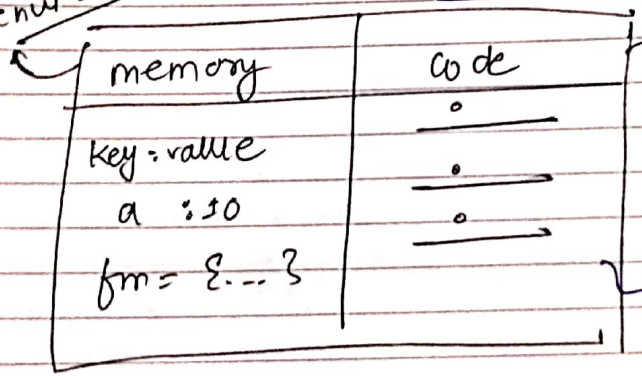
# Everything in JS happens inside an execution context.

↳ considered as big box in which whole JS code is executed.




⇒ JS is (synchronous, single-threaded) language.

(execution of 1 and @ time in a specific order) — execution of one cmd at a time

⇒ When we run a JS program execution context is created.



2023

# How JS code is executed?

Sample code:

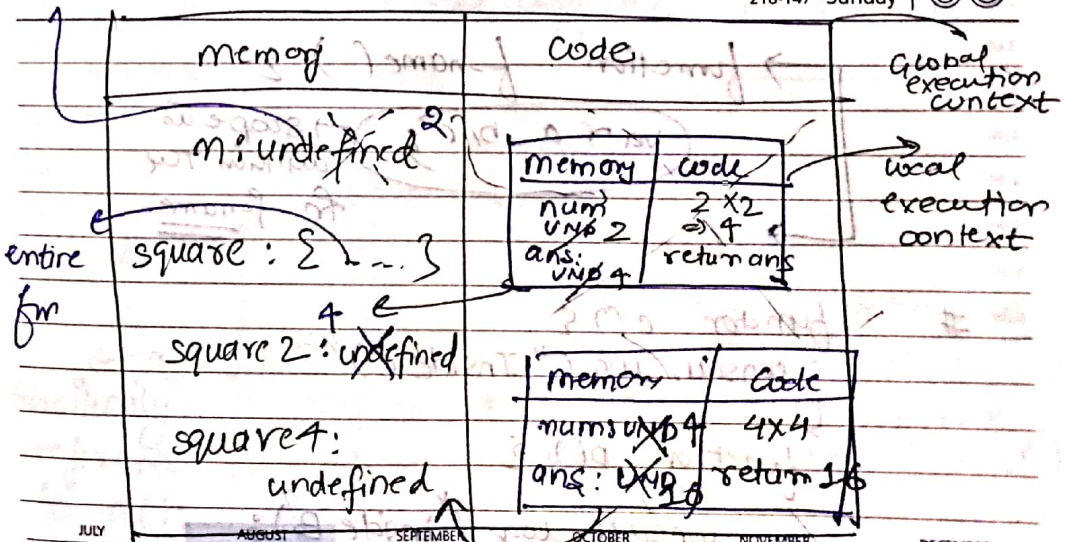
First of all Global execution context is created)

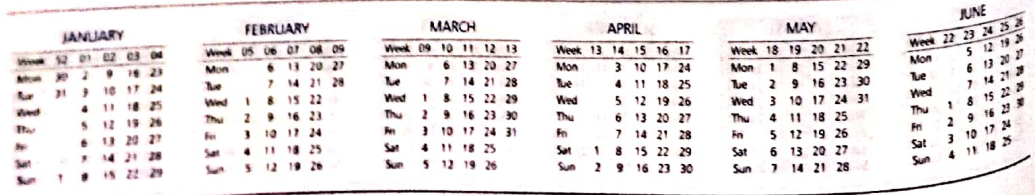
```
var n=2
function square(num){
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

special placeholder

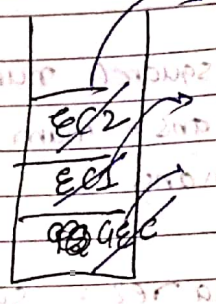
Phase1: memory allocation phase





# Execution stack




" just like call stack in other programming language "

# scope of variable

(within nearby parenthesis)

```
function fname() {
    var a, b, c;
}
```

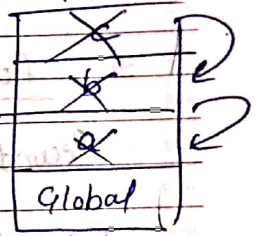
→ scope is within the fn fname

# 

```
function c() {
    consol.log("Inside C")
}
function b() {
    c();
    consol.log("Inside B");
}
```



