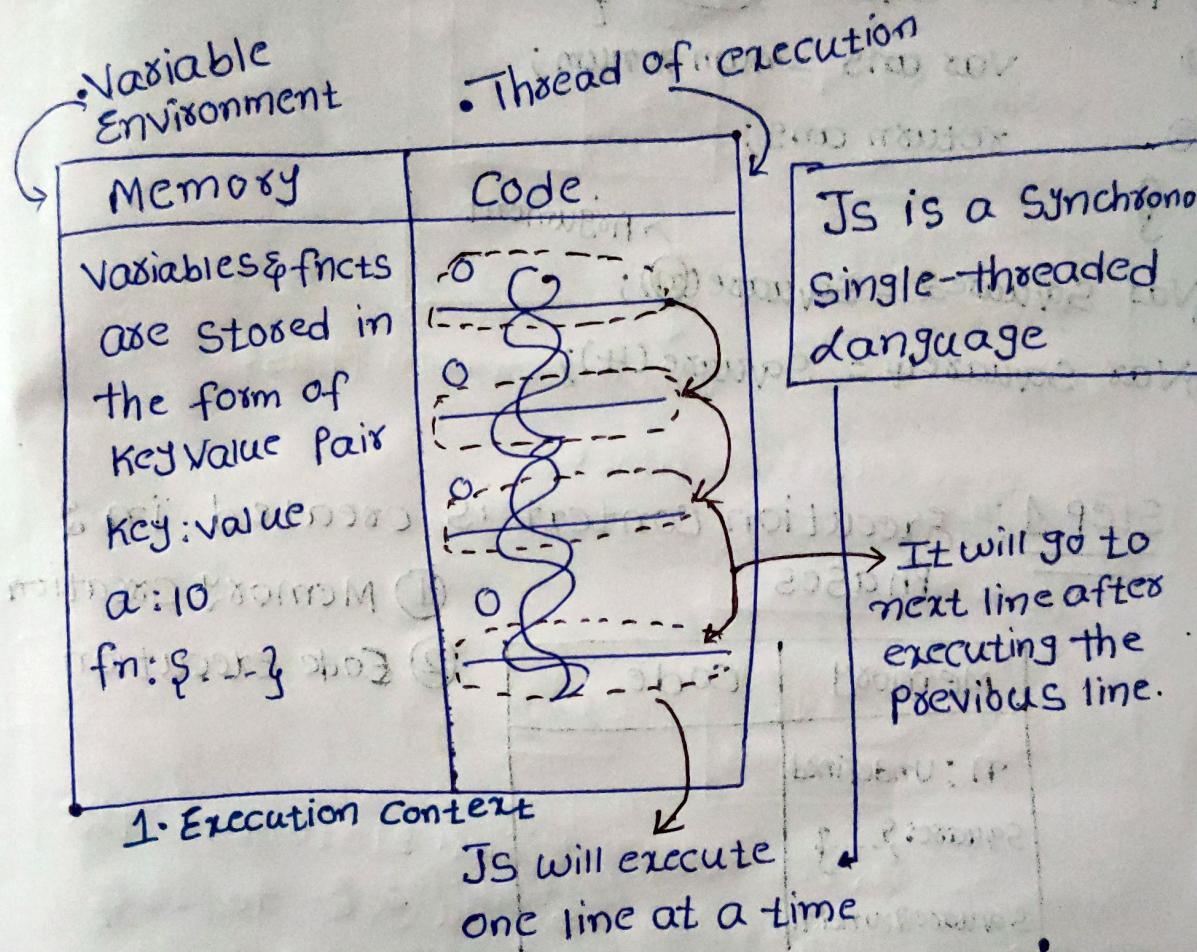


# JavaScript Series

Everything in JavaScript happens inside an "Execution Context". & has 2 components.



JavaScript will execute one line at a time & that too it will go to next line after executing the previous line.

So JS is a Synchronous Single-threaded language

• What happens when you run Javascript Code?

### Example Program

```
① var n=2;           → parameter  
② function Square (num) {  
③     var ans = num * num;  
④     return ans;  
}
```

```
→ Argument  
⑤ var Square2 = Square(2);  
⑥ var Square4 = Square(4);
```

Step 1 - Execution context is created in 2 Phases

- ① Memory creation
- ② Code execution

Memory	Code
n : Undefined	
Square: {}	
Square2: undefined	
Square4: undefined	

In phase 1 it allocates memory to variables & functions

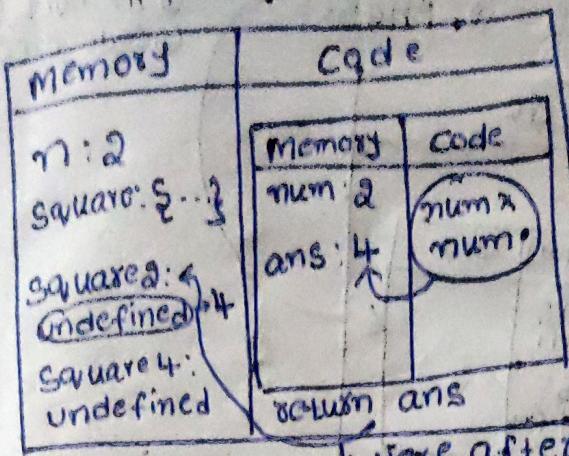
Step 2 - Phase 2 [Code execution]

Memory	Code
n: 2	
Square: {}	
Square2: undefined	
Square4: undefined	

Memory creation  
Code execution

Happened because of line no ⑤ & undergoes 2 phases again

## ② Code execution



"return" keyword tells us to return the control where function was invoked.

