**JavaScript Interview Questions – Part2**

1. **What do you mean by scope and lexical scope?**

In JavaScript, scope refers to the current context of code, which decides which variables are accessible to JavaScript. Basically scope manages accessibility of variables.

**There are two types scopes in JS**

* Global Scope
* Local Scope (Block scope and function scope)

**Global Scope**

Global scope means it is not wrapped with any brackets ({) and it is accessible from everywhere.

const appName = "Learn JS with Nisha";

if(true){

console.log(appName);// Learn JS with Nisha

}

function getAppName(){

return appName; //Learn JS with Nisha

}

getAppName();

Here, appName is a global variable as it is not wrapped with any brackets and therefore it can be accessible from everywhere.

**Local Scope:**

Local scope means a variable can be accessible only from inside that area where it was defined. It is not accessible from outside.

So in JavaScript, two things can create a local scope, one using any bracket {} like if else, for- loops and other is function. That’s why local scope consist of two things

* **Block Scope**
* **Function Scope**

Now one point to remember, when we declare variable with var keyword, there local scope can be created by only functions there is no block scope.   
  
It means if we declare any variable inside any block of if-else and for-loop using var, it will still treated like a global scope , as only functions can create local scope in JavaScript in case of var.

Block scope came into the picture with the introduction of let and const only.

Let’s understand with example

So here, x variable declared with var keyword inside a block(if block) so there is no block scope with var keyword hence it is accessible outside also. On the other side y variable is also created with var but it is inside function, in this case it will create function scope and will only be accessible inside function , will not be accessible outside hence will get “Reference Error” when try to access it outside.

if(true){

var x =10;

console.log(x); // 10

}

function getValue(){

var y=20;

console.log(y); // 20

}

getValue();

console.log(x); //10

console.log(y); //Uncaught ReferenceError: y is not defined

Now when we will try the same thing with let/const, it will have block scope as well and that’s why x and y will not be accessible outside of it block and function scope.

if(true){

const x =10;

console.log(x); // 10

}

function getValue(){

const y=20;

console.log(y); // 20

}

getValue();

console.log(x); //Uncaught ReferenceError: x is not defined

console.log(y); //Uncaught ReferenceError: y is not defined

**2) Difference between let and var?**

var and let are both used for variable declaration in JavaScript but the difference between them is that var is function scoped and let is block scoped.

The let keyword was introduced in the later version of JavaScript known as ES6(ES2015). And it's the preferred way to declare variables

Variable declared with var keyword get hoisted with default value of “undefined” while variable declared with let get hoisted with no value that’s why we get Reference Error as variable is inside Temporal Dead Zone.

console.log(a); //undefined

console.log(b); // Uncaught ReferenceError: Cannot access 'b' before initialization

var a=10;

const b=20;

* “var” allows to declare variables but let doesn’t allow to declare the variables.

var a=10; //10

var a=20; // 20

let a=10;

let a=20; // Error

1. **What are arrow functions?**

ES6 arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.

without Arrow Function with Arrow Function

const calculate = (x ,y) => x + y;

function calculate(x,y) {

return x+y

}

In the above example, arrow function has one expression x+y, so it return the result(There is no need to mention return keyword, if you logic move to multiple expressions then explicit return keyword is required)

***Points to remember:***

An arrow function doesn’t have its own this value. Instead, it uses the value of the enclosing lexical scope. An arrow function also doesn’t have the arguments object.

Avoid using the arrow function for event handlers, object methods, prototype methods, and functions that use the arguments object.

1. **What do you understand by Callback Functions?**   
     
   A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Callbacks are quiet useful in asynchronous programming where we want to execute certain logic after something happens (not sequentially)  
  
**setTimeout** and **setInterval** are built in functions, that takes a function as argument and execute it after certain interval, so here this function will not execute sequentially instead it will execute once those seconds will passed, So the function which we have passed to setTimeout/setInterval is called Callback function

**CallBack Function**

setTimeout(function(){

console.log('Execute it after 2 seconds')

}, 2000);

1. **What are Closures in JavaScript?**   
     
   We can create nested functions in JavaScript (Outer function and Inner function). So when you inner function remembers the scope of outer function even after your outer function has completed it execution.

