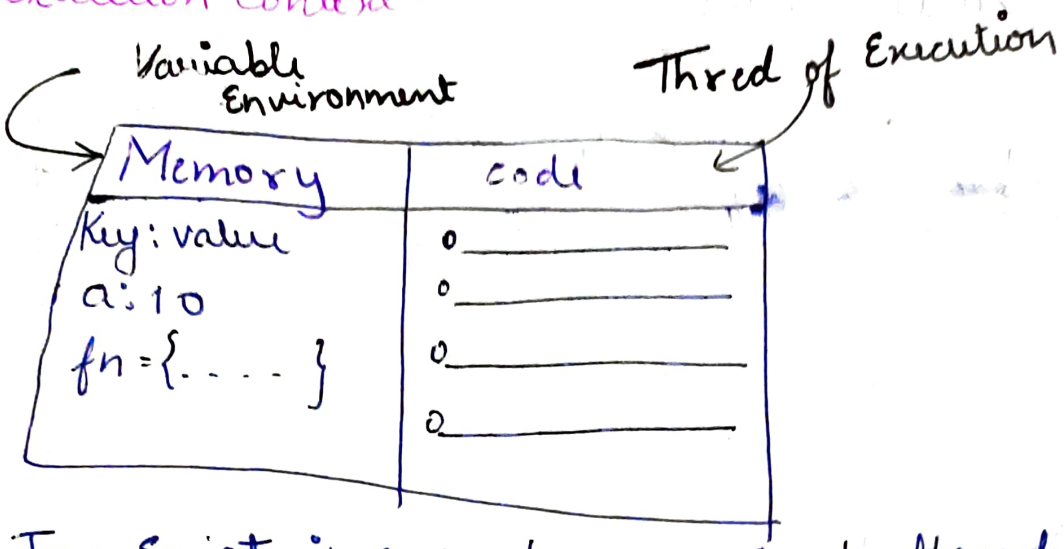
Everything in JavaScript happens inside an Execution Context.



→ JavaScript is a synchronous Single-threaded language

Single threaded → Js can execute one command at a time.

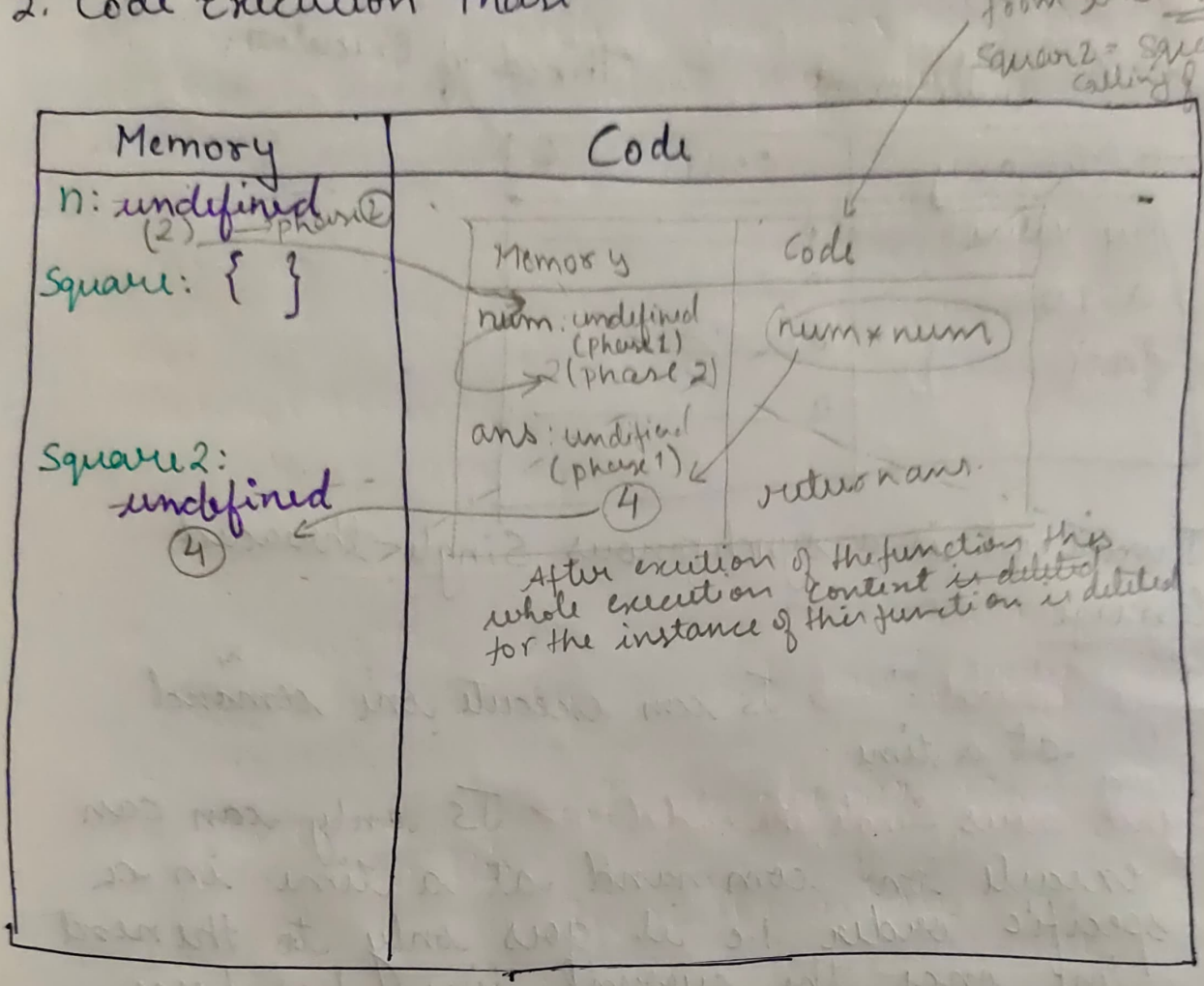
Synchronous Single threaded → Js ~~only~~ can can execute one command at a time in a specific order i.e it goes only to the next line once the current line has been finished executing.