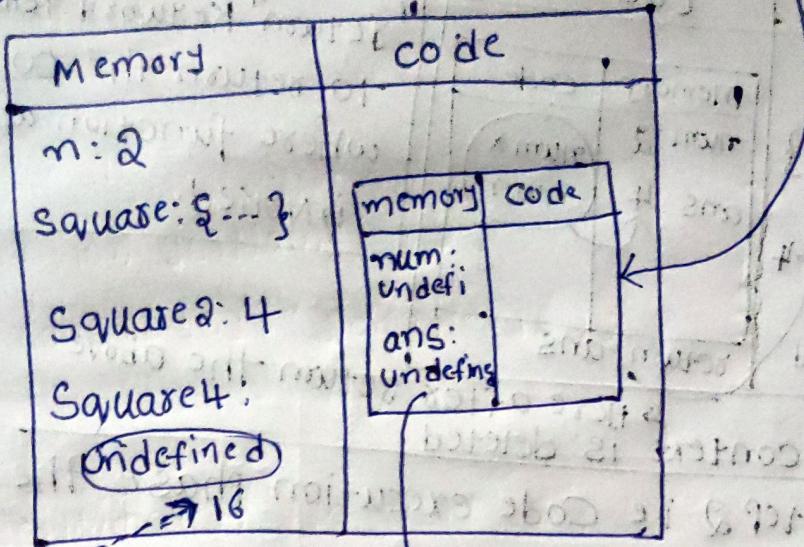
Here after return the above execution context is deleted

Here in Step 2 i.e. Code execution phase the value of n becomes 2 & in next line we have function it remains the same, then in line no 6 "function invocation occurs" Due to this again a global execution context is created as above.

After the func is executed the whole execution context is deleted which was created due to function invocation at line no ⑤.

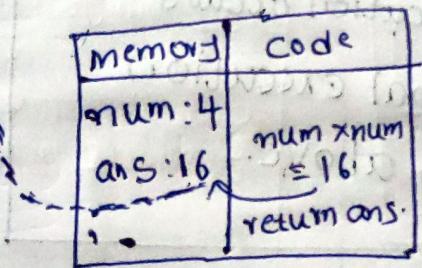
Now the control is at line no ⑦ in the same manner again new execution context is created & undergoes 2. phases.

- ① Memory
- ② Code Execution



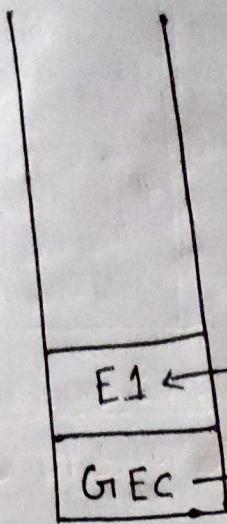
① Memory creation phase

In code execution phase:



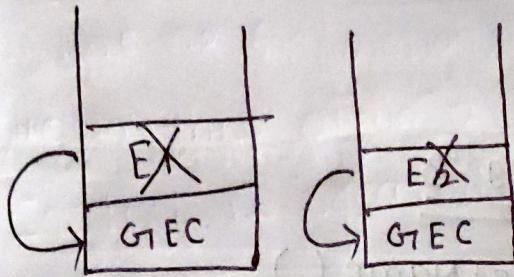
Next i.e. after return statement the value of undefined is changed to 16 & the execution context is deleted & the control is at last line & nothing to execute then the whole execution context is deleted.

For creating & deleting the execution, it maintains a Call Stack.



This happens when function invoked  
& after return it gives the control  
back where it was left & it  
pops off

So it looks like the below figure



& same happens as  
above

After all the code is executed the call stack is emptied.

Call Stack → Different names.

- ① Execution Context Stack
- ② Program Stack
- ③ Control Stack
- ④ Runtime Stack
- ⑤ Machine Stack



## Example programs [Basic Concepts]

### Example 1:-

```
Var x=7;  
function getName() {  
    console.log("Namaste JS");  
}  
getName();  
Console.log(x);
```

O/P:- Namaste JS

= 7

### Eg:-2

```
getName();  
console.log(x);  
  
var x=7;  
function getName() {  
    console.log("Namaste JS");  
}
```

O/P:- Namaste JS  
undefined

### Eg:-3

```
getName();  
Console.log(x);  
function getName() {  
    console.log("Namaste JS")  
}
```

O/P:- Namaste JS  
x is not defined

So based on those examples let me define "HOISTING"

"Hoisting" is a phenomenon in JS by which we can access the variables & functions before we initialize it

### Some more examples

① var x=7;

function getName() {

    Console.log("Namaste JS");

}

Console.log(getName);

o/p:- f getName()

    Console.log("Namaste JS");

}

② Console.log(getName)

var x=7

function getName() {

    Console.log("Namaste JS");

}

o/p:- f getName()

    Console.log("Namaste JS");

}

ex:-

```
getName();  
Console.log(x);
```

```
Console.log(getName);
```

```
Var x=7;
```

```
Var getName = () => {
```

```
    Console.log("Namaste JS");
```

```
}
```

O/P:- getName is not a function.

## How functions work in JS & variable Environment

```
Var x=1;
```

O/P:- 10

```
a();
```

100

```
b();
```

1

```
Console.log(x);
```

```
function a() {
```

```
    Var x=10;
```

```
    Console.log(x);
```

```
}
```

```
function b() {
```

```
    Var x=100;
```

```
    Console.log(x);
```

```
}
```

? Do you know the Shortest JS Program?

→ Empty file ✓

When we run still it creates GEC & also  
Create window <sup>we can</sup> & access all vari & methods.

Window is a global object created along  
with GEC & (this) variable

At global level

this == window

true

When we write code & everything we see at  
top level i.e. not inside a function is a  
global space.

