**asJavascript Quick Questions**

**Basics of the Javscript**

1. **What are different types of variables in javascript?**

* We have three variables in the javascript -> **var,let,const**

1. **What is the difference between the var , let and const ?**

* **Var🡪** This is the old version of variables declaration . var promotes
  + Var has **hoisting**
  + The var is the function scope variable
  + The var value if declared on the global scope will store the variable in the global context window

`🡪 **Let and const 🡪** These are the newer version of the variables which are declared in the ECMA 6 guidelines

* + Let and Const doesn’t have hoisting
  + These are the block scoped variables
  + Const is used to declare the constant variables
  + The Let and const has the temporal dead zone

1. **What is the temporal dead zone ?**

* **Temporal dead zone** -> Temporal dead zone is the region where the let and const variables are inaccessible. Before declaration we can’t access the variables unlike the var type of variables

See the below example to see the temporal dead zone

**{**

// TDZ starts here (at the beginning of this block’s local scope)

// bestFood’s TDZ continues here

// bestFood’s TDZ continues here

// bestFood’s TDZ continues here

// bestFood’s TDZ continues here

// bestFood’s TDZ continues here

// bestFood’s TDZ continues here

**let bestFood = "Vegetable Fried Rice"; // bestFood’s TDZ ends here**

**console.log(bestFood); // returns "Vegetable Fried Rice" because bestFood’s TDZ does not exist here**

// bestFood’s TDZ does not exist here

// bestFood’s TDZ does not exist here

**}**

1. **What are the difference between the == and === (strict operator)**?

* == is type of the comparison operator in javascript which will implicitly type cast comparison of two variables
* === infers to strict variable comparison which the two comparison variables are not type casted.

1. **What are the null and undefined variables in javascript?**

* The undefined variables – are data type which the javascript compiler automatically provides to the variables which are not being assigned.
* The null variables – are data type which the user sets for the values to keep empty placeholder for the variables

1. **What are lambda function in javascript ?**

* The lambda or anonymous functions are the short hand version of describing the functions where only the arguments and logic of the functions needs to be defined . In some cases the we don’t need to also define the return type as it can be inferred if the logic is of single line only .

**Example of the lambda function:-**

**(args)=> logic or (args)=>{**

**//logic**

**}**

1. **What are the function scoped and block scoped variables?**

* **Function scoped ->** The function scoped variable are those types of variable which are scoped which are available is throught the whole function body . **var is the function scoped variable**
* **Block Scoped 🡪** The block scoped variables concept is introduced in the ECMA6 where the **let and const variables which are only scoped within the block like if , else or for blocks**

See the below example for the block scoped and function scoped variables

// Function scoped Variable Definition

function abc(){

    var a = 10

    function def(){

        console.log(a) *// a=10 will be printed function scoped only*

    }

}

console.log(a) *// a is not defined*

*// Block Scoped Variable*

*if(10>8){*

*const d=20 // only available in the block scoped*

*}*

*Console.log(d)*

1. **What are the type of functions are present in the javascript?**

* There are the three syntax to define the functions in the javascript
  + **Primitive function declaration with function keyword and name of the function**:-
    - function abc(a,b){

**return a+b;**

}

* + **Anonymous function declaration with function keyword only and no name generally used for iife**
    - function (){

**return a+b;**

}

* + **Lambda function or arrow function where only arguments and return logic should be used**

(args)=> logic or (args)=>{

//logic

}

1. **What is destructuring in javascript object and array ?**

* The destructuring is the concept or syntax where we can access the elements of the object and array without the looping mechanism we can directly access the elements with the following syntax
* *// Destructing for arrays*
* const arr = [1,2,3,4,5]
* const [a,b]=arr *// a=1, b=2*
* *// Destructuring in objects*
* const obj = {name:'John',age:25,city:'New York'}
* const {name,age} = obj *// name='John', age=25 in objects we can destructure only through keys names only.*

1. **What are the spread and rest operators in javascript?**

* The spread and the rest operators in the javascript are used to merge and converge the arrays and object into single array or object . or can make another object or arrays also
* One Layman language difference between the spread and rest operators are both the operators are quite similar to each other , the difference could be found it spread operator are generally used on the left hand side of the assignment and the rest is generally used on the right hand side of the assignment.

1. **Show the use of the spread operator on the arrays with concatenation of arrays using spread operator?**

🡪

*//  Spread operator in arrays*

const arr1=[1,2,3]

console.log(...arr1) *// 1 2 3 spreading the array elements*

const arr2 = [4,5,6]

const arr3 = [...arr2] *// 4 5 6 copying the array elements to another array*

const arr4 = [...arr1,...arr2] *// 1 2 3 4 5 6 combining/concatenating the arrays*

1. **Show the use of the spread operators on the objects with merging of two objects?**

🡪

*// spread operators in objects*

const obj1 = {name:"Rishabh",age:24}

const obj2 = {...obj1} *// copying one object into another using the spread operators*

const obj3 = {school:'JMS',college:'UPES'}

const obj4 = {...obj1,...obj3} *// merging two object into one new object using the spread operators*

1. **Different Types of for Loops we can used in javascript?**

* In the javascript broadly there are four types of for Loops which are
  + **Basic for Loop :** for(let i=0 ; i<10;i++){}
  + For of Loop in Array :- for (let item of items){}
  + For in loop of Object:- for(let item in items){}
  + ForEach array prototype function.:Array.forEach()

1. **What is the closure in javascript ? (One of the Most Important)**

* Closure is the property in the javascript where the function can access the variables and arguments of it parents function even when the function execution of the parent function is completed.
* This is due to the part where the returning function internally gets attached object in **[[Scopes]] with the name of the closure and it contains access to the variables and arguments.**
* *// closures*
* function abc(){
* const a=10
* return function(){
* console.log(a)
* }
* }
* const callBackFunc=abc()
* callBackFunc(); *// 10*

1. **What are firstClassFunctions , higherOrderFunction and IIFE Functions?**

* **FirstClassFunctions->**The first Class Function is considered to simple functions in the paradigm where the functions are considered to be variables .which can be stored in the variables , pass as arguments to another functions and also returned from the functions
* **HigherOrderFunctions🡪** The Higher Order functions are those types of functions which either takes another functions as argument or returns another function as an argument.

