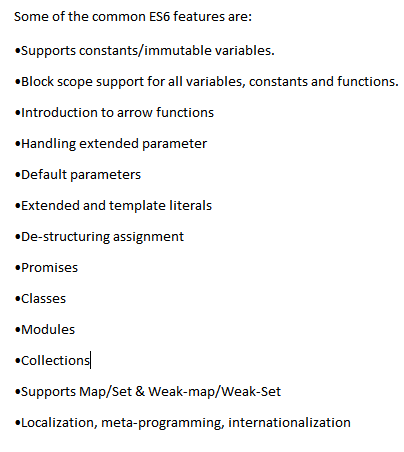
* JavaScript
* Feature of ES6



* Primitive and non primitive

|  |  |
| --- | --- |
| Primitive | Non-Primitive |
| Primitive values are the immutable [which cant be modified after creation.] | Non-Primitive values are the mutable [which can be modified after creation.] |
| Primitive are stored by value. | Non-Primitive are stored by reference. |
| Ex:: number, string, Boolean, null, undefined | Ex: object, array, function |

* Diff == and === operator

Both are compare the operator-

|  |  |
| --- | --- |
| == | === |
| compares value only. | compare value and type. |

* ForEach loop
* The forEach() method calls a function for each element in an array.
* The forEach() method is not executed for empty elements.

*array*.forEach(*function(currentValue, index, arr), thisValue*)

* How JavaScript works and how the code is executed?

Everything in JavaScript happens inside an “Execution context”.

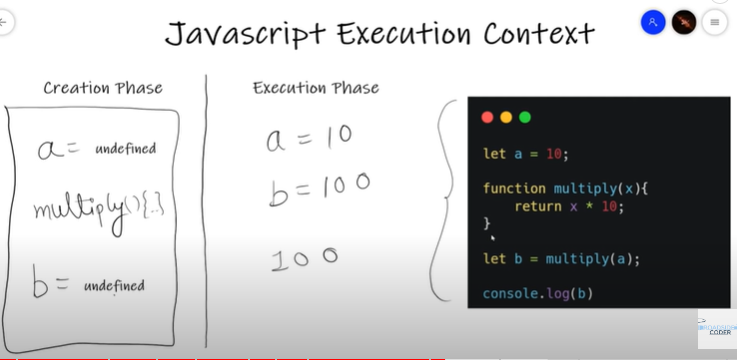
Execution context- its like a big box or we can say that the environment in which our code is expected and evaluated.

I has two components-

Memory component – All the variables, function and parameter are stored as a key value pair. Memory component also known as “variable environment”.

Code component- This is the place where code is executed one line at the time and assigning the values to the variable and execute the function calls. Also when function invoke JS engine created the new execution context. It is also known as the “Thread of execution.”

|  |  |
| --- | --- |
| Memory | Code |
| Key: value  a:10  fn: {  } |  |



* Javascript is single threaded or multi-threaded ?

“JavaScript is a synchronous single threaded language”.

JavaScript execute one line at the time in a specific order means that it can only go to the next line once the current line has been finished execution.

* What is call stack in JS?

Call stack maintains the order of execution of execution contexts.

It also known as –

Execution context stack, program stack, control stack, runtime stack, machine stack.

Its also follow LIFO (last in , first out) structure, which is used to store all the execution context created during the code execution.

By Default JS engin/browser created Global execution content but when we call the code/Whenever function is invoke/ Function is executed, New execution context is created.

* What is Hoisting?

Hoisting is a phenomenon is JS by which you can access variable and function even before you have initialized it. We can access it without any error.

During the creation phase JS engine moves the variable and function declarations to the top of the code this is know as the hoisting.

Ex- If I declare the variable and if I try to console.log before it was declared. We will get undefined. We will not get any error. How the JS execution context works. It will declare all of these variable and function at the top of the code during the creation phase then when the execution happen it checks if the variable already exists during the creation phase is or not so obviously its exists so it give us undefined.

Function Hoisting-

|  |  |
| --- | --- |
| sum(5,4);  function sum(a,b){  add = a+b;  console.log(add);  } | function sum(a,b){  add = a+b;  console.log(add);  }  sum(5,4); |

Variable Hoisting-

|  |  |
| --- | --- |
| console.log(a);  var a =10;  o/p- undefined | var a =10;  console.log(a);  o/p- 10 |

Hoisting is not working in Function expression.

|  |  |
| --- | --- |
| var sum = function(a,b){  add = a\*b;  console.log(add);  }  sum(4,4);  o/p-16 | sum(4,4);  var sum = function(a,b){  add = a\*b;  console.log(add);  } **o/p- Uncaught TypeError: sum is not a function** |

Hoisting is not working in Arrow Function.

|  |  |
| --- | --- |
| var sum = (a,b)=>{  add = a\*b;  console.log(add);  }  sum(3,5);  0/p-15 | sum(3,5);  var sum = (a,b)=>{  add = a\*b;  console.log(add);  }  **o/p- Uncaught TypeError: sum is not a function** |

Hoisting is not working in let and const.

|  |  |
| --- | --- |
| console.log(a);  let a =10;  o/p- **Uncaught ReferenceError: a is not defined** | console.log(a);  const a =10;  o/p- **Uncaught ReferenceError: a is not defined** |

* What is temporal dead zone?

Let and const is also hosted but It will hosted in the temporal dead zone.

Temporal dead zone(TDZ) is the time since when the ‘let and ‘const’ variable hoisted till initialize some value that time between that known as the Temporal dead zone(TDZ).

From hoisting till it assigned or it get some value/ Initialize some value that phase know as the temporal dead zone.

Example -

|  |  |
| --- | --- |
|  | |
|  | |
| **Are let and const declaration hoisted.?**  So, whenever you try to access a variable inside the temporal dead zone it gives you the ‘reference error’. They can only be access once some value initialize to them.  Best way to avoid temporal dead zone is always put declaration and initialization on the top of the scope. | |
|  |  |

“b” is attached to the global object but incase of ‘let and ‘const’ they are allocate memory but they are stored in different / separate memory space and you can not access memory space before you have put some value in them.

Temporal dead zone(TDZ) is the time between the deceleration and the initialization of let and const variable.

In other words we can say that it is the mechanism where variable and function declaration moved to the top of their scope before the code execution.

* Difference between var ,const ,let?

|  |  |  |
| --- | --- | --- |
| Var | Let | const |
| Variables define with the var can be redeclared and reassigned.  var x = “Hello”;  var x= “World”;  x=” wow”; | Variable define with the let can not be redeclared but it can be reassigned.  let x = “Hello”;  let x= “World”; (Can’t)  let=”wow”; | Variable define with the let can not be redeclared and reassigned.  const x = “Hello”;  const x= “World”; (Can’t)  x=”wow”; (Can’t) |
| Variable defined with var have global scope or the function scope.  Means if the var declared inside a {} block can be accessed from outside the block. | Variable defined with let have block scope.  Means if the var declared inside a {} block can not be accessed from outside the block. | Variable defined with let have block scope.  Means if the var declared inside a {} block can not be accessed from outside the block. |
| Mutable variable are the variables whose values is changed any number of the time. | Mutable variable are the variables whose values is changed any number of the time. | Immutable variable are the variables whose values is never changed in the complete life cycle of the program. |

* What is scope and How many types of scope in JS?

