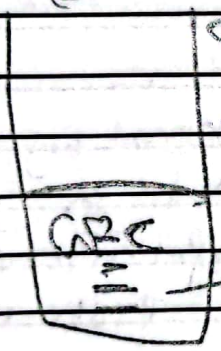
1

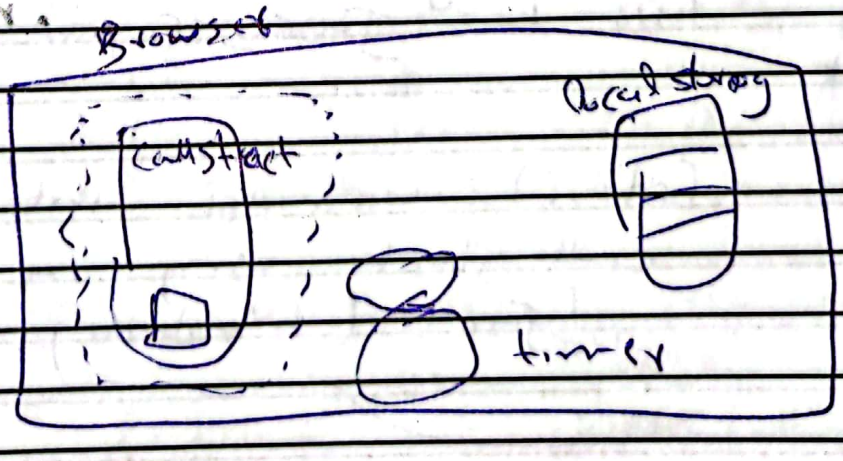
Javascript is a ~~Synchronous~~ synchronic single thread language it can only one call stack and can do only 1 thing at a time.

All the code is present inside call stack.

When Javascript is Run a Global execution context is created and put in call stack