Lets understand with example.

```
1  Var n = 2
2  function square (num) {
3      Var ans = num * num;
4      return ans;
5  }
6  Var square2 = square (n);
7  Var square4 = square (4);
```

There are two phases

1. Creation Phase or Memory Creation Phase.
2. Code Execution Phase



in the code line 6 → var square2 = square(n);

→ as soon as the square function is executed a new execution phase is created

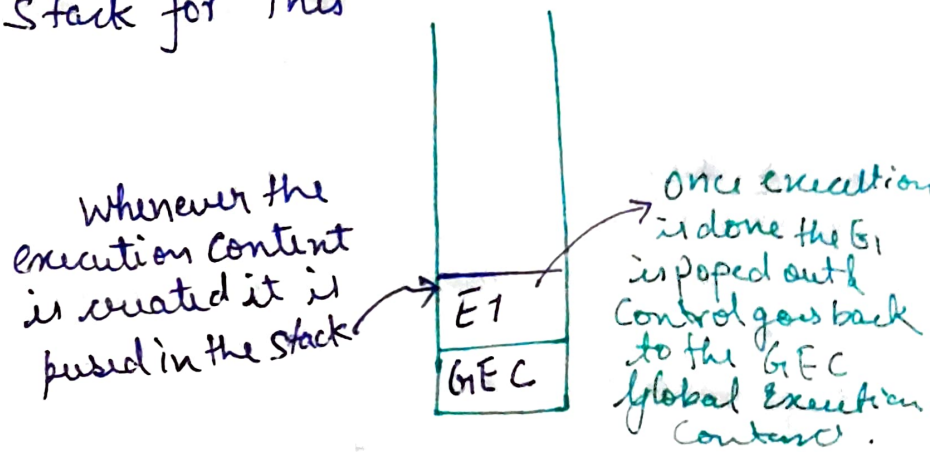
→ in first phase the memory is allocated the variables & functions.

→ Now in Code Execution Phase then code will be executed by the values which is been passed as an arguments.

→ Then return → this return keyword tells that you are done with your work now just return the whole control back to the execution phase Context where the function was invoced

Call Stack

It handles everything to manage this execution Context creation deletion & ~~dele~~ control, it does it by manages Stack for this



Call stack maintains the order of execution of execution of execution contexts.

Call Stack also known by

- Execution Context Stack
- Program Stack
- Control Stack
- Runtime Stack
- Machine Stack

Hoisting

Hoisting is a phenomenon in JS by which you can access variable & functions even before you have initialized it or put any value to it

```
1. getName();
2. console.log(x);
3. console.log(getName);
4. var x= 7;
5. function getName(){
6.     console.log("Something");
7. }
```

Something 6
undefined 2
f getName(){ 3
console.log("Something")
3

- In the phase 1 of the memory creation JavaScript will allocate memory to each and every variable and function.
- It places a special placeholder Undifined to the variables.
- In case of function the whole code is put in the memory. Even before we are trying to run the code.

Var →

```
console.log(c)
var c= 9
```

→ Script runs →

```
var c
console.log(c)
c= 9
```

that's how it's ide interpited
This is the reason it's undefined.

undefined

Any variable declared with var undergoes hoisting:
During the execution of script all variable declaration with var moved up to the top of the script

while in case of let & const Hoisting doesn't occurs.