**Syntax ->** function abc(func){

**return func**

}

* **IIFE Functions 🡪** The iife functions are other functions which are declared and invoked immediately and simulatenoulsy . The IIFE functions are generally used in the cdn js library which needs to inject the functions and functionality as soon as it is injected.

**Syntax 🡪 (**function (){

**// any logic**

Console.log(‘abc’)

)()

1. **What is short circuiting in javascript ?**

* Short circuiting in javascript . The short circuiting is the methodology in javascript where results of the || and && operators
  + See the Below Examples
    - For || operator:- It Returns first true and last False
    - For && operator : It Returns first False and last True

1. **What are the datatypes in javascript ?**

**🡪**Number , Boolean , string, Object , null ,undefined, BigInt,Symbol

1. **What is the null coalescing Operator ?**

**🡪** Null coalescing operator (??) is upgradation of the (||) operator where the only **null and undefined values** will be considered as the falsy values else all the values are considered to truthy values

**Arrays Questions**

1. **What is the data type of the array function?**

* The array is the type of object
  + typeof(Array)==object

1. **What function helps to check whether its array or not ?**

* The Array.isArray(arr) function is used to check whether the passed object is array or not.

1. **What is the difference between the splice and slice operator?**

* Both splice and slice operators are used to modify the elements from the existing array but they have difference in both of them
  + **Slice :-** the Slice returns the new array with the elements from existing array from start index to endindex-1
    - **It does’nt modify the existing array**
    - **It returns the new array with elements which has been sliced**

Syntax :- newArray=Array.slice(startIndex,index)

* + **Splice :-**  the splice is the function which modifies the existing array and deletes the elements from the start index to the count we provide , its also takes the third arguments which can add the new elements in the place from the start index.
    - **Its Does modify the original array**
    - **It takes two mandatory arguments rest it not mandatory , non mandatory arguments are those arguments which will products which need to be added**
* *// Difference between the slice and splice operators*
* const sliceArr =[1,2,3,4,5,6]
* const slicedArr=sliceArr.slice(1,3)*// returns new Array [2,3]*
* console.log(sliceArr) *// same Array [1,2,3,4,5,6]*
* *// splice*
* const spliceArr=[1,2,3,4,5,6]
* const deletedElemt=spliceArr.splice(1,4) *// return the deleted Element*
* console.log(spliceArr) *// [1,6]*

1. **What are the functions which will insert the elements into the array ?**

* **Push –** this function will insert the element at the last of the array
* **Unshift –** This function will insert the elements into the first place of the array

1. **What are the function which will remove the elements from the array ?**

* **Pop –** This function will remove the last element of the array
* **Shift –** This function will remove the first element of the array

1. **What are some searchFunction in arrays ?**

* **indexOf(element)-**This function will return the **first index of the instance** of the element present in the array . **If the no element is found then (-1) will be returned**
* **includes(element)-** This function will return boolean value if the element is present or not present
* **lastIndexOf(element)-** This function will return the last index of the element present in the array
* **firstIndexof(element) –** This function will return the first Index of the element present in the array .

1. **What are the string related method in the array?**

* The string related method in arrays are:
  + **Join :** This method joins the array of string to a string using the separator
* const stringArr=['a','b','c']
* const joinedStr = stringArr.join('.')
* console.log(joinedStr)*// a.b.c*
* **toString():** This method will convert the array to string
* *// TO string*
* const stringArr=['a','b','c']
* const joinedStr = stringArr.toString()
* console.log('joined String',joinedStr) *// Joins the element with comma seperator*

1. **What is the difference Between the map and forEach method?**

* The Difference between the map and forEach Method is
  + **map returns the new Array after iterating and calling a callback function**
  + **forEach Method does’nt Return anything after iterating and calling a call-back function**

**This(Most Most Important)**

1. **What is the this keyword in the javascript?**

* The this keyword is the most hot topic and most important concept in javascript . The this keyword in javascript works differently as compare to other programming languages .

The this keyword **refers to the context where the piece of code is in lexical scope .**

The value of this keyword differs from function to function lets understand the each concept of this keyword in various places .