Scope(Scope is the certain region of the program where a defined variable exist and can be recognized and beyond that it can not be recognized. There can be multiple type of scopes-)

1. Global scope- without any block and any function its a global scope.
2. Block scope - { }
3. Functional scope - function (){ }

* What is block in JS?
* Block is defined by these curly braces.

{

}

* Block is also known a compound statements.
* Block is used to combine multiple JS statement into one group. So, that we can use it where JS expects one statement. OR  
  if (){

//compound statements  
 var a = 10;  
 console.log(a);  
 }

* Wrapping up of the multiple statement group together so that we can use it in a place where JS expects one statement.
* What is Block Scope in JS?
  + - Block scope means what all the variables and functions we can access inside that block that is know as block scope.

**Example**- {  
 var a = 10;

Let b = 20;

const c = 30;  
 }

Block  
 b: undefined

c: undefined

Global

a: undefined

* Block scope is a separate memory space or we can say ‘let’ and ‘const’ hosted in a separate memory space that reversed for this ‘block’.
* You can not access ‘let’ and ‘const’ outside the block that is know as ‘let’ and ‘const’ in ‘Block Scope’.
* Whereas you can access ‘var’ even outside the because ‘var’ is in the ‘Global scope’.
* In short let and const is the block scope (it can be accessible inside the block) and var is the functional scope.

**Exercise -**

**1.**

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

**2.**

{

    var a = 10;

    let b = 20;

    const c = 30;

    console.log(a);

    console.log(b);

    console.log(c);

}

console.log(a);

console.log(b);

console.log(c);

o/p - 10

20

30

10

Uncaught reference-error: b is not defined.

* Shadowing in JS ?

If you have a same named variable outside the block so that inner variable shadow the outside.

***Shadowing in Variable-***

**Example- In Case of ‘var’.**

|  |  |
| --- | --- |
|  | **Block**  b: undefined  c: undefined  **Global**  a:100 then 10  Inner var a = 10; is shadow and it is also modify the outer variable value.  But the both variable referring to the same memory space that is ‘Global Space’. |

**Example- In Case of ‘let’ and ‘const’**

|  |  |
| --- | --- |
|  | **Block**  b:20  c:30  **Script**  b:100  **Global**  a:10 |

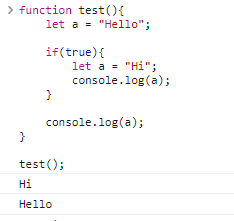
Same thing happen is ‘const’.

***Shadowing in Function-***

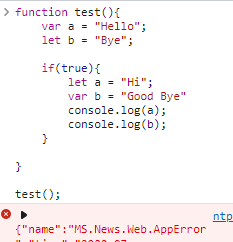
It behave same in the function.

|  |  |
| --- | --- |
|  |  |

Variable Shadowing- Block scope allows shadowing-



inner a is shadow the outer a. It will overlap the value but still it just going to be only accessible inside the block outside the block a sill be “hello”. Shadowing a variable it should not cross the boundary of scope that is we can shadow var variable by using let but can not do the opposite so if we shadow let variable by var variable is know as illegal shadowing and it give us the error that the variable is already defined.



|  |  |
| --- | --- |
| Function Scope- |  |
|  |  |
|  |  |

* Difference between synchronous and asynchronous in JavaScript?

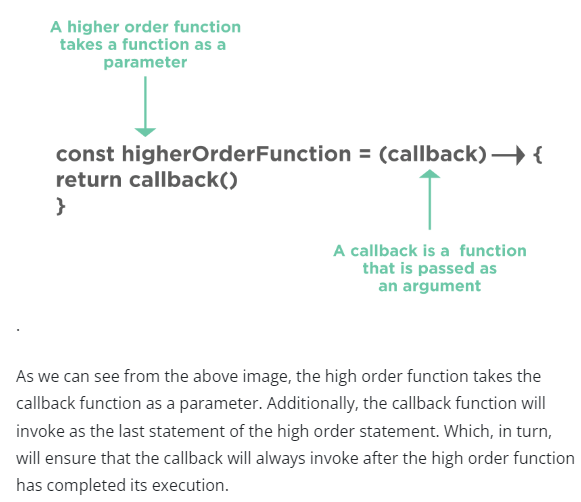
|  |  |
| --- | --- |
| **Synchronous** | **Asynchronous** |
| In Synchronous programming things happen one at the time.  Synchronous programming means code will execute in sequence from top to bottom and get block by long running task such as Http Request. | Asynchronous model allow multiple things to happen at the same time.  Asynchronous Programming fixes this issue by introducing the event loop and Request can run in background without effecting the remaining code and user interface |
| Synchronous functions block the execution of the program until it has finished processing. | Asynchronous functions do not block the execution of the program. |
| Synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed. | Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. |

Few ways to write Asynchronous code in the JavaScript-

* Callback function
* Promises
* Async/Await

Callback Function-

* A callback is a function that is passed as an argument to another function.
* This technique allows a function to call another function.
* A callback function can run after another function has finished executing.
* A Functions that take the additional function as a parameter are called higher-order functions, and the function which passes as an argument is called a callback function.



|  |  |
| --- | --- |
|  | If I want to print second console first. |
|  | |
|  | |

// Suppose there is function function1(), which should be invoked// before function2(), then function1() can take a callback as parameter// which will be invoked as last statement of the function1().

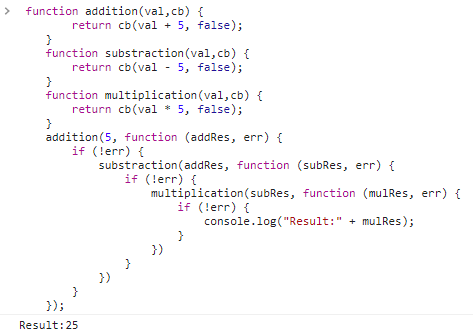
Promises-

* Promises can handle the asynchronous calls in JavaScript.
* A Promise in JavaScript is an object that holds the future value of an asynchronous operation.
* A promise will be "pending" when executed and will result in "resolved" or "rejected", depending on the response of the asynchronous call.

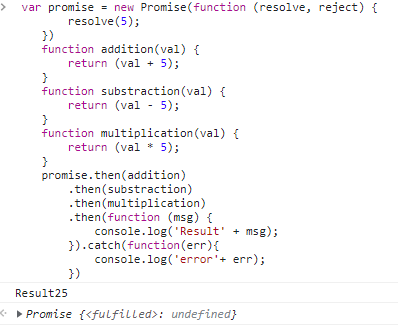
A JavaScript Promise object can be:

* Pending: it is an initial state, where the result is not ready, and it is waiting for the asynchronous operation to get finished.
* Resolved/Fulfilled: it means that performed action completed successfully. i.e., Function returned promised value.
* Rejected: it means that performed action failed or an error occurred. i.e., function did not return the promised value.
* Promises avoid the problem of "callback hell", which happens due to nested callback functions.

