JavaScript Interview Questions


```
// Install the TensorFlow.js library
// npm install @tensorflow/tfjs

const tf = require('@tensorflow/tfjs');

// Generate some example data
const xs = tf.tensor1d([1, 2, 3, 4]);
const ys = tf.tensor1d([3, 6, 9, 12]);

// Define a simple linear model
const model = tf.sequential();
model.add(tf.layers.dense({units: 1, inputShape: [1]}));

// Compile the model
model.compile({
  optimizer: 'sgd',
  loss: 'meanSquaredError'
});
```

JS

1. Explain Closures

A closure is a function that has access to its own scope, the outer function's scope, and the global scope. This allows it to access variables from its outer function even after the outer function has finished executing.

```
function outerFunction(outerVar) {
    return function innerFunction(innerVar) {
        console.log(outerVar + innerVar);
    }
}

const closureExample = outerFunction(5);
closureExample(3); // Output: 8

/*
 * In this example, outerFunction returns innerFunction, creating a
 * closure. *The innerFunction still has access to the outerVar
 * even after outerFunction has finished executing.
 */
```

2. What is the Event Loop?

JS

The event loop is a mechanism that allows JavaScript to handle asynchronous operations. It continuously checks the call stack and the callback queue. When the call stack is empty, it takes the first function from the queue and pushes it onto the call stack.

```
/* Event Loop */

console.log('Start');                  // The output of this code will be:

setTimeout(() => {                     // Start
    console.log('Timeout');            // End
}, 0);                                 // Promise
                                       // Timeout
Promise.resolve().then(() => {
    console.log('Promise');            // This demonstrates how asynchronous operations are
});                                    // managed by the event loop

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

3. What is ‘this’ in JavaScript

JS

‘this’ refers to the object that is currently being operated on. Its value is determined by how a function is called, and it can be influenced by the function's context.

```
/* 'this' in JavaScript: */

const person = {
    name: 'John',
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

person.greet(); // Output: Hello, my name is John

/* In this example, 'this' refers to the person object
 * because greet is called on person.
 */
```

JS

4. Difference between let, const, and var

‘var’ is function-scoped, ‘let’ and ‘const’ are block-scoped. ‘var’ can be redeclared, ‘let’ can't be redeclared in the same scope, and ‘const’ can't be reassigned after initialization.

```
/* let, const, and var: */

function example() {
    var x = 10;
    if (true) {
        let y = 20;
        const z = 30;
        console.log(x, y, z);
    }
    console.log(x, y); // Throws an error because y is not defined
    console.log(x, z); // Throws an error because z is not defined
}

/* This demonstrates the scoping differences between
 * var, let, and const.
 */
```

JS

5. What is Promise?

Promise is an object that represents a future value. It allows you to handle asynchronous operations in a more synchronous-like manner. A promise can be in one of three states: pending, fulfilled, or rejected.

```
/* Promises */

const myPromise = new Promise((resolve, reject) => {
    const success = true;
    if (success) {
        resolve('Promise resolved');
    } else {
        reject('Promise rejected');
    }
});

myPromise.then((message) => {
    console.log(message); // Output: Promise resolved
}).catch((message) => {
    console.log(message); // Not executed in this example
});

/* Here, we create a Promise that can
 * either resolve or reject, depending on the condition.
 */
```

JS

6. Explain the concept of Promises chaining.

Promises chaining is a technique used to perform multiple asynchronous operations in a specific order. It involves returning a promise from the then method, which allows you to chain multiple asynchronous operations together.

```
/* Promises Chaining: */

const promise1 = new Promise((resolve) ⇒ setTimeout(() ⇒ resolve(1), 1000));
const promise2 = new Promise((resolve) ⇒ setTimeout(() ⇒ resolve(2), 2000));

promise1.then((result) ⇒ {
    console.log(result); // Output: 1
    return promise2;
}).then((result) ⇒ {
    console.log(result); // Output: 2
});

/*
 * This demonstrates how to chain promises using .then().
 */
```

7. What is 'async/await'

JS

async/await is a syntactic sugar built on top of Promises. It provides a more synchronous-looking way to write asynchronous code. async defines a function that returns a promise, and await is used to wait for a promise to resolve or reject.

```
/* async/await: */

function resolveAfter2Seconds() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('resolved');
        }, 2000);
    });
}

async function asyncCall() {
    console.log('Calling...');
    const result = await resolveAfter2Seconds();
    console.log(result); // Output: resolved
}

asyncCall();

/*
 * Here, asyncCall uses await to wait for
 * the promise to resolve before continuing execution.
 */
```

JS

8. Explain the concept of Prototypal Inheritance

In JavaScript, objects can inherit properties and methods from other objects through a prototype chain. Each object has a prototype object, and when a property or method is not found on an object, JavaScript looks up the prototype chain until it finds the property or reaches the end of the chain.

```
/* Prototypal Inheritance: */

function Person(name) {
    this.name = name;
}

Person.prototype.greet = function() {
    console.log(`Hello, my name is ${this.name}`);
}

function Student(name, grade) {
    Person.call(this, name);
    this.grade = grade;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

const student = new Student('John Doe', 'A');
student.greet(); // Output: Hello, my name is John Doe\

/* In this example, Student inherits from Person using prototypal inheritance. */
```

JS

9. What is the purpose of the bind() method?

The bind() method is used to create a new function with a specified this value and initial arguments. It's often used to set the context of a function to a specific object.

```
/* bind(): */

const person = {
    name: 'John',
    greet: function() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

const greetFn = person.greet.bind(person);
greetFn(); // Output: Hello, my name is John

/* Here, we use bind() to set the context of greet to the person object. */
```

JS

10. Explain the concept of Hoisting in JavaScript.

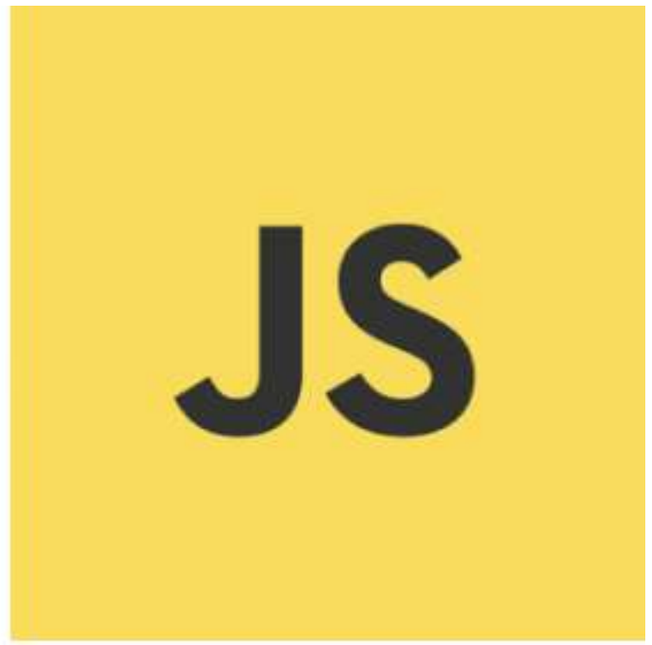
Hoisting is a mechanism in JavaScript where variable and function declarations are moved to the top of their containing scope during the compile phase. This allows you to use a variable or function before it's declared.

```
/* Hoisting: */

x = 5;
console.log(x); // Output: 5
var x;

/*
 * In this example, the variable
 * x is hoisted to the top of the scope,
 * so it can be used before it's declared.
 */
```

Thanks!



Remember, in interviews, it's not just about knowing the answers, but also being able to explain them clearly and possibly provide examples to demonstrate your understanding.