# Objects in JS

Date
Page

* Intro :-

Two Types of data types in JS

① primitive (val Type) - 7 ( num, bigint, string, bool, null, undefined, symbol)

② Non primitive ( Refrance Type) (object, arrays, functions)

* call by val return copy of val & call by refrance return refrance / memory address

* JS objects :-

- An collection of ~~key pair~~ key-value pair
- Each key is a property name (String) & value can be any data

① Creation of objects

```
let peeson = {
    name : "Rao",
    age : 23,
    islogged : false
}
```

② Accessing object values

```
console.log (peeson.name)      // dot notation
c.log ( peeson["age"] )        // Bracket not
```

③ add / update / delete

```
peeson.city = "pune" ;      //add
peeson.age = 24 ;           // update
delete peeson.name ;        // delet
```

④ Writing functions :- (in js)

① function declaration

```
function functionName(parameter){

}
```

```
function greet(name){
    console.log(name + "Hello")
}
```

* can be called before define (Hoisting)

② function expression

```
const functionName = function(para){

}
```

```
const add = function(a,b){
    return a+b
}
```

* Stored in var
* cannot call before function expr.
(not Hoisting)

② Arrow function

```
const functionName = (parameter) => {

}
```

ex:-

```
const multiply = (a,b) => a*b
```

- very useful for short fun
- No own (this) property in oop

| * Now wriing fun inside object |
|---|

- Objct with function

```
let car = {
    brand : "Toyota",
    start : function () {
        c.log("start")
    }
}

// car.start()
```

Default parameter :-

```
* fun add (name) {
    c.log(my name is ${name})
}
```

argument

c.log(lambdaji)

c.log( ) — empty will return undefined

=> my name is undefine so to pass this errore we write

so जेव्हा value येणार नाही तेव्हा हे msg दाखवेल fun ला default value द्या → fun add (name = "enter val") {

}

⑤ Nested objects:-

```
let user={
    name : "Ram",
    address : {
        city : "pune"
        pin : 411041
    }
}
// c.log ( user.address.city )
```

⑥ Looping through object

```
for (let key in person) {
    c.log ( key , person [key] )
}
```

⑦ Check property → (c.log( "age" in person ))

⑧ object methods :-

| | Return |
|---|---|
| ① Object.keys (objname) | keys ] string |
| ② Object.values (objname) | values |
| ③ Object.entries (objname) | array array of [key,val] |

* Object.freeze(obj) & Object.seal(obj)

① object.freeze() :-

cannot → add, remove, change/update

② object.seal()

cannot → change existing property val

can → ~~add, delete~~ only update val

# Array in JS :-

① Array :-
is special variable to store multiple values of multiple data types at one space

```
let fruits = ["apple", "banana", 23, true, null]
```

② Element access, modify

```
fruit[1]
fruit[2] = "kiwi"
```

③ Array length

```
fruit.length    // 5
```

④ Add remove items :-

| | | |
|---|---|---|
| push() | Add to end | arr.push("val") |
| pop() | Remove from end | |
| unshift() | Add to start | arr.unshift("val") |
| shift() | Remove from start | |

⑤ Looping thraigh arrays :-

① for loop

```
for (let i = 0; i < arr.len ; i++){
    console.log( arr[i] )
}
```

② for... of loop

```
For (let i of arr) {
    console.log( i )
}
```

* Imp ~~property~~ method of array in JS :-

① pop()
② push()
③ unshift()
④ shift()
⑤ concate() ⇒ let c = a1.concate(a2) | a1.lonCat(a2,a3
⑥ indexOf() arr.indexOf("Element name") not found -1
⑦ includes() arr.includes("Elmt") t,f
⑧ reverse() arr.reverse()
⑨ sort()
⑩ slice *
⑪ splice()

① sort() :-

* a] → let fruit = ["apple", "chery" "banana"]
r.log(fruit.sort())
It will sort as a → b

* if like apple, Banna (different capital")
It not works

* b] → problem with number
[10,5,2,20] sort → 10,2,20,5 X

c] → Correct number sort

To sort properly pass function to sort()

```
arr.sort(
        function(a,b){
            return a-b;
        }
    )
```

d] → [2, 5, 10, 20]

* small to big => a-b
* big to small => b-a

d] → sort string by length

```
arr.sort(
        function(a,b){
        return a.leng - b.long
        }
    )
```

→ [bat, cat, apple, dolphin]

* Slice vs splice

— slice :-

या मध्ये original array change होत नाही

arr.slice ( startInd , endInd-1)

arr = [ "a" , "b" , 'c' , 'd' , 'E' , 'f' , 'j' ]

