**Total javascript questions here:-52+151+15+123=341(mostly non-data structures and algorithms)**

**Additional 1000 questions, mostly open-ended javascript questions**

***### Easy Mistakes/Topics:***

1. \*\*Syntax Errors\*\*: Forgetting semicolons, parentheses, or curly braces.

2. \*\*Variable Scope\*\*: Misunderstanding var, let, and const and their scoping rules.

3. \*\*Data Types\*\*: Confusion between different data types such as strings, numbers, and objects.

4. \*\*Type Coercion\*\*: Unintended type conversions leading to unexpected behavior.

5. \*\*Equality Operators\*\*: Incorrectly using '==' instead of '===' or vice versa.

6. \*\*Function Invocation\*\*: Misunderstanding function calls and passing arguments.

7. \*\*Array Manipulation\*\*: Incorrectly adding, removing, or accessing elements in arrays.

**### Medium Mistakes/Topics:**

1. \*\*Asynchronous JavaScript\*\*: Mishandling asynchronous operations with callbacks, promises, or async/await.

2. \*\*Event Handling\*\*: Improperly binding event listeners or handling event propagation.

3. \*\*DOM Manipulation\*\*: Inefficient DOM manipulation leading to performance issues.

4. \*\*Error Handling\*\*: Inadequate error handling strategies leading to uncaught exceptions.

5. \*\*Scope Chain\*\*: Misunderstanding lexical scoping and closures.

6. \*\*Memory Leaks\*\*: Not releasing references to unused objects, causing memory leaks.

7. \*\*Hoisting\*\*: Misunderstanding variable and function hoisting behavior.

***### Hard Mistakes/Topics:***

1. \*\*Prototypal Inheritance\*\*: Incorrectly implementing inheritance in JavaScript using prototypes.

2. \*\*Context and this keyword\*\*: Misunderstanding the context of the 'this' keyword.

3. \*\*Module Systems\*\*: Managing dependencies and using module systems like CommonJS or ES6 modules.

4. \*\*Performance Optimization\*\*: Identifying and optimizing performance bottlenecks in JavaScript code.

5. \*\*Security\*\*: Preventing common security vulnerabilities like cross-site scripting (XSS) or injection attacks.

6. \*\*Functional Programming\*\*: Understanding and applying functional programming concepts like map, filter, and reduce.

7. \*\*Browser Compatibility\*\*: Dealing with cross-browser inconsistencies and ensuring compatibility.

Mastering these topics requires practice and continuous learning. It's essential to understand not just how JavaScript works, but also best practices and common pitfalls to avoid.

**Additional resources**:- Check this Github repository for **junior javascript developer** level questions, **151 Questions** in total <https://github.com/lydiahallie/javascript-questions>

**Quality open ended 1000 questions**

<https://github.com/sudheerj/javascript-interview-questions?tab=readme-ov-file#what-is-promise-chaining>

**Set of 15 quality conceptual questions:**-

<https://lokesh-prajapati.medium.com/15-tricky-javascript-interview-questions-that-will-test-your-skills-aca1053b1876>

**Set of 123 junior-mid level questions**

<https://github.com/ganqqwerty/123-Essential-JavaScript-Interview-Questions>

**Another type of question**:-https://education.launchcode.org/intro-to-professional-web-dev/chapters/errors-and-debugging/exercises.html

1. **Overuse of Global Variables**

// Incorrect: Overuse of global variables

let count = 0;

function incrementCount() {

count++;

}

```

To correct this, the interviewee should minimize the use of global variables by encapsulating them within functions or modules.

**2.Not Using Functional Programming Paradigm**

// Incorrect: Not using functional programming paradigm

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

let sum = 0;

for (let i = 0; i < numbers.length; i++) {

sum += numbers[i];

}

```

To correct this, the interviewee should utilize functional programming methods like `reduce()` to achieve the same result more elegantly.

**3.Not Understanding Event Bubbling and Capturing**

// Incorrect: Misunderstanding event bubbling and capturing

document.getElementById('outer').addEventListener('click', () => {

console.log('Outer clicked');

});

document.getElementById('inner').addEventListener('click', (event) => {

event.stopPropagation();

console.log('Inner clicked');

});

```

To correct this, the interviewee should understand event bubbling and capturing and ensure event propagation is handled appropriately based on the application's requirements.

**4.Not Utilizing ES6+ Features**

// Incorrect: Not utilizing ES6+ features

function multiply(a, b) {

return a \* b;

}

```

To correct this, the interviewee should utilize ES6+ features like arrow functions, destructuring, or default parameters to improve code readability and maintainability.

