Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ High Order Functions:

Function which takes another function as an argument or returns a function is known as Higher Order Func.

E.g.:
```
function x() {
    console.log("Hello");
}
```
Callback

```
funition y(x) {
    x();
}
```
Higher order function

E.g. Find area, circumference, diameter of circle by radius

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

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

const circumference = function (radius){
    return 2 * Math.PI * radius;
}

const diameter = function (radius){
    return 2 * radius;
};
```

Date : .....

MON TUE WED THU FRI SAT SUN

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

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

- HOF helps to achieve DRY (Don't Repeat Yourself) principle

→ Map, Filter & Reduce:

1) Map: map method is used when we want transformation of whole array.

```
E.g.  arr.map (function (x) {
          return x * 2
      })

      OR

      arr.map (x => x * 2)
```

Date :
MON TUE WED THU FRI SAT SUN

2) Filter : filter is used when we want to filter the array to obtain required value.

E.g. arr.filter (x => x %2 === 0) // Check even no.

(3) Reduce : Reduce is used when we want to reduce the array to single value, like max, min, avg, sum, difference etc.

It accepts two arguments one function which includes accumulator & current iterative value & another argument is initial value of accumulator.

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

- User can create its own such function like map. (Imp)

E.g. arr = [1, 2, 3, 4, 5] // Double the value of arr.

```
Array.prototype.double = function () {
    const output = [];
    for ( let i=0; i < arr.length ; i++) {
        output.push (this[i] * 2);
    }
    return output;
}

• console.log (arr.double());
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ Callback Issues:

Although callback helps us to achieve asynchronous in JS, but it has two major issues,

1) Callback Hell / Nested Callbacks: A.K.A Pyramid doom

It means when we call nested function as callback it will lead to unmaintainable & not-understandable program. Nested function means callback passed into another function as an argument & another function passed into callback function as nested callback.

```
E.g  function a ( function () {
         function b ( function () {
             function c ( function () {
             }
         }
     }
```

2) Inversion of Control - Call back function is passing to another function as an argument which leads to blindly trust to the function, which follows losing our control on our callback function for execution.

Date :.....

MON TUE WED THU FRI SAT SUN

→ Promise:

Promise is an object which represents eventual completion or failure of an asynchronous operation.

- Promise helps to resolve two major issues of callback, i.e Callback Hell & Inversion of Control.

E.g Create a shopping cart expirence, where order is created then user can proceed to payment & then see order summary, Assume api function for these operations are already created. which are asynchronous.

Previous Approach:

```
api.createOrder( products, function () {
        api.proceedToPayment (orderId), function () {
            api.showOrderSummary (paymentId, orderId);
        }
});
```

Above prog leads to Callback Hell & IOC.

Approach with Promise:

```
                ↓ Promise Object
const orderCreation = api.createOrder (products)
        // Return orderId
```

Date : .....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

// This will also return promise object as we called payment

```
orderCreation.then(function(orderId){
    return api.proceedToPayment(orderId);
}).
.then(function(paymentId, orderId){
    return api.showOrderSummary(paymentId, orderId);
})
```

- Make sure to return, otherwise we will not get response in Promise chain
- Promise has 3 states: pending | fullfilled | rejected.
- Promise is immutable in nature.

Creating Promise:

User can create there own promise function which return promise at end.

E.g. Create Order Promise function for example

```
function createOrder(cart){
    const pr = new Promise(function(resolve, reject){
        // create order
        if(validateCart(cart)){
            resolve("123") // Returns order id
        }else{
            reject(new Error("Cart is not valid"));
        }
    });
    return pr;
}
```

Date :.....

MON TUE WED THU FRI SAT SUN

→ Error Handling in Promise:

We already check how to handle promise return values using "then" keyword. Now to handle errors we have "catch" keyword.

