JavaScript CONCEPTS

Mastering JavaScript Interview Insights and Essentials

INTERVIEW HANDBOOK

@UDBHAV SRIVASTAVA

WHAT IS JAVASCRIPT ?

JavaScript, a synchronous single-threaded language, is a cornerstone in web development, empowering the creation of interactive and dynamic websites. Its versatility and efficiency make it a vital tool for crafting engaging online experiences.

Now,what is this synchronous single-threaded means ?

SYNCHRONOUS: In a synchronous programming language, code is executed line by line in a specific order.

Single-Threaded : JavaScript is single-threaded, means Js will excute one command at a time.

swipe >>

HOW EXECUTION CONTEXT WORKS ?

Everything in Javascript happens inside an Execution context.

Memory Creation Phase	Code Execution Phase
Phase 1 Key : Value var n : undefind	Phase 2 after code execution phase n will gonna assign to its value that is 10 n = 10 Here code executed one line at a time
fun xyz(){ here we copy the whole code } ⇢	Memory Creation Phase / Code Execution Phase Key : Value var x : undefind fun abc(){ }

after code execution phase x will gonna assign to its value that is 20

x = 20

Memory Creation Phase | Code Execution Phase

Key : Value

var x : 20

fun abc(){ }

In JavaScript, think of phase 1 as a storage space holding variables and functions like keys and values. The phase2 is where these keys and values come to life, executing and making things happen in your program.

swipe >>

Call Stack

It is a data structure that keeps track of the execution context of a program. In simpler terms, the call stack is like a stack of function calls that are currently being executed. Maintains the order of execution context.

Hoisting

Hositing is a phenomenon of Javascript by which we can access **var & Functions** even **before it's initialization/**put some value in it.

How it Work's

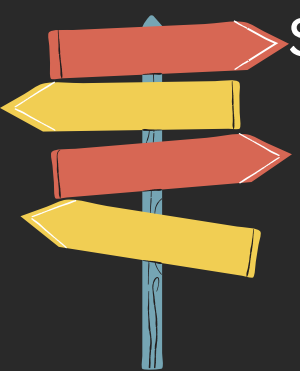
When variables and functions are declared, they are effectively "hoisted" to the top of their respective scopes within the memory, making them accessible throughout the entire scope even before their explicit placement in the code.
So when there are in Phase 1 there are hoisted.

swipe >>

Why we place javascript files at the end of the <body> tag ?

It’s a common practice, and it's often recommended for a couple of reasons:

- When a browser encounters a <script> tag, it stops parsing HTML and executes the script.
- Placing scripts at the end of the **<body>** allows the HTML content to load and render first
- Browsers typically download external resources, including JavaScript files, in parallel. If scripts are placed at the end of the <body>, the browser can start fetching them after the HTML content is already being displayed.
- **Async Attribute:** Adding the **async** attribute to a **<script>** tag allows the browser to download the script asynchronously while not blocking the HTML parsing



swipe >>

UNDEFINED

Undefined is distinct from being empty; rather, it is a special keyword that holds its own memory space. Consider it as a temporary **placeholder**, reserved until the variable is assigned a different value. Until that assignment occurs, the variable remains in this "placeholder" state known as undefined.

Diffrence Between Undefind & Null

Undefined means a variable has been declared but has yet not been assigned a value. Null is an assignment value. It can be assigned to a variable as a representation of no value.

Lexical Enviroment

The lexical environment is created during the memory creation phase 1 of the execution context. when ever an execution context is created a lexical enviroment is also created. So, the lexical enviroment is the local memory along with the lexical enviroment with his parent.

swipe >>

CLOSURES

A closure in JavaScript is formed when a function, along with its lexical scope, is bundled together. This unique combination provides the inner function with access to the variables and parameters of its outer function, even after the outer function has finished executing

