3/Oct/24

"Everything in JavaScript happens inside an Execution Context."

Execution Context → big box → 2 components.

Memory	Code Component
key: value pairs a: 10 fn: {...}	Code is executed there line by line

Variable environment — Thread of Execution

JavaScript is Synchronous → in a order

Single-threaded language.

can execute one line at a time.

→ means it execute line by line code.

AJAX ?

How does JavaScript Code executes? 1st phase: Memory creation ☆ 2nd phase: Code Execution. ☆

It executes in phase, first JS skims through all the program and reserves space in memory component.

allocate memory to global space.

Memory

n: undefined → 2 → in phase 2

square: {... } whole code is there

square2: undefined → 4

square4: undefined → 16

Code

```
1 var n=2;
2 function square(num){
3     var ans = num*num;
4     return ans;
5 }
6 var square2 = square(n);
7 var square4 = square(4)
```

if will be stored in place of undefined in n.

parameter of function

arguments

function is called. (invoked) → a new execution context is created now ☆

new

memory	code
num: undefined → 2	in phase 2 n=2 in memory
ans: undefined → 4	(num*num) num will go to return

keyword.

placeholder

we will again follow 2 phase

return control to the execution context where function was invoked at. at line 6

will now replace this

it will be deleted after the replacement.

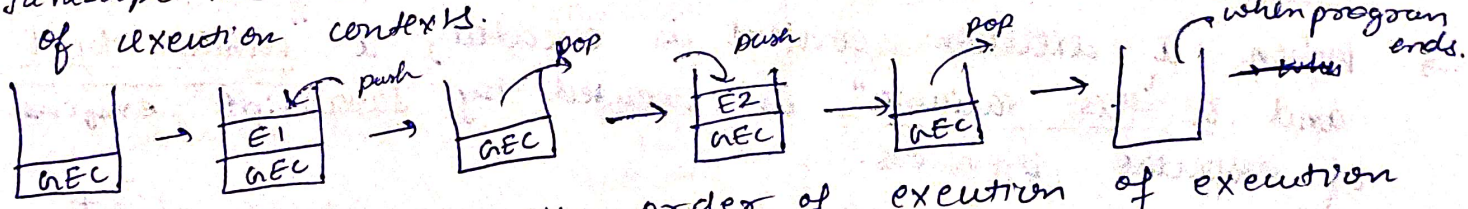
at line 7

memory	code
num: undefined → 4	num*num → 16
ans: undefined → 16	return ans to square4

⇒ After all the code is executed, the global execution context will be also be deleted.

4/Oct/2024

Javascript has a call stack to manage the creation and deletion of execution contexts.



Call stack maintains the order of execution of execution contexts.

Hoisting → You can access the variables and functions before they are defined without any error.

ex: getName();
console.log(x);
var x = 7;
function getName() {
console.log("Hello");
}

→ it will print: → Hello
→ undefined.

→ now if var x=7 is removed, we will get
→ Hello
→ error that x is not defined.

& console.log(getName);
↳ will give function print whole function.

Memory is allocated to all variables and function before execution.

→ if var x = 7 was removed, there will be an error, because for x, memory was not reserved

Arrow function:

var getName = () => {
...
};

⟹ It will behave like an variable and undefined will be stored allocated for it. for
var getName = function (){
} two same thing

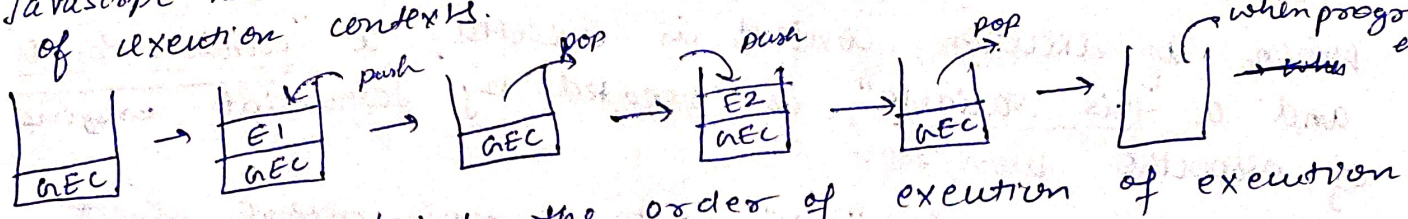
03/Dec/2024

→ How functions are created:

- At first a global execution context is created consisting of Memory and Code → and it will have 2 phases memory allocation phase and code execution phase.
- In the first phase, the variables are assigned "undefined" while functions have their own code.
- Whenever there is a function declaration in the code, a separate EC gets created having its own phases and is pushed into the stack.
- Once the function ends, the EC is removed from the call stack.
- When program ends, the Global EC is also pulled out.

