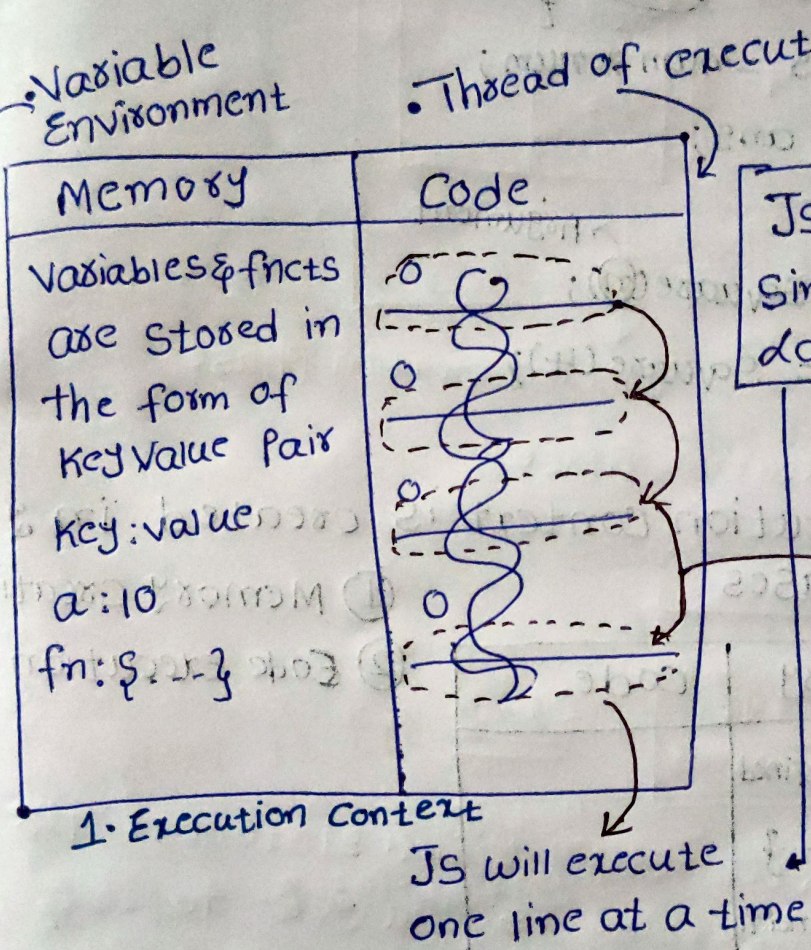
Javascript Series

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

Parameter

Argument

Step 1:- Execution Context is created in 2 Phases

① Memory Creation
② Code execution.

Memory	Code
n : Undefined	
square: {...}	
square2: undefi	
square4: Undef ined	

In phase 1 it allocates memory to variables & functions

Step 2:- Phase 2 [Code execution]

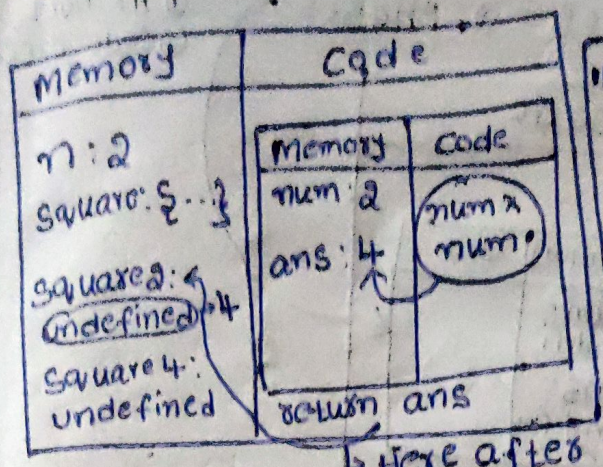
Memory	Code
n: 2	
square: {...}	
square2: undefined	
square4: undefined	

memory	code
num: undefined	
ans: undefi	

① Memory creation
~~② Code execution~~

Happende because of line no ⑤ & undergoes 2 phases again

② Code execution



"return" Keyword tell's 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 fnc 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

