⇒Diff b/w ECMA Script and JavaScript

ECMAScript	JavaScript
★ ECMAScript is Specifications	★ JS is programming language that meet those specifications

⇒ **node:** is a piece of code that includes chrome V8 Engine just like we prodive JS to browser, it also take JS display result to command Line.

Primitive Types	Reference Types
i) String	i) Objects
ii) Number (type of (infinity or NaN))	ii) Arrays
iii) Boolean	iii) functions
iv) undefined ~~Type of undefined~~	★ A-date
v) null Type Of (Object)	

⇒ **Dynamically Typed Language:**
ie we can assigne any type of value to variable at run time.

⇒ Default value of variable in JS is 'undefined'.

⇒ Array: i) Length of an array in JS is Dynamic.

ii) We can store different type of elements in array. So technically "It is an Object".

⇒ DOM Manipulation :-

* tag.innerText = displays only visible text
* tag.textContent = display text content with line break
* tag.innerHTML = " " with HTML tag and Line break

⇒ setAttribute('name', 'value'); ⇒ removeAttribute('name');

⇒ Modifying Classes :-

i) element.classList.add('name');

ii) element.classList.remove('name');

iii) element.classList.contains('name'); // true or false.

⇒ Remove Elements :-

i) element.remove();

DOM: is a Property of browser's window object which is the top level object representing tab in Browser. (Page)

⇒ Event Propagation:- refers "that how Event travels through DOM tree" like an electricity trough wire.

* Event Listeners is 'Umbrella Term' captured in 3 Phases:

(2) i) Event Capturing → (Starts from root it reaches the target) after this enters in 'bubbling'
(3) ii) Event Bubbling (It again goes to root element)
(2) iii) Target

⇒ Event Delegation:- "It allows users to append a single event Listener to parent element that adds it to all of its present and future descendants that match a selector.

↳ Benefits:- There two benefits

i) Performance Improves
ii) It would work dynamically for future Elements.

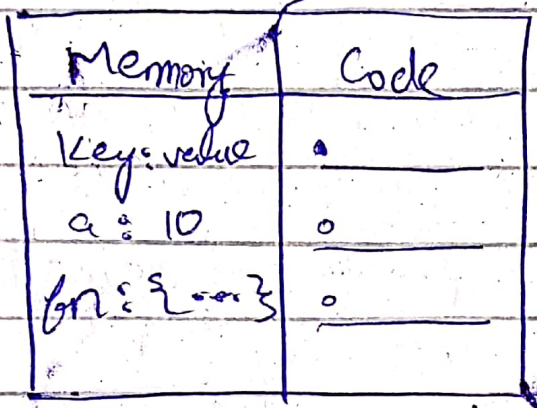
⇒ Event Capturing & bubbling allows us to implement one of most powerful "event handling patterns" called Event Delegation

⇒ How JavaScript Works?

"Everything in JavaScript happens inside an Execution Context"

⇒ Execution Context:

→ Everything happens in JS.
→ Execution Context is a big Container
→ Have two parts/components



→ This is place where code executed one line at a time.

→ Variable Environment
→ Where all veriables & functions stored as key/value pair.

→ It is also called "Thread of Execution"

→ Just like a thread in which whole code is executed one line at a time.

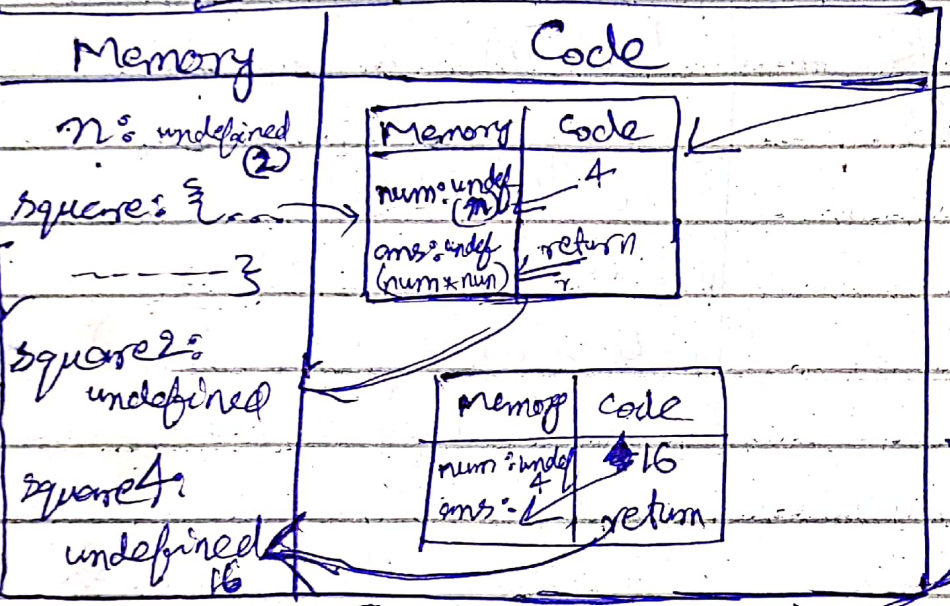
★ "JS is Synchronous Single-Threaded Language"

