Page No.
Date / /

JavaScript :- (The Hard Parts, V2)

=> it executes the code line-by-line known as thread of execution

=> data is stored in memory.

☆ function
it is Just Stored in the memory as a definotion, only executed when we write it or invoke when \boxed{()} is used.

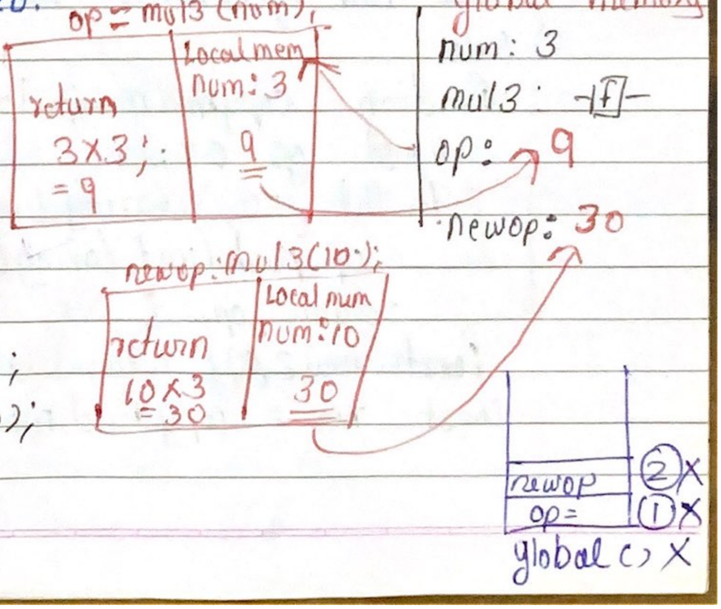
☆ Execution Context :-
- Thread of memory (current line)
- Local memory

☆ Call Stack
- Stack that keep track of what function is currently executing
- Run function -> add to call Stack
- after execution removed from call stack & local memory is relased.

☆ Code :-

```
const num=3;
function mul3(num)
{ return num x3; }

const op = mul3(num);
cont newop= mul3 (10);
```



☆ We have function Parameters & we pass in arguments to the function.
-> That are assigned to the parameters.

☆ Code :-

```
function mul2(array)
{ const op = [];
  for (let i=0; i<array.length; i++)
  {
     op.push(array[i]*2);
  }
  return op;
}

const myarray = [1,2,3];
const result = mul2(myarray);
```

☆ Here we are breaking the DRY Principle (i.e Don't repeat yourself);
if we want to do divide13, add3, then we need to rewrite the function.

We can generalize the function.

=> Code

```
function copyAndmanipulate (array,inst) {
  const op =[];
  for (let i=0; i<array.length; i++){
     op.push(inst (array[i])); }
  return op; }
functio mul2(n){ return n*2}
const res = copyAndmanipulate([1,2,3], mul2);
```

✗ Higher order function :- (copy And manipulate)
the function that takes other function as input ~~&~~ return new function is Higher order function

✱ callback function :- (mul2)
the function are Pass/insert.

✱ Arrow function:-

```
~~const~~ function  mul(num) = {return  num x 2}
const   mul2 = (num) =>{return  num x 2}
const  mul2 = (num) => num x 2
const  mul2 =  num  => num x 2
```

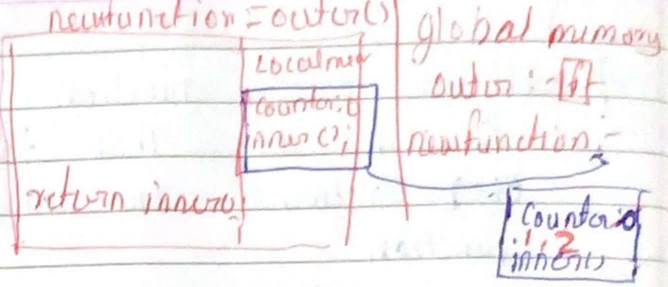
✱ we can also pass in the arrow function as a argument to the Higher Order function (Because the fn is both fn & object)
i.e CopyAndManipulate ([1,2,3], num => num x 2);

✱ Closures:-

=) when a function finishes the execution the Local memory is also relased by the function.
=) if we hold the live data then it is closure

✱ Code :-

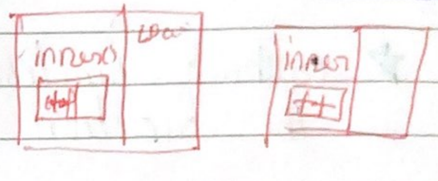
```
function Createfunction () {
  function  mul2(nom) {
    retun num*2;}          → return mul2;
}
const genfunction = Createthefunction ();
cont res = genfunction(2); //4
```



☆ Code :-

```
function  outer(){
    ~~let~~ counter =0;
    function  inner() {
         return  counter++; }
  return  inner:

Const  newfunction  = outer(); ①
     newfunction(); ②
     newfunction(); ③
```



if we run using the another variable then another backpack is created and has it own counter value.

const an = outer
an();
an();

the Backpack is called:-

- C.O.V.E (closed over variable Environment)
- P.L.S.R.D (Persistent Lexical Scope Referenced Data)
- Closure, Backpack

=> this through the Hidden Property [[scope]]
Has access to the data.