Memory	Code
n: 2 square: {...} square2: 4 square4: undefined → 16	memory: num: undefi, ans: undefing code:

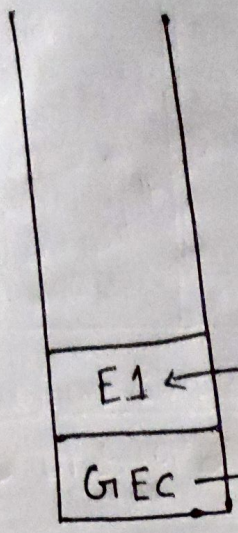
① memory creation phase

In code execution phase:

memory	code
num: 4 ans: 16	num × num = 16 return ans.

Next i.e after return ~~undefi~~ 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.

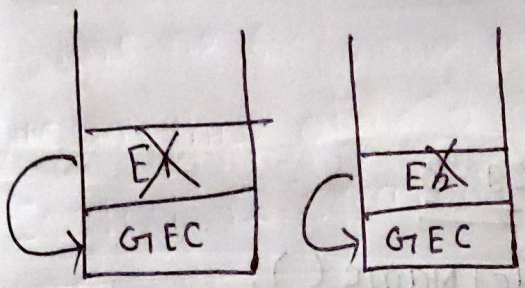
For creating & deleting the execution it maintains a Call Stack.



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

global execution context

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" in JS is a phenomenon 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;
a();
b();
Console.log(x)
function a() {
    Var x=10;
    Console.log(x);
}
function b() {
    Var x=100;
    Console.log (x);
}
```

O/P:- 10
100
1

? Do you Know the Shortest JS Program?

⇒ Empty file ✓

When we run still it creates GEC & also Create window & we can 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 fnction is a global space.

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

Eg:-
```
var a = 10;
function b() {
  var x = 10;
}
console.log(window.a)
console.log(this.a)
```

So only a is in global scope & o/p 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 Loosely typed language (or) weekly typed 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
hello world

The Scope Chain, Scope & Lexical Environmen

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

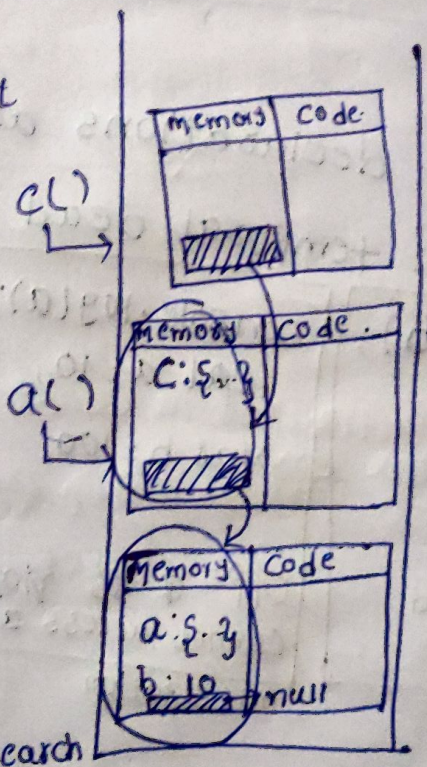
O/P :- 10

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 c will have access to local & its lexical environme

Here it will search in a's lexical environment so it will find & go to reference it will find there & prints 10

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.

eg:-

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

O/P:- undefined

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

O/P: a is not defined
cannot access a before initilization

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" ie 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

eg:-

```
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=190;
let a= 100;
Const b=100;
a =10;
Console.log(a);
```

o/p:- 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

⇒ <u>let & const are block scoped.</u>

```
{
//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.

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

O/P:- 10
20
30
10
b is not defined.

eg:- var a = 100;

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

Here a is in global scope & value is 100 when we execute var a = 10 its value is set to 10 because it is in global scope.