Const and let are not binded to the window

let → value can be reintialized
Const → value can't be reintialized } Difference

For Arrow Function

```
~~Var~~ ~~getName~~()
  function getName2() {
    }

  Var getName = () => {
      Console.log("Something")
      }
```

If we write an arrow function it will be undefined because getName above behaves like another variable.
Same way we do → var getName2 = function() {
} this will also be treated as variable getName2.

Window

Window is a global object which is created along the global execution context.

So whenever any JavaScript program is run a global object is created, a global execution context is created and along with that global execution content a this variable is created.

Global object created in case of Browsers is window

at this Global level or at the base level in the Global Execution Context this === window

this === window
→ true

Anything that is not inside a function is a global space.

```
Var a=10
function b(){
    var x=10
}
```

a & b will be attached to the window but not x.

```
Console.log(window.a) → 10
console.log(a) → 10
Console.log(this.a) → 10
```

Undefined vs Notdefined

When it allocates memory to all the variables and functions, to the variables it tries to put a placeholder. It is like a place holder which is placed in the memory that special keyword is undefined.

```
console.log(a) → undefined
var a=7
```

```
console.log(x) → notdefined.
```

The Scope Chain, Scope & Lexical Environment

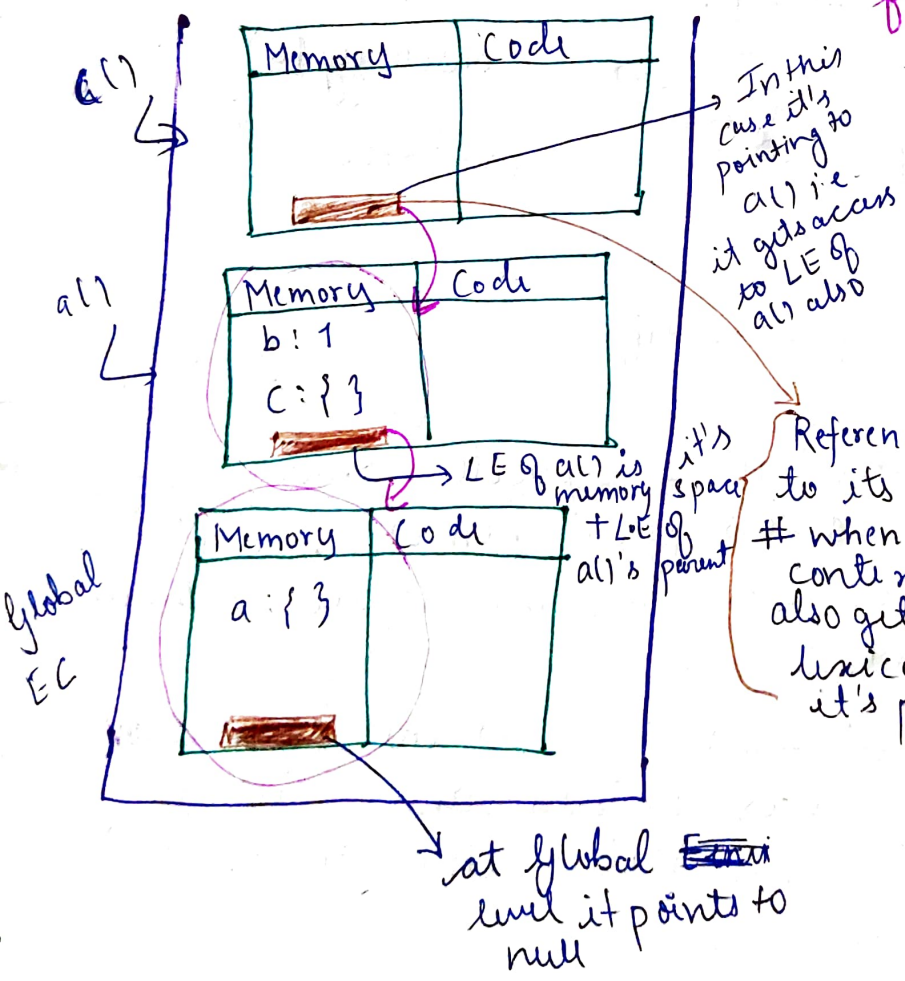
Scope → Scope is means where you can access a specific variable or function in a code

→ What is the scope of this variable — means where can the variable be accessed

→ Is variable inside scope of function - means can the variable be accessed inside the scope of the function.

Scope is directly dependent on the Lexical Environment

Lexical Environment




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

it's space

Reference to lexical environment to its parent

when ever the execution context is created you also get reference to the lexical environment to it's parient.

Scope Chain

Chain of all the lexical environment and the parent references is Scope Chain

Let & Const in JS Temporal Dead Zone

What is a Temporal Dead Zone?
Are let & const declarations hoisted?
Syntax Error Vs. ReferenceError Vs. TypeError?

