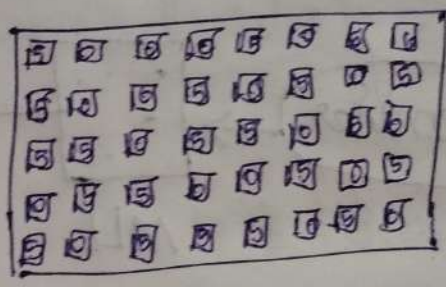
Computer Basic understanding

Hardware Components
* Microprocessor [Brain - CPU]
* Hard Disk
* RAM
* GPU

CPU made of semiconductors where it uses millions of transistors

Transistors - NPN, PNP transitors
↳ It stores only the voltage either
5V or 0V
ON or OFF



Later derived the term Binary/Machine level language 0's & 1's => 0 -> 0V/OFF
1 -> 5V/ON

Program:
Collection of Instructions

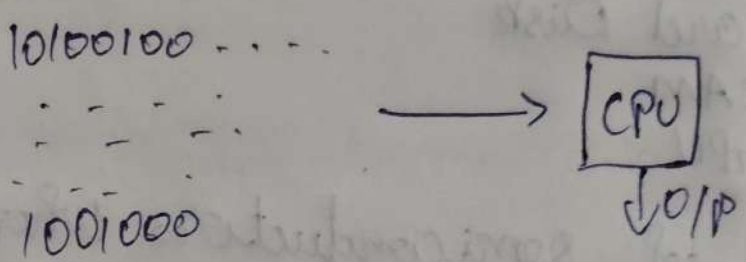
Programming/Coding:
Process of creating program in Hi-fi words is programming.

Stages:

L> MLL (Machine Level Language)

=> written in Binary lang

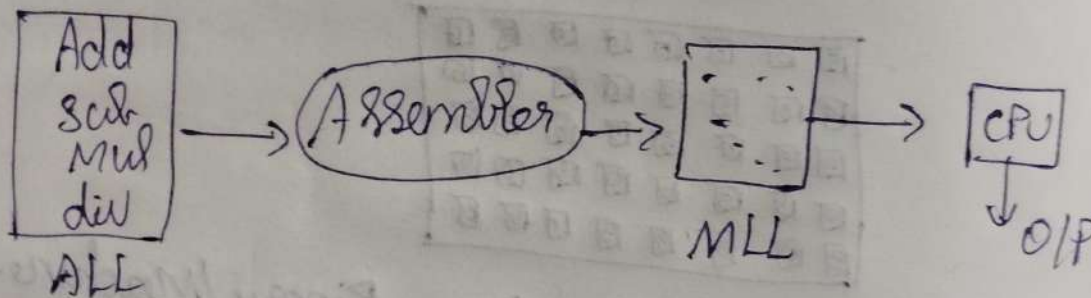
=> for addition, multiple lines needed



L> Assembly Level Language (ALL)

=> Crave the instruction like Add, sub, Mul, div.

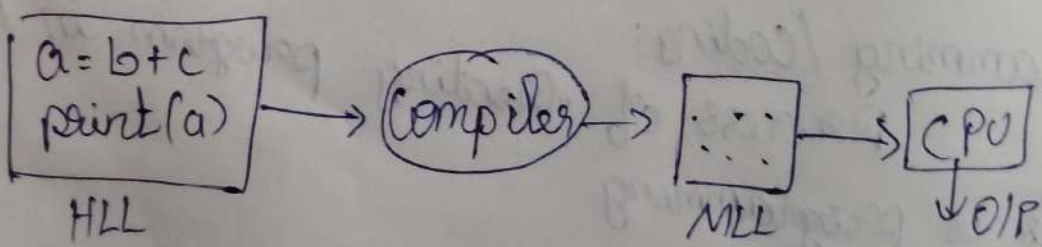
-> Assemblers, a Software converts the ALL to MLL.



L> High-Level Language (HLL)

-> Where the coding exists in different languages.

-> Compilers, a Software converts the HLL to MLL.