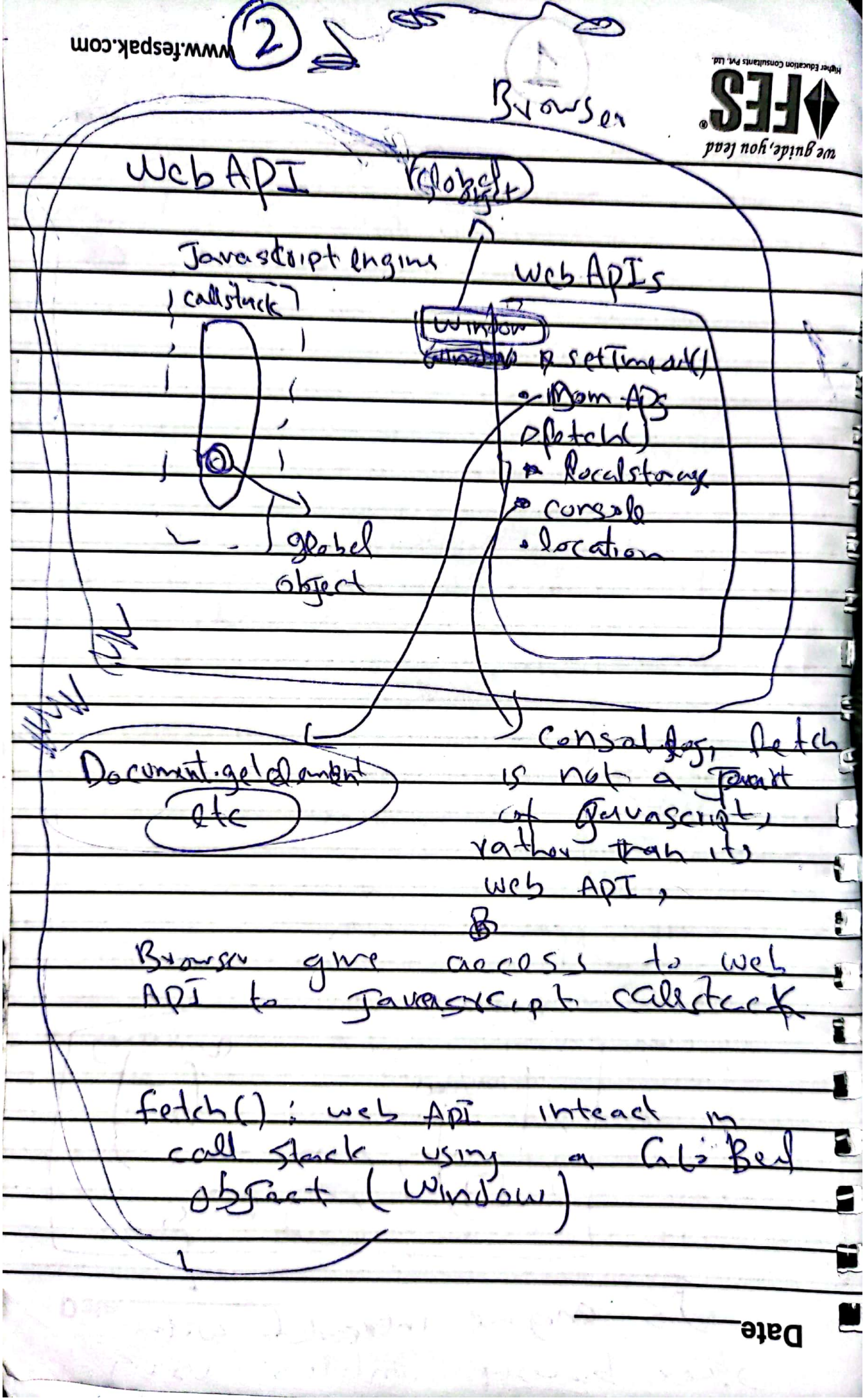


Call stack doesn't have timer.



Date

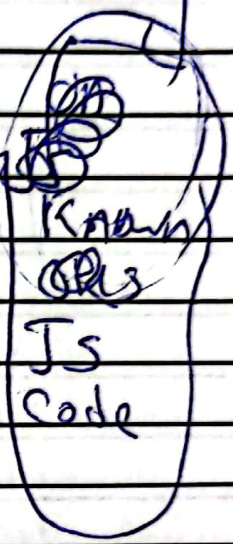
JS engine interacts with other browser entities using



Date

If we want to use setTimeOut inside call stack we can do it by using Global object known as window.like

```
Window.setTimeOut()
Window.fetch()
Window.console.log
```



we use setTimeout it work same as window.setTimeout() because it is a global object

when we put callback in side an setTimeOut function it automatically check if call stack have Global execution context if GEC is not present Call back goes to call back queue and then event loop

Event loop: Event loop Automatically checks for callback in call back Queue if call back is present

Date _____

inside callback Queue it take the call back and push the callback inside of call stack

Event loop continously monitor the callback Queue and callstack

we need a callback Queue because we have to execute the callback in order one by one

setTimeout, DomAPI, console work on Event loop and callbacks Queue mintained above

but fetch() doesn't work like that

used for network calls

fetch() is a which return a Promise which is then Resolved

Date

fetch →
then

Micro Task Queue: Suppose we have a Timeout function and fetch API like

Console.log(A)
setTimeout(cb)
fetch(cb)
Console.log(B)

A
B

What register through web API and web API call Timer

Call stack

Call back Queue

fetch is Register after set Timeout function so in callback As if we got response from fetch faster.

It will not be put here because we got setTimeout Register first, so here

Date

Comes microTask Queue

call stack

microtask Queue

Similar but has higher priority

callback Queue

event loop

So fetch will come inside microtask queue and goes to Call stack context global execution

So suppose we have Already some task in our call stack that take more time then suppose our..

So what we have

call stack

check if empty

microStack Queue

fetch

event loop

then goes to

callback Queue

Settimout

Date

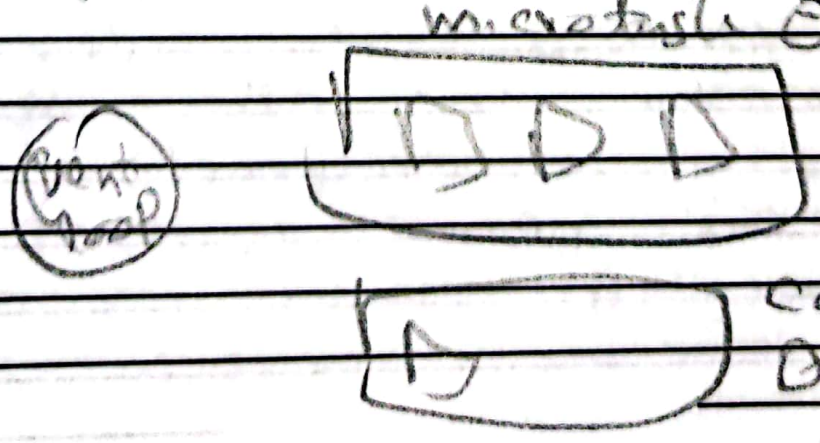
First fetch execute and the setTimeout due micro task Queue

