Output,
Reference Error
undefined.

Temporal Dead Zone:

The time since when this let variable was hoisted and till it initialize some value. The time between that called temporal Dead Zone.
Simple words: Undefine to initialize time between called temporal dead zone.

Whenever you try to access the variable in temporal dead zone it gives you the Reference Error - we can also not re declare the let variable -

```
console.log(a);
let a=10
```

Reference Error:

```
const a=10;
a=10
```

Type Error:

```
const a;
a=10;
```

Syntax Error:

BLOCK SCOPE:

BLOCK: Block is also known as compound statement. It combine multiple JS statement into group - we need to group these statement together so that we can use multiple statement in a place where Javascript expects only one statement -

e.g `if (true)` it gives error

`if (true) { true }` -> it runs fine -

`{}` is a block that group multiple statement -

Block Scope:

Block scope means where all variable and functions can be accessed.

Shadowing:

It occurs when variable has same in block as scope same as in outer (global scop). than variable will be Showdow-

e.g:
```
var x = 10;
{
  var x = 20;
  log(x);
}
log(x)
```
output
20
10

```
let x = 10
{
  var x = 20;   --> ILLEGAL Shadowing
  log(x);
}
```

```
var x = 20
{
  let x = 30;
}
```
Perfectly fine-

Each block {} has its own lexical scope-

CLOSURES IN JAVASCRIPT

A function bind together with its lexical environment.
OR: Function along with its lexical scope form closure.

Uses of Closures:

1- Module Design Pattern 2- Currying 3- Function like Once
4- Memoize 5- Maintain state in async world 6- setTimeout
7- Iterators ...

Garbage Collector:

It is a program in browser or JS that frees the unutilize memory.

Function Statement:

```
function name() { ... }
```

this way of creation a function known as function statement.

Function Expression:

```
var variable = function () { ... }
```

Initillize a variable using function called function expression.

Difference b/w function Statement and function Expression.

The major difference is hoisting.

```
a();
b();
function a() { ... };
var b = function () {...};
```

a	run fine
b	Not defined

Function declaration:

Function statement and function declaration is same.

Anonymous Function:

A function without name called anonymous function.

1) `function () { ... }`

An anonymous function has not has its own identity if you try like above syntax it create an error.

→ Anonymous are better to use when ~~variable~~ function should

use to initallize value to variable.

```
var variable = function () {...}
```

Name Function Expression:

if we give to function while initillize value to variable by using a function:

```
var variable = function name() {...}
    variable();
```

First Class Function:

Function pass as another function OR function inside another function arguments called first class function.

Call back Function:

A callback function is a function pased into another function as an argument.

why we remove event Listener:

Event listener are so heavy means takes alot of memory.

Event Loop:

Web APIs:

* setTimeout
* DOM API
* fetch
* localStorage
* Console
* location

→ These are not part of javascript these are part of browser.
→ Browser allow us to use these web APIs using a keyword **window**.
JS wrap all these web API in a global thing called window which allow us to use it.

Example:

In first line when the program meet console it use the console web api and print. Second line use Timer Web API. Then go line 5 and again use console Web API then GEC from stack is poped out. But Timer of 5 sec push into the stack but it does not go directly to the stack it goes to the callback queue then event ~~listener~~ loop check this execution content then send to the stack

```
1- console.log("start");
2- setTimeOut(function cb(){
3-   console.log("Hello");
4- }, 5000);
5- console.log("End");
```

Example 2:

First Global Execution context is created then line 1 code web API is used. Then an event listener is added by using DOM web API an click event is added an call back function is also added - After that log is printed. GEC is pop out.

when user click the btn the call back function is added to the callback queue and event loop check this and send to the stack (call stack) - if user click multiple times then event loop added in b that times one by one to the call stack -

Code:

```
1- console.log("start");
2- document.getElementbyId("btn", function cb(){
3-     log("call back");
4- }
5- log("End");
```

Output

```
Start
End
Call back
```

Example 3:

First all thing happens like before A Global Execution content is created then start point out. Then setTimeout web API we and CBT function is placed inside web APIs. Then fetch APIs used and CBF () is placed inside web APIs.

