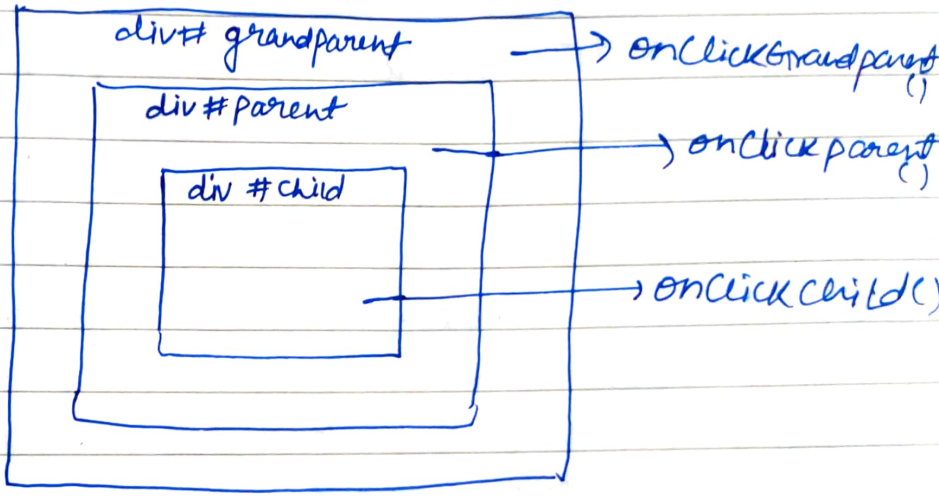
classmate
Date
Page

Event Bubbling and capturing → also known as event trickling

Bath are two ways of event propogation in the DOM tree.



In case of event bubbling
If you click on child, order of execution -
(i) onClickChild()
(ii) onClickparent()
(iii) onClickGrandparent

In case of event capturing
If you click on child, order of execution -
(i) onClickGrandparent
(ii) onClickParent
(iii) onClickChild

onclick of Parent

event bubbling
(i) onClickParent()
(ii) onClickGrandParent

event capturing
(i) onClickGrandParent
(ii) onClickParent

By default, browser use Event Bubbling.
TO override this to use Event Capturing,
Pass third variable as true.

How to use bubbling and capturing -
you can add 3rd argument to
EventListener 'useCapture' boolean
flag:

```
document.querySelector("#child")
.addEventListener('click', () => {
  console.log("child click");
}, false);
```

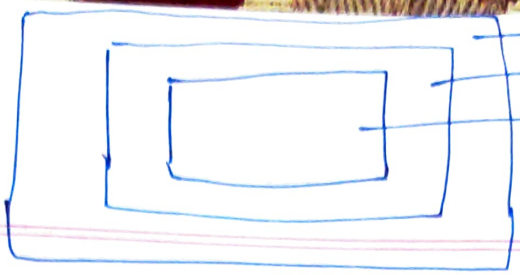
Event capturing

```
document.querySelector("#grandparent")
.addEventListener('click', () => {
  console.log("Grandparent clicked");
}, true);

document.querySelector("#parent")
.addEventListener('click', () => {
  console.log("parent clicked");
}, true);

document.querySelector("#child")
.addEventListener('click', () => {
  console.log("child clicked");
}, true);
```

If click on parent
(i) grand Parent clicked
(ii) Parent clicked

If clicked on grandP
(i) Grand Parent clicked

If clicked on child
(i) GrandPar
(ii) Parent
(iii) Child



(1) If GrandParent → true // capturing
Parent → False // Bubbling
child → true // capturing

on click of child—

(i) GrandParent clicked
(ii) child clicked
(iii) Parent clicked

According to W3C, First ~~the~~ Event Capturing happens then bubbling out happens

when the Event was coming capturing down, because of true in GrandParent, GrandParent clicked called first, then It went to parent and flag was false so parent won't come in capturing cycle so parent callback was not called then it move to child and saw true, the capturing happened and child callback was called. Then the cycle of capturing is completed. Now bubbling cycle starts. Now the parentClicked was called.

What will happen if

GrandParent → true
Parent → false
Child → false

When click on Child
→ Grandparent clicked
→ Child clicked
→ Parent clicked

When clicked on parent
→ Grandparent clicked
→ Parent clicked

How to stop propogation up and down the hierarchy?

Sol^n → e.stopPropagation

```
document.querySelector("#grandparent")
.addEventListener('click', ()=>{
  console.log("Grandparent clicked");
}, false);

document.querySelector("#parent")
.addEventListener('click', (e) => {
  console.log("parent clicked");
  e.stopPropogetion();
}, false);

document.querySelector("#child")
.addEventListener('click', ()=>{
  console.log("child clicked");
}, false);
```

When click on child

→ child clicked

→ Parent clicked

If you write e.stopPropogation in child

When click on child

→ child clicked

```
document.querySelector("#grandParent")
.addEventListener('click', () => {
console.log("GrandParent clicked");
}, true);

document.querySelector("#parent")
.addEventListener('click', () => {
console.log("Parent clicked");
}, true);

document.querySelector("#child")
.addEventListener('click', (e) => {
console.log("child clicked");
e.stopPropogation();
}, true);
```