* + **Value of this keyword in global statement function when use-strict is not their 🡪** In this scenario the value of this inside the function refers to **global window object** in case of browser

function def(){

    console.log(this) *// window object*

}

* + **Value of this keyword in global statement function when use Strict is their -🡪** In this scenario the value of this keyword in the **global statement function will be undefined**

‘use strict’

function def(){

    console.log(this) *// undefined*

}

* + **(Most Most Important)Value of this keyword for the anonymous function inside the Object ( Most Important):🡪** when we use anonymous function inside the object as store it as the object key **, the value of this keyword will be the object itself .References to the object**

*// This keyword of the anonymous function inside the object*

const thisObj={

    name:"Rishabh",

    age:"20",

    anon:function(){

        console.log(this) *// {name:"Rishabh",age:"20",anon:func}*

    }

}

thisObj.anon()

1. **(Most Most Important) This keyword and Arrow Function Relations?**

* Arrow functions doesn’t have their own this keyword . **They inherit the value of this keyword of parent environment where there are lexically scoped .**

Lets see the below example to understand it better and how it works in different environments

* + **Arrow function this keyword inside the object method**
    - When this keyword is used as the property of the object . Then the this keyword will reference to parent environment . for global object it will print the window object

*// This keyword of the anonymous function inside the object*

const thisObj={

    name:"Rishabh",

    age:"20",

    anon:()=>{

        console.log(this) *// window object will be printed*

    }

}

thisObj.anon()

* + **Arrow function this keyword inside the anonymous function inside the object** 
    - When arrow function resides inside the anonymous function which is the property object , **then it will reference to the parent environment this which is the this of the anonymous function . Hence the object will be referenced**
* *// This keyword of the anonymous function inside the object*
* const thisObj={
* name:"Rishabh",
* age:"20",
* anon:function(){
* const def=()=>{
* console.log(this) *// {name:"Rishabh",age:"20",anon:func}*
* }
* def();
* }
* }
* thisObj.anon()

**Internals of Javascript :- Call Stack , Execution Context , MicroTask Queue,Macrotask Queue**

1. **What is the main pillars of the javascript internals ?**

**🡪**The three main pillars of the javascript internals are

* + **Global and Local Memory:** It’s the region where the variables memory and function bodies are stored which are referenced during the execution phase
  + **Call Stack :** The Javascript has only single call stack which follows FILO process of stack and each function enters the call stack and after execution leaves the call stack
  + **Execution Context :**For each and every function call new execution context is created and each execution scopes its local memory , this keyword and reference to outer variables
  + **MacroTask Queuue and Microtask Queue:** The macro and micro task queue holds the async functions execution in queue format first in first out and micro task queue has more priority than microtask queue , Generally the callback functions are stored in microtask queue

1. **What is the event loop in the javascript ?**

* The event loop in the javascript is the constant check process which is being runned for checking and execution of the micro and microtask queue functions if the call stack is empty. So keep on checking whether the call stack is empty or not . Constant loop

1. **Where Did the SetTimeout function is being thrown when executed?**

* The Set Timeout function is thrown in the macro task queue.

**Asynchronous Javascript:Promises,Async- Await , Promise Chaining**

1. **What are the promises in javascript ?**

* The promises in javascript is an object which kindoff handles delayed functions which has three states either **Pending , Fullfilled or Rejected ,** For each state completion we can assign a callback function which will be automatically called when the particular state is reached
* The Promise is object which handles the enventual completion or failure of asynchronous operations
* Promise is immutable object , and guarantees with three state pending , fulfilled and rejected

1. **(Most Most Important) What are the properties of promises?**

* A promise object has important properties assigned to it
  + **[[PromiseState]]-** This property holds the value of the promise current state , it can have three values **Pending , FulFilled and Rejected**
  + **[[PromiseResult]]-**This property holds the data or result which is emitted when the promise is fulfilled or rejected
  + **[[PromiseFullFillmentReactions]]-**This property holds the callback function which will be executed automatically when the promise is fulfilled and the promise result will be automatically passed as the first argument
  + **[[PromiseRejectionReactions]]-**This property holds the callback function which will be executed automatically when the promise is rejected and the promise rejection result will be automatically passed as the first argument.

1. **(Most Most Important) Syntax to declare the promise using the constructor function?**

*// Promises*

const promise = new Promise((resolve,reject)=>{

    resolve(2)

}).then((data)=>{console.log(data)}).catch((rejection)=>console.log(rejection))

Above syntax is to show to declare the promise object using the constructor function with new keyword

* (Most Most Important)Some Insights of the above syntax first one is **Promise Object will take one callback function which has two argument resolve and reject and both are function itself these function will set the PromiseResults and If the Resolved is called the PromiseState will set to fulfilled and reject is Called then PromiseState will set to rejected**
* Then then and catch call back function is being called based on the promise state.
* The **then ,catch and finally are present in the** Promise object Prototype

34 ) **What is Promise Chaining ?**

* To be learnt

**Async Await**

1. **What are async functions in javascript and what are the return type of the async function**

* The async in javascript is special keyword which when used with function wraps the function return type to a promise .
  + **The async function always returns a promise if returning value is not a promise it wraps it into the promise**

1. **Show me the syntax to declare the async function?**

**🡪**Below is the syntax to declare the async function

*// async function*

async function abc(){

    return 2

}

const asyncPromise = abc() *// the async function will return a promise object*

*//  To access the Results we need to call a thenable function on it*

asyncPromise.then((data)=>console.log(data)) *// 2*

1. **Using the await with the async function (Most Important Concept)?**

* As we know that Promises are non blocking in nature . The both **async and await** is used to handle promises and kinda make them synchronous in nature.
  + The await keyword can only be used with async function
  + (**Most Important Part the await keyword resolves the promise and return the resolved value see the example below)**
* *// some promise*
* const p = new Promise((resolve,reject)=>{
* setTimeout(()=>{
* resolve(2)
* },10000)
* })
* async function def(){
* const data = await p *// output 2 , here the data value will be resolved value which is 2*
* }
* def()

1. **If the await keyword returns resolved value how to handle error when using the await ?(Most Important)**

* Yes , it is true the await keyword used with promise will always return the resolved value .