So these variables in global space gets attached  
to global object (i.e. window)

Eg:- var a = 10;

function b() {

var x = 10;

console.log(this.a)

}  
console.log(window.a)

So only a is in global scope & OLP is

10

10

10

## Undefined Vs not defined

console.log(a)

var a=7;

console.log(x);

O/P:- undefined

x is Not defined

console.log(a);

var a=7;

console.log(a);

O/P:- undefined

7

Javascript is <sup>weakly typed</sup> loosely typed language (or)  
because it is flexible with different  
data types.

var a;

console.log(a);

a=10;

console.log(a);

a="hello world";

console.log(a);

O/P:- undefined

10

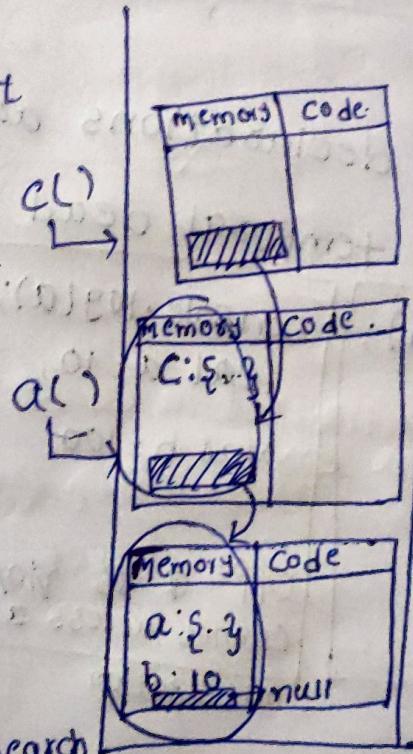
helloworld

# The scope Chain; Scope & lexical Environment

① function a() {  
 console.log(b);  
 }  
 var b = 10;  
 a();  
 Output: 10

② function a() {  
 c();  
 function c() {  
 console.log(b);  
 }  
 var b = 10;  
 a();  
 }  
 Explanation for 2nd code using Call Stack

lexical environment  
of c is a.  
lexical environment  
of a is Global.



Here it will Search  
in a's lexical environment So it will ~~not~~ find & go to  
reference it will find there & prints 10

Here C will  
have access to  
local & its  
lexical environment

Scope chain is like it was unable to find in its local it goes to other scope & continues.

### lexical environment

local memory + reference to lexical env of parent

Whole chain of lexical environment is called "Scope chain"

let & const declarations are hoisted & they are in temporal dead zone.

e.g:-  
console.log(b);  
let a = 10;  
var b = 100;

O/P:- undefined

console.log(a);  
let a = 10;  
var b = 100;

O/P:-  
ReferenceError: a is not defined  
cannot access a before initialization

## What is Hoisting in let & const?

when we declare

Here memory is allocated to both a & b  
& they are in scope.

b is in global scope & a is in script  
& memory allocated to both

var b is attached to global object

let & const has different scope (& cannot  
access before initialization)

Here a has allocated memory & if we do as

console.log(a);

let a = 10;

this stage is "temporal dead zone" if before initializing

⇒ let a = 10;

let a = 100;

Syntax error:- Identifier a has already been declared.

Its possible in var but not in let ✓

⇒ let a = 10;

const b = 1000;

Same as let but more strict than let

es5:

let a;

const b = 1000;

a = 10;

console.log(a)

O/P:- 10

let a;

const b;

b = 1000;

a = 10;

console.log(a);

O/P:- Missing initializer in const declaration  
↳ Syntax error

let a;

const b = 1000;

b = 100;

a = 10;

console.log(a);

O/P:- Assignment to constant variable  
↳ Type error

```
let a=100;  
let a=100;
```

```
const b=100;
```

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

Op:- Identifier 'a' has already been declared.  
↳ Syntax error

```
console.log(a);
```

```
let a=100;
```

```
const b=100;
```

```
a=10;
```

```
console.log(a);
```

} Cannot access a before  
initialization.

↳ Reference error

⇒ let & const are block scoped.

{

// compound statement

}

{

var a=10;  
Console.log(a); } → Block.

}

we group multistmts in a block so that  
we can use it where JS expects one  
Statement.

if (true) {

var a=10;

Console.log(a);

}

whatever the variables & functions can  
be accessed in the block is  
BLOCK SCOPE.

S  
Var a=10;

let b=20;

const c=30;

{  
let & const are in  
separate space, they are hoisted in  
block scope

We cannot access let & const variables  
outside the block scope. & a is in global  
scope we can access it outside.

Eg:-

{  
Var a=10;

console.log(a);

Var b=20;

console.log(b);

const c=30;

console.log(c);

console.log(a);

console.log(b);

console.log(c);

{  
}

Output:- 10

20

30

10

b is not defined.

Q:- var a = 100;

{

  var a = 10;

  let b = 20;

  const c = 30;

  console.log(a);

    .. (b);

    .. (c);

}

  console.log(a);

O/P:- 10

=====

20

30

10

Here  $a$  is in global scope & value is 100 when we execute  $\text{var } a = 10$ , its value is set to 10 because it is in global scope.

So O/P will be 10;

$\Rightarrow \text{let } b = 100; \rightarrow \text{Script scope}$

{  $\nearrow$  Global scope  
  var a = 10;

$\leftarrow$  Block scope  
  { let b = 20;  
    const c = 30;

    console.log(a);

      .. (b);

      .. (c);

O/P:-

=====

10

20

30

100

}

  console.log(b);

[Same with const] ✓

## Closures in Javascript

Closure :- Function along with its lexical scope bundled together

Example :-

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

O/P :-

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

## Uses of Closures :-

- Module Design Pattern.
- Currying
- Functions like once
- Memoize
- maintaining state in asyc world
- SetTimeouts
- Iterators
- & many more

## SetTimeout + closures

```
function x() {
```

```
    var i = 1;
```

```
    setTimeout(function() {
```

```
        console.log(i); // 1
```

```
        i = 3000;
```

```
        console.log("Namaste JS");
```

```
}
```

```
x();
```

Output:- Namaste JS

1

## Interesting example

```
function x() {
```

```
    for(var i = 1; i <= 5; i++) {
```

```
        setTimeout(function() {
```

```
            console.log(i);
```

```
            i *= 1000;
```

```
}
```

```
        console.log("Namaste Javascript");
```

```
}
```

```
x();
```

