Using spread operators in object –

```
const obj1 = {1: "a", 2: "b"}
const obj2 = {3: "c", 4: "d", 1: "e"}
const obj3 = {...obj1, ...obj2}
console.log(obj3) → {1: "e", 2: "b", 3: "c", 4: "d"}
```

Object destructuring in JS –

```
const course = {
    courcename: "JS"
    price: "999"
    courseInstructor: "hitesh"
}

const {courseInstructor} = course
console.log(courseInstructor); ⟶ hitesh

const {courseInstructor: instructor} = course
console.log(instructor) → hitesh
```

JSON format -

```
{
  "name" : "hitesh" ,
  "coursename" : "JS in hindi" ,
}
```

Passing objects in functions -

```
const obj = {
    firstname : "akshat" ,
    lastname : "jain" ,
    age : 23
}

function getobj ( objgot)
{
    console.log ( objgot);
    console.log ( objgot.firstname);
}
```

→ { firstname : "akshat", lastname : "jain", age : 23 }

→ akshat

```
getobj ( obj)

getobj ( { firstname : "Avnish" ,
           lastname : "Singh" ,
           age : 21
} )
```

Passing array to functions -

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

function getobj(getot

function getarr(arrgot)
{
    console.log(arrgot) → [1,2,3,4]
    console.log(arrgot[2]) → 3
}

getarr(arr);
```

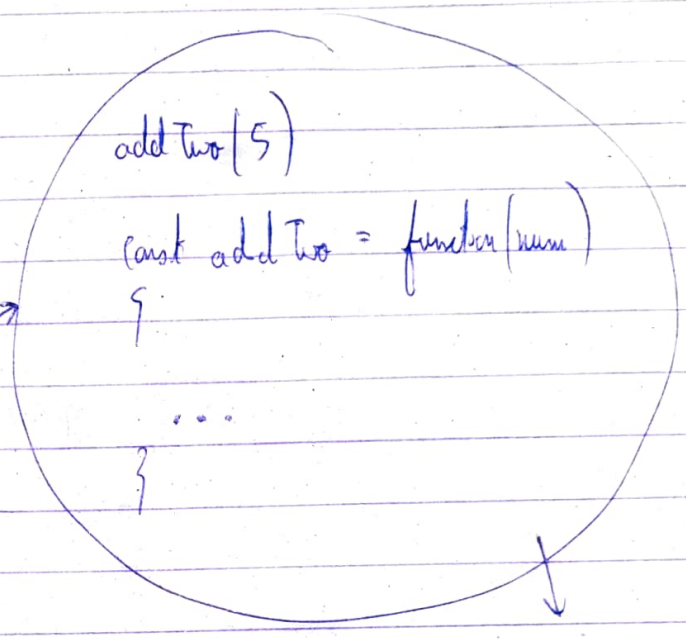
var vs let vs const -

```
if (true)
{
    let a = 10;
    const b = 20;
    var c = 30;
    d = 100;
}

console.log(a) → Throws error as out of scope
console.log(b) → Throws error as out of scope
console.log(c) → 30 (incase of var, variable is declared in global scope)
```

console.log(d) → 100 → This also have global scope and hence 100 will be printed.

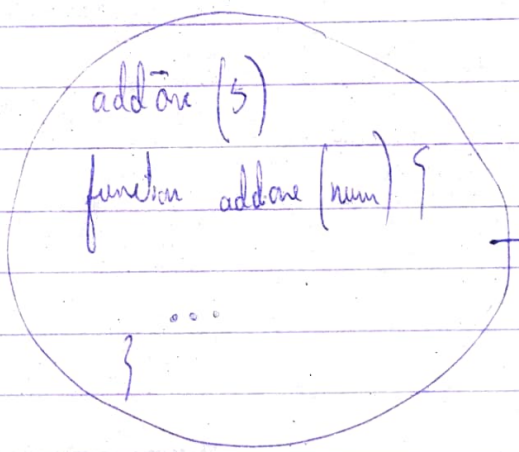
Different ways of writing a function -

① `const addTwo = function (num) {`
`return (num + 2)`
`}`
`addTwo (5)`



gives error

② `function addOne (num) {`
`return (num + 1)`
`}`
`addOne (5)`



→ gives output as 6

no error

③ Arrow functions -

① `const addTwo = (num1, num2) => {`
`return (num1 + num2)`
`}`

② `const addTwo = (num1, num2) => num1 + num2;`

③ `const addTwo = (num1, num2) => (num1 + num2)`

`console.log (addTwo (3, 4))` → (7)

①, ②, ③ All are same

Iterators in arrays -

