**Note:- You have to prepare a google drive sheet name it as Javascript interview questions keep all theoretical points as question answer form**

**==========================================================**

**https://shorturl.at/sMS17**

**call, apply, bind**

**Closure**

**Data Structures in JS**

**IIFE**

**Storage Management**

**Array Manipulators 01**

**==========================================================**

==========================================================

call, apply, bind

==========================================================

call():-

- This function is used to create relationships between two unknown memory locations.

apply():-

- This is same as call function

- When we have to pass arguments as an array this function is used.

(array implies no independent arguments)

bind():-

- this function is used to merge two unknown memory locations.

- this function returns new function

let obj = {

num: 10

}

console.log(obj)

function myFun(arg) {

return this.num + arg

}

console.log(myFun.call(obj, 10))

function newFun(arg1, arg2, arg3) {

return this.num + arg1 + arg2 + arg3

}

console.log(newFun.call(obj, 20, 30, 40))

let arr = [10, 20, 30]

console.log(newFun.apply(obj, arr))

let bindFun = newFun.bind(obj)

console.log(bindFun(20, 30, 40))

==========================================================

Closure

==========================================================

- Inner function can have access of data from outer function.

- Outer function returns inner function.

- Closure means inner function can have access of data

from outer function even after returning inner function.

function addn(x) {

return (y) => {

return x + y

}

}

//here outer function returned inner function

let var\_a = addn(5)

let var\_b = addn(10)

/\*here we called returned inner function with access of variable

from outer function\*/

console.log(var\_a(2))

console.log(var\_b(4))

==========================================================

Datastructures in JS

==========================================================

i) Map

ii)WeakMap

iii)Set

iv)WeakSet

let obj = {}

//let key1 = {}

let key1 = { k1: 'v1' }

let val1 = `Hello\_1`

obj[key1] = val1

//console.log(obj)

//let key2 = {}

let key2 = { k2: 'v2' }

let val2 = `Hello\_2`

obj[key2] = val2

console.log(obj)

Problem with JSON

- if we are having key as object for more than one key

value pairs, latest value overrides previous value.

- to overcome this issue Map() and WeakMap() are used

Note:- to check the internal structure 'dir()' function is used.

i) Map()

- size

- Map()

- get()

- set()

- has()

- delete()

- clear()

- keys()

- values()

let map = new Map()

console.dir(map)

//Eg01

let key1 = {}

let key2 = {}

let val1 = `Hello\_1`

let val2 = `Hello\_2`

map.set(key1, val1)

map.set(key2, val2)

console.log(map)

//Eg02

map.set(`key1`, `Hello1`)

.set(`key2`, `Hello2`)

.set(`key3`, `Hello3`)

.set(`key4`, `Hello4`)

.set(`key5`, `Hello5`)

console.log(map)

console.log(map.size) //5

console.log(map.keys()) //MapIterator {'key1', 'key2', 'key3', 'key4', 'key5'}

console.log(map.values()) //MapIterator {'Hello1', 'Hello2', 'Hello3', 'Hello4', 'Hello5'}

console.log(map.has(`Key5`))//false

console.log(map.get(`key5`))//Hello5

map.delete(`key5`)

console.log(map)

for (let [k, v] of map)

console.log(k, v)

map.clear()

console.log(map) //Map(0)

//Eg03

map.set(`Key1`,`Hello\_1`).set(`Key2`,`Hello\_1`)

console.log(map) //duplicate values allowed and accepted

map.set(`Key3`,`Hello\_3`).set(`Key3`,`Hello\_4`)

console.log(map) //duplicate keys allowed older values replaced with new one

ii)WeakMap

- it wont allow primitives as keys

- WeakMap()

- delete()

- get()

- set()

- has()

let wm = new WeakMap()

//console.dir(wm)

let key1 = {}

let val1 = `Hello\_1`

wm.set(key1, val1).set({}, `Hello\_2`)

console.log(wm)

//wm.set(`key3`,`Hello\_3`) //TypeError: Invalid value used as weak map key

iii)Set:- duplicates are discarded

- Set()

- has()

- add()

- delete()

- clear()

- values()

- keys()

let set = new Set()

//console.dir(set)

//Eg01

set.add(10)

.add(20)

.add(30)

.add(10)

.add(20)

console.log(set)

//Eg02

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

let set = new Set(arr)

console.log(set) //Set(4) { 10, 20, 30, 40 }

//Eg03

let set = new Set()

set.add(10)

.add(20)

.add(30)

.add(40)

.add(50)

console.log(set) //Set(5) {10, 20, 30, 40, 50}

set.delete(50)

console.log(set) //Set(4) {10, 20, 30, 40}

console.log(set.has(40)) //true

console.log(set.keys()) //[Set Iterator] { 10, 20, 30, 40 }

console.log(set.values()) //[Set Iterator] { 10, 20, 30, 40 }

for(let x of set)

console.log(x)

set.clear()

console.log(set) //Set(0) {}

iv)WeakSet():-

- It wont allow primitives

let ws = new WeakSet()

//ws.add(10) //TypeError: Invalid value used in weak set

let key1 = { data: 10 }

let key2 = { data: 20 }

ws.add(key1).add(key2)

console.log(ws)

==========================================================

IIFE:-

==========================================================

- Immediately invoked function expression.

- Introduced in ES9.

- These are self invokable functions, i.e. no need to call IIFEs.

- Syntax

(()=>{})()

//Eg01

(() => {

console.log(`Welcome to IIFE`)

})()

//Eg02

((arg1, arg2) => {

console.log(arg1 + arg2)

})(10, 20)

//Eg03

let res = (()=>{

return `Good Afternoon`

})()

console.log(res)

==========================================================

Storage Management

==========================================================

Local Storage, Session Storage

- Local storage is persistent storage of browser.