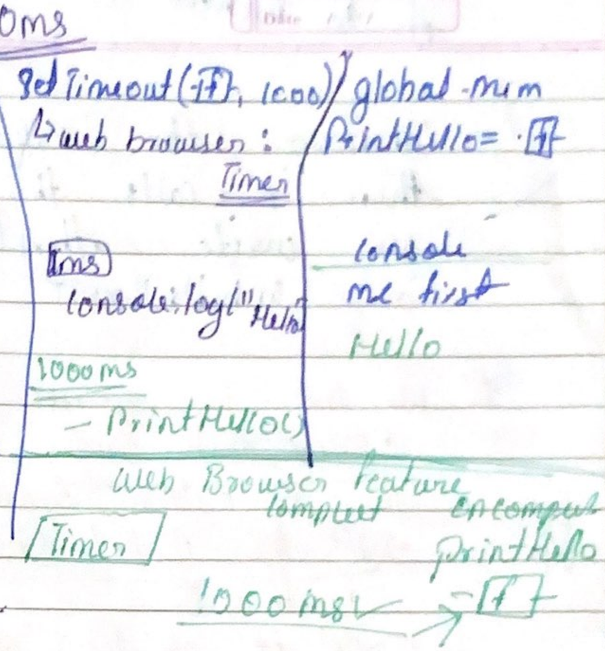
☆ Asynchronous JavaScript :-

```
function sayHello(){
     log("Hello"); }
setTimeout(sayHello, 10);
log("Hi");
```

Order of Execution is:-
(Hi)
(Hello) } why?

JS runs in the web browser so the webbrowser has its own features like
=> Dev/console/tools
=> Dom (Document)
=> Network request (XHR/fetch)
=> Sockets
=> Timer (setTimeout)

```
function printHello {
    log ("Hello"); }
setTimeout (printHello, 1000);
log ("Hello");
     mefist
```



✱ while we are using the Async JS, then they are many factor that affect the code execution.

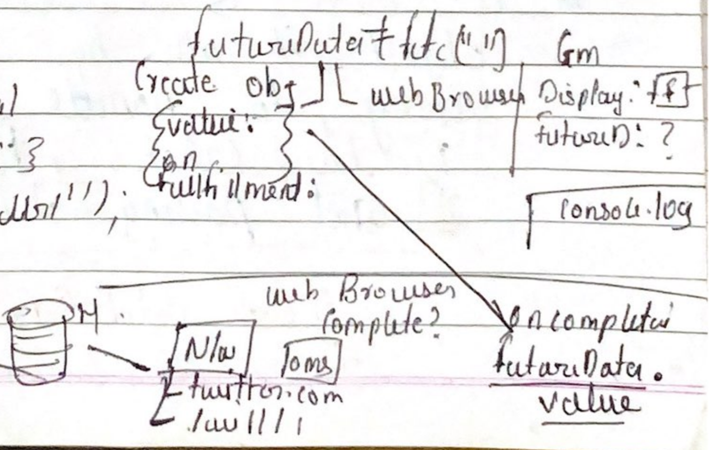
1) Call Stack ⇒ the global() where we keep track of current line
2] Callback queue ⇒ the feature like setTimeout()
3] Event Loop ⇒ the link from stack to cb queue.
4] microtask queue ⇒ we use promise, fetch.

✱ the JS first finish all the code in the global stack & then when global stack is empty then the event loop checks & execute the callback queue.

✱ microtask queue has the high priority than the callback queue.

✱ Promises :-

```
function Display (Data)
{ console.log (Data); }
let FutureD = fetch ("url");
log ("Hello");
```



At the backend it does the .onfullfilled.push(,)
cbfn

Whenever the value property is modified then it calls the callback function that we write .then(cb) this is stored in onfullfillment property what to call

☆ we have to queue,
- micro task Queue
- callback queue

the execution order is that
-> 1st the global stack completes all the global functions
-> 2nd it checks wheather the Microtask queue is empty or not.
-> if micro task queue is empty then it checks the callback queue.

☆ for the things that go on to the callback queue & microtask queue is that whatever does in the browser & returns some function callback like setTimeOut, & the things that connect with the outside info goes (like API) call goes on microtask queue

☆ if there is a error in the promise object it also has an onrejection prop array, to handle it there are 2ways
1) try, catch , futureData.catch()
2] and passing cbfn, as 2nd argument

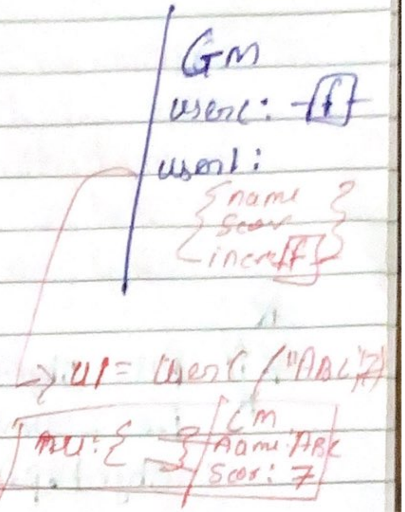
1) Data.catch(-$\boxed{f}$-)
2) Data.then(-$\boxed{f}$-, -$\underset{fn}{\boxed{Err}}$-).