Output:- Namaste Javascript

6

method object custom

Object

6

function

function

function

function

```
function x() {
```

```
    for(let i=1; i<=5; i++) {
```

```
        setTimeout(function () {
```

```
            console.log(i);
```

```
        }, i * 1000);
```

```
}
```

```
    console.log("Namaste JS");
```

```
}
```

```
x().
```

O/P:-

Namaste JS

1

2

3

4

5

```
function x() {
```

```
    for(var i=1; i<=5; i++) {
```

```
        function close() {
```

```
            setTimeout(function () {
```

```
                console.log(x);
```

```
            }, 1000 * i);
```

```
        } close();
```

```
    } close(i);
```

```
    console.log("Namaste JS");
```

O/P:- Namaste JS

(0) 1 0ms

2

3

4

5

(1) 1000ms 2000ms 3000ms

(2) 2000ms 3000ms 4000ms

(3) 3000ms 4000ms 5000ms

## Advantages of closures

```
function counter() {  
    var count = 0;  
    return function incrementCounter() {  
        count++;  
        console.log(count);  
    };  
}
```

function Counter () {

    var Count = 0;

    return function incrementCounter () {

        Count++

        console.log (Count);

}

3

Var Counter1 = Counter()

Counter1();

Counter1();

O/P:- 1

2

+

O/P:- 1

2

3

+

4

+

[Same code with changes]

Var Counter2 = Counter();

Counter2();

Counter2(counter());

Counter2(counter());

Function Statement, Function expression,  
Function Declaration, Anonymous Function,  
Named Function expression, Difference b/w  
Parameters & Arguments, First class functions,  
Arrow Functions.

## // Function Statement

- function a() {

```
    console.log("a is called")
```

```
}
```

```
a()
```

## // Function expression

- var b = function() {

```
    console.log("b is called")
```

```
}
```

```
b()
```

O/P:- a is called  
b is called.

```
a();
```

```
b();
```

```
function a() {
```

```
    console.log("a called")
```

```
}
```

```
var b = function() {
```

```
    console.log("b called")
```

```
}
```

O/P:- a called

b is not a function

Function Declaration is same as  
Function Statement.

## Anonymous Function

Anonymous Functions are used in a place where functions are used as values.

Var b = function () { }

Console.log("b called");

{ } → f() is not defined

## Named function expression

Var b = function xyz() { }

Console.log("b called");

{ } → xyz is not defined

xyz → f

xyz() is not defined

O/P:- xyz is not defined.

⇒ Var b = function xyz()

Console.log(xyz);

{ };

b();

O/P:- f xyz

console.log(xyz);

## Difference b/w arguments & parameters

Var b: function (Param1, Param2) {  
    <sup>parameters</sup>  
    // code here  
};

```
console.log('bcalled');
```

94

b(1, 2)  
    ^  
    arguments

## First class functions

```
var b = function(param1, param2) {
```

```
console.log("Hello " + param1);
```

۲

b(function l) §

3)

Op: f ( ) s

۳

Var b = function (Param1) {

```
Console.log(param1);
```

3

function xyz( ) {

2

b(2y3)

$$\text{OP} \equiv f(\cdot) \in \mathcal{S}^{\text{admissible}}$$

3

```
var b = function x( ) {
```

```
    return function xyz( ) {
```

3

```
console.log(b());
```

## Callback function:

~~function ex~~

3

~~ex function ex~~

3

```
setTimeOut(function () {
```

```
    console.log("timer");
```

```
, 5000);
```

OP:- x

y

function x(y) {

timer

```
    console.log("x")
```

```
y();
```

3

```
x(function y() {
```

```
    console.log("y");
```

3

## ⇒ Event listeners-

(index.html)

```
<html>  
<head>  
<meta ..... >  
.....
```

```
<title> Javascript </title>
```

```
</head>
```

```
<body>
```

```
<h1 id="heading"> Namaste Javascript </h1>
```

```
<button id="clickme"> Click me </button>
```

```
<script src="index.js"></script>
```

```
</body>
```

```
</html>
```

(index.js)

↳ id given above

```
document.getElementById("clickme")
```

```
.addEventListener("click", function(e) {
```

```
    console.log("button clicked")
```

