Page
Date

Namaste Javascript

By Akshay Saini Sir

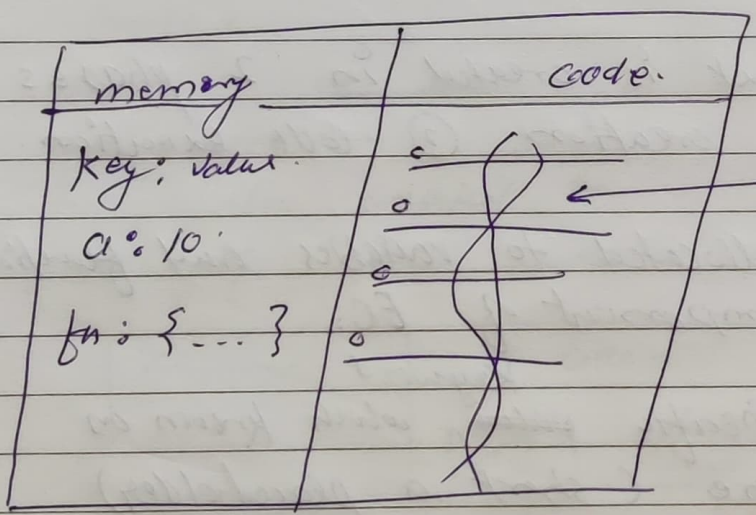
- Vermadeen.

Page
Date

★ Namaste Javascript ★

★ Everything in javascript happens inside an Execution Context,
inside a container / Big box

Execution Context
- memory Component
  - also known as variable environment
  - It is a sort of environment in which all the variables and functions are stored as the key - value pairs
- code Component
  - also known as thread of Execution
  - a line executed at a time.



★ Java script is a synchronous. single-threaded language
- synchronous
  - In a specific order
- single-threaded
  - can execute single line at a time
    - goes to next line only when current line is executed

Page
Date

what happens when you run JS. code?
An Execution Context is created.

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

→ A global execution context will be created for the code is runs

Execution Context is created in 2 phases
① memory creation ② code execution
↓
memory is allocated to variables and functions in memory component of EC:

Variables — a specific ~~value~~ Keyword which known as undefine (short a placeholder)

functions → Exact code of function is copied to memory component.

After phase 1 of EC

memory Creation	code Execution
n: undefined square: {...} square2: undefined square4: undefined	

★ Code Execution

Is runs whole code ness line by line.
Every calculation and logical operation happens in phase only.
Actual value to variables is assigned.

→ In 6th line of code square function is invoked.

Function are heart of JS. They act as mini programm. Hence a execution context is also created It will again through 2 phases of EC. ~~creation~~

→

memory	code
n=2 square = {...}	local memory → phase1 →

memory	code. Execution
~~xxx~~ undefin-ed ans: undefined	

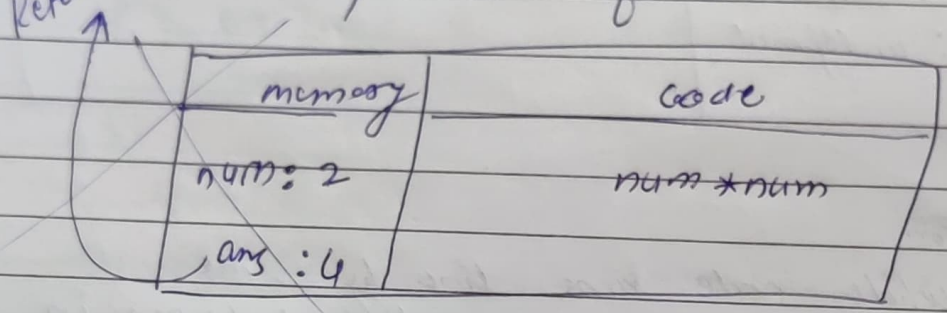
phase 2 →

Page
Date

Control

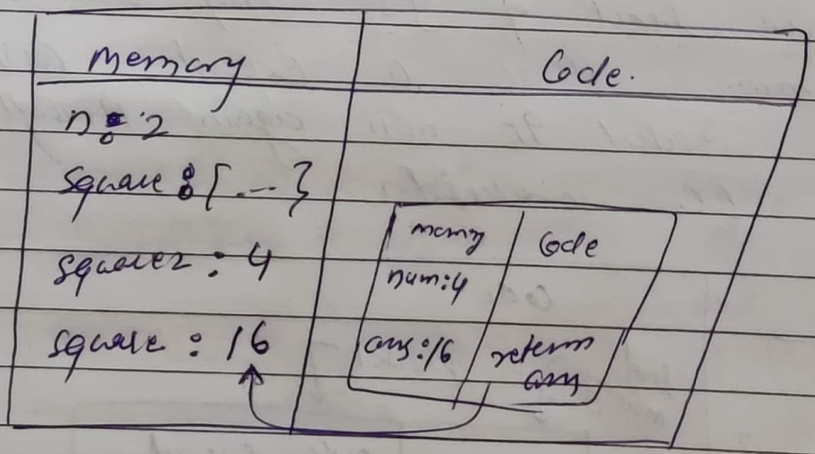
Return will return value back to Execution Context where the function was invoked.

★ After phase 2 of Execution Context of square() function



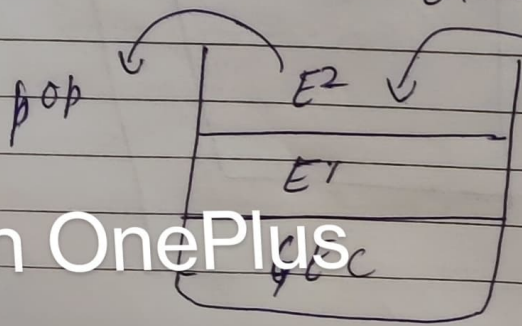
★ for line 7 — a brand new Execution Context will be created. and similar process will happen.

after phase 2 of Global EC.



★ JS has own call stack

Global Execution Context — GEC



JS call stack.

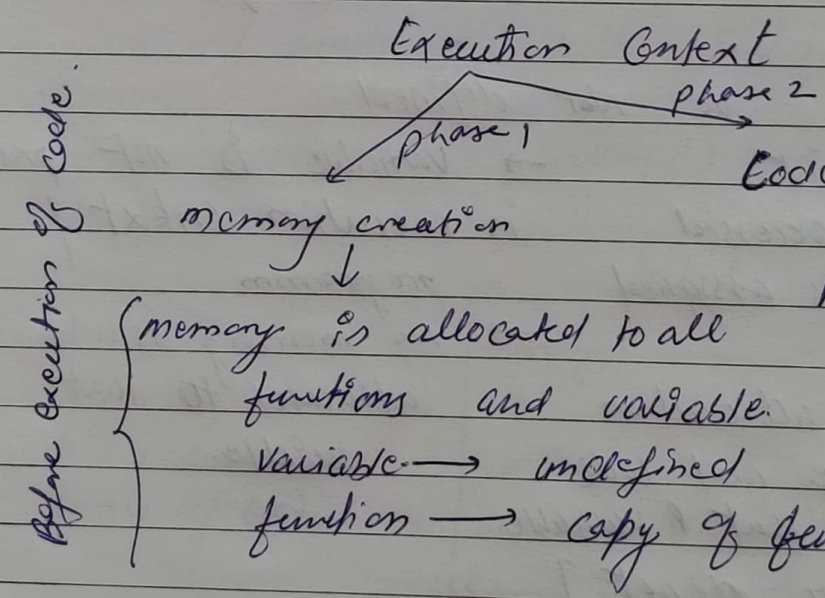
Page
Date

Call stack memory manages call execution contexts.

★ · Call stack · maintains · the order of execution of execution · contexts

Call stack is also popular as
1. Execution Context Stack
2. Program Stack
3. Control Stack
4. Runtime Stack
5. Machine Stack.

Hoisting in JS.



① var x = 7;
function getName()
{ console.log ("Namaste JS"); }
getName();
console.log (x);

Output
Namaste JS
7

Page
Date

② getName();
console.log(x);
function getName()
{ console.log("Namaste JS"); }

var x = 7;

op
Namaste JS
undefined

③ getName();
console.log(x);
~~function~~ console.log(getNamne);

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

op
① Namaste JS
② Uncaught ReferenceError : x is not defined at index.js : 2
③ ƒ getName()
{ console.log("Namaste JS");

memory is allocated but value is not assigned

undefined Vs Not defined

variable is present but is being accessed before value is assigned to it
[only memory allocation phase is done and code execution for that variable is not over yet]

→ variable is not present in execution context / program.
→ No memory is allocated to that variable.

☆ Before Actual actual code execution, memory is allocated to all variables and function in memory creation phase of Execution Context.

Page
Date

⑥

```
getName();
console.log(x);
console.log(getName);      console.log(getName2);
var x = 7;
var getName = () =>  ⟶ Arrow function syntax
{   console.log("Namaste JS"); }

function getName2()  ⟶ copy whole code in
{ console.log("Hello JS"); }   execution context.
```

op

① Namaste JS

② undefined

③ undefined

④ f getName2()
{ console.log("Hello JS"); }

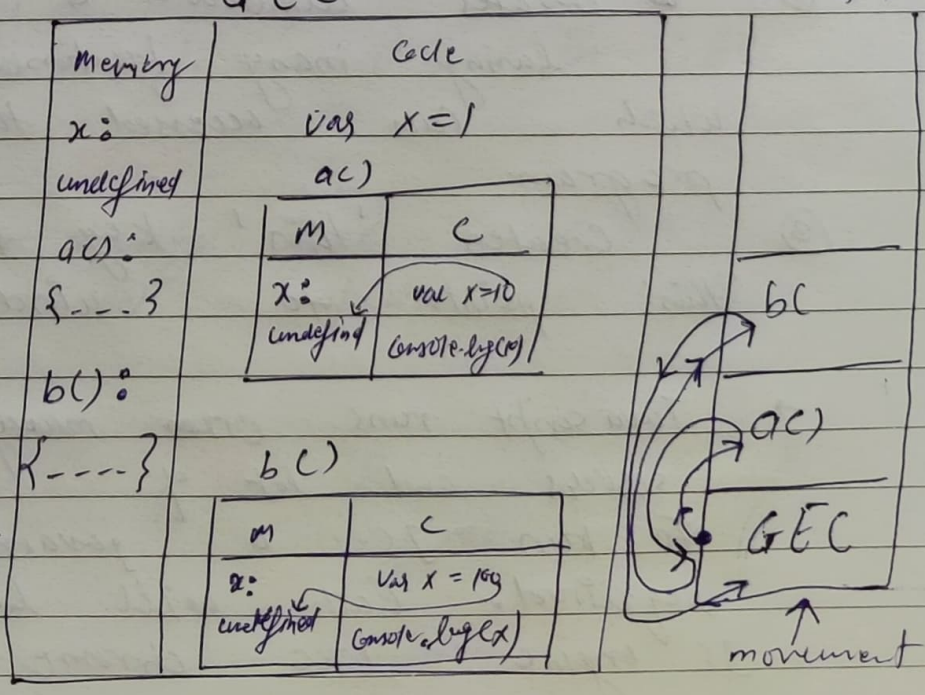
☆ Arrow function syntax act as variable in memory creation phase.

```
var getName2 = function()  → will act as variable
{ ---- }
```

☆

```
var x = 1;
a();
b();
console.log(x);

function a()
{ var x = 10;
  console.log(x);
}
function b(){
  var x = 100;
  console.log(x);
}
```



→ a() creates a local execution context and prints value of x (10) and control returns back to GEC

→ b() creates a local execution context and prints value of x (100) and controles returns back to GEC

→ GEC prints value of x (1).
→ Call back Now becomes empty.

