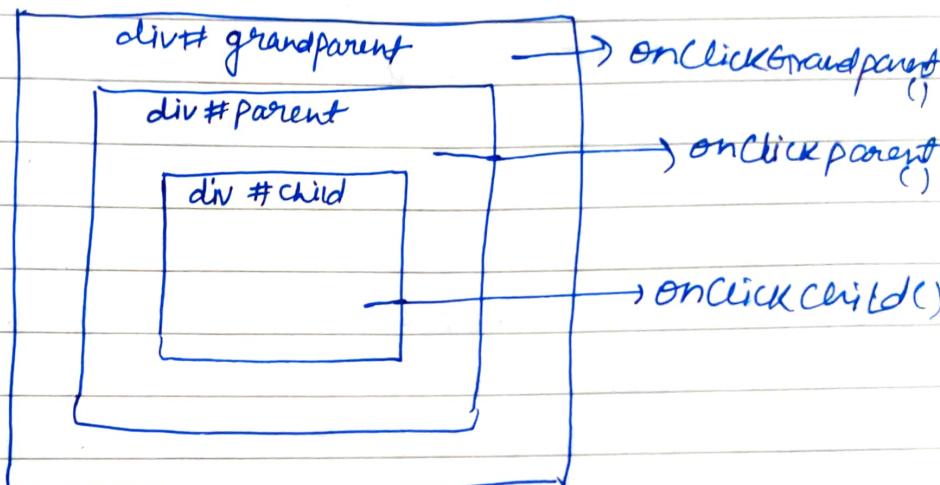


Event Bubbling and capturing

also known as event trickling

There are two ways of event propagation in the DOM tree.



In case of event bubbling

If you click on child, order of execution -

- onclickchild()
- onclickparent()
- onclickgrandparent

In case of event capturing

If you click on child, order of execution -

- onclickGrandparent
- onclickParent
- onclickChild

onclick of Parent
event bubbling

- onclickParent()
- onclickGrandparent

event capturing

- onClickGrandparent
- onClickParent

By default, browser uses Event ~~bubbling~~
To override this to use Event ~~capturing~~
Pass third variable as ~~true~~ ^{page}.

How to use bubbling and capturing -
you can add 3rd argument to
event.listener '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("closest parent clicked");
}, true);

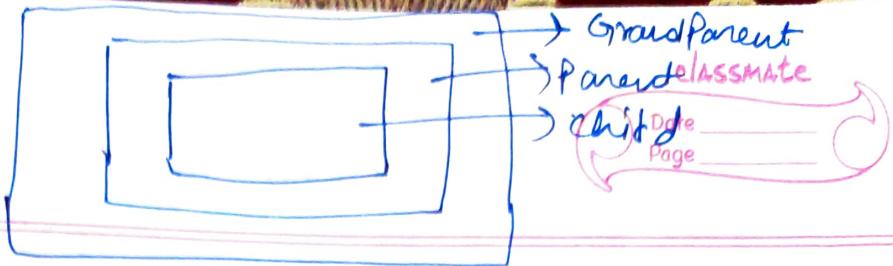
document.querySelector("#child")

• 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) GrandP
(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 propagation up
and down the hierarchy?

Set → e.stopPropagation

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

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

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

- When click on child
- child clicked
 - Parent clicked

If you write e.stopPropagation in child

- When click on child
- child clicked

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

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

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

When clicked on child

- Grand Parent clicked
- Parent clicked
- Child clicked

If e.stopPropagation is on grandParent
only

When clicked on child

- Grand Parent clicked -

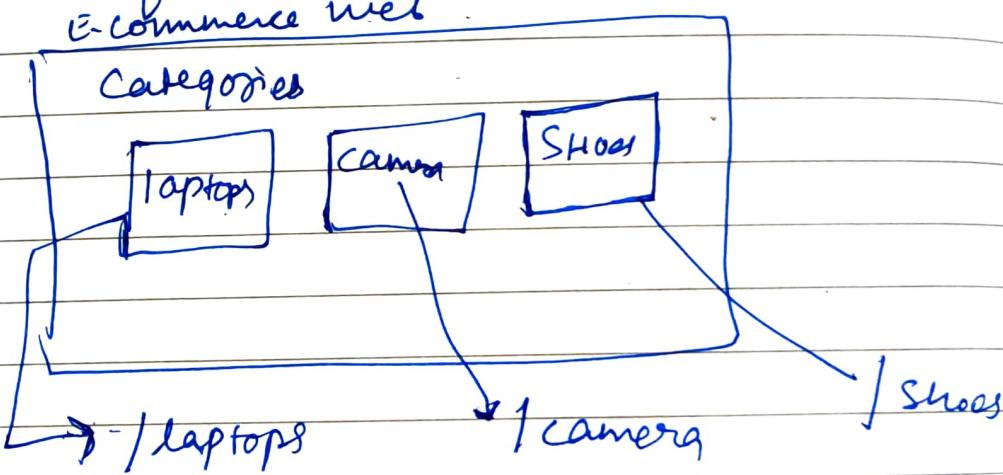
Event Delegation

classmate

Date _____
Page _____

Event Delegation 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 delegation we can fix that problem.