USE'S OF CLOSURE

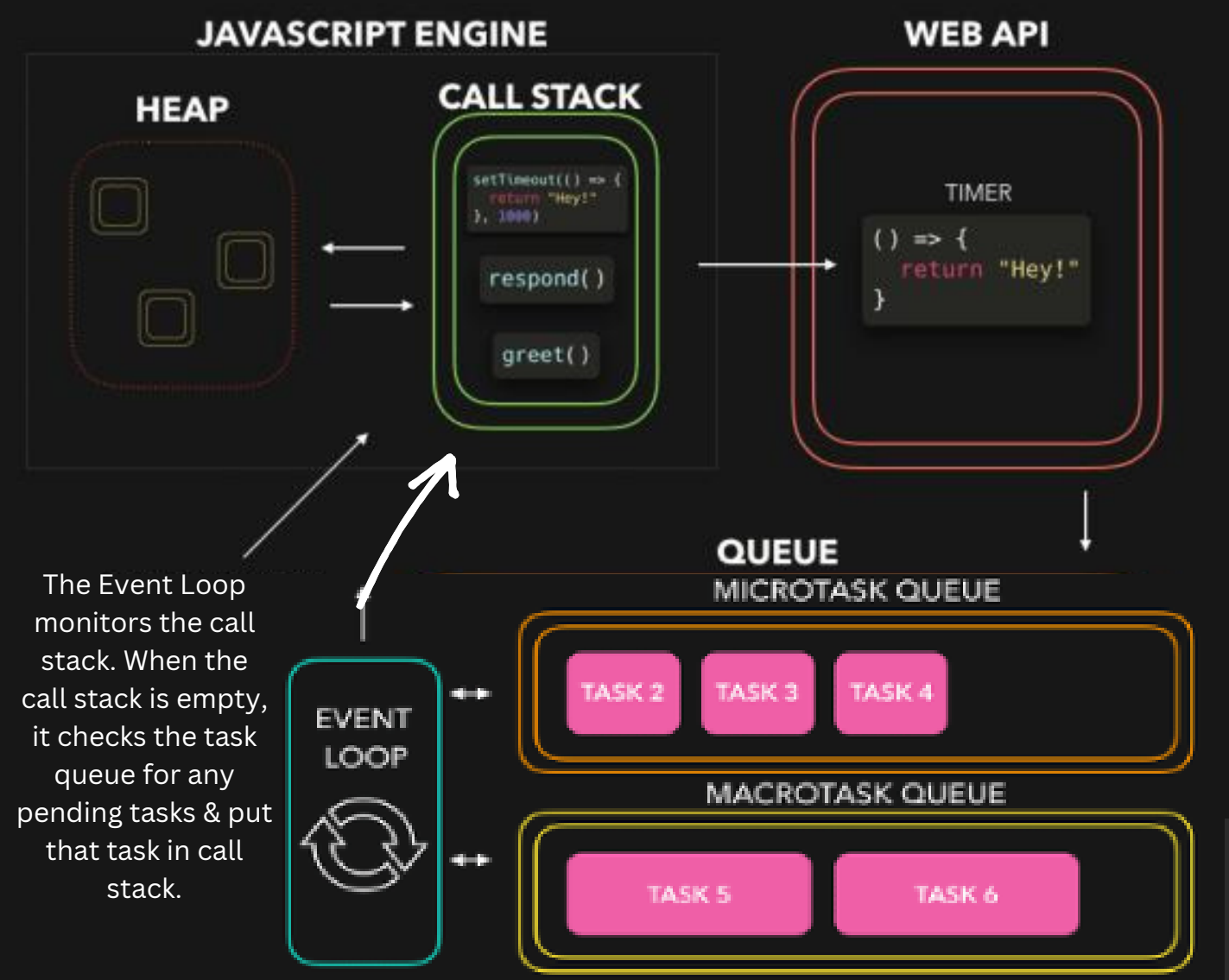
Closures are powerful mechanisms for

- Creating private variables
- Module design patterns
- Memorize
- setTimeOut
- Maintaining state in Async world

swipe >>

EVENT LOOP

In JavaScript, numerous operations, like timers and network calls, operate asynchronously. The Event Loop plays a crucial role in managing these asynchronous tasks efficiently. It ensures that such operations are handled without blocking the execution of other code, allowing JavaScript to exhibit asynchronous behavior seamlessly.




```
function greet() {
  return "Hello!"
}

function respond() {
  return setTimeout(() => {
    return "Hey!"
  }, 1000))
}

greet()
respond()
```

