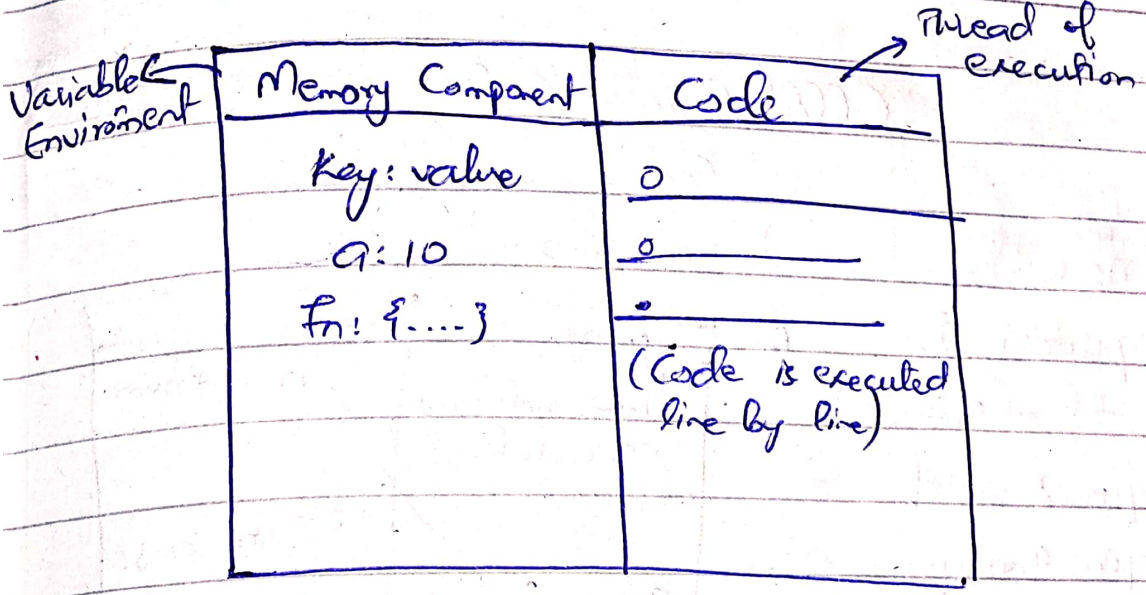
LECTURE 01:- HOW JAVASCRIPT WORKS AND EXECUTION CONTEXT:

"Everything in JavaScript happens inside the execution context."

Execution Context:



Execution context

-. Javascript is a synchronous single threaded language i.e. Javascript can only execute one command at a time in a specific order.

LECTURE 02- HOW JAVASCRIPT IS EXECUTED; & CALL STACK:

- When you run a javascript program an execution context is created

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

Global Execution Context

Memory	Code
1st phase n: undefined square: {...} whole code of fun. square2: undef square4: undef 2nd phase n: 2 Square2: 4	1) var n=2

5)

memory	Code
1st phase num: undefined ans: undefined 2nd phase n: 2 ans: num*num/4 local memory deleted	num*num return ans;

6)

	memory	Code
1st	num: undefined ans: undefined	num*num
2nd	num: 4 ans: 16	return ans