4/Oct/2024

Javascript has a call stack to manage the creation and deletion of execution contexts.



Call stack maintains the order of execution of execution contexts.

Hoisting → You can access the variables and functions before they are defined without any error.

ex: getName();
Console.log(x);
var x = 7;
function getName(){
console.log("Hello");
}

→ it will print: → Hello
→ undefined.

→ now if var x=7 is removed, we will get
→ Hello
→ error that x is not defined.

& console.log(getName);
↳ will give function print whole function.

Memory is allocated to all variables and function before execution.

→ if var x=7 was removed, there will be an error, because memory was not reserved for x.

Arrow function:

var getName = () => {
...
};

⟹ It will behave like an variable and undefined will be allocated for it. for
var getName = function (){
} too same thing

03/Dec/2024

→ How functions are created:

- At first a global execution context is created consisting of Memory and Code → and it will have 2 phases memory allocation phase and code execution phase.
- In the first phase, the variables are assigned "undefined" while functions have their own code.
- Whenever there is a function declaration in the code, a separate EC gets created having its own phases and is pushed into the stack.
- Once the function ends, the EC is removed from the call stack.
- When program ends, the global EC is also pulled out.

⇒ **Window:** Even when the code is empty a window object is being created → by the JS engine.

→ When a execution context is created, a "window object" and a "this variable" is created by JavaScript engine of respective browsers.

→ at global level "this" points to the global object (which is window object in case of browsers).

→ anything that is not inside a function is in the "global space".

→ whenever we create any variables or functions in the "global space", they get attached to the global object (window objet in case of browsers).

→ To access variables/ functions defined in global space: we can use any of the below:

```
Console.log (window.a);
Console.log (a);
Console.log (this.a);
```

→ JS assumes by default if we mean (window.a) ☆

⇒ **Undefined vs Not defined:**

→ Undefined is a placeholder till a variable is assigned a value.

→ Not defined means we try to console or access a variable, which is not defined in the code

→ JS is loosely typed / weakly typed language → means it does not attach its variables to specific data types like in C++ and Java.

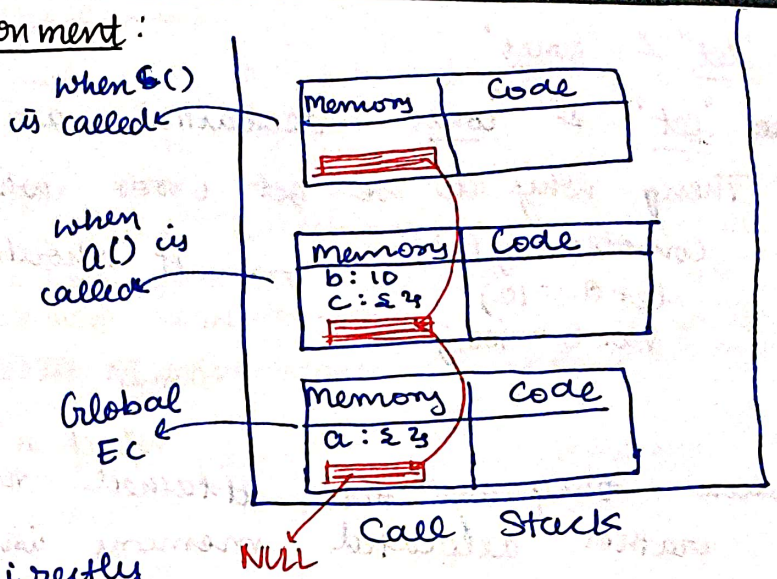
Ex:
```
var a;
console.log(a);
a = 10;
console.log(a);
a = "hello";
console.log(a);
```

⇒ This will give this output: ☆
```
undefined
10
hello
```

Now if I add
```
console.log(x);
```
here it will throw the error that x is "not defined" since x has not been defined in the code anywhere.