Types to convert HLL to MLL:

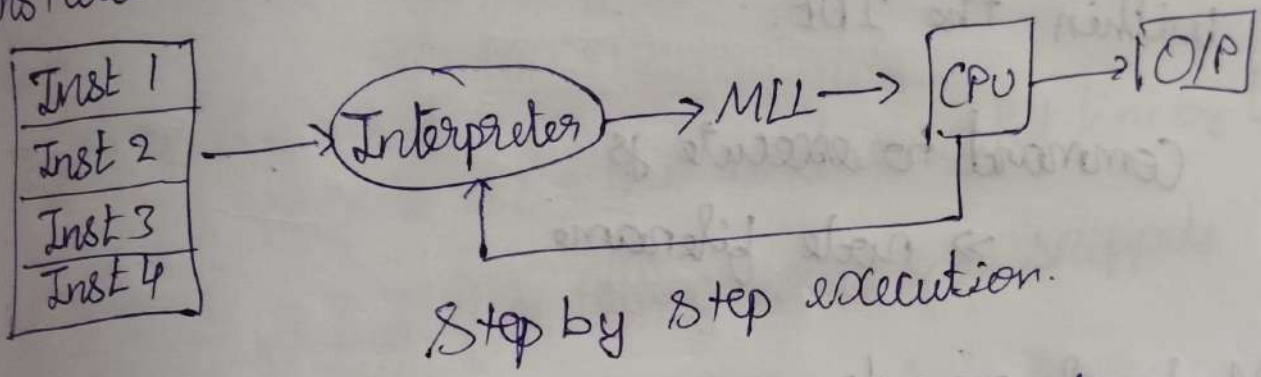
1. Compilation: [Compiler]

In one shot it will take the program (all high level instruction) and all the taken instructions are converted into MLL.

Once MLL ready in one shot it will give it to MP(Microprocessor)

2. Interpretation: [Interpretor]

It will never take all the instructions in one shot.



Step by step execution.

Compiler
⊗ Interpretor is very faster in execution
└> is slow in execution

JavaScript:

It is an interpreted language

Every browser have their own engine to execute the js.

For example, Google chrome have -> V8

So they took the V8 engine and modify them to run outside the browsers also

This is where called as | node.js | ✓

After installing it we can execute the js within the IDE.

Command to execute js

=> node filename.

Mode of execution:

1. Script mode

   L> where save file and command to run => node file_name.

   ex: Script.js

   ```
   Console.log("Hello")
   Console.log(2+2)
   ```

   O/P:

   ```
   Hello
   4
   ```

2. Interactive mode.

   Where line by line execution with the node.

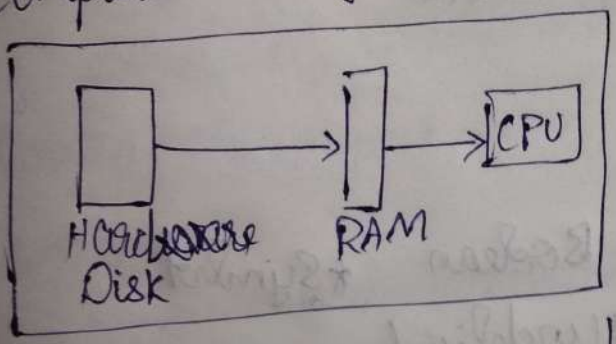
   Command => Node.

Ex: => node
=> console.log("Hello")
O/P: Hello
=> console.log(2+2)
O/P => : 4

Script Mode:
=> Code is written in a file and saved before execution
=> used for writing and running larger programmes.
=> Requires saving the file before execution
=> O/P is shown after the entire script runs.

Interactive Mode:
=> Code is written and executed line by line directly.
=> used for testing small code snippets quickly.
=> No need to ~~sa~~ save code runs immediately.
=> O/P is shown immediately after each line.

Data Types:
Computer mainly consists



where CPU has no memory where it uses the RAM and stores the data of the Running process

HardDisk is connected to RAM
RAM is connected to CPU.

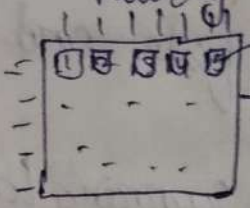
RAM: [A collection of Bytes

For, example: 2 GiB

2 → Giga → Bytes
10^{9}

i.e 2×1000000000 Bytes

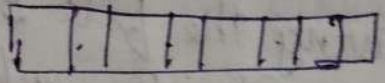
nearly 200 Crore Bytes.