Op -	10	100	1

☆ Shortest Javascript program - file with No code. Run this empty file.
JS engine still did many tasks.

① Creates global execution context
② Creates window - a global object having many funtions and variable. which can accessed throughout the program.
③ Creates 'this' key word
'this' points to window objects

→ Javascript runs many browsers, sesvers and lot of other devices. To run JS a javascript engine is required. Each will have different JS engine like chrome will have

different, firefox will have different.

→ Each engine creates a global variable.
→ Chrome JS engine — V8 and global object/variable is as window.

☆ this === window ; checking if 2 things are exactly equal or not
op : true

☆ Global — means which are not inside any functions. when global variables/functions are created they get attached to window.

```
var a = 10;
function bc() {
    var x = 10;
}
console.log(window.a);
console.log(a); → default is global EC
console.log(this.a)
console.log(x); → x is not present in GEC.
```

op :
10
10
10
uncaught referenceError.

☆ Javascript is a loosely type language.
→ variables are not specific to data structure.

```
var a;
console.log(a);
a = 10;
console.log(a);
a = "Vamadeen";
console.log(a);
```

op
undefined
10
Vamadeen

Page
Date

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

op
undefined;

But not a healthy practice. It may leads to lot of inconsistencies.

★ Scope:
It means where you can access a specific variable or function in a code.
Scope is directly depend on lexical environment

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

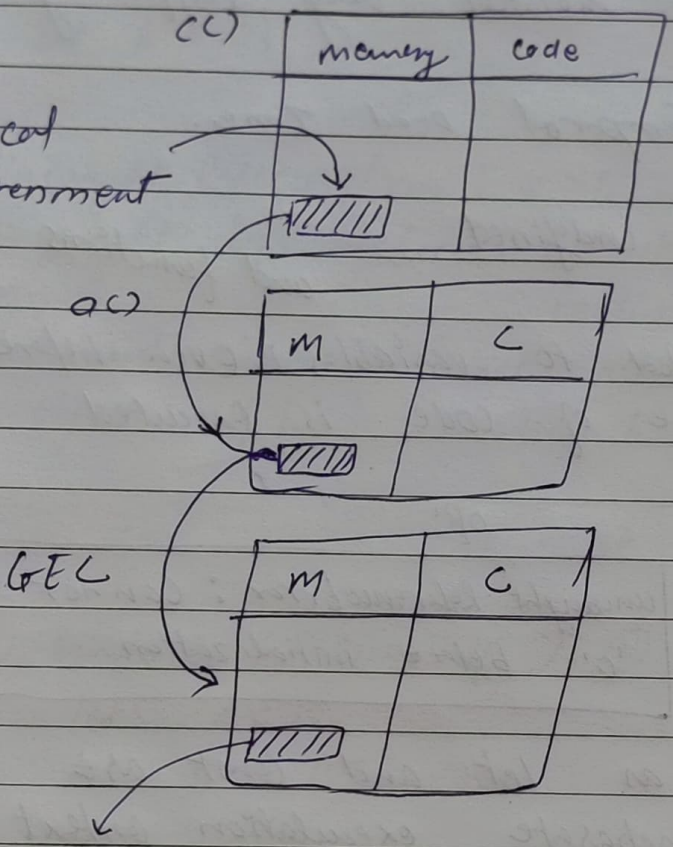
op
10

②
```
function b() {
    var x = 10;
}
b();
console.log(x);
```

will through Uncaught ReferenceError as scope of variable x is inside b() only.

★ When local execution context is created then it holds the lexical environment of its parents. alongs with variables and functions of that Execution Contexts.

In code ① lexical environment of cc() will point to aa() EC and aa()'s lexical environment will point to GEC. and GEC lexical environment point to null.



scope chain: chain of lexical environment references.

JS engine first will search memory of that EC if not found. then through lexical environment, will go parents of it and will perform search and if again not found variable. it will repeat untill it encounters a null in GEC lexical environment.

And finally JS engine will throw an error, variable is 'not undefined'

Page
Date

★ Let and Const in JS ★

Temporal Dead Zone

★ let and const declarations are Hoisted
But they are hoisted very differently
↓
They are in Temporal Dead Zone.

→ console.log(b); → undefined
var b = 100;
memory is allocated to variables & even before a single of line of code is executed and functions

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

op:

uncaught ReferenceError: Cannot access 'a' before initialization

Throwing error as let and const are hoisted in a separate execution context with value • undefined. (Not in global EC and we are accessing in • GEC (by default) Hence throwing an error). let and const variables can't be accessed until • unless they are assigned

Temporal dead zone - The time period between allocating memory and assigning values to let and const.

let and const remains in Temporal dead zone until they are assigned a value..

after

and ↑ that only they can be accessed.

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

let b = 10;
```

op:

① Uncaught ReferenceError: x is not defined
② Uncaught ReferenceError: Cannot access 'a' before initialization.

→ let and const are not attatched to window/this object. [i.e window.b will give undefined]

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

op: 10

☆ one can't redefine let.

① `console.log("Vermadeen");`
`let a = 10;`
`let a = 100;`

② `console.log("VK");`
`let a = 10;`
`var a = 100;`

op:- Syntax Error - Even a single of code will not executed.

↑ for both codes → Not possible in same scope.

while.

```
console.log("VK");
var a = 10;
var a = 100;
```

→ op: VK perfectly fine.

☆ let can be initialized anywhere in program However const must be initialized at the time of declaration only. And their value can't be change

Invalid JS code

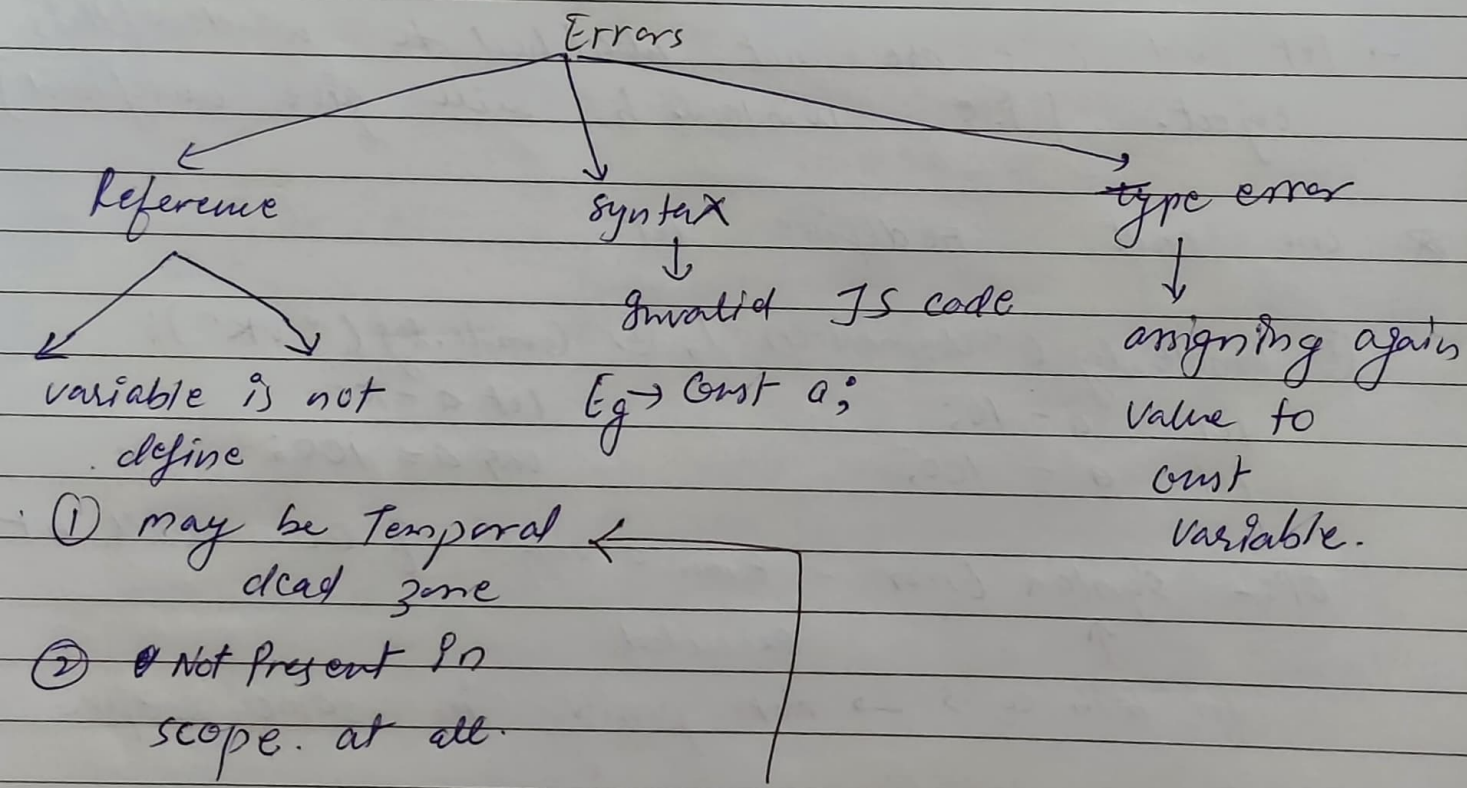
For

const a; → Throws Syntax Error: Missing initialization in const declaration

const a=10; → Type Error: Assignment
a = 100; to constant variable.

var < let < const

Restrictions. [preference of use is reverse order]



To avoid Temporal dead zone all declaration and initialization at the top of the JS code.

↓

we shrinked Temporal zone to zero.

☆ Block Scope And Shadowing In JS. ☆

→ Block is defined by {.....}. Every thing written inside these brackets comes in a block.
It is a compound statement which starts with '{' and ends with '}'. It is used to group multiple items in a single unit.
↓
These group of statement can be use where JS expects a single statement.

→ Each block acts separate block scope. / execution context where variable / fenctions defined inside a block will be hoisted.
Out side of this block no element will be accesible which is declaration inside this block.

→ let and const are block scoped because they have separate block in which they are hoisted.

☆
```
var a = 10;
{
  a = 12;
  var b = 11;
  console.log(a);
  console.log(b);
}
console.log(a);
console.log(b);
```

OP:
10
11
10
ReferenceError: b is defined in this scope.

Page
Date

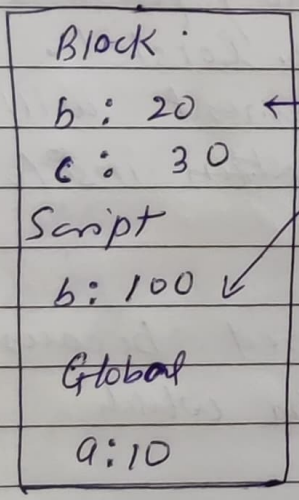
Shadowing

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

op:
10
20
30
100

★ If there exists a same named variable outside the block then this variable shadows the value of outside variable.

Scope:



Both b indicates different memory location.

→ similar thing happens with const also.