⇒ Scope Chain & Lexical Environment:

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



→ Scope of a variable is directly dependent on the lexical environment.

→ Lexical means sequential/hierchal (literal meaning).

→ Whenever an EC is created, a lexical environment is created.

→ Lexical environment is the local memory + the lexical environment of its parent.
(in this example c has reference ~~pointer~~ to a (its pointer) which further has reference ~~pointer~~ to global EC)

→ Having reference of parent's lexical environment means the child or the local function can access all the variables and functions defined in the memory space of its lexical parent.

→ The JS engine first searches for a variable in the current local memory space, if it's not found, it searches for the variable in the lexical environment of its parent, and if still not found, it searches that variable in the subsequent lexical environments, and the sequence goes on until the variable is found or the lex. env. becomes NULL.

→ The mechanism of searching variables in the subsequent lexical environment is known as "Scope Chain".

→ If a variable is not found anywhere, we say it is not ~~found~~ in the Scope Chain.

⇒ "let" & "const"

⇒ "let" & "const" declaration are hoisted.

→ Then why do we get error when we write:

```
console.log(a);
let a = 10;
var b = 100;
```

→ It should give undefined, right?

→ will give a Reference Error: Cannot access 'a' before initialization. while trying to access variable which is not yet put in global memory.

⇒

→ Because they are not attached to the global object but are rather allocated memory in different memory space known as "Script".

→ Temporal dead zone: It is the 'zone' or time since when let and const variables were hoisted (set as 'undefined') and till it is initialized with some value. If we try to access a "let" or "const" variable in the temporal dead zone, it will give "reference error".

→ let and const are more strict than var.

→ Redecalration of let is not allowed.

```
let a = 10;
let a = 100;
```

→ Syntax Error: Identifier 'a' has already been declared. → violation of JS syntax

→ const is even more strict as we have to initialize it just when we declare it.

```
const b;
b = 1000;
```

→ Syntax Error: Missing initializer in const declaration.

→ Once const has been assigned a value, it cannot be changed again in code.

```
const b = 100;
b = 10000;
```

→ Type Error: Assignment to constant variable. → trying to re-initialize const variable.

→ Minimize Temporal dead zone by always putting the declarations on the top!

⇒ **Block:** → code inside curly brackets.

→ Multiple JS statements formed in a group enclosed in brackets and it forms a block. { } → block.

→ JS sometimes expects a single line to run but we need to run commands with multiple statements. ex. in if statement

```
{ var a = 10;
  let b = 20;
  const c = 30;
}
```

→ when this code is hoisted, var is attached to Global env. and other 'let' and 'const' go to 'Block environment'. [separate memory space].

→ This helps in understanding scope.

→ a can be accessed outside of block. (since it exists in global env.)

→ But b and c cannot be acced outside of block.

→ 'let & const are Block Scoped'.

What is shadowing? → It occurs when a variable declared in an inner scope has the same name as a variable in outer scope.

```
var a = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(a);
  console.log(b);
  console.log(c);
}
console.log(a);
console.log(b);
console.log(c);
```

⇒ Now when we try to print this we will get the output:

10 → here a was overshadowed
20
30
10 → and even outside the block the value was changed.
20
30

⇒ But in case we had let b = 100; outside. we would get the output:

10
20 → Here also overshadowing occurs, but it does not change the value outside the scope of block.
30
100 → console.log(b)

→ Same happens in case of const.

Illegal shadowing example:

```
let a = 20;
{ var a = 20; }
```
→ Illegal but
```
var a = 20;
{ let a = 20; }
```
→ legal.

```
let a = 20;
function x(){
  var a = 20;
}
```
→ legal.

Lexical block scope:

```
const a = 20;
{
  const a = 100;
  {
    const a = 200;
    console.log(a);
  }
}
```

→ Each block will have its own scope and it will follow scope chain.

☆ These scope rules works on arrow functions too.