```
function a() {
    b();
    consol.log("Inside A");
}
a();
consol.log("Inside global");
```

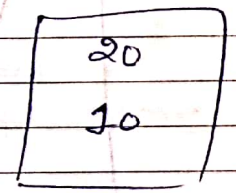



# Global an local variable

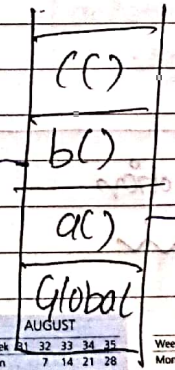
```
var a = 10;
function test() {
    var a = 20;
    consol.log(a);
}
test();
consol.log(a);
```

global variable

local variable




#




Similarly reference to a();

This will all the details of the variables created and have a reference to outer context (Global)

have

JANUARY

| Week | 52 | 01 | 02 | 03 | 04 |
|---|---|---|---|---|---|
| Mon | 30 | 2 | 9 | 16 | 23 |
| Tue | 31 | 3 | 10 | 17 | 24 |
| Wed | | 4 | 11 | 18 | 25 |
| Thu | | 5 | 12 | 19 | 26 |
| Fri | | 6 | 13 | 20 | 27 |
| Sat | | 7 | 14 | 21 | 28 |
| Sun | 1 | 8 | 15 | 22 | 29 |

FEBRUARY

| Week | 05 | 06 | 07 | 08 | 09 |
|---|---|---|---|---|---|
| Mon | | 6 | 13 | 20 | 27 |
| Tue | | 7 | 14 | 21 | 28 |
| Wed | 1 | 8 | 15 | 22 | |
| Thu | 2 | 9 | 16 | 23 | |
| Fri | 3 | 10 | 17 | 24 | |
| Sat | 4 | 11 | 18 | 25 | |
| Sun | 5 | 12 | 19 | 26 | |

MARCH

| Week | 09 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|
| Mon | | 6 | 13 | 20 | 27 |
| Tue | | 7 | 14 | 21 | 28 |
| Wed | 1 | 8 | 15 | 22 | 29 |
| Thu | 2 | 9 | 16 | 23 | 30 |
| Fri | 3 | 10 | 17 | 24 | 31 |
| Sat | 4 | 11 | 18 | 25 | |
| Sun | 5 | 12 | 19 | 26 | |

APRIL

| Week | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|
| Mon | | 3 | 10 | 17 | 24 |
| Tue | | 4 | 11 | 18 | 25 |
| Wed | | 5 | 12 | 19 | 26 |
| Thu | | 6 | 13 | 20 | 27 |
| Fri | | 7 | 14 | 21 | 28 |
| Sat | 1 | 8 | 15 | 22 | 29 |
| Sun | 2 | 9 | 16 | 23 | 30 |

MAY

| Week | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|
| Mon | 1 | 8 | 15 | 22 | 29 |
| Tue | 2 | 9 | 16 | 23 | 30 |
| Wed | 3 | 10 | 17 | 24 | 31 |
| Thu | 4 | 11 | 18 | 25 | |
| Fri | 5 | 12 | 19 | 26 | |
| Sat | 6 | 13 | 20 | 27 | |
| Sun | 7 | 14 | 21 | 28 | |

JUNE

| Week | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|
| Mon | | 5 | 12 | 19 | 26 |
| Tue | | 6 | 13 | 20 | 27 |
| Wed | | 7 | 14 | 21 | 28 |
| Thu | 1 | 8 | 15 | 22 | 29 |
| Fri | 2 | 9 | 16 | 23 | 30 |
| Sat | 3 | 10 | 17 | 24 | |
| Sun | 4 | 11 | 18 | 25 | |

JULY

| Week | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|
| Mon | 31 | 3 | 10 | 17 | 24 |
| Tue | | 4 | 11 | 18 | 25 |
| Wed | | 5 | 12 | 19 | 26 |
| Thu | | 6 | 13 | 20 | 27 |
| Fri | | 7 | 14 | 21 | 28 |
| Sat | 1 | 8 | 15 | 22 | 29 |
| Sun | 2 | 9 | 16 | 23 | 30 |

AUGUST

| Week | 31 | 32 | 33 | 34 | 35 |
|---|---|---|---|---|---|
| Mon | | 7 | 14 | 21 | 28 |
| Tue | 1 | 8 | 15 | 22 | 29 |
| Wed | 2 | 9 | 16 | 23 | 30 |
| Thu | 3 | 10 | 17 | 24 | 31 |
| Fri | 4 | 11 | 18 | 25 | |
| Sat | 5 | 12 | 19 | 26 | |
| Sun | 6 | 13 | 20 | 27 | |

SEPTEMBER

| Week | 35 | 36 | 37 | 38 | 39 |
|---|---|---|---|---|---|
| Mon | | 4 | 11 | 18 | 25 |
| Tue | | 5 | 12 | 19 | 26 |
| Wed | | 6 | 13 | 20 | 27 |
| Thu | | 7 | 14 | 21 | 28 |
| Fri | 1 | 8 | 15 | 22 | 29 |
| Sat | 2 | 9 | 16 | 23 | 30 |
| Sun | 3 | 10 | 17 | 24 | |

OCTOBER

| Week | 39 | 40 | 41 | 42 | 43 |
|---|---|---|---|---|---|
| Mon | 30 | 2 | 9 | 16 | 23 |
| Tue | 31 | 3 | 10 | 17 | 24 |
| Wed | | 4 | 11 | 18 | 25 |
| Thu | | 5 | 12 | 19 | 26 |
| Fri | | 6 | 13 | 20 | 27 |
| Sat | | 7 | 14 | 21 | 28 |
| Sun | 1 | 8 | 15 | 22 | 29 |

NOVEMBER

| Week | 44 | 45 | 46 | 47 | 48 |
|---|---|---|---|---|---|
| Mon | | 6 | 13 | 20 | 27 |
| Tue | | 7 | 14 | 21 | 28 |
| Wed | 1 | 8 | 15 | 22 | 29 |
| Thu | 2 | 9 | 16 | 23 | 30 |
| Fri | 3 | 10 | 17 | 24 | |
| Sat | 4 | 11 | 18 | 25 | |
| Sun | 5 | 12 | 19 | 26 | |

DECEMBER

| Week | 48 | 49 | 50 | 51 | 52 |
|---|---|---|---|---|---|
| Mon | | 4 | 11 | 18 | 25 |
| Tue | | 5 | 12 | 19 | 26 |
| Wed | | 6 | 13 | 20 | 27 |
| Thu | | 7 | 14 | 21 | 28 |
| Fri | 1 | 8 | 15 | 22 | 29 |
| Sat | 2 | 9 | 16 | 23 | 30 |
| Sun | 3 | 10 | 17 | 24 | 31 |

2023

# function expression

Second method of creating the function

Basically we can initialize avariable with some function also.

Ex =

```
function factorial (n){
    var ans = 1;
    for (var i = 1; i<=n; i++)
        {
            ans = ans * i
        }
    return ans;
}
```

Normal function



2023

```
var factorial = function fact (n){
    var ans = 1
    for (var i = 1; i<=n; i++){
        ans *= i
    }
    return ans;
};  (Important)
```

⟹ var a = {function -- }; (either)

# It's our choice to give name to function or not.

But it's advisable to do so.

```
var factorial2 = function (n){
    var ans = 1
    {
    }
};
```

it will basically takes the name of variable factorial (on this case)

Const → value update x (reinitialize x)
↳ initialization while declaration

console.log(c) → Reference Error
const c = 24

(In const variable hoisting takes place in diff manner)

~~var~~ console.log(v) → undefined
var v = 24;

---

# Let: → Block scope

{
console.log(b) ⇒ Reference Error
let b = 24;
}

"we can't use them before they have reached their definition"

"let b; declaration still moves upward (hoisting) But we can't access it"

---

# Let vs var

var a = 12;
a = 24;
a = 9;
var a = "chirag"
console.log(a)

Reinitialize

✓ Let a = 12
Let a = "chirag"
↓
Error

var a = 12
Let a = 12
↳ Error

---

# for (var i = 0; i < 3; i++) { → o/p?
console.log(i);
}

0
1
2

(cond false) ← 3 < 3

# * * * for (var i = 0; i < 3; i++)
{
setTimeout (function() { runs after certain time
console.log(i);
}, 1000);
}

o/p
3
3
3
???

BRAINSTORMING

Web API
setTimeout

once call stack → empty then call back → call stack

Call stack

Call back

① for (var i = 0; ③ →) { ⑤
② setTimeout (function() {
④ console.log(i);
}, 1000);

var i → change → globally

function ke Bahar Access Nhi hai.

var → global scope
let → Block scope

var username = 'chinag';
function login() {
var username = 'john';
}
console.log(username);
↳ (o/p ⇒ chinag)

In function

(Alag Alag variable)

Block scope