→ let & const declarations are Hoisted. but they are declared very differently than var declarations.

→ these (let & Const) are in the temporal Dead Zone for time being.

```
console.log(b)
let a=10
var b=100
```
} undefined

```
console.log(a)
let a=10
var b=100
```
} Reference Error.

Global
└ b: undefined

Script
└ a: undefined

→ they are stored in a seperate memory space & you cannot access that memory space before you have put in some value in them so that is what is hoisting in let.

Temporal Dead Zone → is the time this let variable was hoisted and till it is intialised some value that time in b/w is that temporal dead zone.

→ When ever you try to access a variable inside a temporal dead zone it gives a reference error.
'Cannot access 'a' before initialization'

→ * We cannot do redeclaration of let. It will throw an error (this time Syntax error

→ * ~~Const~~ Const is very Strict

```
Const b
b= 1000
```

SyntaxError: Missing initializer in Const declaration.

Type Error → when you are reinitializing a const variable to a var.

Syntax Error → Suppose when you are not initializing the value to const variable at the time of declaration.

Reference Error → When we try to access a variable in the temporal deadzone.

1. Const → whenever you can use const you should use it, whenever you want to put in some value which is not changed later use const.
2. Let → if not const try to use let wherever possible because let has a temporal deadzone & you will not run into ~~excepted~~ expected error.

Best way to avoid this Temporal dead zone is to always put your declarations & initialization at the top of the scope. so that as soon as your code start running it hits the initialization part first then you go to the logic.

You can also call it as we are ~~Shri~~ Shrinking the temporal Dead zone Window to zero while moving our intization at the top.

Block Scope & Shadowing

~~if (true)~~ {

~~var~~ a = 10;

{ } ← This is a block.

Block is also known as

We group multiple statement in block so that we can use it where javascript expects one statement.

```
if (true){
//
var a=10;
console.log(a)
}
```

We combine multiple statement in a block so and use it where Javascript expects a single statement.

```
{
  var a = 10;
  let b = 20;
const c = 30;
}
```

Block
b: undefined
c: undefined
a

Global
a: undefined.

Shadowing

```
var a = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(a); ⟶ 10
}
console.log(a) ⟶ 10
```

Shadowing (var) If you have same named variable outside the block then that variable shadows the variable & the value will be changed or updated to that you give it in the block as in above example. This is called Shadowing. (This is because both variable are pointing to same reference

Shadowing (let)

```
let b = 100          → this has the script scope.
{                    → this as a Global scope.
  var a = 10;
  let a = 20;        → this b has a blog scope
  const c = 30;
console.log(a) ⟶ 10
console.log(b); ⟶ 20
console.log(c); ⟶ 30
} console.log(b); ⟶ 100   → from script scope.
```

~~Sahe~~

Shadowing (Const)

```
Const c = 100 ——→ script scope
{
  var a = 10;
  let b = 20;
  const c = 30 ——→ block scope
  Console.log (c); ——→ 30
}
console.log (c) ——→ 100 (b'script scope.
```

Shadowing Concept works same for the function as well.

Illegal Shadowing

→
```
let a = 20;
{
  var a = 20
}
```

Error: SyntaxError: 'a' has already been declared.

```
let a = 20
function x () {
  var a = 20
}
```

Not illegal reason being var is functional scope here in this case it is accesible only inside function.

```
var a = 20
{
  let a = 20;
}
```

reason being let is the block scope and it will not effect the outside scope so it will not throw error

Lexical Block Scope

```
const a=20;
{
  const a=100;
  {
    const a=200;
    console.log(a)
  }
}
```

Block
a: 200

Block
a: 100

Script
a: 20

Closures

```
function x(){
  var a=7;
  function y(){
    console.log(a)
  }
  y();
}
x();
```

Closure → A function bind together with it's lexical environment.

→
```
function x(){
  var a=7;
  function y(){
    console.log(a);
  }
  return y;
}
var z=x();
console.log(z)
```
→
```
f y(){
  console.log(a)
}
```

```
function x()
{
  var a = 7;
  function y(){
    console.log(a);
  }
  return y
}
```

```
var z = x()
console.log(z)
z() ⟶ 7
```

```
{ y(){
  console.log(a)
  }
```

When the function is returned from another function they still maintain their lexical scope.

Function along with it's lexical scope bundled together forms the closer.