one way is to add event listeners to each categories i.e

document.querySelector('#laptop').

addEventListener('click', ()=>{
 // logic
})

Similarly for camera & shoes

What is better way?

Event delegation 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 delegation

```
<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 Delegation

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

Event Loop

callback queue
Microtask queue
Page

JS is a synchronous single-threaded language.

It has one call stack and it can do one thing at a time.

The ~~stack~~ 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 creates Global execution context (GEC). It gets pushed inside call stack.

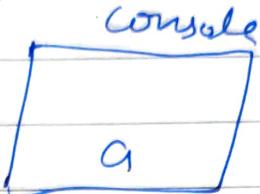
Call Stack 2 → First line is function

definition. So no change in call stack. A will be allocate memory & function will be stored.

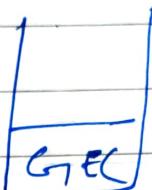
As soon as control reaches to a(), execution context will be created for function a and pushed inside call stack.



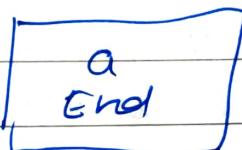
Content inside a() will be executed line by line



after printing a , execution context of a finishes



Now "End" will be console



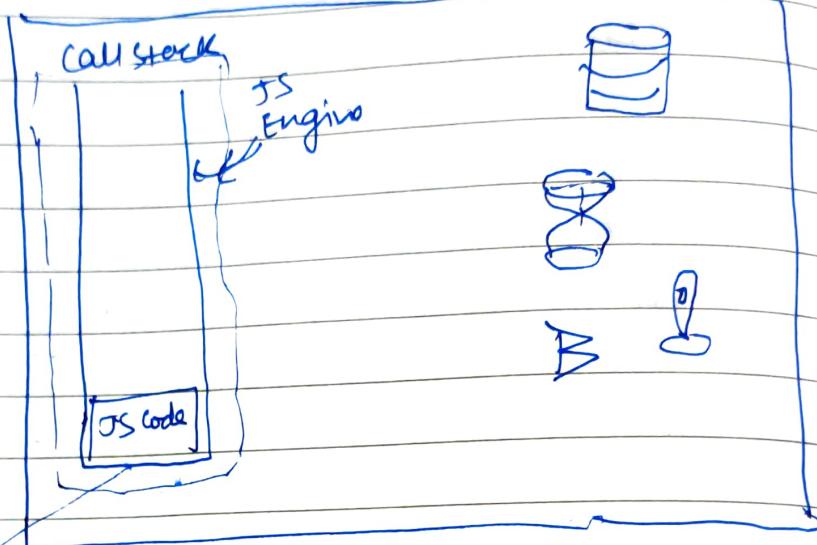
and GEC will be finished

- * The main job of call stack is execute anything which comes inside it. It waits for no one.

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?
Browser



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 -