0 1 2 3 4 5 6

* If I want only → c

arr.slice ( 2 , 3)

↳ 3 - 1 = 2

* If I want → c, d, E

arr.slice ( 2, 5 )

↳ 5-1 = 4

— splice :-

हा deted array element ची array return करतो

① Start index
② Delete How many items
③ optical insert

a) Remove element

let arr = [ a , b , c , d , e ]

0 1 2 3 4

~~arr.splice (0, 2) → a, b~~

~~Length = 1~~

* arr.splice (0 , 2))

start ↳ How many del from start index

[ a, b, c ] deleted

c.log (arr) → [ d, e ]

b) Add items :-

arr.splice ( 1, 0, "x", "y" )

— start index 1 पासून पुढे 0 काही delet न करता x, y add करणे

[ a, x, y, b, c, d, e ]

2) Replace (delet + add)

arr.splice ( 2, 1, "z" )

arr = [ a, x, y, d, e ] → [ a, x, z, de ]

# Multidimensional array :-

let arr = [ [ ], [ ], [ ] ]

array inside array

```
[ [ 1, 2, 3 ],
  [ 4, 5, 6 ],
  [ 7, 8, 9 ], ]
```

- Access :-

  [Raw, Col]

  arr [ 1, 2 ] => 6

- loop :-

```
for (let i = 0; i < arr.leng ; i++) {
    for (let j = 0 ; j < arr[i].length ; j++) {
        ~~console.log( arr[i][j]~~
        console.log (arr [i] [j]
    }
}
```

- Examples.
  - Tic Tae Toe
  - cross board
  - Table data in website

* Deep copy ve Shallow copy

① Shallow copy :-

- A shallow copy occurs when you copy the refrance of an object to a new variable
- only top level properties are copied while nested objects / array are still refrance to the orignal memory location

| array :- | objects :- |
| --- | --- |
| | let obj = { name = "Ram" ; |
| let arr = [ "Ram", 23, true] | age = 23 ; |
| let arr2 = arr. | address { |
| | city : "Pune" |
| | } |
| | } |
| | let obj 2 = obj 1 — shallow copy |
| | obj 2. age = 24 also change in obj 1 |

| | |
| --- | --- |
| 1) Spread operator | 1) Spread operator |
| let copy = [... arr] | let obj 2 = { ... obj } |
| — Shallow | But nested not get copied |
| ② Slice () | ② Assign |
| let copy = arr1. slice () | let obj 2 = Object. assign ({}, obj 1) |
| — Shallow | |
| ③ ~~copy~~ = map | |
| let copy = arr. map ( x => x) ; | |
| — Shallow | |
| ④ JSON method | |
| let copy = JSON. parse (JSON. ~~straight~~ stringify (arr) ; | |
| — deep | |

* Loops in JS

1. for loop
2. while loop
3. do while loop
4. for ... of loop
5. for ... in loop

1) for loop :-

- Use when you know how many times to loop
- Good for index based loops

```
for (let i=0 ; i< arr.length ; i++){
    console.log()
}
```

2) while loop :-

- Used when you don't know end point exactly
- Runs as long as condition

```
let i= 0;
while (i<5){
    console.log()
    i++
}
```

3) do while loop :-

- Runs at least once even condition is false
- Runs once minimum
- used to you to execute before check

```
let i= 0
do{ console.log(i);
    i++;
} while(false)
```

even false is runs for one

o/p => 0

4) for..of loop :-

- best for array
- Used to loop arrays, string
- loops values directly
- not used for objects ✗

```
let fruits = ["apple", "banana", "mango"]
for (let f of fruits){
    console.log(f);
}
```

5) for..in loop:-

- used to loop over objects keys
- loop over keys only

```
let obj = { name: "Ram"
            age: 22 }

~~for (let key in use~~
~~for (let key in ob~~
for (let key in obj){
    console.log (key, obj[key])
}
```

o/p:- name : Ram
age : 22

* Hoisting :-

JS moves declaration to the top of the scope before running code

| Type: | Hoisted | Usable before declare |
|---|---|---|
| — var | yes | ✓ but undefined |
| — let/const | yes | ✗ error TDZ |
| ✓ — function declaration | Yes | ✓ |
| — function expression | partially | ✗ |

* function declaration is safely hoisted

① var :-

```
console.log(a)   // undefined
var a = 10
```

declaration is Hoisted but not val

② let/const are Hoisted but in temporal dead zone

```
console.log(a)   ✗ error
let/const a = 10
```