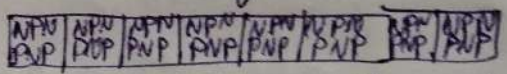
Hierarchy

RAM
↓
Bytes
↓
Bits
↓
Transistors

1-Byte = 8 bits

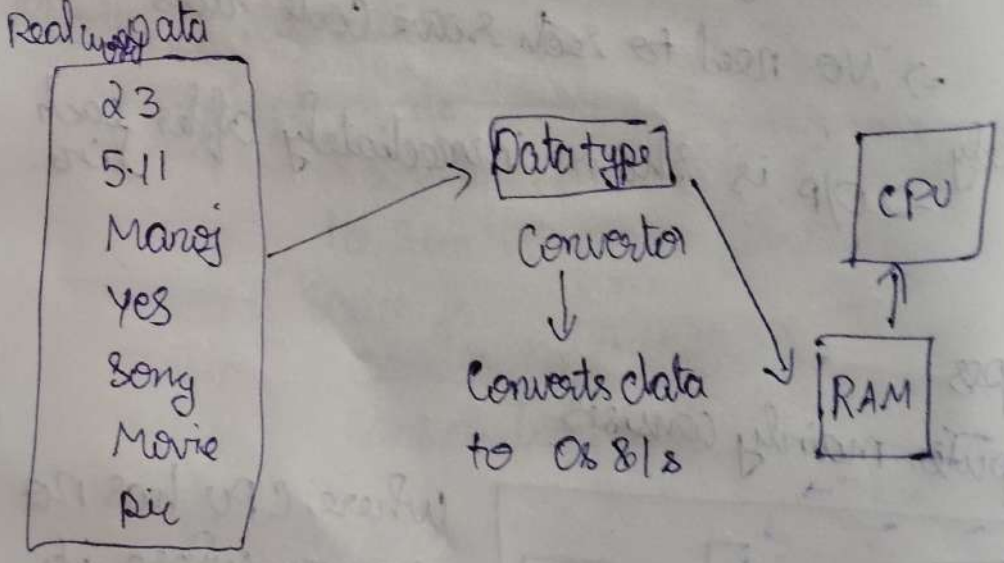
ie  → 1 Byte

Bit Consists of two transistors



Where RAM is nothing but collection of transistors.

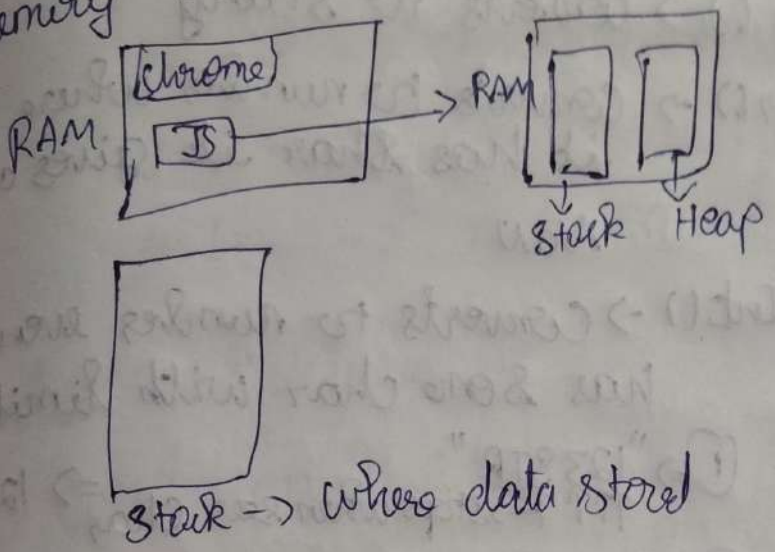
RAM is only 0s & 1s



Data Types in JS:

* Number
* BigInt
* String
* Boolean
* undefined
* Null
* Symbol

Whenever we run the js, it takes memory in RAM, where it divides into two partitions



Stack -> where data stored

In, Js, a variable can store any types of data as it's [loosely Typed powerprogramming lang]

```
let a=12;
a=12.05;
```

1] Number:
Range: $-2^{53}-1$ to $2^{53}+1$
Types: Real, Integer, infinity, NAN

2] BigInt:
↳ To store beyond the range
Ex: let a = 900719925474099 1n;
↳ BigInt only used for integers.