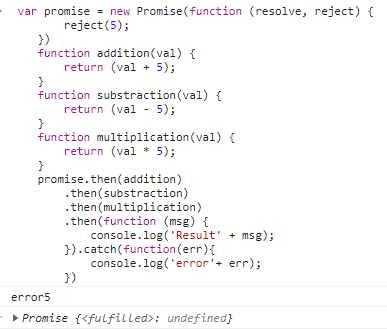
Callback hell- This is a big issue caused by coding with complex nested callbacks. Here, each and every callback takes an argument that is a result of the previous callbacks. In this manner, The code structure looks like a pyramid, making it difficult to read and maintain. Also, if there is an error in one function, then all other functions get affected.



Condition:Resolve-



Condition:Reject-



".then", a method that will perform some of the action only when a given promise is fulfilled or resolved

 ".catch " block that will allow handling the broken or instances of error.

|  |
| --- |
| // Promise Syntax  let myPromise = new Promise(function (myResolve, myReject) {  // "Producing Code" (May take some time)  myResolve(); // when successful  myReject(); // when error  });  myPromise.then( ).catch(); |

Call, Promises, syn- await Examples-

<script>

    // Create the problem statement

    // Array of Object

    const datas = [{

            name: "Pragati",

            profession: "software engineer"

        },

        {

            name: "Shubhi",

            profession: "software engineer"

        },

    ];

    function getDatas() {

        setInterval(() => {

            let output = "";

            datas.forEach((data, index) => {

                output += `<li>${data.name}</li>`;

            });

            document.body.innerHTML = output;

        }, 1000);

    }

    getDatas();

    function createdata(newdata) {

        setInterval(() => {

            datas.push(newdata);

        }, 2000);

    }

    createdata({name: "kanchan",profession: "software engineer"})

 //Print after 2000 sec

</script>

For resolving this issue –

<script>

    // Create the problem statement

    // Array of Object

    const datas = [{

            name: "Pragati",

            profession: "software engineer"

        },

        {

            name: "Shubhi",

            profession: "software engineer"

        },

    ];

    function getDatas() {

        setInterval(() => {

            let output = "";

            datas.forEach((data, index) => {

                output += `<li>${data.name}</li>`;

            });

            document.body.innerHTML = output;

        }, 1000);

    }

    function createdata(newdata, callback) {

        setInterval(() => {

            datas.push(newdata);

            callback();

        }, 2000);

    }

    createdata({

        name: "kanchan",

        profession: "software engineer"

    }, getDatas)

    //Jb data push ho gaya tb getData call hua.

</script>

Promises-

<script>

    // Create the problem statement

    // Array of Object

    const datas = [{

            name: "Pragati",

            profession: "software engineer"

        },

        {

            name: "Shubhi",

            profession: "software engineer"

        },

    ];

    function getDatas() {

        setInterval(() => {

            let output = "";

            datas.forEach((data, index) => {

                output += `<li>${data.name}</li>`;

            });

            document.body.innerHTML = output;

        }, 1000);

    }

    function createdata(newdata) {

        return new Promise((resolve, reject) => {

            setInterval(() => {

                datas.push(newdata);

                let error = false;

                // let error = true;

                if (!error) {

                    resolve();

                } else {

                    reject("kuch sahi nahi hai");

                }

            }, 2000);

        })

    }

    createdata({

        name: "kanchan",

        profession: "software engineer"

    }).then(getDatas).catch(err => console.log(err))

    //Jb data push ho gaya tb getData call hua.

</script>

Async/Await-

<script>

    // Create the problem statement

    // Array of Object

    const datas = [{

            name: "Pragati",

            profession: "software engineer"

        },

        {

            name: "Shubhi",

            profession: "software engineer"

        },

    ];

    function getDatas() {

        setInterval(() => {

            let output = "";

            datas.forEach((data, index) => {

                output += `<li>${data.name}</li>`;

            });

            document.body.innerHTML = output;

        }, 1000);

    }

    function createdata(newdata) {

        return new Promise((resolve, reject) => {

            setInterval(() => {

                datas.push(newdata);

                let error = false;

                // let error = true;

                if (!error) {

                    resolve();

                } else {

                    reject("kuch sahi nahi hai");

                }

            }, 2000);

        })

    }

    // async & Await

    async function start() {

        await createdata({

            name: "kanchan",

            profession: "software engineer"

        })

        getDatas();

    }

    start();

</script>

Async/Await-

* One can declare the Async functions in JavaScript by specifying the "async" keyword in front of the function definition.
* For handling the async functions, you can use the "await" keyword while invoking to function to wait for the promise to resolve.
* Usage of async and await makes the program very clean and understandable. The reason being, there is no need to explicitly specify the dependencies of function calls using the ".then()" method.
* What is template literals in ES6?
* Template literal allows concatenation and interpolation.
* In ES6 it is done by “backtick” in a single line. To interpolation a variable simply put into {} braces forwarded by $ sign.

let a = "Hello";

    let b = "Pragati";

    let c = `${(a)} ${b}`;

    console.log(c);

* Rest and Spread operator in ES6?
* Rest spread operator provides a new way to manipulate array and object in ES6
* Rest spread operator is represented by … followed by the variable name.

|  |  |
| --- | --- |
| Rest | Spread |
| Used to merge a list of function arguments into an array.  Or  Rest operator allows us to collects all the remaining elements into an array. | Used to split up array or object properties.  Or  The spread operator allows to spread the value of an array(or any iterable) into single arguments/elements. |

// Example of Rest with Array

    // element pass kr rahe vo combine ho jaate hai

    function add(a, b, c, ...others) {

        // console.log(others);

        console.log(others[0]);

        return a + b + c;

    }

    let res = add(2, 3, 4, 8, 9);

    console.log(res);

    // Example of Spread with Array

    // combine ko divide karege

    var names = ["Pragati", "Shubhi", "Simmi"];

    function getNames(name1, name2, name3) {

        console.log(name1, name2, name3)

    }

    getNames(names[0], names[1], names[2]);

    getNames(...names);

    getNames(names);

// Example of Rest with Object

    var students = {

        name: "Ajay",

        age: "28",

        hobbies: ["dancing", "cooking"]

    }

    // const age = students.age;

    // const {age,name} = students;

    // console.log(name);

    // const { age, ...rest } = students;

    const { ...rest } = students;

    console.log(rest);

    // Spread

    var newStudent={

        ...students,

        age:"29"

    }

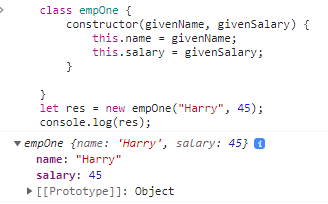
    console.log(newStudent);

let a = [7,8,9];

let b = [1,2,3,4,5, 6,...a,10]

console.log(b);

* Explain Set, weakest, and weak map in ES6?
* Explain Generator function in ES6?
* What is Babel?
* What is Webpack?
* Explain Destructing assignment in ES6?
* Destructing assignment is another improvement in ES6
* It allows us to extract data from array and object into separate variable.
* How to create a JavaScript class in ES6?



* Difference between Local and Session Storage?

|  |  |
| --- | --- |
| Session Storage | Local Storage |
| Data are persistence only for the particular session. It stores the data for only one session. The data is deleted when the user closes the specific browser tab. | In local storage data with no expiration date. The data will not be deleted when the browser is closed and will be available the b=next day, week or year. |
|  |  |

* Diff between undefined and Null?
* Arrow Function in JavaScript?  
    
  Arrow function also called the “fat arrow function”.