```
let a = 10
if (true) {
    let a = 20
    console.log(a);
}
```
→ 20

console.log(a); → 10

o/p

var username = 'chinag'; → no new variable is created

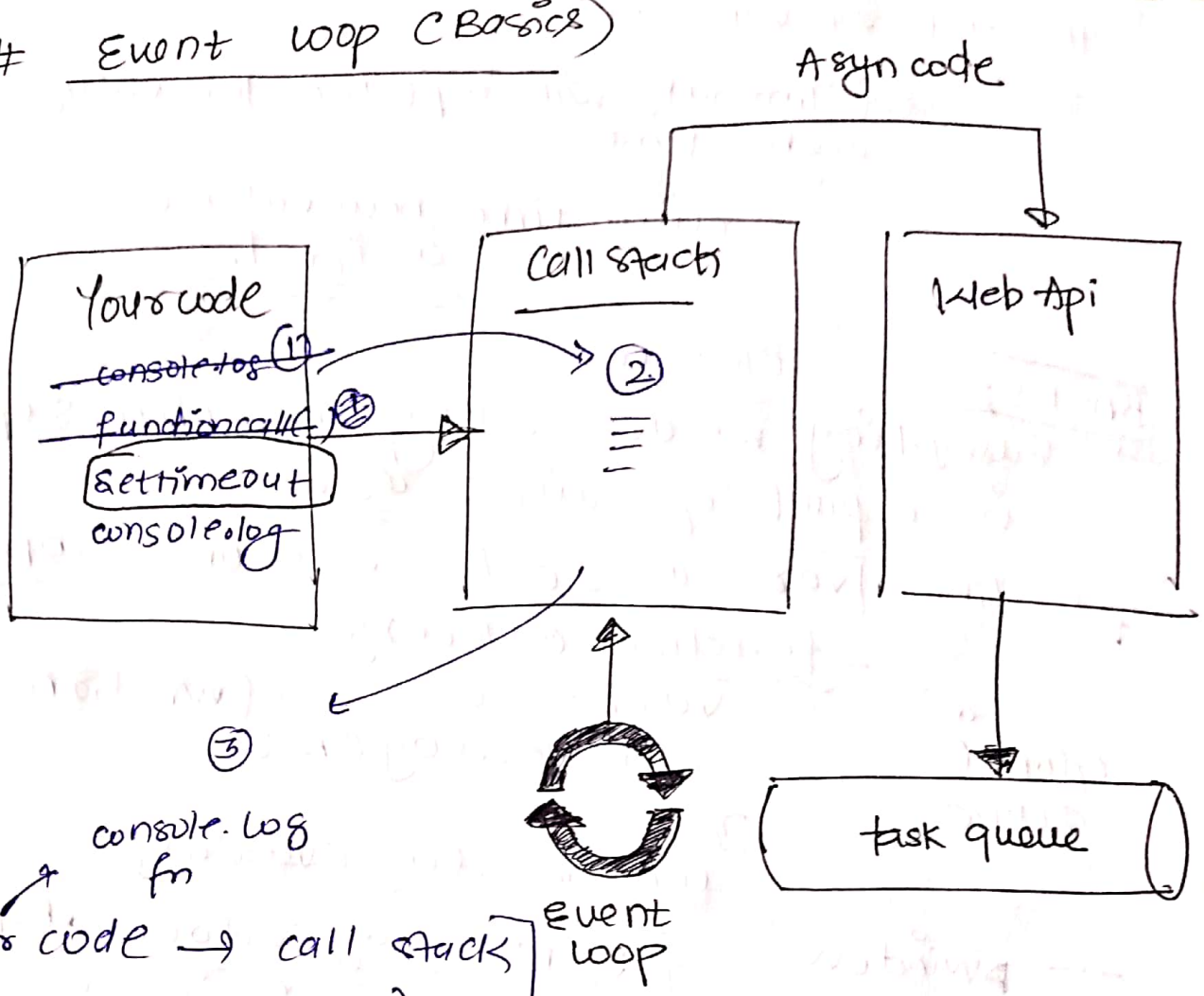
Normal if

(var → function scope)

```
if (true) {
    var username = 'maini';
    console.log(username);
}
```
→ maini

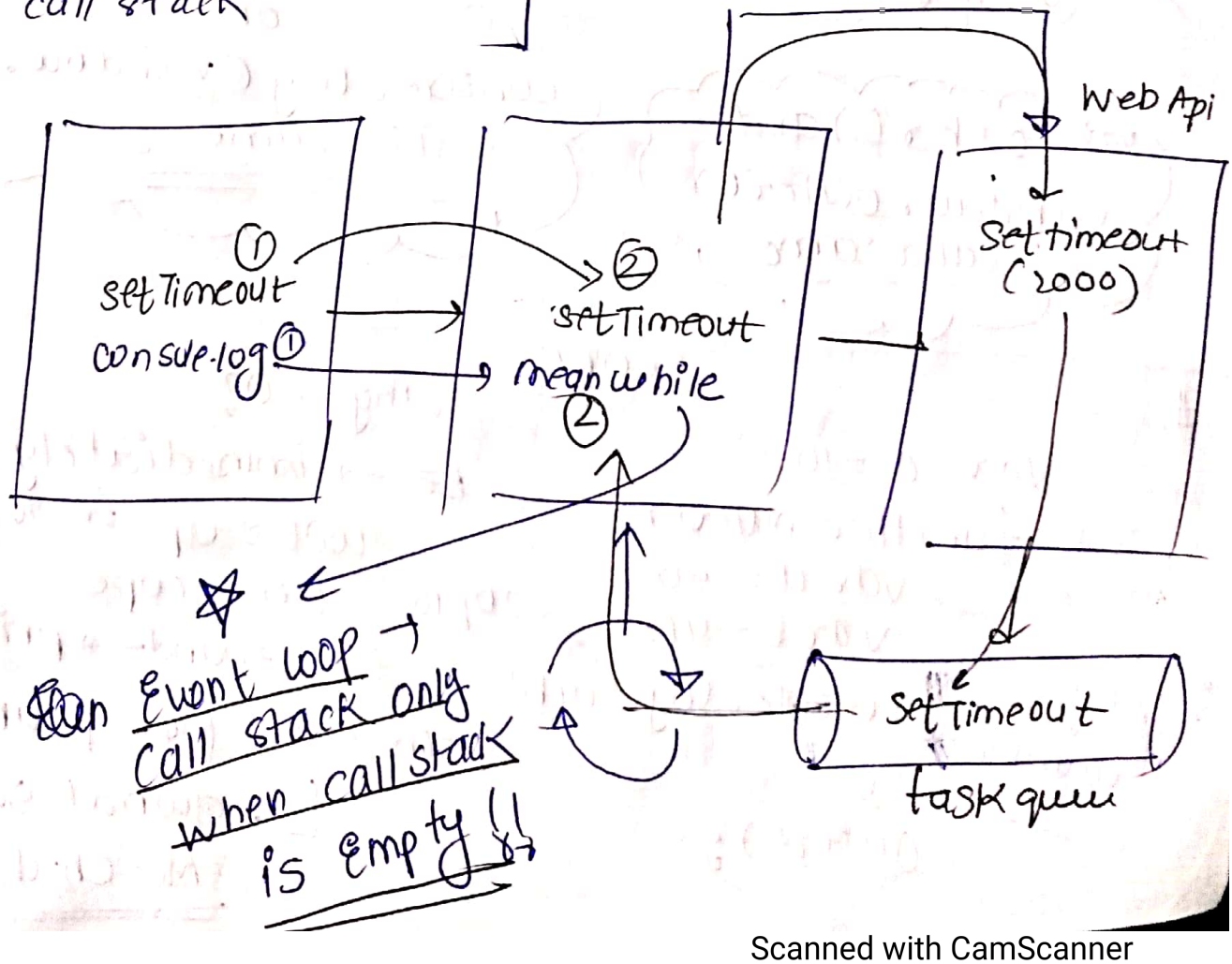
console.log(username);
↳ maini

# Event loop (Basics)




console.log fn

Your code → call stack → out of call stack




Event loop → Call stack only when callstack is empty !!

# Not ~~with~~ with let

# setTimeout will capture the value each time
↓
every time new value is assigned.

---

## BASICS: Closures:

→ Everything we are creating in global scope is a part of window.

→ like `var a = 10` → global scope