- catch function will take care of all the errors which might generate above it. but not below:

E.g.

```
promise.then(function(orderId){
        console.log(orderId);
})
.catch(function(err){
        console.log(err.message);
});
```

Error handling

- Make sure to avoid promise hell, in previous example, we have proceed to payment function as well, check the same example to ~~handle~~ avoid promise hell.

MON TUE WED THU FRI SAT SUN

Promise - all, allSettled, race, any

all: Promise.all take array of promises & return the results in an array.

E.g:

```
                    2s    1s    3s
                    ↑     ↑     ↑
Promise.all([p1, p2, p3]).
.then(function(res) {
    console.log(res);
});
```

// Takes 3s to return the result

- It will return array of result only when all the promise succeed, Otherwise it will give error.
- Maximum time of result of all() is maximum time any one promise take.

allSettled: Promise.allSettled take array of promises & return the results of promises in an array, only difference is it will return even if any promise fail, it wait for all promise to settle.

E.g:

```
                    S     F     S
                    2s    1s    3s
                    ↑     ↑     ↑
Promise.allSettled([p1, p2, p3]).
then(res => {
    console.log(res);
})
.catch(error => {
```

Take 3s to return the result

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

O/P : [P1 Success, P2 fail, P3 Success]

- Format of allSettled result is object, which is

```
{ status: "fulfilled" | "rejected"
  value: <Result of Promise>
  reason: <If any promise fail>
}
```

Hence, O/P :

```
[
  {status: "fulfilled", value: "P1 success"},
  {status: "rejected", reason: "P2 fail"},
  {status: "fulfilled", value: "P3 success"}
]
```

race - Promise.race() take array of promises & return the result of promise which either success or fail first.

E.g:

```
Promise.race([p1, p2, p3])   // p1: 3s, p2: 1s, p3: 2s
  .then(res => {
    console.log(res);
  })
  .catch(err => {
    console.log(err);
  });
```

O/P : P2 success.

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

any() : Promise.any() take array of promise & return the result of promise which success first.

✓3s ✗ ✓2s

E.g. Promise.any([p1, p2, p3]) O/P : P3 success

```
.then(res => {
    console.log(res);
})
.catch(err => {
    console.error(err);
});
```

- If all promises fail then it will give Aggregate error i.e return all promise errors in an array. To access all the errors,

```
console.error(err.errors);
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ async, await :

async: async is a keyword that is used to create async function, which return Promise ; If we didn't return Promise type value, it will implicitly wrap into Promise.

E.g.
```
async function getData() {
    return "Hello World";
}

const data = getData(); // Return Promise obj
data.then( res => console.log(res));
```

O/P: Hello world.

async/await: This combo is used to handle Promises.

await is keyword that can only be used inside an async function, which resolve Promises.

Major difference of handling Promise using async/await is that JS engine will wait for promise to resolved even it take time.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

E.g.

```
const p = New Promise ((resolve, reject) => {
    resolve ("Hello world")
    setTimeout (resolve ("Hello World"), 5000);
});

async function getdata () {
    console.log ("Start");
    const data = await p;
    console.log (data);
    console.log ("End");
}

getdata ();
```

O/P: Start → Print Quickly
Helloworld, End → Print after 5 secs.

Note: As mentioned earlier, JS is not wait for anything, it appears that it is waiting.

In background, once JS come on line where await there, it suspends the getdata () function till the time "p" to be resolved, once it is resolved it come back to call stack. In the suspend instance, JS is executing other code of the program.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

E.g: fetch() also return Promise

```
async function getData() {
1)      const data = await fetch(API URL); // Return Promise
2)      const jsonValue = await data.json();
3)      console.log(jsonValue);
};

getData();
```

Line 1 return resolved Promise in response object, then we need to convert into json, which is again a promise i.e we add await in line 2 in front of data.json(), and then console the value.

Note: Always use try, catch while using fetch & call API.

- Let's says we have two promises, p1 → 10 sec & p2 → 5 secs & we are calling these p1 & p2 using await keyword.

So as soon as, these promises are registered, timer is started for this p1 & p2. So we when we are calling p1 & p2, the one after another, it will return both at same time

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ this Keyword:

- this keyword in global space refers / return to global object depends on environment, for browser, it is Window.

- this keyword inside function returns depend on mode of program, if it is strict mode, it return undefine & else it return Window

E.g.
```
function x() {
    console.log(this);
}
```
- Non-strict mode
- O/P : Window object