**5.Ignoring Code Reviews and Testing**

```javascript

// Incorrect: Ignoring code reviews and testing

function divide(a, b) {

return a / b;

}

```

To correct this, the interviewee should actively participate in code reviews and write tests to ensure code quality and reliability.

**6.Not Handling Promises Rejection**

```javascript

// Incorrect: Not handling promises rejection

async function fetchData() {

const response = await fetch('https://api.example.com/data');

if (!response.ok) {

throw new Error('Failed to fetch data');

}

return response.json();

}

```

To correct this, the interviewee should handle promise rejections using `.catch()` or `try-catch` blocks to avoid unhandled promise rejections.

**7.Over-complicating Callback Functions**

```javascript

// Incorrect: Over-complicating callback functions

function processArray(array, callback) {

for (let i = 0; i < array.length; i++) {

if (callback(array[i])) {

console.log(array[i]);

}

}

}

```

To correct this, the interviewee should simplify callback functions and ensure they follow the single responsibility principle.

**8.Not Optimizing Network Requests**

```javascript

// Incorrect: Making too many network requests

function fetchData() {

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data));

// Additional fetch requests

}

```

To correct this, the interviewee should optimize network requests by batching them or using techniques like data caching.

**9.Overlooking Semantic HTML**

```html

<!-- Incorrect: Not using semantic HTML -->

<div onclick="openPage()">Click here</div>

```

To correct this, the interviewee should use semantic HTML elements like buttons, anchors, or inputs to improve accessibility and maintainability.

**10.Not Using ES6 Modules**

```javascript

// Incorrect: Not using ES6 modules

import { fetchData } from './api.js';

fetchData();

```

To correct this, the interviewee should use ES6 modules for better code organization and maintainability, ensuring they are properly imported and exported.

Sure, here are some code snippets illustrating common mistakes along with explanations:

**11. \*\*Misunderstanding Asynchronous Operations\*\*:**

```javascript

// Incorrect: Not waiting for asynchronous operation to complete

function fetchData() {

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data));

}

fetchData();

```

In this example, `fetchData()` initiates an asynchronous operation but doesn't wait for it to complete before moving on, potentially resulting in race conditions or unexpected behavior.

**12. \*\*Memory Leaks\*\*:**

```javascript

// Incorrect: Not removing event listeners

function addClickListener() {

const button = document.getElementById('myButton');

button.addEventListener('click', () => {

console.log('Button clicked');

});

}

addClickListener(); // Assume this function is called multiple times

```

In this example, `addClickListener()` adds a click event listener to a button but never removes it. If this function is called multiple times (e.g., during component re-renders), it will keep adding new event listeners, leading to memory leaks.

**13. \*\*Variable Scope Issues\*\*:**

```javascript

// Incorrect: Incorrect usage of variable scope with var

function printNumbers() {

for (var i = 0; i < 5; i++) {

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

}

}

printNumbers();

```

In this example, the variable `i` is declared using `var`, which has function-level scope, not block-level scope. As a result, the value of `i` will be `5` for all the setTimeout callbacks, rather than the expected sequence of numbers.

**14. \*\*Incorrect DOM Manipulation\*\*:**

```javascript

// Incorrect: Inefficient DOM manipulation

const container = document.getElementById('container');

for (let i = 0; i < 1000; i++) {

container.innerHTML += `<div>${i}</div>`;

}

```

In this example, innerHTML is repeatedly modified within a loop to add 1000 div elements, which is highly inefficient and can cause performance issues. Using methods like `appendChild` or creating elements using `document.createElement` would be more efficient.

**15. \*\*Not Handling Errors Properly\*\*:**

```javascript

// Incorrect: Not handling errors properly

try {

undefinedFunction(); // Function does not exist

} catch (error) {

console.error('Error:', error);

}

```

In this example, `undefinedFunction()` is called within a try-catch block, but since the function does not exist, it will result in an uncaught reference error. The catch block should handle this error appropriately.

