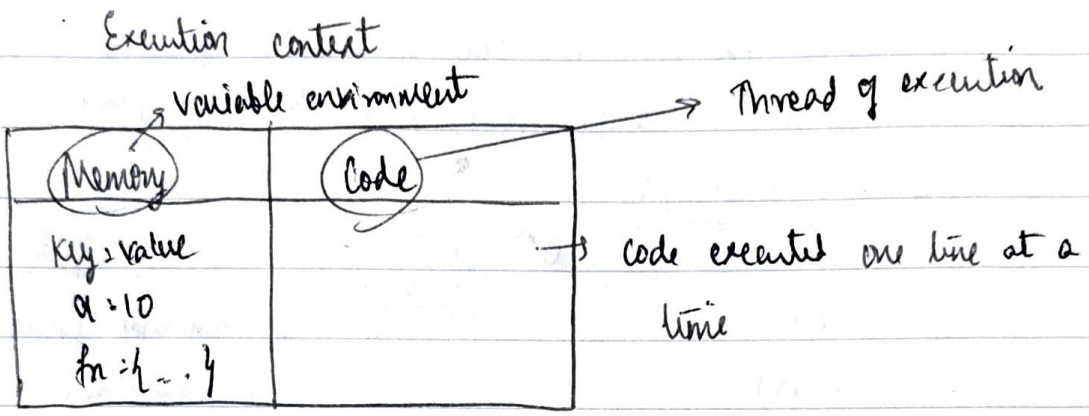
Javascript

→ Everything in JS happens inside an execution context.



→ JS is a synchronous single-threaded language — can exec one command at a type and in a specific order.

- What happens when you run a program?

1) Memory allocation Phase
2) Code Execution phase

When a fn is called a new execution context is formed inside code when return is encountered it returns back to the caller execution context then the created execution fn gets deleted

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

Once the whole program is done, the entire global execution context is deleted.

JS has its own call stack.

→ Call stack is populated with global execution context and a fn is called even that is ~~fonnn~~ is added to the stack

→ Call Stack maintains the order of execution of execution contexts

Hoisting in Javascript → Javascript's default behaviour of moving declarations to the top.

→ other programs throw error

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

Output

"Namaste Javascript"
undefined

Now if you remove x = 7 we get, through reference x is not defined.

console.log(getName) → prints the entire function

If we put ; on top also it prints the entire function. How? Why?

With arrow functions it will behave like a variable.

var x = 1;
a(); b();
function

```
var x = 1;
a(); b();
function a() {
  var x = 10;
  console.log(x);
}

function b() {
  var x = 100;
  console.log(x);
}
```

→ Output → 10, 100, 1

Shortest JS program

Global object

→ Empty program but it has some window → created by JS engine. JS engine also creates a this keyword. Global level this points to window.

this === window → true

→ Any code that's not written inside a fn is called global space. This global space variables will be inside the window.

Undefined vs not defined

undefined → takes memory, will act like a placeholder

JS → loosely typed language.
↳ does not attach its variable to any particular datatype

Lexical Environment (Scope, scope chain)

Scope → where we can acces a particular variable.
Lexical → hierarchical.

Let & const declarations are hoisted but different (wisting than var.
↳ they are in the temporal dead zone for the time being.

~~Let a = 10~~
~~var~~

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

```
console.log(a)
let a = 10;
var b = 100;
```
→ Cannot access 'a' before initialization.

```
let a = 10;
console.log(a);
var b = 100;
```
→ Stored in a seperate memory space and not global.

→ present in global

Temporal dead zone
↳ memories allotted
console.log(a)

Temporal dead zone
→ memories allotted

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

→ error cannot access.

Phase from hoisting to getting some value is called temporal dead zone.

But if we do console.log(x) → x is not defined.

We cannot do something like re-declaration, we get a syntax error;

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

duplication of let for same variable in the same scope will give syntax error

↓ possible in var but not let.

→ const declaration behaves similar to let but is even more stricter.

```
let a;
const b = 1000;
a = 10;
console.log(a)
```

} completely fine but

```
let a;
const b;
b = 1000;
a = 10;
```

} → syntax error ↓ Missing initializer in const declaration

```
let a = 1900;
const b = 1000;
i = 10000;
```
} → Type Error : Assignment to a const

```
console.log(a);
let a = 1900;
```
} → Reference error.

const → whenever we dont want to change the value.
let → want run into unexpected errors like undefined as some error could pop-up.
But use these consciously.

→ But way to avoid temporal dead zones is, declare and initialize at the top always. So that the zone window is shrinked to 0.