```
function outer() {
    var b = 40 → function scope
    console.log(b);
}
console.log(a);
```

(function → Global scope)

→ window ↑ property of window object

var a: 10
outer: f outer()

That's why
console.log(a);
OR
console.log(window.a);
is same.

outer() and window.outer() also same

---

## scopes

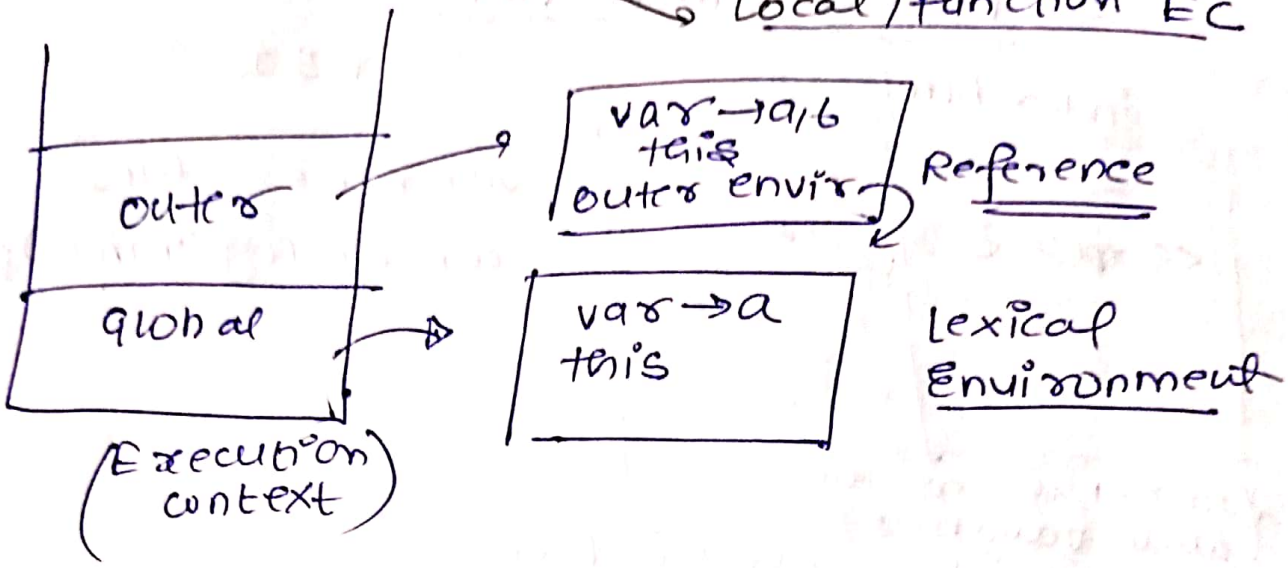
#

