New JavaScript Mock questions

1. Deep Copy Shallow copy ✓
2. Flatten the Object ✓
3. Polyfill for Array.reduce() ✓
4. Closure in details ✓
5. Var, let, Const explaination and difference ✓
6. Null, Undefined, Undeclered difference ✓
7. Curring Concept ✓
8. Callback ✓
9. Promise, Promise.All implement ✓
10. async/await ✓

Shallow copy creates a new array but it pointes

① Deep copy implement

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

let b = JSON.parse(JSON.stringify(arr));
```

If we write like this → `let b = [...arr]` this will not work for deep copy

```
b[2].push(5)

console.log(arr) => [1, 2, [3, 4]]

console.log(b) => [1, 2, [3, 4, 5]];
```

Custom deep copy

```
Const deepCopy = (input) => {
  if(intut === null || type of input !== 'object'){
     return input;
  }
Const copiedval = Array.isArray(input)?[]:{};
  let keys = Object.keys(input)

for(let i=0; i<keys.length; i++){
  copiedval[keys[i]] = deepcopy(input[keys[i]]);
```

If array it will give index, If object It will give properties means keys

}

return CopiedVal;

Use

```
Const Copy = deepCopy(Obj)
Console.log(copy);
```

It Creates a completly new object with nested objects. This means that if you change the value of an element in new Object the Value will not change in original Object.

Shallow Copy

In Shallow Copy when we change any Value in new copy object then Original Object value will also change if the original object has nested Object inside it

```
Const obj2 = [... obj1]
```

---

* Flatten Object

Steps

(ii) make a function with three arguments (obj, parentkey, res = {})

(iii) Use For in loop to iterate on each key in obj

(iii) 'PropName' is constructed based on parent key. If parentkey exist the current key is appended with a dot as a seperator. Otherwise propName is just set to Current key.

(iv) Now there is a condition to check if the value associated with the current key is an object and not 'null'

(v) Then we write recursive code
flattenObj(obj[key], propName, res);

(vi) Then we store the value in accumulator obj 'res'

(vii) Finally we return the res

Code for flatten object

```
function flattenObj = (obj, parentkey, res={}) {

  for(const key in Obj){
```

```
const propName = parentKey ? parentKey + '.' + key : key
    if (typeof obj[key] === 'object' && obj[key] !== null) {
        flatten obj(obj[key], propName, res)
    }
    else {
        res[propName] = obj[key]
    }
}

console.log(flatten obj(obj));
```

---

* Array.Reduce Polygill

```
Array.prototype.myReduce = function (cb, initialValue) {

  let acc = initialValue

  for (let i = 0; i < this.length; i++) {
    acc = acc ? cb(acc, this[i]) : this[i];
  }
   return acc;
};

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

```
let res = arr.myReduce((acc, cur) => {
    return acc + cur;
}, 0);
```

```
Console.log(res);
```
⇒ 15 output

---

* Closure

A closure is a javascript feature in which the inner function has access to the outer function scope. Even after the outer function has finish executing The closure has access to the global scope also. This is also called lexical Scope

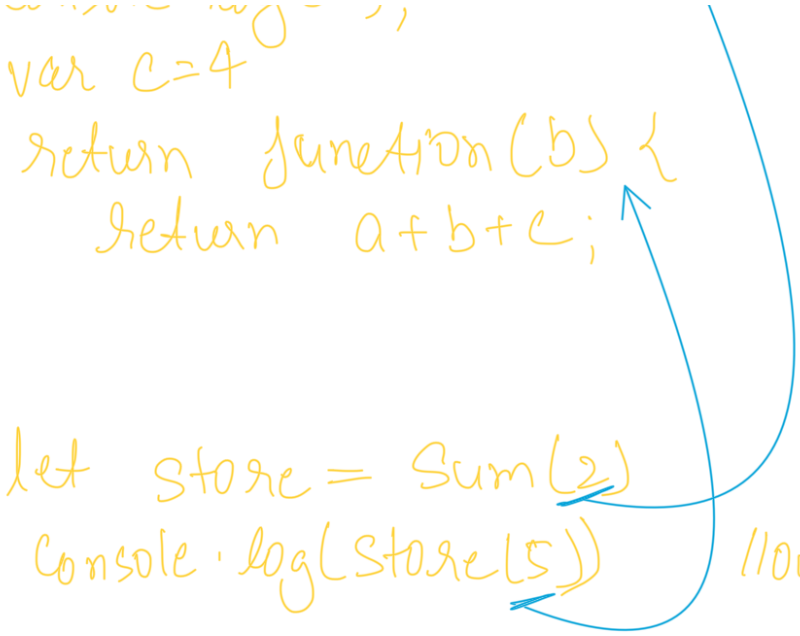
Closure has three scope
→ Local Scope
→ Enclosing scope
→ Global scope