Block Scope

```
{
}
```
→ Block

Block also known as compound statement.

combines multiple statements into one.

What is block scope?
What all variables & function we can access inside this block.

to a constant

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

→ Hoisted in a Block memory space and 'a' is hoisted in the global scope
Hence they say let & const are block scoped.

Hence outside this we can't access let and const

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

This variable is shadowing the a=100 variable on

→ we get a 10 again it shadowed and modified because it is in the global scope.

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

→ In Script scope

→ In block scope

But 100 comes here

↳ Same thing happens with const

CLOSURES

function x() {
var a = 7;

Even in function scope similar thing happens

Illegal Shadowing

```
let a = 20;
{
    var a = 20;
}
```

→ 'a' has already been declared Syntax Error.

→ But this inside fn it'll be fine.

```
var a = 20;
{
    let a = 20;
}
```

→ This is perfectly alright

→ Block scope also follows lexical scope (order & one inside another) lexical scope chain pattern.

~~const a~~

→ Arrow function is similar to function Keyword function.

CLOSURES

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

Output: 7

→ This is what closure is

function + lexical scope = closure

Closure = (x)
a: 7

→ functions bundled together (enclosed) with references to its surrounding state.
(the lexical environment).

→ We can assign functions to a variable as well in Javascript.
→ We can pass a function as a parameter as well in JS.
→ We can also return a function.

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

→ What is y? It is the whole function code.

After returning from x, it won't be in the call stack.

var z = x()
console.log(z)

z(); → will try to find a, but that scope is gone right?
what will happen?
It will print 7 correctly. Here closures come into play.

So instead they write like this:

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

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

→ O/p not 7 but 100.

function statement aka function a() {
}

Function Statement aka function declaration

```
function a() {

}
```

→ function statement

Function expression

```
var b = function() {

}
```

→ Function acts like a value.

Major difference between these 2 is hoisting.

Before the above functions if we do

```
a();
b();
```

a is called but b gives a TypeError.

Because b is a variable and it will be undefined, so when a function is undefined it returns an error saying b is not a fn.

Anonymous function

A function without a name is called anonymous function.

They do not have their own identity.

```
function() {

}
```

→ Similar to fn statement but no name

→ Gives Syntax Error → function statement requires a name.

Anonymous functions when functions are used like values.

like

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

```
var b = function xyz() {
    ...
}
xyz() →
```

Throws an error

Reference error.

Values we pass inside a calling function is called an arguments.

b(2,3)

labels / identifiers in function statement which gets these values are called parameters.

First Class Functions

We can pass functions inside other functions as arguments and we can receive them as parameters.

We can also return an anonymous function from a function

→ Ability to use functions as values and can be passed as arguments and can return functions are called as first class functions.

like values.

First class citizens ==first class functions
↳ used interchangeably.

Arrow functions

Callback functions in Javascript → Synchronous single threaded but because of callbacks we can do async things.

Ex
function x (y) { → opto x when to call y.

y
x(function y(){ });

Lets see how it is used asynchronously. → whatever to be done after 5000

setTimeout (function () { }, 5000) ← 5000ms = 5sec

console.log ("timer")

Output :
x
y
timer

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

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

→ Whatever is executed is executed through call stack only.

→ If any operation blocks this call stack, it is called blocking main thread.

→ setTimeout takes a callback and executes it sometimes later.

EventListeners

```
document.getElementById("clickme").addEventListener("click", function() {
});
```

callback function

When the click event happens it will call this callback function. It will magically appear in our call stack.

Closures along with Event Listeners to know how many times button was clicked

closure (attachEventListener)
count: 4

```
let count = 0;
```

In the end inside fn do ++count.

forms a closure with count & attachEventListener

```
function attachEventListener() {
    let count = 0;
    document.getElementById("clickme").addEventListener("click", function xyz() { console.log("Button clicked", ++count); });
}

attachEventListener();
```

Garbage collection & removeEventListener

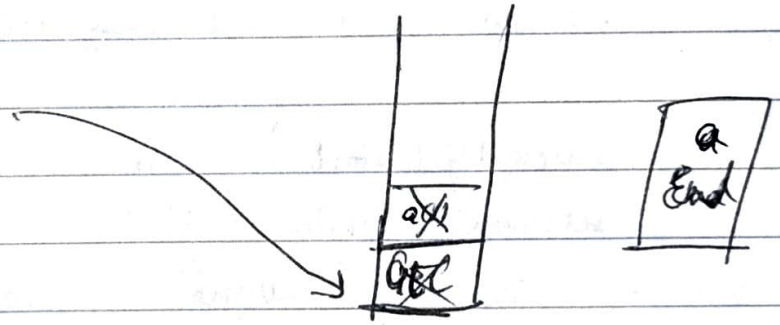
Garbage collection & removeEventListener