The Event Loop monitors the call stack. When the call stack is empty, it checks the task queue for any pending tasks & put that task in call stack.

swipe >>

BLOCK & SCOPE

A block is used to combine multiple Javascript statement into one, So that which can use multiple statment in a place where javascirpt expects one statement.
A block defines a scope, which is the region of code where variables and functions are accessible.

There are typically two types of scope:
Global scope & **Local scope**

FUNCTION DECLARATION & FUNCTION EXPRESSION

Function Declaration:

Hoisted to the top of their scope.
Defined with the function keyword and a name

Function Expression:

Not hoisted in the same way as declarations.
Creates an anonymous function assigned to a variable.

The primary difference lies in hoisting behavior and whether the function has a name. Function declarations are hoisted along with their names, while function expressions (whether anonymous or named) are not hoisted in the same way.

swipe >>

DIFFERENCE BETWEEN PARAMETR OR ARGUMENT

```
function (Param , Param){

};
function (Args,Args);
```

The value which we will be pass **inside** a function are known as Argument.

These **lable and identifier** known as Parameter which get those value in the function.

HIGHER ORDER FUNCTION & FIRST CLASS FUNCTION

A function is **First-class** if it can be treated like any other **value**

A function that takes another function as **arguments** or **returns a function** from it known as a **Higher-Order function.**

CALL BACK FUNCTION

A callback function is a function that is passed as an argument to another function and is intended to be invoked or executed at a later time, often in response to a specific event or condition.

swipe >>

(...)

(...) this can be Rest parameter or Spread operator.

Rest parameter

if it is in function Arguments then it is Rest parameter it will combing the statement else everywhere in js it is spread operator.

Spread operator

Spread operator it will create another copy of that object. And it will go to new memory

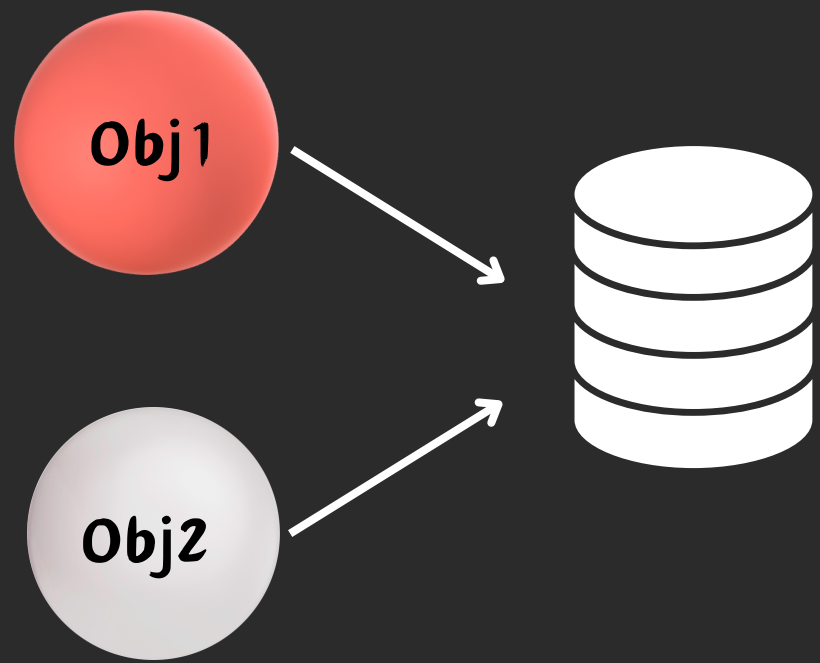
some common use cases of the spread operator :