```
★ ① let a = 100;
  ② { var a = 10;
  ③     let b = 20;
  ④     const c = 30;
  ⑤   console.log(a);
  ⑥   console.log(b)
  ⑦   console.log(c)
  ⑧ }
  ⑨ console.log(b);
  ⑩ console.log(a);
```

both associated to global scope → points same memory location

op:
10
20
30
100

Page
Date

Scope :
→ Block
b : undefined
c : undefined
→ Global
a : 10

→ when code is executed upto ③ ↓ value of global will be modified to 10. When Block is executed then again value of a will Not change to 100.

- line (5) and (10) → both are refering to same memory location.

★ Function shadowing also works in similar fashion.

```
Const c = 100;
function x(){
  const c = 30;
  Console.log(c);
}
x();
Console.log(c);
```

op - 30
100

★ ① let a = 10
```
{
  let a = 100;
}
```

② var a = 10
```
{
  var a = 100;
}
```

③ let a = 10
```
{
  var a = 100;
}
```

④ var a = 10
```
{
  let a = 100;
}
```

①, ②, ④ → shadowing (works fine)

③ → shadowing is not possible. - Illegal shadowing

Page
Date

let a = 10
function x(){
  var a = 100;
}

✓ [Not illegal shadowing]

→ similar behaviour with const variable also

Block scope also follows lexical scope.
↓
follows scope chain pattern.

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

op- 200

Scope
Block
a: 200
Block
a: 100
~~Block~~ Script
a: 20

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

op - 100

☆ All ~~#~~ scope concepts work exactly in same fashion for arrow function also.

function
var a = abc(){ .... } ✓

Page
Date

★ Closures in JS.

A Function along with its lexical scope forms a closure.

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

op- 07

Scope
Local
this : Window
Closure (x)
a: 7

★ The function y() is ~~bind~~ wind to variables of x(). Hence y() is ~~forming~~ a closure.

→ A function along lexical environment of its parents is known as closure.

→ A function bundled together with references to its lexical environment.

★ JS allows us to pass a function as argument in a function

```
function x( var z)
{ console.log(z); }.

function y() {     };
var z = 10;
x(y);
```

op- 10

★ you can a function into a variable.

Eg. var a = x();

JS is synchronous language

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

op- ƒ y() { console.log(a); }

immediately after return x(), will go out of call stack; and hence y also. and 'a' will also not exist.

But accessed out of scope of x

```
z();
```

→ op: 7

closure comes into action.

due to this y() will remember its lexical environment

[function + variable] + LS

when returned y(), it also returns its lexical environment. and ie. it returns complete closure. (It also remembers the variable reference)

☆
```
function y() {
  console.log(a);
}
return y;
```
==
```
return function y() {
  console.log(a);
}
```

☆
```
function x() {
  var a = 7;
  return function y() {
```

```
function x(){
    var a = 7;
    function y(){
        console.log(a);
    }
    a = 100;
    return y;
}
var z = x();
console.log(z);
```

function y() returns reference of 'a' [not value at 'a']

↓

Closure Returns reference to variables

```
z();
```

→ op - 100 (Not 7)

Scope

- Local
- this: Window
- Closure (x)
  - a: 7
- Closure (z)
  - b: 900

☆

```
function z(){
    var b = 900;
    function x(){
        var a = 7;
        function y(){
            console.log(a,b);
        }
        y();
    }
    x();
}
z();
```

← debugged

Page
Date

Uses of Closures:
- module design pattern
- currying [function currying]
- functions like once ← Runs only once.
- memoize
- maintaining state in async world
- setTimeouts
- Iterators
- and many more.
- data hiding and encapsulation

☆ setTimeouts ☆

→
```
function x()
{
    var i = 1;
    setTimeout ( function () {
    {   console.log (i);
    }, 3000) ;
        ↑Time in milliseconds
}
x();
```
→ It will log 1 after 3 seconds.

→
```
function x()
{   var i = 1;
    setTimeout ( function () {
        console.log (i);
    }, 3000) ;
    console.log ("Namaste Javascript");
}
x();
```
op:
Namaste Javascript
Across After 3s → 1

on OnePlus

Page
Date

Explaination :

setTimeout take call back function and stores it somewhere else. and sets times of given time. And Rest do not waits. It keeps executing. When time overs, setTime function / variable also logged on Console

→ JS do not waits. for anything

★ Print 1 after 1s, 2 after 2 ~~nd~~ second and so an till 5 ?

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

wrong code.

JS do not behaves in this way.

Op:

Namaste Javascript
6 → after 1s
6 → " 2s
6 → " 3s
6 → " 4s
6 → " 5s

JS working in this due to Closure

Page
Date

Closure

→ variables bundled with closures points to same memory location. i.e. their reference is bundled.

→ JS do not waits for anything

→ when for loop breaks value of i will be 6.

→ setTimeout puts functions in a separate place.

☆ → JS runs for loop 5 times and each setTimeout put separate copy function. but variable pointing to same memory location [references].

→ without waiting for anything.

↓

When time overs, it prints value pointed by variable i [6].

☆
```
function x(){
    for( let i=1; i<=5; i++){
        setTimeout( function(){
            console.log(i);
        }, i*1000);
    }
    console.log("Namaste Javascript");
}
x();
```

Page
Date

cp

Namaste Javascript

1 → After 1s
2 → " 2s
3 → " 3s
4 → " 4s
5 → 4s

let variables are Block scoped. for each iteration a completely different variable let variable. will be used having same name.

Each time a separate closure will be formed with new variable.
↑ each time referring to different memory location

→ Without use of let variable.

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

value of i will be copied into j and each new j variable will used.

op:

Namaste Javascript
1 → 1s
2 → 2nd
3 → 3rd
4 → 4rth
5 → 5th

Page
Date

★
```
function outer(){
    function inner(){
      console.log(a);
    }
    let a = 10;
    return inner;
}
var close = outer();
close();
```

we can replace by

```
outer()();
```

both syntax are valid and meaning is same.

OP - 10

No difference using let instead of var

★
```
function outer(b){
    let a = 10;
    function inner(){
      console.log(a,b);
    }
    return inner;
}
var close = outer("Hello World");
close();
```

OP:
10 "Hello World"

★ inner() will have access of parameters & func, variables of outer(); (i.e. will be included in closure)

★
```
function outest(){
    var c = 20;
    function outer(b){
    let a = 10;
      function inner(){
        console.log(a,b,c);
      }
}
```

outer("hello world") ~~=~~ (outest())("hello world");

Page
Date

```
    return inner;
}
return outer;
}
var close = outest() ("Hello World");
close();
```

will return outer() and again outer() will be called "Hello world" argument and will return inner()

// in close → inner function will be assigned

op- 10 "Hello World" 20.

☆
```
function outest() {
  var c = 20;
  function outer(){
    function inner(){
      console.log(a, b, c)
    }
    let a = 10;
    return inner;
  }
  return outer;
}
let a = 100;
var close = outest()("hello world");
close();
```

op:
10 "hello world" 20
however if a in outer is removed then output will be
100 "hello world" 20
[a is present in global execution context and will be accessed through scope chain resolution)

☆ If a is removed from both sites, then it will throw referenceError: a is not defined.

Page
Date

☆ Advantages of JS.
Earlier mentioned.

★ ① data hiding and encapsulation

```
function counter(){
  var count = 0;
  return function incrementCounter(){
    count++;
    console.log(count);
  }
}
var count1 = counter();
count1();
count1();
```

OP: 1 2

count can be accessed through count1 but ~~any~~ if we try to access count variable ~~directly~~ directly then it will give Error.

→
```
var count = 0;
function incrementCounter(){
  count++;
}
```
available

Now count is ∧ for all function and operation (~~short~~ public variable)

So data hiding and encapsulation is possible through closure.

```
function counter(){
  return function increment count(){
    count++;
    Console.log(count);
  }
}
Var count1 = counter();   ] separate closure and E.C.
count1();
count1();
Var count2 = counter();   ] separate closure.
count2();
count2();
```

op:
1] count1
2
1] count2
2

☆
```
function Counter(){
  Var count = 0;
  this.counter increment = function(){
    count++;
    Console.log(count);
  }
  this.counter decrement = function(){
    count++; count--;
    console.log(count);
  }
  Var counter1 = new Counter();  // Constructer hence new keyword is used / required
Counter1.counter increment();
Counter1.counter increment();
Counter1.counter decrement();
```

OP:	1
	2
	1

Disadvantages

till the program runs

① Variable do not ~~auto~~ garbage collected. Hence over - consumption of memory.

② If not handled properly then it will lead to memory leak.

Javascipt is a high level programming language. It is upto programmer how memory should used. A Garbage collector is a program which freezes the variable ~~etc~~ ~~no~~ which are no longer needed.

★ Relation between closure and Garbage collection

```
function a() {
    var x = 100;
    return function b() {
        console.log(x);
    }
}
var y = a();
```

★ b() forms closure with x.

y(); → variable x will not be garbage collected.

Page
Date

★ JS engine in chrome browser [V8] smartly does the task of garbage collection

```
function a() {
var x = 10; z = 10;
return function b() {
  console.log(a);
}
}
var y = a();
y();
```

★ Although z is part of closure still it will be garbage collected.

console.log(x) → 10

console.log(z) → ReferenceError: z is not defined.

Interview Questions.

① function statement VS function expression

① function a() {
console.log("a called");
}
a();
op: a called

② a way to create functions

③ a();
function a() {
console.log("a called")
}

① var b = function() {
console.log("b called");
}
b();
op: b called

② a way to create functions

③ b();
var b = function() {
console.log("b called");
}

op: a called

op: TypeError: b is not a function

☆ differ in hoisting. In memory creation phase of Execution context, exact copy of a() will be stored as a() is complete function while b will a variable and hence it will • undefined. If you try to run as a function Throws an Error

(2) Function Declaration == function Statement

(3) Anonymous function : → function without a name

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

← invalid syntax. → SyntaxError: Requires function name

→ Do not have its own identity.
→ It is an invalid statement
→ These are used where we want to function as values.

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

✓ Function Expression

(4) Named Function expression.

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

b(); → b called

x(); → ReferenceError : x is not defined

because x() is created as local function only. If you try to access in out of scope. it will give ReferenceError.

④ var b = Argument Vs paramters

```
var x = function ( parameter1, parameter2){
    console.log (" x called");
}
```

x (1, 2)

values passed are called arguments.

labels or identifiers are called parameters. These are local.

⑤ First Class function.

passing a function inside a function as an argument

①
```
var b = function ( param) {
console.log( param) ;
}
function xyz() {
}
b( xyz);
```

op: f xyz(){

}

Page
Date

★ Callback Functions ★

→ functions are first class citizen.
The functions which is being passed as argument is known as callback functions.

→ Javascript is a synchronous and single-threaded language.
due to callback function, we can do asynch things.

one thing at a time in a specific order.

