Follow

1. <https://www.interviewbit.com/javascript-interview-questions/#output-of-following-code-2>

2. <https://www.interviewbit.com/javascript-mcq/>

1.Write the code given If two strings are anagrams of one another, then return true.

solutions:

function areAnagram(str1,str2)

{

let n1 = str1.length;

let n2 = str2.length;

if (n1 != n2)

return false;

str1.sort();

str2.sort()

for (let i = 0; i < n1; i++)

if (str1[i] != str2[i])

return false;

return true;

}

let str1=['g', 'r', 'a', 'm' ];

let str2=['a', 'r', 'm' ,'g'];

// Function Call

if (areAnagram(str1, str2))

document.write("The two strings are"

+ " anagram of each other<br>");

else

document.write("The two strings are not"

+ " anagram of each other<br>");

2.Write the code to find the vowels

Solutios:

const findVowels = str => {

let count = 0

const vowels = ['a', 'e', 'i', 'o', 'u']

for(let char of str.toLowerCase()) {

if(vowels.includes(char)) {

count++

}

}

return count

}

or

function isVowel(ch) {

ch = ch.toUpperCase();

return ch == "A" || ch == "E" || ch == "I" || ch == "O" || ch == "U";

}

// Returns count of vowels in str

function countVowels(str)

{

var count = 0;

for (var i = 0; i < str.length; i++)

if (isVowel(str[i]))

// Check for vowel

++count;

return count;

}

// Main Calling Function

// string object

var str = "abc de";

// Total numbers of Vowel

document.write(countVowels(str));

document.write("<br>");

3. In JavaScript, how do you turn an Object into an Array []?

solutions:

let obj = { id: "1", name: "user22", age: "26", work: "programmer" };

//Method 1: Convert the keys to Array using - Object.keys()

console.log(Object.keys(obj));

// ["id", "name", "age", "work"]

// Method 2 Converts the Values to Array using - Object.values()

console.log(Object.values(obj));

// ["1", "user22r", "26", "programmer"]

// Method 3 Converts both keys and values using - Object.entries()

console.log(Object.entries(obj));

//[["id", "1"],["name", "user22"],["age", "26"],["work", “programmer"]]

4. What is the output of the following code?

const b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

for (let i = 0; i < 10; i++) {

setTimeout(() => console.log(b[i]), 1000);

}

for (var i = 0; i < 10; i++) {

setTimeout(() => console.log(b[i]), 1000);

}

solutions:

1

2

3

4

5

6

7

8

9

10

undefined

undefined

undefined

undefined

undefined

undefined

undefined

undefined

undefined

undefined

5.A BigInt value, also sometimes just called a BigInt, is a bigint primitive, created by appending n to the end of an integer literal,

or by calling the BigInt() function (without the new operator) and giving it an integer value or string value.

var a = BigInt("567")+BigInt("3")

console.log(a);

typeof 1n === "bigint"; // true

typeof BigInt("1") === "bigint"; // true

Symobl:

Symbol - It is a new data type introduced in the ES6 version of javascript. It is used to store an anonymous and unique value.

Example :

var symbol1 = Symbol('symbol');

Because symbols are the only primitive data type that has reference identity (that is, you cannot create the same symbol twice), they behave like objects in some way.

Symbol("foo") === Symbol("foo"); // false

The following syntax with the new operator will throw a TypeError:

const sym = new Symbol(); // TypeError

6.// Hoisting takes place in the local scope as well

function doSomething(){

x = 33;

console.log(x);

var x;

}

doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

Note - Variable initializations are not hoisted, only variable declarations are hoisted:

function doSomething(){

var x;

console.log(x);//undefined because Variable initializations are not hoisted

x = 33;

}

doSomething();

var x;

console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted

x = 23;

Note - To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code:

7."use strict";

x = 23;

console.log(x);// Gives an error since 'x' is not declared

function strict\_function() {

'use strict';

x = 'I am a strict function';

console.log(x);

}

strict\_function(); // ReferenceError: x is not defined

8. Difference between “ == “ and “ === “ operators.

Both are comparison operators. The difference between both the operators is that “==” is used to compare values whereas, “ === “ is used to compare both values and types.

Example:

var x = 2;

var y = "2";

(x == y) // Returns true since the value of both x and y is the same

(x === y) // Returns false since the typeof x is "number" and typeof y is "string"

9.async vs defer in script tag in JavaScript? (https://codedamn.com/news/javascript/async-and-defer-in-script-tag)

Ans:

When it comes to loading external JavaScript files on your website, should you use async or defer? Both methods have their own advantages and disadvantages,

and there is no definitive answer. Ultimately, it depends on your specific needs. This blog post will explain what async and defer actually is and explore

the differences between async and defer, so you can make an informed decision about which method is best for your use case.