- **Copying Arrays**
- **Concatenating Arrays**
- **Passing Function Arguments**
- **Creating Array Copies with Modifications**
- **Spreading Object Properties**
- **String to Array**

swipe >>

Shallow copy

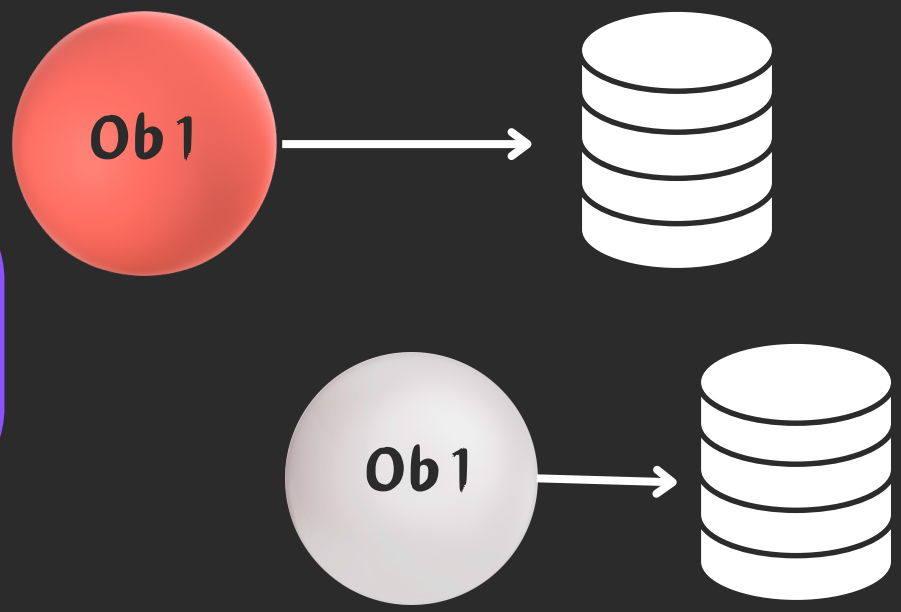
when the cloned object is having reference to old object
then it’s called as shallow copy



Deep Clone

In Deep Clone obj2 should not have any refernce of obj1.
It‘s a completely independent copy of an object .

```
const obj2 = JSON.parse(
JSON.stringify(obj1)
);
```



swipe >>

Polyfill of Deep Clone

```
const deepClone = (obj) => {
    const type = typeof obj;
if (type !== 'object' || !obj) return obj;

    if(Array.isArray(obj)){

return obj.map(item => deepClone(item));
}
let arrObj = Object.entries(obj);

let deepCloneArrOfObj = arrObj.map(([key, value]) => [key,
deepClone(value)]);

let finalObj = Object.fromEntries(deepCloneArrOfObj);
return finalObj;
}
```

Checking the type of parameter

If parameter is not object return it

obj is an array

Deepcloning value if it is in nested obj

returing deepCloned version of obj

Function Barrowing :

In function Barrowing we Barrow a function from some other object and use it with the data of some other object.

swipe >>

THIS KEYWORD

In JavaScript, the this keyword refers to the current execution context, and its value is determined by how a function is called. The behavior of this can vary depending on whether the function is called in the global scope, as a method of an object, with the new keyword, or using other methods.

EXPLAIN CALL(), APPLY() & BIND() METHODS.

In JavaScript, the call, apply, and bind methods are used to manipulate the **this** value within a function. They provide a way to explicitly set the context in which a function is executed.

Call

Use any fuction which is present in another object.which explicitly binds functions to any given object

```
function sayHello(){ return "Hello " + this.name;}
var obj = {name: "Sandy"};
sayHello.call(obj);
```

swipe >>

Apply ()

The apply method is similar to call, but it takes an array-like object as its second argument for passing arguments to the function.

```
function sayHello(greeting, punctuation) {
console.log(`${greeting}, ${this.name}${punctuation}`);
}
const person = { name: 'John' };
sayHello.apply(person, ['Hello', '!']);
```

Bind ()

This method returns a new function, where the value of “this” keyword will be bound to the owner object, which is provided as a parameter.