⇒ Closures: → gives you access to outer fn's scope from an inner fn.

→ A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).

→ Simply: function + its lexical environment.

→ When a function is returned, even if its EC vanishes, the reference where it is was pointing to is remembered. Because a function alone is not returned but the entire closure.

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

→ This code will output:

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

→ 7

then when z() is called, 7 is printed in console. Even though the EC of above clone of code should have been deleted by now.

Instead of this, we can also write:

```
return function y(){
    console.log(a);
}
```

⇒ setTimeout:

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

This will output "Hello" and then wait for 3 second to also print 1

∴ Hello
1 → after 3 second.

⇒

→ SetTimeout stores the function in different place and attach a timer to it, when the timer is finished it rejoins the call stack and executed.

→ JS doesn't wait it will print out "Hello" and then print the 1.

Q: Print Numbers after 1 second:

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

⇒ This will output:

```
Hello
6
6
6
6
6
```

that is the set value printed. That is the value of loop i=6, and same place, thus at end and the i inference to the but all these have value of i All created for all setTimouts this loop runs, different functions
↳ since it has global scope

How to fix? Use let instead of var. let is block scoped and thus a new copy is created everytime the loop runs. we will get: 1 2 3 4 5
New value of i forms closure with this function. (updated)

How to fix using var?
So our main problem was, we need fresh copy of i everytime. so we do this!

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

we enclosed this in the function close, so now everytime close() is called, a new value of i is passed and printed.

7 | Dec | 2024

⇒ Use of double parentheses ()() in JS:

So if we have this function:

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

⇒ then to call the inner function:
→ we can do:

```
outer()();
```

instead of:

```
var close = outer();
close();
```

if we change this's var to let, will it still form an closure? Yes! Even though let is block scoped and cannot be accessed outside it will form a closure!

Are function parameters closed over?

```
function outer(b){
    function inner(){
        console.log(a, b);
    }
    let a = 10;
    return inner;
}
var close = outer("hello");
close();
```

(b) → also part of outer environment!

Yes, they are.

This function will print: 10, hello ☆

If there was another function outside it:

```
function outest(){
    var c = 20;
    function outer(b){
        function inner(){
            console.log(a, b, c);
        }
        let a = 10;
        return inner;
    }
    return outer;
}
let a = 100;
var close = outest()("hello");
close();
```

→ it will form closure and allow to outest too!

→ if we comment this, it will print a = 100!

→ completely new variable.

→ returns outer function & is called with this paramter.

global variable with conflicting name (it will check scope chain).

Advantage of Closure?

Used in data hiding and encapsulation!

other part of program cannot access these.

Data Encapsulation:

```
var counter = 0;
function incrementCounter(){
    counter++;
}
```

→ counter is a global variable and thus anyone in code can access it.

⇒ To protect this, we can make closure of this fn:

```
function counter(){
    var count = 0;
    return function incrementCounter(){
        count++;
        console.log(count);
    }
}
var counter1 = counter();
counter1(); → will print 1;
counter1(); → will print 2;
```

→ now if we call this again
`var counter2 = counter();`
`counter2;` → it will give 1, as it will be completely new variable.

Is this Scalable? No, we can use constructor function

```
function Counter(){
    var count = 0;
    this.incrementCounter = function(){
        count++;
        console.log(count);
    }
    this.decrementCounter = function(){
        count--;
        console.log(count);
    }
}
var counter1 = new Counter();
```

→ since we used a constructor function, so we have to use this keyword new

```
counter1.incrementCounter();
counter1.decrementCounter();
```

→ we get 1, 0

Disadvantage of Closures:

⇒ Overconsumption of Memory.

What is Garbage Collector?

In high level programming languages like JS, unused variables are automatically deleted.

```
function a(){
    var x=0; z=10;
    return function b(){
        console.log(x);
    }
}
var y = a();
y();
```

Example of Smart Garbage Collecting; since z was not being used in here, z will be garbage collected.

if we have y declared there, then the variable x won't be garbage collected as we may after sometime call y();

⇒ What are Function statements, Function Expression & Function Declaration?

Function Statement:

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

there is a naming convention and these functions are hoisted. So we can do: a() fn def;] legal.