→ Means JS can't only move to next line once the one line completed it execution.

How JS Code Is Executed ?

Execution Context :— Created in two phases.

1) Memory Creation phase




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

In case of function it store whole code of function.

In case of variable it store undefined

2) Code Execution Phase

How JS Engine Knows about the nested Execution Context?

Ans: It manages it by a "Call Stack"

Global Execution Context

Once the code Execution is completed, global Exe context is destroyed

⇒ Hoisting in JavaScript:

"Hoisting is a phenomenon in JS by which we can access variables and functions even before initializing them."

Practice: Must Check the working of callstack.

Example:

```
i) getName();
ii) console.log(x);
iii) console.log(getName2);
iv) var x = 7;

v) var getName = () => {
       console.log("Namaste JS");
   }
function getName2() {
       console.log("Hello");
}
```

Because in Execution Context in Memory Allocation phase these are variables & considered to be undefined.

And this is considered function in Memory allocation phase.

Result:

⇒ ~~Namaste JS~~ undefined: getName2

⇒ undefined: x

⇒ Hello

⇒ How Functions Work in JS:

i) Function has its own Execution Context.
ii) Function's Execution Context placed into 'Call Stack' whenever the function is involved.

Question ?  How Execution Content of Console.log is managed?

⇒ Practice:

i) Browser → Source → script.js to see Step-by-step execution by debugger.
ii) Local / Global variable 'x'

⇒ window & this Keyword

★ Whenever the Global Execution Context of JS Program is created, the window object also created by the JS Engine.

★ JS Engine also creates 'this' Keyword at that time and it points to 'window' object.

★ whereEver the JS Program run Global Object would be created like On browser (window), in node (some other).

object name

"Any variables and functions created in global spaces, these are attached to window object"

These variables can be accessed with window object
window.a ⟹ Display 10

```
var a = 10;
function b(){
    console.log('Hello');
}
```

window.a, a and this.a all are refering to same thing ← Here

⇒ Undefined & not defined:

It doesn't means that it is not taking any memory like ~~memory~~ empty

It is like placeholder until value is assigned to variable

It is coming from place JS Engine where has set 'undefined' in memory allocation part.

```
console.log(a); //undefined
```

Example: var a = 7;

```
Console.log(x); //not defined
```

⇒ JS is Loosely Typed Language.

⇒ Also known as weakly typed language.

⇒ Things that shouldn't Do:

i) Don't assign 'undefined' to variable explicity.

⇒ Scope Chain, Scope & Lexical Environment:-

Example 1: It will first look at local Execution context, if not found value, then it would get value from outer function's execution context.

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

Example 2: error: b is not defined

```
function a(){
    var b = 10;
    c();
    function c(){
    }
}
a();
console.log(b); // not defined
```