It Basically introduced the new way of writing the concise function or we can say that it will simplify the function or it is a shorter way to writing the function.

**Basic function-**

Function hello(){

Return Hello word;  
}

Hello = function(){

Return Hello word;

}

**Arrow function-**

|  |  |  |  |
| --- | --- | --- | --- |
| hello=()=>{  return "Hello word";  }  hello();  o/p- 'Hello word' | hello=()=>"Hello word"  hello();  o/p-'Hello word' | hello=(val)=>"Hello"+val;  hello("Pragati");  o/p-'HelloPragati'; | hello=val=>"Hello"+val;  hello("Pragati");  o/p-'HelloPragati'; |

Few things to note:

* It reduces the size of the code.
* For a single-line function, the return statement is optional.
* Bind the context lexically.
* For a single-line statement, functional braces are not required.
* Doesn’t work with ***new***
* Shortest JavaScript Program i.e., Empty file
* What is window in JS?
* Difference between undefined and not define?
* JS is loosely typed language or weakly typed lang?
* Scope Chain?
* What is “Temporal dead zone”?
* Are let and const declaration hoisted?
* Diff. between syntax Error vs Reference Error vs. Type Error?
* What is block and Scope in JavaScript?
* What is shadowing in JS?
* What is Closers?

Closers in action that is inner function can have access to outer function variable as well as all the global variables.  
Closers is the combination of the function. Its bind together with the lexical scope.

Suppose we have nested function.

Function outer(){

Function inner(){

//so this inner function access to the outer function.

}

}

<script>

    function x() {

        var a = 7;

        function y() {

            console.log(a);

        }

        y();

    }

    x();

</script>

function x() {

        var a = 7;

        function y() {

            console.log(a);

        }

        return y;

    }

    var z = x();

    console.log(z);

    z();

 var sum = function (a) {

        var c = 5;

        return function (b) {

            return a + b + c;

        }

    }

    var res = sum(2);

    console.log(res(5));

Data Hiding, Data Privacy and Encapsulation we can achieve by closers.

function sum(a, b, c) {

        return a + b + c;

    }

    sum(2, 3, 4);

function sum(a) {

        return function (b) {

            return function (c) {

                console.log(a + b + c);

            }

        }

    }

    var res = sum(2);

    var data = res(3);

    var data1 = data(4);

    console.log(data1);

|  |  |
| --- | --- |
|  |  |
|  |  |

* What is Anonymous Function?
* What is the first-Class Function?
* Diff between Function statement vs. Function Expression vs. Function Declaration?
* Diff between Parameter and the Argument?
* Call, Appy and Bind in JS?

|  |  |  |
| --- | --- | --- |
| Call | Appy | Bind |
| call() method, you can write a method that can be used on different objects. | the apply() method, you can write a method that can be used on different objects.  **The apply() method is similar to the call() method.** | Bind () same as like the call () only diff is it will not directly call it will make the copy and whenever is needed it will call. |
| The call() method takes arguments separately. | The apply() method takes arguments as an **array**. |  |

<script>

    // Problem Statement (Call ())

    let userdetails = {

        name: "Pragati",

        Age: 26,

        Designation: "Software Engineer",

        printDetails: function () {

            console.log(this);

        }

    }

    userdetails.printDetails();

    let userdetails2 = {

        name: "Shubhi",

        Age: 27,

        Designation: "Software Engineer"

        // It is not the right way for calling the function again

        // printDetails:function() {

        //     console.log(this.name);

        // }

    }

    // function borrowing

    userdetails.printDetails.call(userdetails2);

</script>

<script>

    // Problem Statement (Call ())

    let userdetails = {

        name: "Pragati",

        Age: 26,

        Designation: "Software Engineer",

    }

    let printDetails = function () {

        console.log(this.name);

    }

    printDetails.call(userdetails);

    let userdetails2 = {

        name: "Shubhi",

        Age: 27,

        Designation: "Software Engineer"

    }

    // function borrowing

    printDetails.call(userdetails2);

</script>

<script>

    // Problem Statement (Call ())

    let userdetails = {

        name: "Pragati",

        Age: 26,

        Designation: "Software Engineer",

    }

    let printDetails = function (state,country) {

        console.log(this.name+" "+state+" "+country);

    }

    printDetails.call(userdetails,"Delhi","India");

    let userdetails2 = {

        name: "Shubhi",

        Age: 27,

        Designation: "Software Engineer"

    }

    // function borrowing

    printDetails.call(userdetails2,"Delhi","India");

</script>

Apply Example-

<script>

    // Problem Statement (Apply ())

    let userdetails = {

        name: "Pragati",

        Age: 26,

        Designation: "Software Engineer",

    }

    let printDetails = function (state, country) {

        console.log(this.name + " " + state + " " + country);

    }

    printDetails.apply(userdetails, ["Delhi", "India"]);

    let userdetails2 = {

        name: "Shubhi",

        Age: 27,

        Designation: "Software Engineer"

    }

    // function borrowing

    // printDetails.call(userdetails2,"Delhi","India");

    printDetails.apply(userdetails2, ["Delhi", "India"]);

</script>

Bind example-

<script>

    // Problem Statement (Bind ())

    let userdetails = {

        name: "Pragati",

        Age: 26,

        Designation: "Software Engineer",

    }

    let printDetails = function (state, country) {

        console.log(this.name + " " + state + " " + country);

    }

    // printDetails.apply(userdetails, ["Delhi", "India"]);

    let userdetails2 = {

        name: "Shubhi",

        Age: 27,

        Designation: "Software Engineer"

    }

    // function borrowing

    // printDetails.call(userdetails2,"Delhi","India");

    // Apply

    // printDetails.apply(userdetails2, ["Delhi", "India"]);

    // Bind

    let newfun = printDetails.bind(userdetails, "Delhi", "India");

    console.log(newfun());

    // newfun();

</script>

Example for interview-

let objs = {

name:"Riya",

age:33,

printfun:function(state,country){

console.log(this.name+" "+state+" "+country);

}

}

let objtwo = {

name:"fake",

age:36

}

objs.printfun.call(objtwo,"Pune","India");

objs.printfun.apply(objtwo,["Pune","India"]);

let fun = objs.printfun.bind(objtwo,"Pune","India");

console.log(fun());

42.What is ‘This’ in JS?

* In JavaScript, the this keyword refers to an object.
* Which object depends on how this is being invoked (used or called).
* The this keyword refers to different objects depending on how it is used:
* In an object method, this refers to the **object**.
* Alone, this refers to the **global object**.
* In a function, this refers to the **global object**.
* In a function, in strict mode, this is undefined.
* In an event, this refers to the **element** that received the event.
* Methods like call(), apply(), and bind() can refer this to **any object**.

43.What is Map () in JS?