When clicked on child

→ GrandParent clicked
→ Parent clicked
→ child clicked

If e.stopPropogation is on grandParent only

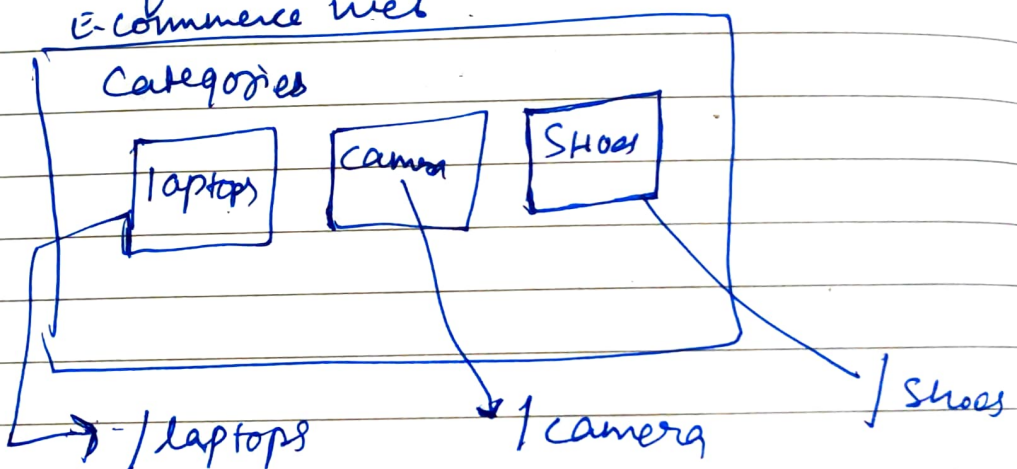
When clicked on child

→ GrandParent clicked.

Event Deligation

Event Deligation is a technique of handling events in our webpage in a better way. It is only possible because of Event bubbling.

Sometimes our webpage has so many event handlers hanging around. It causes performance issues. With Event deligation we can fix that problem



one way is to add EventListeners to each categories i.e

```
document.querySelector('#laptop').
addEventListener('click', () => {
});
```

similarly for camera & shoes

What is better way?

Event deligation says that instead of attaching event handlers to each & every child elements, we should rather attach event handlers to parent of these elements.

```
<div>
<ul id="category">
    <li id="laptops">laptops</li>
    <li id="cameras">cameras</li>
    <li id="shoes">shoes</li>
</ul>
</div>
```

```
document.querySelector("#category")
.addEventListener('click', (e) => {
 if (e.target.tagName == 'LI') {
 window.location.href = "/" + e.target.id
 }
});
```

Another example of Event deligation

```
<div id="form">
<input type="text" id="name" data-uppercase>
<input type="text id="pan">
<input type="text" id="mobile">
</div>
```

```
document.querySelector("#form")
.addEventListener('keyup', (e)=>{
if(e.target.dataset.uppercase != undefined){
e.target.value= e.target.value.toUpperCase()
}
})
```

ABHINAV	abhinav	abhinav

Limitations of Event Deligation

1) All events are not bubbled up.
   Ex- blur, focus, window resize
2) if e.stopPropogation is used in child, then events are not bubbled up.

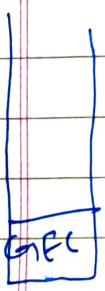
Event Loop

Callback Queue
Microtask queue

JS is a synchronous single-threaded language.
It has one call stack and it can do one thing at a time.
The ~~class~~ call stack is present inside JS Engine.
All the code of JS executes inside call stack.

How this function executes?]

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

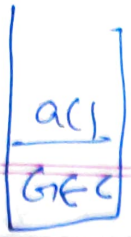


1→ Whenever any JS program runs, it create Global execution context (GEC). It gets pushed inside call stack.

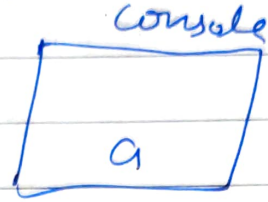
Call Stack 2→ First line is function defenition. So no change in call stack a will be allocate memory & function will be stored.

as soon as control reach to a(), execution context will be created for function a and pushed inside call stack

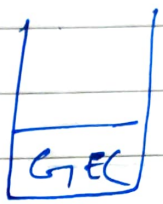
call stack



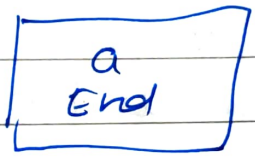
Content inside a() will be executed line by line



after printing a, Execution context of a finishes



Now "End" will be console



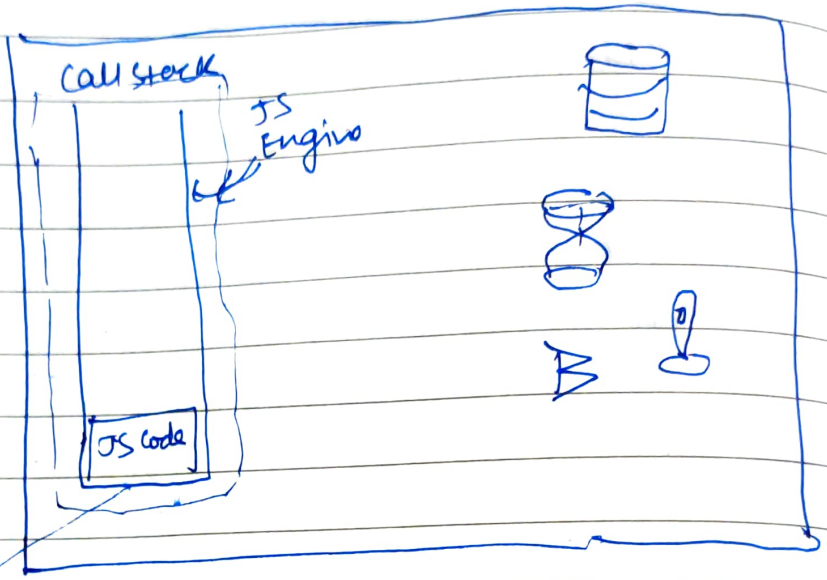
and GEC will be finished

* The main job of call stack is executes anything which comes inside it. It waits for noone.

What if we need to wait for some time?

* Suppose we have to keep track of time to execute any function after 2 sec, we will need some superpower i.e superpower of timer.

How we can get those superpowers?