We will use the **try and catch block to handle the** reject state , because internally when the promise is rejected , it throws a Error ( with message) and as function is wrapped with try and catch will be handled in catch block

See the below Example

const p2 = new Promise((resolve,reject)=>{

    reject('Some Error Occurred')

})

async function ghi(){

    try {

        const data = await p2

    }

    catch(error){

        console.log(error) *// The reject value will be printed here*

    }

}

ghi();

1. **How Fetch Works with async and await ?**

* To understand the syntax each every feature of the fetch see the below syntax with the explanation
* *// How Fetch Works*
* async function jkl(){
* const fetchResponse = await fetch('API') *// fetch resolves into a promise so fetchResponse will be promise*
* const data = await fetchResponse.json() *// now fetchResponse is readable stream to convert we need json function will also a promise and await resolves a promise*
* }

This is how the fetch Works will async await

1. **Do the await holds the main thread of execution as the function is stopped ?**

* No , The await don’t holds the main thread of execution
  + Instead when the await hits the execution phase , the javascript engine suspends the function from removing it into the call stack and after the promise is resolved or rejected then the function re-enters the call stack and then the execution start from next line of await keyword**(!!Most Most Important concept)**

**Callback functions**

1. **What are the higher Order functions ?**

* The higher order functions are those functions which either take another function as the argument or return the another function.

The use of the higher function is to create closure mechanisms.

1. **What are the call back functions?**

* The callback function is the function which is being passed as the arguments
  + In my layman language the callback function is that type of whose derive and pass , but the calling of that function is dependent upon another abstract function.

1. **What are the some examples and uses of the callback function ?**

* The best example and uses of the call back functions are :-
  + Event Listeners – in event listener we attach a function and when the event is hit your function will be automatically called
  + Promise Callback Function – In promise we can attach a callback function in thenable and catch functions.

A callback function helps a to open the doors of the async programming where we can define the function at a declaration and calling of these functions will based on async completion

1. **Can you show me some example of the callback functions?**
2. *// callback function*
3. const abc = (resultFunc)=>{
4. resultFunc(2)
5. }
6. abc((data)=>console.log(data)) *// 2*

**Prototypal Inheritance in Javascript**

44) **What is the prototypal inheritance in javascript ?**

**🡪** In javascript every thing is a object , because every datatype is Originated from the parent Called **Object which is actually referred to as root object .**

And Every datatype has access to Root object properties + other their own properties for the data type

As we have seen that if we define an array we can access to many inbuilt properties like map , forEach , Reduce . This is because the Array.Prototype is object which has all those properties assigned to it .

Similarly any object will automatically inherit the properties which are defined in Object.Prototype

Similarly any Function will automatically inherit the properties which are defined in Function.Prototype

45) **how to Assign a custom function which is access to all the arr object which we will declare further?**

**🡪** *// Prototype function*

Array.prototype.myFunc= customFunction

function customFunction(){

    console.log('This is my function')

}

const abcArr=[1,2,3]

abcArr.myFunc() *// This is my function*

**46) What is Prototypal Chaining ?**

**🡪**  Prototypal chaining is the mechanism where the a particular object can access its **prototype properties + its parent properties.**

**47) How javascript checks for unknown properties on any object ?**

**🡪** Array.unknownProperties🡪First Checks for Property on **Array.Prototype🡪**Then Checks for Property on **Object.Prototype 🡪**Even not found then **Throws the Unknown Property Error**

**Call , Apply And Bind Concept (Most Most Important )**

Call ,apply and Bind are the properties method which are attached to every function object we create or declare and its Present on **Function.ProtoType object**

48) **What is call and apply function methods in javascript?**

**🡪 All the three call , apply and bind are used to configure the reference to this object for which a function is attached manually through our declaration.**

* Both call and apply method will immediately invoke the function , with reference of this object of the object we pass into the call and apply method
* The call and apply method functionality is almost same just the difference is into the second argument which shows how we pass the arguments of the on which call and apply is called.

Below Is the syntax for the call and bind method

*// call and apply functions*

const custom = function(a,b){

    console.log('Arguments are',a,b,this.name)

}

const obj10={name:"Rishabh"}

const obj20={name:"Amit"}

custom.call(obj10,2,3) *// Arguments are 2 3 Rishabh in call we pass the arguments with comma seperated*

custom.apply(obj20,[2,3])*//In apply we pass the arguments in the array format*

1. **What is the bind function methods in javascript?**

* The bind function also helps to configure the this object of the function in the javascript .
  + The difference of the bind function from call and apply is the bind function **will return the new function binded with the this keyword of the object we have passed**
  + The bind function will take the one argument which is the obj with which we need to bind our function.