* Map is similar like object.
* We can use any type of key or value. (Function, string, blank object can be use as a key).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | new Map() | Creates a new Map object | | set() | Sets the value for a key in a Map | | get() | Gets the value for a key in a Map | | clear() | Removes all the elements from a Map | | delete() | Removes a Map element specified by a key | | has() | Returns true if a key exists in a Map | | forEach() | Invokes a callback for each key/value pair in a Map | | entries() | Returns an iterator object with the [key, value] pairs  in a Map | | keys() | Returns an iterator object with the keys in a Map | | values() | Returns an iterator object of the values in a Map | |  |  | |

<script>

    // create the empty map

    const myMap = new Map();

    const key1 = 'myStr', key2 = {}, key3 = function () { };

    // Setting map values

    myMap.set(key1, 'This is a string');

    myMap.set(key2, 'This is blank object');

    myMap.set(key3, 'This is a empty function');

    console.log(myMap);

    // Getting the values from a Map

    let value1 = myMap.get(key1);

    let value2 = myMap.get(key2);

    console.log(value1);

    console.log(value2);

    // Get the size of the map

    console.log(myMap.size);

    // You can loop using for...of to get keys and values

    for (let [key, value] of myMap) {

        console.log(key, value);

    }

    // Get only keys

    for (let key of myMap.keys()) {

        console.log('key is ', key);

    }

    // Get only values

    for (let key of myMap.values()) {

        console.log('value is ', key);

    }

    // you can loop through a map using for each loop

    // the order will be value and key in the parameter

    myMap.forEach((value, key) => {

        console.log('key is-', key);

        console.log('value is-', value);

    });

    // Coverting map into an Array

    let myArray = Array.from(myMap);

    console.log('Map to array is', myArray);

    // Coverting map keys into an Array

    let myKeysArray = Array.from(myMap.keys());

    console.log('Map to keys into array is', myKeysArray);

    // Coverting map value to an Array

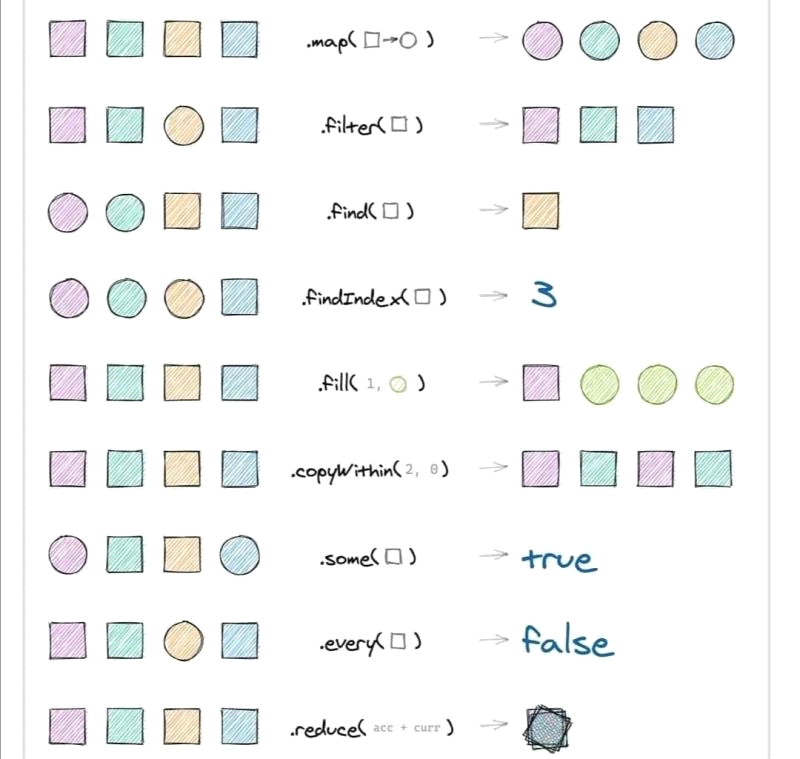
    let myValueArray = Array.from(myMap.values());

    console.log('Map to values into array is', myValueArray);

</script>

44.Map(), Filter() and reduce()?

|  |  |  |
| --- | --- | --- |
| Map(), | Filter() | reduce() |
| map creates a new array by transforming every element in an array individually. | filter creates a new array by removing elements that don't belong. | reduce takes all of the elements in an array and reduces them into a single value. |



|  |  |  |
| --- | --- | --- |
|  | **forEach()** | **map()** |
| **Functionality** | Performs given operation on each element of the array | Performs given "transformation" on a "copy" of each element |
| **Return value** | Returns undefined | Returns new array with transformed elements, leaving back original array unchanged. |
| **Preferrable usage scenario and example** | Performing non-tranformation like processing on each element.  For example, saving all elements in the database. | Obtaining array containing output of some processing done on each element of the array.  For example, obtaining array of lengths of each string in the array |
|  | **forEach() example**  chars = ['Hello' , 'world!!!'] ;    var retVal = chars.forEach(function(word){  console.log("Saving to db: " + word)  })    console.log(retVal) | **map() example**  chars = ['Hello' , 'world!!!'] ;    var lengths = chars.map(function(word){  return word.length  })    console.log(lengths) //[5,8] |

<script>

    const companies = [

        { name: "Google", category: "product based", start: 1981, end: 2004 },

        { name: "Amazon", category: "product based", start: 1982, end: 2008 },

        { name: "Paytm", category: "product based", start: 1999, end: 2007 },

        { name: "Coforge", category: "Service based", start: 1983, end: 2009 },

        { name: "Mindtree", category: "Service based", start: 1986, end: 2010 },

    ];

    const ages = [33, 12, 20, 16, 54, 21, 44, 61, 13, 15, 45, 25, 64, 32];

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

        console.log(companies[0]);

    }

    // OR

    // forEach

    companies.forEach(function (company, index) {

        console.log(companies[0]);

    });

    // OR

    // forEach with Arrow function

    companies.forEach((company, index) => (console.log(company.name)));

</script>

    // FILTER

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

        if (ages[i] >= 20)

            console.log(ages[i]);

    }

    // using forEach

    const age = ages.filter(function (age) {

        if (age > 20) {

            return true

        }

    })

    console.log(ages);

    // using forEach Arrow function

    const finalAge = ages.filter(age => (age >= 20))

    console.log(finalAge);

////////////////////////////////////////////////////////////////

    const sb = companies.filter(function (company) {

        if (company.category === "Service based") {

            return true;

        }

    });

    const sb = companies.filter(company => company.category === "Service based");

    console.log(sb);

 // MAP

    companies.map((company, index) => {

        console.log(company, index)

    })

    const dummy = companies.map((company, index) => {

        return `${company.name} ${company.category}`

    })

    console.log(dummy);

// sort

    const sorted = companies.sort(function (c1, c2) {

        if (c1.start > c2.start) {

            return 1;

        } else {

            return -1;

        }

    })

    console.log(sorted);

    // Using Ternary

    const sortedd = companies.sort((c1,c2) => (c1.start < c2.start ? 1 : -1));

    console.log(sortedd);

    // const sortedAge = ages.sort((c1,c2) => (c1.start < c2.start ? 1 : -1));

    // console.log(sortedAge);

    const sortedAge = ages.sort((a,b) => (b-a));

    console.log(sortedAge);

45.What is set in ES6?

