JAVASCRIPT INTERVIEW QUESTIONS AND ANSWERS FOR JUNIOR DEVELOPERS

Updated on 01/04/2025

codersguild.net

WHAT'S INSIDE	You'll find the most commonly asked JavaScript interview questions tailored for junior developers. Each question includes a detailed and professional answer to help you feel confident and well-prepared for your interview.

1. What's the difference between `var`, `let`, and `const`?

As a developer, I always prefer using `let` and `const` instead of `var`.

- `var` is function-scoped and has issues with hoisting and re-declarations. It can lead to bugs that are hard to debug.
- `let` is block-scoped and allows reassignment. I use it when I know a variable's value will change later.
- `const` is also block-scoped but doesn't allow reassignment. I use const by default unless I know I need to reassign.

This helps me write cleaner and more predictable code. I avoid var in modern codebases.

2. What is hoisting in JavaScript?

Hoisting means JavaScript moves variable and function declarations to the top of their scope during the compilation phase.

For example, if I declare a function using the `function` keyword, I can call it before its actual declaration in the code because it's hoisted.

However, variables declared with `var` are hoisted without their values, so using them before the declaration gives `undefined`.

`let` and `const` are also hoisted but stay in the **Temporal Dead Zone** until the code reaches the declaration line — accessing them earlier throws a `ReferenceError`.

3. Explain the difference between `==` and `===`.

`==` checks for equality **after** doing type coercion, while `===` checks both value **and** type.

I always use `===` in professional code because it avoids unexpected bugs due to implicit type conversions. For example:

```
0 == '0' // true

0 === '0' // false
```

Using `===` ensures that I'm comparing exactly what I expect, both in value and type.

4. What are closures and where would you use them?

A closure is when a function remembers variables from its lexical scope, even if it's executed outside that scope.

I use closures to create private variables or to maintain state between function calls. For example, in a counter function, closures allow me to encapsulate the `count` variable so it can't be accessed or modified directly from the outside.

Closures are useful in module patterns, callbacks, or when building factory functions.

5. What is the difference between `null` and `undefined`?

Both represent absence of value, but in different ways:

- `undefined` means a variable was declared but not assigned a value.
- `null` is an intentional assignment to show that a variable should be empty.

I `use` null when I want to explicitly clear a variable or object reference. Knowing the difference is crucial for writing precise conditionals and debugging.

6. How does the `this` keyword work in JavaScript?

`this` refers to the object that is executing the current function. Its value depends on how the function is called:

- In a method, `this` refers to the object.
- Alone (in strict mode), it's `undefined`.
- In event listeners, `this` points to the element.
- Arrow functions don't have their own `this` — they inherit it from the parent scope.

I always consider the context in which a function runs to predict how `this` will behave.

7. What is event delegation and why is it useful?

Event delegation is a technique where I attach a single event listener to a parent element instead of adding listeners to each child.

This works because events in JavaScript bubble up through the DOM. It improves performance and makes it easier to handle dynamic elements added later to the DOM.

I often use event delegation in lists or menus where elements are generated dynamically.

8. What's the difference between synchronous and asynchronous code?

Synchronous code executes line by line, blocking the next line until the current one finishes.

Asynchronous code doesn't wait and allows other operations to run while waiting for things like HTTP requests or timers.

I use async code to improve performance and responsiveness, especially in network operations. Tools like callbacks, promises, and async/await help handle asynchronous behavior cleanly.

9. What is the event loop in JavaScript?

The event loop is the mechanism that allows JavaScript to handle non-blocking operations despite being single-threaded.

It works with the **call stack** and **task queue** to process synchronous and asynchronous code.

When async code (like `setTimeout`) is called, it's sent to the browser APIs, and once completed, the callback is queued. The event loop moves queued tasks back to the call stack when it's empty.

Understanding the event loop helps me write efficient async code and avoid unexpected delays or race conditions.