```
"use strict"
function x() {
    console.log(this);
}
```
- strict mode
- undefined

- this keyword refers to the context where a piece of code, such as a function's body, is supposed to run.

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

// this keyword inside object's method.

function v/s method :

a function which is inside an object as an attribute is called method.

- If we use this keyword inside object method, it will refer to object.

E.g.
```
const student = {
    name : "Ash",
    fullName : function () {
        console.log (this);
    }
}

student.fullName()
```

→ O/P : {name : "Ash", fullName () }

- Arraw function differ in their handling of this, they inherit this from the parent scope at time they are defined.

E.g
```
const student {
    name : "Ash";
    fullName : () => {
        console.log (this);
    }
}

student.fullName()
```

→ O/P :
Window

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ Arrow functions retains this value of enclosing lexical context.

→ this keyword inside nested arrow function;

E.g.
```
obj emp {
    name: "Ash",
    fullName : function () {
        x = () => {
            console.log (this);
        } x();
    }
}
emp.fullName ();
```

O/P:
emp object

The reason is lexical environment of arrow fun is its parent function.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ Coercion in JS:

The unexpected typecasting in javascript known as coercion.

- While working with data you need to convert it from one to another type like number to string or something like that,

- So to minimize your efforts javascript does it automatically for you., though it is not great because most of the time get out of control & cause bugs in system.

- That's why most people avoid it but it's totally up to you, If you are familiar with this you can use otherwise just try to avoid it.

Three types of coercion:

(a) Numbers & string -

```
console.log ("100" - 10)  // O/P : 90
```

(b) Boolean:

```
console.log (True + 100)  // 101
```

(c) Equality:

```
console.log ("100" == 100)  // True
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ Polyfill of Bind:

- Polyfill - It is a fallback for a method that is not supported by the browser by default.

- In conclusion, code implementation that provides modern functionality in older environments where it might not be available.

- call(), apply() & bind() method- Function borrowing Helpers

(a) bind(): bind takes an object as an argument & returns a new function whose this refers to the object we passed as an argument.

```
E.g. const myfunc = function () {
        console.log (this.name);
     }

     const obj = {
        name : "Ash",
     };

     const bindfunc = myfunc.bind(obj);
     bindfunc();
```

Date :.....

MON TUE WED THU FRI SAT SUN

(b) call(): calls method calls the function directly & sets this to the first argument passed to the call method, & if any other sequences of arguments preceding the first argument are passed to the call method, then they are passed as an argument to function.

Syntax: call(objInstance, arg1, ... argN);

Note: call method doesn't return function

(c) apply(): apply method calls the function directly & set this to the first argument passed to the apply method.

- In apply method, we pass arguments in the form of an array, only difference between call & apply.

Syntax: apply(objIns, argsArray)

Origin of two sites:
1) protocol [HTTP, HTTPS]
2) Domain / Host
3) Port

VIMP

Date :.....

MON TUE WED THU FRI SAT SUN

- Polyfill of Bind means create our own bind function.

```
const obj = {
        name : "Ashutosh",
        getName : function () {
                console.log(this.name);
        }
}

const obj1 = {
        name : "Jake",
}

obj.getName()        // "Ashutosh"

// Create bind function
Function.prototype.customBind = function (obj) {

        let func= this; getName; // Returns to function
        return function () {
                func.call(obj);

        }
}

obj.getName.customBind
let callfunc = obj.getName.customBind(obj1);
callfunc();          // "Jake"
```

Date :.....