Async in script tag in JavaScript is a way to load scripts asynchronously. That means, if a script is async, it will be loaded independently of other scripts on the page, and will not block the page from loading.

If you have a page with several external scripts, loading them all asynchronously can speed up the page load time, because the browser can download and execute them in parallel.

To use async, simply add the async attribute to your script tag:

<script async src="script.js"></script>

By using the defer attribute in HTML, the browser will load the script only after parsing (loading) the page. This can be helpful if you have a script that is dependent on other scripts, or if you want to improve the loading time of your page by loading scripts after the initial page load.

To use defer, simply add the defer attribute to your script tag:

<script defer src="script.js"></script>

The async and defer attributes both allow the browser to continue parsing the HTML document while JavaScript files are being downloaded, but they differ in when those files are

executed.Async downloads and executes JavaScript as soon as it’s available, while defer attribute waits until the HTML document has been parsed before downloading and executing any external scripts.

In most cases, it doesn’t matter which attribute you use – both will improve performance by allowing the browser to continue parsing while waiting for JavaScript to download.

However, there are some situations where one attribute may be preferable to the other.

10.Dynamically typed?

In dynamic-typed languages, we don’t need to define the type of a variable as an interpreter at run-time will handle it.

In the above code: Line 1: We declare a variable number and assign a value to it. Note that we have not defined its type here.

Line 2: We print the value of variable number on the console.

number = 10

print(number)

variable Namin:

11.Camecase : nameFunctionParida = "Bubu;

pascelcasecase : NameFunctionParida = "pari"

Snake case: name\_parida = "jioir"

Except \_ and $ we can't use any special symbol in variable.We can write number in between but we can't write number with special symbol

var 123hd5=8;//like this also we can't

console.log(123hd5);///error

var h45d5=8;//correct

console.log(h15d5);//8

var $123 = 56;//we can write this

console.log($123);//56

var 123$ = 56;//we can't write like this

console.log($123);//error

12.Coercien

Type coercion occurs when values of different data types are compared or used together in an expression and can lead to unexpected results and errors in your code.

There are 2 types :

There are two types of type coercion: implicit and explicit.

implicit:

Type coercion is the process of converting a value from one data type to another. In JavaScript, type coercion happens implicitly when operators or functions are applied to values of different types. For example, if you try to add a string and a number using

the "+" operator, JavaScript will convert the number to a string and concatenate the two values.

let num = 10;

let str = "Hello";

console.log(str + num); // "Hello10"

explicit:

Explicit Coercion: Type conversion is done explicitly in code using the inbuilt functions like Number(), String(), Boolean(), etc.

let number = Number("35");

console.log(number);//35

Logical operators:

Logical operators in javascript, unlike operators in other programming languages, do not return true or false. They always return one of the operands.

OR ( | | ) operator - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.

AND ( && ) operator - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

Example:

var x = 220;

var y = "Hello";

var z = undefined;

x | | y // Returns 220 since the first value is truthy

x | | z // Returns 220 since the first value is truthy

x && y // Returns "Hello" since both the values are truthy

y && z // Returns undefined since the second value is falsy

if( x && y ){

console.log("Code runs" ); // This block runs because x && y returns "Hello" (Truthy)

}

if( x || z ){

console.log("Code runs"); // This block runs because x || y returns 220(Truthy)

}

13.

What is NaN property in JavaScript?

NaN property represents the “Not-a-Number” value. It indicates a value that is not a legal number.

typeof of NaN will return a Number.

To check if a value is NaN, we use the isNaN() function,

Note- isNaN() function converts the given value to a Number type, and then equates to NaN.

isNaN("Hello") // Returns true

isNaN(345) // Returns false

isNaN('1') // Returns false, since '1' is converted to Number type which results in 0 ( a number)

isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)

isNaN(false) // Returns false

isNaN(undefined) // Returns true

14.

Explain passed by value and passed by reference?

In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

\* For understanding passed by value and passed by reference, we need to understand what happens when we create a variable and assign a value to it,

\* Basically, pass-by-value means that the actual value of the variable is passed and pass-by-reference means the memory location is passed

where the value of the variable is stored.

var x = 2;

In the above example, we created a variable x and assigned it a value of “2”. In the background, the “=” (assign operator) allocates some space in the memory, stores the value “2” and returns the location of the allocated memory space. Therefore, the variable x in the above code points to the location of the memory space instead of pointing to the value 2 directly.

Assign operator behaves differently when dealing with primitive and non-primitive data types,

Assign operator dealing with primitive types:

var y = 234;

var z = y;

In the above example, the assign operator knows that the value assigned to y is a primitive type (number type in this case), so when the second line code executes, where the value of y is assigned to z, the assign operator takes the value of y (234) and allocates

a new space in the memory and returns the address. Therefore, variable z is not pointing to the location of variable y, instead, it is pointing to a new location in the memory.

var y = #8454; // y pointing to address of the value 234

var z = y;

var z = #5411; // z pointing to a completely new address of the value 234

// Changing the value of y

y = 23;

console.log(z); // Returns 234, since z points to a new address in the memory so changes in y will not effect z

From the above example, we can see that primitive data types when passed to another variable, are passed by value. Instead of just assigning the same address to another variable, the value is passed and new space of memory is created.