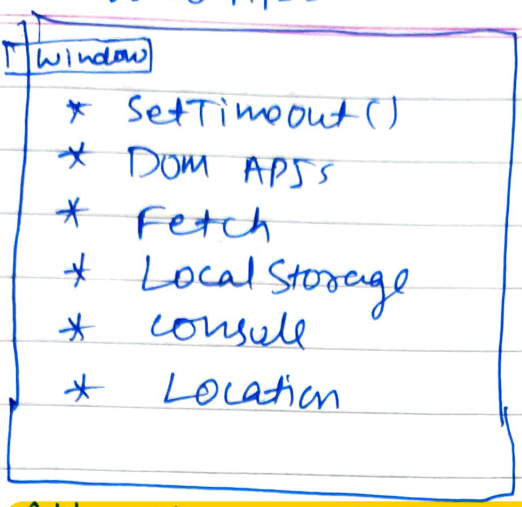
Call Stack is inside JS Engine.
JS Engine is inside Browser.
Our programs (JS code) runs inside Call Stack.

Apart from JS Engine browser has many more things -
Local Storage, timer, Bluetooth, location.

If you need these superpower, you need to have some connection. JS Engine needs to have some ways to access those superpowers.
How?

via Web APIs

Web APIs



All these things are not a part of JS. they belongs to browser.

Browser allow us to use all these superpowers inside JS Engine using Web APIs:

* If you want to use timer in your code - Browser gives SetTimeout API
* If you want to use Dom tree - Browser gives DOM APIs etc.

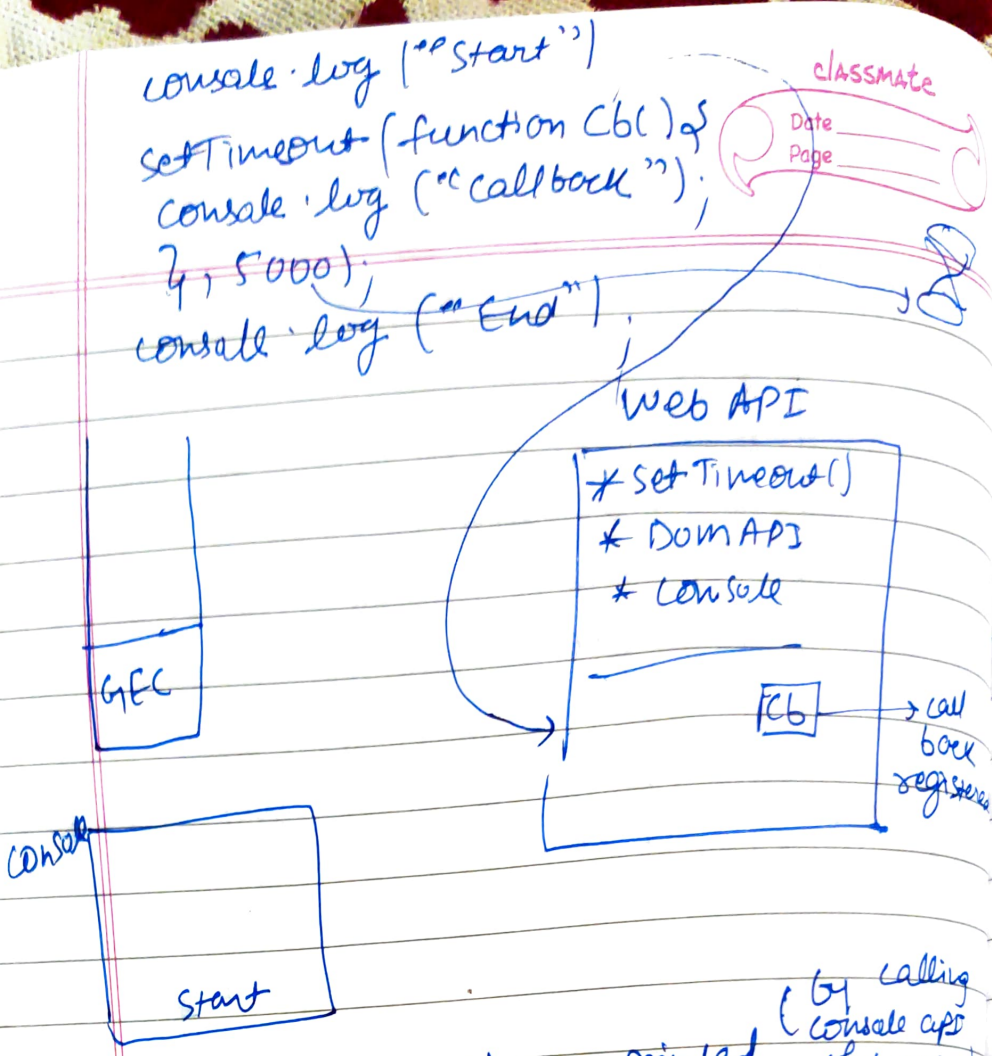
We use all these superpowers through a keyword "window".

window.SetTimeOut()
window.fetch().

you can able to access without writing window. explecitely bcoz window is global object & all those are present in global object or global scope.

Browser wraps all these superpower apis into a global object "window" and gives access of this window object to ~~class~~ JS Engine or call stack.

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



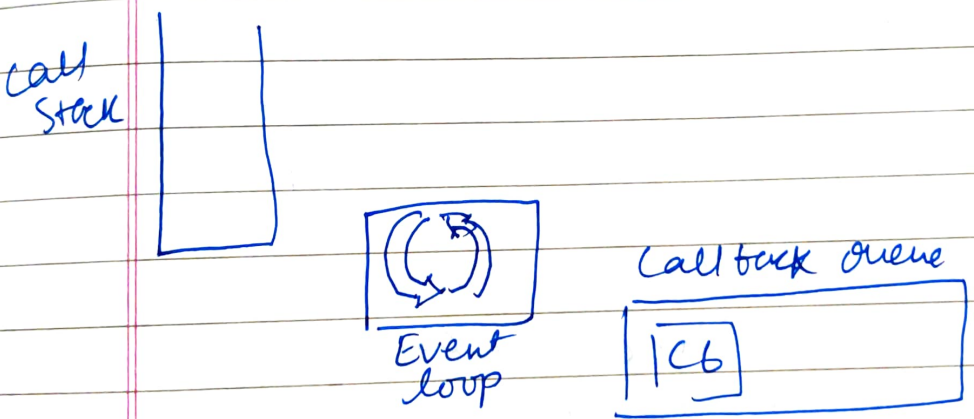
* The first line will be printed (By calling console api of browser)
* the setTimeout will call setTimeout Web API and register callback function and start the timer of 5000. and code move to next line.