MON TUE WED THU FRI SAT SUN

→ Function Currying :

It is technique of transforming a function that takes multiple arguments into a sequence of functions that each take a single argument.

We can achieve function currying using two method:

1) bind :

```
function multiply (x, y) {
    return x * y
}

const multiplyByTwo = multiply.bind (this, 2);
multiplyByTwo (5);        // 10
```

(2) Closure function:

```
function multiplyByTwo (x) {
    return function (y) {
        return x * y;
    }
}

const func = multiplyByTwo (2);
func (5);    // 10
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

→ Local Storage, Session storage & Cookies :

Local Storage	Session Storage	Cookies
Storage capacity is 5 MB / 10 MB	Storage Capacity is 5 MB	Storage Capacity is 4 KB
It's not expired until deleted manually	Session expired once tab or window closed	Cookie expire based on setting.
No data transfer to server	No Data transfer to server	Data transfer to server

(a) Local Storage:

(i) setItem(): localStorage.setItem(key, value);

(ii) getItem(): localStorage.getItem(key)

(iii) removeItem(): localStorage.removeItem(key)

(iv) clear: localStorage.clear()

(b) Session Storage: Same function for session storage, just put sessionStorage instead of localStorage.

Note: When we store object in any storage, just stringify it before, otherwise implicit conversion of object to string gives wrong result. & reverse of stringify is parse.

i.e JSON.stringify(obj) — JSON.parse(stringify.data)

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

let & const :

let & const are hoisted but its memory is allocated at other place than window which cannot be accessed before initialization.

Temporal Dead Zone - It exist until variable is declared & assigned a value

window - window variable / this keyword will not give value of variable defined using let or const.

Level of strictness : var → let → const

- For var : No temporal dead zone, can re-declare & re-initialize stored in global execution context.
- For let : Cannot re-declare & ~~cannot~~ can re-initialize, stored in seperate memory.
- For const : Cannot re-declare & cannot re-initialize, stored in seperate memory.

=> Errors :

(a) Syntax error : It is similar to compile error, while type & reference error falls under run time error, it will not run JS code, it gaves error instantly.
Usually occurs when we re-declare let & const variable

Date .....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

(b) type error - While try to re-initialize const variable

(c) reference error: While trying to access variable which is not there in global memory.

Lexical ~~environment~~ scope: Lexical environment means combination of current scope memory & lexical scope of parent.

Lexical ~~environment~~ scope = Current Scope + Lexical scope of memory.

Block Scope:

Block: It is a combination of one ~~of~~ or two more statements.

We can simply create block using '{}'.

e.g:
```
{
    var a = 10;
    let b = 100;
    const c = 1000;
}
```

- In the above statement, the memory is allocated to following variables which are.

Date :.....

MON TUE WED THU FRI SAT SUN

a : Memory allocated to Global scope

b : Memory allocated to different scope known as 'Block'.

c : Memory allocated to different scope known as 'Block'

Now, if you notice b & c are in 'Block' scope, but in general they are in 'Script' scope. Reason they stored in 'Block' scope is because they are in a block.

If we declare variable outside block, it is allocated in 'Script' scope.

E.g.:

```
let b = 100
{
    let b = 10
}
```

Script :
b : 100
Block :
b : 10

The above that we did is known as "Shadowing".

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

setTimeOut():

- setTimeOut() method of the window interface sets a timer which executes a function or specified piece of code once the timer expires.

Syntax:

```
setTimeOut (code, delay?);
        or
setTimeOut (func, delay?);
```

code: It allows you to include string instead of function, which is compiled & executed when the timer expires. This syntax is not recommended, cuz. It making use of 'eval()', a security risk

func: A function to be executed after the timer expires

delay: The time, in milliseconds that the timer should wait before the specified function or code is executed. If parameter is omitted, a value of 0 is used. means execute "immediately" or more accurately, next event cycle.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

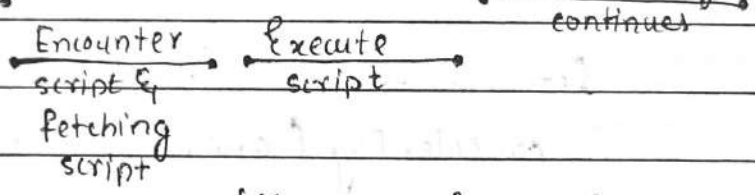
async & defer attribute:

Boolean attributes used with script tag to efficiently load external files. or scripts

- On web page load, there are two things happen in browser.
  (a) HTML parsing
  (b) Script loading
    (i) Fetching Script
    (ii) Executing script line by line.

(i) Normal — `<script src = "script.js" />`

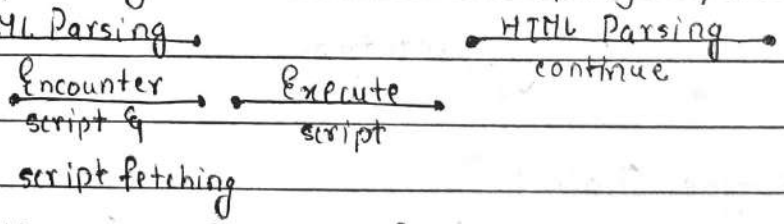
HTML Parsing
Script Load



- Blocks the HTML parsing until script loads & executes

(ii) Async : `<script async src = "script.js" />`

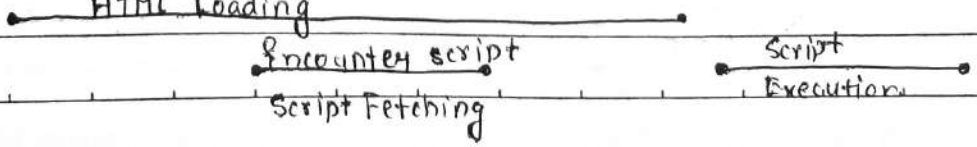
HTML Parsing
Script Load.



- HTML parsing & script fetching goes parallely.

(iii) defer : `<script defer src = "script.js" />`

HTML Parsing
Script Load



Script executes only after HTML parsing completes.

Defer maintains the order of execution.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

=> ~~Event Propogation~~

=> Event Bubbling :

It is a concept in the DOM. It happens when an element receives an event, & that event bubbles up (or you can ~~stay~~ say is transmitted or propogated) to its parent & ancestor elements in the DOM tree until it gets to root element.

This is default behaviour of events on elements unless you stop the propogation.

- How to stop event bubbling ?

event.stopPropogation()

Propogation is an act of spreading something. The stopPropogation method is used to prevent the spreading of events when an event is triggred on an element.

* event.preventDefault() :

This method prevents default action that browsers make when an event is triggered.

Date :
MON TUE WED THU FRI SAT SUN

→ target & currentTarget property in event :

There are two different properties of event object to access the element that was clicked.

event.target : target refers to the element on which the event was initially fired, ~~while~~

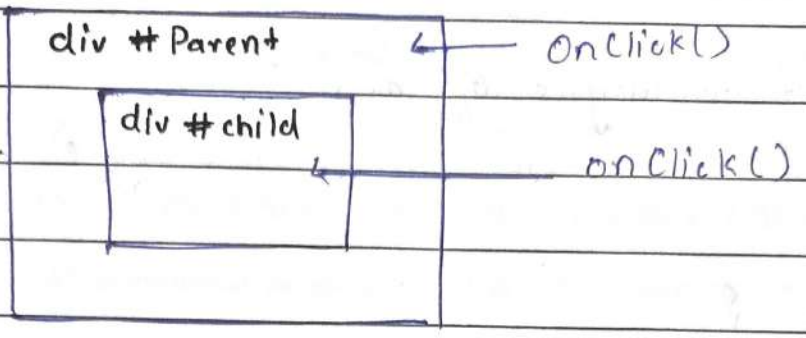
event.currentTarget : refers to the element to which this event handler has been attached

- target remains same while event bubbles, currentTarget will be different for event handlers that are attached to different elements in hierarchy.

MON TUE WED THU FRI SAT SUN

Event Bubbling & Capturing:

Ways of event propogation in DOM tree.



Every event occurs on child will propogate to parent as well.