Assign operator dealing with non-primitive types:

var obj = { name: "Vivek", surname: "Bisht" };

var obj2 = obj;

In the above example, the assign operator directly passes the location of the variable obj to the variable obj2. In other words, the reference of the variable obj is passed to the variable obj2.

var obj = #8711; // obj pointing to address of { name: "Vivek", surname: "Bisht" }

var obj2 = obj;

var obj2 = #8711; // obj2 pointing to the same address

// changing the value of obj2

obj2.name = "Akki";

console.log(obj2);

// Returns {name:"Akki", surname:"Bisht"} since both the variables are pointing to the same address.

From the above example, we can see that while passing non-primitive data types, the assign operator directly passes the address (reference).

Therefore, non-primitive data types are always passed by reference.

15.In object redeclaration is possible because it's reference type:

const course = {

lect:30,

sub:"Math",

title:"javascript",

notes:{

intro:"welcome to the course"

},

error(){

console.log("Bye")

},

price:20

}

console.log(course);

course.price=300;//redeclare

console.log(course);

16.In JavaScript, arrays come with a variety of methods that allow you to manipulate their content. Here's a brief explanation of the methods you've mentioned:

push:

Purpose: Adds one or more elements to the end of an array and returns the new length of the array.

Syntax: array.push(element1, ..., elementN)

javascript

Copy code

let fruits = ['apple', 'banana'];

fruits.push('orange', 'grape');

console.log(fruits); // Output: ['apple', 'banana', 'orange', 'grape']

shift:

Purpose: Removes the first element from an array and returns that removed element. It also updates the length property of the array.

Syntax: array.shift()

javascript

Copy code

let fruits = ['apple', 'banana', 'orange'];

let removed = fruits.shift();

console.log(removed); // Output: 'apple'

console.log(fruits); // Output: ['banana', 'orange']

unshift:

Purpose: Adds one or more elements to the beginning of an array and returns the new length of the array.

Syntax: array.unshift(element1, ..., elementN)

javascript

Copy code

let fruits = ['banana', 'orange'];

fruits.unshift('apple', 'grape');

console.log(fruits); // Output: ['apple', 'grape', 'banana', 'orange']

slice:

Purpose: Returns a shallow copy of a portion of an array into a new array object. It does not modify the original array.

Syntax: array.slice(start, end)

javascript

Copy code

let numbers = [1, 2, 3, 4, 5];

let slicedArray = numbers.slice(1, 4);

console.log(slicedArray); // Output: [2, 3, 4]

console.log(numbers); // Output: [1, 2, 3, 4, 5] (original array is not modified)

splice:

Purpose: Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

Syntax: array.splice(start, deleteCount, item1, ..., itemN)

javascript

Copy code

let fruits = ['apple', 'banana', 'orange', 'grape'];

fruits.splice(1, 2, 'kiwi', 'melon');

console.log(fruits); // Output: ['apple', 'kiwi', 'melon', 'grape']

pop:

Purpose: Removes the last element from an array and returns that removed element. It also updates the length property of the array.

Syntax: array.pop()

javascript

Copy code

let fruits = ['apple', 'banana', 'orange'];

let removed = fruits.pop();

console.log(removed); // Output: 'orange'

console.log(fruits); // Output: ['apple', 'banana']

<hello name="{{ name }}"></hello>

<p>

Start editing to see some magic happen :) {{formattedDate}}

</p>

<p>{{send\_date|date:"MM-dd-yyyy hh:mm:ss a"}}</p> in ts send\_date=new Date();

<hello name="{{ name }}"></hello>

<p>

Start editing to see some magic happen :) {{formattedDate}}

</p>

<p>{{send\_date|date:"dd-MMMM-yyyy hh:mm:ss a"}}</p>

var a=2;

function b(){

var x= 3;

}

console.log(a);//2

console.log(window.a);//2

console.log(this.a);//2

console.log(x);//error

console.log(window.x);//undefined

1. Shortest Program in JS: Empty file. Still, browsers make global EC and global space along with Window object.

2. Global Space: Anything that is not in a function, is in the global space.

3. Variables present in a global space can be accessed by a "window" object. (like window.a)

4. In global space, (this === window)//true object.

• window object is created by the JS engines of the respective browsers when global execution context is created.

• whenever an execution context is created a "this" keyword is also created.

• at the global level "this" points to the global object( window object in case of browsers).

• anything that is not inside a function is the "global space".