Bind method it actually gives us the copy of the method and it does not invoke it directly

swipe >>

Explain Map (), Filter () & Reduce ()

Map Method ()

A method which runs the given fucntion over each and every value of given array and create new array out of it.

```
// Example: Doubling each element in an array
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(function (num) {
  return num * 2;
});

console.log(doubledNumbers);
```

Filter Method ()

Fliter method creates a new array but selecting elements from an exisiting array that meets given condition. It essentially acts as a filter,

```
// Example: Selecting only even numbers from an
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = numbers.filter(function (num
  return num % 2 === 0;
});

console.log(evenNumbers);
```

Reduce Method ()

It’s technically used at place where we need to iterate over all the elements of an Array and find out particular value

```
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce(
  function (accumulator, current) {
    return accumulator + current;
  },
  0
); // 0 is the initial value

console.log(sum);
```

swipe >>

Polyfills of Map , Filter & Reduce

Polyfill of Map

Map method expects a call back function from user

```
Array.prototype.myMap = function(callback) {
  let newArr = [];

  for (let i = 0; i < this.length; i++) {
    newArr.push(callback(this[i], i, this));
  }

  return newArr;
};
```

For each element, it calls the callback function with the element and index as arguments and pushes the result into the newArr.

Polyfill of Filter

if my callback is giving true for this[i]

if callback(ele1) is true or not
true -> [ele1]
else -> []

```
Array.prototype.myFilter = function(callback) {
  let newArray = [];

  for (let i = 0; i < this.length; i++) {
    if (callback(this[i], i, this)) {
      newArray.push(this[i]);
    }
  }

  return newArray;
};
```

swipe >>

Polyfill of Reduce

```
Array.prototype.reduce = function(callback, initialValue) {
    let accumulator = initialValue;

    for (let i = 0; i < this.length; i++) {
        if (accumulator === undefined) {
            accumulator = this[i];
        } else {
            accumulator = callback(accumulator, this[i], i);
        }
    }

    return accumulator;
```

- callback is the function that will be called for each element in the array, as well as the accumulator.
- InitialValue is an optional parameter. If provided, it is used as the initial value of the accumulator. If not provided, the first element of the array is used as the initial value.
- If the accumulator is not undefined, it calls the callback function with the current accumulator, the current element (this[i]), and the index (i). The result becomes the new value of the accumulator.

swipe >>

OBJECTS

Any value that's not of a primitive type (a string, a number, a boolean, a symbol, null, or undefined) is an object. Objects are always **passed by reference.**

OBJECT CLONING

Here we are creating a copy of an object.
There are three main types of object cloning:

ASSIGNING

It does not create a new object; instead, both variables point to the same object in memory.

```
let oldObj = { key: 'value' };

let newObj = oldObj;

console.log(newObj === oldObj);
```

SHALLOW CLONE:

```
const originalObject = { key1: 'value1', key2: { nestedKey: 'nestedValu

const shallowClone = { ...originalObject };
```

A shallow clone creates a new object, but it does not create copies of nested objects within the original object. Instead, it copies references to the nested objects. This means that changes to the nested objects will be reflected in both the original and cloned objects.

swipe >>

DEEP CLONE:

```
const originalObject = { key1: 'value1', key2: { nestedKey: 'nestedVal

// Deep clone using JSON.stringify and JSON.parse
const deepClone = JSON.parse(JSON.stringify(originalObject));
```