deleted

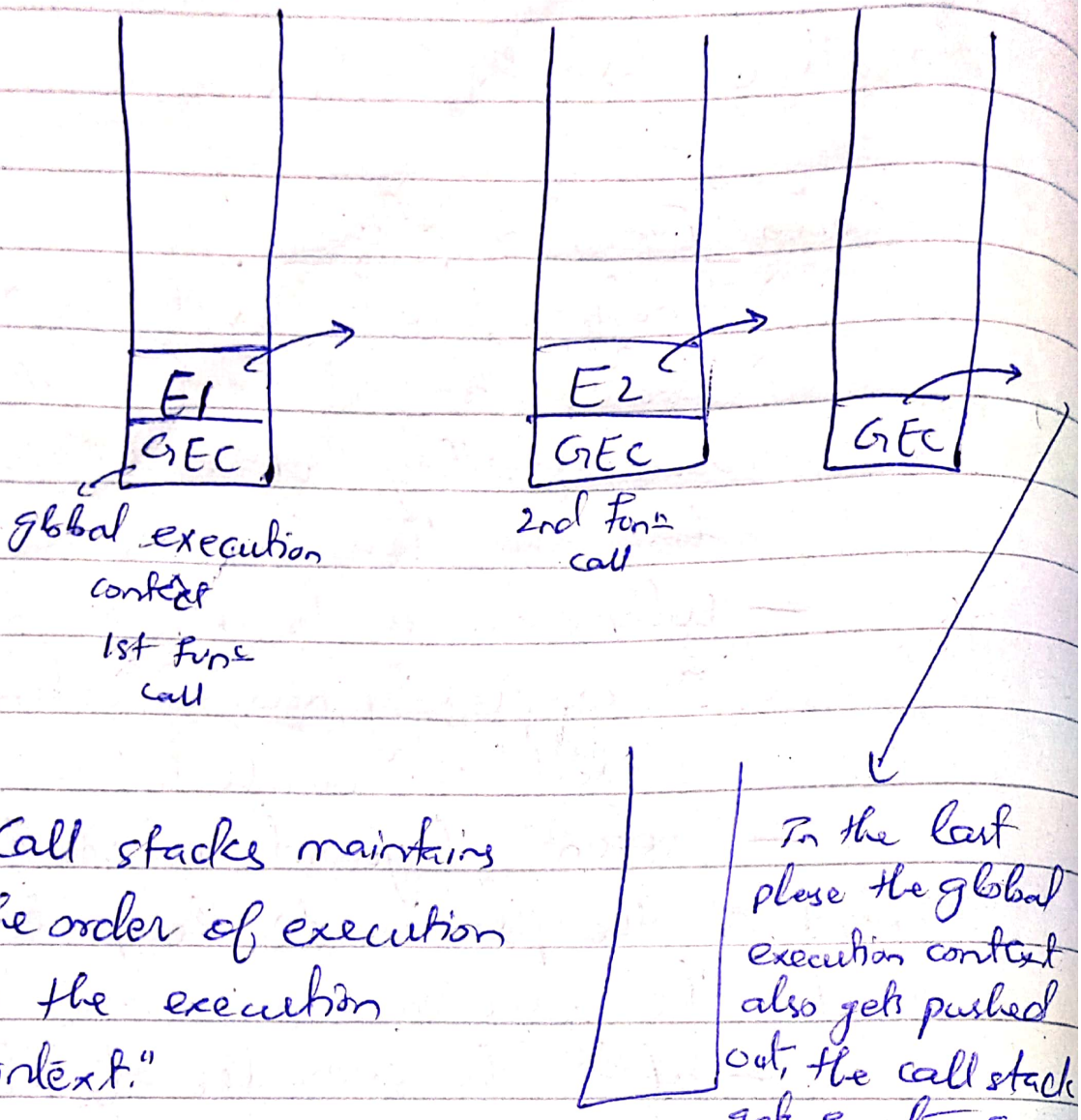
deleted

- Execution Context is created in 2 phases
  1) Memory Creation phase
     In the 1st phase of memory creation, javascript will allocate memory to all variables & functions.

  2) Code Execution phase:
     1) Function is invoked at line 06
     2) Functions are at the heart of Javascript, they behave differently than any other languages.
     - Whenever a function is invoked a completely new execution context is created.
     - "return" keyword returns the control of the program to the point at which the function was invoked.
     - after the encounter of return keyword & returning the value, whole of the execution context will be deleted
     - Once the program is finished the global execution context is also deleted.

CALL STACK:

The bottom of the stack always contains a global execution context.



Global execution context
1st func call

2nd func call

"Call stacks maintains the order of execution in the execution context."

In the last phase the global execution context also gets pushed out, the call stack gets empty & the program is finished.

Other names of Call stack:-

0) Call Stack
1) Execution Context stack
2) Program stack
3) Control stack
4) Runtime stack
5) Machine Stack

LECTURE 03: HOISTING IN JAVASCRIPT (VARIABLES AND FUNCTIONS)

CODE:-

```
var x=7;
function getName(){
    console.log("Namaste Javascript");
}

getName();
console.log(x);
```

OUTPUT:-

```
Namaste Javascript
7
```

BUT!!!