So o/p will be 10;

O/P:- 10
20
30
10

⇒ let b = 100; → Script scope

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

Global scope

Block scope

O/P:- 10
20
30
100

[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();
console.log(z);
// ....
z()
```

O/P :-

```
f y() {
  console.log(a);
}
7
```

Uses of Closures :-

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

SetTimeout + closures

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

O/P:- 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();
```

O/P:- Namaste Javascript
6
6
6
6
6

```
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(i) {
      setTimeout(function () {
        console.log(x);
      }, x*1000);
    }
    close(i);
  }
  console.log("Namaste JS");
}
x().
```

O/P:- Namaste JS
1
2
3
4
5

Advantages of closures

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

```
function counter() {
    var count =0;
    return function incrementCounter() {
        count++
        console.log (count);
    }
}
var counter1 = counter()
counter1();
counter1();
```

O/P:- 1
2

[Same code with changes]

```
var counter2 = counter();
counter2();
counter2() counter(); counter2();
```

O/P:- 1
2
1
2
3
4

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 b(){
  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");
}
```

Named function expression

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

```
xyz()
```

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

Parameters

```
Var b = function (Param1, Param2) {
    console.log('b called');
}
b(1, 2)
```

arguments

First class functions

```
Var b = function(Param1, ~~Param2~~) {
    console.log( ~~"b called"~~ Param1);
}
b(function () {
});
```

O/P:-

```
f () {
}
```

```
Var b = function (Param1) {
    Console.log(Param1);
}
function xyz ( ) {
}
b(xyz)
```

O/P:-

```
f () {
}.
```

```
var b = function ~~xy~~() {
    return function xyz() {
}
console.log(b());
```

Callback function :-

~~function x(){~~

~~}~~

~~function y(){~~

~~}~~

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

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

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

o/p :- x
y
timer

⇒ Event Listners:-

(index.html)

```
<html>
<head>
<meta ..... >
- - - -
- - - -
<title> Javascript </title>
</head>
<body>
<h1 id="heading"> Namaste Javascript </h1>
<button id="clickme"> Clickme </button>
<script src="index.js"></script>
</body>
</html>
```

(index.js)