Example

Uses of Closures:

- Module Design Pattern
- Currying
- Functions. like once
- memoize
- maintaing state in async world
- set Timeouts
- Iterators

setTimeout + Closures

```
function x(){
  var i=1
  setTimeout(
    function(){
    console.log(i);
    },3000);
    console.log("Shaln")
  }
   x();
```

this forms a closer

setTimeout takes this callback function and stores it into some place & attaches a timer to it and JavaScript proceeds.


```
function x(){
 for(var i=1; i<=5; i++)
 setTimeout(function(){
    console.log(i)
  },i*1000
  console.log("Shaln")
   }x();
```

Shaln
6
6
6
6
6

It is working this way because of the closures. As it (closures) is a function along with it's lexical environment so, here when SetTimeout takes this function and attaches a timer and store it somewhere ~~and~~ so that ~~to~~ function remember the reference to 'i' in above example

so when loop runs the first time it makes a com copy of function attaches a timer and also remember the reference of i
They are pointing to same reference of i because the environment for all of these functions are same.

→ Secondly JS doesn't wait for anything it will run the loop again and again the setTimeout will store that function and JS will move it will not wait for the settime out to execute or expires.
So when the timer expires the val of i in memory is 6.

Quick fix

```
function x(){
  for (let i=1; i<=5; i++){
    setTimeout (function(){
      consdo.log(i);
    }, i*1000);
  }
  consdo.log("Shaln")
}
x();
```

Shaln
1
2
3
4
5

→ let is block scope, so whenever the loop runs the 'i' is the new copy each time setTimeout runs it has new copy of 'i' with it.

Solution using closures

```
function x()
  {for (var i=1; i<=5; i++){
    function close(x)
      {setTimeout(function() {
          to console.log(x);
        }, x*1000);
      }close(i);
      console.log("Shaln")
      }
    x();
```

Now here using close we kinda created a new copy of "i"

→ so everytime this close function is called it the setTimeout function is stored in a seperate memory

Function Statement aka Function Declaration

```
function a() {
  console.log("a called");
}
```

Function Expression

```
var b = function() {
  console.log("b called")
}
```

The difference between Function Statement and function Expression is Hoisting.

function Statement

```
a()

function a() {
  console.log("a called");
}
```

a called.

Function Expression

```
b()
var b = function() {
  console.log("b
  called");
```

Error TypeError: b is not defined.

Function Statement

During the memory creation phase 'a' is created a memory and the function is assigned to 'a'

Function Expression

While in case of a function expression this b is treated like any other variable, it is assigned undefined initially until the code hits the function line itself, so JS executes code line by line and assign function to this variable until it is undefined and if you try to call undefine it gives error.

Anonymous Function

Anonymous functions are used when the functions are used as values

```
function () {
}
```

} UNcaught Syntax Error: Function Statements require a function name.

Anonymous function is a function without a name.

```
var b = function () {
  console.log("b called")
}
```

Named Function Expression

```
var b = function xyz() {
  console.log("b called")
}
```

b() ⟶ b called

xyz() ⟶ xyz is not defined Reference Error.

Difference between Parameters & Arguments.

```
var b = functin (param1, param2)
{ console.log ("b called")
}
b(1,2);
```

Parameters

Arguments

Function as First Class $Citizen

The ability of function to be used as values and can be passed in as an argument to another function and can be returned from the function, this ability is known as first class citizin in javascript.

Callback Functions

As functions are first class citizens in JavaScript, so that means you can take a function and pass it into another function and when you do so this function which you pass into another function is known as a callback function. This callblack function let us achive Asynchronous behaviour.

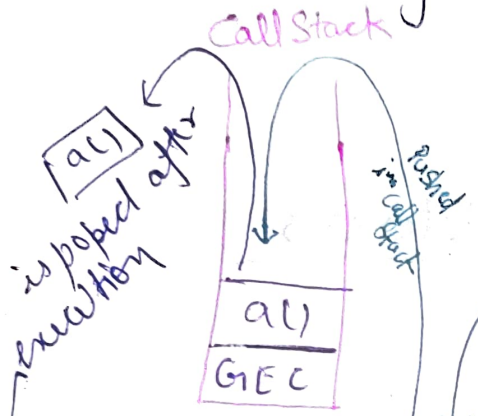
```
setTimeout(function(){
    console.log("timer");
},5000);

function x(y){
  console.log("x");
  y();
}

x(function y(){
    console.log("y");
});
```

o/p
x
y
timer → after 5sec.