It will do console. The program executed but our timer is still running. Our global execution context finishes and our call stack is empty

As soon as timer end, the callback function cb needs to be executed. But as we know all of JS code executes inside call stack, somehow we need cb inside call stack.

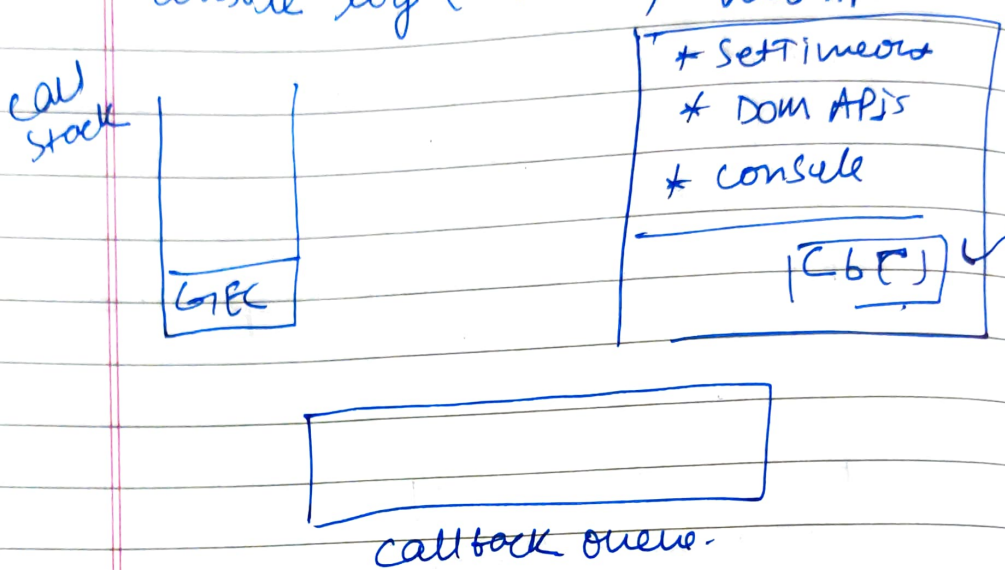
Now comes Event loop and callback queue

After timer expires, the cb goes to callback queue.

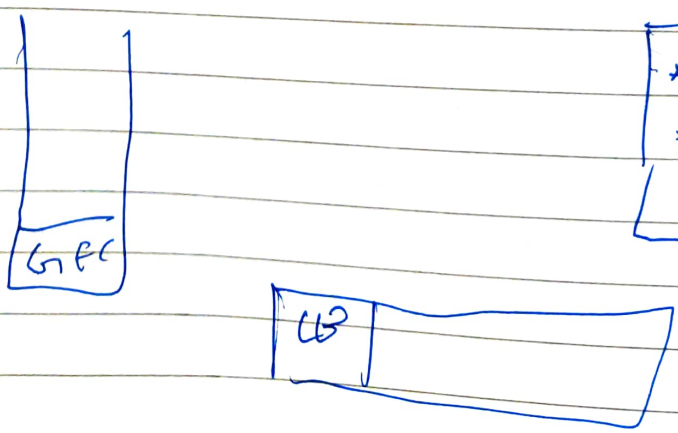


Event loop act as a gatekeeper and continuously keep checking if we have anything in Callback queue. If we have, It pushes it inside call stack it and only it Call stack is Empty the call stack creates Execution context of cb and executes it.

```
console.log("start");
document.getElementById("btn")
.addEventListener("click", function cb(){
  console.log("call back");
});
console.log("End");
```



cb() is waiting inside web API. As soon as you click the button, It get pushed inside callback queue and wait for call stack to get empty so that event loop will push cb from callback queue to call stack so that it can get executed

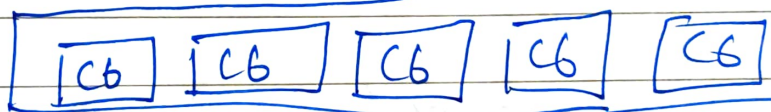


Event loop has only one job — continuously monitor call stack and callback queue. As soon as call stack is empty and there are any function in callback queue, it takes that function & push inside callstack and call stack executes it quickly

And microtask queue

Why do we need callback back queue? Event loop would have directly picked callback from web api and pushed to call stack

