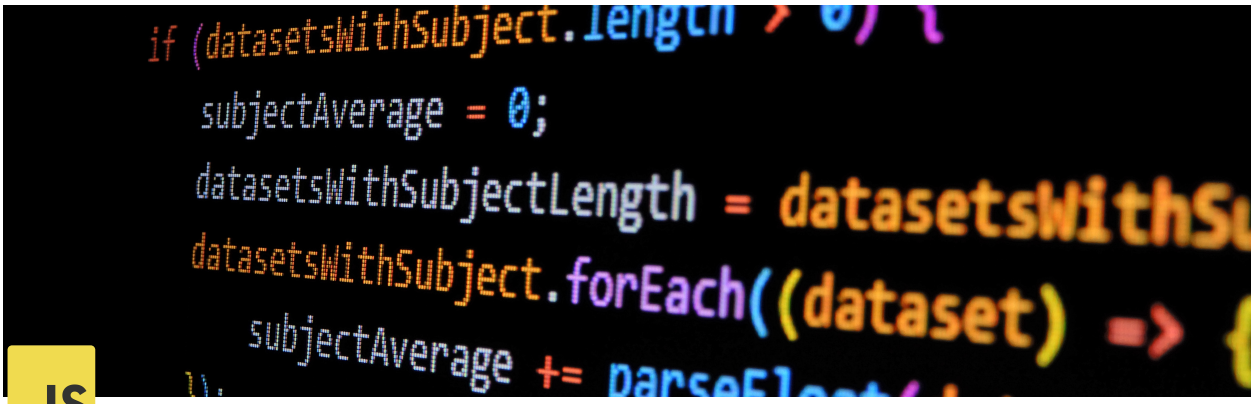

Javascript Questions

▼ What is prototype?

Prototype is a property on a function that is referencing an object

▼ What are closures? Advantage and Disadvantage

Closure is a combination of functions bundled together in a lexical scope. The inner function has access to an outer function scope, variables and parameters even after the outer function has returned.

7 Question - https://dmitripavlutin.com/javascript-closures-interview-questions/#questions-1-closures-raise-your-hand

```
function outerFunc() {
  let outerVar = 'I am outside!';
  return function innerFunc() {
    console.log(outerVar); // => logs "I am outside!"
  }
}


function exec() {
  const myInnerFunc = outerFunc();
  myInnerFunc();
}
exec();
```

Advantages of closures:

1. Callbacks implementation in javascript is heavily dependent on how closures work
2. Mostly used in Encapsulation of the code
3. Also used in creating API calling wrapper methods

Disadvantages of closures:

1. Variables used by closure will not be garbage collected
2. Memory snapshot of the application will be increased if closures are not used properly

▼ Difference between regular function vs arrow function?

1. Syntax - No implicit return for arrow function
2. Argument Binding - In regular function, Arguments keywords can be used to access the arguments of which passed to function.

```
function regularFunction(a,b) {
    console.log(arguments)
}
regularFunction(1,2)
// Arguments[1,2]
```

```
const arrowFunction = (a,b) => {
    console.log(arguments)
}
arrowFunction(1,2)
//ReferenceError: arguments is not defined


// workaround for arrow function
var arrowFunction = (...args) => {
    console.log(...args)
}
arrowFunction(1,2)
// 1 2
```

3. this keyword - Normal function depends on how it is being called and arrow function is on where it is being called.
4. New keyword - Regular functions are constructible, they can be called using the new keyword.

```
function add (x, y) {
    console.log(x + y)
}
let sum = new add(2,3);
// 5
```

```
let add = (x, y) => console.log(x + y);
const sum = new add(2,4);
// TypeError: add is not a constructor
```

5. Function Hoisting - In regular function, function gets hoisting at top.

```
normalFunc()
function normalFunc() {
    return "Normal Function"
}
// "Normal Function"
```

In arrow function, function get hoisted where you define. So, if you call the function before initialisation you will get referenceError.

```
arrowFunc()
const arrowFunc = () => {
    return "Arrow Function"
}
// ReferenceError: Cannot access 'arrowFunc' before initialization
```

▼ What is Lexical Environment / lexical Scope?

Lexical scoping means that the accessibility of variables is determined by the position of the variables inside the nested scopes.