```
function x(y) {
...
}
x( function y(){

});
```

known as callback function

→ It is upto x() when y() should be called.

☆ Asynchronous

callback function

Registers timeout in a seperate space of 5s

JS engine do not wait for finishing of timeout.

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

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

op: x
y
times — after 5s

Call Stack.

x
y
] → after that → empty

Call Stack

[After 5s → (Anonymous)

for setTimeout function (comes automatically)

Call stack

JS has only one call stack.
If anyone operation blocks the call stack then it is known as blocking the main thread.

Very-very heavy operation which takes around 20-30 s to execute.

→ at this stage JS will not able to execute any other operation
→ we should never block main thread.
Hence we should use asynch operation which take time

↓

setTimeout take call back and execute it sometime else. this peice of code and just get out of call stack.

☆ If there are no callback function, first class functions. — No asynchronous operations will be executed.

Page
Date

☆ web Api [the setTimeout] and Call back function we can acheive asynchronous operations

☆ Deep about Event listners. ☆

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Akshay saini</title>
</head>

<body>
    <h1 id="heading">Namaste Javascript</h1>
    <button id="clickMe">Click Me</button>
    <script src="js/index.js"></script>
</body>

</html>
```

Now attaching Event listeners to above html file.

dir → JS | index.js

script is executed in html file.

```
document.getElementById("clickMe").
addEventListener("click", function x() {
    console.log("Button clicked");
}
```

addEventListener("click", function x() { ... }) ← callBack

Adds an Event listener click to html/JS when "Click Me" Button is clicked then the function x() will called automatically.

automatically comes in call stack.

OP - Browser

Namaste Javascript

Click Me. → When event of clicking "Click Me" Button happens then.

OP: Console

Button Clicked

Call stack

x()

★ How to count No of time Button clicked

```
let count = 0;
document.getElementById("clickMe").addEventListener
("click", function x() {
    console.log("Button clicked", ++count);
});
```

Page
Date

Using a global variable is not good solution. → Use closure
↓
data hiding

```
function attachEventListener() {
  let count = 0;
  document.getElementById("clickMe").addEventListener
  ("click", function x() {
    console.log("Button clicked", ++count);
  }
}
attachEventListener();
```

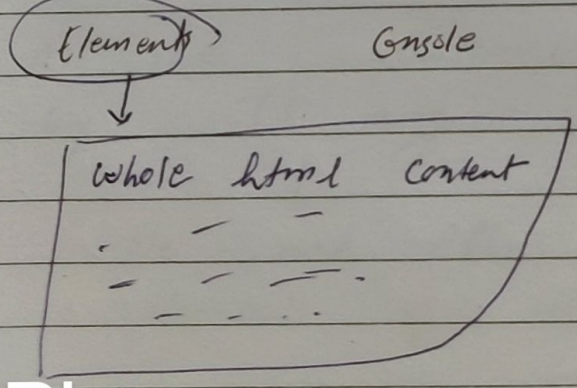
Debugger →

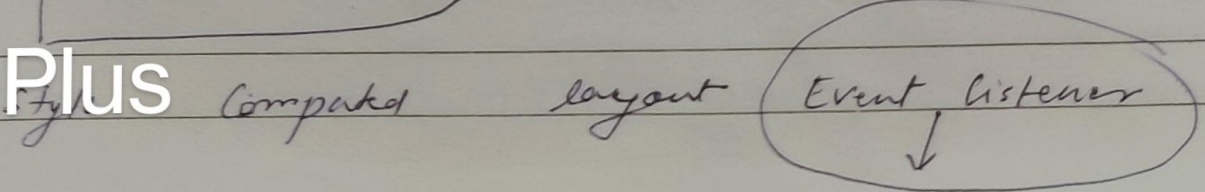
Call stack:
x

Scope
★ local
this: button # clickMe
★ Closure (attachEventListener)
count: 3

whenever Button is clicked then count will be increased by 1

★ In dev tools





↓

click

→ button # clickMe

→ handler : f x()

→ [[Scopes]]: Scopes[2]

1: Global

0: Closure (attachEventListener)

{ count : 0 }

→ __proto__ : f()

→ prototype : { constructor: f }

name : "x"

Why ~~to~~ remove Event listeners ?

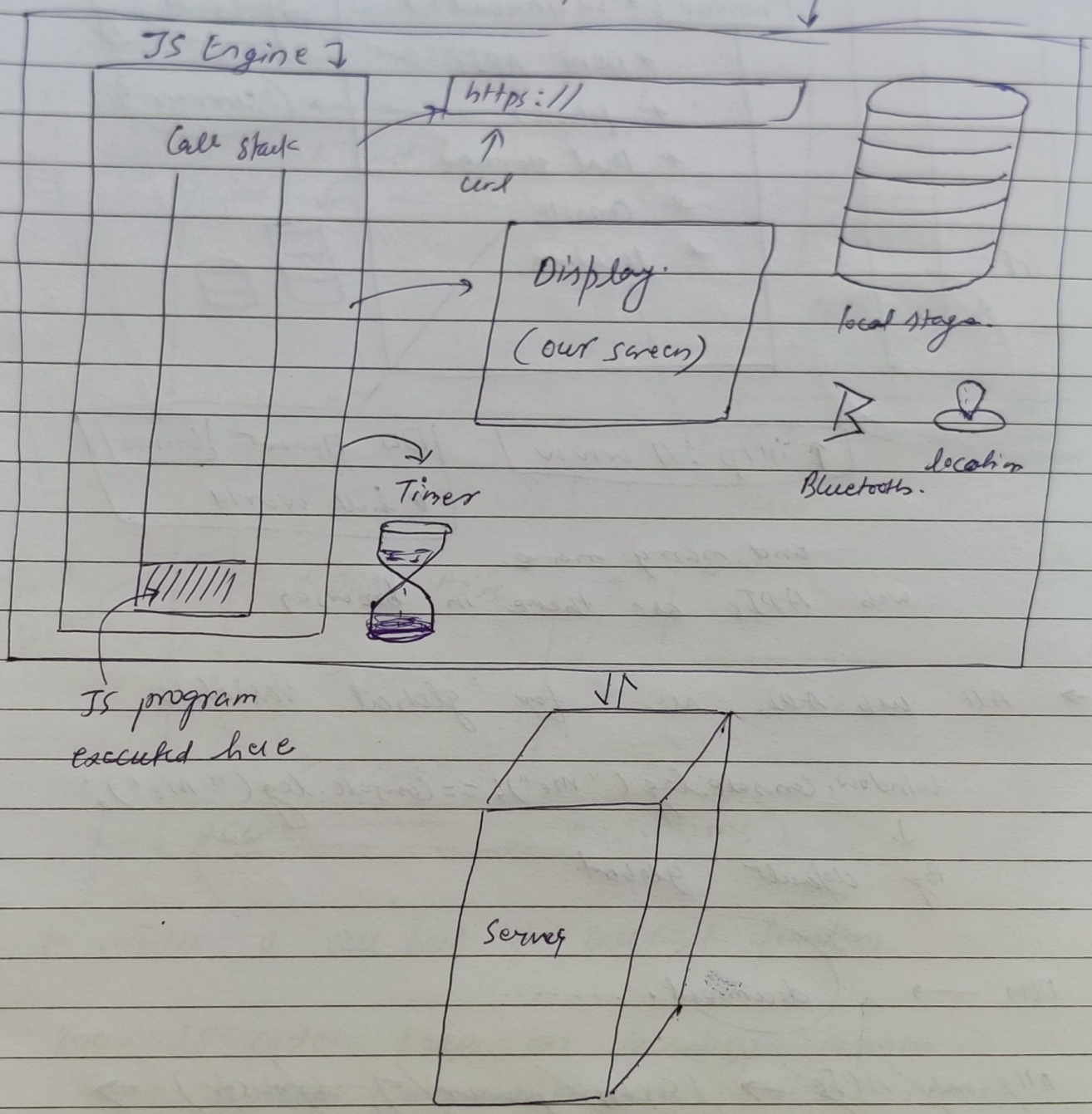
→ Event listeners are heavy i.e. It takes memory.

→ Even though ~~the~~ call stack is empty memory is not released as they forms closures.