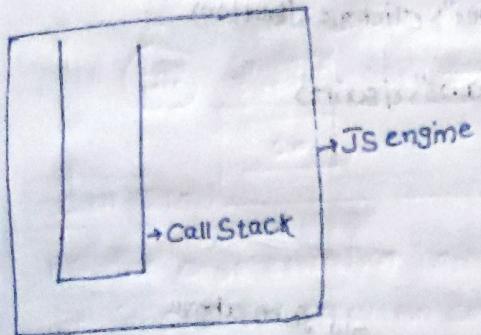
```
});
```

↳ Call back function

## Event loop



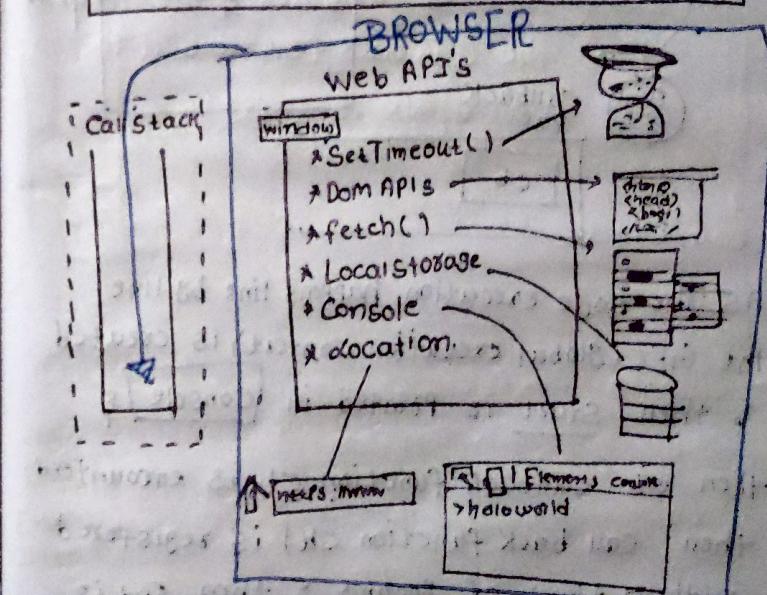
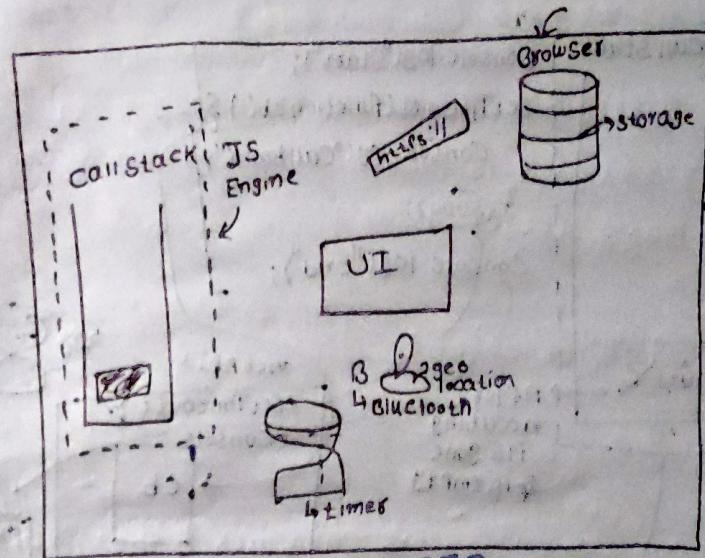
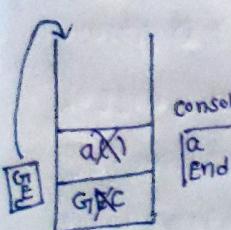
JS is a synchronous single-threaded language  
It has one Call Stack & perform one thing at a time



Whenever a JS code is executed a global execution context & pushed into call stack

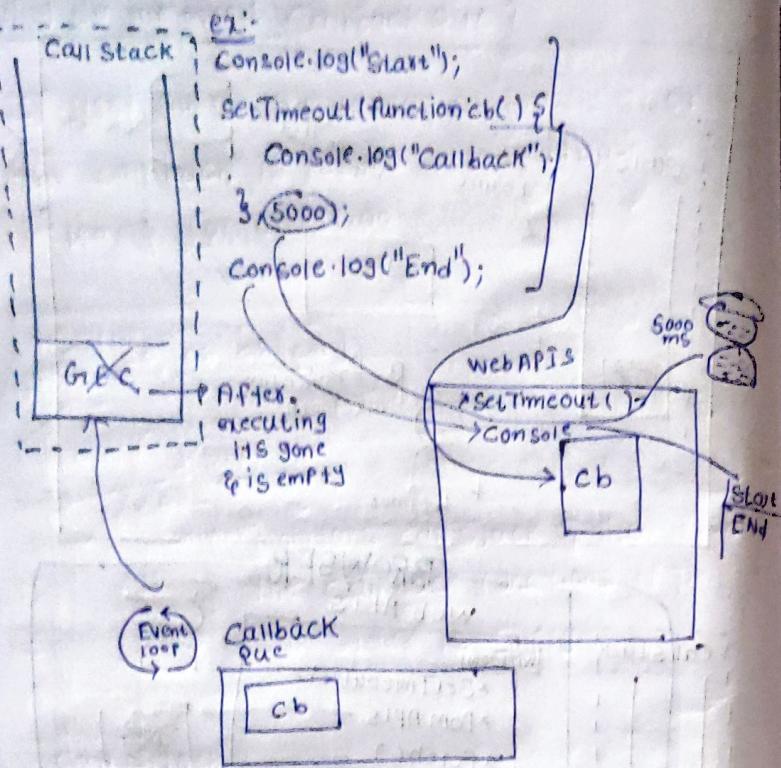
Example :-

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



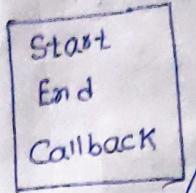
Browser gives access to all these superpowers to JS Engine.

We get access in call stack by the help of global object i.e window.

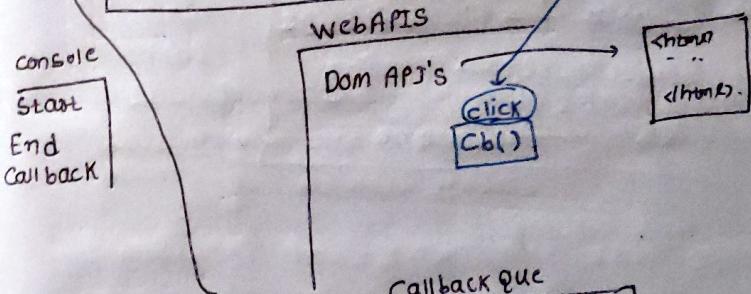
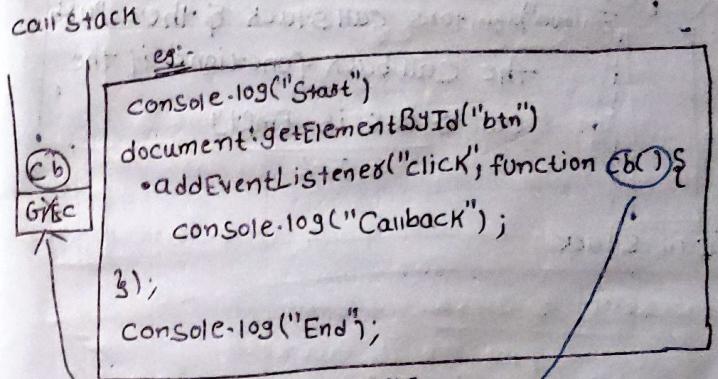