* Set is the collection of well-defined objects.
* Set store the unique values. (It preserved the unique values)
* Means it will not repeat/ count the values.

|  |  |
| --- | --- |
| **Method** | **Description** |
| new Set() | Creates a new Set |
| add() | Adds a new element to the Set |
| delete() | Removes an element from a Set |
| has() | Returns true if a value exists |
| clear() | Removes all elements from a Set |
| forEach() | Invokes a callback for each element |
| values() | Returns an Iterator with all the values in a Set |
| keys() | Same as values() |
| entries() | Returns an Iterator with the [value,value] pairs from a Set |
|  |  |

<script>

const mySet = new Set();

    console.log('The set looks like:', mySet);

    // Adding value to this set

    mySet.add('this');

    mySet.add('My Name');

    mySet.add('this');

    mySet.add('that');

    mySet.add(345);

    mySet.add(true);

    console.log('The set looks like this', mySet);

    // Buse a constructor to initialize the set

    let mySet2 = new Set([1, 45, 'this', false, { a: 4, b: 5 }, 'this']);

    console.log('New set', mySet2);

    // Get the value of the set

    console.log(mySet.size);

    //has will return the true and false value

    //check whether set has 346 or not

    console.log(mySet.has(345)); //true

    console.log(mySet.has(346)); //false

    // Delete the value

    console.log('Before removal', mySet);

    mySet.delete('that');

    console.log('After removal', mySet);

    console.log('Before removal', mySet.has('this'));

    mySet.delete('this');

    console.log('After removal', mySet.has('this'));

    // Iterating a set

    for (let item of mySet) {

        console.log('item is :', item);

    }

    // Itereate by using foreach()

    mySet.forEach(element => {

        console.log('item is forEach :', mySet);

    });

    // Coverting set into an Array

    // We can remove the duplicate

    let myArray = Array.from(mySet);

    console.log('Set to array is', mySet);

</script>

46.Prototype in JS?

All JavaScript objects inherit properties and methods from a prototype.

Example-

The Object.prototype is on the top of the prototype inheritance chain:

* Date objects inherit from Date.prototype
* Array objects inherit from Array.prototype

Date objects, Array objects, and Person objects inherit from Object.prototype.

function emp(name, salary) {

        this.name = name;

        this.salary = salary;

    }

    let h = new emp("Harry", 44);

    function pro(name, salary, lang) {

        emp.call(this, name, salary);

        this.lang = lang;

    }

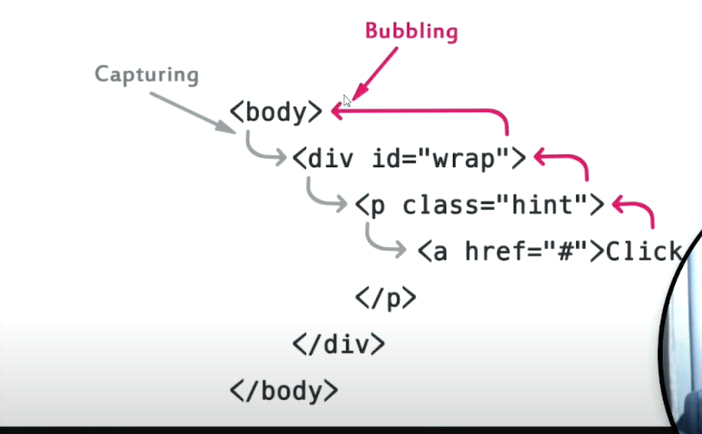
    pro.prototype = Object.create(emp.prototype);

    pro.prototype.constructor = emp;

    let hh = new pro("Potter", 87, "js");

    console.log(hh);

47.Event Bubbling, Capturing and Stop Propagation, Immediate Propagation & Prevent Default?



|  |  |
| --- | --- |
| Event bubbling | Event capturing (Event Trickling) |
| event bubbling means propagation is done from child element to ancestor(parent) elements in the DOM. | Event capturing means propagation of event is done from ancestor(parent) elements to child element in the DOM |

Event bubbling example-

<body>

    <div class="boreder">

        <button>click</button>

    </div>

</body>

<script>

    let div = document.querySelector('div');

    let button = document.querySelector('button');

    div.addEventListener('click', function () {

        console.log('div');

    });

    button.addEventListener('click', function () {

        console.log('button');

    })

</script>

o/p- button

Div

JQuery-

$(document).ready(function(){

$("div").click(function(){

console.log('div');

});

$("button").click(function(){

console.log('button');

});

});

Event capturing example-

<body>

    <div class="boreder">

        <button>click</button>

    </div>

</body>

<script>

    let div = document.querySelector('div');

    let button = document.querySelector('button');

    div.addEventListener('click', function () {

        console.log('div');

    }, true);

    button.addEventListener('click', function () {

        console.log('button');

    }, true)

</script>

o/p- div

button

we can stop Event bubbling and Event capturing by using stop.propogation()

<body>

    <div class="boreder">

        <button>click</button>

    </div>

</body>

<script>

    let div = document.querySelector('div');

    let button = document.querySelector('button');

    div.addEventListener('click', function () {

        console.log('div');

    });

    button.addEventListener('click', function(event) {

        event.stopPropagation();

        console.log('button');

    });

</script>

The preventDefault() –

The preventDefault() method stops the default action of a selected element from happening by a user.

Example-

* It prevents a link from following the URL so that the browser can't go another page.
* It prevents a submit button from submitting a form.

<body>

    <a href="home.html">click here</a>

</body>

<script>

    let a = document.querySelector('a');

    a.addEventListener('click', function (event) {

        event.preventDefault();

        console.log('button');

    });

</script>

Event Deligation:-

Event Delegation is basically a pattern to handle events efficiently. Instead of adding an event listener to each and every similar element, we can add an event listener to a parent element and call an event on a particular target using the .target property of the event object.

48.Currying in JS?

**Currying**is a technique of evaluating function with multiple arguments, into sequence of functions with single argument. In other words, when a function, instead of taking all arguments at one time, takes the first one and return a new function that takes the second one and returns a new function which takes the third one, and so forth, until all arguments have been fulfilled.

**Uses of currying function**

  a) It helps to avoid passing same variable again and again.

  b) It is extremely useful in event handling.

* It is a new way / approach of writing the function and passing the argument.
* Takes the first one and return a new function that takes the second one and returns a new function which takes the third one and passing the arguments at one time. Using the currying technique multiple arguments passing into sequence of functions with the single arguments.

<script>

    // Normal Case

    function Addition(a, b, c) {

        return a + b + c;

    }

    let rec = Addition(2, 3, 4);

    console.log(rec);

    // Suppose we have the senario like this

    function Addition(a) {

        return function (b) {

            return function (c) {

                return a + b + c;

            }

        }

    }

    // let rec = Addition(2);

    // let data = rec(3);

    // let data1 = data(4);

    // console.log(data1);

    // Using Curring

    let rec = Addition(2)(3)(4);

    console.log(rec);

</script>

// Real Case

    userObj = {

        name: "Prgati",

        age: 26

    }

    function userInfo(obj) {

        return function (userInfo) {

            return obj[userInfo];

        }

    }

    let rescs = userInfo(userObj);

    console.log(rescs('name'));

49.Infinite Currying?

<script>

    // function add(a) {

    //     return function(b) {

    //         return function() {

    //             return a + b;12

    //         }

    //     }

    // }

    // console.log(add(2)(3)());

    // console.log(add(2)(3)(7));

    // Again and again return the function

    // Infinity currying

function add(a) {

        return function (b) {

            if (b) return add(a + b);

            return a;

        }

    }

    console.log(add(2)(3)(7)());

    // a= 2

    // b= 3

    // a= 5

    // b= 7

    // a= 12

</script>

50.Higher order function?