LocalStorage, timer, Bluetooth, Location.

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

How?

via Web APIs

Web APIs

Window

- * setTimeout()
- * DOM APIs
- * Fetch
- * Local Storage
- * console
- * Location

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`
- * If you want to use Dom tree - Browser gives DOM APIs etc.

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

`window.setTimeout()`

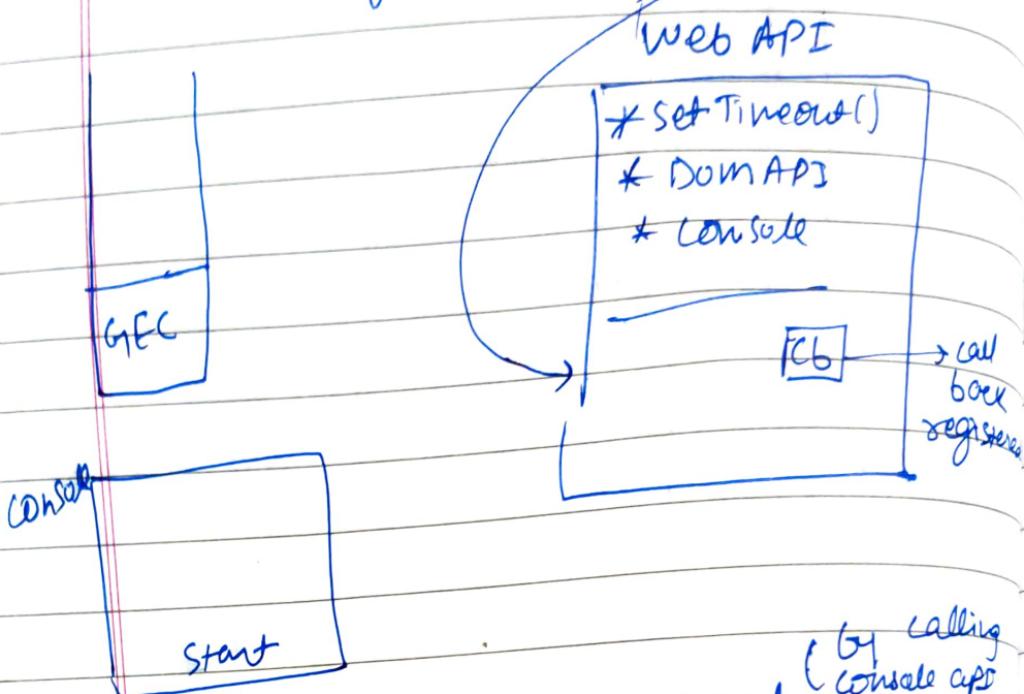
`window.fetch()`

You can able to access without writing `window`. Explicitly because `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").

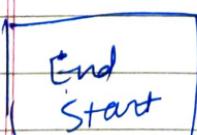
classmate
Date _____
Page _____



- * The first line will be printed
- * 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

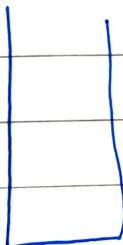


As soon as timer ends, the callback function cb needs to be executed. But as we know all of JS code executes inside call stack, somehow we need cb inside call stack.

Now comes Event loop and call back queue

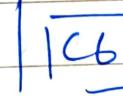
After timer expires, the cb goes to call back queue.

call
stack



Event
loop

callback queue



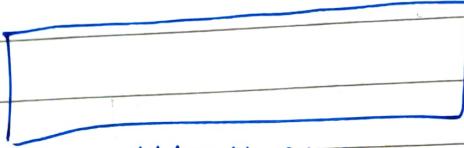
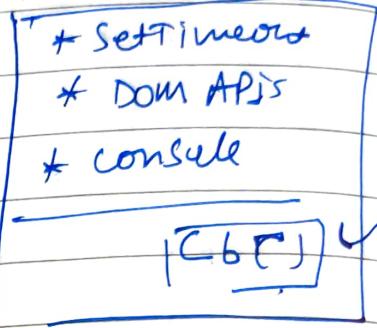
Event loop act as a gate keeper and continuously keep checking if we have anything in callback queue. If we have, it pushes it inside call stack if and only if call stack is empty the call stack creates execution context of cb and executes cb.

callback
End
Stack