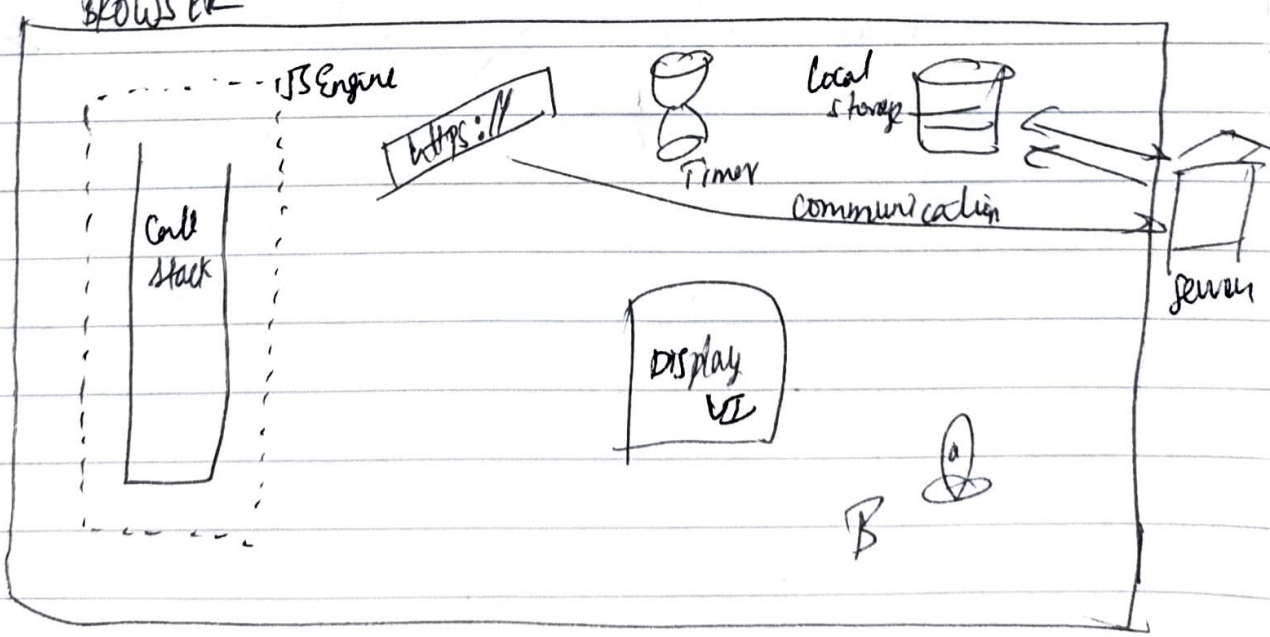
→ EventListeners are heavy and it takes a lot of memory.
→ Hence EventListeners are removed when we don't use it.
→ After removing eventlistener then all the variables held by closure will be added to garbage collection.

Event LOOPS

JS synchronous single threaded language - It has only one call stack present inside JS Engine.

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





Now our JS engine/callstack needs some to their superpowers do
Some way should be there for connection.

WebAPIs

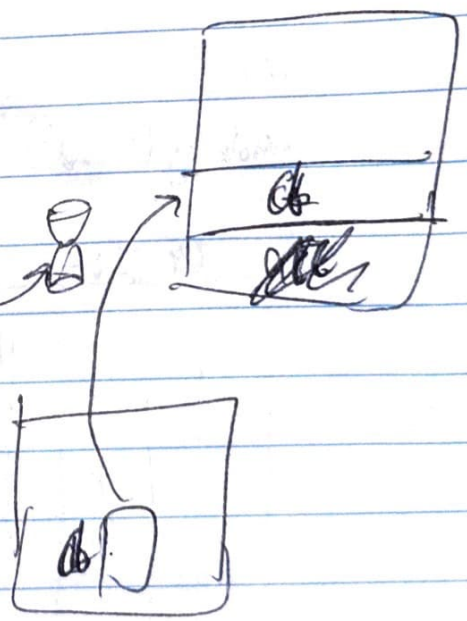
WINDOW

setTimeOut()
DOM APIs (document. etc
fetch() (connection to external servers)
LocalStorage → part of browser
console
Location & lot more

present in global sco
So we dont have
to use the
window keyword.

We can get it inside using window in global object.

```
console.log("start");
setTimeout(function cb() {
    console.log("callback");
}, 5000);
console.log("End");
```



Start
End

After timer
done

Event Loop and callback queue

→ 5000 ms timer was running, as soon as timer expires it cannot directly go to the call-stack. It goes through the callback queue.