10. What's the difference between `map()`, `forEach()`, and `filter()`?

All three are used to iterate over arrays, but for different purposes:

- `forEach()` executes a function on each element but doesn't return a new array.
- `map()` transforms each element and returns a new array.

- `filter()` creates a new array with elements that meet a condition.

I use `map` when I want to modify elements, `filter` when I need to remove or select items, and `forEach` for side-effects like logging or DOM updates.

11. What is the difference between `function` declarations and arrow functions?

A function declaration creates a function that has its own `this`, while an arrow function does not.

Arrow functions are more concise, but they don't bind their own `this` — they inherit it from the surrounding lexical scope. That's useful in callbacks or inside methods where I want to keep the parent context.

Also, function declarations are hoisted, so I can call them before they're defined. Arrow functions are expressions and not hoisted in the same way.

12. What is the difference between `call()`, `apply()`, and `bind()`?

All three are used to control the value of `this` when calling a function:

- `call()` invokes the function immediately and lets me pass arguments one by one.
- `apply()` also invokes immediately, but arguments are passed as an array.
- `bind()` returns a new function with bound `this`, but doesn't execute it immediately.

I usually use `bind()` when I need to set `this` for later execution, like in event handlers.

13. What are template literals in JavaScript?

Template literals are string literals enclosed by backticks. They support **interpolation** using `${}` and allow **multiline strings**.

I use them to write cleaner, more readable code when combining strings and variables. It's a huge improvement over string concatenation using `+`.

```
const name = "John";

console.log(Hello, ${name}!);
```

14. What is the spread operator (`...`) and how is it used?

The spread operator expands elements of an array or object. I use it to copy, merge, or pass elements.

For example:

- Copy an array: `const newArr = [...oldArr];`
- Merge objects: `const obj3 = {...obj1, ...obj2};`

It's a clean, immutable way to handle data structures without side effects.

15. How is `async/await` different from promises?

`async/await` is syntactic sugar over promises that makes asynchronous code easier to read and write.

Instead of chaining `.then()` and `.catch()`, I can write code that looks synchronous:

```
const data = await fetchData();
```

I always wrap `await` calls inside `try...catch` to handle errors. It's much cleaner for complex flows like fetching data or multiple sequential API calls.

16. What are primitive and reference types in JavaScript?

Primitive types (like `string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`) are stored by value. Reference types (like `objects`, `arrays`, `functions`) are stored by reference.

That means when I assign a primitive, I'm copying the value. But when I assign an object, I'm copying the reference. Changing one affects the other.

Understanding this helps me avoid unintended mutations in code, especially with objects and arrays.

17. What is the purpose of `typeof` and `instanceof`?

I use `typeof` to check the type of primitive values:

```
typeof "Hello" // "string"
```

I use `instanceof` to check if an object is an instance of a constructor:

```
new Date() instanceof Date // true
```

They are both useful in debugging or writing type-safe functions, but I always keep in mind that `typeof null` is "`object`", which is a known quirk in JavaScript.

18. What is the difference between deep copy and shallow copy?

A shallow copy only duplicates the first level of an object or array. If the structure has nested objects, they still refer to the same memory.

A deep copy duplicates everything recursively, so the new object is completely independent.

For shallow copies, I use `Object.assign()` or the spread operator. For deep copies, I might use structured cloning (`structuredClone()`), or JSON methods:

```
const deepCopy = JSON.parse(JSON.stringify(obj));
```

—but that approach has limitations with functions and special objects like `Date`.

19. What is `NaN` and how do you check for it?

`NaN` stands for “Not a Number”. It’s the result of invalid or undefined mathematical operations like `0 / 0`.

The tricky part is that `NaN !== NaN`. So I use the `Number.isNaN()` method to check for it reliably:

```
Number.isNaN(NaN) // true
```

I avoid using global `isNaN()` because it tries to coerce non-numbers and can give misleading results.

20. What is a promise and how does it work?

A promise represents a value that might be available now, later, or never. It has three states: `pending`, `fulfilled`, or `rejected`.