```
var a = 10
function outer() {
    var a = 20
    var b = 40
    console.log(a,b);
}
outer();
```

(diff variable) → 20/40

why 20?
→ immediately looks up in the fn scope (if found → print)
In case (not found)
↳ global scope me check krega

Execution context → Global EC
→ Local / function EC




Eg

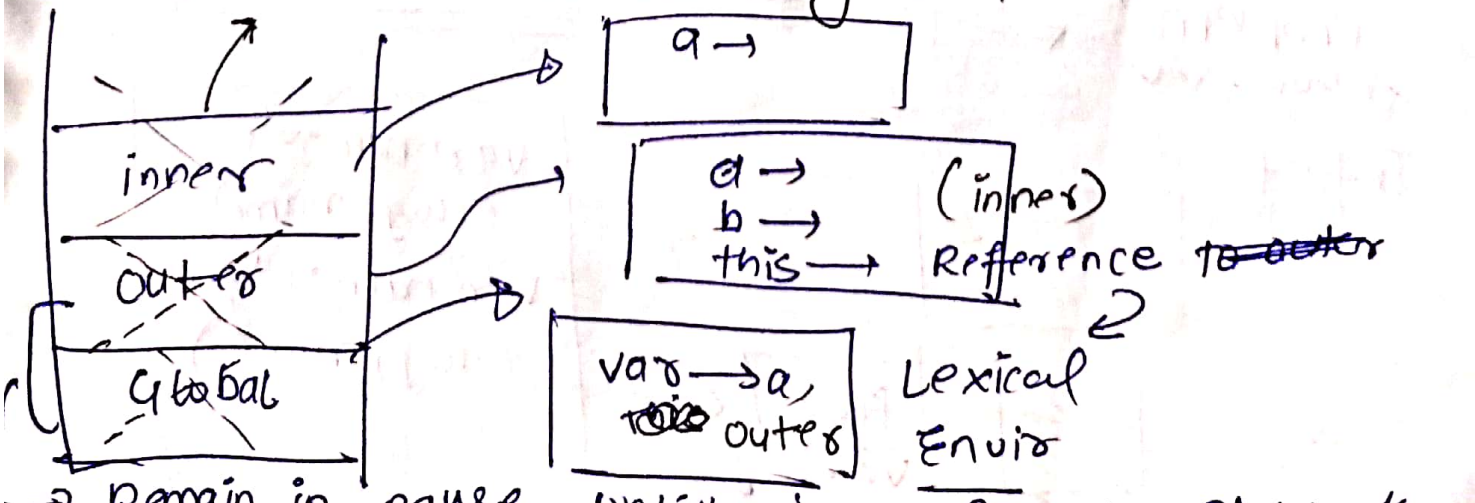
① var a = 10

③ function outer() {
→ var a = 20; — ④
→ var b = 40; — ⑤
function inner() { — ⑦
var a = 100 — ⑧
console.log("Inner", a, b); — ⑨
} — 10
inner(); — ⑥
console.log("outer", a, b); — ⑪
}

② → outer();
console.log("global")

O/P

Inner 100 40
outer 20 40
global

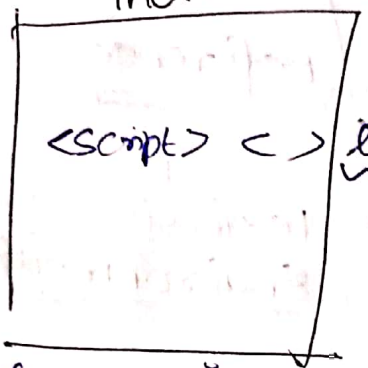



→ Remain in pause until inner finishes it work

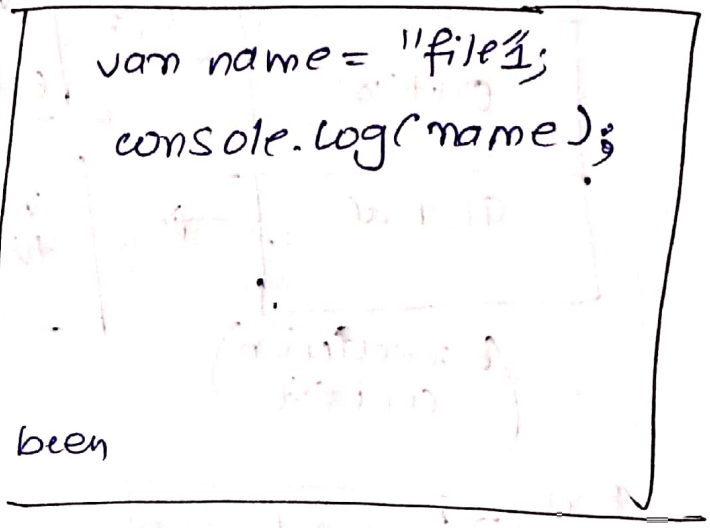
# Avoid global variables

## # Problem with var :-

index.html

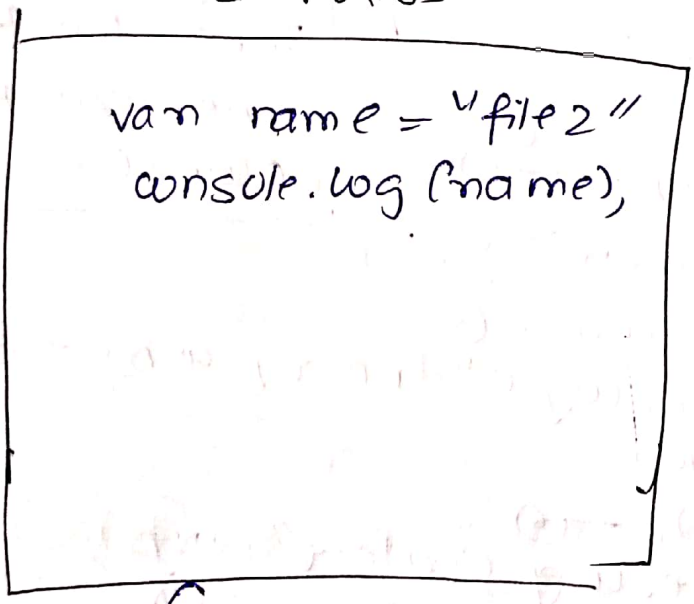



script1.Js




name → is not ~~the~~ (two variables) same variables which is been overwritten.

script2.Js




o/p

file 1
file 2

But Imp thing

▶ name

file 2

⇒ (var → global variable)

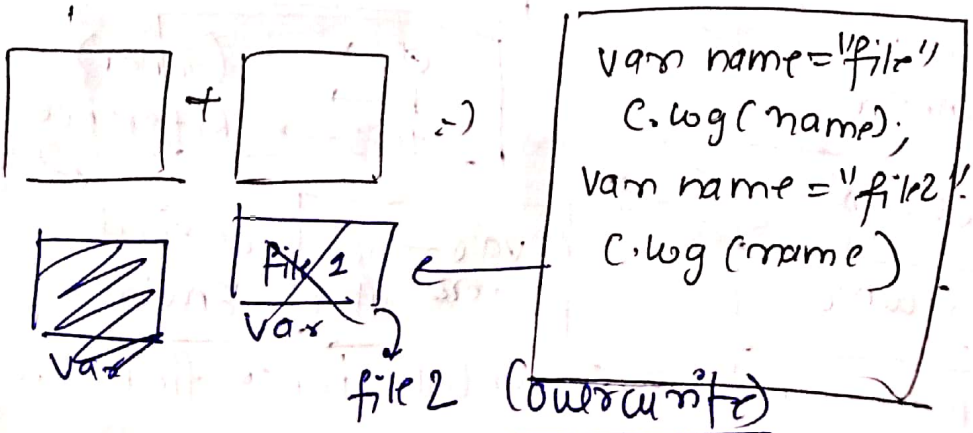
(created)

Ot ~~ve~~ no new variable ~~form~~

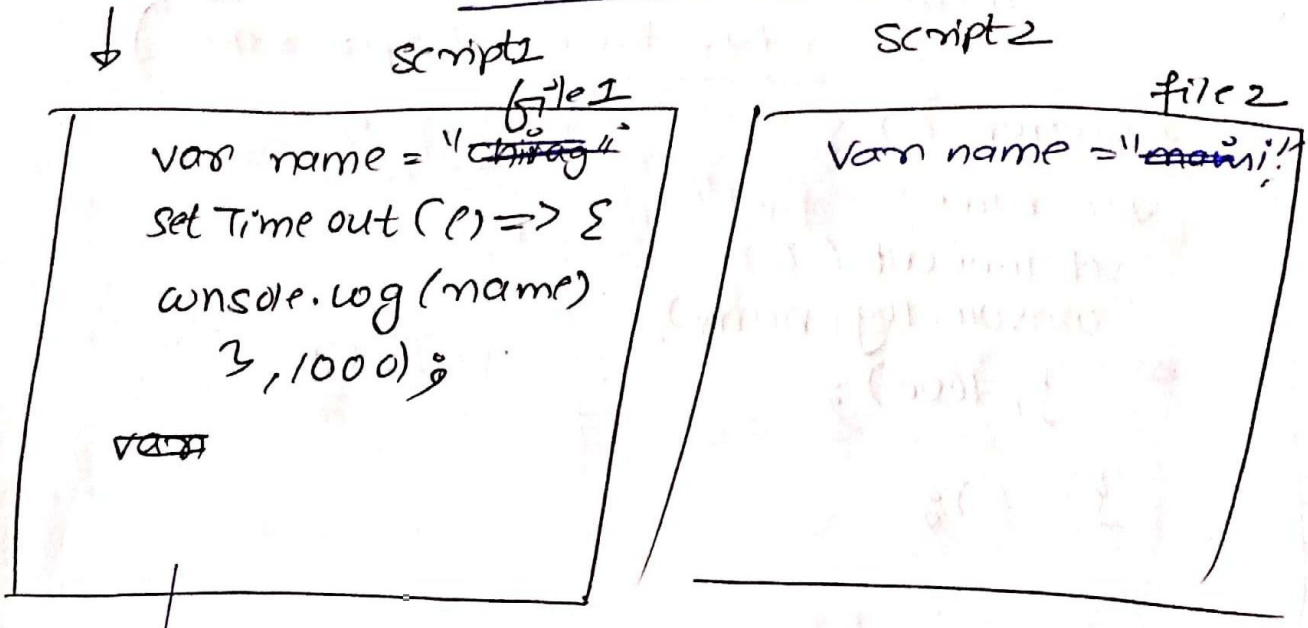
↳ (overwrite)

"Problem"

Infact



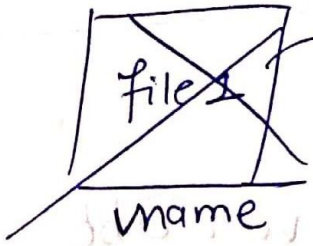
→ • That's why creating global variable is a bad habit.




→ o/p : ~~mam~~ Expectations ⇒ file1
Reality ⇒ file2

→ (var → updated globally)

→ setTimeout → webAPI → ~~call~~ task queue




→ file2 → after that task queue → web API → call stack

o/p ⇒ file2

* To Avoid this we can enclose them in fn :→ Still there may be some chances like (same. fn & name)

# IIFE

~> (Immediate Invoke function Expression)

script1.js