* When We call the function then the (args) pass are stored in Local memory of the function called & maped to the Parameters of the function.

* Class :-

```
function UserCreator(name, score)
{
  const newUser1 = Object.Create(null);
  newUser1.name = name;
  newUser1.score = score;
  newUser1.increment = function() {
            newUser1.score++; }
  return newUser1;
}

   const user1 = UserCreator("ABC", 7);
   Const user2 = UserCreator("LMN", 8);
     user1.increment();
```



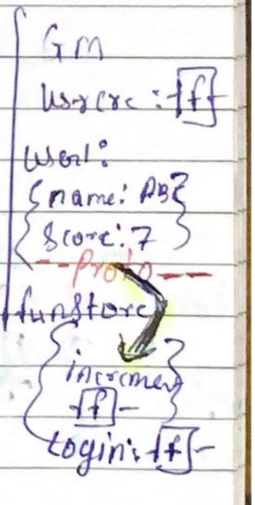
* Here we have increament function stored for every user, so DIY is voilated.

* for this to avoid we will do a Prototype chain that links to the function.

```
function UsrCreator(name, score) {
  Const newUser = Object.Create(funcStore);
   newUser.name = name;
   newUser.score = score;
   Return newUser; }
 ~~function~~ funStore = {
   increment : function() { this.Score++; };
   Login: function() { Log(Loged in) }; }

  Const user1 = UsrCreator("Abc", 7);
        User1.increment();
```

the --proto-- link is created to point if not found in obj



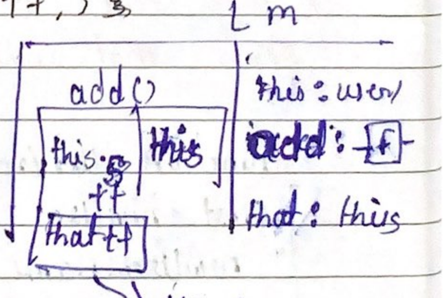
Whenever we run this in a function that is called expliitly then this is assinged implicitly.
that is this: gets Assigned to the object that is calling the function.

★★★★ All objects in JS has a hidden property i.e __proto__ that is linked to the Object.prototype obj. & its __proto__ is null means no chain upwards.

```
Object.prototype { hasOwnProperty
                   : [f]
                   toLocaleString:
                   ----------
                   __proto__ : null }
```

```
Const funStore = {
    increment : function{
        function add() {this.score ++; }
        add() }.
    };

    user1. increment () ;
```



this is a old design pattern to run the code on the obj called, bcz, if we do this++, is points to [gm] which is undefined.

if we want to run a function with new keyword make the function name start letter capital. (to standard form)

★ Solution 2:- Using New ★

the New keyword does some predifined things for us.

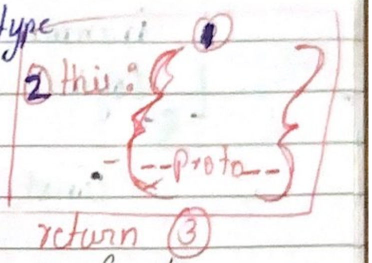
1] make a this keyword.
2] make an empty object.
3] return that object.

Note: the proto has the link to obj prototype

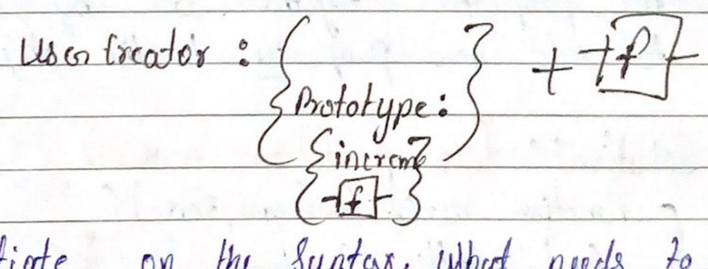
```
function UsrCreator (name, score) {
    this.name = name;
    this.score = score;
}

function UsrCreator.prototype.increment = function ()
  { this.Score++; }

const user1 = new UsrCreator("ABC", 7);
```



★★ function in JS are both function & Objects. so when we store a property on function using (.) it is stored in the prototype property of the object part of function



★ it differentiate on the Syntax, what needs to be called
if () => function call, if . => object property.

✱ When we return from fun: Ex we return the Value outside,

✱ Solution: Class Syntatic sugar.

```
Class UserCreator {
  Constructor (name, Score) {
    this.name = name;
    this.Score = Score; }

  increment () { this.Score ++; }
  Login () { log (" logedin"; }
};

  Const User1= new UserCreator ("ABC", 7);
        User1.increment();
```

```
funct UsrCr(n,s)
 this.name=n;
 thisScore=S;
```

```
userCreator.Prototype
 .increment
 = function() { }
```