→ If Event listners are removed then all those which were held by closure will be garbage collected

Page
Date

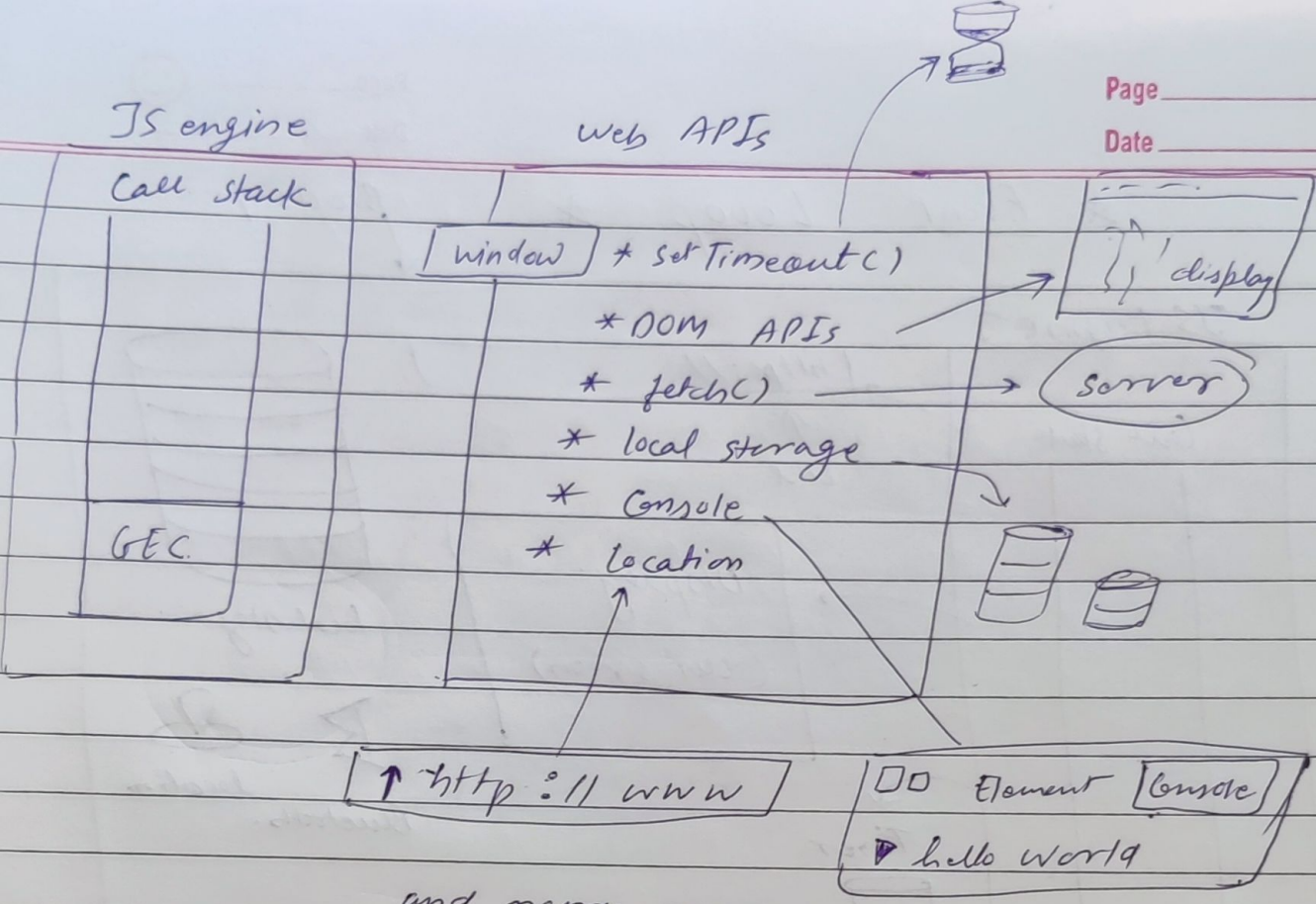
☆ Event Loop ☆



All the funtionality, how can be accessed by JS engine?

↓

we web APIs to access all the elements of browser



and many more web APIs are there in browser

→ All web APIs are for global variable.

window.console.log("Me"); == console.log("Me");

↓

By default global.

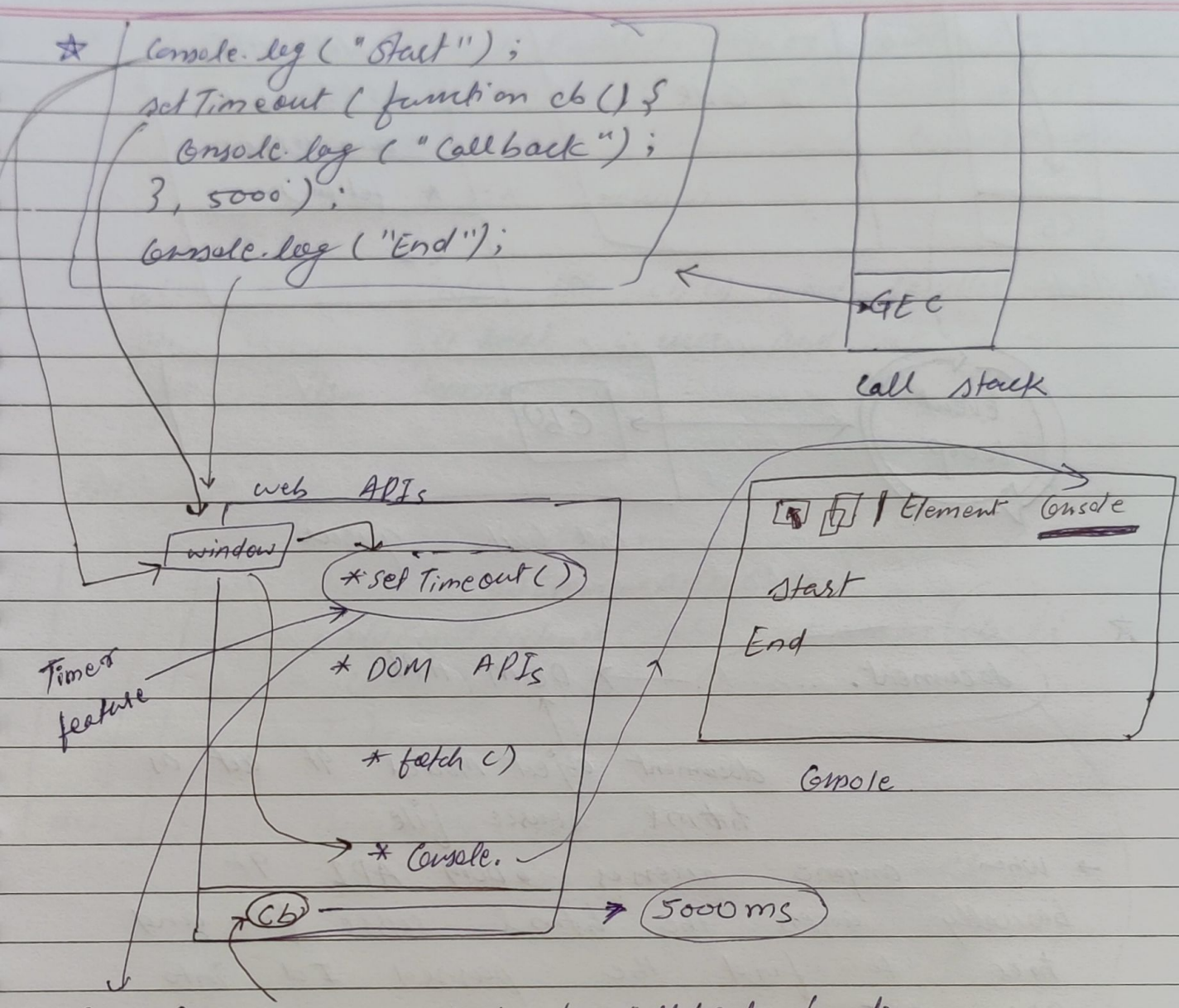
DOM → document. .....

All web APIs → (super power of browser) → can be accessed by JS engine.

Browser wraps all its super powers (like timer, server, local storage etc) into window object and makes accesible to JS engine / call stak through web APIs

→ APIs are plugged into JS code through window

Page
Date

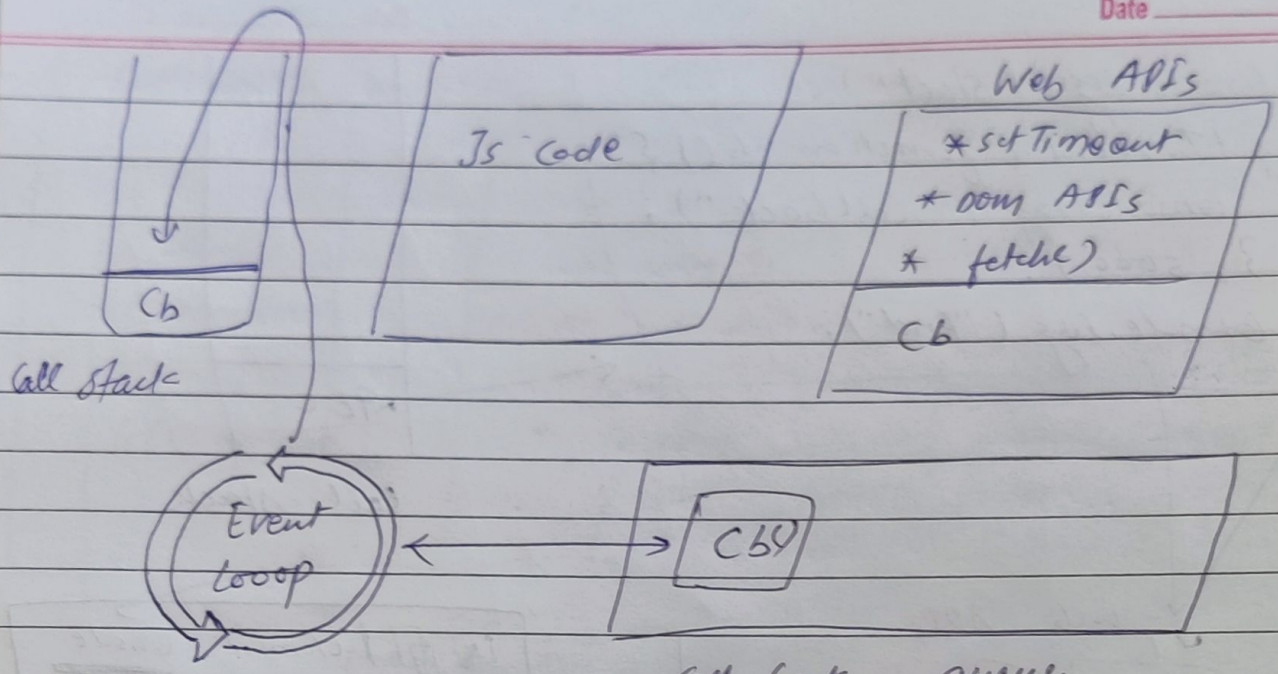


It registers a callback for callback functions

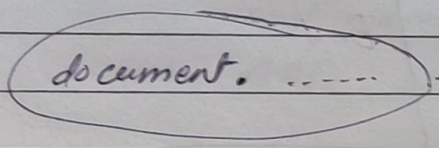
The JS code keeps on executing meanwhile timer keeps on running. When Timer expires. This cb() goes into callback Queue. Event loop keeps checking in callback queue if there is something When Event loop find cb() ~~into~~ in queue it just ~~put~~ pushes into call stack.