These examples demonstrate common mistakes in JavaScript development that interviewees should be familiar with and able to identify.

***Notes for next questions:-***

**Easy Mistakes:**

1. Scope Issues (using let and understanding scope)
2. Type Coercion Confusion (using === for strict comparison)
3. Missing return Statements
4. Basic for Loop Errors (incorrect loop conditions or index access)
5. Array Index Out of Bounds (checking index before accessing)
6. String Concatenation Issues (using template literals or concatenation operator)
7. Accidental Global Variables (using var cautiously or declaring within functions)
8. Confusing == and === (using === for strict comparison)
9. Forgetting Semicolons (adding semicolons after statements)
10. Using Incorrect Operators (choosing appropriate operators)

**Medium Mistakes:**

1. Asynchronous Callback Confusion (ensuring proper execution order)
2. Closures and Scope (using arrow functions or binding this)
3. Incorrect Array Methods (using appropriate methods and understanding behavior)
4. try...catch Blocks (catching specific errors and providing informative messages)
5. Event Delegation (using event delegation for efficiency)
6. Callbacks and Promises (understanding asynchronous behavior and choosing appropriate approach)
7. Closures and Event Listeners (being mindful of variable scope and memory leaks)

**Hard Mistakes:**

1. Memory Leaks (using weak maps or cleanup functions)
2. Prototype Chain Confusion (setting prototype correctly and considering class syntax)
3. Advanced Error Handling (utilizing techniques like error boundaries and custom error messages)
4. Webpack Configuration Issues (understanding configuration options and troubleshooting methods)
5. Complex State Management (choosing appropriate state management library and handling updates efficiently)

**EASY**

**16: Scope Issues**

JavaScript



function calculateArea(width, height) {

let area = width \* height; // Fix 1: Declare area inside the function

if (width > height) {

let message = "Width is greater than height."; // Fix 2: Use let for message

}

return area;

}

let result = calculateArea(10, 5);

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

// Expected output: 50

**Explanation:**

* **Mistake:** Variables declared within an if block are not accessible outside it.
* **Fix:** Move area declaration inside the function and use let for message to limit its scope.

**17: Type Coercion Confusion (Unintended Behavior)**

JavaScript



function compareValues(value1, value2) {

if (value1 == value2) { // Fix: Use === for strict comparison

console.log("Values are equal.");

} else {

console.log("Values are different.");

}

}

compareValues(0, "0"); // Output: Values are equal. (Incorrect)

// Expected output: Values are different.

**Explanation:**

* **Mistake:** Loose equality (==) can lead to unexpected results due to type coercion.
* **Fix:** Use strict equality (===) to compare both value and type.

**18: Missing** return **Statement (Incorrect Function Behavior)**

JavaScript



function greet(name) {

let greeting = "Hello, " + name + "!"; // Fix: Add return statement

}

let message = greet("Alice");

console.log(message); // Output: undefined (Incorrect)

// Expected output: Hello, Alice!

**Explanation:**

* **Mistake:** Omitting return in a function might not produce the desired output.
* **Fix:** Explicitly return the value you want the function to provide.

**19:** for **Loop Issues (Incorrect Iteration or Index Access)**

JavaScript



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

let sum = 0;

for (let i = 0; i < numbers.length - 1; i++) { // Fix: Iterate up to numbers.length - 1

sum += numbers[i + 1]; // Fix: Access correct element with i + 1

}

console.log(sum); // Output: Incorrect sum

// Expected output: 14 (1 + 2 + 3 + 4)

**Explanation:**

* **Mistake:** for loop conditions and index access can be off by one.
* **Fix:** Adjust the loop condition to iterate up to the second-to-last element and use i + 1 to access the correct element within the array.

**MEDIUM QUESTIONS**

**20: Asynchronous Callback Confusion (Unexpected Order)**

JavaScript



function fetchData(url, callback) {

setTimeout(() => {

const data = "Retrieved data"; // Fix: Handle potential errors

callback(data);

}, 1000);

}

fetchData("https://api.example.com/data", (data) => {

console.log(data); // Might not be guaranteed to execute after alert

});

alert("Processing data...");

// Output order might be unpredictable.