What can come inside micro task Queue;

All the callback function which come from fetch or promise will come inside micro task Queue

and mutation observer;

Check whether there is some mutation in the Dom tree, it can execute some callback (mutation



Date

Callback Event will execute only

After All micro task Queue is Executed

Starvation :

- it a concept when I there is allot of micro task create another micro task so that callback Queue wait too long for it.

Hoisting

You can excess the javascript function and variable even before initializing it or put some value in it.

Java script allocate Memory to all code variable and function Write in the function event before it got executed

If you write a function as Arrow function like

```
var setname = () => {
    console.log('name');
}
```

It will be undefined in memory because it is treated as a variable

If you write like function getname() {

}

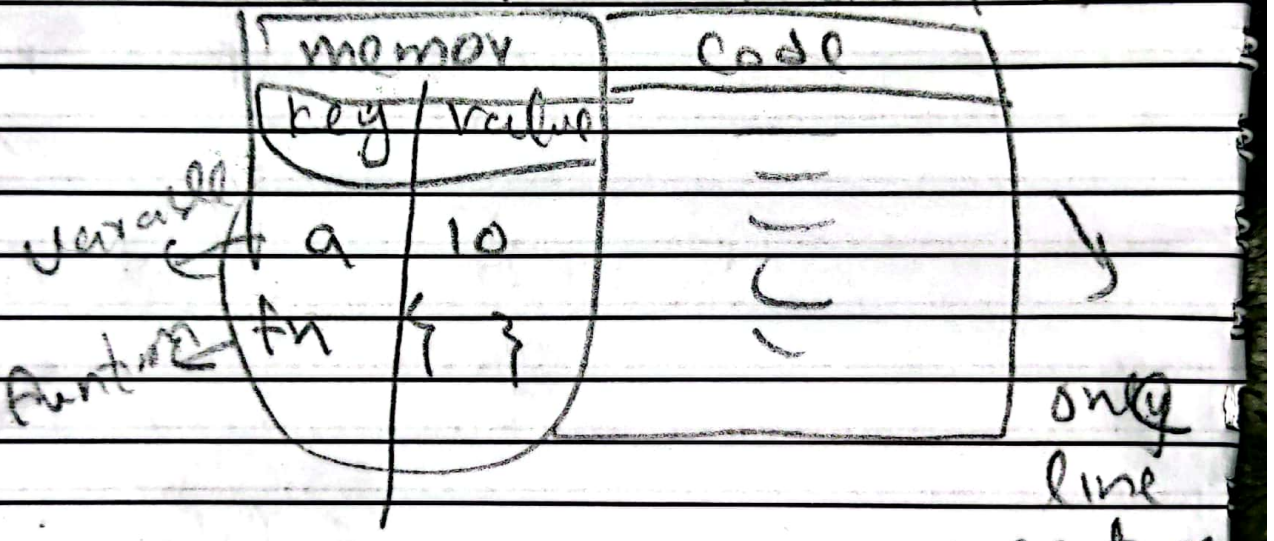
It will be save the function in memory

Everything in Javascript happens inside Execution Context. (10)

Execution context

have two things

- Variable environment / memory
- Code / Thread of execution



only one line as time

Java script is a synchronous single threaded language

one command at a time in a specific order

undefined : Place holder in memory in JS

Date

Scope: Scope is a place where you can access a function or a variable.