As the code execution happens line by line  
the GIEC (Global execution context) is created  
& then Start is printed in Console &  
then as callback function cb is encountered.  
then callback function cb is registered  
with a timer of 500ms & then End is  
logged in Console because we know that  
JS won't wait for anyone & then once  
timer expires cb is pushed into  
Callback queue & as we know there is  
nothing in Call Stack the event loop

pushes cb into call stack & it gets executed  
& the output results in Callback in the console  
so the output is :-



### More Examples



Callback queue

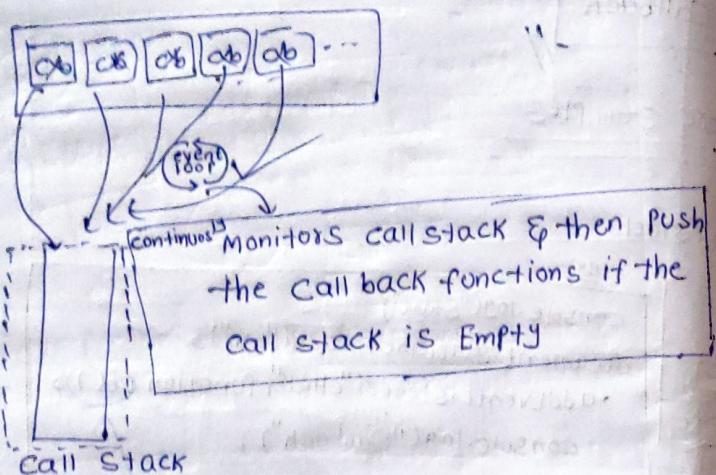
cb

Monitors Call Stack. If empty then pushes callback into stack.

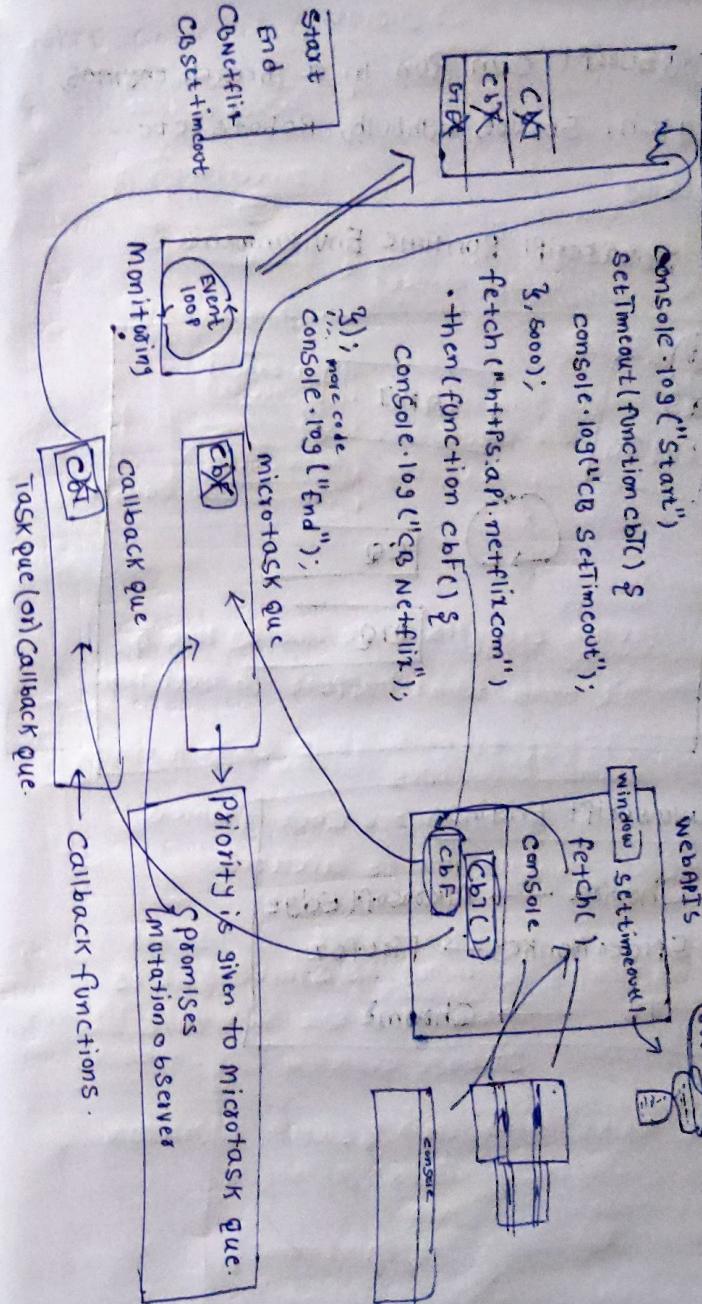
When user clicks the callback (cb) will be pushed into callback queue & waits for execution.

## Why do we need Callback Queue?

If user clicks the button continuously then the callback is pushed into callback queue & waits for execution

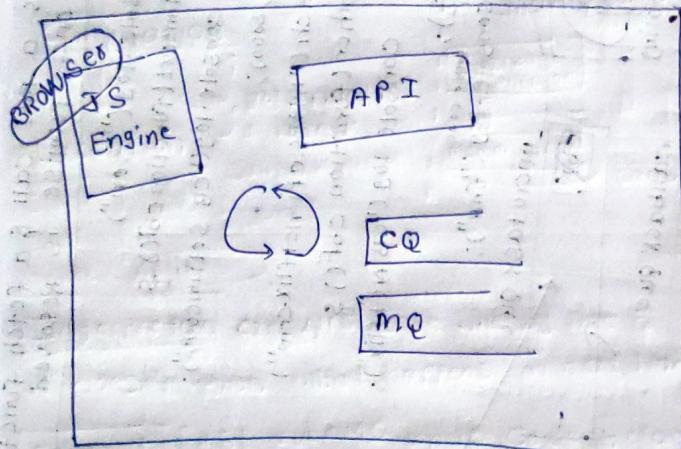


Fetch requests a API call & a fetch function returns a promise & we should pass a callback func once promise is resolved



JavaScript Can Run in a Search engines,  
Watch, Server, lightbulb, Robots etc...