CODE:-

```
~~var x~~ getName();
console.log(x);

var x=7
function getName(){
    console.log("Namaste Javascript");
}
```

OUTPUT:

Namaste Javascript
undefined

- Most of the programming languages will give you error for the following code but Javascript is a bit different that's why it gave us this error.
- This phenomena is called "Hoisting"
- Hoisting in Javascript enables you to access variables & functions before you have initialised them.

If you make 'getName()' an arrow function it will just behave like a variable, and if you run the 2nd code it will give you error.

```
var getnam = () => {
    console.log("This is a func");
}
```

LECTURE 04 - HOW FUNCTIONS WORK IN JAVASCRIPT AND VARIABLE ENVIRONMENT:

Already learnt all of this in C language.
Same concepts about call stacks, function calls etc.

'WINDOW' AND 'this' Keyword:

window # is the global object created with the global execution context.

- The global object in the case of browsers is called window

→

- Chrome Javascript engine is known as V8. Mozilla Safari have their own separate Javascript engines.

- At the global level.
  this === window.
  > true

- Any code which is not inside a function, is called Global space
  All global variables/functions get attached to the global object.

  console.log(a) or console.log(this.a) or console.log(window.a) will return the same output.

LECTURE 06- UNDEFINED V/S NOT DEFINED IN JAVASCRIPT

"Undefined" in javascript is a special keyword, it takes up its own memory you can think of it like a placeholder, waiting for its value to be put in it.

Whereas "not defined" means that a variable doesn't exist in any of the programs.

- Javascript is a loosely typed language i.e. it doesnot attach its variables to any specific data type.

- Loosely typed languages are also called weakly typed language

- Never do this
  a = undefined;
  it not a good practice to do this although its not a mistake.

LECTURE 07- THE SCOPE CHAIN & SCOPE & LEXICAL ENVIRONMENT

CODE:

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

Explanation:

Although b doesnot exist inside the function but still the output of the program would be
>10

It means that you can access variables outside of the scope of a function (Also valid in the case of function within a function).

- But you cannot access a variable which is inside a function from the global scope.
- Pretty similar to C & C++

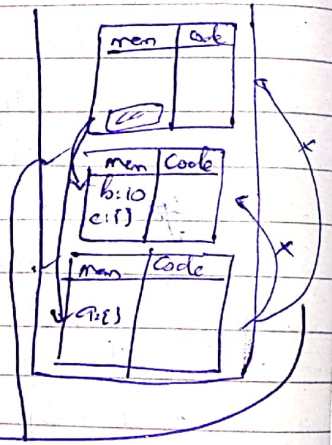
Scope is directly related to the lexical environment.

Lexical environment is the local memory along with the lexical environment of its parent.

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

function c is lexically inside a function. Lexical means in hierarchy

Can access down, but cannot access up.



This search of finding a variable is known as scope chain.

When a variable is encountered Javascript engine tries to find that variable in the local memory, if not found then it is searched in the memory of its parents & soon. But if its not found anywhere then javascript throws an error.

LECTURE 08: let & const in JS

Temporal Deadzone:

- "let" and "const" declarations are hoisted. but they are hoisted very differently than "var" declarations.
- "let" & "const" declaration are in the "Temporal dead zone" (more about it later).

CODE:-

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

Explanation: Trying to print b will give undefined but trying to print "a" will give you an error

"Cannot access "a" before initialisation"

- wherever you try to access a variable in a temporal deadzone it gives you a reference error.
- "let" variables are stored in a different memory space as compared to normal variables.
- And you cannot access the memory space of "let" & "const" before you have but some value in the variable.

Temporal dead zone:

It is the time between which the "let" variable is hoisted & when a value is put in the variable

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

will give error (bcoz a is not initialized)

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

will execute fine and give output 10 because this time "a" holds value of 10.

- let and const variable as stored in a separate space as compared to var variables (which are in GEC).

- You cannot access let & const variables by window.<let-var-name> you will get the output:
  > undefined.

- We can say that let is a level more strict than var.

```
var a=10;
var a=100;
```
is acceptable in the same scope

but

```
let a=10;
let a=100;
```
will throw you an error.

```
let a=10;
var a=100;
```
is also not acceptable

- "const" is even more strict than let
  You can declare a let variable and initialize it later in the program

```
let a;
// some code
a=10;
```
- is perfectly valid, but in the case of const the case it different.