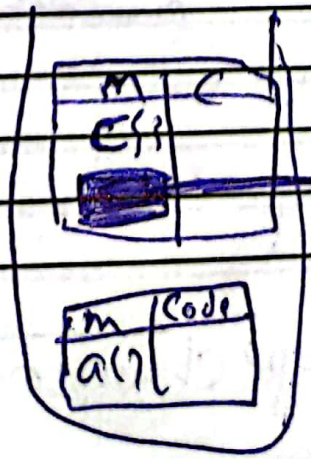
Scope of variable: where I can access a variable

Lexical Environment: lexical environment is the local memory along with the lexical environment of the parent (parent local environment)

Lexical: Hierarchy

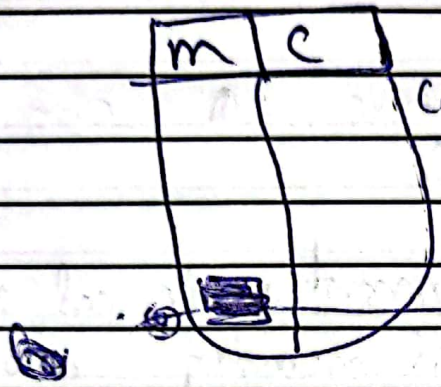
```
fun a(){
  c();
  fun c(){
  }
}
```

C is lexically in a, where C is present the a.



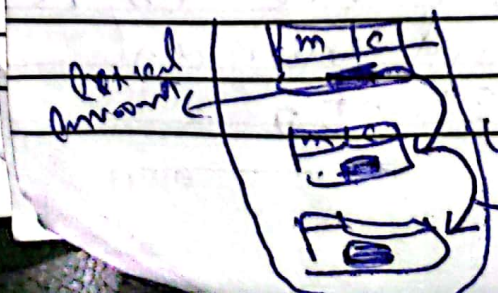
Date

In Case of Global Execution context lexical environment of parent will be null



When execution the code first importance will be given to local memory lexical environment.

Scope chain : It is when we want to search of particular value and we dont find it in local memory and we goes to parent's lexical environment and doesn't find there too and we goest to parent's lexical environment local memory.



Date

(B)

Scope chain: Chain of lexical environment and its parent references.

When ever a execution context is created lexical environment is also created

Let and Const are hosited

because let and const variable are also allocated memory but are stored in a different memory space

In case of var and function

Global

In case of let const

Script

Date

Temporal Deadzone :- The time when Let variable was hoisted and till it is assigned some value. That time is known as temporal deadzone.

Reference error :- When ever you try to access a let, const variable inside a temporal deadzone it get you a reference error.

Let variable cannot be redeclare again

```
let a = 10;
let a = 100; // already declare error
```

const is more strict, it cannot be assigned value again. inside a scope.

Const must be assigned value directly other wise it will throw error

Date

Type Error :- Assigning a value to already used const variable throw the error.

Syntax error :- When const is not initialized, or in case of const

and in case of let cannot redeclare same let variable

Block Scoped

A block is used to combine multple JavaScript statement into a group. it is also known as compound statement.

We Group so Javascript use as one statement

```
{
//
}
```

Date

Date

Let

if(true) true; // Single Statement

if {

true;
a=10;
}

// multiple statement so we use Blocks, so to give Function or Statement

Block scope: what functions and variable we can access inside block

{

// access function and variable will be Block Scope.

}

{

var b = 10;
let b = 20;
const = 30;

}

console.log(a);
// (b);
// (c);

will print b and c throw error due to Block scope

Scope
Block
b : undefined
c : //
Global
a : undefined

let and const are placed in a separate memory space which are reserved for a block

Shadowing

When a variable is shadowed or replaced in a block because of both reference to the same memory

```
var a = 10;
{
var a = 100;
}
Console.log(a) // 100 will be output because global and block var a both Reference to same memory. also known as shadowing.
```

Date

Let and const cannot be shadowed because let and const are block scoped in a block

Illegal shadowing

```
let a = 20;
{
var a = 20;
}
// will throw an error, because let a cannot be convert to var in case of block scope.
```

const and let also have lexical environment where child has access to parent memory

Date

Closure

A closure is a function that is bind towards lexical environment function along with lexical bundal form a closure
closure give you access to an outer function scope from an inner function.

when a function is returned the still remember the lexical scope

```
fun x(){
    var a=7;
    fun y(){
        consol.log(a);
    }
    return y;
}

var z = x();

z(); // will Print 7 because closure still remember a lexical scope
```