Console API → has console() method

Page
Date



☆



document object model. It act as html source file

→ when anyone access DOM API it basically access the html source file and tries to find the passed Id into DOM API

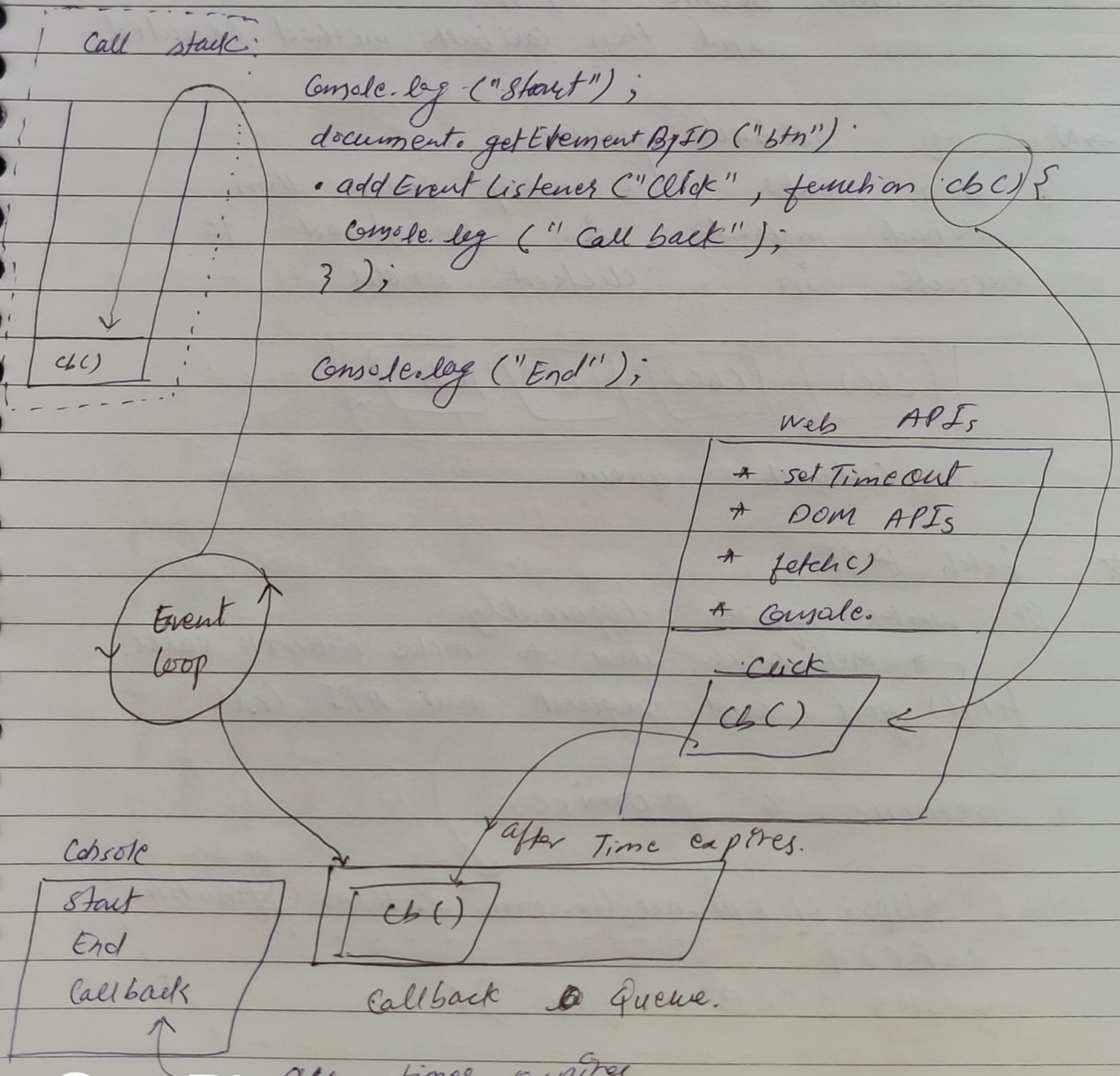
→ addEventListener registers a callback cb() on event "click" in web APIs environment.

method → Event handler.

→ After all lines of code of JS are executed GEC goes out of call stack.

→ This cb() event handler method will stay in the APIs environment untill and unless explicitly we remove this event listener or we close the browser.

→ When a user clicks, ~~the~~ cb() event handler method goes into call back queues and it waits for its turn.



Page
Date

☆ Event loop — Super hero

↓

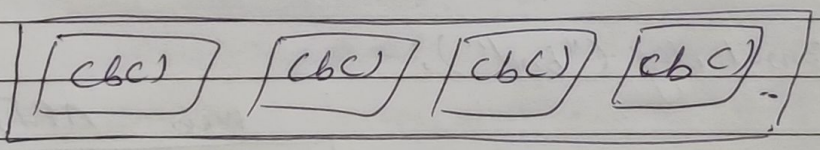
It has only one job.

→ Continuously monetors the call stack and callback queue

→ If it find a function in callback queue wait list, it ~~take~~ takes and pushes into call stack. after call stack become empty. ↓ and then callback method executed

→ Need of callback queue

If user clicks so many times then callback methods has to stored to execute in clicked orders.



callback queue

☆ Fetch API

It works quite differently.

→ Method is used to make Network calls.

fetch() goes and requests and API Call

↓

It returns a promise.

```
fetch("https://api.netflix.com").then(function
--- cbF(){
  console.log("CB Netflix");
});
```

callback function

Page
Date

The call back function is executed once the promise is resolved.

→ JS engine does a task at a time but browsers can do multiple tasks at a time

Page
Date

☆ micro task queue VS Callback Queue
also known as Task Queue

→ It has higher priority than callback Queue.
→ All the methods of this executed then only callback queue gets chance to execute.
→ If methods forms a infinite loop inside microtask queue then call back's methods/ functions will never get opportunity to execute. This stage is called Callback Queue starvation.
→ In microtask queue promises and mutation observer are executed.
→ In callback queue setTimeout and Dom APIs are executed.
→ Waiting period of microtask Queue depends upon speed of server. Server jitni jaldi response dega utni jaldi hi available ho jayega.
→ Callback Queue waiting period timer par depend karta hai.
→ Event loop keeps on checking if call stack is empty. If call stack is empty then only methods of microtask are pushed into call stack. When microtask queue becomes empty then only callback queue methods are pushed into call stack.

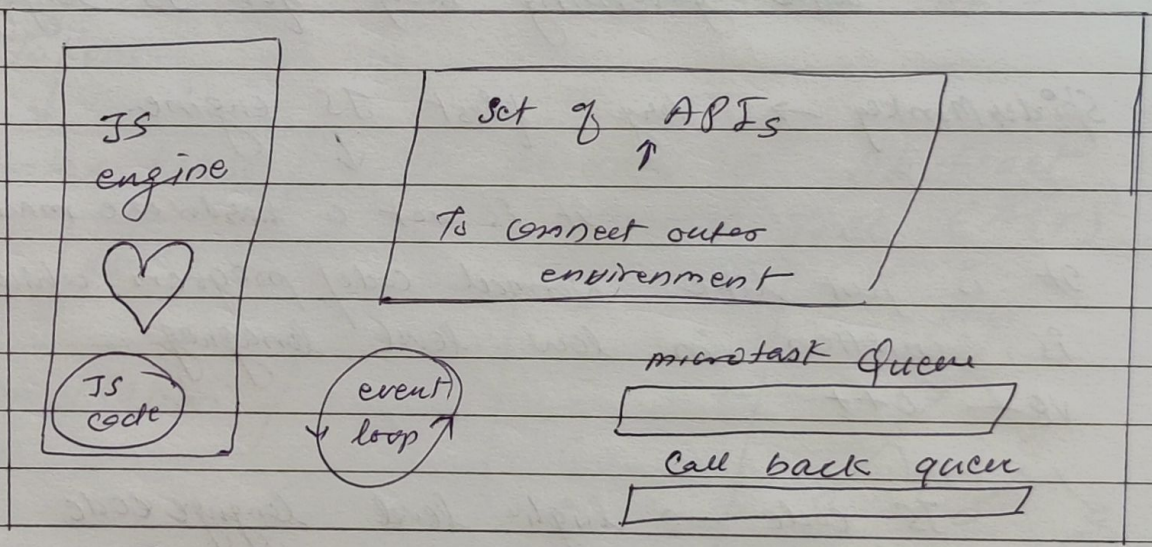
JS Engine.

JS — can be executed in a browser
— " " " " " server
smart watch
Robot
smart bulbs

→ It is possible because of JSV environment. has Runtime.

→ Js runtime environment, all the things required to run the js code.

→ Js runtime environment is a big container.



Js RunTime Environment. Elements

→ Every Broswer is having JS RT Environment

→ Node.js — It is an open sources JS RTE. It can Run JS code outside the broswer.

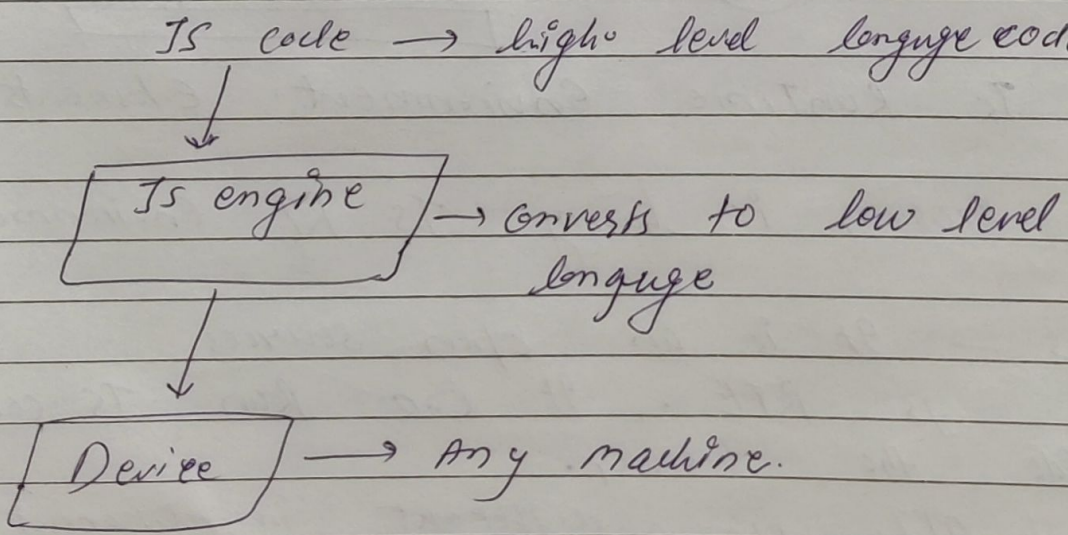
→ Set of APIs may different in different RTE [Node.js and Broswers may differ in APIs]

→ SetTimeout API's of Node.js and Browser may differ in their implementation.

★ List JS engines.