```
document.getElementById("clickme")
.addEventListener("Click", function() {
  console.log("Button clicked")
});
```

↗ id given above

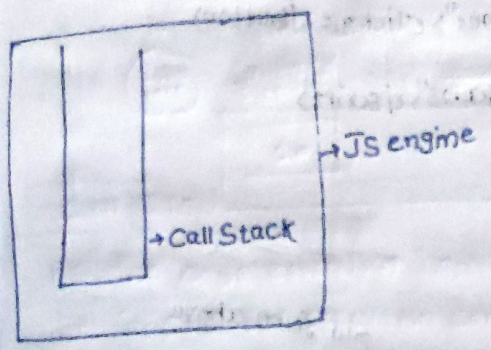
↳ event

↳ Call back function

Event loop

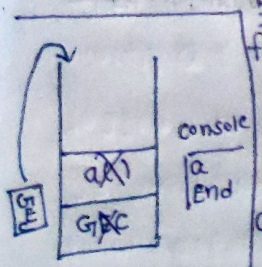
JS is a synchronous single threaded language

It has one call stack & perform one thing at a time

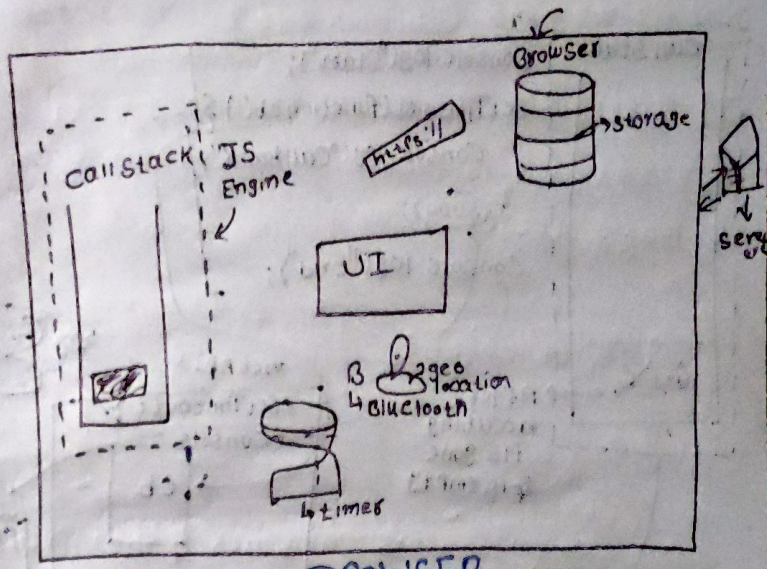


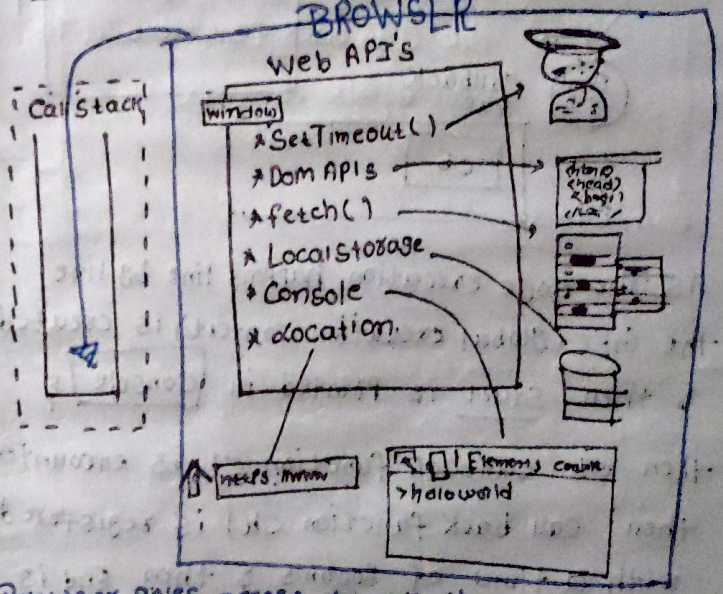
whenever a JS code is executed a global execution context & pushed into call stack

Example :-



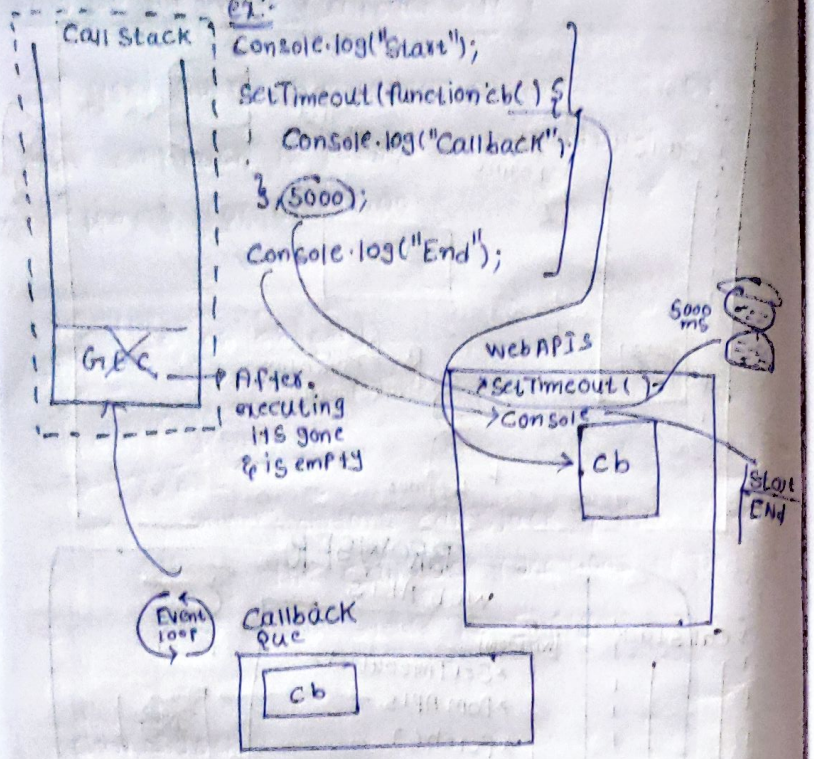