Date

```
for (var i=0 ; i<5 ; i++)
{
    setTimeout(function() {
        console.log(i);
    }, i*1000);
}

// it will print
5
5
5
5
5
// because var is a global scope every time i will be refrence to old value.

for (let i=0 ; i<5 ; i++)
{
    setTimeout(function() {
        console.log(i);
    }, i*1000);
}

// it will print
1
2
3
4
5
// because let is block scope it will create new copy of i in every iteration of loop.
```

will run like 1,2,3,4,5 seconds due to timeout

Garbage collector :

It is a program in Java engine which free the unutilize memory when ever you are creating a function.

like unused variable are taken out of memory

Function statement

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

function expression

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

function expression is assigning a function to a variable.

Date

Difference between function expression and function statement is hoisting

function statement is also known as function declaration.

Anonymous function.

A function without a name is known as anonymous function.
It doesn't have there own identity.

It is used where function are used as value. or assign to some variables,

~~Anonymous function~~

```
Var b = function () {
    Console.log("b called");
}
```

Date _____

Name function expression

when a function is given a name and put in to a variable. ↑ that is treated as value

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

we have given xyz to a function (value) and given a variable b also a name.

Parameters → parameters

```
function abc(parm1, parm2){

}
abc(1,2);
```

arguments

Date

A parameters is known as variable which passed to a function call.

Argument is when ↑ a function is called.

Variable passed to

First class functions

the ability to use function as value and can be passed as argument and can be return as function is known as first class functions.

first class citizens is also it First class functions.

ES6 also known as script 2015

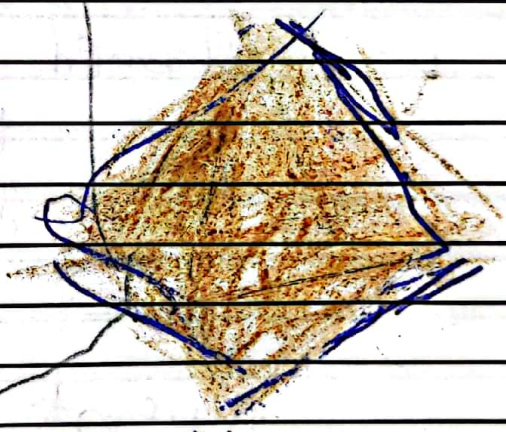
first class function are functors that are treated as object (or assignable to a variable)

Date

Call back function

A function which is passed to Another function is called call back function.

Javascript is a single thread Synchronous but Due to callback We can do async thing in Javascript



```
setTimeout( function() { }, 1000)
```

↓

Callback function

Call stack is also known as main thread, Java Script has one call stack

Date

Block the main thread

Suppose we have a function that take time, Javascript will never goes execution the other code below that function because it has only one single call stack, so actually we are blocking the main thread