Function declaration = Fun. state.

Function Expression:

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

assign a fn to variable, not hoisted and there as we will have b: undefined!

→ Anonymous Functions:
fn w/o names, used in place where fn is treated as value.

```
function(){
}
```

→ Named Function Expression:

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

if we do xyz(); → error, xyz not defined

if we log(xyz) we will set

```
f xyz(){
    console.log(xyz);
}
```

⇒ Difference b/w Parameter and Arguments:

```
function ab (param1, param2) {        → These are 'parameters'.
    ...
}
```

& when we call this function & pass variable in (), it's 'arguments' : ab(4,5);

⇒ First class Functions / First class citizens:

→ ability to use function as value.
→ can be passed as an argument, can be executed inside a closured function & can be taken as return form.

```
ex: var b = fn.(param) {
        return fn.xyz() {
            console.log("FCF");
        }
    }

    b ( function() {
        });
```

→ argument

→ return type

also legal:

```
fn. xyz() {
}
b(xyz);
```

⇒ Callback function:

- functions that are passed on as an argument to another function.

Javascript is a synchronous and single-threaded language.

```
ex: function x(y) {
        console.log("x");
        y();
    }
    x( function y() {
        console.log("y");
    });
```

y is callback fn. → because it will be called sometime later in code.

Callback functions sort of like help make JS async.

setTimeout (function() {
    console.log("time");
}, 5000);

(setTimeout in async operator)

↳ callback function, will be called after 5000 ms.

Call Stack → Main Thread ☆

after x & y are executed, ~~setTi~~ function will pop in • call stack.

⇒ Event listener:

adds Event listener with the activity "click"

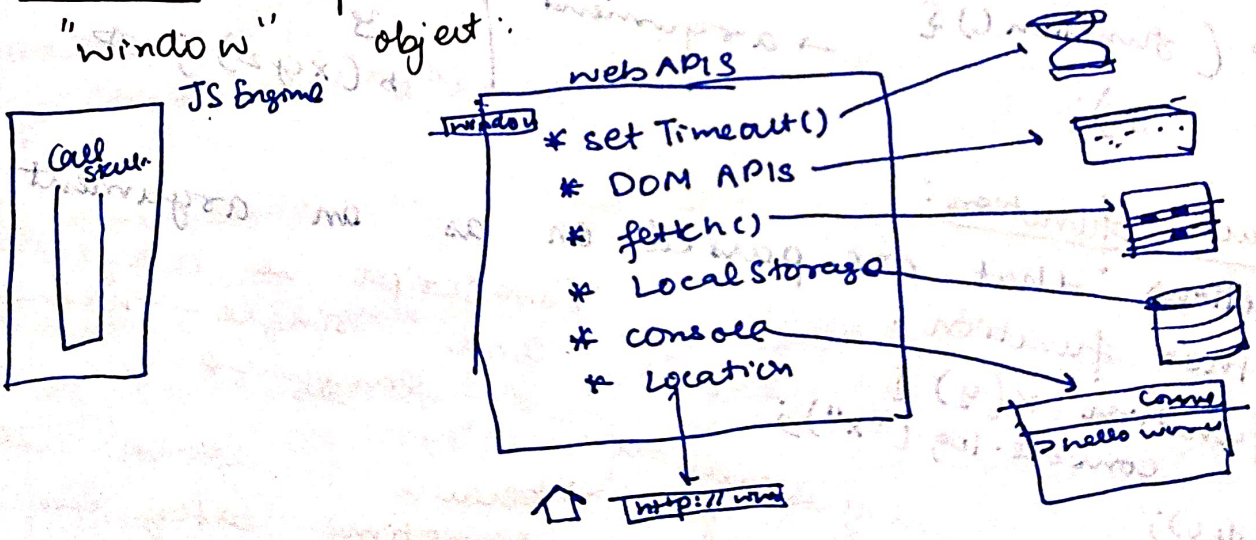
```
document.getElementById("ClickMe").addEventListener(
"click", function xyz() { console.log("Button Clicked") }
);
```

→ Everytime we click the "ClickMe", it will call the function xyz and put it in the call stack to execute.

→ ~~When Event Listeners~~ In the closure function, we will have scope of parent. (Lexical envir.).