I use promises to handle asynchronous operations like API calls. They allow me to chain `.then()` and `.catch()` to handle success and error:

```
fetch(url)
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error(error));
```

Promises make code more readable and help avoid "callback hell".

21. What is the difference between undefined and not defined in JavaScript?

undefined means a variable was declared but hasn't been assigned a value.

not defined means the variable hasn't even been declared anywhere in the accessible scope.

For example:

```
let x;
console.log(x); // undefined
console.log(y); // ReferenceError: y is not defined
```

I understand the difference is subtle but important for debugging. I always initialize variables clearly and avoid accessing anything before declaration.

22. What is lexical scope?

Lexical scope means the scope of a variable is defined by its position in the source code. Functions can access variables from the scope where they were defined, not where they're called.

This is the foundation of closures. I use this knowledge when structuring modules or functions that need access to outer variables, especially in nested callbacks or async logic.

23. How is an array different from an object in JavaScript?

Arrays are a special type of object designed for ordered data, indexed by numbers. Objects store unordered key-value pairs.

I use arrays when I care about the order and need indexed access. I use objects when I want to map properties or manage structured data by keys.

Also, arrays have special methods like `map`, `filter`, and `reduce`, which are not available on plain objects.

24. What is `event bubbling` and how does it work?

Event bubbling means that when an event occurs on a nested element, it first runs the handlers on the target element and then propagates up through its ancestors.

For example, clicking on a `<span>` inside a `<div>` will trigger the click event on the `<span>` first, then on the `<div>`.

I use this concept when working with **event delegation**, or when I need to control the flow using `event.stopPropagation()`.

25. How do you prevent default behavior in an event?

I use `event.preventDefault()` to stop the browser from executing its default action, like submitting a form or following a link.

I always check the use case. For example, in form validation, I call `preventDefault()` to validate input manually before submission.

26. What is the difference between `==` and `Object.is()`?

`==` does type coercion and can give unexpected results.
`Object.is()` is like `===` but with slightly different behavior for edge cases:

```
NaN === NaN       // false

Object.is(NaN, NaN) // true



+0 === -0         // true

Object.is(+0, -0)  // false
```

I use `Object.is()` when I want a precise equality check, especially when dealing with edge cases like `NaN`.

27. What is `typeof null` and why does it return "object"?

`typeof null` returns `"object"` because of a historical bug in JavaScript that was never fixed for backward compatibility.

I know that `null` is a primitive and should ideally return `"null"`. When I need to check for `null`, I always do it explicitly:

```
if (value === null)
```

This helps me avoid confusing results from `typeof`.

28. What is the difference between synchronous and asynchronous iteration?

Synchronous iteration goes through items one after another, blocking the thread.
Asynchronous iteration allows waiting for promises using `for await...of`.

I use `for...of` for normal arrays, and `for await...of` when dealing with async data sources like streams or API calls.

Async iteration helps me control flow in real-time applications or large data operations without freezing the UI.

29. What is `debouncing` and `throttling` in JavaScript?

Both are performance optimization techniques:

- **Debouncing** delays function execution until a certain time has passed since the last call. I use it in search inputs or resize events to avoid excessive execution.
- **Throttling** ensures a function executes at most once in a given interval. It's useful for scroll events or continuous triggers.

They prevent performance bottlenecks in high-frequency event handlers.

30. What are the different ways to clone an object in JavaScript?

There are several ways, depending on the depth of the object:

- **Shallow copy**:

```
const copy = {...original}

const copy2 = Object.assign({}, original)
```

- **Deep copy**:

```
const deepCopy = JSON.parse(JSON.stringify(original));
```

But this doesn't handle methods, `undefined`, or `Date`.

In modern JavaScript, I prefer using `structuredClone()` for deep copies when available.