• whenever we create any variables or functions in the "global space", they get attached to the global object( window object in case of browsers).

so to access the variables/function defined in the global space ,

we can use any of the below:

console.log(window.a);

console.log(a);

console.log(this.a) //at the global space level, where this points to the window object

Undefined: a variable is 'declared', it has its own placeholder but not having the value of itself 'defined' hence

'undefined' and until the variable has assigned a value, the 'undefined' fills that particular placeholder

and 'undefined' is itself a datatype

(really sorry for referencing with c++ concepts)

Not Defined: This case comes in error where js engine neither find that particular variable nor its placeholder

and cannot find the variable in 1st phase of context (Memory allocation context)

and yes "flexibly typed language" sound pretty good instead of "weakly typed language"

Scope:

1. Scope of a variable is directly dependent on the lexical environment.

2. Whenever an execution context is created, a lexical environment is created. Lexical environment is the local memory along with the lexical environment

of its parent. Lexical as a term means in hierarchy or in sequence.

3. Having the reference of parent's lexical environment means, the child or the local function can access all the variables and functions

defined in the memory space of its lexical parent.

4. The JS engine first searches for a variable in the current local memory space, if its not found here it searches for the variable in the lexical

environment of its parent, and if its still not found, then it searches that variable in the subsequent lexical environments, and the sequence goes

on until the variable is found in some lexical environment or the lexical environment becomes NULL.

5. The mechanism of searching variables in the subsequent lexical environments is known as Scope Chain. If a variable is not found anywhere, then we say that

the variable is not present in the scope chain.

function a(){

var b= 10;

c();

function c(){

console.log(b);

}

}

a();

so many things here. I would like to sum up in notes, Scope:

-> let and const are hoisted. we cant use them before initialization is result of "temporal dead zone".

-> js use diff memory than global execution context to store let and cost. which is reason behind "temporal dead zone"

-> level of strictness ... var<<let<<const.

-> var //no temporal dead zone, can redeclare and re-initialize, stored in GES

let //use TDZ, can't re-declare, can re-initialize, stored in separate memory

const //use TDZ, can't re-declare, can't re-initialize, stored in separate memory

-> syntax error is similar to compile error. while type and reference error falls under run time error.

-> syntax error ... violation of JS syntax

type error ... while trying to re-initialize const variable

reference error ... while trying to access variable which is not there in global memor

Things learned: Block:

1. Code inside curly bracket is called block.

2. Multiple statements are grouped inside a block so it can be written where JS expects single statements like in if, else, loop, function etc.

3. Block values are stored inside separate memory than global. They are stored in block. (the reason let and const are called block scope)

4. Shadowing of variables using var, let and const.

5. The shadow should not cross the scope of original otherwise it will give error.

6. shadowing let with var is illegal shadowing and gives error.

7. var value is stored in nearest outer function or global scope and hence can be accessed outside block as well whereas same is not the case with let and const.

Closure with setTimeout():

Things learned:

1. setTimeout stores the function in a different place and attached a timer to it, when the timer is finished it rejoins the call stack and executed.

2. Without closure the var reference gives the latest value as it does not retain the original value but rather has the reference so any update in value after timeout will be shown.

3. If we use let/const because they have block scope, every time a new copy of variable is attached, thus this can be done without closure.

function x() {

for (var i = 1; i <= 5; i++) {

setTimeout(() => {

console.log(i);//6 will be 5 times

}, i\*1000);

}

console.log("Namaste");

}

x();

//letfunction x() {

for (let i = 1; i <= 5; i++) {

setTimeout(() => {

console.log(i);//1 2 3 4 5

}, i\*1000);

}

console.log("Namaste");

}

x();

function x() {

for (var i = 1; i <= 5; i++) {

function close(x) {

setTimeout(() => {

console.log(x);//1 2 3 4 5

}, x \* 1000);

}

close(i);

}

console.log("Namaste");

}

x();

Things learned:

1. An inner function can be directly called using two parenthesis ()().

2. Even parameters can be passed this way (Remember that the function needs to be returned to do this)

3. Closures can also be used for data hiding and encapsulation. So other code cannot access this value.

4. Unused variables are automatically deleted in High Level Programming language by garbage collector. Closures allocate a

lot of memory which cannot be deleted so this acts as a disadvantage.

5. Some browsers now have smart garbage collectors that automatically deletes variables that are not used outside closures.

function outest(){

var c= 20

function outer(b){

function inner(){

console.log(a,b,c);

}

let a=10;

return inner;

}

return outer;

}

var z=outest()(8);

z();//10 8 20

Function()

Things learned:

1. What is Function Statement ?

A. A normal function that we create using Naming convention. & By this we can do the Hoisting.

For Ex - function xyz(){

console.log("Function Statement");

}

2. What is Function Expression ?

A. When we assign a function into a variable that is Function Expression. & We can not do Hoisting by this becz it acts like variable.