51.Event Loop

52.Array and String Methods?

53.Object creation Methods?

54.For in and for of?

Complexity/ Readability will increase by using the loops. People can easily understand the code.

|  |  |
| --- | --- |
| for...of | for...in |
| The for...of loop is used to iterate the values of an iterable such as arrays and strings. | The for...in loop is used to iterate the keys of an object. |
| The for...of loop cannot be used to iterate over an object. | You can use for...in to iterate over an iterable such as arrays and strings but you should avoid using for...in for iterables. |

 // Traditional For Loop

    let people = ["Harry", "skillup", "Vikrant"];

    for (let index = 0; index < people.length; index++) {

        const element = people[index];

        console.log(element);

    }

    // Iterate by using for...of loop.

    for (let name of people) {

        console.log(name);

    }

// Iterate the iterator such as array string by using for...in loop but try to avoid using it.

    for (let name in people) {

        const element = people[name];

        console.log(element);

    }

    // OR

    for (let name of people) {

        console.log(name);

    }

    let obj = {

        name: "harry",

        laungage: "hindi",

        location: "Mp"

    }

    // console.log(obj);

    // Iterate the object by using traditional method

    for (let index = 0; index < Object.keys(obj).length; index++) {

        const element = Object.keys(obj)[index];

        console.log(element);

    }

    // Iterate the object by using for...in loop

    for (let key in obj) {

        const element = obj[key];

        console.log(element);

    }

    // we can use for in with string to loop through all the charater

    myString = "This is my string";

    for (const index in myString) {

        console.log(myString[index]);

    }

55.Inheritance?

56.Ajax in JavaScript?

* AJAX stands for Asynchronous JavaScript And XML.(Blocking calls will not happening and DOM rendering will not block)
* AJAX is not a programming language.rather, it is a set of existing technologies(Asynchronous JavaScript And XML)
* AJAX is a technique for accessing web servers from a web page.
* AJAX helps in fetching data Asynchronous without interfering with the existing page.
* No page reload/refers.
* Modern website use JSON instead or XML (human readability is less)for data transfer.

Advantages-

* Read data from a web server - after the page has loaded
* Update a web page without reloading the page
* Send data to a web server - in the background.

How it works?

* AJAX uses XMLHttpRequest object (also called xhr object) to achieve this.
* Data can be transferred in any format and protocol. (http, https)

Steps-

1. Create the object-

const xhr = new XMLHttpRequest();

1. open the object (It have three parameter. Get/Post, from where the data will come, Asynchronous (true) or synchronous )

xhr.open('GET', 'harry.txt', true);

Get- only fetch the data (only provide the url )

Post- will give the data along with the url. According to the data it will provide the response.

1. What to do on progress (optional) When we are using the spinner

xhr.onload = function () {

          console.log(this.responseText);

    }

<script>

    console.log('ajax tutorial');

    let fetchBtn = document.getElementById('fetchBtn');

    fetchBtn.addEventListener('click', buttonClickHandler);

    function buttonClickHandler() {

        console.log('You have click the fetchBtn');

        // Instantiate an xhr object

        const xhr = new XMLHttpRequest();

        // open the object

        // xhr.open('GET', 'harry.txt', true);

        // xhr.open('GET', '<https://jsonplaceholder.typicode.com/todos/1>', true);

        xhr.open('POST', '<http://dummy.restapiexample.com/api/v1/create>', true);

        xhr.getResponseHeader('Content-type', 'application/json')

        // What to do on progress (optional) When we are using the spinner

        xhr.onprogress = function () {

            console.log('On Progress');

        }

        // Old function (Total ready state is 0,1,2,3,4,5)

        // Value    State                   Description

        // 0        UNSENT                  Client has been created.open() not called yet.

        // 1        OPENED                  open() has been called.

        // 2        HEADERS\_RECEIVED        send() has been called, and headers and status are available.

        // 3        LOADING Downloading;    responseText holds partial data.

        // 4        DONE                    The operation is complete.

        // xhr.onreadystatechange = function () {

        //     console.log('ready state is', xhr.readyState);

        // }

        // What to do when response is ready  (We can check the status)

        // if (this.status === 200) {

        xhr.onload = function () {

            console.log(this.responseText);

        }

        // }

        // else {

        //     console.log("Some error is occur");

        // }

        // send the request

        params = `{"name":"test","salary":"123","age":"23"}`

        xhr.send(params);

        console.log('We are done');

    }

    let popBtn = document.getElementById('popBtn');

    popBtn.addEventListener('click', popHandler);

    function popHandler() {

        console.log('You have click the popHandler');

        // Instantiate an xhr object

        const xhr = new XMLHttpRequest();

        // open the object

        xhr.open('GET', '<http://dummy.restapiexample.com/api/v1/employees>', true);

        // What to do when response is ready  (We can check the status)

        xhr.onload = function () {

            // if (this.status === 200) {

            let obj = JSON.parse(this.responseText);

            console.log(obj);

            let list = document.getElementById(list);

            str = "";

            for (key in obj) {

                str += `<li>${obj[key].employee\_name}</li>`

            }

            list.innerHTML = str;

            // }

            // else {

            //     console.log("Some error is occur");

            // }

        }

        xhr.send();

        console.log('We are done fetching employee');

    }

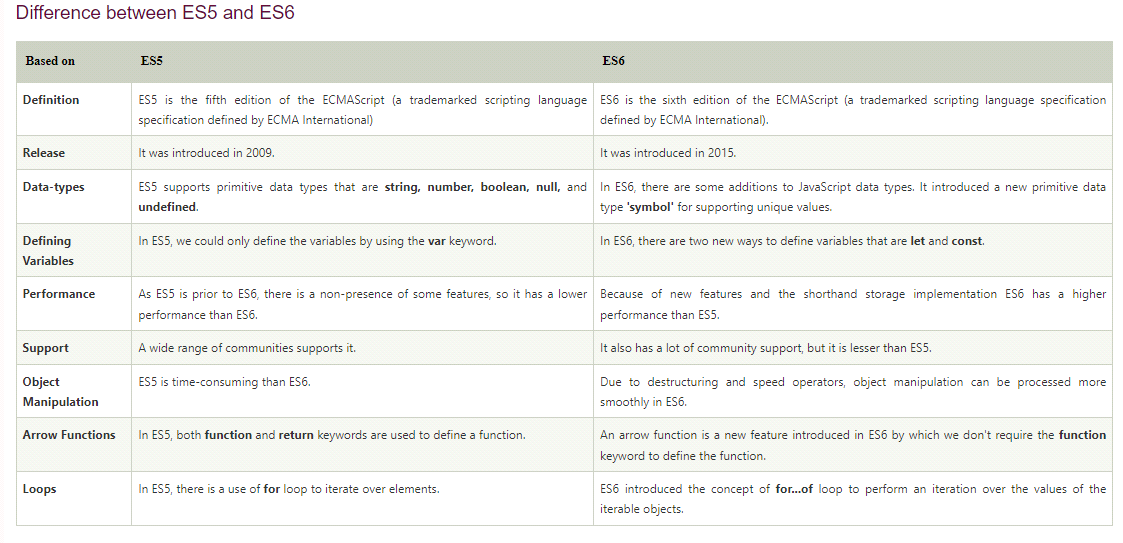
</script>

57. Difference between Get() and post()?

58.Cookies

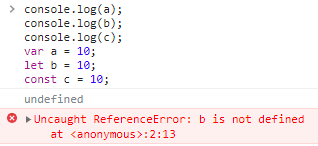
* Cookies let you store user information in web pages.
* Cookies are data, stored in small text files, on your computer.
* When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.
* To create the cookies we can use  document.cookie

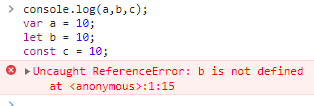
Ex:- document.cookie = "key1 = value1; key2 = value2; expires = date";

59.Memoization and How to Memoriz

Output Question and Answers-

Que1:-

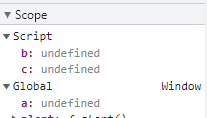




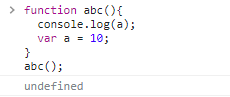
a is going to be undefined but the variable b and c going to be hoisted as well. But they are going to be hosted in the temporal dead zone.