```
(function () {
  var name = "file1"
  setTimeout (() => {
    console.log(name);
  }, 1000);
})();
```

→ IIFE

script2.js

```
(function () {
  var name = "file";
  console.log (name);
})();
```

o/p ⇒

file2
file1

---

# Instead of IIFE we can use let variable

↳ Block scope

```
{
  let name = "file1";
  console.log (name);
}
```

→ within the Block ←

# closures

Imp thing about functions in JS

→ fn's when ✓ returned from another function they're they still maintain their lexical scope. ↳ they remember where they are actually present

→ (closure ⇒ functions + Lexical Environment)
*** 

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

Imp thing

It's not only the function which is returning to z

↓

(It's the function and its lexical environment)

***

z() → you can say this function is called and it will remember the value of "a" (ie of it's lexical scope)

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

Ex 2

```
function sum(a) {
    console.log(a)  ⟶ 2
    var b = 10
    var v = function() {
        console.log(a);
        console.log(b);
    }
    return v;
}
var z = sum(2)
console.log(z) ⟶
```

```
z();
```

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

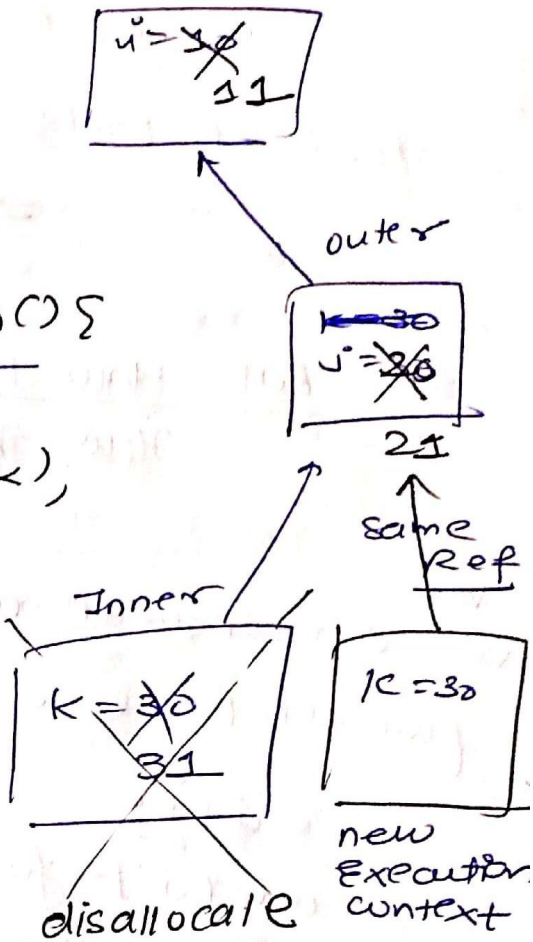
o/p ⟶

| 2 |
|---|
| 10 |

⟹ (Remembers the value of it's lexical environ)

# Mind Bending problems (closures)

```
var i = 10;
function outer(){
    var j = 20
    var inner = function(){
        var k = 30;
        console.log(i,j,k);
        i++
        j++
        k++;
    }
    return inner
}
```




L17 → var inner = outer;

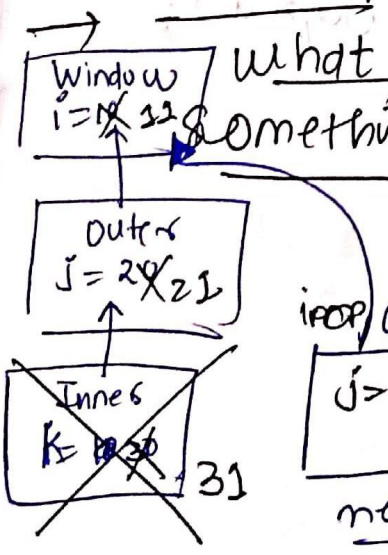
inner(); → o/p 10 20 30

inner(); → o/p 11 21 30

(After L17 new call is made) ↳ (new execution context is made with same reference)

follow up.

what if we had done something like that




~~var inner = outer();~~

inner(); → 10 20 30

inner = outer(); 11 20 30

inner(); ↑

with old Reference

# Let in for loop

## ↳ Block scope

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

⇒ ( i: 1 | i: 2 | i: 3 | i: 4 | i: 5 )

# But How things are working

How boundation of i is changing ??