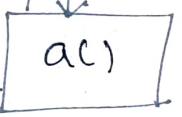
JavaScript has one CallStack and you can call it also a main thread

Asynchronous JavaScript

JavaScript is a synchronous single threaded language. It has one call stack and it can do only one thing at a time

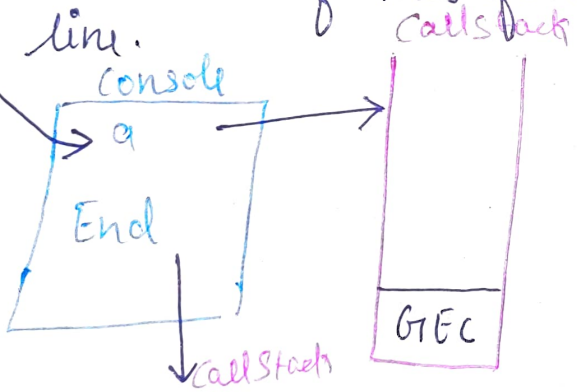


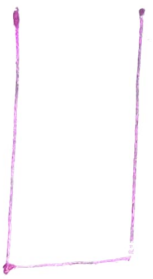

```
function a(){
  console.log("a");
}
a() function call
console.log("End")
```



Execution context is created for function a() to execute this function

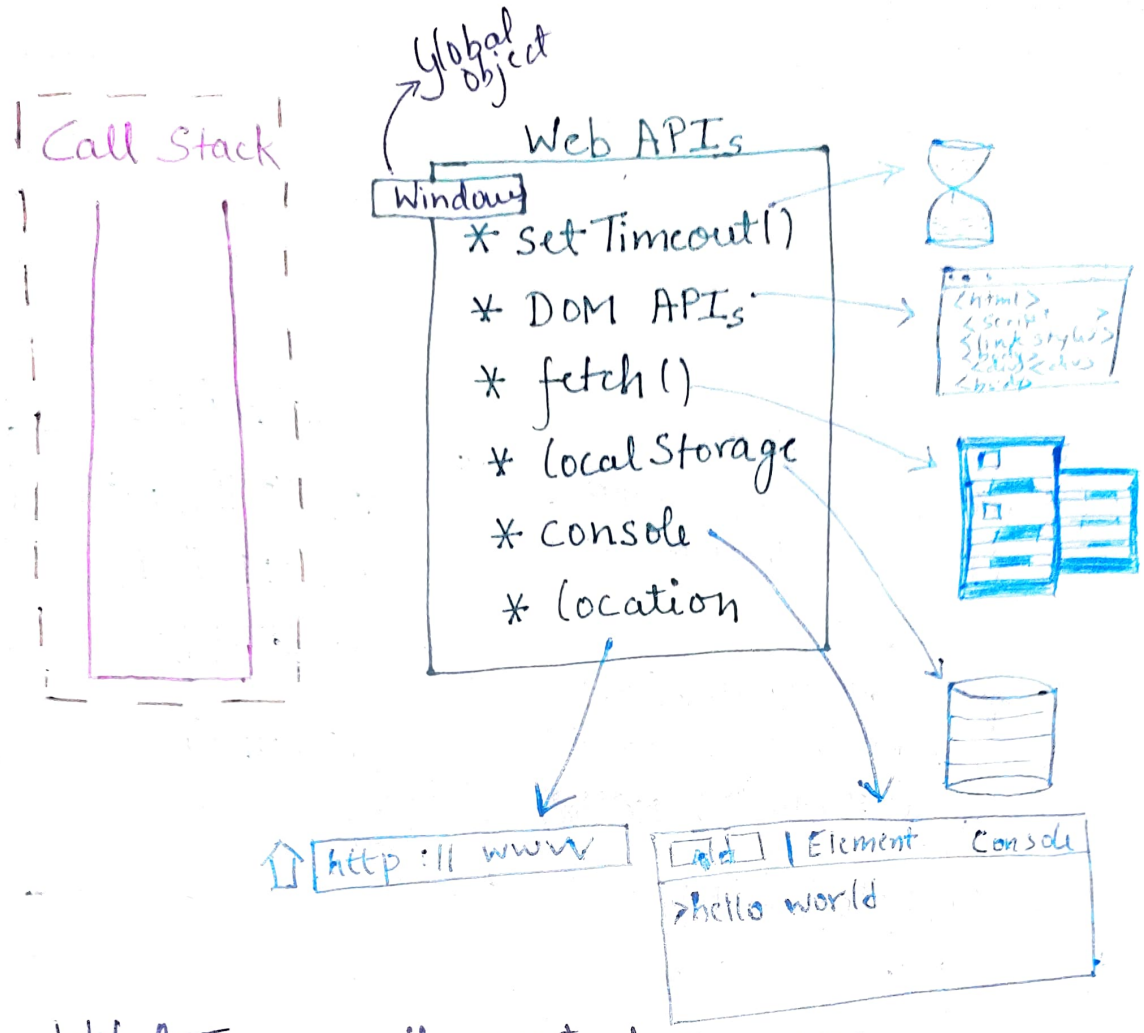
After a() has pushed into the Call Stack the code of this function is executed line by line.