Temporal dead zone is term to describe the state where the variables are in the scope but they are not yet declared.

When we put the debugger a is in the global scope and b,c is in the script.



Que2:-



Because it has initializes this function but it has not initializes the variable yet.

a is undefined at this moment but if the console after this variable a then obviously it will print10

Implicit and Explicit Binding-

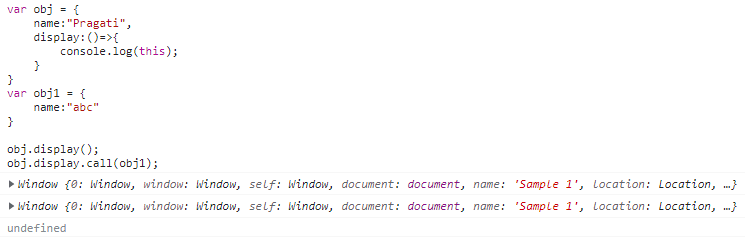
Implicit Binding is applied when you invoke a function in an Object using the dot notation.

Explicit Binding can be applied using call(), apply(), and bind().

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
| Normal Anonymous function | Arrow Function |
|  |  |

Arrow function perform different from an normal anonymous function because arrow function pointing towards the global or the window object and inside of the window object there is no such variable called “name” so that is why it is giving us an empty console log.

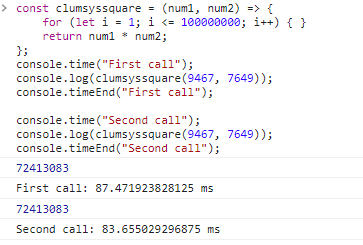


Que3:-

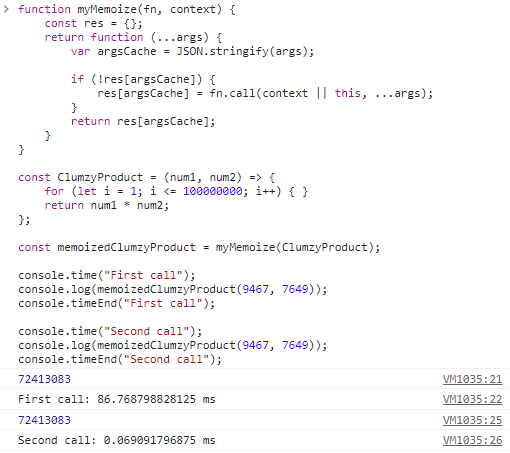
Implement Caching/Memoize Function-

I have given with this function which has some expensive calculation inside of it which when executed every time takes a decent amount of time to run so if I go on and run this.

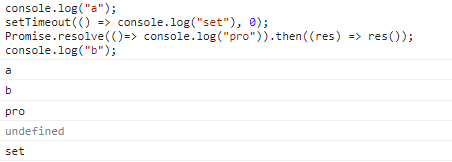
So I am gonna use console.time to measure how much time did this took to run and I am gonna run it twice actually with the same parameters. So first time it took 40milisecond and second time took 42miliseconds so how do we minimize this time calculation. If the parameter of the function are same we need to cache the result of the previous function somewhere right so that is what we are going to implement the functions.



Memoize-



Que4:-



**JavaScript Visualizer 9000 (Real Example we can see on this site)**

First going to be console.log(a). We have setTimeout and setTimeout is not the part of JS . This is the part of web API this the part of our browser. setTimeout will runs when the complete code inside our js file has ran successfully .Even the time inside the setTimeout 0 seconds still it going to run at the very last. So this will go inside of our task queue(call stack queue). promises.resolve it will run also very last of our program. After our complete code inside of our JS has ran but this won’t go inside of our task queue(call stack n queue) it’s gonna go inside of our micro task queue or the priority queue which again .After that we have console.log(b). Then we will have the output of b. Since micro task queue or the priority queue has the highest priority than the call stack queue. So this will run first and this will give the output of pro then the output of set.

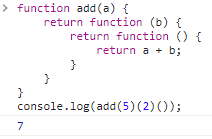
Que5-

Infinite Currying

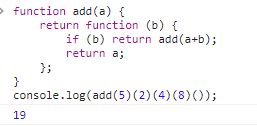
console.log(add(5)(2)(4)(5)());

Interview provided this code snippet and he asked to implement this add function.

First way- (It will works only for the 2 parameter)



So if the code have more complexity . we want the code which works infinitely.

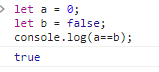


We have two of these but then I am gonna check if we have any more params or not. So I will check b has a param then just return add again. But I does not have any value inside of it that means its coming to an end so I am just return a.

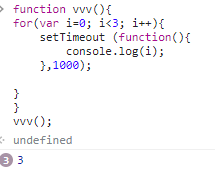
So let say we have 5,2,4,8 over here so first of all we provided it with the 5 and it returned us a function. Then we again called b with 2 but if its gonna check if 2 is there or not . If there is some value inside of b or not only then its going to return the addition otherwise its just going to return the initial 5 value the sum of those numbers up until that point so we have 2 at that point. If 2 is there return a + b so 5+2=7 went inside of this function so now its value is 7 over here but then again its going to return another function then its going to check if we are calling that function as empty but no this is not empty it has 4 inside of it so then again its going to check if there is something inside of the b then returned a + b at this point a value was 7 so its going to say 7 plus 4 that is 11 so its going to pass 11 to that so this cycle run again and again until we reach this point so up until this point the value was 19 so then its gonna have a 19 value over here and its gonna return a function again and then its gonna check if b has something but here b does not have anything so its not gonna go inside of it and its going to return a that is 19 from here so our answer is going to be 19.

Global Logic Ques-

Que1 -



Que2-



SetTimeOut is basically the web API . Use for after certain amount of time function will form parallary . Here event-loop comes in the picture.It continuously monitor the call stack and call stack queue. If the call stack is empty it will push the function from call stack queue to call stack.

Now the for loop will work.

In the first iteration i=0 it will go inside and print the value and go outside.

In the second iteration value will be 1

In the third iteration value will be 2

Here is forth iteration will happen does not matter i<3 value will be false when the value will iterate. Because the value is equal to 3 not smaller then 3. So it will go outside from the for loop.

That is the reason 3 will print.

But the queue is why it will print 3times 3??

Immediate function invocation(IIF)-