Below is the syntax for the bind function in javascript

*// bind function*

const customBind = custom.bind(obj10) *// bind will return a function with the context of the object*

customBind(2,3)

**Most Trivial Javascript Coding Questions(All the concepts are Very Very Important)**

1. **Write the Custom Polyfill of the map function ?**

**🡪**/\*\*Create A Custom Map Polyfill \*/

Array.prototype.customMap = customMap

function customMap(cb) {

    const updatedArr = []

    for (let i = 0; i < this.length; i++) {

        updatedArr.push(cb(this[i], i))

    }

    return updatedArr

}

const someArr = [1, 2, 3, 4, 5]

const customizedArr = someArr.customMap((data) => data \* 2)

console.log(customizedArr)

1. **Write the custom Polyfill function of the Array Filter Function?**

**🡪**/\*\*Create A Custom filter Polyfill function \*/

Array.prototype.customFilter = customFilter

function customFilter(cb) {

    const filterdArr = [];

    for (let i = 0; i < this.length; i++) {

        const cbResult = cb(this[i], i)

        if (cbResult) {

            filterdArr.push(this[i])

        }

    }

    return filterdArr

}

const someArr2 = [1, 2, 3, 4, 5, 6]

const filteredArray = someArr2.customFilter((elem) => elem % 2 === 0)

console.log(filteredArray)

1. **Write a custom Polyfill function of the Array Reduce function?**

**🡪**/\*\*Create A Custom Reduce Polyfill function \*/

Array.prototype.customReduce = function(cb, initialValue) {

    //  Initial Value Declaration

    let reducedValue = initialValue

    for (let i = 0; i < this.length; i++) {

        reducedValue = reducedValue ? cb(reducedValue, this[i]) : this[i]

    }

    return reducedValue

}

const someArr3 = [1, 2, 3]

const finalReducedValue = someArr3.customReduce((acc, elem) => acc + elem)

const finalReducedValueWithInitialValue = someArr3.customReduce((acc, elem) => acc + elem, 0)

console.log(finalReducedValue)

1. **Write a custom call() function Polyfill ?**
2. /\*\*Create a custom Call Method Polyfill \*/
3. Function.prototype.customCall = function(context = {}, ...args) {
4. // To check whether the calling is function or not
5. if (typeof this !== 'function') {
6. throw new Error('It is not callable')
7. }
8. // as context is object attaching the this ie function to another property function
9. context.fn = this
10. // Then calling the function on the context so object this will be taken
11. context.fn(...args)
12. }
13. const obj1 = {
14. name: "Rishabh",
15. age: 24
16. }
17. const obj2 = {
18. name: "Amit",
19. age: 26
20. }
21. function abc() {
22. console.log(`${this.name}---${this.age}`)
23. }
24. abc.customCall(obj2)

56) **Write the Polyfill for the apply method?**

/\*\*Create a custom Apply Method Polyfill \*/

Function.prototype.customApply = function(context = {}, args = []) {

    if (typeof this !== 'function') {

        throw new Error('It is not callable')

    }

    context.fn = this

    context.fn(...args)

}

const obj3 = {

    name: "Rishabh",

    age: 24

}

const obj4 = {

    name: "Amit",

    age: 26

}

function abc() {

    console.log(`${this.name}---${this.age}`)

}

abc.customApply(obj3)

**57) Write the Custom Polyfill for bind Method?**

Function.prototype.customBind = function(context = {}, ...args) {

    if (typeof this !== 'function') {

        throw new Error('It is not callable')

    }

    context.fn = this

    // Here we have to return another function which will call the function on the context object so this of the context will be considered

    return function(...newArgs) {

        return context.fn(...args, ...newArgs)

    }

    /\*\*

     \* Important

     \* Instead if we return the function directly

     \* like this  and then call then it will return undefined because the function will be called on the global environment

     \* return context.fn

     \*/

}

const obj5 = {

    name: "Rishabh",

    age: 24

}

const obj6 = {

    name: "Amit",

    age: 26

}

function abc() {

    console.log(`${this.name}---${this.age}`)

}

bindedFunc = abc.customBind(obj5)

bindedFunc()

**58) Write a function which will run only once?(Closure Concept)**

**🡪**// Create a custom only Time Calling function

function callOnlyOnce(fn) {

    let callCount = 0

    return function(...args) {

        if (callCount < 1) {

            ranResult = fn(...args)

            callCount++;

            return ranResult

        }

    }

}

const onceWrappedFunc = callOnlyOnce((a, b) => console.log('a+b'))

onceWrappedFunc(1, 2)

onceWrappedFunc(1, 2)

onceWrappedFunc(1, 2)

onceWrappedFunc(1, 2)

**59) Create Memoize function which will store the cache results?**

--> // Write a HOC Memo Function

function memoizedFunction(fn) {

    const cacheArgs = {}

    return function(...args) {

        const funcArgs = JSON.stringify(args)

        if (!cacheArgs[funcArgs]) {

            const runResult = fn(...args)

            cacheArgs[funcArgs] = runResult

            return runResult

        } else {

            return cacheArgs[funcArgs]

        }

    }

}

const memoMultiply = memoizedFunction(function multiply(a, b) {

    console.log('Function Runned result is', a \* b)

    return a \* b

})

console.log("Runned", memoMultiply(123, 123))

console.log("Memoed", memoMultiply(123, 123))

56) **Write a Debounce Function in Javascript?(Most Important) (Most Asked Js question)**

🡪 // Write a debounce function in javascript

function debounce(fn,delay) {

    let timerId=null;

    return function(...args){

if(timerId) clearTimeout(timerId)

        timerId=setTimeout(()=>fn(...args),delay)

    }

}

function abc(){

console.log("Hello i am Delayed")

}

const debouncedFunc=debounce(abc,5000)

debouncedFunc()

**57) Write the Custom Throttle function? (Most Asked javascript question)**

// simple way

function simpleThrottleWrapper(fn, delay) {

    let last = 0

    return (...args) => {

        const now = new Date().getTime()

        if (now - last < d) return

        last = now

        fn(...args)