③ function dec ✓ works

④ function expression wit (var, let, const) not Hoisted

```
console.log(
sayHi();   // error
var sayHi() = function() {
    console.log("Hii");
}
```

# Higher order function:- Imp Topic

① function as object:-

JS functions are objects with special power (callable)

greet();

we can add properties like greet.lang = "Hindi";

- function can Have properties, passed/store, return from other fun

② What data pass to function():-

```
function show(data){
    c.log(data);
}
show(10)
show("Hi")
show([1,2,3])  //array
show({a:1})    //object
show(()=>{})   //function
```

③ Call back function:- Imp

used for event, async, array methods

- A call back is a function passed as an argument to another function

④ Higher order function:- Imp

takes function as input, returns a function or both

- map, filter, reduce)

* setTimeout() in js :-

runs a function after a delay

```
setTime(callback, delay in ms, arg1, arg2 ..);
```

function to run — 1000ms = 1s

ex:-

```
setTimeout(
    function(){
        console.log("Hello after 2 sec");
    }, 2000
);
```

// it prints after 2 sec

— Run once

— Async — JS won't wait, continues code

— cancel it by [ clearTimeout() ] to stop

ex :- आपण प्रत्येक setTimeout() चा id माहिती पाहिजे to आपण clearTimeout(id) pass करतो. So हा id browser मध्ये दिसतो

Buffer code editor we can in node js do like --

```
let var1 = setTimeout( function(){console.log("one")}, 5000 )
let var2 = setTimeout (fun(){console.log("two")}, 6000)
clearTimeout(var2)
```

या मध्ये मुळे फक्त var चा setTimeout print "one" after 5sec will display

* setTimeout fun मध्ये nodejs मध्ये जाणारा callback function pass करावा लागतो

- Browsers मध्ये आपण string टाकू शकतो (console)

* How to pass argument in setTimeOut()

function add

```
function add (a,b) {
    return a+b
}
```

→ setTimeout ( add , 1000 , 10, 5)

* Argument keyword
  - special object inside normal fun^n
  - It holds all argu passed to fun^n even not written parameter
  - Not available in arrow fun^n

```
function showAll () {
    console.log(arguments)
}
```

showAll ( "name", 120, true, obj={} )

o/p :- { 0 : name
1 : 120
2 : true
3 : obj }

* Rest parameter keyword ( ...args)

```
function showAll (...args) {
    console.log (args)
}
```

0 1 2 3 4
showAll (10, 12, 13, 14, true)

If you want perform operation

```
function add (...args) {
    console.log ( args[0] + args[1] )
}
```

// 10+12 = 22

Problem :-

Nao combine setTimeOut(), Set Interval & ~~setCl. clearTimeout~~ clearInterval() to start program for a particular (repeats) time interval & then stop after some given time

Concepts

* clearIntervals() | clearTimout() uses id of setIntval, Tinout
  =
* SetInterval, Tinout use call back argument fun() fun()

```
function greet () {
        clg ("will be run after every 2 sec")
}

let id = setInterval ( greet() , 2000 )

setTimeout (
    function () {
    clearInterval (id)
    } , 5000
)
```

Set करना

clear करना..

# * Event Loop & callback Queue in JavaScript

## * Event loop :-

JS runs one line at a time (single threaded) but still Handle delayed or Async tasks like (setTimeout()) using Event loop

म्हणजे जो code आहे त्यामधील simple sync code असेल की normal run call, c.log() हे Run होतात But महत्वाचे Async fun असतो तर ते web API कडे जात (Handle होतात) आणि समजा हे wait करतंय पूर्ण झालं तर ते call stack कडे जाते आणि exeact होता तर call stack busy असेल/empty नसेल तर हे callback queue मध्ये जाते आणि call stack empty झाला की call back to stak जाते.

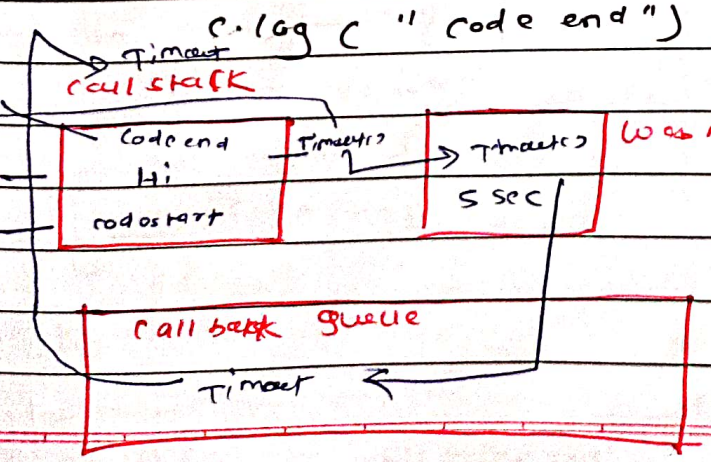
ex:-

```
c.log("code start")

function greet() {
    c.log("Hi")
}

setTimeout( function() {
    "This is setime")
}, 5000
)

c.log("code end")
```