Simpler, the lexical scoping means that inside the inner scope you can access variables of outer scopes.

It's called *lexical* (or *static*) because the engine determines (at lexing time) the nesting of scopes just by looking at the JavaScript source code, without executing it.

▼ What is Currying?

Currying is the process of taking a function with multiple arguments and turning it into a sequence of functions each with only one single arguments.

```
function multiply(a) {
  return function executeMultiply(b) {
    return a * b;
  }
}
const double = multiply(2);
double(3); // => 6
double(5); // => 10
const triple = multiply(3);
triple(4); // => 12


currying upto n number
======================
function sum(a){
  return (b)=>{
    return b ? sum(a+b) : a
  }
}

sum(2)(3)(4)(5)(6)()



const createURL = baseURL => {
  const protocol = "https";

  // we now return a function, that accepts a 'path' as an argument
  return path => {
    return `${protocol}://${baseURL}/${path}`;
  };
};

// we create a new functions with the baseURL value in it's closure scope
const createSiteURL = createURL("mysite.com");

// create URLs for our main site
const homeURL = createSiteURL("");
const loginURL = createSiteURL("login");
const productsURL = createSiteURL("products");
const contactURL = createSiteURL("contact-us");

console.log(loginURL)
```

▼ Difference between Call, Apply and Bind?

Call and Apply calls a function, while bind returns a new function. Arguments are passed individually on call while apply expects an array of arguments.

call and apply➖

```
let participant1 = {
  name:"lily",
  battery: 40,
  chargeBattery: function(a,b){
    this.battery = this.battery + a + b;
  }
}

let participant2 = {
  name:"john",
  battery: 30
}

participant1.chargeBattery.call(participant2, 20, 30)
participant1.chargeBattery.apply(participant2, [ 20, 30 ])

-----------------------------------------------------------------------------------------------
var pokemon = {
    firstname: 'Pika',
    lastname: 'Chu ',
    getPokeName: function() {
        var fullname = this.firstname + ' ' + this.lastname;
        return fullname;
    }
};

var pokemonName = function(snack, hobby) {
    console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
};

pokemonName.call(pokemon,'sushi', 'algorithms'); // Pika Chu  loves sushi and algorithms
pokemonName.apply(pokemon,['sushi', 'algorithms']); // Pika Chu  loves sushi and algorithms
```

Bind -

```
var pokemon = {
    firstname: 'Pika',
    lastname: 'Chu ',
    getPokeName: function() {
        var fullname = this.firstname + ' ' + this.lastname;
        return fullname;
    }
};

var pokemonName = function(snack, hobby) {
    console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
};

var logPokemon = pokemonName.bind(pokemon); // creates new object and binds pokemon. 'this' of pokemon === pokemon now

logPokemon('sushi', 'algorithms'); // Pika Chu  loves sushi and algorithms
```

▼ What is an Event Loop?

An event loop is something that pulls the task of the callback queue / microtask queue and places it onto the execution call stack whenever the call stack becomes empty

▼ Prototype in Javascript?

Prototype is mechanism by which javascript object inherit features from another object.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
let lydia = new Person("Lydia", "Hallie");
Person.prototype.getFullName = function(){
  return this.firstName + this.lastName
}
lydia.getFullName() // This now works!
```

▼ what is memoization?

It is a technique used to speed up a program by storing the result of expensive function calls and returning the cached results when same input occurs again.

▼ What is higher order functions?

It is a function that accepts another function as an argument and even can return the function as a value.

▼ What is event delegation?

It is a useful pattern of adding an event listener to a single element (parent) instead of assigning events on multiple elements.

▼ What is promises?

Promise is an object that produce a single value. With promises, we can postpone the execution of a code block until an asynchronous operation is completed. This way, other operations can keep running without interruption.

Promises have three states:

1. Pending: the initial state of the promise
2. Fulfilled: the specified operation was completed.
3. Rejected: the operation did not complete.