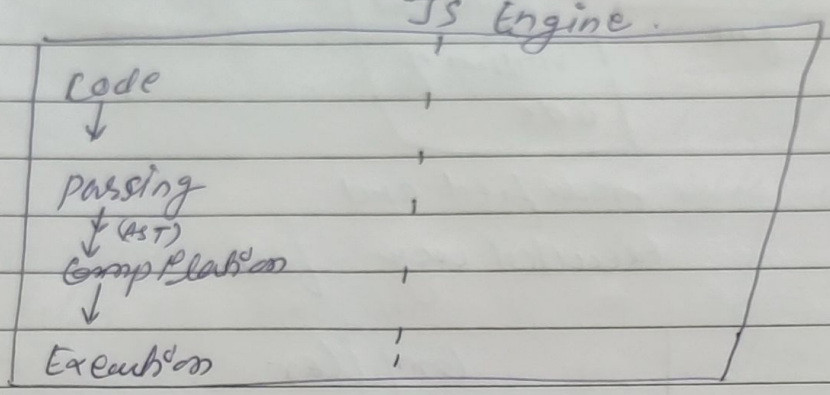
Chakra → Internet explorer (edge)
SpiderMonkey → Firefox
V8 → Google chrome & Nodjs, Deno
one can also generate a JS engine
↓
must follow ECMAScript Standards
↓
like governing body for JS language

SpiderMonkey → very first JS engine.
↓
It is not a hardware machine
It is just like Normal code/program which is written in low level language.
V8 — C++.

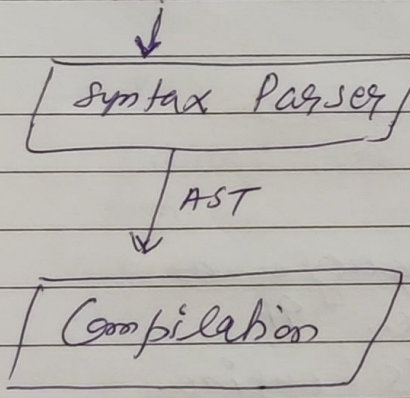


★ JS Engine Architecture ★

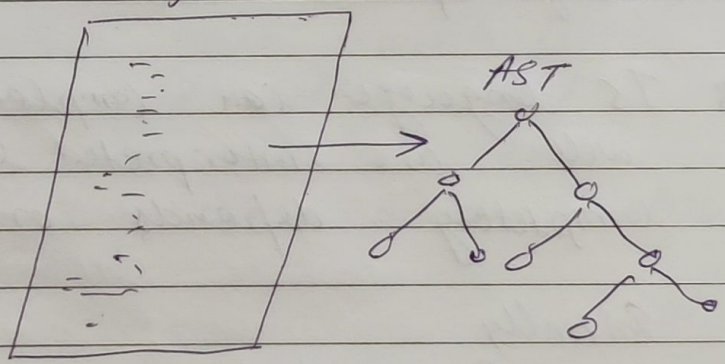
check out — astexplorer.net



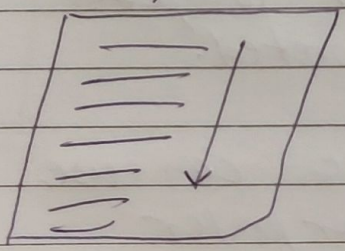
★ Passing - It breaks humen readable JS code (high level) into smaller units called Tokens.



It generate a Abstract Syntax Tree (AST)

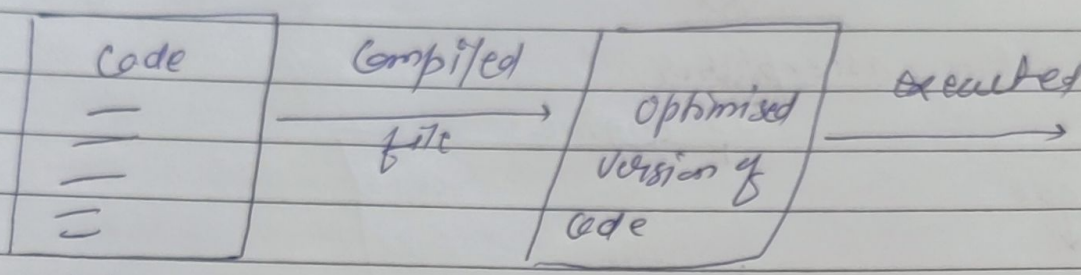


★ JIT Compilation [just in time]



Code is executed line by line.

Compiler



origin code

modified and executed very fast.

Interpreter	Compiler
→ code executed very fast. It does not have to wait for compilation of code. → Immediately start execution → high speed	→ more efficiency

☆ JS language can compled languages as well as interpreted language. It completely depends on JS engine.

Initially
JS - Interpreted as broswer donit have to wait

Now
Inter preted + Complited.
↓
modern JS engines are both compiled + Interpreted.

JIT compilation → JS engine can use compiler along with interpreter

Various optimizing → { inlining, copy elision, Inline caching and many more } which make just in Time Compilation

☆ Compilation → AST → Interpreter ↔ Compiler

Converts high level code to low level (byte code) → Execution

Interpreter takes helps of compiler to optimise the process.

→ Compiler - It just optimises the process as much as it can. on the run time. [Just In Time Compilation]

→ In some JS engine there is AOT. (ahead of Time Compilation). Here Compiler takes the code, which is supposed to be executed later, and optimise as much as it can. and generates a byte code which is then passed for execution.

☆ Execution

Execution components → memory heap, call stack.

same call stack that we are using

Page
Date

Memory heap → All memory is stored here
→ It constantly in sync with call stack, Garbage collector and lot of other thing

All the variables and functions → memory is assigned from here only.

★ Garbage Collector → It try to free up memory space whenever possible. Collects all the undesirable functions and variable and sweeps it.

uses Mark and Sweeps Algorithm] H.W.
~~widely used~~ algo across the garbage collectors

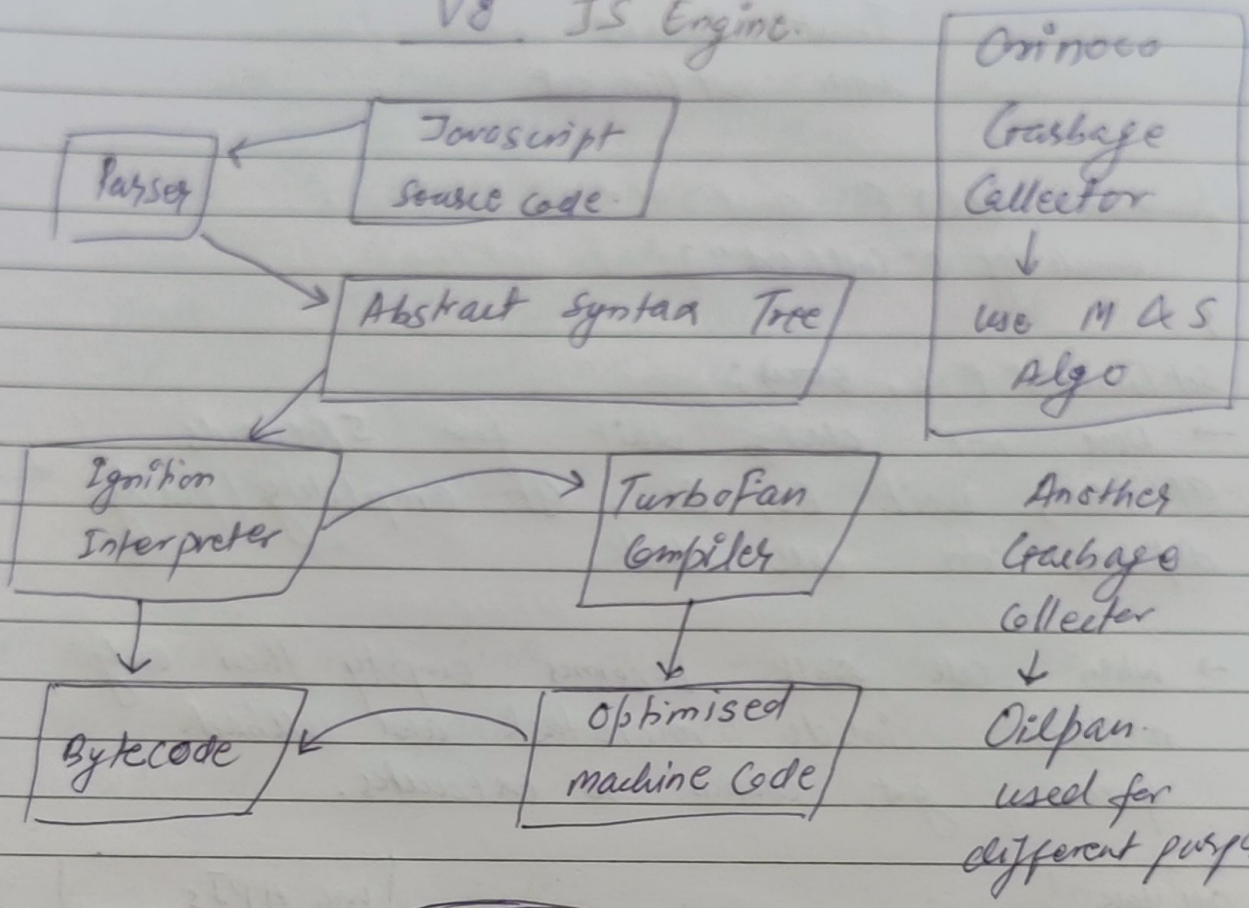
Compiler optimises by various operations
① Inlining
② Copy elision
③ Inline Caching
and many more.
] H.W.