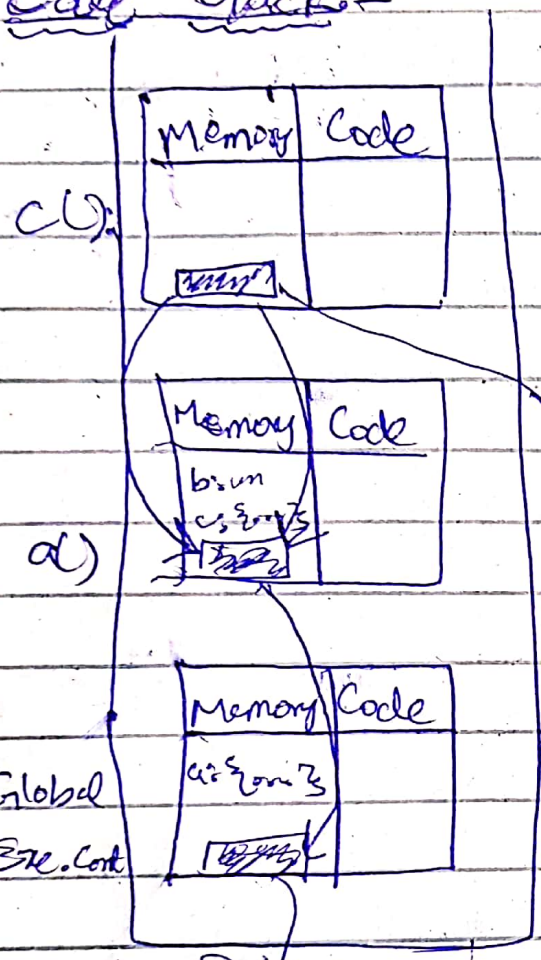
⇒ Lexical Environment:

★ WhereEver the execution Context is created a lexical Environment is also created.

★ "Lexical Environment" is the local Memory along with the lexical environment of its parent.

→ Scope Chaining

Call Stack:-



It points to the lexical Environment of its parent

```
function a(){
    var b = 10;
    c();
    function c(){
        console.log(b); // 10
    }
}
a();
console.log(b); // not def
```

★ Whenever the Execution Context is created, we also get the access to lexical environment of its parents.

⇒ This way of finding variable in different Lexical Environment is called Scope Chaining.

⇒ let & Const Declarations are Hoisted?

★ Variables delared using let & const also allocated memory but they are stored in different Memory space than global.

★ We Cant access this Memory Space until we put some values in them.

↳ Console.log (a) // Error
↳ let a = 10;
↳ var b = 100;

★ let variables doesn't attached to the window object.

eg :- In above 'a' is not attached to windou

⇒ Temporal Dead Zone :- is a time since when the let variable was hoisted until it is initialized some value.

Means memory has been assigned with placehoder

Note: ★ Whenever we access the variable that is inside "Temporal Dead Zone", we get a "Reference Error".

Type Error	Reference Error	Syntax Error
When we assign val to const variable, it gives "Type error." eg const a = 10; a = 100	When JS tries to access var that is not in his memory space.	1) When we didn't assign val to const variable at time of declaration 2) When we redefined vars using let.

⇒ How to avoid Temporal Dead Zone?

Always initialize variables on the top.

⇒ Block Scope & Shadowing in JS:-

* let & const are "block scoped variables."

* They are allocated in Execution context of block Memory.

* It would stop execution on line no 8 and throw error because block Exe Context is not in scope.

```
{
i) var a = 20;
ii) let b = 20;
iii) const c = 30;
iv) console.log(a);
v) console.log(b);
vi) console.log(c);
}
vii) console.log(a);
viii) console.log(b);
console.log(c);
```

Shadowing

⇒ Shadowing in JS is a process of permanently getting overriding val of outer scope var to inner block.

var b.

This variable var a is shadow of outer variable

It will modify val of outer variable also

Because 'var a' in the global scope not in the block scope

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

⇒ Shadowing in JS :

It is not only related to block scope but related function scope also.

⇒ Illegal Shadowing :-

We cannot shadow a variable using let a = 20 outer scope and var a = 20 inner scope.

Closures in JS

"A function bind together with its lexical environment"

"A function along with its lexical scope forms a closure"

Note: When a function returned from another function, they still maintain their lexical scope. (where it was actually present)

⇒ So, function along with its lexical scope was "Returned"

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

⇒ setTimeOut + Closures :-

What setTimeout does?

⇒ It takes the callback function & store it somewhere when the timer completes. It takes func put in CallStack and execute it.

```
function x(){
  var i = 1;
  setTimeout(function(){
    console.log(i);
  }, 3000);
  console.log("Namaste JS");
}
x();
```

⇒ This is because JS Continue its code until timer Completes and due to closure (function binds to its lexical environment).

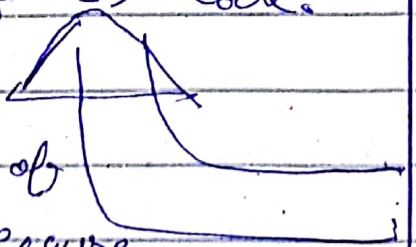
⇒ Given callback function would always get reference to 'i' variable.