```
function a(){
  console.log("a");
}
a();
console.log("End").
```





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.

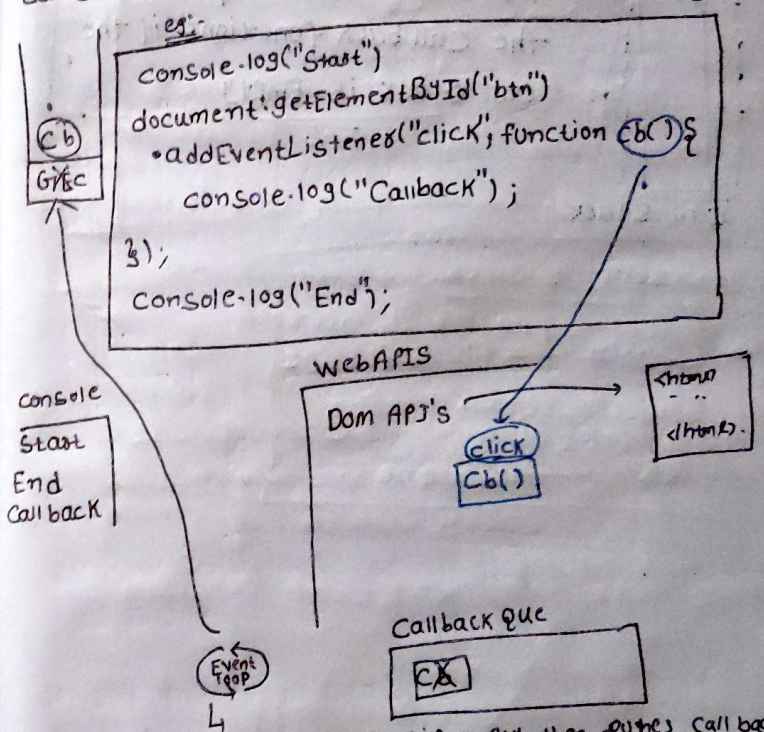


As the code execution happens line by line the GEC (Global execution context) is created & then Start is printed in Console & then as callback function cb is encountered then call back function cb() is registered with a timer of 5000ms & then End is logged in Console because we know that JS wont wait for anyone & then once timer expires cb is pushed into Callback que. & as we know there is nothing in Call Stack the event loop pushes cb into callStack & it gets executed & the o/p results in Callback in the console so the o/p is :-