i.e onClick() of child also propogate parent onClick.

event bubbling:

In this method, event propogation goes up to the hierarchy till the DOM tree.

like bubble always comes out.

On clicking of child, it will automatically trigger on click event of parent as well. So order would be

child OnClick() → parent # onClick

event capturing:

Opposite to event bubbling, here order is coming down i.e order would be,

parent # OnClick → child # onClick

This order when we click on child container.

Event capturing also known as Trickling / Event Trickling.

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

Syntax for event capturing:

Boolean Value

addEventListener('click', () => { }, useCapture (true/False));

Default value of useCapture is False, i.e by default it act as ~~bub~~ event bubbling.

→ Note: This has performance issue, so to stop propogation we can use. e.stopPropagation()

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

Event Delegation:

Technique to handle events in a web page, it based upon event bubbling.

Event delegation is basically a pattern to handle events efficiently. Instead of adding an event listener to each & every similar element, we can add an event listener to a parent element & call an event on a particular target using the .target property of the event object.

Benefits:

(a) Improves Memory space
(b) Mitigates risks of performance bottle necks
(c) DOM manipulation
(d) When elements get added dynamically, the process of adding events is slow.

Limitations:

(a) All the events are not bubbled up, some events like blur, focus are not bubbled up.

(b) if e.stopPropogation is used in child, events are not bubbled up.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

Debouncing :

Debouncing is a technique that helps improve the performanc of web applications by controlling the frequency at which time-consuming tasks are triggered.

If a task is triggered too often- like when a user types quickly or rapidly clicks a button, - it can lead to performance issues.

Debouncing provides a solution by limiting how frequently a function can be executed

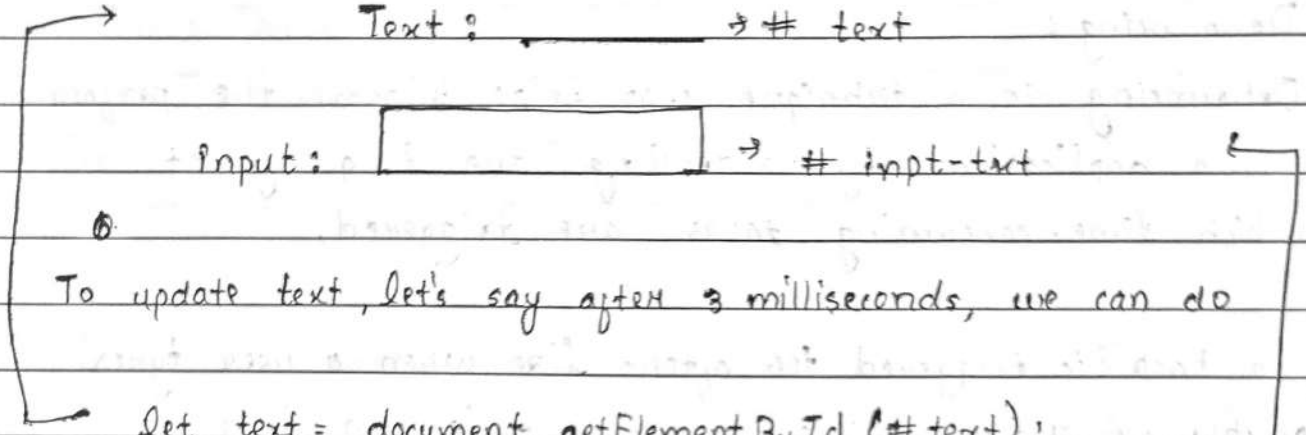
Debouncing ensures that expensive operations (like complex calculations, API calls, DOM updates) are executed only when necessary.

JS handles one operation at a time, when certain actions are triggered too frequently, such as continous scrolling or typing, it can overload browser & cause slugish performance

So, debouncing ~~improving~~ improves the performance based on delay.

Let's take an example, to update ~~an~~ paragraph with text that user mention in input, but not on every operation because it will call multiple operation, so in that we can use debouncing.

Date : .....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐



To update text, let's say after 3 milliseconds, we can do