⇒ At that time val 'i' would be 6 due to continuous execution of JS code.

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

Output:

Namaste JS
6
6
6
6
6



⇒ Solution:

⇒ use let i instead of var i in loop. Because it has block scope and each iteration have its own value pass to

⇒ Corner Cases in Closures:

What value would be consider for a? ← Here

When we get/use variable we play with its reference not value. so,
First:- Reference of a got 7,
Second:-
got 100.

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

⇒ Uses of Closures

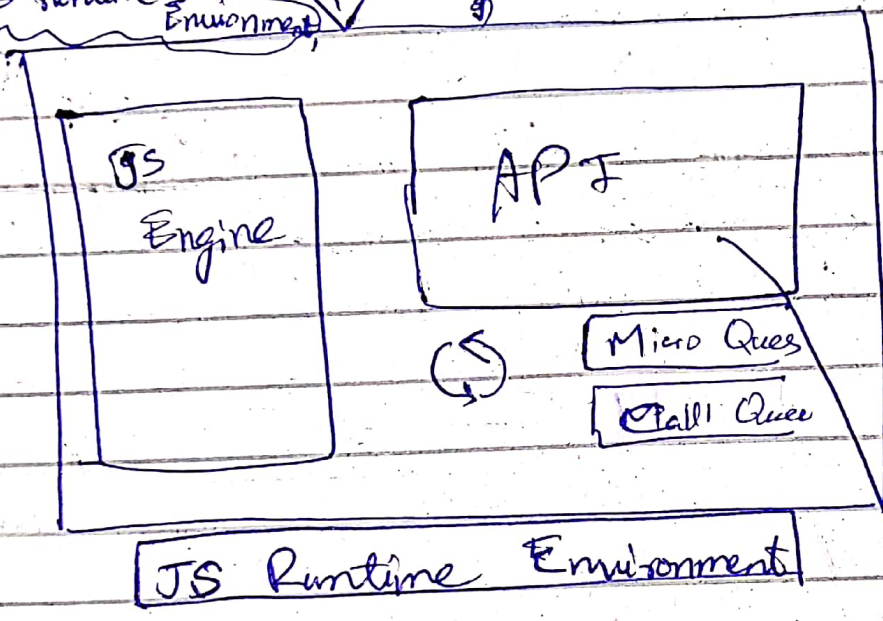
i) Module Design Pattern
ii) Currying (Have to watch video)?
iii) Functions like Once
iv) Memorize
v) Maintaing State in async World.
vi) set Timeouts
vii) Iterators
viii) More....