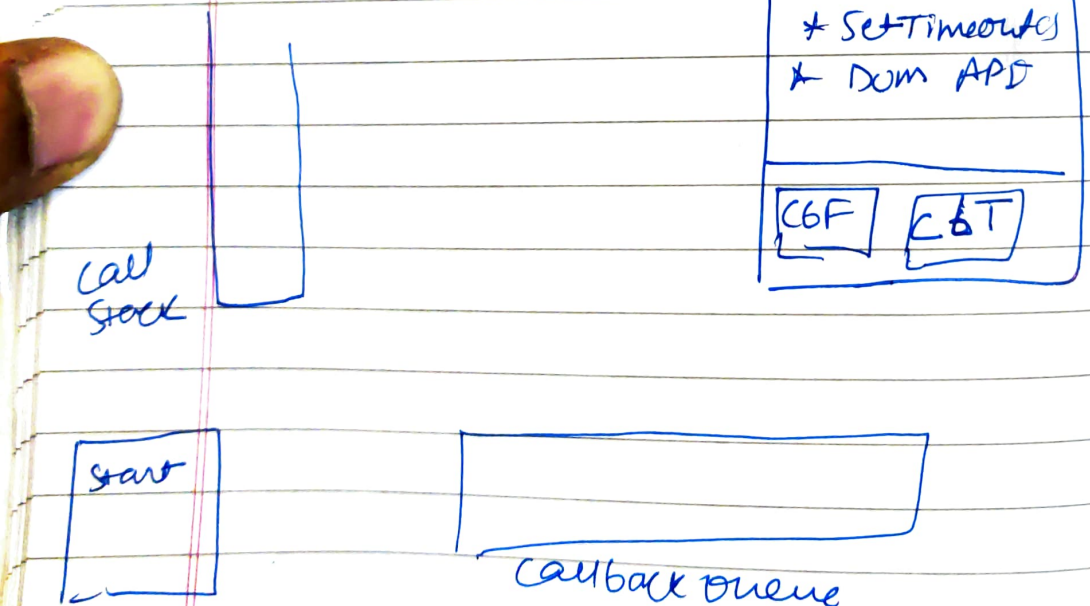
Soln — Suppose user click on button 5-6 times continuously, 5 6 callback will be pushed in callback queue and will executes sequencially.



Callback queue.

Fetch Api works differently from setTimeout & DOM and other APIs.

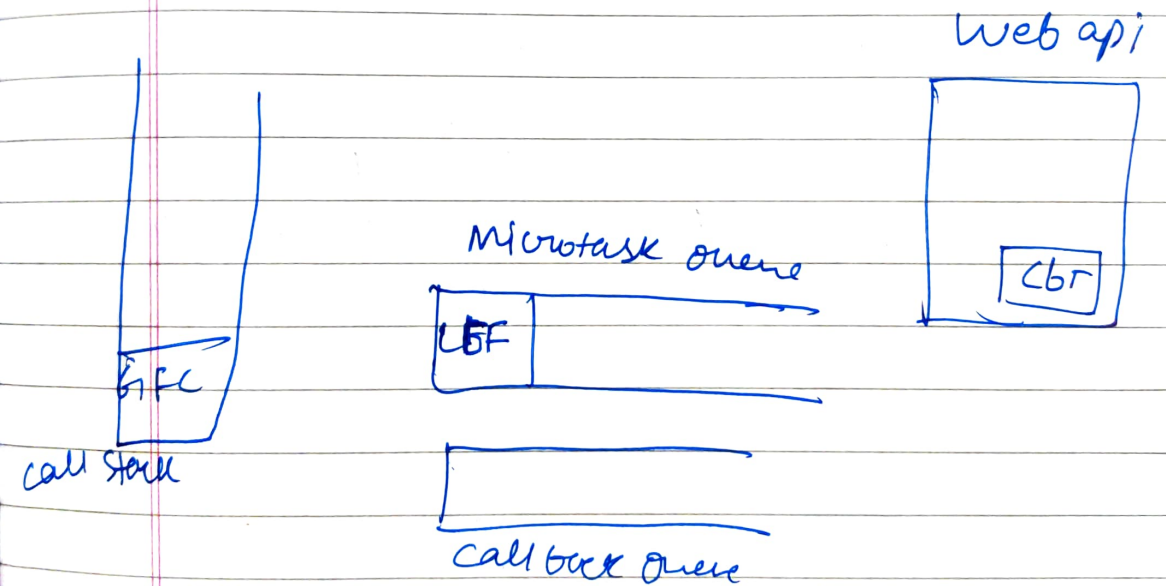
```
console.log("Start");
setTimeout(function CBT(){
  console.log("CB setTimeout");
}, 5000);
fetch("https://api.netflix.com")
.then(function CBF(){
  console.log("CB Netflix");
});
console.log("End");
```



CbF is waiting to get data from Netflix
CbT is waiting for timer to finish

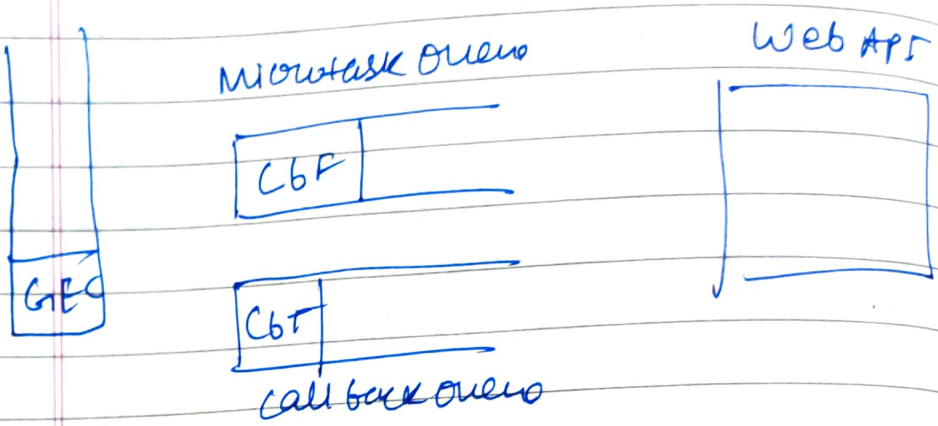
Suppose we get Data from Netflix quickly. So will CbF will go to callback queue?
NO!
It will go inside Microtask queue.
It is similar to callback queue but it has higher priority.

What comes inside Microtask queue?