- You must initialize a const variable when you declar it.
- and you cannot reassign another value to a const variable.

NOW:-
1) Use const as the 1st preference
2) Use let as the 2nd preference
3) Use var as the last preference

Always put the declaration & initialisation part at the start of the file.

LECTURE 9 - BLOCK SCOPE AND SHADOWING IN JAVASCRIPT:

Block: { }

This is a Block

- a block is also known as a compound statement.
- a block is used to combine multiple javascript statements into one group.

Q. WHY THE NEED TO GROUP JS STATEMENTS?

Ans. we group multiple statements together in a block so that we can use it where javascript expects only 1 statement

```
if (true)
{
  // multiple statements
}
```

Block

Normally if (true) true; is the basic "if statement" form.

BLOCK SCOPE

variables & functions accessible inside a block is called Block scope

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

look like this in the debugger in a brower

BLOCK
  b: undefined
  c: undefined
GLOBAL
  a: undefined

You can access "a" outside of the block, but you can access "b" & "c" outside the block scope.

Thus we say that:

"let and const are blocked scope."

Shadowing:

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

the "a" inside the block "shadows" the 'a' outside the block.

But

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

will also give you an output of 10 because the "a" inside the block modified the value of a.
And since "a" outside & "a" inside both point to the same memory location, the value of a is changed.

BUT!!!

"let" variable behave differently.

```
let b=100;
{ let b=10;
  console.log(b);
}
  console.log(b)
  >10
  >100
```

Because the "b" outside the scope & "b" inside the scope point to different memory locations.
In debugging mode the look like

BLOCK

b=10;

SCRIPT:

b=100;

Similar thing happens with const type of variable.
Shadowing also applies to functions.

- You can't shadow a let variable using a var variable but you can shadow var variable using a let variable.

- Block scope also follow lexical scope principles.

- Scope for arrow functions & normal functions are the same!!!

LECTURE 10- Closures in Javascript

CODE:

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

Closure: It basically means that a function binds together with its lexical environment

you can also return a function from within a function.

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

We know how would function y() behave inside x(); but now that x() contains function y(); how would it behave now?

Q. Now if you call z() somewhere inside your code, will it output? or not?

Ans Yes it will, this is where scope comes into the picture in javascript

Remember this

> "When functions are returned from other functions, they remember their lexical scope."

Instead of return y; you can also write

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

is the same as

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

Uses of Closures:

1. Module Design Pattern
- Currying
- Functions like once
- maintaining state in async world
- setTimeouts
- Iterators
- & many more....

LECTURE 11 - set Timeout + Closures interview Questions

CODE:

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

x();
```

OUTPUT:-

>1 (will be printed after 3secs)

But

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

OUTPUT:

> Hi

> 1 (after 3 secs)

Thus javascript doesnt wait for "1" to print, it Prints Hi first & then 1 is printed

2. How setTimeout works?

Ans setTimeout functions takes the function, stores it separately & attaches a timer to it. Then the function is executed when the timer completes.

For loop with var & let

```
for (var i=0; i<5; i++) {
    setTimeout(function(){
      console.log(i);
    }, i*1000);
}
```

will print	Because var is in global scope & every "i" of every function points to the same i, when the functions start to execute it will see the value of i as 5.
5	
5	
5	
5	
5	

```
for(let i=0; i<=5; ++i)
{
    setTimeout(function() {
        console.log(i)
    }, i*1000);
}
```

will print
1
2
3
4
5

Because "let" has Block scoped, thus each function call will creates its own copy of "i" instead of refering to the same memory location

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

will give
1
2
3
4
5

Coz a new copy of i will be passed in the function everytime

LECTURE 13 - FIRST CLASS FUNCTIONS ft. ANONYMOUS FUNCTIONS:

Function Statement: / Function Declaration

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

This is a function statement.

Function Expression

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

This is a function expression.

Function Statement v/s Function Expression:

You can call a function statement before its declaration, it works. But a function expression invoked before its declaration gives you an error.

Anonymous Function:

function(){}

- A function without its identity is called an anonymous function.
- But if you use it like this you will get an error, you can't use anonymous functions like this.

- Anonymous functions are used when the functions are used as values i.e. you assign a function to a variable only then you can use an anonymous function.

Named Function Expression:

Assigning a name to an anonymous function & storing it in a variable is called named function Expression.

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

b(); will output
> b called
but !!!

xyz(); will give you an error that xyz is not defined.
Because if you assign a function to a variable then a function's name is not assigned in the global scope, it is only available locally.

- So you can call it locally but you can't call a named function in a function expression by its name globally.

Difference Between Parameter & Arguments