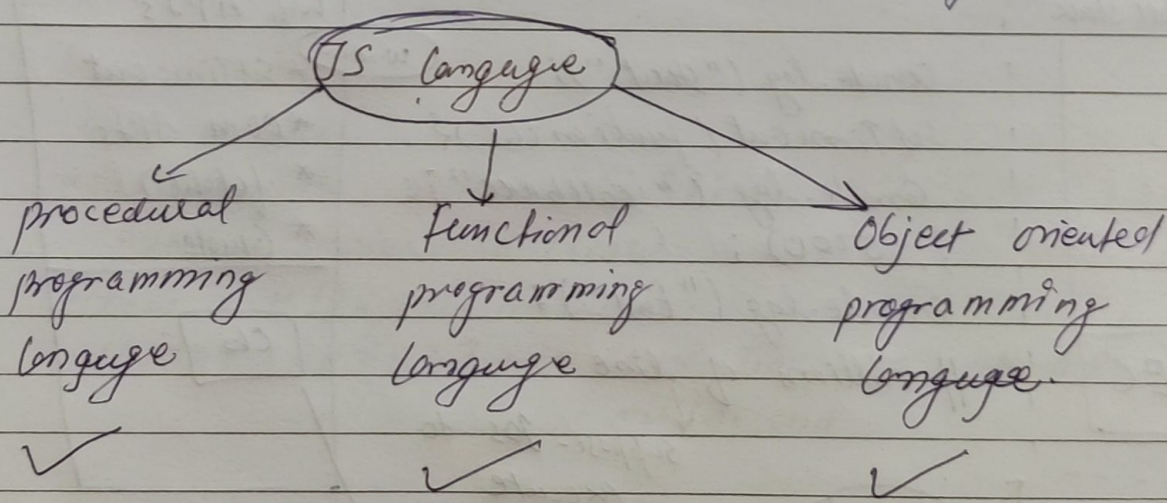
V8 - fastest JS engine right now
→ name of interpreter given by google
→ ignition Interpreter

→ Turbo fan v compiler for optimisation
~~keep~~ optimising

Page
Date

V8 JS Engine





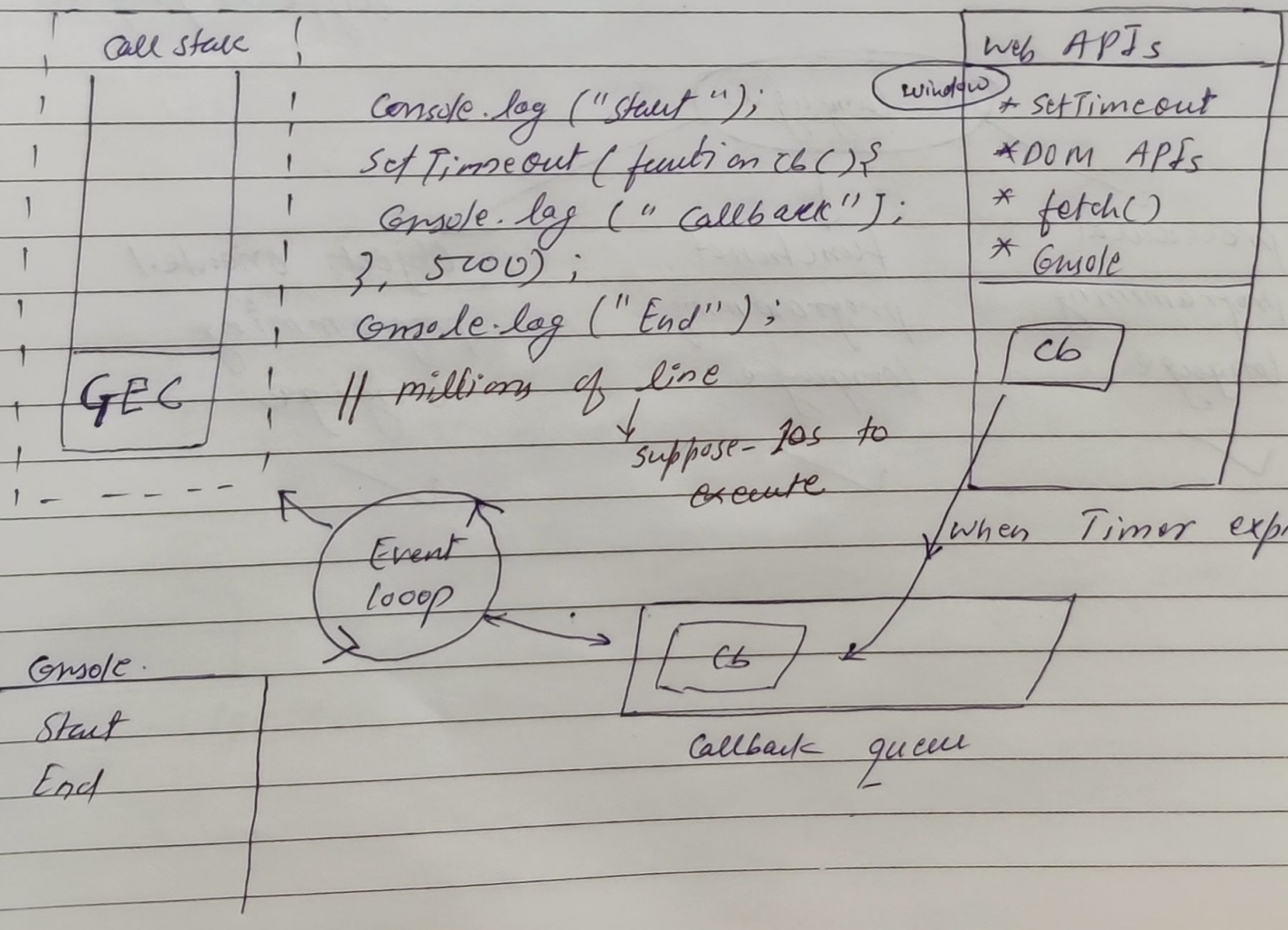
✓ ✓ ✓

Trust Issues with setTimeout

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

→ Does not always wait for 5 seconds. It may wait 6s, 8s . It completely / purely depends on call stack.

→ When call stack becomes empty then only methods inside micro task and callback queue get chance to execute.



concurrency model

Page
Date

→ Main thread / call stack become empty 10s. Meanwhile Timer will keep on running and will expires in 5s Then cb() have to wait for 5s more in callback Queue. After that event loop will push it into call stack then cb() method will be executed.

→ setTimeout will Take 10s instead of seting Times of 5 seconds.

→ We should never block main thread because. asynchronous part of JS will not be executed.

☆ Date APIs explanation
on console.
new Date();
→ Sat Jan 16 2021 22:37:42 GMT+0530 (India Standard Time)
new Date().getTime();
1610816887527 ← exact Time in millisecond

☆

```
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 ");
```

op - Start
End
while expires } After 10s.
callback

☆ `console.log ( " Start ");`

```
setTimeout( function cb(){
    console.log (" callback ");
}, 0);

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

op → Start
End
callback

Page
Date

☆ Functional Programming ☆

→ Higher order function - A function which takes a function as argument returns a function is called higher order function.

→
```
function x() {
  console.log("Namaste Javascript");
}
function y(x){
  x();
}
```
y() → is higher order function
x() → call back function

☆
```
const radius = [3, 1, 2, 4]; // radius of 4 circle.

const Area = function (radius) {
  const output = [];
  for ( let i=0; i< radius.length ; i++){
    output.push ( Math.PI * radius[i] * radius(i));
  }
  return output;
};

console.log ( Area(radius));

const circumference = function (radius){
  const output = [];
  for ( let i=0; i< radius.length; i++){
```

```
    output.push( Math.PI * 2 * radius[i] );
  }
  return output;
};

console.log( circumference(radius) );

const Diameter = function(radius){
  const output = [];
  for (let i=0; i< radius.length; i++){
    output.push( Math 2 * radius[i] );
  }
  return output;
}
console.log( Diameter(radius) );
```

O/P: (4) [28.2743, 3.1415, 12.566, 50.265]

(4) [18.849, 6.283, 12.566, 25.1327]

(4) [6, 2, 4, 8]

→ The above code is fine and gives required output

But

In software Engineering — Dry Principle

don't Repeat Yourself

↓

Not optimised code.

Page
Date

Functional programming deals with

1. Pure functions
2. Composition of functions
3. Reusibility
4. modularity

```
const radius = [3, 1, 2, 4];
const area = function (radius) {
    return Math.PI * radius * radius;
};
const circumference = function (radius) {
    return 2 * Math.PI * radius;
};
const Diameter = function (radius) {
    return 2 * radius;
};

const calculate = function (radius, logic) {
    const output = [];
    for (let i=0; i< radius.length; i++)
        output.push (logic (radius[i]));
    }
    return output;
};

console.log (calculate (radius, area));
console.log (calculate (radius, circumference));
console.log (calculate (radius, diameter));
```

OP: same as previous code.

Page
Date

★ [].map(), [].filter(), [].reduce() ★

map() – These are used to ~~Hello~~ Transform array.

Example
```
const arr = [5, 1, 3, 2, 6];
// double - [10, 2, 6, 4, 12]
// Binary - ["101", "1", "11", "10", "110"]

function double(x){    // Transformation logic
  return x * 2;
}
const output = arr.map(double);
console.log(output);
```
↓
op- 5 [10, 2, 6, 4, 12]

★
```
const arr = [5, 1, 3, 2, 6]

function binary(x){
  return x.toString(2);
}
const output = arr.map(binary);

console.log(output);
```
op- "101", "1", "11", "10", "110"]

equivalent

```
const output = arr.map(function binary(x){
  return x.toString(2);
});
```

→ Const output = arr.map((x) => {
    return x.toString(2);
});

↕ equivalent.

→ Const output = arr.map(x ⇒ x.toString(2));

☆ filter() - used to filter values inside an array.

Const arr = [5, 1, 3, 2, 6];

function isEven (x) { // filter logic
  return x%2 ===0;
}
Const output = arr.filter(isEven);

Console.log (output); →(2) [2, 6]

function isOdd (x) { // filter logic
  return x%2;
}
Const output = arr.filter (isOdd);
Console.log(output);
op - (3). [5, 1, 3]

→ filter logic could be anything like double, greater than a particular value, divisible by a particular value. any many more.

const output = arr.filter ((x) ⇒ x>4);

output = [5, 6]

★ reduce()

It does not reduce anything

Takes an array and come up with a single value out of them.

logic — sum, maximum, minimum,

→ const arr = [5, 1, 3, 2, 6];

```
function findSum(arr){
    let sum = 0;
    for(let i=0; i<arr.length; i++){
        sum = sum + arr[i];
    }
    return sum;
}
console.log(findSum(arr));
```

→ functional parameter

```
const output = arr.reduce(function(acc, curr)
{
    acc = acc + curr;
    return acc;
}, 0);
console.log(output);
```

→ for each value in arr this will be executed

0 ← initial value for acc

op — 17
17

acc = accumulate
curr — current

• accumulator - generic term

Page
Date

acc — it stores the current ~~return~~ result

Here	acc = sum curr = arr[i]

☆ Const arr = [5, 1, 3, 2, 6];
// function findMax(arr) { }

```
const output = arr.reduce(function(max, curr){
    if(curr > max){
        max = curr;
    }
    return max;
},0);
console.log(output);        → 6
```

☆ Const users = [{firstname: "akshay", lastname: "saini", age: 26},
{firstname: "donald", lastname: "trump", age: 75},
{firstname: "deepika", lastname: "padukone", age: 26}
];

// list of full name

```
Const output = users.map((x) => x.firstname + " " + lastname);
Console.log(output);
```

// - how many people are there for particular age.

// {26: 2, 75: 1,}

```
Const output = users.reduce ( function(acc, curr) {
    if( acc[curr.age]){
        acc[curr.age] = ++ acc[curr.age];
    }
    else {
        acc[curr.age] = 1;
    }
    return acc;
}, {});

Console.log (output);
```

op 26 : 2
    75: 1

// first name of all the users who age is less than 30

output

```
Const output = users.filter ((x) => x.age < 30). map((x)
    => x.firstname);

Console.log ( output);
```

chaining is allowed

op- ["akshay", "deepika"]