3] String:
Series of characters

4] Boolean: True or False

5] undefined:
↳ only declaration

Type Casting / Conversion:

=> String() -> converts to string

=> Number() -> converts to number where if it has char it gives as NaN

=> ParseInt() -> converts to number even if it has some char with limits

① => "123STR" => 123 O/P
↑↑↑↑→ stops when sees char

② => "STR123" => NAN
↑→ stops as it is NAN

③ => " 1233TR" => 123
space ↑↑↑↑→ stops
consider next will be number if there were no numbers before

④ => "-1-2-3" => -1
↑↑↑→ stops
consider after 1 seen

=> Boolean

True	False
1	0
123	NAN
-123	null
-∞	" "
+∞	[]
"a"	
"123"	

Comparison operators

1. == → checks for value
2. === → checks for type & value
3. !== → not equal types
4. != → not equal

Control structures in JS:

1. Conditional statements

* if () {
  }
* if-else
  if() { }
  else { }

* Multiple if else
  if() { }
  else if() { }
  else if() { }
  else { }

* Switch (condition variable)

```
switch(a){
  case 1:
    break;
  case 2:
    break;
  default:
}
```

use break to stop the code from falling through

* Ternary:
(cond) ? exp1 : exp2.

2. Loops:

Functions:

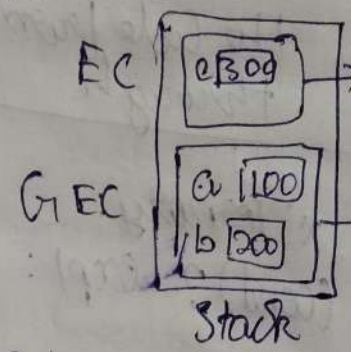
* Iterating over elements modifying array contents
* Accessing DOM nodes and updating DOM elements
* Processing user input Transforming data sets
* Creating Animations, Controlling Animation Frames.
* Handling click events and Responding to keyboard input

Functions: [DRY Principle] → Do not Repeat Yourself

```
function func_name(parameters)
{
    - + - -> Body
    Return value;
}
```

usually four kind
1. No input, No output
2. No input, return output
3. Get input, no output
4. Get input, return output

When a js program is executed a js code environment is created on RAM and follows the process




```
let a=100;
let b = 200;
function add() {
    let c;
    c = a+b;
}
add();
```

① Whatever is on the top of the stack is currently executing

② After execution the EC will be erased automatically

③ When the program finish the GEC and all memory are erased

Types of function
Function
1. Declaration
2. Function Expression
3. Arrow function
4. IIFE (Immediate Invoked Function Expression)
5. Generator Function

FD	FE	AF
function b(){	let fe = function(){	let af = (x,y)=>{
}	}	}
b()	fe()	af(a,b)

IIFE:

```
(function(){

})();
```

Hoisting:
Moving to the top -> declarations.

Example 1:

```
console.log(a)
let a=100;
console.log(a)
```

O/P => ReferenceError

example2:

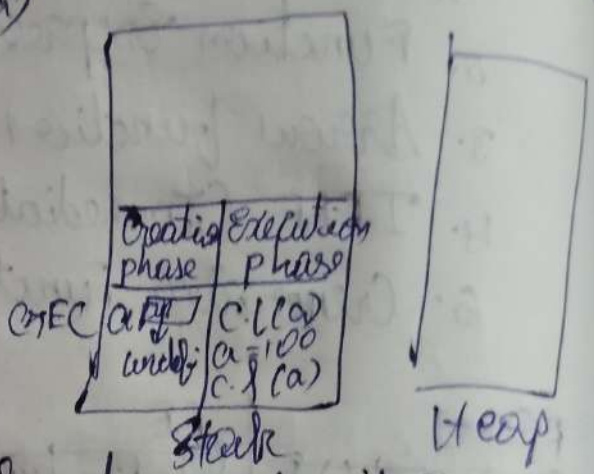
```
console.log(a)
var a=100;
console.log(a)
```

O/P => undefined
100

①
```
c.l(a)
var a = 100;
c.l(a)
```

②
```
c.l(a)
var a;
a = 100;
c.l(a)
```
Splits the declaration & Initialization

③
```
var a;
c.l(a);
a = 100;
c.l(a);
```
Move to the top



But in let → The undefined is not initialized to the variable by hoisting.