```
const greetings = "Hello world"

for (const greet of greetings) {
    console.log(`Each char is ${greet}`)
}
```

→ gives each element

→

H
e
l
l
o

w
o
r
l
d

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

for (const i of arr) {
    console.log(i)
}
```

→ gives each element

→

1
2
3
4

Maps in JS -

The map objects holds the key value pairs and remembers the orginal insertion order of the keys.

```
const map = new Map()
map.set ('IN', "India")
map.set ('USA', "United States of America")
map.set ['Fr', "France")
```

```
console.log (map)  ⟶   'IN' => "India"
                       'USA' => "United States of America"
                       'Fr' => "France"
```

In JS, we can enter any combination of key value in the map, only constraint is that the key should be unique.

i.e.,
```
const map2 = new Map()
map2.set ( "1", "Akshat")
map2.set ( 2, [1,2,3] )
map3.set ( { name: "Akshat" }, 3 )
```
→ This will also get successfully inserted in a map.

Running a loop in map :-

~~for (const [key, value] of map)~~

```
for (const [key, value] of map)
{
    console.log (`Value of key is ${key} and the corresponding value is ${value}`)
}
```

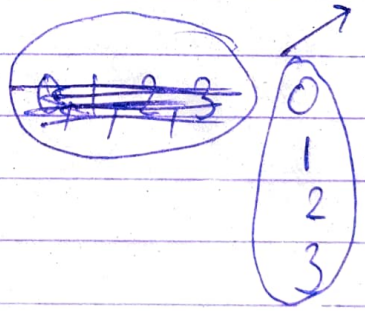
Note # We can't iterate over objects using for of

Iterating in objects -

For iteration in objects, we use ~~for of~~ for in loop instead of for of loop.

for in loop in JS -

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

for ( const i in arr)
{
  console.log (i)
}
```

→ 

gives the index of all the elements one by one in an array.

Using for in in Oject iteration -

```
const myObj = { name : "akshat"
                age : 23
              }
```

```
for (const i in myObj)
{
    console.log(`key is ${i} and value is ${myObj[i]}`)
}
```

for each loop in JS -

```
const coding = ["js", "ruby", "java", "python"]
coding.forEach(function(item){ console.log(item); })
```

- call back function (does not have any name)
- acts like an iterator which goes to each element of the array. We can name it anything.

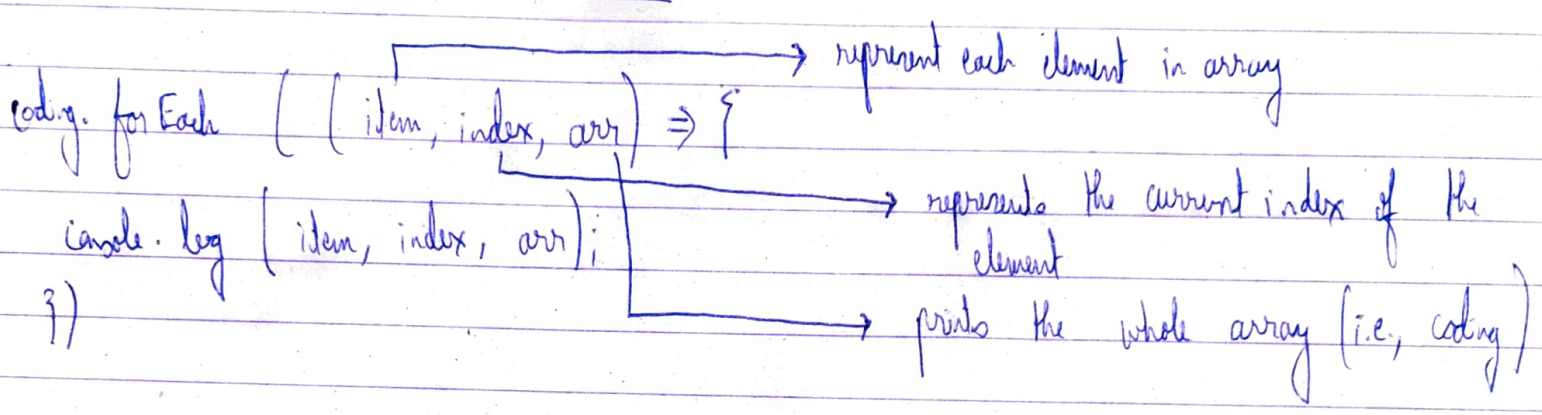
Using arrow fn