we should use async function to not encounter Blocking the main thread

---

Event listeners

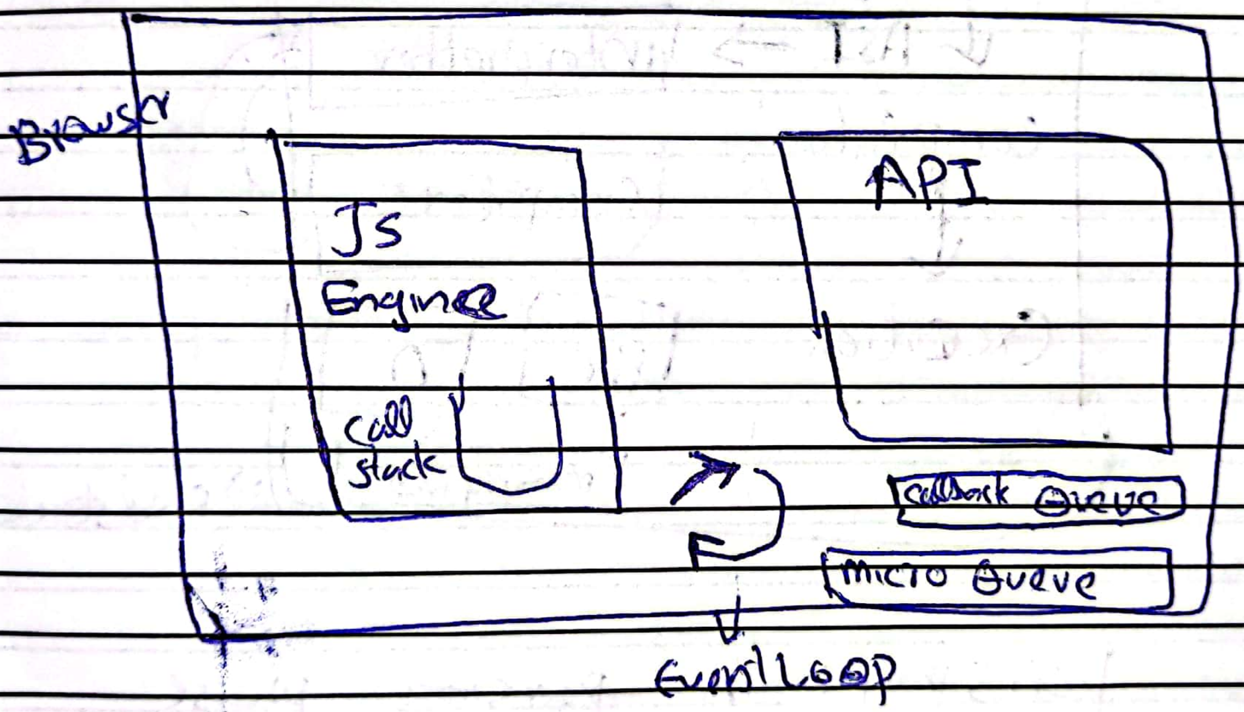
```
document.getElementById("clickme")
.addEventListener("click", function
xyz(){

Console.log("button clicked");
});
```

Garbage collections and Remove Event listeners:

we have to Because it take memory

Javascript runtime environment

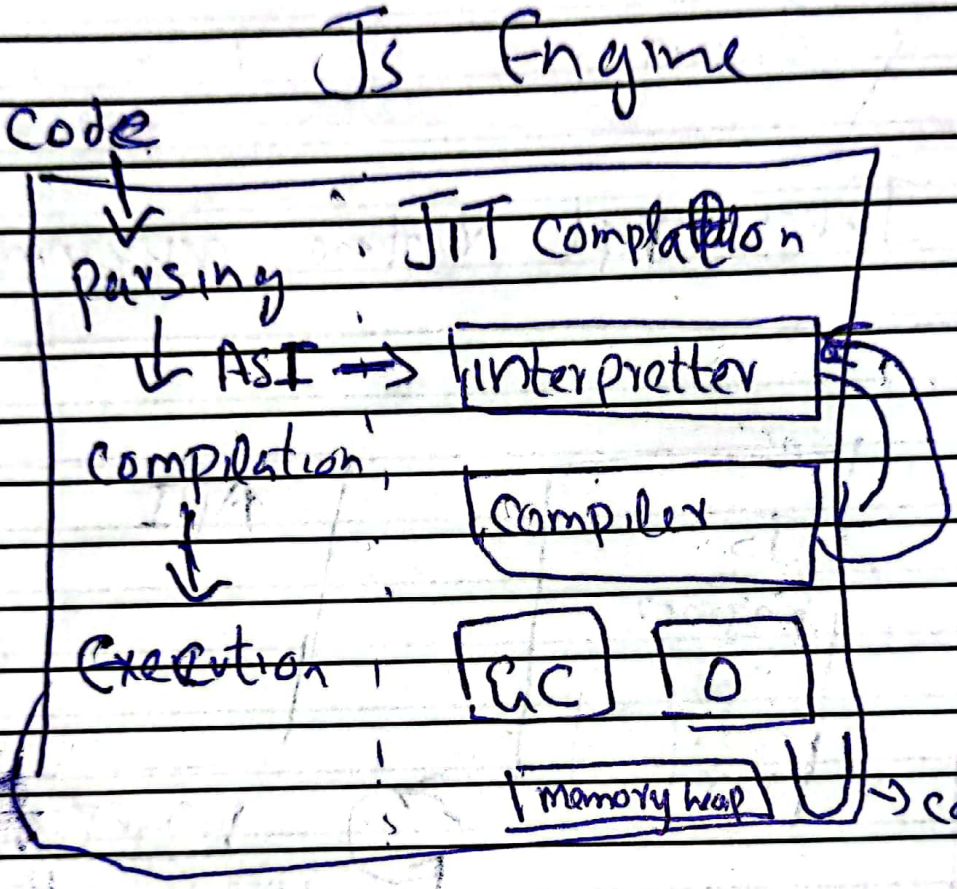


Javascript runtime environment ~~needs~~ consist of container which is need to run Javascript code. like

JsEngine, API, Event Loop, etc

Node Js has als its own javascript runtime environment

Js Engine



Parsing: in parsing phase token are generate like.