```
var b = function (param1, param2){
    console.log("b called");
}

b(1,1);
```

Parameters

↳ argument

First class Functions:-

- You can pass a function as an argument inside another function

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

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

- You can also return a function from a function

- The ability of functions to be used as values i.e. passed as arguments & returned as values is called first class functions.

- This is also called first class citizens().

ARROW function (Part of ES6)
~~Later~~ it will be covered later

LECTURE 14: Call back functions in Javascript ft. Event Listeners:

Q. What is a callback function?

A function passed into another function is called call back function.

- Call bank functions give us the power to do asynchronous tasks in a single threaded synchronous language like Javascript.

```
function x(y){ }
x(function y(){ });
```

Q. Why use callback functions?

Ans. giving the responsibility to call a function to another function

For Eg: In the above code x can call y whenever it wants.

- SetTimeout() takes a call back functions and enables us to do asynchronous tasks.

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

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

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

OUTPUT:

> x

> y

> timer (after 5 seconds)

- Call stack is also called main thread, if any operation blocks the call stack its known as blocking the main thread.

- That's why we try to always avoid blocking our call stack & try async operations for things which take time.

Scope Demo with Event listeners:-

Did a bunch of things in the browser, saw event listeners working and a bunch of other things.

Garbage Collectors & Event Listeners:

→ Event listeners are heavy. They form closure with their parent functions & they cannot free up the extra memory.

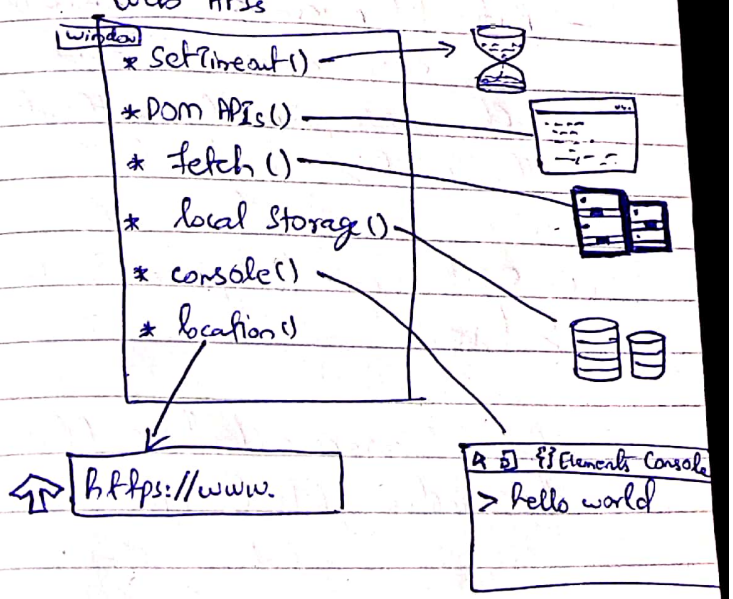
→ A lot a scope harnessing event listeners may slow the performance of your website.

→ Thus it is always a good idea to remove eventlisteners accordingly.

LECTURE 15- Asynchronous Javascript & EVENT LOOP from scratch

- The call stack of the JS engine doesnot have a timer.