```
coding.forEach((item) => { console.log(item); });
```

If function is declared somewhere else -

```
function printMe(item){
    console.log(item);
}
coding.forEach(printMe)
```

Different parameters available in forEach -



#0 Real use of forEach()

When we have objects nested inside an array, forEach can be used to extract the object and its properties more easily.

eg- const arr = [

{ name: "Akshat",
age : 23
},

{ name: "Sam",
age : 27
},

]

arr. forEach ((val) ⇒ { console. log (val. name); })

Akshat
Sam

DOM manipulation

Filter() in JS -

Filter is used to give results based on some condition.

Eg -

```
const myNums = [1,2,3,4,5,6,7,8,9,10]

const newNums = myNums.filter( (num) => { return num>4 })

console.log(newNums)  → [5,6,7,8,9,10]
```

(filter → function; (num) => { ... } → callback fn; num>4 → condition)

If we want to use the same thing in for each -

```
const newNums = []

myNums.forEach( (num) => {

    if(num > 4){
        newNums.push(num)
    }

})

console.log(newNums)  → [5,6,7,8,9,10]
```

(Here, we have to give if condition)

Use of filter with array objects -

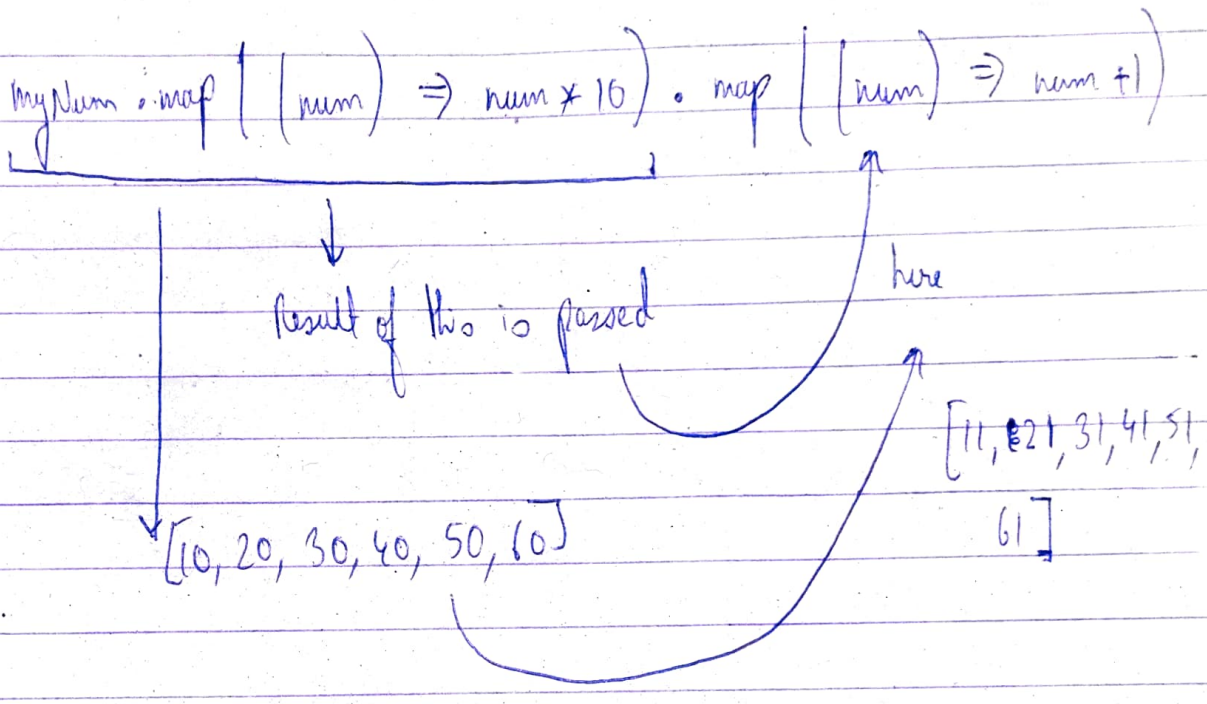
```
const books = [
  { title: 'Book One', genre: 'Fiction', publish: 1981, edition: 2004 },
  { title: 'Book Two', genre: 'Non-Fiction', publish: 1992, edition: 2008 },
  { title: 'Book Three', genre: 'History', publish: 1999, edition: 2007 },
  { title: 'Book Four', genre: 'Non-fiction', publish: 1989, edition 2010 }
]