Start
End
Callback

//

More Examples

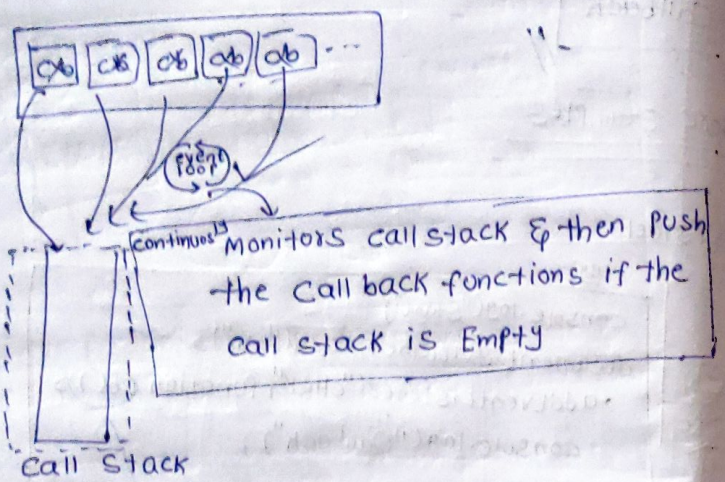


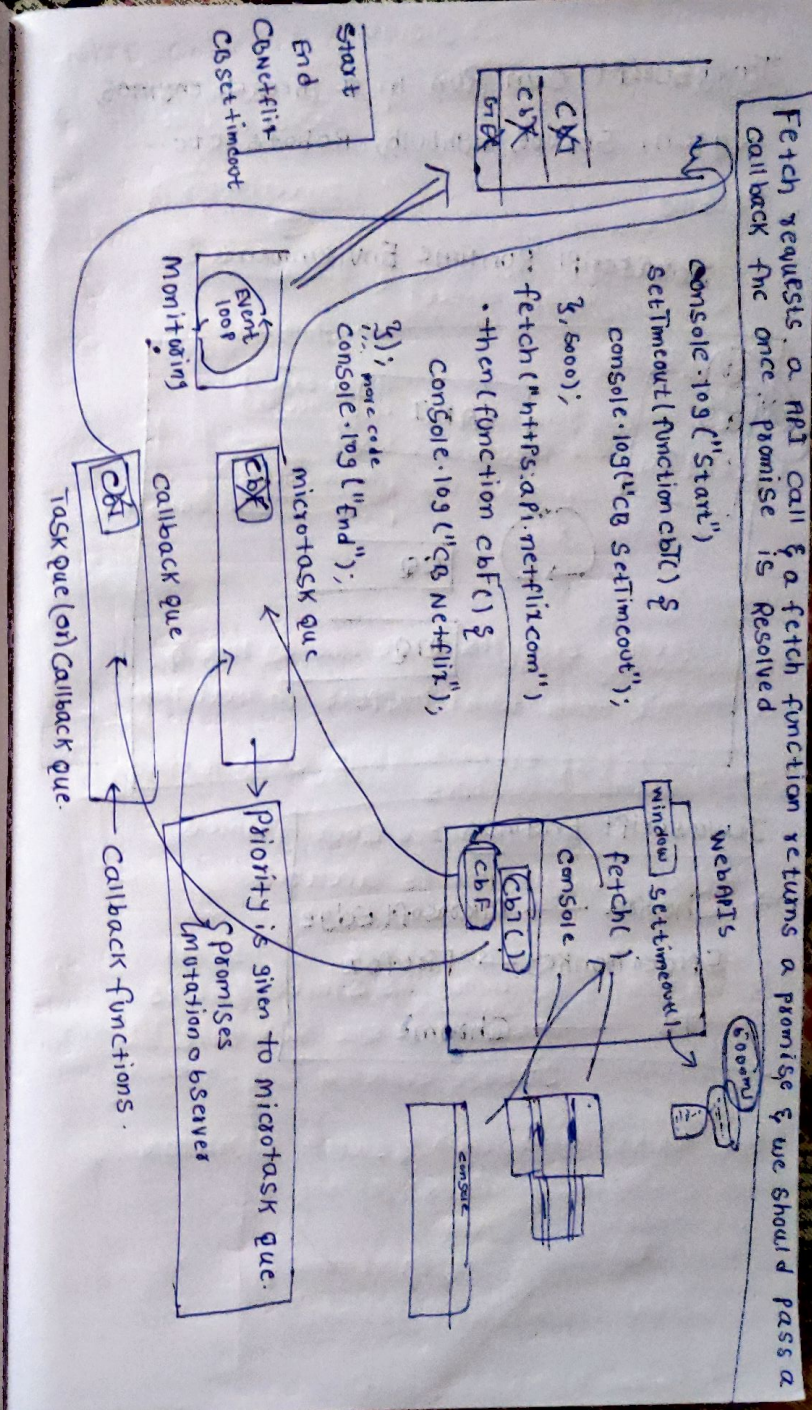
Monitors Call Stack. & if empty then pushes Call back into Stack

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

Why do we need Callback Que?

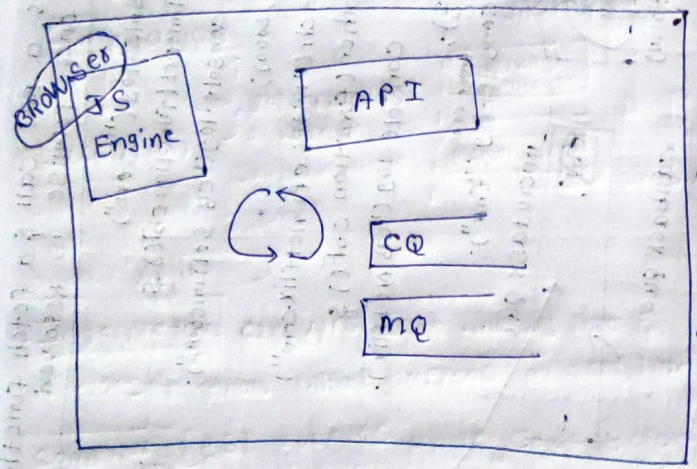
If user clicks the button continuosly then the call back is pushed into callback que & waits for execution





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

Javascript Runtime Environment



Javascript Engines :-

⇒ Chakra → Microsoft Edge