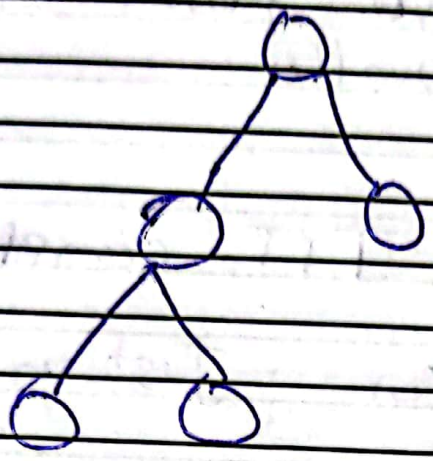
int a = 10

tokens T1 T2 T3 T4

Syntax parser convert the whole code into AST.

Date _____

Abstract syntax Tree



Flowers Js code to Abstract syntax tree through

astexplorer.net

```
Const best = "Name";
{ "type" : "Program",
  "start" : 0,
  "end" : na,
  "body" : {
    {
      type
      ;
      ;
      ;
      ;
    }, "kind" : "const"
  }
  }, "sourceType" : "module"
}
```

Date

Then it goes from AST to interpretter

Javascript is JIT compilation

JIT stand for Just in time compilation

(Fast) Interpretter: takes the code and start executing it line by line. It doesn't know what will happen in the next line

(Efficiency)
Compiler: Takes the code and first compiles it a new code is formed which is the optimized version of the new code. the executed.

Javascipt can behave like both interpretter and compiler. It depends on JS Engine

Date

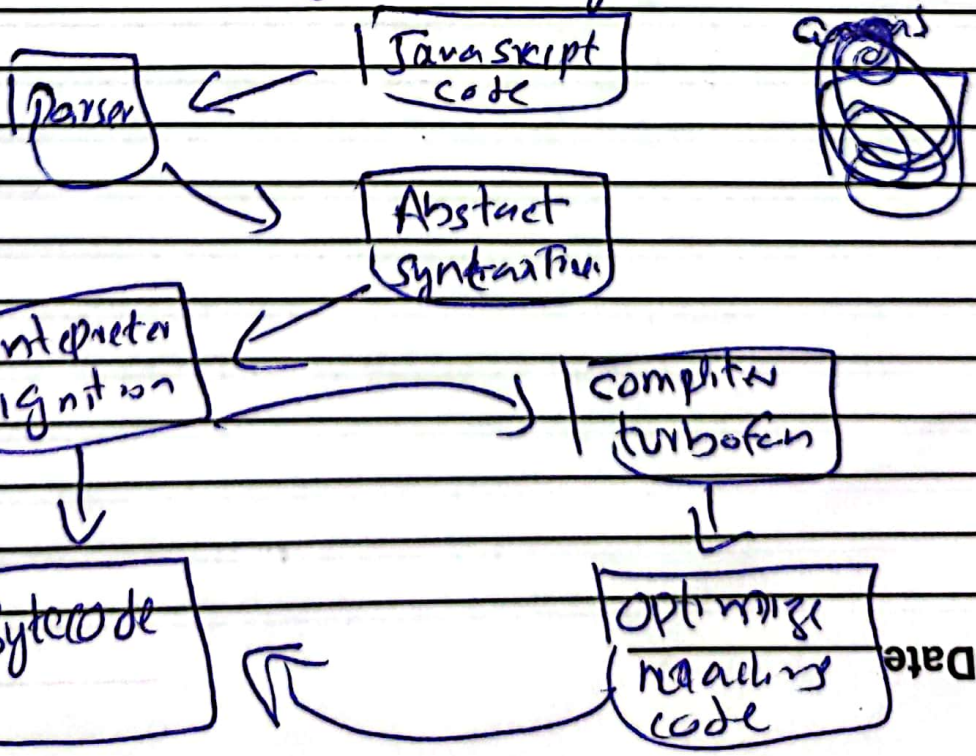
Execution: It used two component memory and callstack