→ Job of eventloop is to check the call-back queue and put it back to the callstack. So Event loop acts as a gate keeper.

```
console.log("start");
document.getElementById("btn").addEventListener("click", function cb()
    { console.log("callback"); });
console.log("end");
```

registering callback to click event.

WebAPIs env.

> start
> End
> (when user clicks) then it comes "Callback"

If callstack empty & something waiting in callback queue then eventloop pushes it to the callstack.

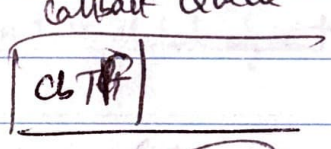
Imp Qn, Why do we need a queue?

Suppose a user click 5-6 times continously.

→ fetch() works a different way

```
console.log("start");
setTimeout(function cbT() {
        console.log("CB setTimeout"); }, 5000);
fetch("https://api.netflix.com").then(function cbF() {
        console.log("CB Netflix"); });
console.log("End");
```

→ fetch requests api call, it returns a promise and we have to pass a CB function, which will be executed once this promise is resolved. ↔ Meaning → When data is received we execute callback function.

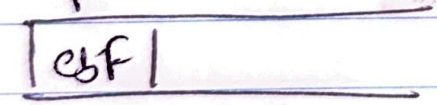


> Start
> End
>

Suppose Netflix server are fast so callback function of this should come to the queue first, right? But this doesn't happen.

So here MicroTask queue comes to picture.

→ Similar to callback Q but has higher priority.



One 5000ms is done

MicroTask Queue has higher priority. So cbF pushed to callstack & then once cbF is done ← popped. cbT is executed in callstack.

Every browser has JRE.

Question is who can come inside Microtask queue ?

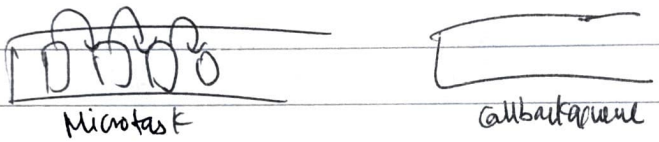
→ All callback function which comes from promises

→ "Mutation Observer" → (If there is some mutation in DOM T

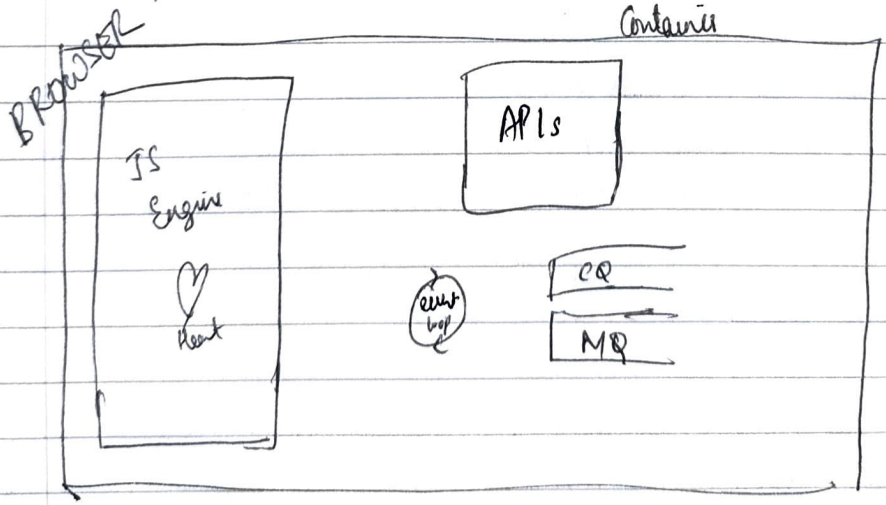
All other callback functions goes to callback queue.

⟹ Callback queue also called Task Q

** If Microtask recursion keeps happening then callback queue wont be able to execute this is called Starvation



Javascript Runtime environment



- Every browser has JRE.

→ Node.js also has Javascript run-time environment

↳ open-source JS runtime → means it has everything to run a JS piece of code.

↳ can run outside the browser.

console API present both in API & node.js

→ Only APIs can be implemented differently.

→ V8 : JS engine used in Chrome and node.js

ECMAScript → governing body for Javascript.

High Order functions

A function which takes another function as an argument or returns a function is called as Higher Order function.

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

```
function y(x) {
    x();
}
```
} → Higher order function.