④ TDZ: Temporal Dead Zone

```
let a;
con.log(a);
a = 100;
c.l(a)
```
→ TDZ we can't use only them in the zone
From the declaration to initialization

var	let
* When hoisted, it is automically initialized with undefined. * There is no TDZ	* When hoisted, it is not initialised with undefined. * There is a TDZ.

In other programming
```
fun()
def fun():
```
↓
Throws error

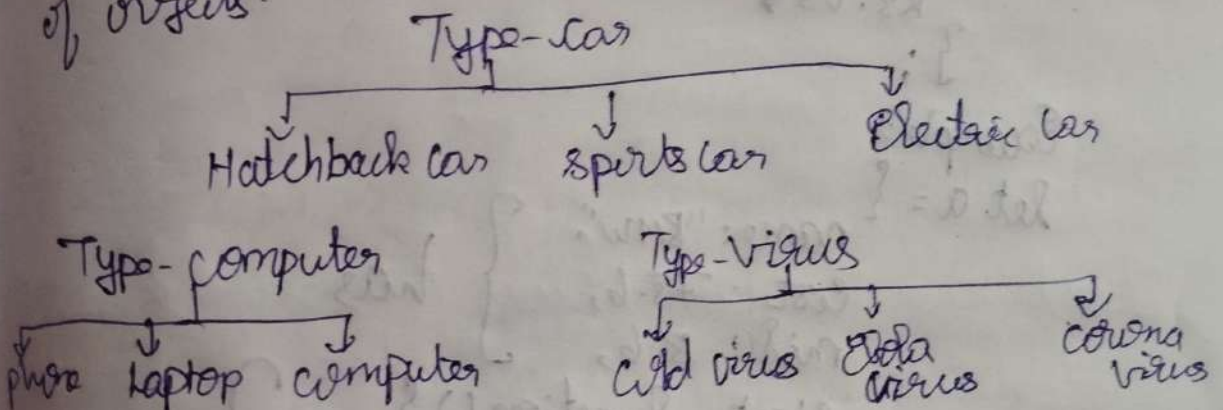
But in JS,
```
fun()
function fun(){
}
```
↓
Executes because of hoisting as it moves to the top declaration.

Scopes Types:

1. Global Scope:
   => Declared outside
   => Risk of overwriting
2. Function Scope
   => Declared Inside
   => Local Accessibility
3. Block Scope
   => Introduced with Let/const
   => Block-Level Accessibility

Object Oriented Programming:

OO -> The world is a collection of objects. Perspective that everything is a collection of objects.



Type is nothing but imagination, we can't give them

i.e technically

Type = Class

Real care Objects.

Objects:

- State / properties : Has
- Behaviour = Does

Example:

Type/Class = Car

Object = Sports car

- State/prop (Has) → mileage, name, color, cost,
- Behaviours (Does) → start, stop, accelerate, ...

Object Creation Methods:

1. Object literal Notation
2. New object constructor
3. Constructor function
4. Classes (ES6+)

1. Object literal Notation [key-value pairs]

```
{
  key: value,
  k2: v2,
  k3: v3,
};
```

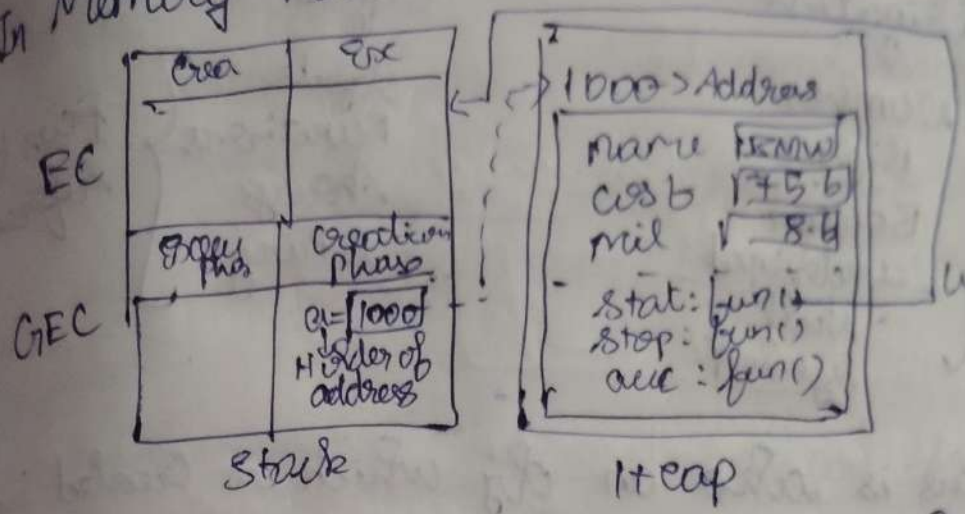
Example:

```
let a = {
    name: "BMW",
    cost: 75.6,
    mil: 8.6,
    start: function() {
        c.l("started");
    },
    stop: function() {
        c.l("stopped");
    },
    acc: function() {
        c.l("Running");
    }
}
```

name, cost, mil → has

start, stop, acc → class

In Memory POV:



Heap comes into play when only objects comes into play.

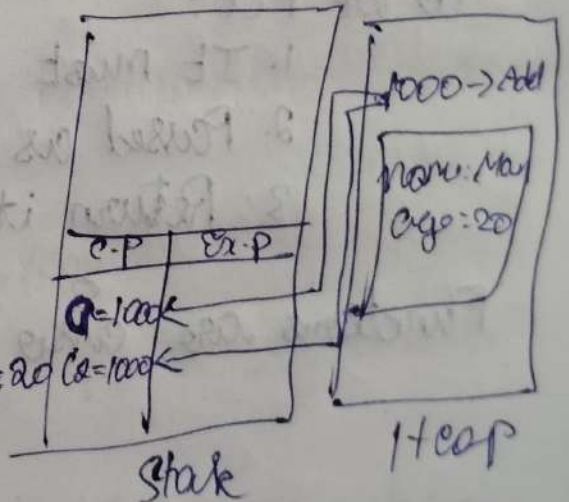
Heap is a memory segment designed to create and manage objects.

Pass by value:

```
let a = 100,     // Doesnt affect original
let b = a;
b = 200;
c.l(a) => 100
c.l(b) => 200
```

pass by Reference:
For objects // affects original

```
let c1 = {
    name: "Manoj",
    age: 20};

let c2 = c1;
c2.name = "Changed";
c.l(c1) => name: 'Changed', age: 20
c.l(c2) =>
```



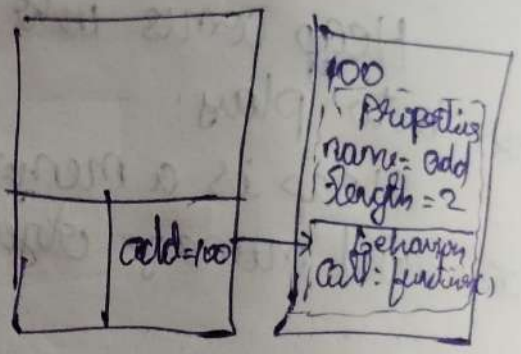
Primitive
↓
Pass by value { Number, BigInt, Boolean, undefined, null, Symbol

Objects
↓
Objects, Functions, Arrays, String } Pass by reference

Functions is also an obj which is created in the memory.

F.E:

```
function add(a,b){
    c.l(a+b);
}
```



properties:
name = add
length = 2 # parameter

Behaviour
call: function()
apply: function()
bind: function()

Objects are the first-Class Citizen in js.

To be FCC:

1. It must be stored
2. Passed as an argument
3. Return it.

Functions are also FCC:

Closures:

When a func returns another function closures comes to play.