Garbage collection: free up memory which is not used

uses [Mark and Sweep Algorithm]

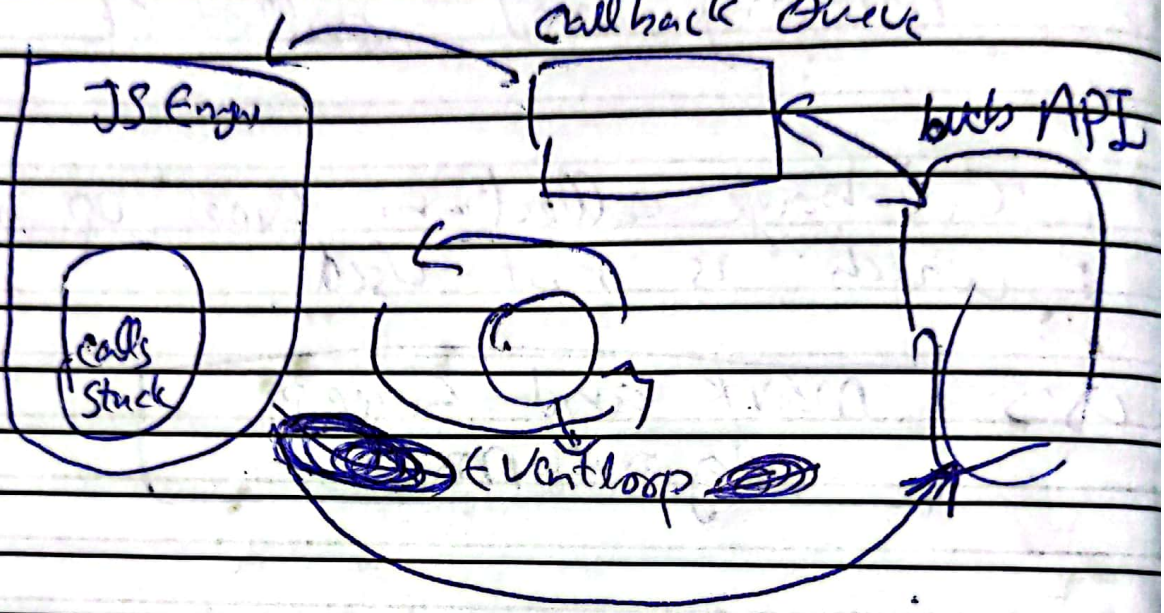
0 Optimization: inline elision copy elision inline caching

V8 JS Engine



Date

Concurrency Model



Vs (Extension (Live server)

Functional Programming

Higher Order Functions:

A higher order function is a function that takes a function as an argument, or returns a function (first call function)

```
funtom x (){

}

funtion y (x){

    return x;

}
```

y is high order funtion

Logic according to function is functional Programming

Date _____

Functional prog

Characteristics

- Reusability
- Modularity
- [redacted] higher order function
- pure function
- composition function

Array.prototype.calculate :

calculate function will be available on all arrays

eg

```
Array.prototype.calculate = function(value){
  const output = [];
  for(i=0; i<this.arr.length; i++)
  {
    output.push(logic(this.arr[i]));
  }
  return output;
}
```

//

console.log(~~vadius.calculate~~

Console.log (radius.calculate(area)) ;

polyfill : Array.prototype
of Calculate

① [ ].map( ) //

② [ ].filter( ) // All are high order functions.

③ [ ].reduce( )

Map : Iterate through array and returns a new array

filter : Iterate through an array an filter array and return new array

reduce : takes an array return only one item

Date _____

~~Reduce function~~

reduce

arr.reduce(function(acc, curren)

{

}

acc // accumulate like sum / total etc

(To store the single value)

current // item in array

If we want to give accumulate a default value, we write

const output = arr.reduce(~~function~~ function(acc, curr) {

}, 0)

we assign value to acc to 0

Chaining: We can Chain filter and map etc together e.g

user.filter((x) => x.age < 30).map

((x) => { x.firstname }) ;