```
var arr = new Array(6)
for (let i=1; i<=5; i++)
{
  i++;
  arr[i] = function (){
    console.log(i);
  i--;
}
```

converted →

```
var arr = new Array(6)
var loop = function-
        loop(_i){
  _i++;
  arr[_i] = function()
  {
    console.log(_i);
  };
  _i--;
  i = _i;
};
for (var i=1; i<=5;
        i++){
  loop(i);
}
```

☆ in each pass i is passed and a local i created

## Arrow functions :

```
var mult = function (x,y) {
    return x*y;
}
```

↓

```
var mult = (x,y) => x*y;
```

→ for single line we can omit {} and return

or

```
var mult = (x,y) => { return x*y };
```

# Constructors & Prototypes

this → refers to JS context ⊖ object in which the current code is running.

context → global
→ functions

JE

```
console.log(this);
function demo(){
  console.log(this);
}
var v = function demo2(){
  console.log(this);
}
demo();
v();
```

→ referring to the window object

op ▷ window

(These will again point to window object)

# Let's create an object

```
var obj = {
  'prop1': 12,
  'psint': function(){
    console.log(this);
  }
};
obj.psint();
```

Now it will refer to object "obj"

→ (No longer will point to window object)

# This in strict mode

what if :

In sloppy mode

demo is not bounded by any object so it picked window to be as default.

```
'use strict';
function demo(){
    console.log(this);
}
var obj = {
  'prop' = 12,
  'print': function(){
     console.log(this);
  }
}
demo();
obj.print();
```

o/p undefined

: obj

{prop 1 : 12, print : f }

in order to bound it we can use call fn  demo..call (object)

like  demo.call (window);
      demo.call ('obj');

" we can use apply function as well "

Q. Diff b/w call and apply

→ demo.call (object, a, b);

→ demo.~~call~~ apply (object, [a, b]);

aw

```
"use strict"
function bike(){
  console.log(this.name);
}
var obj1 = {
  name = "Pulsar";
  bike = bike ⟶ function bike{
  };
obj1.bike()
```

in short we have to print obj1.name

⇒ "Pulsar" ↳ o/p

this → obj

1

---

OOPs concept

"In JS all that we have is object" ← everything is obj

C++ ↳ (classes → template, object → instances)

⇒ In JS things are diff

→ object

```
var student1 = {
  name : "abc"
  rollNo : 10,
  marks : 80,
};

var student2 = {

}
```

→ (just like classes in C++).

simply create a fn

```
function create(name, rollNo, marks)
{
  var student = {};
  student.name = name;
  student.rollNo = rollNo;
  student.marks = marks;
  return student;
}
```

# this fn (which we've created) is like constructor (in C++/Java)

So function create(name, rollno, marks){
// var student = {};
this.name = name
;
;
// return student;
}
var s1 = new create("abcd", 92, 100); //

new keyword

(this will create an object and fn create will bound to this)

→ If we'll not use new keyword then fn won't be bounded by any object.

this print → window

Conventions: constructors → Capital
Other fn → Camel case

---

# Adding Behaviour to Objects

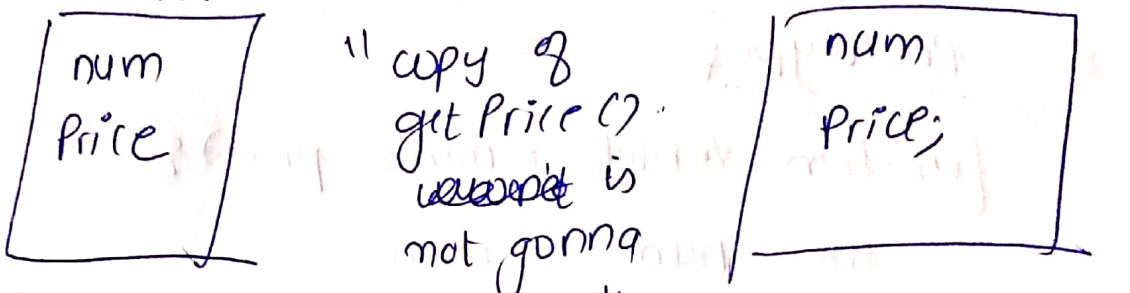
function vehicle(num, price){
this.num = num;
this.price = price;
this.getprice = function(){
return this.price
}
}

we've added a behaviour

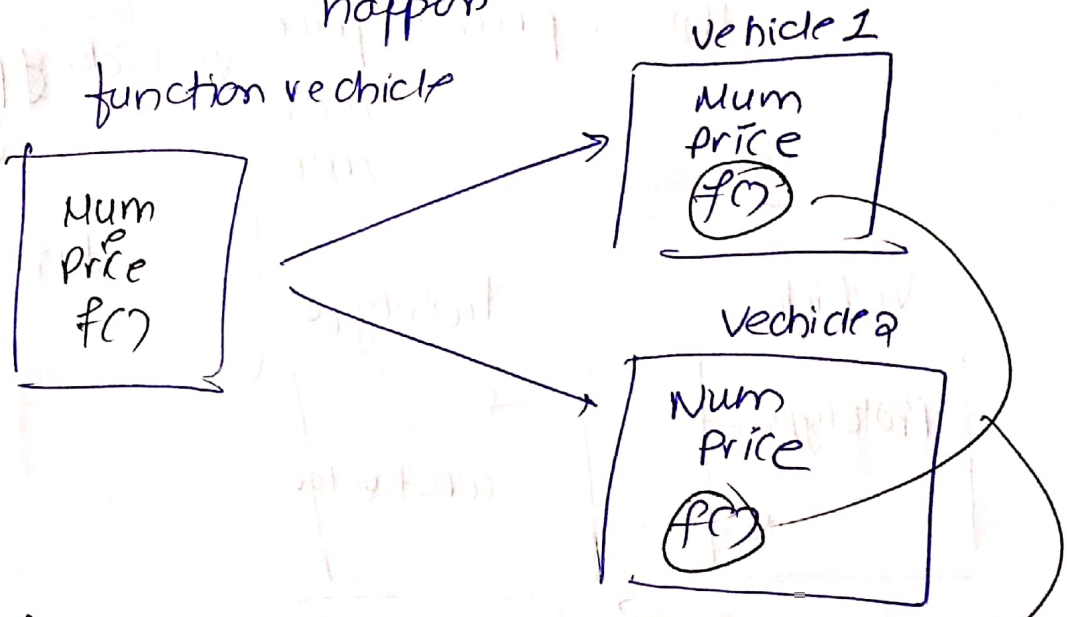
var veh1 = new vechile(20, 20000);
veh1.getpria();

Issues: ~~write~~ by adding behaviour cause,

In C++ class ⇒ template


But in JS function vechicle

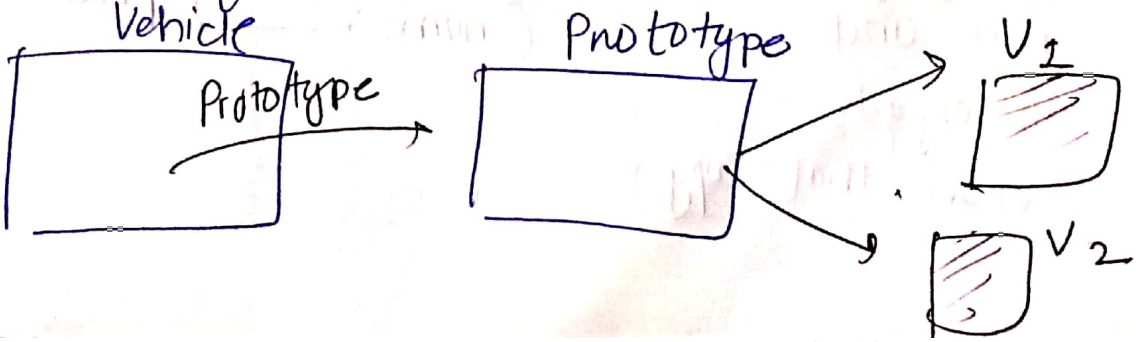



* (consuming extra memory/space)

---

# Prototypes

# foo for every function we're creating JS engine will create objects for the function itself and the prototype



→ It's useful where we're creating objects in constructor mode. multiple

⇒ Vehicle.prototype → to constructor
Vehicle.prototype.constructor
↳ back to constructor

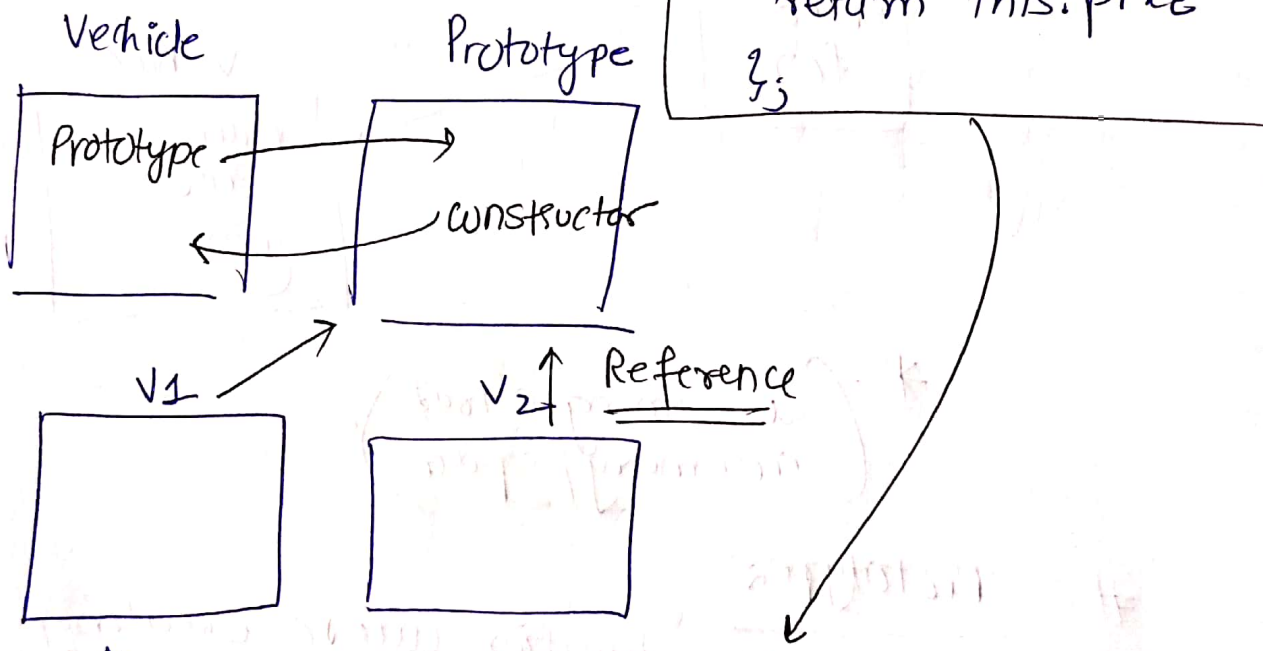
---

## Prototypes