```

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

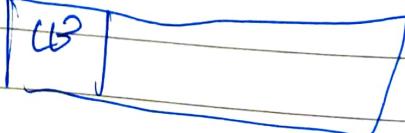
call stack



callback queue

cb() is waiting inside Web API.
As soon as you click the button, it gets pushed inside callback queue and waits 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

web api

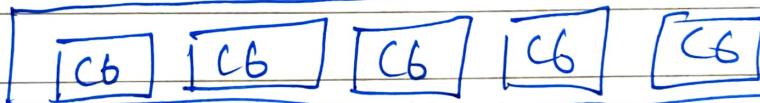


Event loop has only one job - continuously monitor call stack and call back queue.

As soon as call stack is empty and there are any function in call back queue, it takes that function & push inside call stack and call stack executes it quickly

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 execute sequentially.



Callback queue.

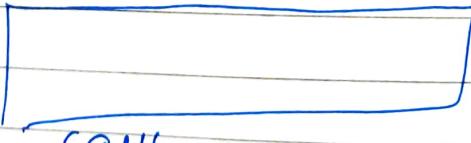
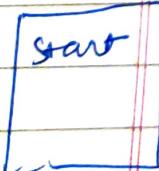
Fetch API works differently from
setTimeout & DOM and other APIs

```
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");
```

Web API

call stack



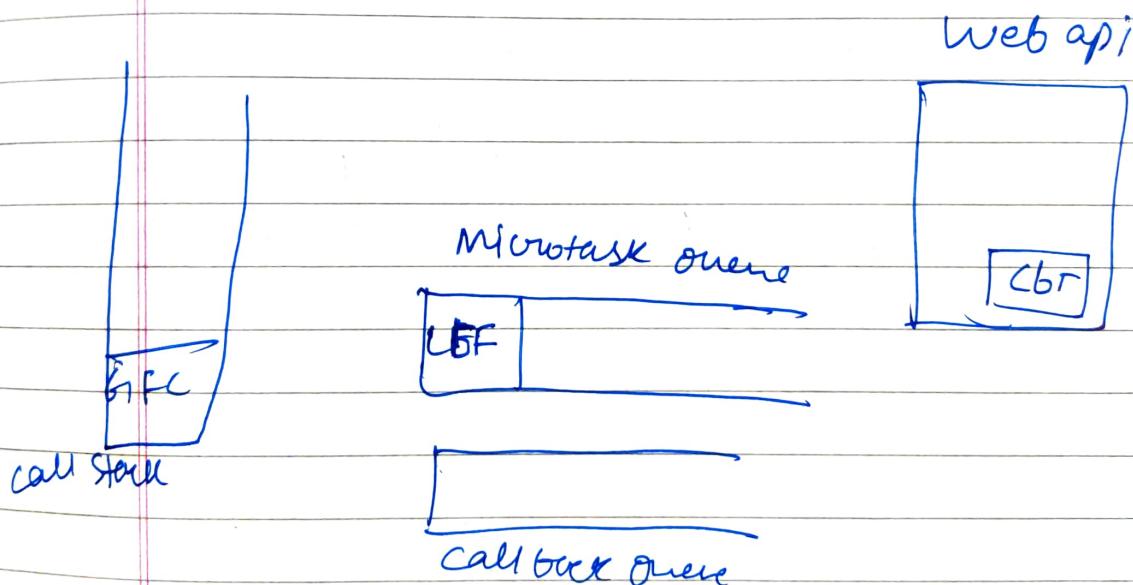
CBF is waiting to get data from Netflix
CFT is waiting for timer to finish.

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?



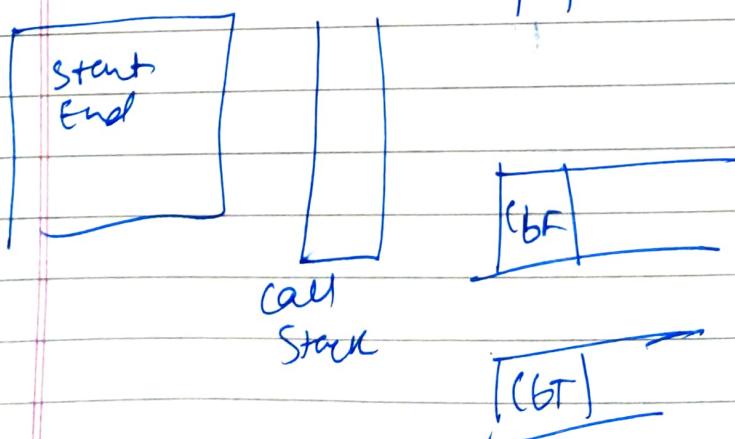
CBF will wait inside Microtask queue and wait for call stack to finish.

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



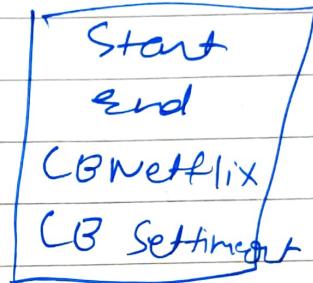
Call stack's code is still running and functions in both of our queues are waiting to get executed.

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



now Event loop will push CBF to call stack (because it has high priority).

After CBF completes, CBT will be pushed to call stack and CBT executes.



What goes inside microtask queue?

- * The callback function which comes through promises

- * Mutation observers

~~It~~ keeps on checking whether there is some mutation in DOM or not. If there is any, ~~it~~ It executes some callback 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) clearing the time  
        timer = setTimeout(() => {before calling  
            fn();  
        }, d);  
    }  
}
```

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

works but has scope issue

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

CLASSTIME

Date _____

Page _____

```
const doSomeMagic = (fn, d) => {
```

```
let timer; fn
```

```
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

(if delay is 1000)

- * Everytime you type before 1 sec, 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 closure.