const userBooks = books.filter( (bk) => bk.genre === 'History' )

console.log (userBooks) → { title: 'Book Three', genre: 'History'
                            publish: 1999, edition: 2007 }
```

map() in JS -

```
const myNumbers = [1, 2, 3, 4, 5, 6]

const newNums = myNumbers.map ( num => num + 10 )

console.log (newNums) → 11, 12, 13, 14, 15, 16
```

Chaining in JS -

```
const myNum = [1,2,3,4,5,6]

const newNums = myNum.map((num) => num * 10).map((num) => num + 1)
```




```
const newNums = myNum.map((num) => num * 10).map((num) => num + 1)
                .filter((num) => num > 40)
```

→ [41, 51, 61]

Reduce in JS :

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

const initialValue = 0;

const sumWithInitial = arr.reduce( (accumulator, currentValue) => (accumulator + currentValue),
                                   initialValue );

console.log(sumWithInitial)  ---> 10
```

Step 1 -

accumulator is set to initialvalue. i.e., 0.
current value = 1 (∴ arr[0] = 1)
accumulator = accumulator + currentvalue (0 + 1) = 1

Step 2 -

accumulator is 1
current value = 2 (∴ arr[1] = 2)
accumulator = 2 + 1 = 3

Step 3

a = 3
c = 3
a = 3 + 3

Step 4

a = 6
c = 4
a = 6 + 4

∴ sumwithInitial = a = 10

∴ O/P = 10

Events in JS -

① `<li> <img width="200px" id="owl" src="" onclick="alert('owl')" > </li>`

② 2nd Approach -

```
<script>
    document.getElementById("owl").onclick = function() { alert("owl") }
</script>
```

✓③ 3rd Approach -

```
<script>
    document.getElementById('owl').addEventListner('click',
    function() { alert("owl"); }, false)
```

false → event propagation

Learning events -

```
document.getElementById("owl").addEventListner('click', function(e) {
    console.log(e)
})
```

e → event object

Event propogation -

① Event bubbling - false (used mostly)
② Event capturing - true → used in some specific scenerios.

Eg -

```
<ul id="images">
    <li id="owl"> <img src="" > > </li>
</ul>

<script>
    document.getElementById('images').addEventListner('click', function(){
        consol.log("clicked inside the ul"); }, false)
    document.getElementById('owl').addEventListner('click', function(){
        consle.log("clicked on owl"); }, false)
```

O/P

clicked on owl
clicked inside the ul

② document.getElementById('images').addEventListener('click', function(){
console.log("clicked inside the ul"); }, true)

document.getElementById('owl').addEventListener('click', function(){
console.log("owl clicked"); }, true)

O/P

clicked inside the ul
owl clicked

To avoid execution of nested function,

stopPropogation() is used

document.getElementById("images").addEventListener('click', function(){
console.log("clicked inside the ul"); }, false)

document.getElementById("owl").addEventListener('click', function(e){
console.log("owl clicked"); e.stopPropogation(); }, false)

O/P

owl clicked → e.stopPropogation() stops execution of console.log() in "images"

```
document.getElementById("images").addEventListner('click', (e)=>{console.log(1);
e.stopPropogation(), true})

document.getElementById("out").addEventListner('click', (e)=>{console.log(2),
    true})
```

O/P

1 → 2 won't be printed as stopPropogation() is used.