    }

}

function throttleWrapper(fn,delay){

  let shouldAwait = false

  let remaingArgs;

  return (...args)=>{

    if (shouldAwait){

        remaingArgs=args

        return

    }

    fn(...args)

    shouldAwait=true

    setTimeout(()=>{

        if(remaingArgs===null){

            shouldAwait=false

        }else{

            fn(...remaingArgs)

            remaingArgs=null

            shouldAwait=false

        }

    },delay)

  }

}

**58) Define a Promise which will resolve only when the button is clicked**

// Write a Promise which will resolve only when the button is clicked

const resolveBtn = document.querySelector('.resolve')

const p = new Promise((resolve, reject) => {

    resolveBtn.addEventListener("click", (event) => {

        resolve('resolve')

    })

})

p.then((data) => console.log(data))

p.catch((error) => console.log(error))

**59) Create a Promise without the promise Constructor?**

// create a Promise without the promise constructor

async function abc() {

    return p

}

// async function always returns the Promise value

console.log(abc().then((data) => console.log("Resolllled", data)))

**60 ) What is Function Currying and Solve the infinite sum function problem Sum(10)(8)(7) – Three Level currying function**

* *// In this file we will be solving the three level currying function which is sum(10)(20)(30)*
* function sum(a){
* return function (b){
* return function(c){
* return a+b+c
* }
* }
* }
* console.log(sum(10)(20)(30));

**61) Provide us the solution for the infinite currying which is Sum(10)(12)(13)(14)(5)……(n) ? (Most Asked Javascript question Function Currying)**

**-** *//  Now we have to implement the infinite currying problem which sum(10)(20)(30).....(n)*

function sum(a){

    return function (b){

        if (b) return sum(a+b)

        return a

    }

}

console.log(sum(10)(20)(30)(40)(50)(60)(70)())

**62) What is difference between Slice and Splice in javascript? (Most frequent and Most Important Question)**

*/\*\**

*\* This is generally the question which is asked by the interviewer what is difference between the splice and slice*

*\**

*\* Both slice and splice is used to get the sub part of the array but the*

*\* The main difference between slice and splice is*

*\* - Splice - Modifies the original array on which it is being used , it returns the spliced Array  and also modifies the original array .*

*\* - Slice - Creates and Returns the new Array and doesnt modify the actual array*

*\**

*\* Lets see the example one by one*

*\*/*

*// Slice example*

const someArr=[1,2,3,4,5,6]

const slicedArray=someArr.slice(0,3)

console.log(slicedArray) *// sliced array*

console.log(someArr) *// Original one remains the same*

*// Splice Example*

const someArr2=[1,2,3,4,5,6]

const spliceArr=someArr2.splice(0,3)

console.log(spliceArr)

console.log(someArr2)

***Advance Javascript Topics***

1. **What is Intersection Observer?**

**🡪**The Intersection is the Browser API which keep observing the particular element intersection with ancestor component or the document viewport

and as soon as that element enters the viewport , the observe changes its properties **isIntersecting to True and we can do some Function from it.**

*// creating the Intersection obbserver*

let observer = new IntersectionObserver((entries)=>{

    // logic

}, options);

// setting which element to observe

observer.observe(domelement)

Lets see the above syntax :-

* To declare the Intersection Observer we need to call a New Constructor , the new Constructor takes two arguments
  + **Callback function –** this is important callback function . the callback function will be always called when the observing element is entered or leave the container.
    - The callback function always takes one argument which is **entries**

the entries is the array for the observer elements

* + - * Each entry object has some important properties
        + **isIntersecting-** This property will tell whether the element is intersecting or not
        + **boundingClientRect**:- Provides the dimension of the observing element boundaries
        + **intersectingClientrect:** Provides the dimension based on how much the element is inside the element boundaries
        + **Other Propeties are also their**
  + **Options-** To alter the intersection observer behaviour
    - **Threshold-** This tells when to set isIntersecting value to true , based on the percentage of element is in the screen . **0 means even the entry isIntersecting even 1 pixel into the container.**

**1 means the 100% element should be visible on screen then only isIntersecting is set to true**

* **Observe-** Method takes the element which needed to be observed.
* *// React Example for the use Effect*
* useEffect(()=>{
* if(intersection.current===null) return
* *// creating the new intersection observer*
* const observer = new IntersectionObserver(([entry])=>{
* if(entry.isIntersecting){
* *// write execution logic here*
* }
* },{
* threshold:0
* })
* *// setting the element which needed to be observed*
* observer.observe(intersectionRef.current)
* },[intersectionRef.current])

1. **What is Promise.all api ?(Important Advance Javascript Question)**

**🡪**Promise.all is the method attached to the Promise Prototype which has the following properties

* + **Scenario-** Consider a Scenario where you want to make three parallel fetch calls and you should only get the result of all three calls if all the three calls are success , if any one of the fetch calls fails then return me a error .
  + **For the Above Scenario we use Promise.all method**
    - **Properties of Promise.all**
      * **Input(Array):**The Promise.all method takes **array of promises as the argument**
      * **Return Type:-**
        + **All Promise Success(Array):**If all the promise are successful then it will return the **Array where each element is the data return for each Resolved Promise**
        + **Error Scenario(Value):**If any of the promise gets rejected , then whole Promise.all gets rejected and the **Error is returned as the return type.**
* */\*\*Promise.all code example\*/*
* const p1 = new Promise((res,rej)=>{
* setTimeout(()=>{
* res("First Promise Resolved")
* },0)
* })
* const p2 = new Promise((res,rej)=>{
* setTimeout(()=>{
* res("First Promise Resolved")
* },1000)
* })
* const p3 = new Promise((res,rej)=>{
* setTimeout(()=>{
* res("First Promise Resolved")
* },2000)
* })
* console.time('Started')
* Promise.all([p1,p2,p3]).then((data)=>console.log(data)).then(()=>console.timeEnd('Started'))