Throttling

CLASSMATE

Date _____
Page _____

window.addEventListener ("resize",
betterExpensive);

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

```
const betterExpensive =  
    throttle (expensive, 300limit);
```

```
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
not too many
Techniques
YouTube



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 bind 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.querySelectorAll('.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, has 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 accidentally lose reference to the this variable set? -

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,
};

classmate

Date _____

Page _____

const Jane = {
name: 'Jane',
age: 22,
};

function greeting() {
console.log(`I am \${this.name}
and I am \${this.age} years
old`);
}

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

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

const counter = {

count: 0,

incrementCounter: function() {

console.log(this);

this.count++;

};

document.querySelector(`#btn`).addEventListener
(`click`, counter.incrementCounter.bind
(counter));

the this keyword inside the incrementCounter method will now correctly refer to the Counter object instead of the Element object.

Bind() can also accept 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 ~~inside~~ ^{class} that function immediately where as ~~bind()~~ creates a copy of that function & sets the `this` keyword ~~inside~~ ^{outside}.

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

first example from Bird -

`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 separated 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 object",
```

```
    log: function() {
```

```
        console.log (this);
```

```
    }
```

```
c.log();
```

```
out: Object{
```

```
    name: "the object",
```

```
    log: function
```

Syntax
call!
this is
pointing to
Object.

Hoisting

→ **med** Ramu shakha - 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 Insan **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 var word

```
function hoisting () {  
    console.log ('test');  
    let test _ hoisting Test;  
}  
  
hoisting ()
```

Output - uncaught ref error: cannot access
test before initialization.

Variations declared with let are still hoisted, but not initialized, inside their nearest enclosing block.
If we try to access it before initializing will throw ReferenceError due being into Temporal Dead Zone.

Hoisting Function

Function Declaration Gets Hoisted

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

! Function expression does not get hoisted.

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

out - uncaught Error: sayHello is not defined

order of precedence → menu zakira (not)

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

Code output →

Var name = "dev"

Var show = function () {

console.log(name)

var name = "value"

console.log(name)

↑

• show()

out → undefined → same output with
value

normal function ("fun declared")

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 = value] & print value to

next console

display()

function display() {

 console.log("hi")

}

function display() {

 console.log("hi")

}

out = type

During creation phase hoisting occurs and JIT will see keyword function and then first it will see function display and JIT will allocate memory for JIT. 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 Akshay

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

- | | | |
|---------------|----------------------|-------------------------|
| 1 | open | |
| 2 | let a = 10 | both a and b have |
| 3 | console.log(a) | assigned memory |
| 4 | var b = 100; | with value undefined |
| 5 | script | but b is in global |
| a : undefined | space or is inside | Script (same for const) |
| 6: undefined | they are not global. | You cannot access |
| | | let and const unless |
| | | you have put |
| | | any value in them. |

Debugger
 → Let a = 10

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)

Output - 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

still b = 100

this.a = undefined

```
let a = 10
```

```
let a = 100
```

Syntax Error: Identifier 'a' has already been declared

```
let a = 10;
```

```
var a = 100;
```

Syntax Error: Identifier 'a' has already been declared

```
var b = 10
```

```
var b = 100
```

```
No Error
```

```
const b;
```

```
b = 100
```

```
console.log(b)
```

Syntax Error: Missing initializer in const declaration

```
const b = 100
```

```
b = 1000
```

```
console.log(b)
```

uncaught typeError: 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
a:	Global 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

~~b: 100~~

script

b: 100

~~b: 20~~

Same with const

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 = 20;
```

}

works fine



variable
If any ~~other~~ is
shadowing, it
should not cross
the boundary
of its scope.

Var is a function
scoped.

Same for
const

there var is inside
function and not interfering with
one side a

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
func

```
(() => {})  
)()
```

superfish super medium

classmate

Date _____

Page _____

Difference between Regular & Arrow functions

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 behave same.

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

(2)

Arguments binding

In regular function, Arguments key words 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' }

Constructor Invocation

`function constructorInvocation() {`

`console.log(this);`

`};`

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 newObjet = {
```

```
name: "Supi"
```

```
arrowfunc: () => {
```

```
console.log(this.name);
```

```
};
```

```
regularfunc() {
```

```
console.log(this.name);
```

```
}
```

```
}
```

```
newObjet.arrowfunc(); // Suprabha
```

```
newObjet.regularfunc(); // Supi
```

(2)

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.

normal func()

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 reference error

arrowFunc()

const arrowFunc = () => {

 return "Arrow Function"

}

Out - Reference Error: 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);
any = 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: }
```