DATE

Scope

Scope determines the accessibility (visibility) of variables.

Javascript variables have 3 Types of scope
- → Block Scope
- → Function scope
- → Global scope

→ Block Scope
let and const are block scope variable declared inside a {} block cannot be accessed from outside the block

ex:-

```
{
  let x=2;
}
// x can NOT be used here
```

```
{
  const x=2
}
// x can NOT be used here
```

· Var is not a block scope variable declared inside a block {} can be accessed from outside the block

ex:-

```
{
  var x=2;
}
// x can be used here
```

DATE

→ Function Scope

Javascript has function scope. Each function creates a new scope, variables defined inside a function is not accessible from outside the function.

ex:-

```
function myFunction() {
    var carName = "Volve";
}
```

→ Global Scope

Variable declared Globally (outside my function) have Global Scope. Global variables can be accessed from anywhere in a JavaScript function.

ex:-

```
var x = 2;

let x = 2;

const x = 2;
```

DATE

⇨ Implicit and Explicit Binding

```
var obj = {
    name: "Pixush",
    display: () => {
        console.log(this.name);
    },
};

var obj1 = {
    name: "ABC",
};

obj.display.call(obj1)
```

Output:- In arrow function (nothing) → (because this belong to global window)

```
var obj = {
    name: "Pixush",
    display: function() {
        console.log(this.name);
    },
};

var obj1 = {
    name: "ABC",
};

obj.display.call(obj1)
```

Output:-
ABC

DATE

Implement Caching / Memoize Function

```
const clumsySquare = ( ()=> {

    const cache = {};

    return (num1, num2) => {

        const key = `${num1} - ${num2}`;

        if (cache[key]) {
            return cache[key];
        }

        for (let i=1; i<=10000000; i++) {}

        const result = num1 * num2;
        cache[key] = result;
    }
})();

console.log("First Call");
console.time("First_times");
console.log(clumsySquare(9467, 7649));
console.timeEnd("First_times");
console.log("First Call");

console.log("Second Call")
console.time("Second_times");
console.log(clumsySquare(9467, 7649));
console.timeEnd("Second_timer");
console.log("Second Call")
```

DATE

```
const seperateCache = {}

const optimisedClumsySquare = (num1, num2) => {

    const key = `${num1} - ${num2}`;

    if (seperateCache[key]) {
        return seperateCache[key];
    } else {

        for (let i = 1; i <= 10000000; i++) {}
        cont result = num1 * num2;
        seperateCache[key] = result;
        return result;
    }
}
```

Output Based Question on Event Loop

```
console.log("a")
setTimeout(() => console.log("set"), 0);
Promise.resolve(() => console.log("pro")).then((res)
                                        => res());

console.log("b")
```

Output

a
b
pro
set

jsv9000.app (Paste code there)

DATE

Explanation:-

→ setTimeOut and Promise will always run after the entire code gets executed

→ Promise will be waiting in Microtask Queue and setTimeout will be waiting in Task Queue.

→ Task Queue has high ~~priority~~ priority then Micro task Queue so Promise will run First then setTimeout

Infinite Currying

```
function add(a) {
    return function (b) {
        if(b) return add(a+b);
        return a;
    };
}

console.log(add(5)(2)(4)(8)());
```

DATE

Implement this code

```
const calc = {

    total: 0,
    add(a) {
      this.total += a;
      return this;
    },
    multiply(a) {
      this.total *= a;
      return this;
    },
    subtract(a) {
      this.total -= a;
      return this;
    },
};

const result = calc.add(10).multiply(5).subtract(30)
console.log(result.total);
```

DATE

map vs forEach

```
const arr = [2, 5, 3, 4, 7]

const mapResult = arr.map (ar) => {
                    return ar+2;
                  });

const forEachResult = arr.forEach (ar) => {
                        return ar+2;
                      });

console.log (mapResult)
console.log (forEachResult)
```

Output:

```
[4, 7, 5, 6, 9]
undefined
```

→ map return us an new array do not modify the original array & forEach didnot return as anything.

→ To modify the original array in forEach replace return ar+2 ⟶ `arr[i] = ar+2;` `(ar,i) => {}`

→ we can chain stuff to map we can not do in case of forEach

Ex:- `.map(ar => { return 3}); filter((ar) => { return 3});`

DATE

null vs undefined

→ null is an actual value and undefined means that the variable is declared but it's not initialized

Ex:-

```
console.log (typeof null)
console.log (typeof undefined)
```

output:-

```
object
undefined
```

undefined vs notdefined

Ex:-

`let a;` `console.log(a)`	`console.log (a)`
output:- `undefined`	output - `a is not defined`

```
console.log (null == undefined)
console.log (null === undefined)
```

Output:-

```
true
false
```

DATE

Explain Event Delegation

→ In a Ecommerance website if we have to route us ~~on~~ within the page of click of the product we use Event Delegation

For Eg:-

~~shoes~~

```
<div id="products">
    <li id="shoes">shoes</li>
    <li id="shirt">shirt</li>
    <li id="wallets">wallets</li>
</div>
```

```
document.querySelector("#products").addEventListener
        ("click", (event) => {
    console.log(event)
    if (event.target.tagName === "LI") {
        window.location.href += "#"+event.target.id;
    }
})
```

Flatten the Array

```
let arr = [
    [1,2],
    [3,4],
    [5,6,[7,8],9],
    [10,11,12],
];
```

output:-

// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

DATE

→ approach 1

```
let flattened = [].concat(...arr);
console.log(flattened)
```

→ approach 2

```
console.log(arr.flat())      // flattened for 1 levels
console.log(arr.flat(2))     // flattened for 2 levels
```

→ approach 3

```
function customFlat(arr, depth=1){
    let result = [];
    arr.forEach((ar) => {
        if(Array.isArray(ar) && depth>0) {
            result.push(...customFlat(ar, depth-1));
        }else {
            result.push(ar);
        }
    });
    return result;
}

console.log(customFlat(arr, 2));
```

DATE

var vs let vs const

→ var is function scope and let / const are block scope.