1. **What is Promise.allSettled api ?(Important Advance Javascript Question)**

**🡪**Promise.allSettled is the method attached to Promise Prototype which has following Properties:

* **Scenario:**Consider the Scenario where you need to make three fetch api calls and you want the result of all the three fetch api calls whether they are rejected or they are successful
* **For the Above Scenario we use Promise.**allSettled api , as the name suggest all the api Should be settled.
  + - **Properties of Promise.allSettled**
      * **Input(Array):**The Promise.allSettled method takes **array of promises as the argument**
      * **Return Type:-**
        + **Whether the Promises are Successful or error(Array):**It returns the array of settled data for each promise which has been passed as an argument ,if the promise is successful it gives the data value and if its error it returns the error value for the promise as element of the array.

Below is the Syntax for Promise.allSettled

*/\*\*Promise.Settled code example\*/*

const p1 = new Promise((res,rej)=>{

    setTimeout(()=>{

        rej("Promise First Rejected")

    },0)

})

const p2 = new Promise((res,rej)=>{

    setTimeout(()=>{

        res("First Promise Resolved")

    },1000)

})

const p3 = new Promise((res,rej)=>{

    setTimeout(()=>{

        res("First Promise Resolved")

    },2000)

})

console.time('Started')

Promise.allSettled([p1,p2,p3]).then((data)=>console.log(data)).then(()=>console.timeEnd('Started')).catch((error)=>{

    console.log('Erroorrr',error)

    console.timeEnd('Started')

})

*/\*\**

*Promise.settled return value see how for each promise object is returned*

*(3) [{…}, {…}, {…}]*

*0*

*:*

*reason: "Promise First Rejected"*

*status: "rejected"*

*1*

*status: "fulfilled"*

*value: "First Promise Resolved"*

*2*

*status: "fulfilled"*

*value: "First Promise Resolved"*

*\*/*

1. **What is Promise.race api ?(Important Advance Javascript Question)**

**🡪**Promise.race is the method attached to Promise Prototype which has following properties:

* **Scenario:-** In the case you want the result of the first settled promise ,whether its successful or it gives error

**As the Na**me suggest race so it’s the race between the promises which ever will be first settled either successful or reject the value is returned.

* + - **Properties of Promise.race**
      * **Input(Array):**The Promise.race method takes **array of promises as the argument**
      * **Return Type:-**

**Whether the First Setteled Promise is Successful or error(Value):**It returns the value of the first settled promise if its successful then the data will be returned and If its failed then the it will return the error value .

* */\*\*Promise.race code example\*/*
* const p1 = new Promise((res,rej)=>{
* setTimeout(()=>{
* res("Promise First Resolved")
* },0)
* })
* const p2 = new Promise((res,rej)=>{
* setTimeout(()=>{
* res("First Promise Resolved")
* },1000)
* })
* const p3 = new Promise((res,rej)=>{
* setTimeout(()=>{
* res("First Promise Resolved")
* },2000)
* })
* console.time('Started')
* Promise.race([p1,p2,p3]).then((data)=>console.log(data)).then(()=>console.timeEnd("Started")).catch((error)=>{
* console.log("Erroor",error)
* console.timeEnd("Started")
* })
* */\*\*\**
* *Promise.race Return type*
* *Promise First Resolved*
* *\*/*

1. **What is Promise.any api ?(Important Advance Javascript Question)**

**🡪**Promise.any is the method attached to Promise Prototype which has the following properties

* **Scenario:**There is a case where you want the result of the first successfully settled promise ,Then we will use Promise.any

**As the** name suggests the promise.any means the any of the first Settled Promise will be the result.

* + - **Properties of Promise.any**
      * **Input(Array):** The Promise.any method takes **array of promises as the argument**
      * **Return Type:**
        + **Any Success(Value):­**It will return the first successfully settled promise resolved value
        + **If All the Promises get Failed(Array):**It will return the Aggreate Error Array where it will give the Error of each failed Promise as the element.

*/\*\*Promise.any code example\*/*

const p1 = new Promise((res,rej)=>{

    setTimeout(()=>{

        rej("Promise First Resolved")

    },0)

})

const p2 = new Promise((res,rej)=>{

    setTimeout(()=>{

        rej("First Promise Resolved")

    },1000)

})

const p3 = new Promise((res,rej)=>{

    setTimeout(()=>{

        rej("First Promise Resolved")

    },2000)

})

console.time('Started')

Promise.any([p1,p2,p3]).then((data)=>console.log(data)).then(()=>console.timeEnd("Started")).catch((error)=>{

    console.log("Erroor",error)

    console.timeEnd("Started")

})

*/\*\*\**

*Promise.any Return type*

*Promise First Resolved (For Successful)*

*If all the Promise are rejected a text will be printed*

*AggregateError: All promises were rejected*

*\*/*

**Performance Optimization Techniques**

**Bundle Splitting**

When you write your UI code that code is taken by **Bundlers like Webpack or Rollup.**Then it converts your code into a bundle and number of bundles depends on how you have written the code(and that’s the core optimizing technique 😊) and then sent it to the Browser.

So when the user comes to the page , this bundle is on JIT is uncompressed , then loaded and Executed. Although the modern Browser has evolved to quickly parse and compile the code .