```
let prom = new Promise((res, rej) => {
  console.log('synchronously executed');
  if (Math.random() > 0.5) {
    res('Success');
  } else {
    rej('Error');
  }
})


prom.then((val) => {
  console.log('asynchronously executed: ' + val);
}).catch((err) => {
  console.log('asynchronously executed: ' + err);
}).finally(() => {
  console.log('promise done executing');
});


console.log('last log');
```

▼ What is strict mode in JS?

It is useful for writing secure JS code. It prevents from bugs from happening and throws more exception.

▼ Difference between null and undefined?

- null is a special keyword that indicates an absence of value.
- undefined property indicates that a variable has not been assigned a value including null too.

▼ Difference between asynchronous and synchronous?

Synchronous are blocking and asynchronous are not, synchronous complete running the current code before executing the next code are executed while asynchronous continue on the next code without completing the current code.

▼ Var, let and const?

- var are function scoped,
- Let and const are block-scoped.

▼ What is DOM?

It stands for Document Object Model. this can be used to access and change the document structure, style, and content.

▼ Different Data types in JS?

Number, string , Boolean , object , undefined , null

▼ Primitive Data types?

- primitive data type is data that is not an object and has no methods and is immutable.
- There are 7 primitive data types: string, number, bigint, boolean, undefined, symbol, and null. ... All primitives are immutable, i.e., they cannot be altered.
- All primitive data type interact by value

▼ What is Hoisting?

In the creation phase of the code, memory space is set up for variables and functions. This is called hoisting

▼ What is scope chain?

When a variable is used in JavaScript, the JavaScript engine will try to find the variable’s value in the current scope. If it could not find the variable, it will look into the outer scope and will continue to do so until it finds the variable or reaches global scope.

▼ Difference between arrow function and normal function?

- Normal function depends on how the this keyword is called or on which object is it called.
- Arrow function only depends on where the function is called.
- We can make an new instance of a function(constructor) using new keyword for normal functions.
- With arrow function there is no constructer involved.

▼ Difference between Object.freeze() and Object.seal()?

- Object.freeze() - Read only
- Object.seal() - Read and Update only

▼ Deep Copy Vs Shallow Copy?

Primitive data type are deep copy by default

Non primitive have shallow copy and Deep copy

- Ways to copy an object
- Assignment operator (=) // shallow copy
- JSON.parse(JSON.stringify(obj)). // only work for object and not for functions inside object // deep copy
- Object.assign({}, obj). // partial deep copy but cannot deep copy nested object
- Spread operator - {…obj} // partial deep copy .. can’t copy nested obj

————Total deep copy using spread ————————————————

```
Let copiedValue = {…originalValue}

copiedValue = {

    …copiedValue,

    name: “Alisha”,

    address: {

    …copiedValue.address,

    city: “Goa”

  }

}
```

▼ What is Async/Await?

- The newest way to write asynchronous code in JavaScript.
- It is non blocking (just like promises and callbacks).
- Async/Await was created to simplify the process of working with and writing chained promises.
- Async functions return a Promise. If the function throws an error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

```
(async function() {
  const result = await asyncOperation(params);
  // Called when the operation completes
})();

const sendGetRequest = async () => {
    try {
        const resp = await axios.get('your url', {
            headers: {
                'authorization': 'Bearer YOUR_JWT_TOKEN_HERE'
            }
        });

        console.log(resp.data);
    } catch (err) {
        // Handle Error Here
        console.error(err);
```

```
    }
};
```

▼ ES6 Features

1. Let and const
2. Arrow functions
3. Template Literals
4. Default parameters
5. Object / Array destructure
6. Spread

▼ Explain event bubbling and how one may prevent it

Event bubbling is the concept in which an event triggers at the deepest possible element, and triggers on parent elements in nesting order. As a result, when clicking on a child element one may exhibit the handler of the parent activating.

One way to prevent event bubbling is using `event.stopPropagation()` or `event.cancelBubble` on IE < 9.

▼ What is Coercion

JavaScript is a **dynamically typed language**: we don't specify what types certain variables are. Values can automatically be converted into another type without you knowing, which is called *implicit type coercion*. **Coercion** is converting from one type into another.

ex- sum(1, “2”) → 12

▼ Debouncing

```
const debounce = (func, time) => {
    let timer;
    return () => {
      clearTimeout(timer);
     timer = setTimeout(() => {
        func();
      }, time);
    };
  };

  const debounceTest = () => {
    console.log("testing debounce");
  };

  console.log(debounce(debounceTest, 1000));
```

Newly Questions