Closures give your inner function access to its outer scope (Lexical environment).  
  
The closure has three scope chains listed as follows:

* Access to its own scope.
* Access to the variables of the outer function.
* Access to the global variables.

function counter(){

let counter =0;

return function (){

return counter+=1;

}

}

const count = counter();

console.log(count()); // 1

console.log(count()); // 2

console.log(count()); // 3

Here, counter variable we have declared inside outer function and inner function is remembering it value with every function call, this all is happening because of Closures.

**Where to use closures?**  
Closures are frequently used in JavaScript for object data privacy, in event handlers and callback functions, and in partial applications, currying, and other functional programming patterns.

1. **What is Prototype in JavaScript?**  
     
   JavaScript is a prototype-based language; every function and object has a prototype property by default.

Prototype property is basically an object (also known as Prototype object), where we can attach methods and properties in a prototype object, which enables all the other objects to inherit these methods and properties

function User () {

this.name = 'Nisha'

}

const user1 = new User();

User.prototype.city='Delhi';

console.log(user1.city) // Delhi

In the above example, we have added city property to User constructor function using  
User.prototype.city='Delhi';

Then object user1 will inherit the property city from prototype property of User constructor function

**7. What are Higher Order functions ?**

If a function accepts other function as argument or if a function return another function as argument then that function is called Higher Order function.

There are many built in higher order function in JavaScript like map, sort, reduce, filter, forEach are the example of Higher Order functions.

Higher-order functions provide the code reusability.

const num= [1,2,3];

num.map(function(x){

console.log(x\*2); // 2,4,6

});

Here, as map function is taking another function as argument, which is a callback function, and map is a Higher order function.

**8) Explain Object destructuring syntax.**

The Object destructuring introduced in ES6.

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables

const user = {

id: 121,

name: 'Nisha'

}

const { name } = user;

console.log(name); // Nisha

1. **What is null and undefined in javascript**

**Null**

* Null represents an empty value and carry no meaningful information. It is intentional absence of value.
* It is type of Object

.  
When we define a variable to null then we are trying to convey that the variable is empty.

**Undefined**

* Undefined represents any value that has not been defined anywhere in the program.
* It is type of undefined
* When we define a variable to undefined then we are trying to convey that the variable does not exist

console.log(a); // undefined

var a = null;

console.log(a); // null

**Points to remember**

* null and undefined are values that contain nothing.
* null and undefined are loosely equal to each other but are strictly not equal to each other.

console.log(null == undefined); // true

console.log(null === undefined); // false

1. **What is Recursion and list out its use cases**

Recursion is a process of calling itself. So when a function calls itself is called a Recursive function.

Recursive function must have a terminate condition also known as base condition ( otherwise function is called indefinitely with Error ***Uncaught RangeError: Maximum call stack size exceeded.*** )

Its very similar to if…else, like in if block we can have our exist or base condition and in else part we can have our recursive call

function factorial(num){

if(num === 1){

return num;

}

return num \* factorial(num-1)

}

console.log(factorial(2));

**Use Case:**

Recursion can be useful wherever you need to iterate to a complex structure Like DOM Nodes, Networks or Graphs.

1. **What is Function Currying?**

We usually pass all the arguments to a function at one go like this

function sum(x,y){

return x+y;

}

sum(10,20);

But now If I do something like this ?

Sum(10)(20);

Looks confusing, right? But when we call function with this pattern, this is a concept of Currying.