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

const calculateArea = function (radius) {
    const output = [];
    for (let i = 0; i < radius.i++) {
        output.push(Math.PI * radius[i] * radius[i]);
    }
    return output;
};
calculateArea(radius);
```

→ Now if we want to calculate circumference then we write a similar function like above, right?

→ Say, to also calculate diameter, again one more function

DRY principle → Don't repeat yourself

→ So we have to try and make everything in one function.
So what we can do?

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

} call back function

```
const calculate = function (radius, logic){
    const output = [];
    for(let i=0; i<radius.length; i++) {
        output.push(logic(radius[i]));
    }
    return output;
};
```

Higher order functions

→ Good code (Beauty of functional programming).

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

so now we just write circumference fn and do call

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

Reusability
Modularity.

exactly similar to the map function

```
radius.map(area)
```

Map, filter and Reduce

Map, filter, Reduce are higher order functions in Javascript.

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

Map is used for transformation of above array to another array

Eg:- Double, Triple, Binary.

```
[].map()
const output = arr.map(double);
```

we can pass function

```
function double(x){
    return x*2;
}
```

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

Filter function

Filter function used to filter the values inside an array.

Filter all values >4 etc etc.

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

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

Reduce

Reduce function is such that you take all values from an array and convert it into a single value.

Eg:- sum (or) max

```
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))
```

const output = arr.reduce (function (acc, curr) { });

Reduce → iterates through each & every element of the array.

curr represents the value here - Accumulator is used to accumulate the results used in the array.

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

1st argument fn,
2nd argument is initial value of acc.

This fun will be called for each and every value

To find the max inside an array

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

Callback Hell

0. Callback Hell
1. Inversion of Control

```
const cart = ["shoes", "pants", "kurta"];
api.createOrder(cart, function(){
    api.proceedToPayment()
});
```

Call this callback function after everything is done in createOrder.

Now say we want another page which has to show the order summary after 1. creating Order, 2. Payment & 3. Order summary.

```
api.calculateOrder(cart, function(){
    api.proceedToPayment(function(){
        api.showOrderSummary()
    })
});
```

But, the problem is :-

Say we want to update Wallet then again

So here we are falling the callback hell. This type of code is not readable and maintainable.
This structure is also called as the pyramid of doom.

Inversion of Control

↳ Another problem we see when using callbacks.

We are ~~blindly~~ blindly trusting createOrder API. What if our callback fn was never called. What if it gets called twice.
Very risky thing which needs to be controlled.

PROMISES

Promises are used to handle async operations in JS.
Say e-commerce websites.
So, before promises, it was like that code callback hell example.

```
const promise = createOrder(cart);

// { data: undefined }
```

Other stuffs below here keeps on executing.
At later point once the function execution is done the orderDetails gets filled
{ data : orderDetails }

Now lets attach callback function to this promise object.

```
promise.then(function (orderId) {
        proceedToPayment(orderId);
    });
```

Promises gives trust and guarantee that it calls the callback function once its promise object has data.

Lets exactly see how promise objects works :-

```
const GITHUB_API = "https://api.github.com/users/prodymro(26
~~const fetch~~
const user = fetch(GITHUB_API);
```

→ async operation will take some time.

promise object returned.

Prototype : Promise
PromiseState : "pending"
PromiseResult : undefined

(pending, fulfilled, Rejected)

Whatever data fetch returns will be stored in Result & Promise State initially is in pending state. After its executed it changes to fulfilled state.

```
console.log(user)
```

→ user.then

this promise object.

```
user.then(function(data){
    console.log(data);
});
```

→ Promise objects are immutable.

⇒ What is Promise in a JS ?
Placeholder which will be filled later.
Container for a future value.

→ A promise is an object representing eventual completion or failure of an asynchronous operation.

→ So IOC is resolved here.

⇒ Promise chaining helps us resolve the callback hell as well

Before

```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart, function(orderId){
    proceedToPayment(orderId, function(paymentInfo){
        showOrderSummary(paymentInfo, function(){
            updateWalletBalance();
        });
    });
});
```

Creating a

After :

```
createOrder(cart).then(function(orderId){
    proceedToPayment(orderId);}).then(function(orderId){
        showOrderSummary(paymentInfo);})
```

↓

Promise chaining

→ One more imp thing is Data piping - Like we want data to flow
Eg :- We want data/response of createOrder to flow into proceedToPayment, then the data/response of that function to pass down the chain
So for that we return a promise.

```
createOrder(cart).then(function(orderId){
    return proceedToPayment(orderId);})
  .then(function(paymentInfo){
    return showOrderSummary(paymentInfo);
  })
  .then(function(paymentInfo){
    return updateWalletBalance(paymentInfo);});
```

Creating a Promise