  Timers enable us to execute commands after a specific period of time.

- To access various functions of a website you need the help of web APIs.



- setTimeout, DOM, console.log are not parts of Javascript

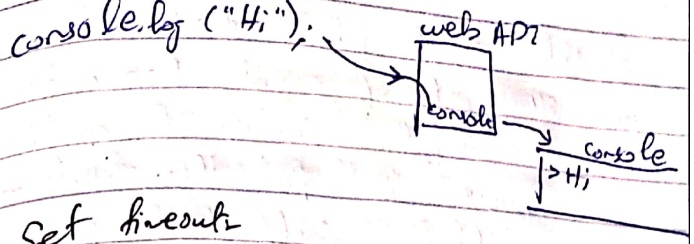
- These are all the functions of a browser which enable JS engine to perform all those functions.

- Browsers gives access to all those functionalities to Javascript engine through the "window" keyword.

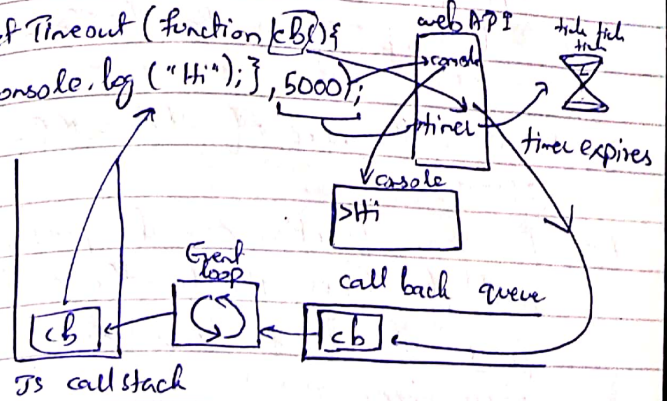
- But when you are in global scope you can access all these web APIs directly, without using the "window" keyword.

CODE: window.console.log === console.log
> true.

- Broaser wraps all those APIs in the window global object.



Set Timeout



- Event loop checks the callback queue, & put the function of "cb" into the call stack once the counter expires

DOM API: (Document object model):

document. → DOM API

Q. What is the job of the event loop?

Ans The job of event loop is to continuously monitor the call back queue and put the function in the call stack if it ever encounters ~~call~~ that the call back queue has function

Q. What is the need of the call back queue?

Ans A same function can be pushed in multiple times inside the call back queue ~~if~~ in the case of multiple occurances

Fetch () ~~function~~ WEB API call:

- fetch goes & requests an API call

- We pass a callback function ~~call~~ which will be executed when the promise of API call is received

- Its an webAPI which is used to make function calls.

Q. Microtask Queue:

Just like the call back queue, we also have a microtask queue

Microtask queue

- Its exactly similar to the call back queue but it has higher priority

- Whatever functions come inside the microtask queue are executed first, while functions inside the callback queue get executed later.

- "Event loop" only pushes task from microtask & call back queues into the call stack when it is empty. If there are some functions already in the call stack, then the functions inside the microtask/call stack queue will have to wait

- If there is 1 task in both microtask & call back queue then call stack will get called first.