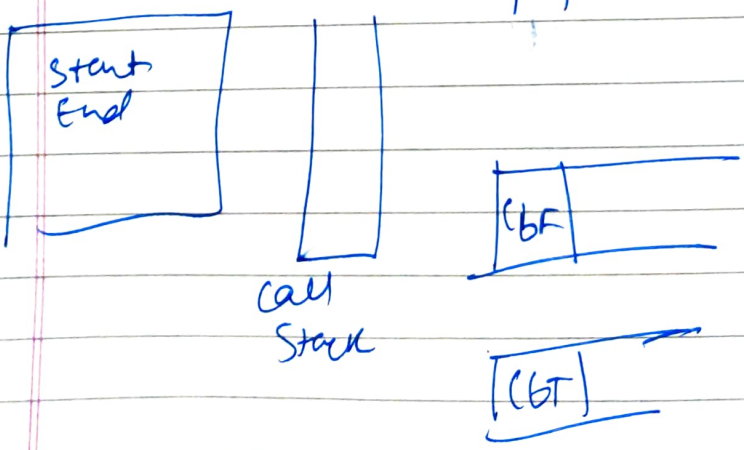
CbF will wait inside Microtask queue and wait for call stack to finish.

Suppose code inside call stack has millions of lines. Mean while data has come from Netflix and timer has also finished



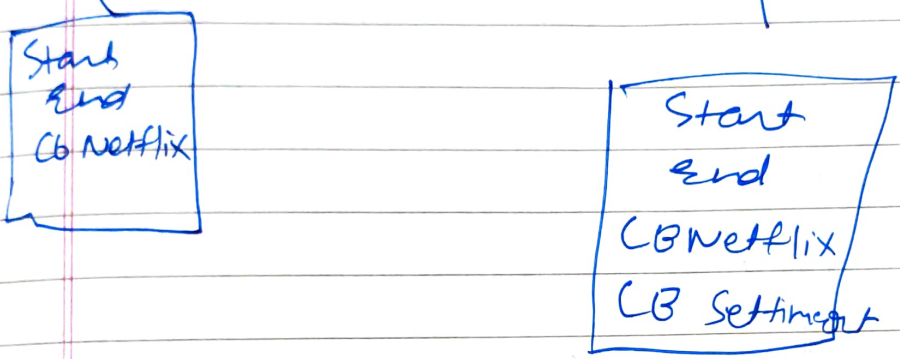
Call stack's code is still running and function in both of our queue is waiting to get executed.

Suppose now code completes so end will be printed. Call stack is empty



now Event loop will push CbF to call stack (∵ it has high priority)

After CbF completes, CbT will be pushed to call stack and CbT executes



What goes inside Microtask Queue?

* The callback functions which comes through promises
* Mutation observer(keeps on checking whether there is some mutation in DOM tree or not. If there is any, ~~they~~ If executes some call back function

Debouncing

```
<input type="text" onkeyup = "betterFunction()">

const getData = () => {
  // call api and get data
  console.log("data");
}

const doSomeMagic = (fn, d) => {
  let timer;
return function() {
  clearTimeout(timer)
  timer = setTimeout(() => {
    fn();
  }, d);
}
}

const betterfunction = doSomeMagic(getData, 300);
```

clearing the timer before calling setTimeout to restrict creating multiple timer instances.

Works but has scope issue

better function will be called once you won't type anything for 300 ms.

```
const doSomeMagic = (fn, d) => {
  let timer;
  return function () {
    let context = this
    let args = arguments;
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, d);
  }
}
```

my observations

* doSomeMagic will run only 1 time no matter how many times you type as it will return a function during its first call only
* Everytime you type before 1 sec (if delay is 1000), clearTimeout will clear the timer and setTimeout will set for next 1 sec.
* let timer variable will be accessible even after returning function because of closer.

Throttling

```
window.addEventListener("resize",
                betterExpensive);

const expensive = () => {
  console.log("Expensive");
};

const betterExpensive =
    throttle(expensive, 300);

const throttle = (fn, limit) => {
    let flag = true;
return function() {
  if (flag) {
    fn();
    flag = false;
    setTimeout(() => {
      flag = true;
    }, limit);
};
```

Works fine but has scope issues and missed some edge cases

can take ref from Technical Sumaya on youtube

class
Date
Page

What if expensive method takes arguments? How to pass that to better expensive method?

```
const throttle = (fn, limit) => {
  let flag = true;
  return function () {
    let context = this;
    let args = arguments;
    if (flag) {
      fn.apply(context, args)
      flag = false;
      setTimeout (() => {
        flag = true;
      }, limit);
    }
  }
};
```

Application → window resize

(ii) on click of button, you don't want to call api multiple times. If you want to limit the rate, you can use throttling.

Shooting game.

Understanding call and Apply Methods in JS

In other object-oriented programming languages, the 'this' keyword always refer to the current instance of the class. Whereas in JS, the value of 'this' depends on how a function is called.

```
const person = {
  firstName : 'John',
  lastName : 'Doe',
  printName : function(){
  console.log(this.firstName + ' ' + this.lastName)
  }
};
Person.printName();
```

Out - John Doe.

Here I am calling the printName() method using the person object, so the 'this' keyword inside the method refers to the person object.

```
const printFullName = person.printName;
PrintFullName();
```

Out - undefined undefined.

Why ?
Here we are storing a reference of person.printName to printFullName variable. After that we are calling it without an object reference so this will now refer to the window (global) object or undefined (in strict mode).

Ex-2

```
const counter = {
count : 0,
incrementCounter : function() {
console.log(this);
this.count++;
}
}
document.querySelector('.btn').
addEventListener('click', counter.
incrementCounter);
```

What would be the value of this inside incrementCounter() method.

In the above, the this keyword refers to the DOM Element where that event happened, not the counter object.

We can see that the this keyword inside a function refers to different objects depending on how the function is called and sometimes we accidently lose reference to the this variable

Soln ?-

call(), Bind(), Apply()

We use call, bind and apply methods to set the this keyword independent of how the function is called.

Bind

The bind method creates a new function and sets the this keyword to the specified object.