**Explanation:**

* **Mistake:** Asynchronous nature of setTimeout can lead to unexpected execution order.
* **Fix:** Move the code that relies on the fetched data (console.log in this case) inside the callback to ensure it executes after data retrieval. Additionally, handle potential errors within the fetchData function.

**Best Practices:**

* Use const and let appropriately for variables.
* Employ strict equality (===) for type-safe comparisons.
* Utilize meaningful variable names.
* Read error messages and stack traces carefully during debugging.
* Comment your code for better readability and maintainability.

By incorporating these questions and best practices, you can effectively assess an interviewee's JavaScript debugging skills and understanding of common pitfalls.

Absolutely, here are 10 more JavaScript debugging coding questions for interviewees:

**21: Closures and Scope (Unexpected Variable Behavior)**

JavaScript



function createCounter() {

let count = 0;

return function increment() {

count++; // Fix: Return the incremented value instead of modifying count

return count;

};

}

const counter1 = createCounter();

const counter2 = createCounter();

console.log(counter1()); // Output: 1

console.log(counter2()); // Output: 1 (Incorrect, should be 1)

// Expected output: 2 (counter2 should be independent)

**Explanation:**

* **Mistake:** Closures can lead to unexpected behavior if not understood properly.
* **Fix:** Instead of modifying the count variable inside the returned function, return the incremented value to maintain separate state for each counter instance.

**22:** this **Keyword Confusion (Incorrect Object Reference)**

JavaScript



const person = {

name: "Alice",

greet: function() {

console.log(this.name); // Fix: Use arrow function or bind `this`

},

};

person.greet(); // Output: undefined (Incorrect)

// Expected output: Alice

**Explanation:**

* **Mistake:** The value of this can change depending on context, leading to unexpected references within object methods.
* **Fix:** Use an arrow function within the object method to preserve the context of this or use person.greet.bind(person) to explicitly bind this to the person object.

**23: Incorrect Array Methods (Unexpected Mutation or Output)**

JavaScript



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

const doubledNumbers = numbers.map(function(number) { // Fix: Use arrow function for simpler syntax

return number \* 2;

});

console.log(numbers); // Output: [1, 2, 3, 4] (Incorrect)

// Expected output: [2, 4, 6, 8] (Original array shouldn't be modified)

**Explanation:**

* **Mistake:** Array methods like map create a new array, but some developers might accidentally modify the original.
* **Fix:** Use an arrow function for simpler syntax and remember that map creates a new transformed array.

**24: Callback Hell (Nested Callbacks for Complex Logic)**

JavaScript



function fetchData(url, callback) {

setTimeout(() => {

const data = { name: "John" };

callback(data, (user) => {

console.log(`Welcome, ${user.name}!`); // Fix: Refactor with promises or async/await

});

}, 1000);

}

fetchData("https://api.example.com/user", (data, successCallback) => {

successCallback(data);

});

// Output might be hard to read and maintain.

**Explanation:**

* **Mistake:** Nesting callbacks can lead to "callback hell," making code harder to read and maintain.
* **Fix:** Refactor the code using promises or async/await for cleaner asynchronous handling.

**HARD QUESTIONS**

**25: Prototype Chain Confusion (Unexpected Inheritance Behavior)**

JavaScript



function Animal(name) {

this.name = name;

}

Animal.prototype.makeSound = function() {

console.log("Generic animal sound");

};

function Dog(name, breed) {

Animal.call(this, name); // Fix: Use Object.create or class syntax

this.breed = breed;

}

Dog.prototype = Object.create(Animal.prototype); // Fix: Set prototype correctly

Dog.prototype.constructor = Dog; // Fix: Reset constructor

const dog = new Dog("Fido", "Labrador");

dog.makeSound(); // Output: Generic animal sound

// Expected output: (Breed-specific dog sound)

**Explanation:**

* **Mistake:** Misunderstanding the prototype chain can lead to incorrect inheritance behavior.
* **Fix:** Use Object.create to create the prototype correctly, set the constructor property to avoid prototype pollution, and consider using class syntax for modern JavaScript.

These additional questions address various aspects of JavaScript debugging, giving you a well-rounded assessment of an interviewee's problem-solving skills.