→ Event listenor forms closures with functions. Thus they use a alot of memory. Thus we need to remove Eventlisteners.

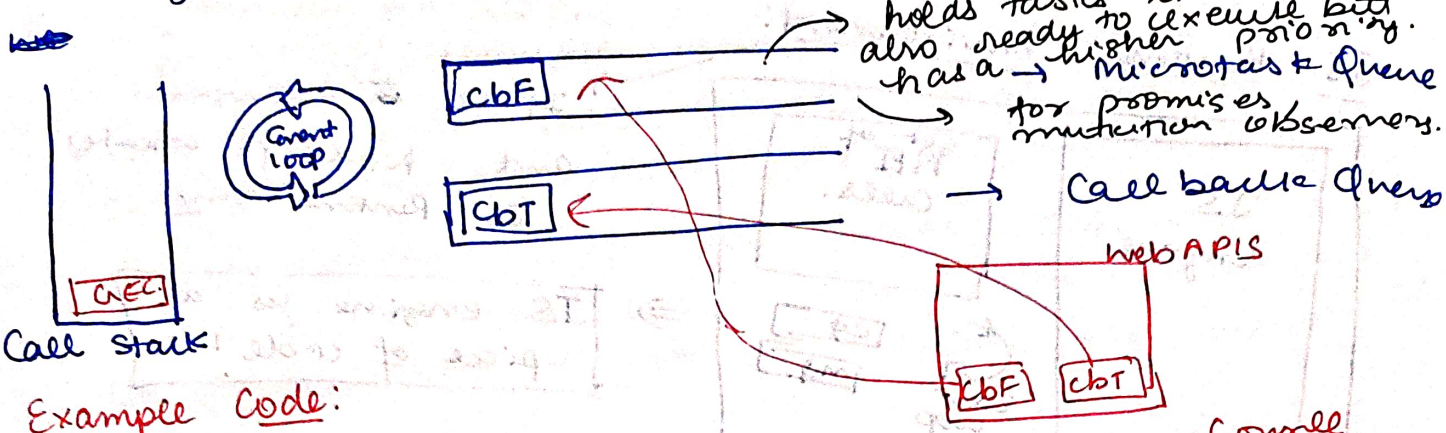
⇒ Web APIs: provided by Browserser: to JS through "window" object:



⇒ ~~EVENT LOOP~~:

⇒ ASYNCHRONOUS JAVASCRIPT:

We have Call Stack, Callback queue, Microtask queue and Event loop → all the key components to manage the asynchronous nature of Javascript.



Example Code:

```
Console.log ("Start");
SetTimeOut (function cbT() {
    console.log ("SetTime");
}, 5000);
fetch ("https://api.netflix.com")
.then (function cbF() {
    console.log ("Netflix");
});

console.log ("End");
```

Console
Start
End
Netflix
SetTime

☆ Event Loop: It is responsible for continuously checking the call stack and callback queue. If call stack is empty, the event loop takes the first task from callback queue and pushes it onto the call stack for execution. But if Microtask Queue is not empty, first the task that ~~event~~ will be pushed to call stack. ~~And then callback stack~~

Starvation: If the tasks in Microtask queue keep on adding up, the tasks in callback queue will "starve".

⇒ Event Loop → is a single-thread, loop that is always running.

⇒ Only async. ~~web API~~ callback function are registered in Web API environment.

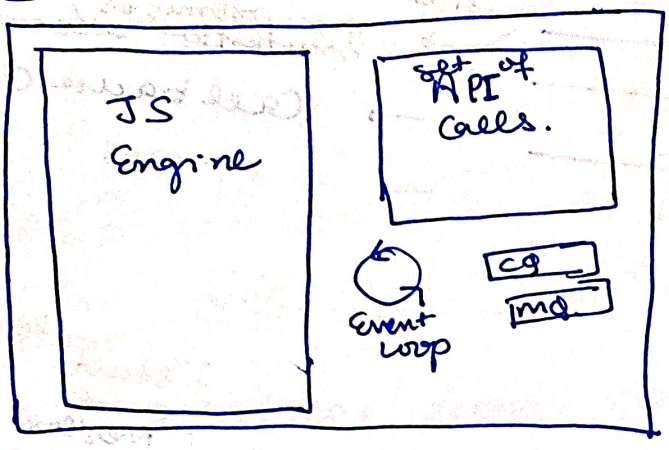
→ If we have setTimeout() set to 0ms, it will still be executed after the whole call stack is empty.