- Session storage is temporary storage of browser, till

that session only.

- We can store data in the form of key and value pairs.

- It supports only string data.(to store objects stringify them)

- Keys are unique

- All functions are common for local and session storage.

- setItem() function is used to store item.

- getItem() function is used to read item.

- removeItem() function is used to delete item.

- localStorage and sessionStorage belong to the 'window' object.

=========================================================

Array Manipulators

=========================================================

**01. map():-**

- **This function is used to manipulate each and every element in array**

- **it returns an array**

//Eg01

let arr1 = [10, 20, 30, 40, 50]

//multiply each element by 2

console.log(arr1.map((element, index) => {

return element \* 2

}))

//Eg02

let arr2 = [1, 2, 3, 4, 5]

//o/p ['$1','$2','$3','$4','$5']

console.log(arr2.map((element, index) => {

return '$' + element

}))

//Eg03

let arr31 = [1, 2, 3]

let arr32 = ['one', 'two', 'three']

//o/p [ [ 1, 'one' ], [ 2, 'two' ], [ 3, 'three' ] ]

console.log(arr31.map((element, index) => {

return [element, arr32[index]]

}))

**02. filter():-**

- **this function creates array based on condition**

//Eg01

let arr1 = [10, 20, 30, 40, 50]

//create an array with elements greater than 30

console.log(arr1.filter((element, index) => {

return element > 30

}))

//Eg02

let arr2 = [10, 100, 20, 200, 30, 300, 40, 400, 50, 500]

//create array with elements greater than or equal to 100

console.log(arr2.filter((element, index) => {

return element >= 100

}))

//Eg03

let arr3 = [10, 20, 30, 40, 50]

//o/p [300,400,500]

console.log(arr3.filter((element, index) => {

return element > 20

}).map((element, index) => {

return element \* 10

}))

**03. reduce() left to right 0 -> 1**

**04. reduceRight() right to left 0 <- 1**

//Eg01

let arr1 = [1, 2, 3, 4, 5]

console.log(arr1.reduce((fv, nv) => {

return fv + nv

}))

console.log(arr1.reduceRight((fv, nv) => {

return fv + nv

}))

//Eg02

let arr2 = [1, 2, 3, 4, `5`]

console.log(arr2.reduce((fv, nv) => {

return fv + nv

}))

console.log(arr2.reduceRight((fv, nv) => {

return fv + nv

}))

//Eg03

let arr3 = [`1`, 2, 3, 4, 5]

console.log(arr3.reduce((fv, nv) => {

return fv + nv

}))

console.log(arr3.reduceRight((fv, nv) => {

return fv + nv

}))

**05. forEach**

**06. for...of**

**07. for...in**

**08. push():- add element at end, returns new length of array**

**09. unshift():- add element at beginning, returns new length of array**

**10. pop():- remove element from end, returns removed element**

**11. shift():- remove element from beginning, returns removed element**

let arr = [20, 30, 40]

console.log(arr) //[ 20, 30, 40 ]

console.log(arr.push(50)) //4

console.log(arr) //[ 20, 30, 40, 50 ]

console.log(arr.unshift(10))//5

console.log(arr) //[ 10, 20, 30, 40, 50 ]

console.log(arr.pop()) //50

console.log(arr) //[ 10, 20, 30, 40 ]

console.log(arr.shift()) //10

console.log(arr) //[ 20, 30, 40 ]

**12. some():- if any one element in the array satisfies the condition then it will return true, otherwise false.**

**13. every():- if all elements in the array satisfy the condition then it will return true, otherwise false.**

let arr = [10, 20, 30, 40, 50]

console.log(arr.some((element, index) => {

return element > 10

})) //true

console.log(arr.every((element, index) => {

return element > 10

})) //false

console.log(arr.some((element, index) => {

return element > 50

})) //false

console.log(arr.every((element, index) => {

return element <= 50

})) //true

**14. find() :-**

- **this function is used to find an element in array**

- **if element found it will return the same element**

- **if an element is not found it will return undefined.**

**15. includes() :-**

- **it is boolean function used to check element is present in array or not**

let arr = [10, 20, 30, 40, 50]

console.log(arr.find((element, index) => {

return element == `30`

})) //30

console.log(arr.find((element, index) => {

return element === `30`

})) //undefined

console.log(arr.includes(30)) //true

console.log(arr.includes('30')) //false

**16. splice() -> swiss army knife for arrays**

**https://javascript.info/array-methods**

let arr = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

console.log(arr) //[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

arr.splice(5, 2) //from index 5 delete TWO elements

console.log(arr) //[10, 20, 30, 40, 50, 80, 90, 100]

//delete 80

arr.splice(5, 1)

console.log(arr) //[10, 20, 30, 40, 50, 90, 100]

//delete 100

//arr.splice(6, 1)

arr.splice(-1, 1)

console.log(arr) //[ 10, 20, 30, 40, 50, 90 ]

arr.splice(2, 2)

console.log(arr) //[ 10, 20, 50, 90 ]

//before 90 add 60, 70, 80

arr.splice(3, 0, 60, 70, 80)

console.log(arr) //[10, 20, 50, 60, 70, 80, 90]

//delete 50 and add 30, 40, 50

arr.splice(2, 1, 30, 40, 50)

console.log(arr) //[10, 20, 30, 40, 50, 60, 70, 80, 90]

//add 100 at end

arr.splice(9, 0, 100)

console.log(arr) //[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]