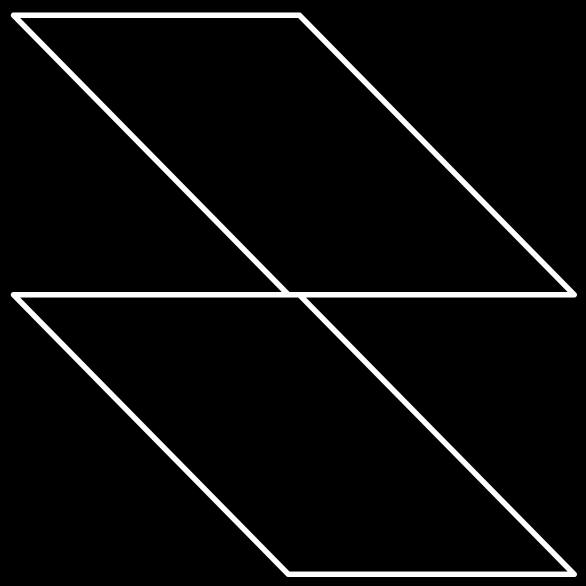
JSON. stringify

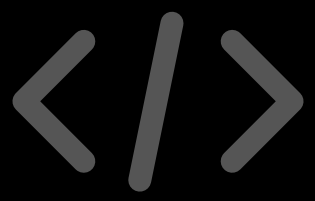
The JSON.stringify function converts a JavaScript object into a JSON string. It takes an object as input and returns a string representing that object in JSON format.

JSON.parse

The JSON.parse function does the opposite; it takes a JSON string as input and converts it into a JavaScript object.

Through Deep Cloning we can create a completely independent copy of the original object and all of its nested structures.





swipe >>

Promise

Promise is an Object representing the eventual completion or failure of an asynchronous operation.

```
const myPromise = new Promise((resolve, reject) => {
  // Simulate an asynchronous operation
  setTimeout(() => {
    const randomNumber = Math.random();

    if (randomNumber > 0.5) {
      resolve(randomNumber); // Fulfilled
    } else {
      reject("Operation failed"); // Rejected
    }
  }, 1000);
});

myPromise
  .then((result) => {
    console.log("Success:", result);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

The .then() method is used to handle the fulfillment of the promise, and .catch() is used to handle the rejection.

Promise is nothing but you can assume it to be an empty object. Empty object with some data value in it & this data value hold what ever the data comes.

There can be only three states in Promise

- **Pending** : Initial state, niether pending nore rejected.
- **Fullfilled** : meaning that operation was completed.
- **Rejected** : meaning that the operation is failed.

swipe >>

Async & Await

async and **await** are features in JavaScript that make working with asynchronous code more readable and easier to reason about.

async Function

- The async keyword is used to declare a function as asynchronous.
- An async function always returns a promise, and the value of the promise is the value returned by the function when it completes.

await Operator

- The await keyword is used inside an async function to wait for the resolution of a promise.
- It can only be used within an **async** function, and it pauses the execution of the function until the promise is resolved.

```
async function fetchData() {
  const result = await fetch('https://api.example.com/data');
  const data = await result.json();
  console.log(data);
}
```

In this example, fetch returns a promise, and await is used to wait for that promise to resolve. The second await is used to wait for the JSON parsing to complete.

Error Handling:

When using **async** and **await**, error handling can be done using **try** and **catch.**
So whenever If any of the promises being awaited rejects, the control jumps to the nearest catch block.

swipe >>

DEBOUNCING

Delays invoking a function until a certain period has passed without additional invocation.
Particularly useful in scenarios such as search bars. It helps limit the frequency of API calls by ensuring that the function is only triggered after a specified time period has elapsed since the last user input

Throttling

Throttling is like a traffic cop for function calls, making sure they don't happen too quickly basically managing the frequency of function calls, ensuring it’s executed at a controlled rate, particularly useful for performance optimization.
throttling will delay the function call for certain time basically

DEBOUNCING VS THROTTLING

- In debouncing we are ignoring last event according to given conditions.
- Wait for a moment of silence before doing something
- Preventing unnecessary API calls
- Focuses on ensuring a pause before action.

- In throttling we are ignoring upcoming event according to given conditions.
- Do it, but not too often—pace yourself!
- Preventing functions from being called too frequently.
- Focuses on controlling the rate of action.

swipe >>

Udbhav Srivastava

Thanks for reading! If you found value, give it a like, share, or drop a comment. Your suggestions are appreciated!

Let's Connect

udbhav-srivastava-a9305321b/