```
let text = document.getElementById(#text);
let int-txt = document.getElementById(#inpt-txt);

int-txt.addEventListener("keyup", onUpdate());

function onUpdate(e) {
    text.innerHTML = e.target.value;
}
```

→ This is normal logic, to update the text on every new character or key pressing.

→ Now to update the text after certain delay, we need to improve our function.

```
function delayfunction(func, delay) {
    let timer;
    return function() {
        let context = this; let args = arguments;
        clearTimeOut(timer)
        timer = setTimeOut(() => {
            func.apply(context, args);
        }, delay);
```

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

// Update this line

inp-txt, addEventListener ("keyup", delayfunction (onUpdate, 300));

- Now, when there is a change in input box, delay function invokes.
- It takes two parameter, one is function which we need to call & second is delay timer that invoke function after certain delay.

- We use setTimeOut() to mention the delay & storing the timerId in a global timer variable

- When user frequently type in input box, it will call this delay function ~~bit~~ & every time it will register new timeout & clearing old timeout ~~funti~~, with that it will invoke the function after certain delay, if user did not type in that interval.

- We used context & args, because it ~~will~~ on keyup take event as parameter, so for that we pass context & argument as parameter.

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

Throttling:

Throttling limits the frequency at which function is executed, ensuring it runs at most once in a specified time interval.

It's useful for optimizing performance in scenario involving rapid, continuous events like scrolling or resizing.

Implement Throttling:

(a) Create a Throttle function
(b) Track last execution time
(c) Check time interval
(d) Return Throttled function.

Eg:

```
const throttle func = ( func, delay) => {
    let prev = 0
    return (...args) => {
        let now = new Date().getTime();
        console.log (now - prev, delay);

        if (now - prev > delay){
            prev = now;
            return func(...args)
        }
    }
}
```

Date :.....

MON TUE WED THU FRI SAT SUN

- Another approach is using setTimeOut

```
const throttlefunc = (func, delay) => {
    let flag = True;
    return () => {
        let context = this;
        let args = arguments;

        if (flag) {
            func.apply(context, args);
            flag = false;

            setTimeOut (() => {
                flag = true
            }, delay);
        }
    }
}
```

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

→ Object prototypes : (Inheritance in Js)

Every object in JavaScript has built-in property, which is called its prototype.

The prototype is itself an object, making what's called a prototype chain.

The chain ends when we reach a prototype that has null for its own prototype.

The standard way to access's an object prototype is the Object.getPrototypeOf() method.

→ When you try to access a property of an object & if the property can't be found in the object itself, the prototype is searched for that prototype. If the prototype still can't be found, then the prototype's prototype is searched & so on until either the property is found or the end of chain is reach, in which case undefined is returned.

E.g.
```
const obj = {
    value: "abc",
    value1: "xyz"
};
console.log(Object.getPrototypeOf(obj));
// O/P: Object {}
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

Prototype of Date Object:

```
const date = new Date();
let object = date;

do {
    object = Object.getPrototypeOf(date);
    console.log(object);
} while (object);
```

O/P: Date.prototype
Object { }
null

=> Setting a prototype:

Two ways to ~~describe~~ set object prototype are Object.create() & constructors.

(a) Object.create():

It helps to creates a new object & allows you to specify an object that will be used as the new object's prototype.