Spider Monkey → Firefox

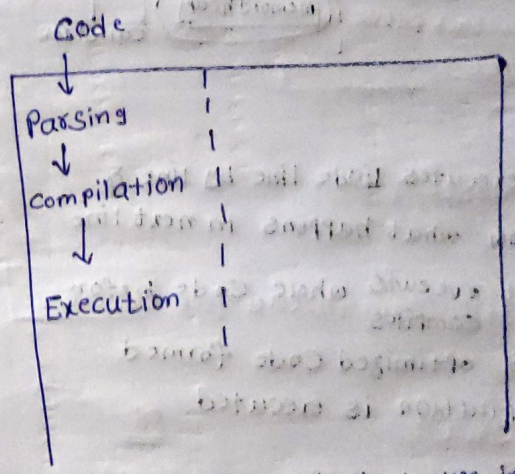
V8 → Chrome

First Javascript engine ?

⇒ Brendon eik & it has evolved a lot & being used as Spider Monkey in Firefox

↳ First discovered

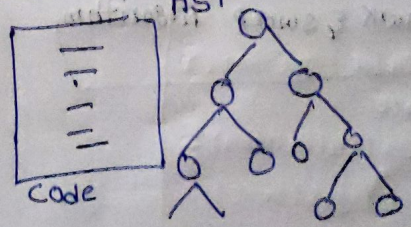
Js engine Architecture



During Parsing the code is broken into tokens

eg let a = 7

Syntax Parser → AST → Abstract Syntax tree.



JS Engine

Code
↓
Parsing
↓
AST [~~astexpl~~ astexplorer.net]
↓
Compilation
↓
Execution

} goes hand in hand

JIT compilation
↳ Just in time

Interpreter
⊕
compiler

Mark & sweep Algo

Garbage collector
↑↓
Memory heap

Call stack