Java Script

⇒ Java Script Engine :—

(i) JS can run inside a) Browser d) Smart Watch
b) Server e) Light Bulb
c) Robot

This all possible because of JS runtime Environment



⇒ Every Browser has JS Environment

⇒ Node.js is JS runtime environment

e.g to run JS in water cooler, we need JS runtime

In Case of water cooler, these API's might be different

like

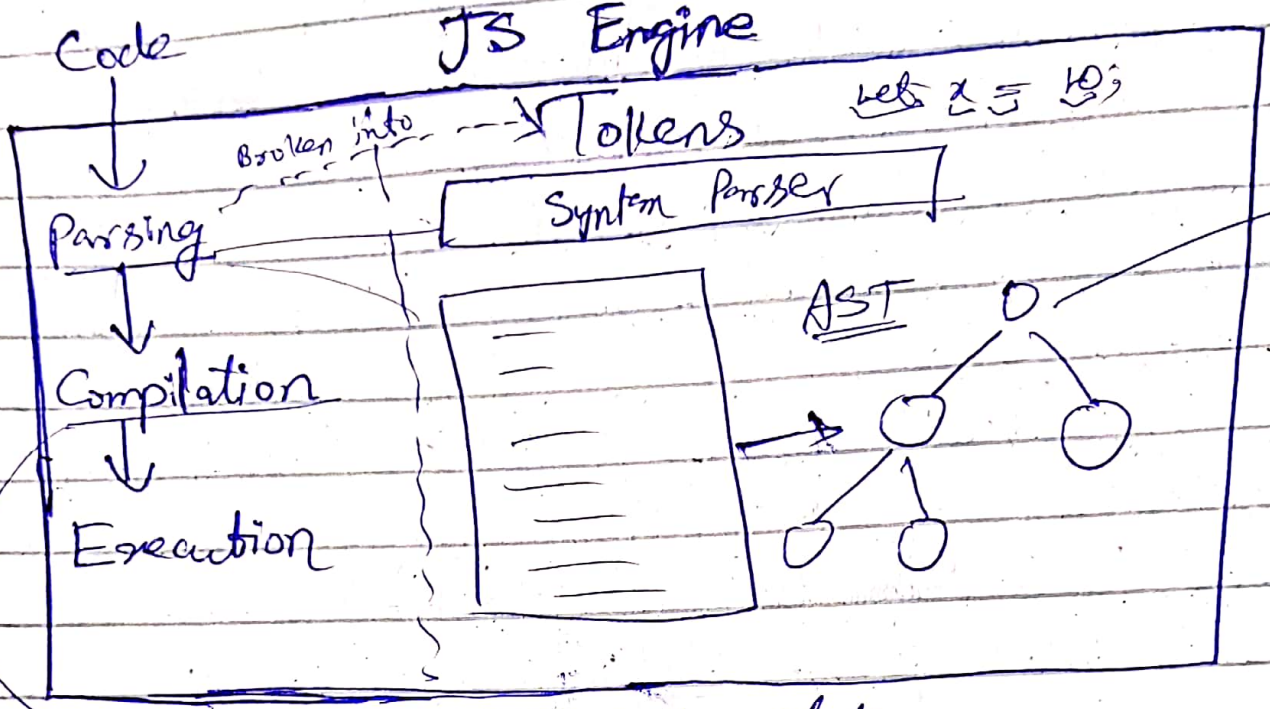
get water level for check water-level of cooler

⇒ ECMAScript is the governing body of JS language.

Read More on Google

JS Engine

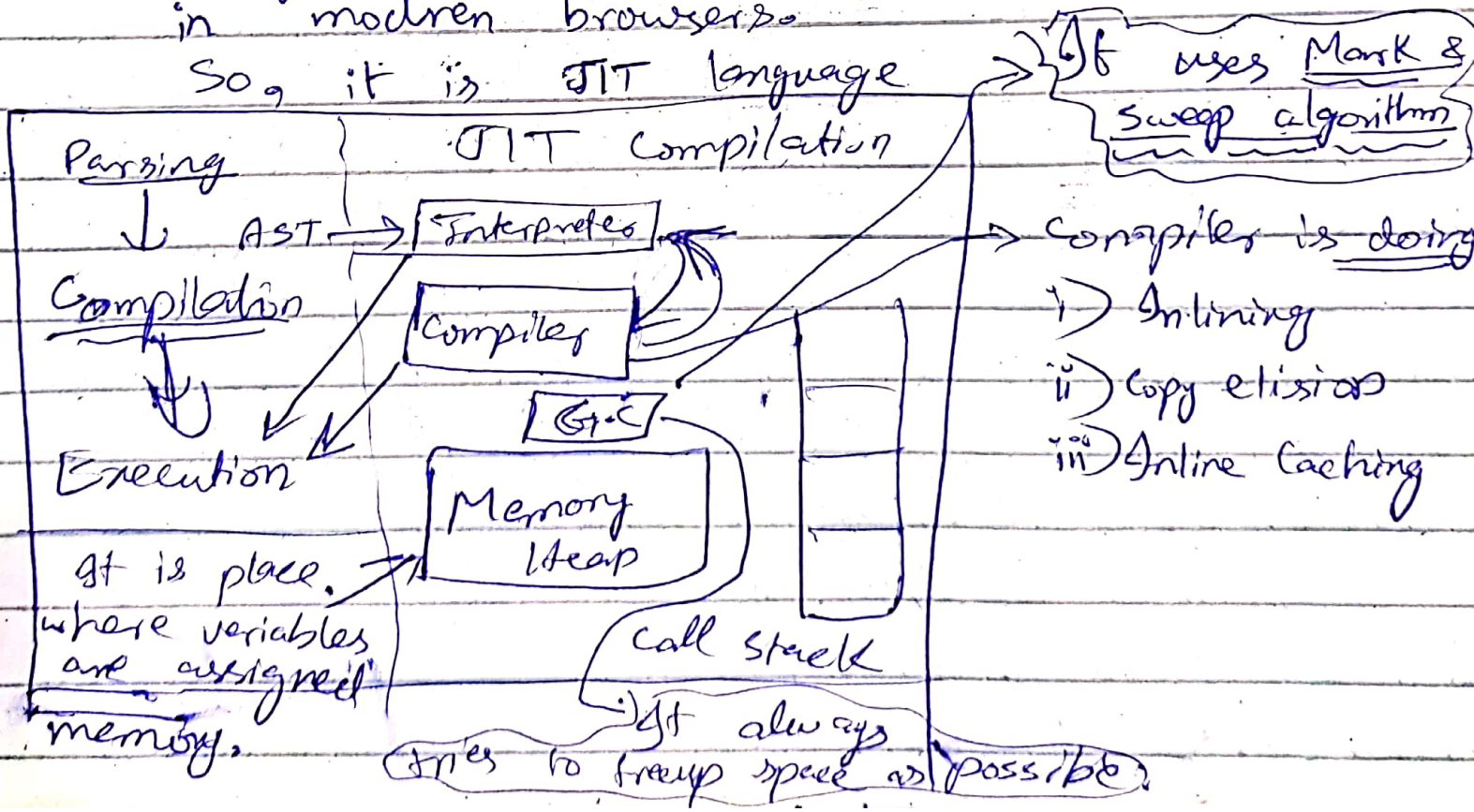
→ It is a program written in low level language like C++;