31. What is the difference between `for...in` and `for...of`?

The key difference between `for...in` and `for...of` is what exactly they iterate over.
`for...in` is used to iterate over the **enumerable properties** (keys) of an object, including those inherited from the prototype chain (unless filtered with `hasOwnProperty`). It's commonly used for objects, not arrays. I tend to avoid using `for...in` on arrays because the order of keys isn't guaranteed, and it can accidentally include inherited properties, which can cause unexpected behavior.

`for...of`, on the other hand, is specifically designed for **iterable structures** like arrays, strings, maps, sets, etc. It gives access to the **values** themselves, not their keys or indices. It's more intuitive and safer when working with lists of values where order and content matter. Internally, it uses the iterable protocol, which means the object must implement a `Symbol.iterator` method.

As a rule of thumb, I use `for...in` only for iterating over object properties and `for...of` when working with collections where I care about the actual data stored inside.

32. What are falsy values in JavaScript?

Falsy values in JavaScript are those that, when evaluated in a boolean context (like inside an `if` statement), automatically convert to `false`. Understanding falsy values is essential for writing predictable conditions and avoiding subtle bugs.

The most commonly known falsy values are: `false`, `0`, `""` (empty string), `null`, `undefined`, and `NaN`. These values are often unintentionally introduced in code — for example, when fetching a value from an API or dealing with optional user input.

Knowing what values are considered falsy helps me write defensive code. For example, when checking if a user has entered a value in a form field, I need to be sure I'm not accidentally treating `0` or an empty string as invalid unless it's truly required.

I try to write conditions that are clear and expressive, especially in larger applications where assumptions about truthiness or falsiness can lead to bugs that are hard to detect.

33. What does `use strict` do in JavaScript?

"use strict" is a directive introduced in ES5 that enables **strict mode** in JavaScript — a mode that enforces stricter parsing and error handling in your scripts or functions. Its purpose is to catch common coding mistakes and prevent potentially unsafe actions.

When strict mode is enabled, JavaScript becomes less forgiving. For instance, you cannot use undeclared variables, assign values to read-only properties, or delete variables that are not deletable. It also disables the silent failure of assignments to non-writable properties, which helps me find bugs earlier in the development cycle.

Strict mode makes JavaScript behave more predictably, especially in complex or modular codebases. It also makes debugging easier because mistakes that would be ignored in non-strict mode now throw meaningful errors.

34. What is the difference between setTimeout and setInterval?

Both setTimeout and setInterval are functions that allow you to schedule code execution at a later time. But they differ significantly in **timing behavior** and **use cases**.

setTimeout is used when I want to run a function once, after a delay. It's perfect for tasks like loading screens, lazy initialization, or delaying an action without repetition. After the delay is complete, the function is added to the task queue and executed once.

setInterval is used to repeat a task continuously at specified intervals, until I manually stop it with clearInterval. It's often used for polling, animations, or recurring updates — though it needs to be handled carefully. If a repeated task takes longer to execute than the interval itself, it can cause overlap or memory issues.

When working with intervals, I'm mindful of performance and ensure the logic inside is lightweight or wrapped in proper control mechanisms. In modern applications, I often consider setTimeout inside a recursive function as a safer and more flexible alternative to setInterval, especially when working with dynamic timings.

35. How do you handle errors in asynchronous code?

Error handling in asynchronous code is critical because traditional try...catch blocks don't work across asynchronous boundaries unless you're using async/await. If errors aren't handled properly, they can lead to unresponsive applications, bad user experiences, or even security vulnerabilities.

For promises, the standard approach is chaining .catch() after .then() to catch any rejected promise. This gives a clear way to handle success and failure paths separately. In more complex flows, I may chain multiple .then() calls, and a final .catch() at the end to capture any failure in the whole chain.

With async/await, I rely on try...catch blocks to write asynchronous logic in a synchronous style, which improves readability and maintainability. I always wrap critical async operations — such as

network requests or file I/O — in try...catch, and I log errors meaningfully or show appropriate UI feedback to the user.