Date
Page

* Main concepts:-

call stack → where normal code runs line-by-line
Web API → Browser/node Handles Timer (setTimeout(), fetch)
callback queue → stores async tasks ready to run
Event loop → middleman checks if stack is empty, then runs queued tasks

① JS runs top to Bottom
↓
② setTimeout fetch send to web API
↓
③ After time done/API done → result goes to callback queue
↓
④ event loop checks if stack empty move callback to stack

# Closure

* A function remembers variables from the place where it was created - even if called later or outside that place

When function is created inside another function, it remembers the variables from the outer function even after outer function finishes.

* ~~unlike~~ a normal function हे नेहमी call केल्या जाता नेहमी एखादा task/use complete झाल्या ने delete होतो. झाणि त्याचे var ही delete होतात.

```
function normal() {
    let a, b;
    a = 10;
    b = 20;
} return a+b;
```

- When finishes function, variables like a, b are removed from memory (garbage collected) because nothing using them any more.

* Closure

```
fun outer() {
    let x = 100;
    fun inner() {
        console.log(x);
    }
}
let show = outer()
show() // still print 100
```

या मध्ये inner हे outer चे var use करतो आहे outer ची scope संपली/use संपला तरी त्याचे var remember करतो inner or function called at that scope.

* fun outer() is Higher Order fun — it return fun

# Method vs function in js

| | function | method |
|---|---|---|
| - | Normal Block of code you can call | - function inside object, that is property of an object |
| - | called by just by name or refrence | - called using object name (obj.method()) |
| - | scope - Global/local | - scope - object |
| - | ex:- function add(){ } | obj.greet = fun(){ } |

* All methods are functions But All functions are not methods
* they can use "this" to access obj properties

ex:- method :-

```
let obj = {
    age : 24,                    // Obj.function();
    greet : function(){
        key :val console.log("Hi")
    }
```

ex: modern way to write method

```
let obj = {
    age: 24,
    greet(){
        console.log("Hi")
    }
}
```

# Arrow function

- function keyword not used
- No .this, arguments, super, new.target (lexical)
- can not use argument

- one line return

```
let double = x => x*2;
```

- Block :-

```
let greet = name => {
        console.log("Hi")
    }
```

- returning object

```
let getUser = () => (
                { nom: "Lambs",
                  age: 3
                }
            )
```

* when use arrow function :-
  - for callbacks ( map, filter, forEach, etc)
  - one-lines
  - when you don't need this

# for ... of & for ... in

* for ... of :-

- works on array, strings, maps, sets
- Loop over values
- output :- Each element / val

ex *

```
for...of (values)
let fruits = ["apple", "banana", "mango"];
    for (let fruit of fruits){
        console.log(fruit);    // apple, banana, mango
    }
```

* for ... in :-

- loop over keys
- objects, arrays
- output :- each key, index

ex:-

```
for...in (keys)

let person = {
        name : "Ram",
        age : 23
    }

~~for (let obj~~

for (let key in person){
        console.log (key)
        console.log (person[key])
    }
```

↳ This return val / Bracket notation

# forEach method

- built in array method (arr.forEach() { })
- take callback function that runs every element

```
arr.forEach (element, index, arry) {
});
```

current val | index of | Full array

or

```
arr.forEach ((element, ind) => { })
```

ex:-

```
let nums = [10, 20, 30]
nums.forEach ((num, i) => {
    console.log ("val:", num, "index:", i)
})
```

o/p: val : 10 index 1
val : 20 index 2
val : 30 index 3

By default पहिला parameter → value / element
दूसरा → index
तिसरा → complete array होगा

* जेवढे elements तेवढे वेळा Repeat होतो
* for return nothing.

# Map, Filter & Reduce :-

Theory :-

- यापण foreach method पहिली हा array चा elements चा, index चा iterate होतो But हा return काही करत नाही हा फक्त यापण console log करुन o/p देतो.

- So for get return & some modification in array to make operation simple we use map, filter, reduce JS array methods.

- The map(), filter(), reduce are array methods that transform the array according to the applied fuction & return the updated arr.