### Javascript Runtime Environment



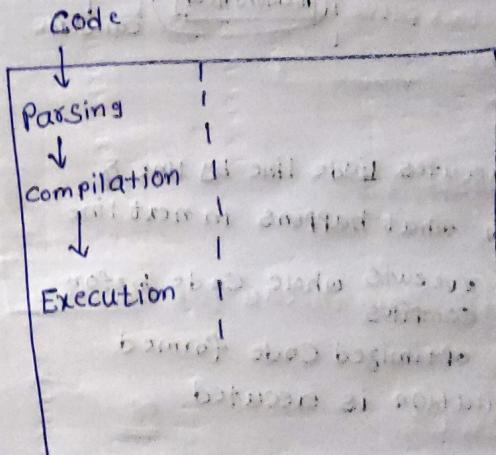
### Javascript Engines

- ⇒ Chakra → Microsoft Edge
- SpiderMonkey → Firefox
- V8 → Chrome

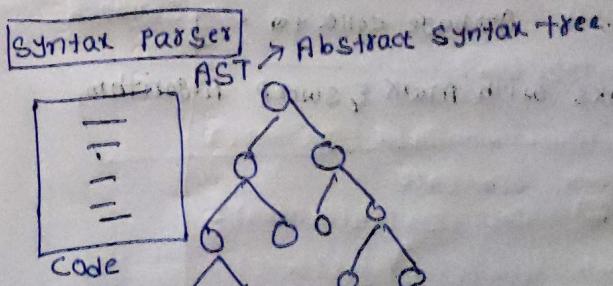
### First Javascript engine?

⇒ Brendon EIK & it has evolved a lot & being used as SpiderMonkey  
↳ First discovered in Firefox