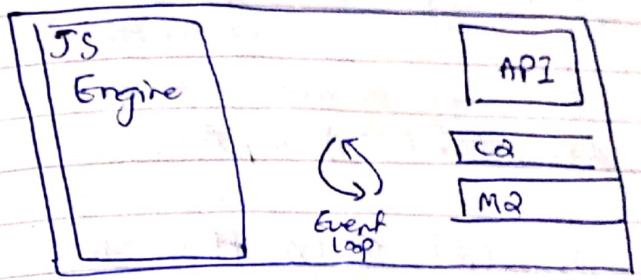
Q. What goes inside the microtask queue

All the callback functions which come through ~~promises~~ promises, go inside the microtask queue.

Promises + mutation observer go inside the microtask queue.

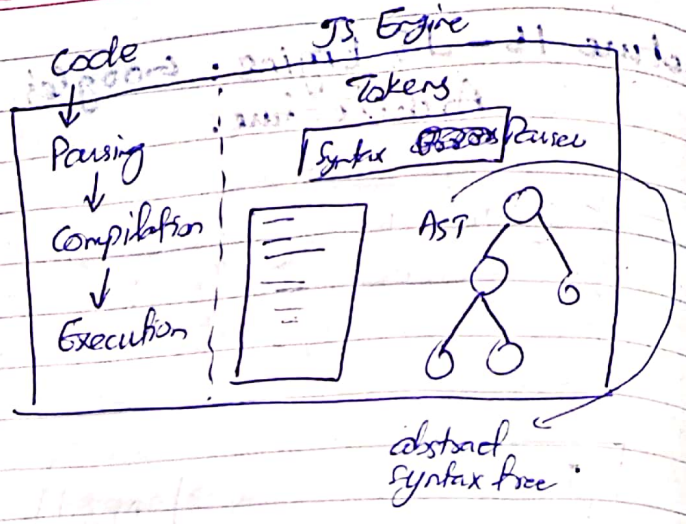
Lecture 16 - JS Engine: Googles V8 Architecture.

- Javascript Runtime Enviroment is like a big container that has all the things required to run javascript code.



Javascript Runtime Enviroment (JRE)

- Node.js is an opensource Javascript runtime enviroment.
- JRE enables browsers to run Javascript.
- JS Engine is not a machine, its just a normal piece of code.



1) Just in time (JIT) Compilation:

Interpreted Execution In interpreted languages, an interpreter executes the code line by line & doesn't know what will come in the next line (Its More FAST)

Compiled Execution In this case the whole code is compiled before execution, & then execution starts (Its more efficient since the original code is first converted into an optimized format.)

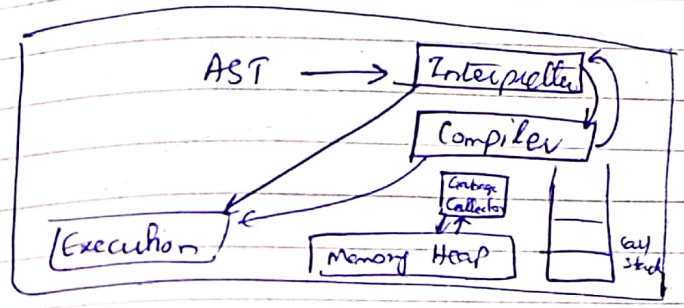
Also Read about Mark & sweep algorithm

- Java Script can behave as an interpretted as well as a compiled language. Everything depends on the Javascript engine.

&

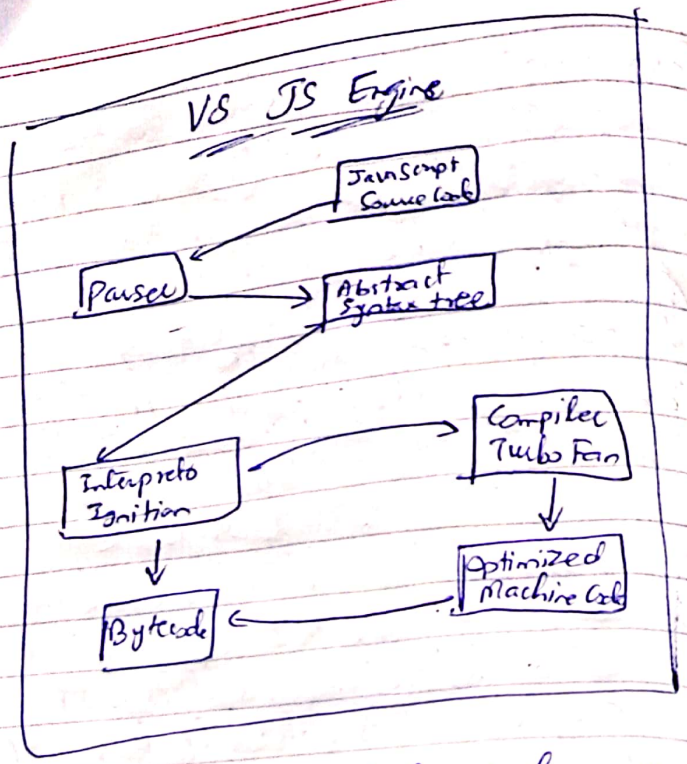
- Javascript was initially supposed to be an interpretted language at the time of its creation.

- Most modern JS engines use an interpretter & and a compiler both. hence its called JIT compiled. Its like the best of both worlds.



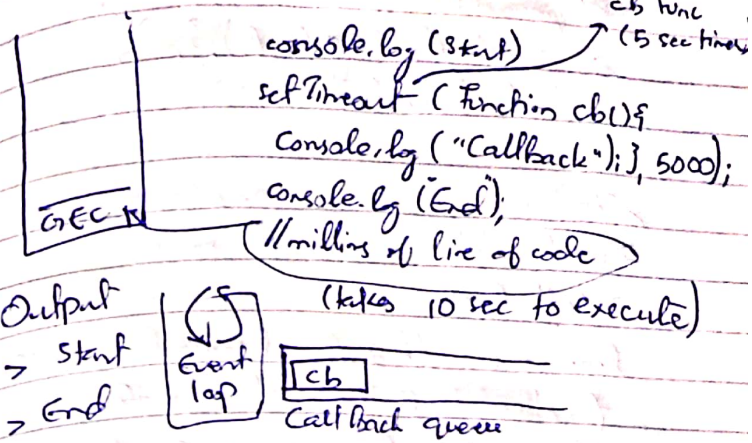
Interpretter starts executing the code but also takes the help of compiler to optimize the code simultaneously.

- This lecture just generically explains things the actual implementations may vary.



V8's Garbage collector is known as Orinoco.

LECTURE 17 - TRUST ISSUES WITH JAVASCRIPT:



- The timeout expires after 5s but the code needs 10s for execution, now the question arises that when will cb function execute?
- It will execute after 10s bcoz the eventloop will keep waiting for global execution context to move out of the call stack.

- So ~~set~~ setTimeout only guarantees that it will atleast wait for the prescribed time, but it can wait more too.