For Ex - var a = function(){

console.log("Function Expression");

}

3. What is Anonymous Function ?

A. A Function without the name is known as Anonymous Function. & It is used in a place where function are treated as value.

For Ex - function(){

}

4. What is Named Function Expression ?

A. A function with a name is known as Named Function Expression.

For Ex - var a = function xyx(){

console.log("Names Function Expression");

}

5. Difference b/w Parameters and Arguments ?

A. When we creating a function & put some variabels in this ( ) that is our Parameters.

For Ex - function ab( param1, param2 ){

console.log("

}

& When we call this function & pass a variabel in this ( ) that is our Arguments

For Ex - ab( 4, 5 );

6. What is First Class Function Or First class citizens?

A. The Ability of use function as value,

\* Can be passed as an Argument,

\* Can be executed inside a closured function &

\* Can be taken as return form.

For Ex - var b = function(param){

return function xyz(){

console.log(" F C F ");

}

}

7. Function are heart of JS. They are called first class citizens or first class functions because they have the ability to be stored in the variables, passed as parameters and arguments. They can also be returned in the function.

In programming languages, a first-class function, also known as a first-class citizen or first-class object, refers to a function that has the following properties:

// Functions assigned to variables

const add = function (a, b) {

return a + b;

};

// Function passed as an argument to another function

const multiply = function (a, b, operation) {

return operation(a, b);

};

// Function returned as a value from another function

const createMultiplier = function (factor) {

return function (number) {

return number \* factor;

};

};

// Functions stored in an array

const functionArray = [add, createMultiplier(2)];

console.log(add(2, 3)); // Output: 5

const result = multiply(4, 5, add);

console.log(result); // Output: 9

const double = createMultiplier(2);

console.log(double(7)); // Output: 14

console.log(functionArray[1](5)); // Output: 10

It seems like you might be referring to a "temporal dead zone" in JavaScript. The temporal dead zone (TDZ) is a specific behavior related to the hoisting of variables declared with let and const in ECMAScript 6 (ES6). In JavaScript, variable declarations are hoisted to the top of their containing scope, but the variables are not initialized until the actual declaration statement is encountered during runtime.

Here's an example to illustrate the temporal dead zone:

javascript

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

let x = 5; // Initialization and declaration are in the same statement

In the above example, even though x is declared using let, trying to access it before the declaration statement results in a ReferenceError. This is because the variable is in the "temporal dead zone" until the interpreter reaches the let x = 5; statement during the execution.

To avoid issues related to the temporal dead zone, it's a good practice to declare variables at the top of their scope before using them.

javascript

let x; // Declaration

console.log(x); // undefined

x = 5; // Initialization

console.log(x); // 5

This way, the variable is hoisted to the top of the scope during the hoisting phase, and even though it's undefined until the assignment statement is encountered, there won't be a reference error when trying to access it.

Callback()

Things learned:

1. Function that is passed on as argument to another function is called callback function.

2. setTimeout helps turn JS which is sinhlethreaded and synchronous into asynchronous.

3. Event listeners can also invoke closures with scope.

4. Event listeners consume a lot of memory which can potentially slow down the website therefore it is good practice to remove if it is not used.

In the context of JavaScript, the term "Web API" (Application Programming Interface) refers to a collection of interfaces that allow developers to interact with specific features or data on the web. These APIs are provided by web browsers to enable client-side web development. Web APIs expose a set of methods and properties that developers can use to perform various tasks, such as manipulating the DOM (Document Object Model), making HTTP requests, handling events, and more.

Here are some common Web APIs in JavaScript:

DOM (Document Object Model) API:

The DOM API provides a way for JavaScript to interact with the HTML and XML documents dynamically. Developers can use methods and properties to manipulate the structure, content, and style of a web page.

Things learned:

1. Browser has superpowers that are lent to JS engine to execute some tasks, these superpowers include web API's such as console, location, DOM API, setTimeout, fetch, local storage.

2. Callback functions and event handers are first stored in Web API environment and then transferred to callback queue.

3. Promises and mutation observer are stored in API environment and then transferred to microtask queue.

4. Event loop continuously observes call stack and when it is empty it transfers task to call stack.

5. Micro task is given priority over callback tasks.

6. Too many micro tasks generated can cause Starvation (nit giving time to callback tasks to execute).

AOT/JIT:

By default, when you run an Angular project using the Angular CLI, the project uses Just-In-Time (JIT) compilation. JIT compilation is the default compilation

mode during development because it allows for faster development cycles, easier debugging, and immediate reflection of changes to templates and components.

When you build your Angular project for production, the default behavior is to use Ahead-of-Time (AOT) compilation. AOT provides performance optimizations,

smaller bundle sizes, and better runtime performance.