Here are 3 more JavaScript debugging coding questions for interviewees:

**27: Memory Leaks (Unintended Object Retention)**

JavaScript



function createPerson(name) {

const person = {

name: name,

hobbies: [],

addHobby(hobby) {

this.hobbies.push(hobby); // Fix: Consider using a weak map or cleanup function

},

};

return person;

}

const people = [];

for (let i = 0; i < 1000; i++) {

people.push(createPerson(`Person ${i}`));

}

console.log(people[0]); // Output: Might not include hobbies due to garbage collection

// Expected output: { name: "Person 0", hobbies: [...] }

**Explanation:**

* **Mistake:** Closures and event listeners can lead to memory leaks if references to objects are not properly managed.
* **Fix:** Consider using a weak map to store hobbies or implement a cleanup function to remove unnecessary references.

**28:** try...catch **Blocks (Incorrect Error Handling)**

JavaScript



function readFile(filename) {

try {

const content = fs.readFileSync(filename, "utf8"); // Fix: Handle specific errors (e.g., ENOENT)

return content;

} catch (error) {

console.error("Error reading file:", error); // Fix: Provide more informative error messages

}

}

const content = readFile("nonexistent\_file.txt");

if (content) {

console.log(content);

} else {

console.log("File not found."); // Fix: Handle specific errors gracefully

}

// Expected output: Handle specific errors (e.g., file not found) and provide informative messages.

**Explanation:**

* **Mistake:** Generic catch blocks can mask specific errors, making debugging harder.
* **Fix:** Catch specific errors like ENOENT (file not found) and provide informative error messages for better debugging and user experience.

**29: Event Delegation (Incorrect Event Handling)**

JavaScript



const container = document.getElementById("container");

container.addEventListener("click", function handleClick(event) {

if (event.target.classList.contains("item")) {

console.log("Clicked on an item!"); // Fix: Use event delegation for efficiency

}

});

const item1 = document.createElement("div");

item1.classList.add("item");

container.appendChild(item1);

const item2 = document.createElement("div");

item2.classList.add("item");

container.appendChild(item2);

// Clicking on container might not trigger the desired behavior.

**Explanation:**

* **Mistake:** Attaching event listeners to individual elements can be inefficient when dealing with dynamically added content.
* **Fix:** Use event delegation to attach a single event listener to the container and filter for the target element within the handler for better performance.

## **Medium Mistakes (Continued)**

**30. Incorrect Array Methods (Solution: Use appropriate methods and understand their behavior)**

JavaScript



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

const doubledNumbers = numbers.map(function(number) { // Fix: Use arrow function for simpler syntax

return number \* 2;

});

console.log(numbers); // Output: [1, 2, 3, 4] (Incorrect)

// Fix: Array methods like `map` create a new array. Use `numbers.forEach` to modify the original array in-place (if intended).

const doubledInPlace = numbers.forEach((number, index) => numbers[index] \*= 2);

console.log(numbers); // Output: [2, 4, 6, 8]

**31.** try...catch **Blocks (Solution: Catch specific errors and provide informative messages)**

JavaScript



function readFile(filename) {

try {

const content = fs.readFileSync(filename, "utf8"); // Fix: Handle specific errors (e.g., ENOENT)

return content;

} catch (error) {

console.error("Error reading file:", error); // Fix: Provide more informative error messages

}

}

const content = readFile("nonexistent\_file.txt");

if (content) {

console.log(content);

} else {

console.log("File not found."); // Fix: Handle specific errors gracefully

}

// Fix 1: Catch specific errors like `ENOENT` (file not found).

// Fix 2: Provide more details in the error message (e.g., filename, error code).

try {

const content = fs.readFileSync(filename, "utf8");

// ...

} catch (error) {

if (error.code === 'ENOENT') {

console.error(`File not found: ${filename}`);

} else {

console.error("Error reading file:", error);

}

}

**32. Event Delegation (Solution: Use event delegation for efficiency)**

JavaScript



const container = document.getElementById("container");

container.addEventListener("click", function handleClick(event) {

if (event.target.classList.contains("item")) {

console.log("Clicked on an item!"); // Fix: Use event delegation for efficiency

}

});

const item1 = document.createElement("div");

item1.classList.add("item");

container.appendChild(item1);

const item2 = document.createElement("div");

item2.classList.add("item");

container.appendChild(item2);

// Fix: Use event delegation to attach a single listener to the container and filter for the target element within the handler.