E.g.
```
const personPrototype = {
    greet() {
        console.log("Hello");
    }
};

const carl = Object.create(personProto);
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

(b) Using constructor:

In JS, all functions have a property named "prototype". When you call function as a constructor, this property is set as the prototype of newly constructed object.

E.g.
```
const personProto = {
    greet() {
        console.log("Hello");
    }
};

function Person(name) {
    this.name = name;
}

Object.assign(Person.prototype, personProto);
// or
Person.prototype.greet = personProto.greet;

const p = new Person();
p.greet(); // Hello.
```

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

=> HasOwn :

- Properties that are defined directly in the object, like name in Person are called own properties.

- We can check that property is ownproperty using Object.hasOwn() method.

```
console.log (Object.hasOwn(p, "name")) ; //true.
```

Date :.....

MON TUE WED THU FRI SAT SUN

☐ ☐ ☐ ☐ ☐ ☐ ☐

Prototype & Prototypal Inheritance:

Prototype is an object which has some properties like toString(), call(), reverse(), & whenever we create an object, JS automatically attach this prototype ~~in~~ with it.

Everything in JS is considered as object internally. even function is considered as object

To get prototype details of any type, like array, we can ~~write~~ get using "__proto__"

```
let arr = ["a", "b"];
console.log(arr.__proto__");
```

Above we got prototype of arr, which is array, we can also see prototype of array like this,

Array.prototype

Prototype Chain:

A ~~sy~~ specific type prototype has also a prototype. like if we do

arr.__proto__.__proto__ → Object.prototype

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

- This is called prototype chain, and it will continue until we get null.

How to set prototype of any type:
Let's say we need to assign prototype of one obj.

```
const obj = {
    name : "Ash",
    city : "Surat",
}

const obj2 : {
    intro() {
        console.log("Hello", this.name, "You are from",
                    this.city);
    }
}
```

If we need to ~~assign~~ set prototype of obj as obj2, we can do like this,

```
obj.__proto__ = obj2;
```

// Note: This is not recommended though, as it leads to performance issue.

And now, we can use intro() using obj.

```
e.g  obj.intro();
     // Hello Ash, You are from Surat
```

- And this process is called "Prototypal Inheritance".

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

Classes & Constructors:

We can create class using "class" keyword.

E.g.
```
class Person {
    name;

    constructor (name) {
        this.name = name;
    }

    introduce () {
        console.log ("Hi, My name is ${this.name}")
    }
}

const mike = new Person ("Mike");
mike.introduce();
// Hi, My name is Mike.
```

- We can class without constructor as well.

Inheritance: In class, we can achieve inheritance using 'extends' keyword.

```
class Student extends Person {
    year;

    constructor (name, year) {
        super (name);
        this.year = year;
    }
}
```

Date :.....

MON TUE WED THU FRI SAT SUN

Encapsulation :

If we want to make any class property 'private' then we can use '#' as prefix at a variable declaration time.

e.g :

```
class Student {
    #year ;          // private variable

    introduce (){
        console.log (`I'm in ${this.#year} year`);
    }
}

const std = new Student ();
std.#year ;   // Syntax Error.
```

- In class method, we can access the private variable like this.#year
- # we need to use every time.

Private Methods : Like class property, if we need to define class method as private.

```
#introduce (){
    //....
}
```

Date :.....
MON TUE WED THU FRI SAT SUN
☐ ☐ ☐ ☐ ☐ ☐ ☐

Working with JSON :

JSON is a text based data format following JS object syntax.

JSON exists as string - useful when you transmit data from across a network.

Methods :

(1) To convert object / Promise into JSON :

.json()

E.g. response.json()

(2) To convert JSON string into JSON object :

```
const vari = "{ ... }"; // JSON in string
const jsonObj = JSON.parse(vari);
```

~~const jsonObj = vari.stringify json.st~~

~~const jsonObj = JSON.stringify();~~

(3) To convert JSON obj into JSON string :

```
const vari = { ... }
const jsonStr = JSON.stringify(vari);
```