To explicitly specify AOT compilation during development (e.g., for testing or specific scenarios), you can use the --aot flag with the ng serve command:

ng serve --aot

For production builds, the Angular CLI automatically uses AOT by default when you use the --prod flag with the ng build or ng serve command:

ng build --prod

or

ng serve --prod

In summary:

Development: By default, JIT compilation is used for faster development cycles.

Production: AOT compilation is used by default for production builds to achieve performance optimizations and smaller bundle sizes.

You can explicitly use the --aot flag during development if needed, and it's common to use a combination of JIT and AOT in different stages of the development lifecycle.

AOT (Ahead-of-Time) Compilation:

When: AOT compilation is performed during the build process before deploying the application.

What: Templates and components are compiled to JavaScript and HTML during build time.

Benefits:

Faster loading times in production due to precompiled templates.

Smaller bundle sizes as the Angular compiler is not needed in the final bundle.

Detects template errors at build time.

Drawbacks:

Longer build times.

JIT (Just-In-Time) Compilation:

When: JIT compilation is performed at runtime in the client's browser.

What: Templates and components are compiled to JavaScript and HTML on-the-fly as the application runs.

Benefits:

Faster development workflow with immediate reflection of changes.

Easier debugging with source code closely matching the running application.

Drawbacks:

Slower initial loading times in production due to runtime compilation.

Larger bundle sizes as the Angular compiler is included in the final bundle.

JS Exposed:

Things learned:

1. JS runtime environment contains all elements required to run JS.

2. It contains JS engine, set of API's, callback queue, microtask queue, event loop.

3. JS engine is a piece of code.

4. Process includes Parsing ---> Compilation -----> Execution.

5. Parsing breaks code into tokens and converts it into AST(Abstract Syntax Tree).

6. Modern JS engine follows JIT compilation, it interprets while it optimises code as much as it can.

7. Execution and Compilation are done together.

8. Execution has Garbage collector and other optimisation such as inlining, copy elusion, inline caching etc.

9. Fell in love with JS. ❤❤

Basic idea about Mark & Sweep Algo:

It comprise of 2 phases -

1. Mark phase

2. Sweep phase

Garbage Collector:

All objects are marked as 0 initially (at creation) and in mark phase the objects that will be accessible are marked as 1 (reachable) by a DFS graph traversal.

During sweep phase, the objects marked with 0 are removed from heap memory. and also all reachable objects are again initialized with 0 (made unreachable) because the algorithm will run again.

SetTimeout() Trust issue:

Things learned:

1. The setTimeout function stores it in the callback queue which is executed only after call stack is empty, even if setTimeout is set to 0ms.

2. setTimeout ensures that minimum it will take the time mentioned because it may be paused due to call stack not empty.

Namaste2:

Issue with calll back

Callbacks are the functions passed to another function as an argument and it helps to write asynchronous operation in JS.

When we are using callbacks, we face two issues:

1. Callback hell - When we have multiple dependent callbacks it will form nested callbacks, code will be unmaintainable and unreadable. This nested callback structure is also called Pyramid of Doom.

2. Inversion of control: We pass the callback function within another function, we are blindly trusting where we don't know whether that function will ever execute our callback function or not, it's so risky for a developer. We loose the control of our program.

Promise:

1. Before promise we used to depend on callback functions which would result in 1.) Callback Hell (Pyramid of doom) | 2.) Inversion of control

2. Inversion of control is overcome by using promise.

2.1) A promise is an object that represents eventual completion/failure of an asynchronous operation.

2.2) A promise has 3 states: pending | fulfilled | rejected.

2.3) As soon as promise is fulfilled/rejected => It updates the empty object which is assigned undefined in pending state.

2.4) A promise resolves only once and it is immutable.

2.5) Using .then() we can control when we call the cb(callback) function.

3. To avoid callback hell (Pyramid of doom) => We use promise chaining. This way our code expands vertically instead of horizontally. Chaining is done using '.then()'

4. A very common mistake that developers do is not returning a value during chaining of promises. Always remember to return a value. This returned value will be used by the next .then()

The Promise API refers to the set of features and methods in JavaScript that allow developers to work with asynchronous operations more easily. Promises were introduced in ECMAScript 2015 (ES6) to address the callback hell problem and make asynchronous code more readable and manageable.

A Promise is an object that represents the eventual completion or failure of an asynchronous operation and its resulting value. It has three states:

Pending: The initial state; the promise is neither fulfilled nor rejected.

Fulfilled: The operation completed successfully, and the promise has a resulting value.

Rejected: The operation failed, and the promise has a reason for the failure.