Currying is an advanced technique of working with functions. So we can say,

“Currying is a function that takes one argument at a time and returns a new function expecting the next argument”

function sum(x){

return (y) =>{

return x+y;

}

}

console.log(sum(10)(20));

**Why should I use currying?**

There are several reasons why currying is ideal:

* Currying is a checking method to make sure that you get everything you need before you proceed
* It helps you to avoid passing the same variable again and again
* It divides your function into multiple smaller functions that can handle one responsibility. This makes your function pure and less prone to errors and side effects
* It is used in functional programming to create a higher-order function

1. **Difference between call, apply ?**

The ***this*** keyword is very important concept in JavaScript. This is basically reference to an object and the object that **this** refer may vary

**call()** and **apply()** are built in methods in JavaScript which used to bind the context of this explicitly

**call**and **apply**are very similar—they invoke a function with a specified this context, and optional arguments.

The only difference between call and apply is that call requires the arguments to be passed in one-by-one, and apply takes the arguments as an array

Example

const Employee = {

getFullName: function(city, age){

console.log(`Name is ${this.firstName} ${this.lastName} city is ${city} and age is ${age}`)

}

}

const customer = {

firstName: 'Sneha',

lastName: 'Singla'

}

Employee.getFullName.call(customer, 'Delhi', 22); // call takes arguments as comma separated

Employee.getFullName.apply(customer, ['Delhi', 22]); // apply takes arguments as array

1. **Difference between map, filter and forEach**

Whenever we need to iterate over array element, we can think of loops like for loop but **map(), filter() and forEach()** are just other ways through which we can iterate over array to perform specific task on each element

**ForEach**:

Its very similar to for-loop. The function inside the parentheses of the forEach() is simply executed "for each" element of the array.

const numbers= [1,2,3,4];

numbers.forEach(num => console.log(num\*2));

// output 2 4 6 8

forEach() doesn’t create any new array. It returns undefined. So it should use only to perform specific action for each element of an array

const numbers= [1,2,3,4];

const result = numbers.forEach(num => num\*2);

console.log(result); // undefined

**Map:**

Map is very much same like forEach but unlike forEach, map returns a new

Array.

const numbers= [1,2,3,4];

const result = numbers.map(num => num\*2);

console.log(result); // [2,4,6,8]

.map() will always return a new array of the same length as the original!

**Filter():**

* .filter() loops through (or iterates) through data, and filters out data that doesn't match the criteria that we set.
* We define what those criteria are through a truth test inside a function
* This one is pretty straightforward: use it when you want to filter an array based on criteria you want to define yourself.
* For example, perhaps you have an array of objects which represent Users, and you want to create a new list of only the Users who live in Delhi:

const users= [

{id: 1, city: 'Delhi'},

{id: 2, city: 'Pune'},

{id: 3, city: 'Delhi'},

{id: 4, city: 'Delhi'},

]

const result = users.filter(user => user.city == 'Delhi');

console.log(result);

// output

// [object Array] (3)

[// [object Object]

{

"id": 1,

"city": "Delhi"

},// [object Object]

{

"id": 3,

"city": "Delhi"

},// [object Object]

{

"id": 4,

"city": "Delhi"

}]

1. **Is Javascript a statically types or dynamically typed language?**

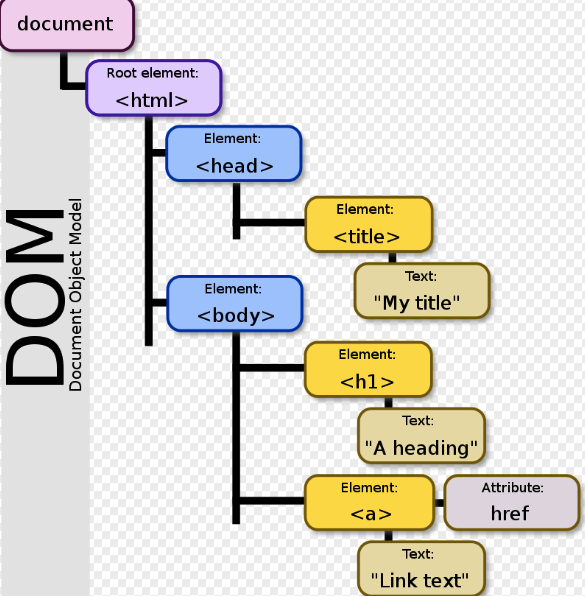
JavaScript is a dynamically typed language.