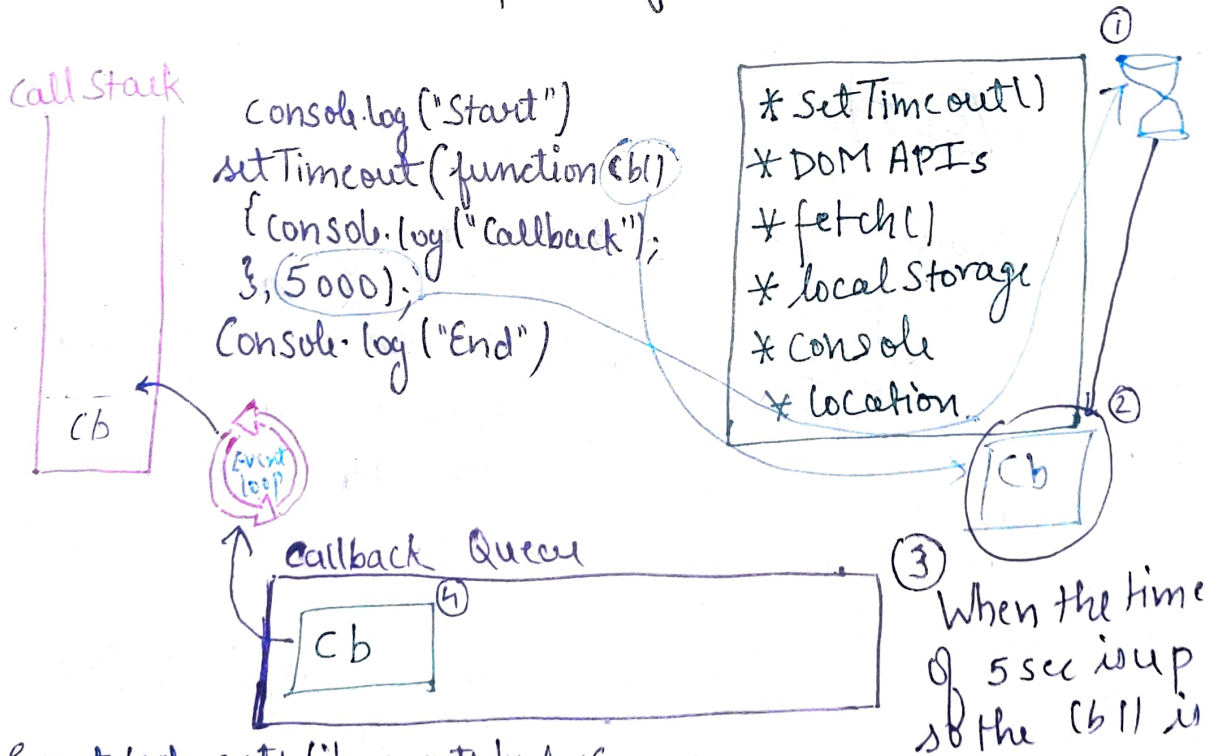


After Executing their's nothing left to execute so the global execution context pops out of the call Stack

→ So the job of this Call Stack is to quickly execute the whatever comes inside it. It does not wait for anything.



Web APIs are the part of Browser.