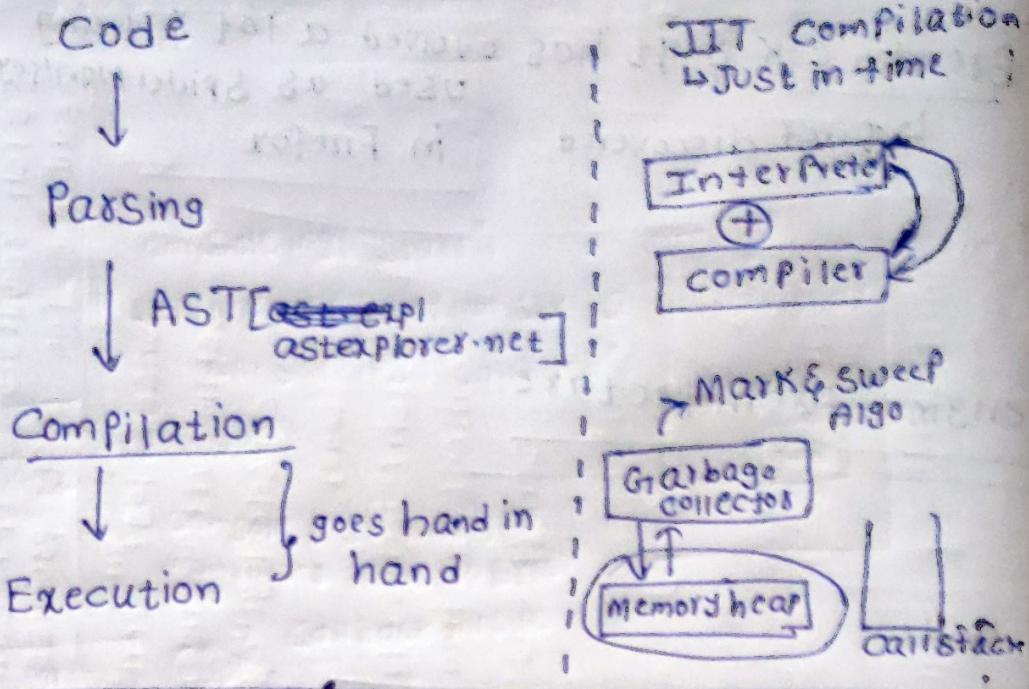
### Js engine Architecture



During Parsing the code is broken into tokens  
eg. `let a = 7`



# JS Engine



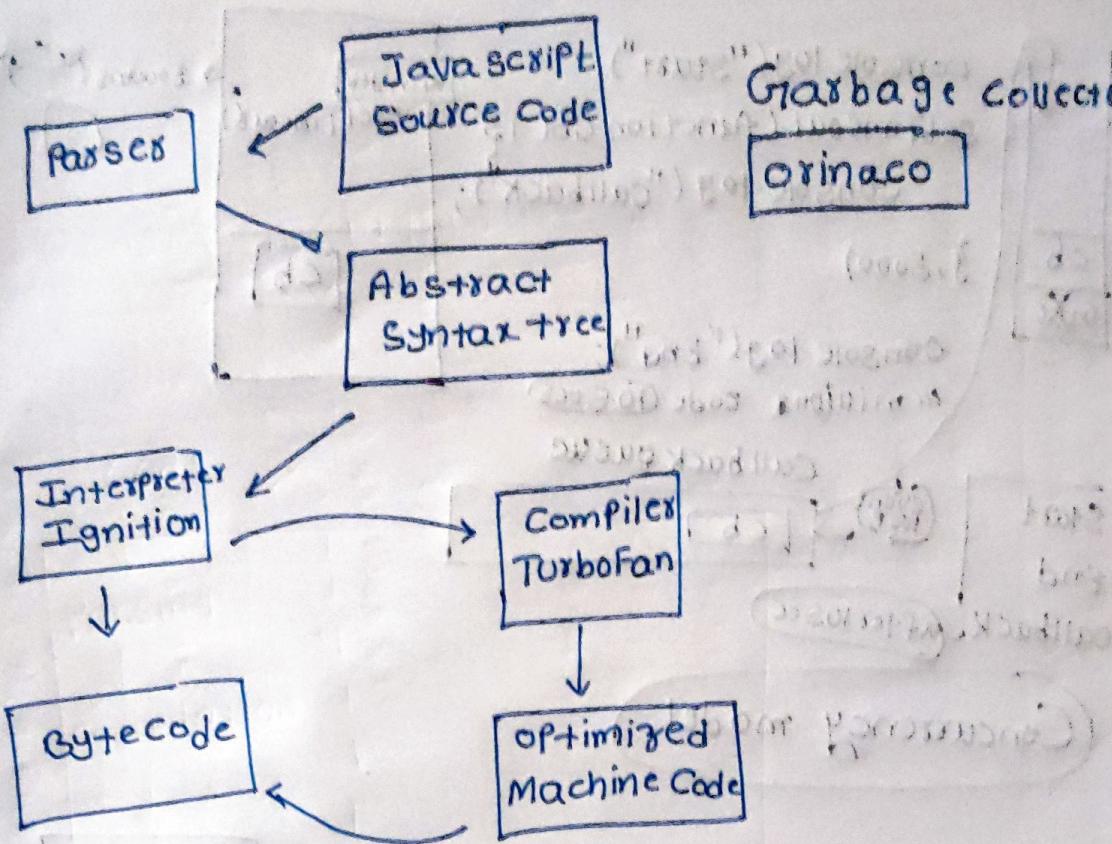
**Interpreter** executes Code line by line & it doesn't know what happens in next line.

**Compiler** first ~~executes~~ whole Code before compiles execution & optimized Code formed after compilation is executed

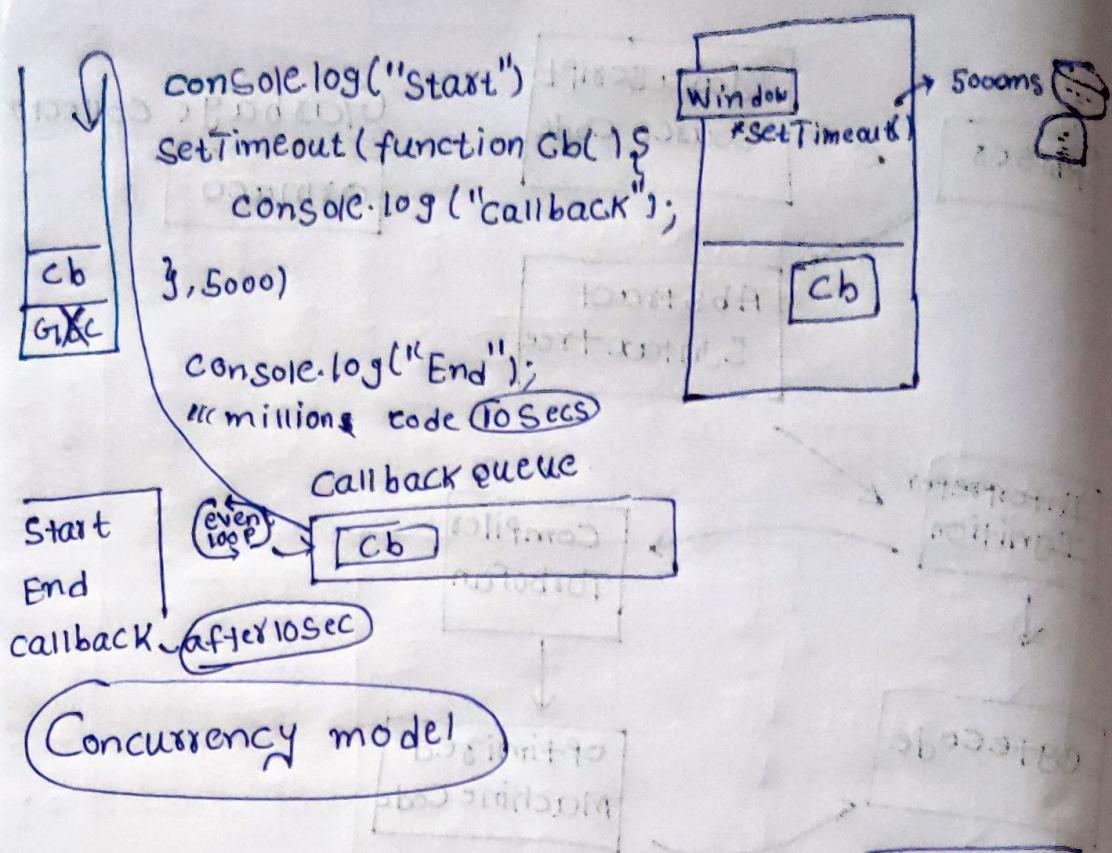
**Memory heap** [All memory is stored & it is constantly in sync with Call Stack]  
Garbage collector

GC works with Mark & sweep Algorithm

# V8 JS Engine



# Trust issues with setTimeout()



lets block the Main thread for 10'seconds

⇒ Eg:-

```
console.log("Start");
setTimeout(function cb() {
  console.log("callback");
}, 5000);
console.log("End")
```

```
let startDate = new Date().getTime();
let endDate = startDate;
while (endDate < startDate + 10000) {
  endDate = new Date().getTime();
```

3 console.log("while expires")

OP:-

Start  
 End  
 while expires  
 Callback.