JavaScript Engine:

JS Runtime Environment: → contain all element required to run JS.

It is browser.

and Node.js creates JS Runtime env.



⇒ JS engine is a piece of code!

what JS engine does:

Code → Parsing → Compilation → execution

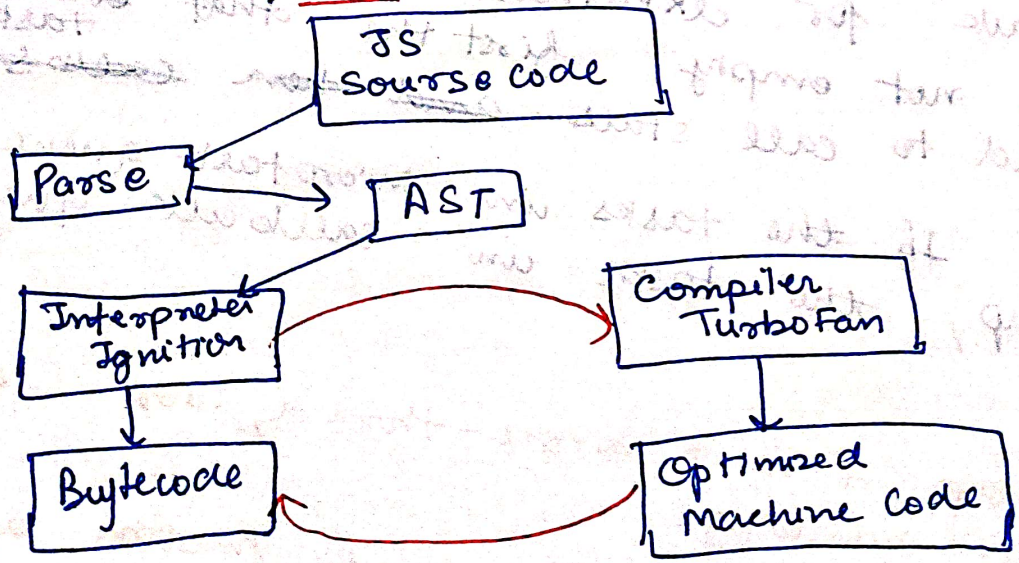
① In parsing code is broken down into tokens, and a abstract syntax tree is created.

② Modern JS engine follows JIT (Just-in-Time) Compilation, thus it interprets while it also optimises the code as much as it can.

(uses both interpreter & compiler)

③ Execution & Compilation is done together.

V8 JS Engine:



⇒ Mark & Sweep Algorithm:
↳ Garbage collection technique used in JS to identify and remove objects and variables that are no longer in use:

1. Mark phase: set the mark bit of all reachable ~~set~~ objects from root (global object) to ~~set~~ true. (we can use DFS for it), and will have to call DFS on all roots tent in the code.

2. Sweep phase: In this phase all the unreachable objects i.e, those with mark bit as false. ~~will be set~~ It clears the heap memory.

```
Mark (root)
If markedBit (root) = false then
        markedBit (root) = true
                For each v ref- by root
                Mark(V)

Sweep()
For each object p in heap
 If markedBit (p) = true then
        markedBit (p) = false
                else
                        heap.release (p)
```

⇒ SetTimeOut ensures that minimum it will take the time mentioned because it may be paused due to call stack not being empty

ex: if we have settime out (after 5s) ....

```
let StartDate = new Date().getTime();
let endDate = StartDate;
while (endDate < StartDate + 10000) {
    endDate = new Date().getTime();
}
console.log(while expires);
```

so first this will finish executing setTimeout even though 5s ∴ will be posted after 10s.