It stores the variables which are required for the inner function even the outer func execution completed

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

→ Lexical scope → where the inner function needs

→ It will be stored

Callback functions:

A function which is passed to another function and be called later

```
function fun1(){
  ...
}
```

callback function

```
function fun2(fun1){
  ...
}
```

→ Higher order function which takes function as an parameter

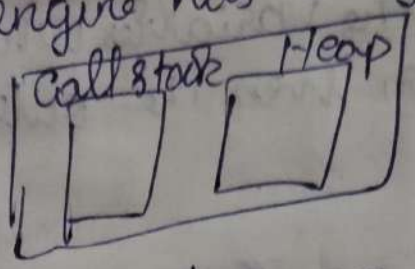
Synchronous JS:

JS is synchronous and are single threaded.

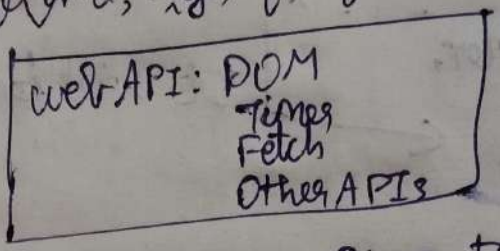
This is the line by line execution which block another's execution

Asynchronous

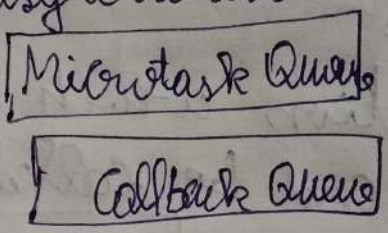
JS engine has only



To achieve asynchronous, browsers gives us features to overcome the synchronous way.
chrome, edge, firefox

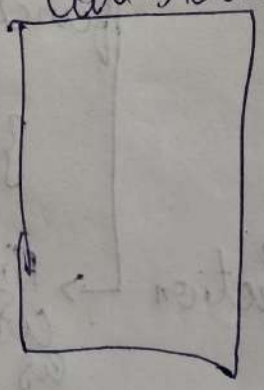


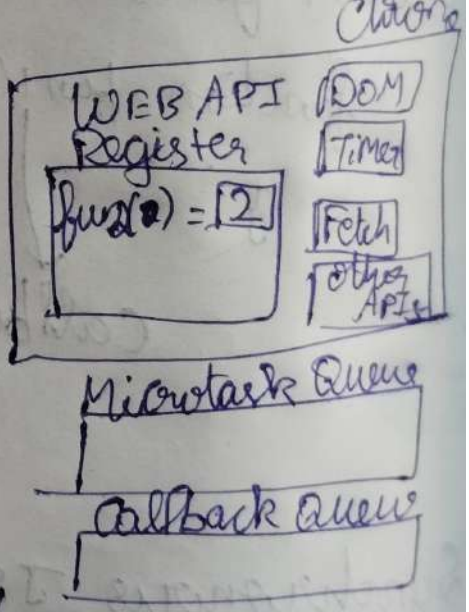
Also browsers give two more data structure to achieve asynchronous



example:

```
function fun1(){
}
function fun2(){
} //delay func
function fun3(){
}
fun1();
setTimeout(fun2,2000);
fun3();
```





In order to execute the code asynchronously JS uses a concept called event loop which runs silently executing in background

Set timeout doesn't belong to javascript, it belongs to the Web APIs of the browser or node.js

setTimeout(CallBack, Time);

From the example, when the seconds of the time becomes zero which will be automatically decremented,

The event loop take the callback function which becomes zero and put it on the Callback queue.

Event loop checks for the availability of call stack.

=> If anything is executing, it'll not touch it

=> But when empty, it takes the function inside the callback queue place it in the call stack and E.C will be created to execute.

Promises:

Promises makes the asynchronous execution very smoothly and with more more readability to avoid "Callback hell or the ~~Pyramid of doom~~" which is caused by nesting the too many setTimeout function

Syntax:

```
new promise(callback_func);
```

Every promise contains

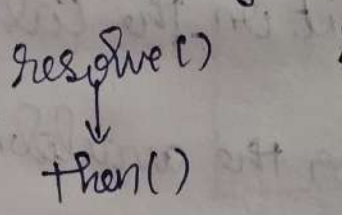
state : pending, fulfilled, unfulfilled

A executor function: callback function

To change the state, builtins

resolve() → success, fulfilled

reject() → failure, unfulfilled



Example

```
let prm = new Promise(() => {
    c.l("Promise executing...");
    resolve("success"); // reject() for failure
});
prm.then((result) => // catch for capture failure
    { c.l(result);
    }
```