**Its still developer responsibility to handle the execution time of the Bundle and release the main thread** .

So till here we can deduce that , If that if the Bundle size is very large it will result in larger load time and larger execution time which in inturn result in slower websites.

So the **Bundle Splitting is Concept where we write and export the code is such a way that rather than one big bundle we able to create multiple small bundles , which are loaded only when they are required😊**

**Compressing Javascript**

There is another factor which helps to optimize the web performance which is the compression of javascript. The javascript before going to the browser is being compressed into the minified versions

**Design Patterns in Javascripts**

Design patterns are important to make code **more modular , scalable and helps to maintain sanity between different modules and along different developers who are committing to the codebase**. In short design patterns helps to safeguard our code from turning into spaghetti codebase

There are different Design patterns in coding and they are language agnostic in nature. Lets learn about those design patterns in javascript

1. **Singleton Pattern**

*// Singleton Pattern - In this file we will learn all things about the singleton Design patterns*

*/\*\**

*\**

*\* Singleton Design Patterns - This Design pattern talks about  a class of which only a  single instance of a class can be instantiated and that instance would be referenced all through out the code .*

*\**

*\* Generally used for*

*\* - DB object*

*\* - Api Services*

*\* - Injectable services*

*\*/*

*// Below is the example of the singleton classt*

let instance;

let counter=0

class SingletonCounter{

    constructor(){

        if(instance){

            throw Error("Only one instance can be created")

        }

        instance=this

    }

    getinstance(){

        return this

    }

    increment(){

        counter++;

    }

    decrement(){

        counter--

    }

    getCounter(){

        return counter

    }

}

const counter1= Object.freeze(new SingletonCounter());

*// const counter2= new SingletonCounter();*

counter1.counter=20 *// cannot change the property as the object is freeezed*

counter1.increment()

counter1.increment()

counter1.increment()

counter1.increment()

counter1.increment()

console.log(counter1.getCounter())*// Two different singleton*

1. **Observer Pattern or Pub Sub Pattern**

*/\*\**

*\* Observable patterns- Its one of the most used pattern . Where we can  subscribe a observer to another object called observable and that observable can notify all its subscribed observer*

*\**

*\* The observable has 3 following important parts*

*\* -Observer - The list of the observers*

*\* - subscribe(obs)- this function will add new observer*

*\* - unsubscribe(obs) - this function will remove provided observer from the list observers*

*\* -notify(data)- this is function will invoke all the observer with the data provided*

*\*/*

class Observable{

    constructor(){

        this.observers=[]

    }

    subscribe(func){

        this.observers.push(func)

    }

    unsubscribe(func){

        this.observers=this.observers.filter((observer)=>observer!==func)

    }

    notify(data){

        this.observers.forEach((observer)=>observer(data))

    }

}

*// Below is the implementation of the observer where we subscribe two functions and then call them using notificatins*

function datelogger(data){

    console.log(`${new Date().getDate()}/${new Date().getMonth()}/${new Date().getFullYear()}`,data)

}

function timelogger(data){

    console.log(new Date().getTime(),data)

}

*// Creating new observable*

const observable = new Observable();

*// subscribing these above datalogger function*

observable.subscribe(datelogger);

observable.notify("Some Random Data")

*// subscribing timelogger function*

observable.subscribe(timelogger);

observable.notify("Some Another Random Data");

*// Unsubscribing the Date Logger function*

observable.unsubscribe(datelogger)

observable.notify("Some another another Random Data");

**3)Proxy Pattern –** As the name suggest this pattern is generally used to connect with a object via some other function , the best example of the proxy pattern is the getters and setters function . They provide a abstracted to connected with the actual object.

*/\*\**

*\**

*\* In this File we will be learning about the proxy design patterns .*

*\* Proxy design Pattern -*

*\* As the name suggest this pattern is generally used to connect with a object via some other function , the best example of the proxy pattern is the getters and setters function . They provide a abstracted layer to get connected with the actual object.*

*\**

*\* Generally speaking, a proxy means a stand-in for someone else. Instead of speaking to that person directly, you’ll speak to the proxy person who will represent the person you were trying to reach. The same happens in JavaScript: instead of interacting with the target object directly, we’ll interact with the Proxy object.*

*\**

*\**

*\* NOTE - JAVASCRIPT HAS INBUILT PROXY OBJECT WHICH HELPS US TO IMPLEMENT THE PROXY PATTERN*

*\**

*\* Proxy(object,handlers)*

*\**

*\* handlers are bunch of functions which are exposed to manipulate the actual object m the most common and most used are*

*\* get and set functions*

*\**

*\* - The Proxy patterns are generally used for the checking before inserting , error handling etc*

*\**

*\* Lets see the below example of Proxy*

*\*/*

const person = {

    name:'Rishabh',

    age:20,

    sex:'Male',

    uid:'1231293484940'

}

*// so we can create a Proxy function which blocks user to change the uid  and also should only return the property which is present in object and should check the age should be Int literal*

const personProxy=new Proxy(person,{

    get:(obj,prop)=>{

        if (Object.hasOwn(obj,prop)){

            return obj[prop]

        }else{

            throw Error(`The propery ${prop} is not present`)

        }

    },

    set:(obj,prop,value)=>{

        if (prop==='age'&& typeof(value)!=="number"){

            throw Error('The Property age should only be number')

        }else if (prop==='uid'){

            throw Error('Oops! you cannot set the uid property')

        }else{

            obj[prop]=value

        }

    }

})

console.log(personProxy.name);

personProxy.age=2323

console.log(personProxy.age);

personProxy.uid='123123123';