```
function vehicle (nums, price) {
    this.nums = nums;
    this.price = price;
}
```

→ after that

```
Vehicle.prototype.
    getprice =
        function () {
    return this.price
};
```




(Now only one copy of get price and all objects share that copy)

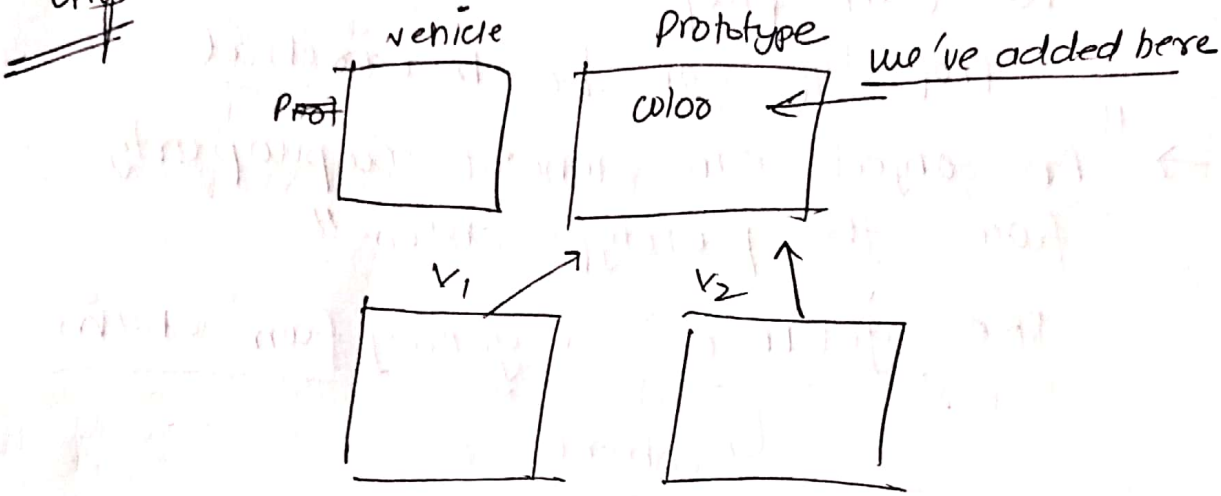
Now if we will do Vehicle1 in console (nums: ___, price: ___)

# We can add a prop too.

Suppose I wanna add a prop to all objects
↳ so rather doing it individually

→ vehicle.prototype.color = "black";

Output:




→ It means we can add properties at the run time also.

---

# more prop. around Prototypes:

→ How to access prototype from objects

⇒ object-name / vehicle.__proto__;

same as ← (we'll get the prototype)

Vehicle.Prototype

__proto__ → dunder

→ Generally this practice is not encouraged

Alternative

Object.getPrototypeOf(vehicle);

→ Check prototype of objects

P3 vehicle.prototype.isPrototype(vehicle1);

↳ True

P4

→ has own prop

Proptotype can be hierarchial

→ "An object can inherit the property from the prototype chain"

like "getPrice" → getting from chain

↳ inherit

v1 (object)

If a property is not present in the object then it looks up in the prototype chain.

# Vehicle.prototype.color = "black"

# Vehicle1.color = "white"

prototype