⇒ Functions in JS:

ex: If we have to calculate the area, circumference & diameter of a circle:

```
const radius = [3, 1, 2, 4];
const calculateArea = function (radius) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
        output.push (Math.PI * radius[i] * radius[i]);
    }
    return output;
};
console.log (calculateArea (radius));
```

Now → we write another fn calculate circum & calculate diameter

...

but as we can see, a lot of code will be repeating as we will only have to change the formula for calculation.

So it is not a good way to write code.

DRY (Don't repeat Yourself)!!

→ So instead of this, now will try to generalise this and make a Calculate fn.

2 we make:

```
const area = function (radius) {
    return Math.PI * radius * radius;
};
const circumference = function (radius) {
    return 2 * MATH.PI * radius;
};
const calculate = function (radius, logic) {
    const output = [];
    for (let i = 0; i < radius.length; i++) {
        output.push (logic (radius[i]));
    }
    return output;
};
```

```
console.log (calculate (radius, area));
console.log (calculate (radius, circumference));
```

Functions that takes another function as argument (callback function) is known as Higher order functions.

If we use Array.prototype.function-name, This function will be accesible to any array in our code.

```
Array.prototype.calwlak = fn(logic) {
    const output = [];
    for(let i=0; i < this.length; i++) {
        output.push (logic (this[i]));
    }
    return output;
};
console.log (radius.map(area));
console.log (radius.calculate(area));
```

↳ Sort of like an implementation of map that we have here

MAP:

Map method is used when we want transformation of whole array.

ex:
```
const arr = [5,1,3,2,6];
function double(x) {
    return x*2;
}
const output = arr.map(double);
```
→ will give the array as [10,2,6,4,12]

can also be written as:
```
const output = arr.map(function double(x) {
    return x*2;
});
```
↳ we put fn. directly as argument.

or also by using arrow functions:
```
const output = arr.map((x) => { return x*2; });
```
& since there is only one line of code, we can omit brackets
```
const output = arr.map((x) => x*2);
```

FILTER: It is used when we want to filter the array to obtain required value.

Let's say we want to find the odd integer from the array:

```
const arr = [5, 1, 3, 2, 6];
function isOdd (x){
        return x%2;
}
const output = arr.filter(isOdd);
```

(will print: [5, 1, 3]

here also we can do:

```
const output = arr.filter( function greaterThan4(x){
        return x>4; });
```

(will print [5, 6]

also:

```
const output = arr.filter ( (x) => {
        return x>4;
});
```

also:

```
const output = arr.filter( (x) => x>4);
```

REDUCE: → It is used when we want to reduce array to single value eg(max, min, avg, sum etc.)

→ It passes two arguments ① function (which includes accumulator and initial value as argument itself) and ② initial value of accumulator.

Let's say we want to find sum of array, then we do:

```
const arr = [5,1,3,2,6];
function findSum(arr) {
    let sum = 0;
    for (let i=0; i< arr.length; i++) {
        sum = sum + arr[i]
    }
    return sum;
}
console.log(findSum(arr));
```

↳ now let's do this with reduce:

```
const output = arr.reduce(function(acc, curr) {
    acc = acc + curr;
    return acc;
}, 0);
```

acc → accumulator; curr → current value in array; acc = acc + curr; → logic; return acc; 0 → initial value of accumulator.

Examples for Map, Filter & Reduce:

Let's say we have:

```
const Users = [
  { firstName: "Harshita", lastName: "Arya", age: 21 },
  { "" "" },
  { "" "" },
];
```

In this Question our curr will be this whole ☆.

Q: Return list of full names.

```
const output = users.map((x) => x.firstNam + " " + x.lastNam
```

Q: Return how many people are of how many ages.
ex: 26: 1, 75: , 50: 2 → no. of people who are 50 = 2

```
const output = users.reduce ( function (acc, curr) {
    if ( acc[curr.age]) {
        acc[curr.age] = ++acc[curr.age];
    } else {
        acc[curr.age] = 1;
    }
    return acc;
}, {});
```

initially empty ← value of acc.

{ 21: 1, 26: 2, 50: 1 ... }

Q: Return 1st name of people who are less than 30 yrs.

```
const output = users.filter ((x) => x.age < 30). map ((x)
        => x.firstName);
```

↳ ✡ These methods can be chained!
first filter will return people whose age are less than 30, then map is used on those results to get their first Name.