- Used to write simple, short, clean code for modifying arrg instead of loops

- let arr = [10, 20, 30, 40]

① map() :-

(element, index, arr)

```
arr.map( ( ) => { } )
```

↑ callback fun

(element, index, arr)

```
arr.map( function() {
} )
```

② filter()

ele, index, arr

```
arr.filter( ( ) => { } )
```

condition

③ reduce()

```
arr.reduce( (accumulator, current) => { } )
```

Concept :- Arrow function syntax rule

① 1 line, 1 param → let squar = x => x * x

② multiple parameter → (a,b) => a+b

③ Block body → (a,b) = {
return a+b
}

* Return mandatory

④ Return obj → ( ) => ( {
nam : "ram"
} )

wrap obj in ( )

* if {} use → Return must

Date
Page

① map (arr.map()) :-

- run function on each element
- Return new array with modified values / Transform

```
let arr = [1, 2, 3]
let result = arr.map(
                function(el, ind, arr) {
                  return el x 2
                })
```

```
arr.map(
  (el) => {
    el x 2
  })
```

o/p :- [2, 4, 6]

Use :- Transforming data (API, UI)

```
let useEnames = ~~user.names~~
              = user.map(user => user.name.ToUpperCase());
```

② filter (arr.filter) :-

- Runs condition on each element

ex :- show active / stock of product

```
let result = arr.filter(
                function(el, ind, arr) {
                  el >= 10
                })
```

```
product.filter(
  item => item.stock)
```

③ Reduce (arr.Reduce(acum, curr)) :-

- Takes all values & reduce to one
- 2 arg accumulator & current

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

This is arr val

This is start point of accumulator

* Method chaining:-

- using one method after another
- It works because each method return an array so we can call another method on it

```
let result = arr.map( n => n*2
                .filter( n => n>5)
                .reduce( (sum,n) => sum+n ,0)

console.log(result)
```

# some(), every() method in array

## some(), every()

① some() :-

→ checks if at least one elements match the conditions

→ return true, false

ex :- check if any user is admin
if any product out of stock

- these call back take three val as prev method
(element, index, arr[])

ex :- let hasEven = nums.some((n) => n%2 == 0);
// true

② every()

- checks all elements matechs the conditions

- return t, f

let num = [2, 4, 6]
let allEven = num.every((n) => n%2 == 0);
// true

ex :- check if all items in cart are in stock

ES6+ feature :-

# arguments keyword :- in function

- It is built in object available in normal function
- It contain all passed arguments, even if not declared in function

- Return :-

- argu is not an array but its an array-like object

```
function add() {
        console.log(arguments) //
return arguments[0] + arguments[1]  //
}

add(100, 200, 300, 4, 5, 6);
```

~~* argument~~

* Rest parameter :-

```
function (...args) {

        args[0] + args[1];

}
```

* arguments not support by arrow fun^n but it support args.

Here's a clear and concise note on **ES6+ JavaScript features** you mentioned:

---

## ☑ 1. Default Parameters

- **Purpose**: Allows function parameters to have default values if no value or `undefined` is passed.
- **Syntax**:

```
function greet(name = "Guest") {
  console.log("Hello, " + name);
}

greet();           // Hello, Guest
greet("Alice");    // Hello, Alice
```

---

## ☑ 2. Spread Operator (...)

- **Purpose**: Expands (spreads) elements of an array or object.
- **Used in**: Copying, merging, passing arguments, etc.
- **Syntax**:

```
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
console.log(arr2); // [1, 2, 3, 4]

const obj1 = { a: 1 };
const obj2 = { ...obj1, b: 2 };
console.log(obj2); // { a: 1, b: 2 }
```

---

## ☑ 3. Rest Parameters (...)

- **Purpose**: Collects multiple arguments into a single array.
- **Used in**: Functions to handle unknown number of arguments.
- **Syntax**:

```
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b, 0);
}

console.log(sum(1, 2, 3)); // 6
```

---

## ☑ 4. **Block Scope (`let`, `const`)**

- **Purpose**: Variables declared with `let` and `const` are **block-scoped** (only accessible within the block `{}` they are declared).
- **Why important**: Avoids hoisting issues with `var`.
- **Syntax**:

```
{
  let x = 10;
  const y = 20;
  console.log(x, y); // 10 20
}
// console.log(x, y); // Error: x is not defined
```

---

## ☑ 5. **Lexical Scope**

- **Meaning**: Scope is determined by the physical location in the source code.
- Inner functions can access variables from outer functions.
- **Syntax**:

```
function outer() {
  let outerVar = "I'm outer";

  function inner() {
    console.log(outerVar); // Accesses variable from outer
  }

  inner();
}

outer();
```