Syntax-

function.bind(thisArg, optionalArg)

```
const John = {
  name: 'John',
  age: 24,
};
const Jane = {
  name: 'Jane',
  age: 22
};
function greeting() {
console.log(`I am ${this.name}
  and I am ${this.age} year
    old`);
}
```

```
const greetingJohn = greeting.bind(John);
out - I am John and I am 24 year old
```

```
const greetingJane = greeting.bind(Jane);
out - I am Jane and I am 22 year old
```

```
const counter = {
count: 0,
incrementCounter: function() {
  console.log(this);
  this.count++;
}}
document.querySelector('.btn').addEventListener
  ('click', counter.incrementCounter.bind
                    (counter));
```

the this keywood inside the incrementCounter method will now correctly refer to the counter object instead of the event object.

classmate
Date
Page

Bind() can also accepts arguments

Syntax - function.bind(this, arg1, arg2, ...)

```
function greeting(lang) {
console.log(`${lang}: I am ${this.name}`);
}
const John = {
 name: 'John',
};
const Jane = {
 name: 'Jane'
};
const greetingJohn = greeting.bind(John, 'en');
 greetingJohn();

const greetingJane = greeting.bind(Jane, 'es');
 greetingJane();
```

Difference between call() and bind() is that call() sets the this keyword and executes the function immediately where as bind() creates a copy of that function & sets the this keyword

call()

The call method sets the this inside the function and immediately executes the function.

first example from bind—

```
greeting.call(John);
I am John and I am 24 year old

greeting.call(Jane);
I am Jane & I am 22 year old
```

call also accepts arguments

```
greeting.call(John, 'Hi');
greeting.call(Jane, 'Hello');
```

Apply()

Apply is similar to call(). The difference is that apply() method accept array of arguments instead of comma seperated values.

```
greet.apply(John, ['Hi', 'en']);
greeting.apply(Jane, ['Hola', 'es']);
```

Objects functions and this

"this" can point to different things depending on how the function is called.

```
console.log (this);
out = window {...}
```

```
function a() {
console.log (this);
}
a();
out = window {...}
```

```
var b = function() {
  console.log (this)
};
b();
out = window {...}
```

```
var c = {
name: "the c object",
log: function() {
      console.log (this);
  }
};
c.log();
out = Object {
      name: "the c object",
      log: function
  }
```

In this case, this is pointing to c object.

classmate
Date
Page

Hoisting

→ (med) Rahil Shaikh – JavaScript only hoists the declarations not initializations. Also variables declared with var keywords are hoisted not with the ones declared with let and const keywords

→ Abdullah Imran (med)

```
function hoisting(){
console.log(test);
var test = "Hoisting Test";
}
hoisting()
out = undefined
```

```
function hoisting(){
console.log(test);
test = "Hoisting Test";
}
hoisting()
out = uncaught ref Error: test is not defined
```

JS only hoists declarations so initialisation can't be hoisted and we get error

Let key word

```
function hoisting () {
  console.log (test);
  let test = "Hoisting Test";
}
hoisting()
```

out - uncaught ref Error: cannot access test before initialization.

Variables declared with let are still hoisted, but not initialized, inside their nearest Enclosing block. If we try to access it before initializing will throw Reference Error due being into Temporal Dead zone.

Hoisting Functions

Function Declarations gets hoisted

```
sayHello();
function sayHello() {
  console.log('Hello');
}
```

out = Hello

Function expression doesnot get hoisted.

```
sayHello();
var sayHello = function() {
  console.log('Hello');
}
```

Out – Uncaught Error: sayHello is not defined

order of Precedence → Meena Zakira (Med)

① Variable assignment over function declaration

```
var double = 22;
function double(num) {
  return (num * 2);
}
console.log(typeof double);
```

Out = number

② function declaration over variable declaration

```
var double;
function double(num) {
  return (num * 2);
}
console.log(typeof double);
```

Out :- function.

Codebrust →

```
var name = "dev"

var show = function () {
    console.log (name)
    var name = "rahul"
    console.log (name)
}
show()
```

out → undefined
rahul

→ same output with normal function (fun declaration)

During the creation phase, JS Engine will see var keyword and then it will give it a value as undefined to both show function and name keyword. And then during the execution phase when it comes to console.log, it will print undefined. Then it will assign name = rahul & print rahul to next console

```
display()
function display() {
console.log("hi")
}
function display() {
console.log("bye")
}
out = bye
```

during creation phase hoisting occurs and it will see keyword function and then first it will see function display and it will allocate memory for it. When it sees display again, it will re write it in the memory; So now we have lost our first function display.

Temporal Dead zone

Are let & const gets hoisted - Yes

Difference between Syntax and Reference and Type Error.

let & const declarations are hoisted but different than var. - they remain in Temporal dead zone for time being.

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

```
console.log(a)
let a=10
var b=100
out = Reference Error.
cannot access a before initialization
```

```
1
2  let a=10
3  console.log(a)
4  var b=100;
```

script
a: undefined
Global
b: undefined

both a and b have assigned memory with value undefined but b is in global space a is inside script (same for const) they are not global. you cannot access let and const unless you have put any value in them.

debugger

→ Let a = 10
console.log(a)
var b = 100

now 10 is assigned to a and it is ready to be accessed.

Script
a: 10
Global
b: undefined

Temporal dead zone is the time between let variable was hoisted and till it is initialized with some value.

Whenever you try to access a variable in temporal dead zone it gives you a reference error

→

console.log(x)
o/p → uncaught reference Error: x is not defined.

let a = 10
var b = 100
window.b = 100
window.a = undefined
Let and const don't get attached to window obj

this.b = 100
this.a = undefined

```
let a = 10
let a = 100
SyntaxError: Identifier 'a' has already been declared
```

```
Let a = 10;
Var a = 100;
SyntaxError: Identifier 'a' has already been declared
```

```
var b = 10
var b = 100
No Error
```

```
Const b;
b = 100
console.log(b)
SyntaxError: Missing initializer in const declaration
```

```
Const b = 100
    b = 1000
Console.log(b)
uncaught type Error: Assignment to Constant Variable
```

Block, Scope & Shadowing

Akshay

What is block in JS?