The Promise API includes the following key methods:

Promise constructor: Used to create a new Promise object. It takes a function as an argument, which in turn takes two parameters: resolve and reject. The resolve function is used to fulfill the promise with a value, and the reject function is used to reject the promise with a reason.

javascript

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

// Asynchronous operation

// If successful, call resolve with a value

// If there's an error, call reject with a reason

});

then method: Attaches callbacks for the fulfillment and/or rejection of the promise. It takes two optional callback functions as arguments, the first for fulfillment (onFulfilled) and the second for rejection (onRejected).

javascript

myPromise.then(

(value) => {

// Handle fulfillment

},

(reason) => {

// Handle rejection

}

);

catch method: A shorthand for handling only the rejection of a promise. It is equivalent to calling then(undefined, onRejected).

javascript

myPromise.catch((reason) => {

// Handle rejection

});

finally method: Attaches a callback that is executed regardless of whether the promise is fulfilled or rejected.

javascript

myPromise.finally(() => {

// This callback will be executed no matter what

});

The Promise API has become an integral part of modern JavaScript, and it provides a cleaner and more structured way to work with asynchronous code compared to traditional callback patterns. It has since been extended and enhanced with features like async/await in later ECMAScript versions.

The main topic of this video is Promise APIs.

02:21 promise.all takes an array of promises as input and makes parallel API calls.

07:13 If any promise in the Promise.all array gets rejected, the Promise.all will throw an error, and the same error will be the output

09:48 The Promise APIs all and allSettled handle the success and failure of multiple promises.

15:00 Promise APIs - all, allSettled, race, any

17:31 In a race between multiple promises, the first promise to settle (either resolve or reject) will determine the result.

22:07 Promise.any waits for the first successful promise

24:38 If all promises fail, an aggregated error will be returned.

28:58 Promise APIs + Interview Questions: Explaining the functionalities of allSettled, all, race, and any

31:03 Promise.all, Promise.allSettled, Promise.race, Promise.any

35:14 Demonstration of using Promise APIs in JavaScript

37:29 Understanding Promise.all and handling errors

41:40 The promise.allSettled() method waits for all the promises to fulfill, collects all the results, and returns them.

43:41 Promise object gives an array of objects with status and value for success or rejection.

47:52 Understanding the lingo of promises: fulfilled, rejected, settle, success, failure

49:47 Promise.any waits for the first settled promise and ignores the rest

53:54 The aggregate errors in Promise APIs are returned as an array.

55:57 The speaker is requesting viewers to share and appreciate the video.

This keyword:

If you are inside function,the value depend on strict and non strict mode.

"use strict"

function x(){

console.log(this);//undefined

}

x();

window.x();//in strict mode also window

//unstrict mode output is window

when we create function inside object called method.

e.g- const obj1={

a:10,

x:function(){

}

};

here x is method of obj1

Debouncing&Throttling::

Debouncing delays the execution of your code until the user stops performing a certain action for a specified amount of time.

Throttling limits the execution of your code to once in every specified time interval.

Debouncing and throttling are two techniques used in JavaScript to control the rate at which a function is executed, especially in scenarios like handling user

input or window resize events. These techniques help improve performance and prevent excessive function calls. Here's a brief explanation of each:

from JS engine's perspective:

debouncing: first you finish your action (typing, clicking) & then after x milliseconds I will fetch the required data

throttling: I am working on something which might take x milliseconds, so don't disturb me for next x milliseconds

Async&Defer:

async and defer are attributes used with the <script> tag in HTML to control the loading and execution of external JavaScript files. They are particularly useful when you want to manage the order in which scripts are loaded and executed on a web page.

async:

When you add the async attribute to a <script> tag, it means that the script will be downloaded asynchronously while the HTML parsing continues.

The script will be executed as soon as it is downloaded, regardless of whether the HTML parsing is complete.

This is beneficial for scripts that are independent and do not rely on the order of execution.

Example:

<script async src="script.js"></script>

defer:

When you add the defer attribute to a <script> tag, it means that the script will be downloaded asynchronously, but it will only be executed after the HTML parsing is complete.

Scripts with the defer attribute are executed in the order they appear in the HTML document.

This is useful when scripts depend on the DOM structure or other scripts.

Example:

<script defer src="script.js"></script>

Usage Example:

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Async and Defer Example</title>

<!-- Scripts with async and defer attributes -->

<script async src="script1.js"></script>

<script defer src="script2.js"></script>

</head>

<body>

<!-- Content of the web page -->

</body>

</html>

In this example, script1.js will be downloaded and executed asynchronously, while script2.js will be downloaded asynchronously but will only be executed after the HTML parsing is complete.

If you do not use either async or defer:

The script will be fetched and executed immediately, blocking the parsing of the HTML document until the script is downloaded and executed.

The browser will wait for the script to be fully downloaded and executed before continuing to parse and render the rest of the HTML.

The absence of both async and defer attributes is equivalent to not using either, and it results in the default behavior where the script is loaded and executed synchronously.

Event Bubbling:

Definition: In the event bubbling phase, the innermost (target) element's event is handled first, and then the event bubbles up through its ancestors in the DOM hierarchy.

Order: The event starts from the target element and bubbles up to the root of the document (window).