→ we can initalize var multiple times but let/const cannot be initalized multiple time

for ex:-

var	let	const
var a = 5;	let a = 5	const a = 5
var a = 10;	let a = 10	const a = 10
output:- 10	output- Identifier 'a' has already been declared	output:- Identifier 'a' has already been declared

→ let can also be initalized multiple time

Ex:-

```
let a = 5

a = 10
```

Output:-

10

DATE

→ Initalization

Ex:-

var a;	let a;	const a;
output:- (fine)	output:- (fine)	output:- (missing initalizer in const declaration)

setTimeout output

```
function ac() {
    for (var i=0; i<3; i++) {
        setTimeout (function log() {
            console.log(i)
        }, i * 1000);
    }
}

ac()
```

output:-

③ 3

for output:-	replace var → let
0 1 2	(for(var i=0;) (for(let i=0;)

→ var is a function scope & let is a block scope

DATE

call, Apply and Bind

```
var person = {
    name : "Amit Singh",
    hello : function (thing) {
        console.log (this.name + "say hello" + thing);
    };
};

person.hello ("world")
```

output :-
Amit Singh say hello world

```
var alterEgo = {
    name : "Gautam Singh",
};

person.hello.call (alterEgo, "world");
```

output :-
Gautam Singh say hello world

```
person.hello.apply (alterEgo, ["world"]);
```

output
Gautam sing say hello world

DATE

```
const newHello = person.hello.bind (alterEgo);
newHello ("world");
```

→ Bind return the completly new function

composition Polyfill

```
function addFive(a) {
    return a + 5;
}

function substractTwo(a) {
    return a - 2;
}

function multiplyFour(a) {
    return a * 4;
}

const evaluate = compose (addFive, substractTwo, multiplyFour);

console.log (evaluate(5)); // 23
```

```
const compose = (...functions) => {
    return (args) => {
        return functions.reduceRight((acc, fn) =>
                                     fn(acc), args);
    };
};
```

initial value

```
const evaluate = compose(addFive, substractTwo, multiplyFour);

console.log(evaluate(5));
```

Output :-

23

Implement Debounce

→ ~~Out~~ Debounce means there are some heavy tasks in software development. Take calling an API, for example. Suppose we have an API that searches a list of users, and we can't afford to use it too often we want to search only when we have typed the whole search query.

```
const [searchTerm, setSearchTerm] = useState('');

const handleSearchDebounced = debounced((term) => {
    // Perform search operation here
    console.log("searching for", term);
}, 1000);

const handleChange = (event) => {
    const term = event.target.value;
    setSearchTerm(term)
    handleSearchDebounced(term);
};

<div>
    <input
        type="text"
        value={searchTerm}
        onChange={handleChange}
        placeholder="Search..."
    />
</div>

function debounce(func, delay) {
    let timeoutId;
    return function (...args) {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => {
            func.apply(this, args);
        }, delay);
    };
}
```

DATE

Pangram String

→ A pangram is a string contains every letter of the English alphabet.

```
function checkPangram(str) {
    str = str.toLowerCase();
    let alphabetPresent = new Array(26).fill(false);
    for (let i=0; i<str.length; i++) {
        let charCode = str.charCodeAt(i);
        if (charCode >= 97 && charCode <= 122) {
            alphabetPresent[charCode - 97] = true;
        }
    }
    return alphabetPresent.every(present => present);
}
console.log(checkPangram("The quick brown fox jumps over the lazy dog."));
```

DATE

→ charCodeAt() gives ASCII code of characters
→ 97 is the ASCII code of "a" and 122 is the ASCII code of "z"

Convert 12 hrs to 24 hrs

```
const convert12to24 = (time12h) => {
    const [time, modifier] = time12h.split(" ");
    let [hours, minutes] = time.split(":");
    if (hours === "12") {
        hours = "00";
    }
    if (modifier === "PM") {
        hours = parseInt(hours) + 12;
    }
    return `${hours}:${minutes}`;
};

console.log(convert12to24("02:02 PM"));  // 14:02
console.log(convert12to24("04:06 PM"));  // 16:06
console.log(convert12to24("12:00 PM"));  // 12:00
console.log(convert12to24("12:00 AM"));  // 00:00
```

DATE

Output question

```
const user = {

    name: "Amit",
    greet() {
        return `Hello, ${this.name}!`;
    },
    farewell: () => {
        return `Goodbye, ${this.name}!`;
    },
};
```

```
console.log(user.greet());        // Hello, Amit!
console.log(user.farewell());     // Goodbye, !
                                  (undefined → (this.name))
```

output

→ Normal function this called his own this in your case it will point to user object.

→ this value inside a arrow function always equal to this of the outer scope.