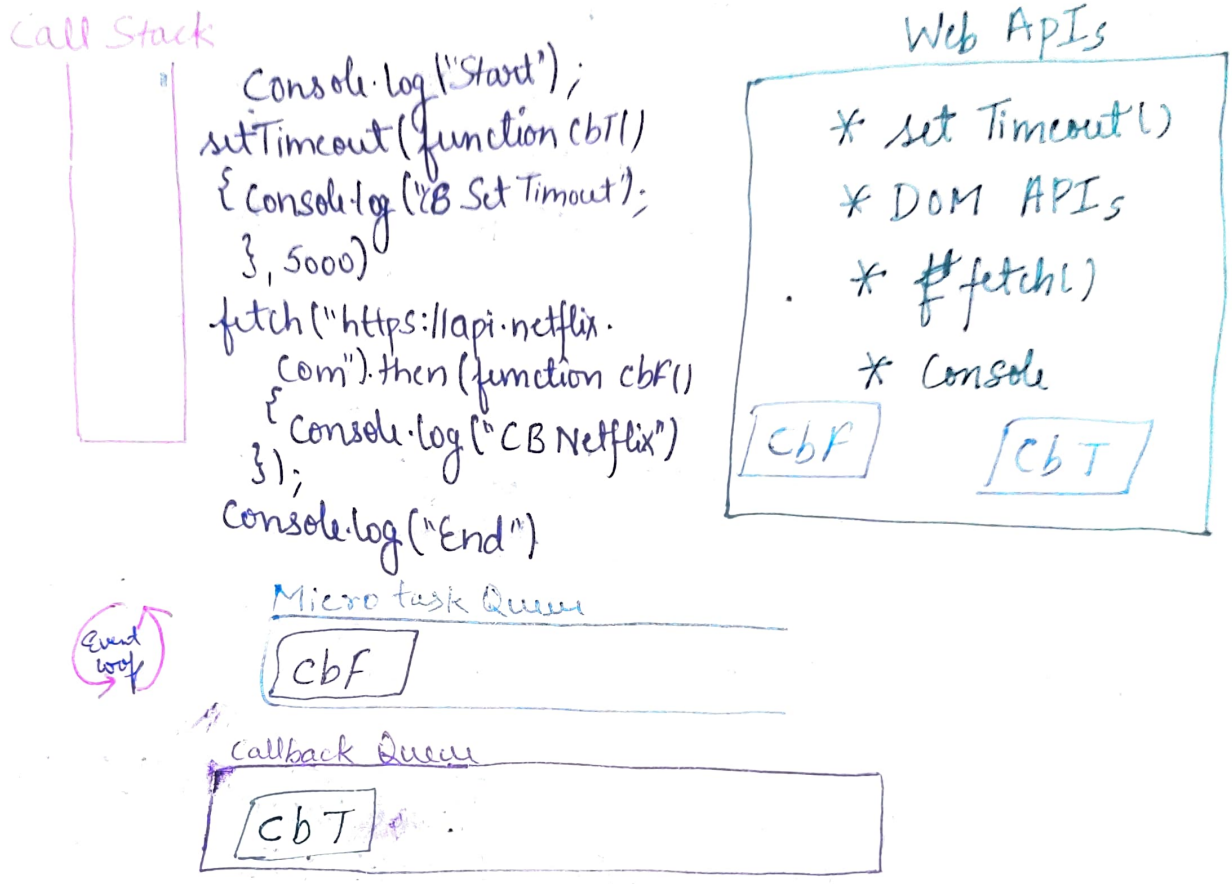


Event loop acts like a gate keeper & checks the callback Queue and it pushes it into the call stack.

When the timer of 5 sec is up so the cb() is ready to execute so it is pushed in callback Queue.

Start
Sto End
Callback.

fetch() Example



Micro task Queue → Has a higher priority

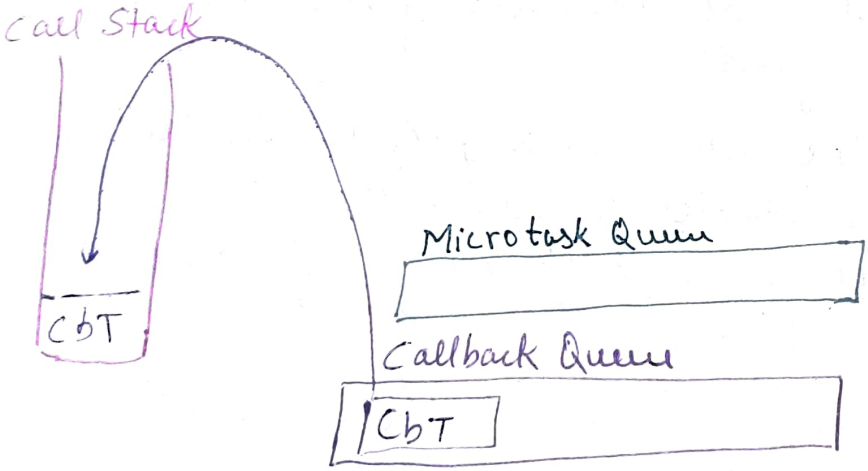
→ CbF will go to micro task Queue and the job of the event loop to check weather the call stack is empty and push the e the function i.e it gives ~~change~~ chance to all the function waiting in the Micro task Queue.

→ The JS code will run and suppose the timer also expires e then that function will be pushed inside the Callback Queue.

→ Suppose JavaScript Engine finish executing the code and reaches the last line Console.log("End") After that the call stack becomes empty

→ Here ~~Micro~~ then eventloop gives the chance to the CbF function from the microtask Queue as it has higher priorty.

Start
End
CB Netflix



Start
End
CB Netflix
CB SetTimeout

Micro Task Queue

→ It has higher priorty

→ All the callback function which comes through promises come under Micro Task Queue.

→ Mutation Observer → keeps on chucking weather there is some mutation in DOM tree or not. if there is some mutation in the DOM tree it can execute some callback function.

Callback Queue / Task Queue

→ setTimeOut()

All the other Callback like

→ setTimeout()

→ callback which is from DOM APIs like the event listener goes inside the callback Queue.

STARVATION

→ Just because this eventloop gives chance to micro task first.

→ Suppose if the micro task present in Micro Task Queue, creates a new micro task creates a new micro task in itself and it goes on like this.

→ So the task in the callback Queue will never get a chance to execute, because the Micro Task has more priorty, that means task waiting in Callback Queue does not get a chance for a long time. This is known as Starvation of the Callback Queue. or So Starvation of the task inside a Callback Queue.