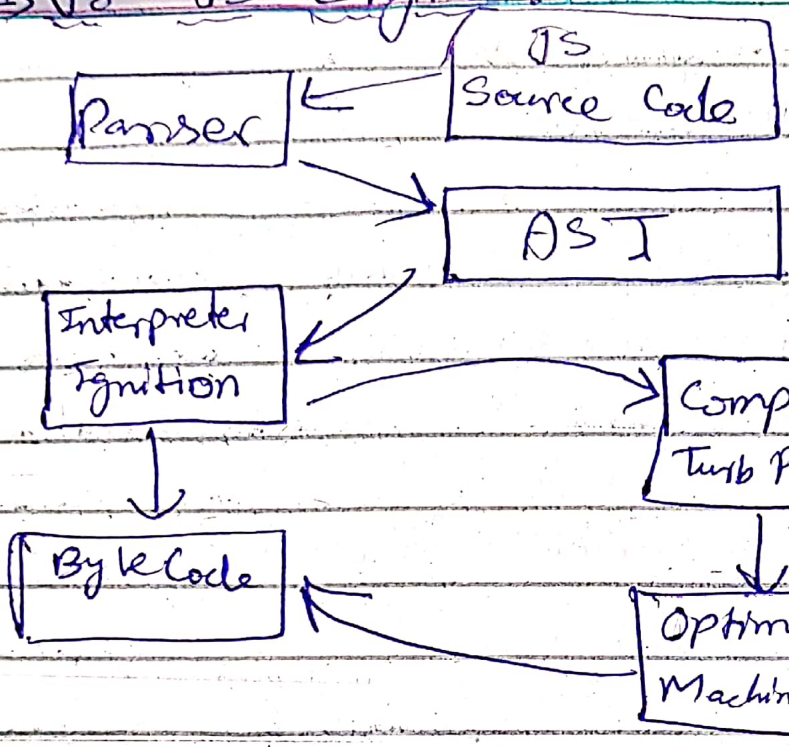
JIT Compilation

⇒ JS Engine uses Interpreter along with compiler in modren browsers.

So, it is JIT language



⇒ V8 JS Engine :-



V8 :

Read V8 Blog

⇒ map, filter & reduce :-

↳ map :-

used to transform an array

For Example,

```
const arr = [5, 1, 3, 2, 6]

function binary(x){
    return x.toString(2);
}

Const output = arr.map(binary)

console.log(output);
```

⇒ Higher Order Function :-

A Function takes function as argument a function or return a function from it is "Higher Order function"

⇒ Filter :-

used to filter some values in an array.

e.g: even values in array, odd values.

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

Example

```
function isEven(x) {
    return x%2 === 0
}

Const output = arr.filter(isEven);

console.log(output);
```

⇒ reduce :-

used in a case where we have to take all the values in the and come up with a single value from them.

↳ Use reduce (this) to find sum or Max value from an array.

→ callback to reduce would take two arguments.

→ It is initial value of an accumulator.

```
Const output = arr.reduce(
    function(acc, curr) {

    }, 0);
```