A block is defined by { }

It is used to combine multiple JS lines into one group.

```
1  {
2  var a=10;
3  let b=20;
4  const c=30;
5  }
```

Block
b: undefined
c: undefined
Global
a: undefined

That's why we say let & const are block scope. When JS executes this block, let and const variables will be cleared but var remains because it's in global scope.

```
{
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);
```

Out =
10
20
30
10
Uncaught ref Error.
b is not defined.

Shadowing

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

out 10
10

∵ var are assigned in global scope 100 will be overridden by 10.

```
let b = 100
{
let b = 20
console.log(b);
}
console.log(b)
```

out = 20
100

Block
b: 20

script
b: 100

Same with consts

Illegal Shadowing

```
let a=20;
{
var a= 10;
}
```

Uncaught Syntax Error: Identifier 'a' has already been declared.

```
let a=20;
{
let a=10;
}
```

No Error

```
var a=20;
{
let a=10;
}
```

Works fine. No Error

```
let a=20;
function x(){
var a=200;
}
```

Works fine

Same for const

Variable shadowing, if something should not cross the boundary of its scope.

Var is a function scoped.

where var is inside function and not interfering with a outside

Types of function

1. Function declaration
   function hello() { }

2. Function expression
   const hello = function() { };

3. Named function expression
   const hello = function hello() { }

4. Arrow function
   const hello = () => { }

5. IIFE
   (function() { }
   ())

   for arrow funct → (() => { }
)()

suppabhas supi medicum

classmate
Date
Page

Difference between Regular & Arrow func

(1) Implicit Return -

In regular function, you have to use return keyword to return any value. If you don't return anything then the function will return undefined.

```
function test() {
  return "test function";
}
test();
```

out - test function.

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

out - test function
undefined.

Arrow functions behaves same. One diff is that if there is only one line, you don't have to write return. It will be returned automatically.

(2) Arguments binding

In regular function, Arguments keywords can be used to access the arguments passed to function.

Ex- function test (a,b){
console.log (arguments)
}
test (1,2)
Out Arguments [1,2]

Arrow functions do not have arguments binding -

const arrow =(a,b) => {
console.log (arguments)
}
arrow (1,2)
Out-- Reference Error: arguments is not defined.

If you want to access arguments in an arrow function, you can use rest operator.

var arrow = (... args) => {
console.log (... args)
}
arrow (1,2)
Out = 1 2

(3) This

In regular function, this changes according to the way that function is invoked.

(i) Simple Invocation: this equals the global object or maybe undefined if you are using strict mode.

(ii) Method Invocation: this equals object that owns the method.

(iii) Indirect invocation: this equals the first argument

(iv) constructor invocation: this equals the newly created instance.

(1) Simple Invocation

```
function SimpleInvocation(){
  console.log(this);
}
SimpleInvocation();
```

out - window object.

(2) Method Invocation

```
const methodInvocation = {
  method() {
  console.log(this);
  }
};
methodInvocation.method();
```

out - logs methodInvocation object

(3) Indirect Invocation

```
const context = {aVal: 'A', bVal: 'B'};
function indirectInvocation() {
  console.log(this);
}
indirectInvocation.call(context);
```

out = {aVal: 'A'}

```
indirectInvocation.apply(context);
```

out = {bVal: 'A'}

(4) Constructor Invocation

```
function constructorInvocation() {
  console.log(this);
}
new constructorInvocation();
```

out = logs an instance of constructorInvocation

Arrow functions don't have their own this,

this inside an arrow function always refers to this from the outer context.

```
var name = "Suprabha"
let newObject = {
name: "supi"
arrowfunc: () => {
 console.log(this.name);
 },
 regularfunc() {
  console.log(this.name);
 }
}

newObject.arrowfunc(); // Suprabha
newObject.regularfunc() // supi
```

(4) New

Regular functions are constructible, they can be called using the new keyword.

```
function add (x, y) {
console.log (x+y)
}
let sum = new add (2,3)
out = 5.
```

Arrow functions can never be used as a constructor functions.

```
let add = (x, y) => console.log(x+y);
const sum = new add (2, 4);
```

out - TypeError : add is not a constructor.

(5) No duplicate named parameters

function add(a, a) { }
Will work

'use strict';
function add(a, a) { }

// Uncaught SyntaxError: Duplicate parameter name not allowed in this context.

Arrow functions can never have duplicate named parameters whether in strict or non-strict mode.

const arrow = (a, a) => { }

Arrow functions don't get hoisted: maybe.

(6) Function Hoisting

In regular function, function gets hoisted at top.

```
normalFunc()

function normalFunc() {
  return "Normal function"
}
```

Out = Normal Function

In arrow function, function get hoisted where you define. So if you call the function before initialisation, it will give referenceError

```
arrowFunc()

const arrowFunc = () => {
  return "Arrow Function"
}
```

Out - ReferenceError: cannot access 'arrowFunc' before Initialization.

When not to use Arrow functions?

object methods

```
let dog = {
  count : 3,
  Jumps : () => {
      this.count++
  }
}
```

When you call dog.Jumps the number of count does not Increase. It is because this is not bound to anything, and will Inherit the value of this from its parent scope.

My analysis

```
let dog = {
name : "scooby",
count : 3,
Jump : () => {
  this.count++;
  console.log(this);
}
}

dog.jump();
console.log(dog.count);
```

ans = 3
window object

```
let dog = {
name : "scooby",
count : 3,
jump() {
  this.count++;
  console.log(this)
}
}

dog.jump().
console.log(dog.count);
```

4
{name: scooby, count: 3, Jump:}