container.addEventListener("click", function(event) {

if (event.target.classList.contains("item")) {

console.log("Clicked on an item!");

}

});

**33. Callbacks and Promises (Solution: Understand asynchronous behavior and choose appropriate approach)**

JavaScript



function getUserData(userId, callback) {

setTimeout(() => {

const user = { name: "John Doe" };

callback(user);

}, 1000);

}

function getPosts(userId) {

return new Promise((resolve, reject) => {

setTimeout(() => {

const posts = ["Post 1", "Post 2"];

resolve(posts);

}, 2000);

});

}

getUserData(1, (user) => {

console.log(user); // Output: { name: "John Doe" }

getPosts(user.id).then((posts) => {

console.log(posts); // Might not work if `user.id` is undefined due to callback order

});

});

// Fix: Use promises or async/await for cleaner asynchronous handling.

async function fetchData() {

const user = await getUserData(1);

const posts = await getPosts(user.id);

console.log(user);

console.log(posts);

}

fetchData();

**34. Closures and Event Listeners (Solution: Be mindful of variable scope and memory leaks)**

JavaScript



const buttons = document.querySelectorAll("button");

buttons.forEach((button) => {

button.addEventListener("click", function() {

let count = 0; // Fix: Move count outside the loop to avoid memory leaks

count++;

console.log(count); // Might output unexpected



## **Easy Mistakes (Continued)**

**35. Missing Brackets in** if **Statements (Solution: Use proper indentation and brackets)**

JavaScript



function isEven(number) {

if number % 2 === 0 // Fix: Add missing brackets

console.log("Number is even.");

else

console.log("Number is odd.");

}

isEven(4); // Incorrect output

// Fix: Ensure proper indentation and brackets for `if` statements.

**36. Incorrect Variable Declaration (Solution: Use** const **and** let **appropriately)**

JavaScript



for (let i = 0; i < 5; i++) { // Fix: Use `let` for loop counter

const message = "Iteration " + i;

console.log(message);

}

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

// Fix: Use `let` for variables that need to be reassigned within a loop.

**37. Array Index Out of Bounds (Solution: Check index before accessing)**

JavaScript



const colors = ["red", "green", "blue"];

console.log(colors[3]); // Output: undefined (Incorrect)

// Fix: Check if the index is within the array bounds before accessing.

if (index < colors.length) {

console.log(colors[index]);

}

**38. String Concatenation Issues (Solution: Use template literals or concatenation operator)**

JavaScript



const name = "Alice";

const greeting = "Hello, " + name + "!";

console.log(greeting);

// Fix: Use template literals for cleaner string concatenation.

const greeting = `Hello, ${name}!`;

**39. Accidental Global Variables (Solution: Use** var **cautiously or declare within functions)**

JavaScript



let message = "Global message"; // Fix: Declare message within a function

function sayHello() {

console.log(message); // Might output unexpected values if message is modified elsewhere

}

sayHello();

message = "New message";

sayHello(); // Might still output "Global message"

**40. Confusing** == **and** === **(Solution: Use** === **for strict comparison)**

JavaScript



const num1 = 1;

const num2 = "1";

if (num1 == num2) { // Fix: Use === for strict comparison

console.log("Numbers are equal."); // Incorrect (type coercion)

} else {

console.log("Numbers are different.");

}

**41. Forgetting Semicolons (Solution: Add semicolons after statements)**

JavaScript



function add(x, y) {

return x + y

} // Fix: Missing semicolon

const result = add(5, 3);

console.log(result);

**42. Using Incorrect Operators (Solution: Choose appropriate operators for desired behavior)**

JavaScript



let x = 10;

x = x - 5; // Fix: Use `-=` for shorthand subtraction

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



## **Hard Mistakes (Continued)**

**43. Advanced Error Handling (Solution: Utilize techniques like error boundaries and custom error messages)**

JavaScript



function fetchData(url) {

return fetch(url)

.then((response) => response.json())

.then((data) => data) // Fix: Handle potential errors (e.g., network issues, parsing errors)

.catch((error) => {

console.error("Error fetching data:", error); // Fix: Provide more informative error messages

});

}

fetchData("https://invalid\_api.com/data")

.then((data) => console.log(data))

.catch((error) => console.error(error)); // Might not provide user-friendly error messages

// Fix 1: Handle specific errors (e.g., network errors, parsing errors) within `catch` block.