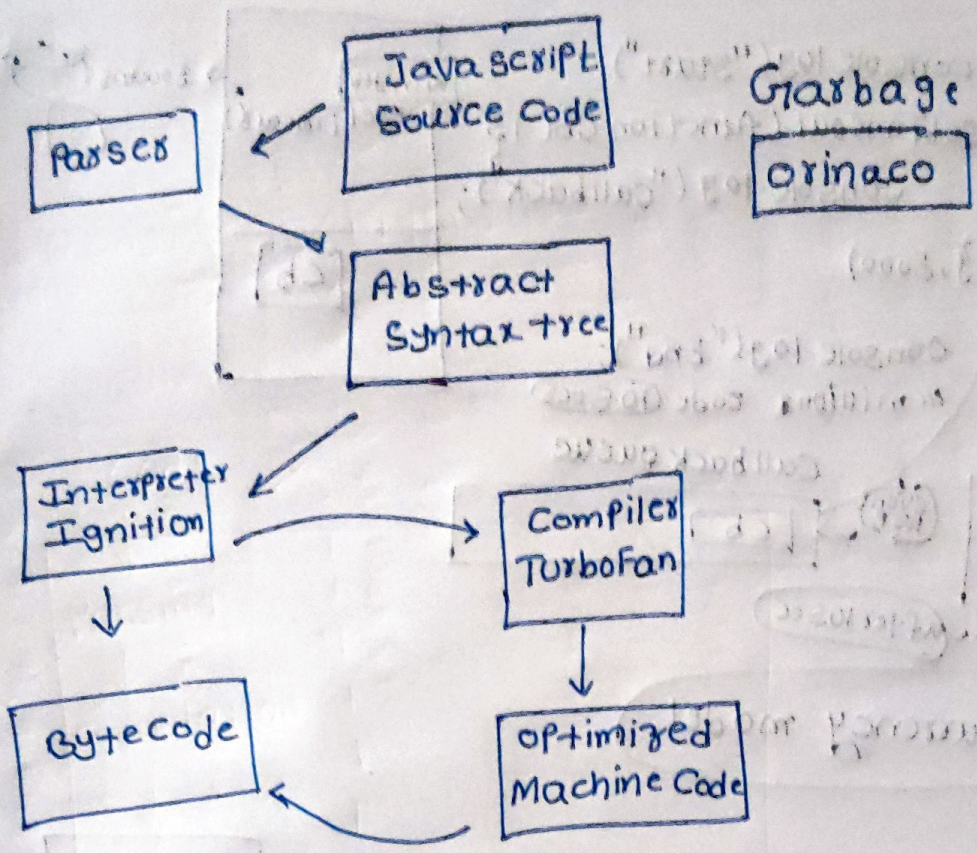
Interpretor executes code line by line & it doesn't know what happens in next line

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

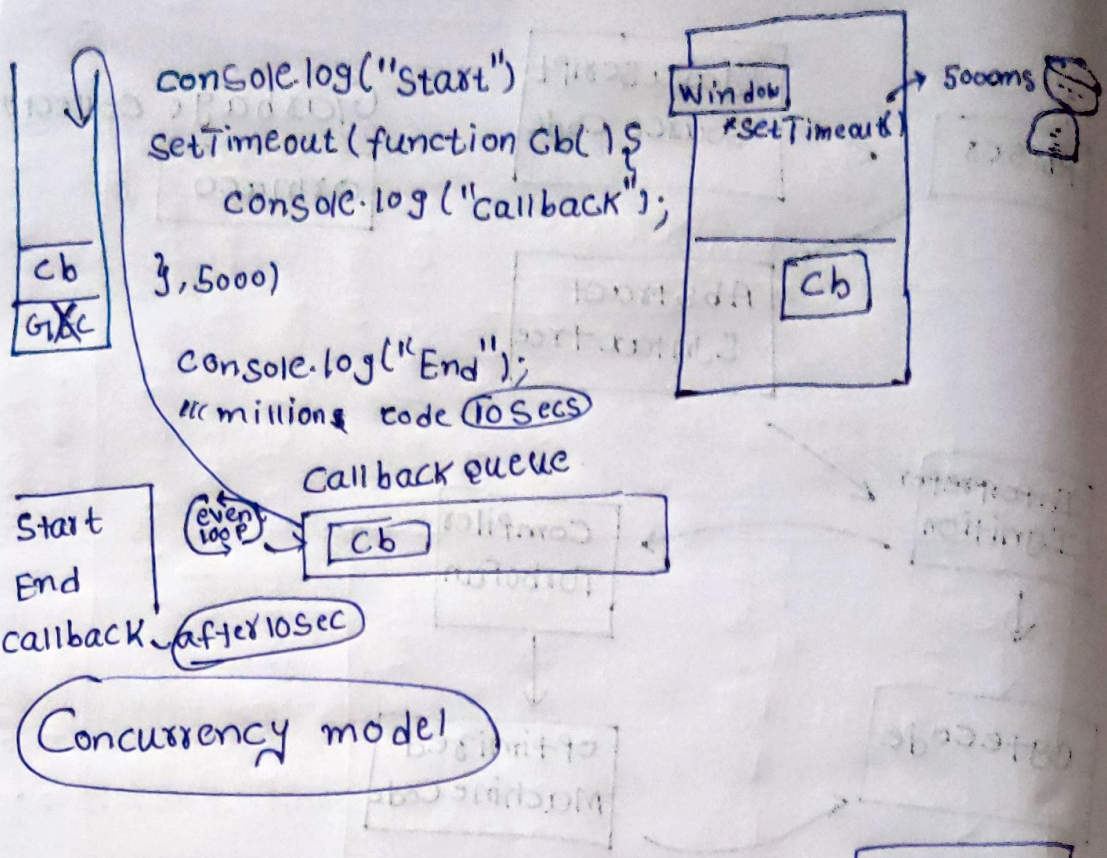
Memoryheap [All memory is stored & it is constantly in sync with call stack garbage collector ---]

GC works with mark & sweep Algorithm

V8 JS Engine


Javascript Source Code
Parser
Abstract Syntax tree
Interpreter Ignition
Compiler TurboFan
Bytecode
optimized Machine Code
Garbage collecto
orinaco

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();
}
console.log("while expires")
```

o/p:-

Start
End
while expires
Callback.