Example code

```
Const sum = function(a) {
  Console.log(a);
```

```
var c=4
return function(b){
    return a+b+c;
}
}

let store = Sum(2)
console.log(store(5))    //output -> 11
```



2) Example 2

```
var sum = function(a,b,c){

return {
  getsumTwo: function(){
      return a+b;
  },
  getsumThree: function(){
    return a+b+c;
  }
}
}

var store = sum(3,4,5);
console.log(store.getsumThree()) // output -> 12
console.log(store.getsumTwo()) // output -> 7
```

* Var, let, Const

→ Var has function or global scoped means variable defined outside the function can be accessed globally and if it is defined inside a function is can be accessed only inside the function.

→ Variable declared with Var can be re-declared with same name with no error

→ Variable declared with Var is Hoisted on top of declaration ex console.log(a) // undefined
var a = 10;

→ Let has block Scope, It is Hoisted but it goes in Temporal Dead zone (TDZ), It can be re-declared in diffrent scope, It can be declared without any value

→ Const has also block scope, it is Hoisted but it goes temporal Dead zone until initialization, with const you cannot update it and you have to assign a value to it at the time of declaration

* Difference between Null, Undefined & Undeclared

→ Null is special value that represent as empty or no value like → const num = null

→ Undefined is occured when you try to access the variable which is declared but not assigned any value Or a function does't return anything explicitly

→ Undeclared → This refers to a variable that has been referenced but not decleared or defined. If you try to access a undeclared variable, Javascript will throw referenceError.

* Curring

Curring is a technique used to transform a function with multiple argument into a nested series of function, each taking a single argument

Ex

```
function sum(a) {
```

```
    return function(b){
        return function(c){
            return a+b+c
        }
    }
}
```

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

Infinit currying code

```
function infiniteCurrying(x){
    return function(y)
        if(!y){
            return x
        } else{
            return infiniteCurrying(x+y)
        }
    }
}
```

```
console.log(infiniteCurrying(1)(2)(3)(5)(10)()) // 21
```

---

* Callback

A function passed to another function as an

argument is called Callback function

Eg

```
function Sum(a, b, callback) {
    return callback(a+b);
}

function displaySum(result){
    console.log("The sum is" + result)//15
}

Sum(5, 10, displaySum)
```

---

* Promise

Promise is used to handle asynchronous operation in Javascript. It is used to find out if the asynchronous operation is completed or not

Promise have three States

- Pending
- fulfilled
- rejected

Promise Starts with pending State. If the process is completed it goes in fulfilled state And if any error occurs It goes in rejected State

ex

```
Const count = true

Const myPromise= new Promise((resolve, reject)=>{
    if(count)
      resolve("There is count value")
    }else{
      reject("There is no count value")
    }
});

myPromise.then(function(data){
    console.log(data)
})

myPromise.catch(function(data){
    Console.log(data)
```

Output → Promise {'There is count value'}

---

* Promise.All Polyfill

Promise.all takes an array of promise as an input and return a single Promise that resolve to an array of the input promise

```
Promise.myAll = function (promises){
    let results = [];
    let counter = 0;
    return new Promise((resolve, reject)=>{
      promises.forEach((promise, idx) => {
      promise.then((val)=>{
            counter++;
            result[idx] = val
        if(counter === results.length){
          return resolve(result);
        }
      }).catch((error)=>{
          return reject(error);
      })
    })
}
```

---

* Async/await

Async/await is a feature of javascript that allow you to work on asynchronous code in more synchronous manner, making it easier to write and understand the asynchronous code

Async function always return a promise
Await keyword is always used only in Async function to wait for promise

Ex

```
Const count = true

Const myPromise= new Promise((resolve, reject)=>{
    if(count)
      resolve("There is count value")
    }else{
      reject("There is no count value")
    }
});

async function run(){
  try{
    const data = await myPromise
    console.log(data)
  }catch(error){
```

```
} catch (error) {
    console.log(error);
}
}

run()
```

---

* How JavaScript handled asynchronous task

Javascript is a Single threaded language
It handle one task at a time

How to get jobs in psypo

DSA important

lexical scope