// Fix 2: Provide more informative error messages that can be displayed to the user.

try {

const data = await fetchData(url);

// ...

} catch (error) {

if (error instanceof NetworkError) {

console.error("Network error occurred. Please check your internet connection.");

} else {

console.error("An error occurred while fetching data. Please try again later.");

}

}

**44. Webpack Configuration Issues (Solution: Understand configuration options and troubleshooting methods)**

JavaScript



const path = require('path');

module.exports = {

entry: './src/index.js', // Fix: Ensure entry point path is correct

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist'), // Fix: Ensure output path is correct

},

// ... other configurations

};

// Fix: Verify entry point and output path are set correctly for your project structure. Refer to Webpack documentation for troubleshooting.

**45. Complex State Management (Solution: Choose appropriate state management library and handle complex state updates efficiently)**

JavaScript



const state = {

todos: [],

filter: 'all',

};

function addTodo(text) {

state.todos.push({ text, completed: false }); // Fix: Consider using a state management library for complex state updates

renderTodos();

}

function toggleTodo(index) {

state.todos[index].completed = !state.todos[index].completed; // Fix: Consider using immutable data structures for state updates

renderTodos();

}

// Fix: For complex applications, consider using a state management library like Redux or MobX to manage state updates and ensure consistency.

These additional questions and explanations provide a well-rounded assessment of an interviewee's ability to handle even the most challenging JavaScript pitfalls. Remember to tailor the complexity of your questions to the specific role and experience level you're seeking.

You're right, I apologize for including more than what you requested. Here are 10 coding debugging questions each for Async/Await, Functional Programming, and Closures, focusing on identifying and fixing errors:

**Async/Await (Debugging):**

46. The following code throws an error "data.name is undefined". Why?

JavaScript



async function getUser() {

try {

const response = await fetch('https://api.example.com/user');

const data = await response.json();

console.log(data.name);

} catch (error) {

console.error(error);

}

}

getUser();

47. This code fetches data but never displays it. Fix the issue.

JavaScript



async function fetchData() {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

// Missing code to handle potential errors (e.g., network failure)

}

fetchData();

1. Debug the following code: it seems to hang and never completes.

JavaScript



async function longRunningTask() {

for (let i = 0; i < 100000; i++) {

await new Promise(resolve => setTimeout(resolve, 10)); // Simulates long task

}

console.log("Task completed");

}

longRunningTask();

**Functional Programming (Debugging):**

48.This code maps over an array but results in undefined values. Fix it.

JavaScript



const users = [{ name: "Alice" }, { name: undefined }, { name: "Bob" }];

const usernames = users.map(user => user.name.toUpperCase());

console.log(usernames);

49. Why does this code using filter return an empty array?

JavaScript



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

const evenNumbers = numbers.filter(num => num % 2 === 0);

console.log(evenNumbers);

50. Debug the following code: it should find the minimum value in the array but returns the wrong result.

JavaScript



const numbers = [5, 10, 2, 8];

const minNumber = numbers.reduce((min, current) => Math.min(min, current));

console.log(minNumber);

**Closures (Debugging):**

1. This code defines a function that should add 5 to a number but always returns 10. Fix it.

JavaScript



function addFive() {

let base = 5;

return function(number) {

return number + base;

}

}

const addFn = addFive();

console.log(addFn(2)); // 10, not 7

51. Why does this code using a closure throw an error "reference is not defined"?

JavaScript



function createGreeter(name) {

let greeting = "Hello, ";

return function() {

console.log(greeting + name);

}

}

const greetAlice = createGreeter("Alice");

name = "Bob"; // Modifying a variable outside the closure's scope

greetAlice();

52. Debug the following code: it seems to create a single counter for all instances, not separate counters.

JavaScript



function createCounter() {

let count = 0;

return function() {

count++; // Might have an issue with variable scope

return count;

}

}

const counter1 = createCounter();

const counter2 = createCounter();

console.log(counter1()); // 1

console.log(counter2()); // 1, not 2

These questions focus on identifying the root cause of the bugs and suggesting fixes, allowing you to assess the candidate's debugging skills.