Code:

```
console.log ("Start");
setTimout ( function CbT()
{
  console.log ("CBT");
}, 5000);

console.log ("End");

fetch ("---")
then ( function(){
  log("CBF");
}
console.log ("End")
```

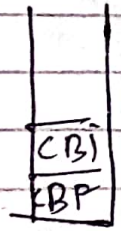
Microtask Queue:

Fetch ()
(High Priority)

Callback Queue:

Callbacks
(low Priority)

Call Stack:

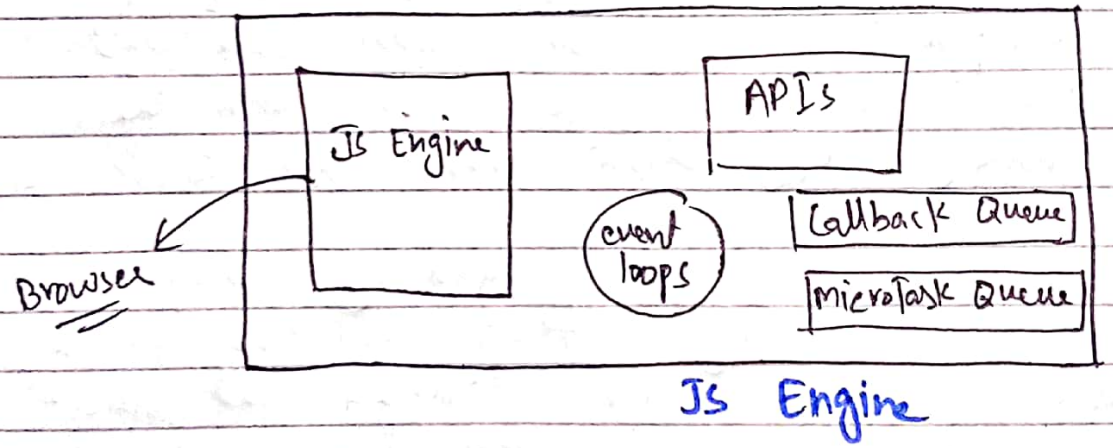


whenever call stack is empty the event loop give priority to the microtasks queue and then callback queues.

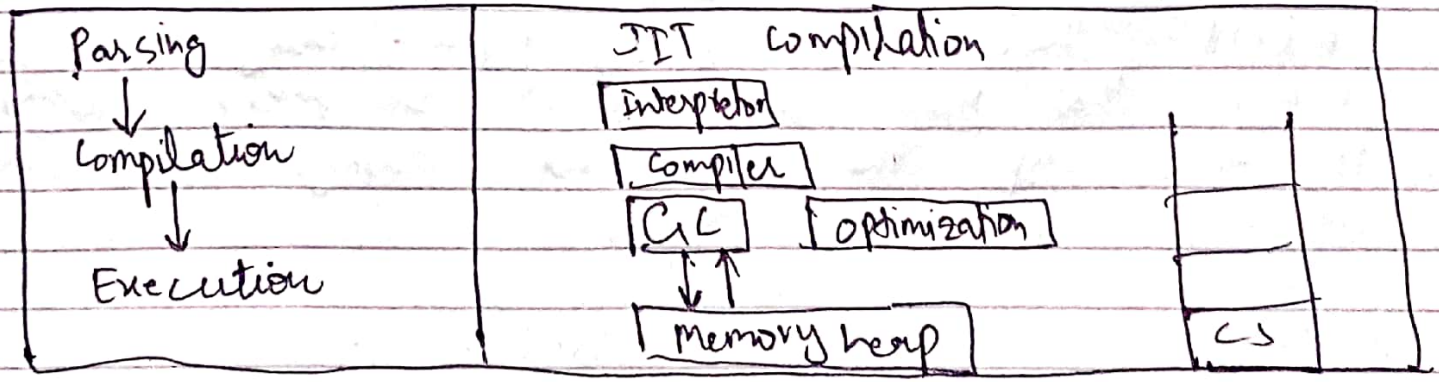
What comes inside MicroTask Queues ?

All the callback function which come through a promise can go to the MicroTask Queue - or mutation observer can go too.

Javascript Runtime Environment:



JS Engine



1- In passing let a = 10 the tokens are generated - let is one token, a is another token = is another and 10 is another token - After that syntax passing is done each code is converted in AST (Abstract syntax Tree).

2- The compilation and execution is done side by side Javascript do Just in Time compilation. It combines an interpretor or compiler - Intrepretor run line and send to compiler than send the byte code into the Execution. It also do inlining, copy elision and inline caching -

3- Execution is never done without memory heap and the call stack - Memory Heap is the place where variables and function are stored - Call stack is just a place where we have execution content - There are one more thing is Garbage collector that uses the Mark & sweep algorithm to free up the unused things in memory heap -