In dynamically typed languages all type checks are performed in a runtime, only when your program is executing. So this means you can just assign anything you want to the variable and it will work

1. **What is DOM and shadow DOM?**

If we are working with JavaScript then we are familiar with this term DOM. DOM stands for Document Object Model, which is a tree like structure that represents the different elements and strings of text appearing in markup document.

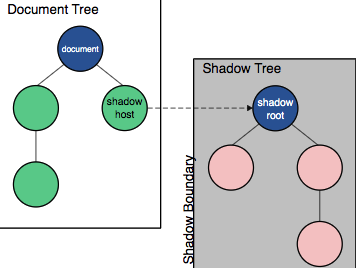
The Document Object Model used to access objects in web pages with eg. javascript



**Shadow DOM:**

An important aspect of web components is encapsulation - being able to keep the markup structure, style, and behavior hidden and separate from other code on the page so that different parts do not clash, and the code can be kept nice and clean. The Shadow DOM API is a key part of this, providing a way to attach a hidden separated DOM to an element

*Shadow* DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a shadow root, underneath which can be attached to any elements you want, in the same way as the normal DOM.



1. **How Promise works in JavaScript ?**

Promises are **used to handle asynchronous operations** in JavaScript.

Prior to promises callback functions were used to handle asynchronous operation but when we have complex requirement or multiple callback calls it leads to callback hell and code also become unmanageable

A Promises has three state:

* **fulfilled**: Action related to the promise succeeded
* **rejected**: Action related to the promise failed
* **pending**: Promise is still pending i.e. not fulfilled or rejected yet

**Create a promise**:

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

//do something

});

Promise constructor takes single argument that is a callback function , callback function takes 2 arguments (resolve, reject)

1. **What is the difference between Function declaration and Function expression**

**Function Declaration/Statement:**

Function Declaration is basically used to declare a function so that we can used it later when we are going to invoke/called it

Like to declare variable we use var/let keyword similarly for function declaration we use “**function**” keyword

function test(){

console.log('this is test function');

}

test();

**Function Expression:**

* A JavaScript function can also be defined using an **expression**.
* A function expression can be stored in a variable:
* let x = function (a, b) {return a \* b};
* After a function expression has been stored in a variable, the variable can be used as a function.
* Functions stored in variables do not need function names.
* They are always invoked (called) using the variable name.

**Difference:**

* Function declarations are hoisted to the top of other code.
* Function expressions aren’t hoisted, which allows them to retain a copy of the local variables from the scope where they were defined.

**Benefits :**

There are several different ways that function expressions become more useful than function declarations

* As closures
* As arguments to other functions
* As Immediately Invoked Function Expressions (IIFE)

1. **What is strict Mode in JavaScript? How you can enable this?**

As we know, JavaScript is a loosely types language. If you have worked with server side language (Java/C#), there if we use a variable before defining it we get an error. But in JavaScript we don’t have such checking.

But in ECMAScript5 or later that is possible with the introduction of “**strict**” mode

1. **What is Coercion in Javascript?**
   * Type Coercion refers to the process of automatic or implicit conversion of values from one data type to another

**Type coercion can be explicit and implicit.**

* + When a developer expresses the intention to convert between types by writing the appropriate code, like Number(value), it’s called explicit type coercion (or type casting).
  + Since JavaScript is a weakly-typed language, values can also be converted between different types automatically, and it is called implicit type coercion. It usually happens when you apply operators to values of different types

One operator that does not trigger implicit type coercion is ===, which is called the strict equality operator. The loose equality operator == on the other hand does both comparison and type coercion if needed

1. **What are Pure Functions?**

A Pure Function is a function (a block of code) that always returns the same result if the same arguments are passed

**A function must pass two tests to be considered “pure”:**

1. **Same inputs always return same outputs**

const sum = (x, y) => x + y; sum(2, 4); // 6

1. **No side-effects :**

A side effect is when a function relies on, or modifies, something outside its parameters to do something

**A few examples of side effects are :**

* Mutating your input
* console.log
* HTTP calls (AJAX/fetch)
* Changing the filesystem (fs)
* Querying the DOM