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

// This function will be called during the bubbling phase

});

Event Capturing (Trickling):

Definition: In the event capturing phase, the outermost (root) element's event is handled first, and then the event trickles down through its descendants in the DOM hierarchy until it reaches the target element.

Order: The event starts from the root and trickles down to the target element.

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

// This function will be called during the capturing phase

}, true); // The third parameter (true) enables event capturing

Event delegation is a programming pattern in web development where instead of attaching an event handler to individual elements, you attach a single event handler to a

common ancestor. This single handler then monitors for events that bubble up from the descendant elements. This approach is particularly useful when you have a large number of similar elements that need the same type of event handling.

//By rethinking UI

const first = "JavaScript";

const middle = "Frontend";

const last = "developer";

//3 ways to concatenate string

console.log(first+" "+middle+" "+last);

console.log([first,middle,last].join(" "));

console.log(`${first} ${middle} ${last}`);

npx kill-port portnumbe//want to delete particular port

npx npkill//to kill node modules

//JSON is a format by which will exchange data to and from web server.

const txt = '{"name":"Bubu","age":27}';

const obj = JSON.parse(txt);//when receiving data from server in string,we need to convert into object

//sending data to server ,need to send as string

var text= JSON.stringify(obj);

//Dot notation and bracket notation

const obj = {

name:"Apurba",

age:27

};

console.log(obj.name);//dot notation

console.log(obj["name"]);//bracket notation

console.log(obj['name']);//bracket notation

//Limitation of dot notation

//if we have invalid identifier in obj go with bracket notation

const obj1 = {

123:'digit',

'123name':'hi hwllo',

$name:'Bubu'

};

console.log(obj1.123);//error but

console.log(obj1["123"]);//bracket notation helps alot

const vari = 'name';

const obj2 = {

name:'Apurba',

age:27

};

console.log(obj2[vari]);//Apurba

console.log(obj2.vari);//undefined

//Annonymus function

let print= function(){

console.log("I am Annonymus");

};

print();

setTimeout(function() {

console.log("I am Annonymus 2");

}, 2000);

//self executive function

//An Immediately Invoked Function ( known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.

//Syntax of IIFE :

(function(){

// Do something;

})();

(function(){

console.log("I am iif")

})();

//splitting

const list = "apples,banana;cheer;orange";

const fruits= list.split(",");//splitting with single

console.log(fruits);

const multifruits= list.split(/[,;]/);//splitting with multiple

console.log(multifruits);

//To refresh browser

<meta http-equiv="refresh" content="5;URL=https://mail.google.com/mail/u/0/#inbox" />

const arr = [1,2,"",false,undefined,NaN,5,8];

const out = arr.filter(Boolean);//To remove falsy value we used filter with boolean

console.log(out);

console.log("b"+"a"+ +"a"+"a");//baNaNa

console.log(Number.MIN\_VALUE>0);//minmum value is not 0 or not -ve,it's smallest positive number

console.log(Math.min());//infinity

console.log(Math.max())//-infinity

console.log(1+2+"Hello"+[1,2,3]);//3Hello1,2,3

console.log([1,2]+[3,4]);//1,23,4

//An array with no keys is falsy values

//0,null,undefined,false,NaN,""

var emptyArray=[];//empty array

if(emptyArray) console.log("hello");

console.log(emptyArray==false)//true ,value of empty array is 0

console.log(emptyArray===false)//false,type of empty array is object

<a href="javascript:void(0)">bubu</a>//i am executing but i am returning undefined

<a href="https://www.w3schools.com/jsref/jsref\_min\_value.asp#:~:text=MIN\_VALUE%20is%20the%20value%20closest,this%20are%20converted%20to%200."

target="\_blank" rel="noopener">bubu</a></body>//rel=“noopener noreferrer” are actually two separate tags with different uses. Noreferrer stops linked website owners

from seeing the origin of traffic you send to their website, while the noopener tab plays a crucial role in protecting your site from reverse tabnapping.

//onpaste="return false " or ondrop="return false "//used to avoid copy paste

//to give border color

input:in-range{

border:2px solid yellow

}

var x=1;

a();

b();

console.log(x);

function a(){

var x = 10;

console.log(x);

}

function b(){

var x = 100;

console.log(x);

}

//output10

100

1

AI:

1. https://chat.openai.com/

2. https://gamma.app/?lng=en --For presentation 40 free in 1 month

3. https://app.yoodli.ai/ --To practice interview

4. https://12ft.io/proxy - read newspaper by inserting link

5. https://chromewebstore.google.com/detail/merlin-1-click-access-to/camppjleccjaphfdbohjdohecfnoikec add it in chrome ,give it content it will give in less line,even it will read video and explain you in text

6. https://bard.google.com/ -For coding

7. https://formula.dog/ -- To write formula

8. signalhire.com - for details

9. debugcode.ai - for debbing code