(we can give any message)

Also simple syntax is

```
prm.then().catch()
prm
.then()
.catch()
```

then() & catch() takes the function to execute

Promises Async Await:

Instead of writing .then .catch
js introduce a concept called async await

Example:

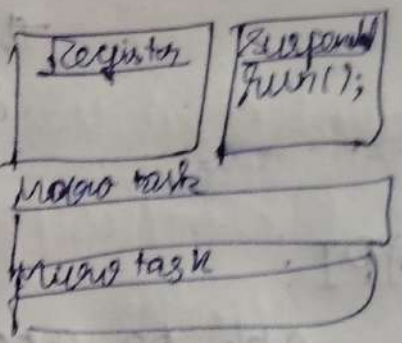
```
function prm() {
    return new Promise((resolve, reject) =>
    {
        setTimeout(() => {
            c.l("Task completed");
            resolve();
        }, 3);
    });
}

async function run() {
    await prm();
    c.l("promises fullfilled");
}
run();
```

Tells the function that it has asynchronous function operation

Tells that it will take time

When it sees await it take the whole line and put it in the suspended & take wait for the async function completion and then resume

In memory



prm -->

State: i
resolve -> then
reject -> catch

JSON:

=> Javascript Object Notation

=> This is the standard format which the way of communication happening b/w two systems.

=> Every programming language understands the json format

=> Acheived using the APIs. and the json which are sent or received is always an stringified json.

Example:

```
const weather = "{\"location\":{\"city\":\"CBE\"
const parsedData = JSON.parse(weather)
```

↓ This will Convert the stringified JSON to Objects in JS

```
const response = {success: true,
                  msg: "Received data"}
const JsonString = JSON.Stringify(response)
```

↓ This converts JS objects to stringified JSON for communication

Fetch API:

Able to access the another softwares output without exposing the code / logic behind the scenes

Prerequisites needed: Asynchronous; promises, promises chaining, Async await

* Fetch will always return the promise
* Fetch always works asynchronous because it uses another application that may take time which should not affect our application

When the fetch is called the returned data will be a text bits which is not human readable.

Ex:

```
fetch(URL)
.then((res) => res.text())
.then((txt) => c.l(txt))
.catch(() => c.l("Failed"));
```

→ This will also return a new promise so that to catch the res this is used.

we can convert it into directly by json for objects

```
fetch(URL)
.then((res) => res.json())
.then((txt) => c.l(txt))
.catch(() => c.l("Failed"));
```

Instead of doing like

```
.then((res) => res.text())
.then((txt) => {
    const data = JSON.parse(txt);
    c.l(data);
})
.catch(() => c.l("Failed"));
```

HTTP Methods:

To communicate between two computers / servers, a common way is introduced called protocal

HTTP -> Hyperlext Transfer protocol

HTTP Methods:

=> Get -> used to retrieve info from server
=> post -> used to create something new on server
=> put -> used to update something completely
=> patch -> used to make a partial update
=> Delete => used to remove something from server

Syntax:

fetch(URL, OPTIONAL) → Here we can give lots message but is optional. By default it is get method. → It is a JS obj

Ex:

fetch(URL, {method: "GET"});

The Request Object will be look like

Method: GET/POST/PUT/PATCH/DELETE
Headers: 'content Type': 'Accept': 'Authorization': 'user-Agent':
Body: JSON: 2 3

Content-Type => Tells what type of data we're sending

Accept => What is the format in which we want to accept data

Authorization => Security for the authorized user

User-Agent => Info about the devices or which browsers, which OS, etc..

Example:

Method : POST

Headers:

'Content-Type': application/JSON

'Accept' : application/json

'Authorization': username/pwd (JWT)

'User-Agent': browser/OS

Body:

"JSON"

```
let data = await fetch(URL,
    { method:"POST",
      headers : {
          'Content-Type : 'application/json';
          Accept        : 'application/json'.
            // Remaining will be automatically
            // captured so it is optional
      },
      body : { "name" : "Manoj"
               "RollNo": 232
             }
    });
let txt = await data.json();
c.l(txt);
```

→ needed to be string so store this in a var and const data = and replace here with JSON.stringify(data)