```
console.log("Start");
setTimeout (function cb(){
    console.log ("Call Back");}, 0);
console.log ("End");
```

>Start
>End
>Call Back

Bcoz even though setTimeout is 0, the function will have to go through the callback queue & that takes time.

LECTURE 18- Higher Order Function ft. functional Programming

Higher order Functions:

A function which takes another function as an argument or return a function as an argument is called a higher order function.

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

In this case y() is a higher order function, & x() is the call back function.

DRY (Don't Repeat Yourself):

If you are copy pasting code over & over again you are definely ~~@~~ not writing the best code.

Functional programming enables you to pass the logic (in the form of a function) as an argument inside a function.

Consider an example:

- You are given an array of numbers containing radiuses of circles. & you have to calculate the area, circumference & diameter.

- Now instead of making separate function for each calculation, we can create a generic function & then pass the logic (in the form of a function) as an argument.

CODE:

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
    return MATH.PI * radius * 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));
```

Using this approach you can write logic for circumference & diameter & also pass it as argument.

- Functional Programming paradigm requires developers to think of logic in the form of functions.

- Functional programming is a huge topic in itself, but this lectures covers only some of its concepts.

Q. How to make a function availa
to all the prototypes?

Ans You can enable a function
to be available to all the
arrays by using <Datatype>.prototy

Array.prototype.calculate = //there
is same

LECTURE 19 - map, filter & reduce

01) Map function:

Map function is used to
transform an array.

```
const arr : [5, 1, 3, 2, 6]
// Double - [10, 2, 6, 4, 12]
// Triple - [15, 3, 9, 6, 18]
```

Transforming an array:

```
const output = arr.map(double);
function double(x) {
    return x*2; }
```

- Internally, the map() function will run the double() function on each & every element of the arry.
- This is also valid.

```
const output = arr.map(function bracket
{ return x.toString(2);})
console.log(output);
```

You can also pass an anonymous function as a callback function in the form of arrow function.

~~const output = arr.map(function~~

argument

```
const output = arr.map((x) => {
    return x.toString(2);});
console.log(output);
```

x — x — x

filter() function:-

```
const arr = [5, 1, 3, 2, 6];
// filter odd values.

const output = arr.filter(isOdd);

function isOdd(x){
    return x%2;
}
console.log(output)
```

> 5, 1, 3,

```
output greaterthan4 = arr.map((x) =>
                                x>4);
console.log(greaterthan4);
```

x — x — x

Reduce() function:-

Reduce() function is used at an array and come up with a single value out of them.

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

CODE:- find the sum of all elements in the array.

```
const output = arr.reduce(function,
```

5,1,2,3,6 → (acc, curr)

final value

```
{   acc = acc+curr;
    return acc; }, 0);
```

initial value of accumulator.

x — x — x

THE END (for now)