---

preventDefault()

It prevents the default settings that trigger when we click on some particular component. Eg- a component may have be associated with some default settings to be triggered. Those will be ignored by using this fn.

```
<img href="" id="image">
<script>
    document.getElementById('image').addEventListner('click', (p)=>
    p.preventDefault();)
</script>
```

```
document.getElementById("image").addEventListener("click", (e)=>{console.log(1)})
```

After clicking on the image, we would be re directed to the href link given in the image tag.

Theory (Async)

JS → Synchronous - Code is executed one after other, only one thread is there.

Single Thread → One thread will execute all the work.

Default JS behaviour

Execution Context

- Executes one line of code at a time
- Next statement is not executed till the ~~the~~ time the older statement is not executed.

Blocking Code ↓ Block the flow of program ↓ Read file Sync

vs

Non-Blocking Code ↓ Do not block execution ↓ Read file Async

Timeout functions in JS –

```
<script>

    const sayHitesh = function () {
            console.log("Hitesh");
        }

    const changeText = function () {
        document.querySelector('h1').innerHTML = "best JS series";
        }

    setTimeout(changeText, 2000);

    setTimeout(sayHitesh, 2000);

    setTimeout(function () { console.log("Hello World"); }, 2000);

</script>
```

DOM

document.getElementById("id1") → <h1 id="id1" class="classh">H1</h1>

document.getElementById("id1").className → classh

Setting attribute with help of DOM --

document.getElementById("id1").setAttribute('class', 'classnew')

- "id1" → element to be targeted
- 'class' → attribute to be targeted
- 'classnew' → new attribute to be set

Before Para1

After Para1

(OR)

Overwrite →

Before Para1

After Para1

To overcome overwrite →

document.getElementById("id1").setAttribute("class", "something classnew")

o/p Para1

id = "id1" >

  this is inside of span this is inside para

document.getElementById("id1").innerText → this is inside span this is inside para

document.getElementById("id1").innerHTML → this is inside span this is inside para

Main diffrence b/w both -

We can inject HTML elements using innerHTML and not with innerText.

Eg- ① this is inside para

document.getElementById("id2").innerHTML = "<h1> this is inside para </h1>"

o/p

this is inside para → h1 is applied on the text

When we use innerText,

document.getElementById("id2").innerText = "<h1> This is inside para </h1>";

→ <h1> This is inside para </h1> → As we can see, the html tag is converted to string and then is placed inside the paragraph.

innerText vs textContent -

<h1 id="title" class="heading"> DOM learning test text </h1>

- document.getElementById("title").innerText → DOM learning
- document.getElementById("title").innerContent → DOM learning test text.

Query Selector -

Eg. <h1> Akshat </h1>
<h1> Jain </h1>
<h1> LG </h1>

document.querySelector('h1') → <h1> Akshat </h1>

↳ Returns the first h1 element.

document.querySelector('#title') → DOM manipulation

↓
selecting id

document.querySelector('.heading') → DOM learning

Q Use query selector to change the color of the first list item in the ul.

```
<ul>
    <li> First </li>
    <li> Second </li>
    <li> Third </li>
</ul>
```

Ans - ~~document.getQuery~~ document.querySelector("ul") → Selects the first ul in the document.

const oli = oul.querySelector("li") → Selects the first li in the first ul of the document.

oli.style.backgroundColor = "green" → Sets the background color of the first li of unordered list to green color.

Q. Changing the color of the second element in the ul -

const ul = document.querySelector('ul')

const secondli = ul.querySelectorAll('li')

→ of li will be stored [li, li, li]

NodeList

secondli[0].style.color = "green" → Changes the color of second list item

Better example-

```
let myPromise = new Promise((resolve, reject) => {
    let success = true;
    if (success) {
        resolve("Operation Successful");
    } else {
        reject("Operation failed");
    }
});

myPromise.then((message) => {
    console.log(message);
}).catch((errorMessage) => {
    console.log(errorMessage);
});
```

Message stored in resolve() is passed

message stored in reject() is passed