

# JavaScript Interview Questions

40 Essential Q&As — Beginner to Advanced

## ■ Beginner Level (Q1–Q10)

### 1. What are the different data types in JavaScript?

Primitive types: **string**, **number**, **bigint**, **boolean**, **undefined**, **null**, **symbol**. Non-primitive: **object** (which includes arrays and functions). Everything that is not a primitive is an object.

### 2. What is the difference between **var**, **let**, and **const**?

**var** is function-scoped, hoisted, and can be redeclared. **let** is block-scoped and cannot be redeclared in the same scope. **const** is block-scoped, must be initialised, and cannot be reassigned — though object properties can still be mutated.

### 3. What is the difference between **==** and **===**?

**==** performs type coercion before comparing, so `"5" == 5` is true. **===** checks both type and value strictly, so `"5" === 5` is false. Always prefer **===** in production code.

### 4. What does **typeof null** return, and why?

It returns **"object"**. This is a well-known legacy bug in JavaScript dating back to its first version. `null` is not actually an object, but the bug was kept to avoid breaking existing code.

### 5. What is hoisting?

Variable and function declarations are moved to the top of their scope during the compilation phase. **var** is hoisted and initialised as `undefined`. **let** and **const** are hoisted but not initialised, creating a Temporal Dead Zone (TDZ) until the declaration line is reached.

### 6. What is a closure?

A closure is a function that retains access to variables from its outer (enclosing) scope even after that outer function has returned. Closures are fundamental to patterns like data privacy and factory functions.

```
function counter() {  
  let count = 0;  
  return () => ++count;  
}  
const increment = counter();  
increment(); // 1  
increment(); // 2
```

### 7. What is the difference between **null** and **undefined**?

**undefined** means a variable has been declared but not yet assigned a value. **null** is an intentional, explicit absence of a value assigned by the developer. `typeof undefined` is "undefined"; `typeof null` is "object".

## 8. What is the JavaScript event loop?

JavaScript is single-threaded. The event loop continuously checks the call stack; when it is empty, it picks the next task from the callback (macro-task) queue and pushes it onto the stack. This mechanism enables asynchronous, non-blocking behaviour.

## 9. What are Promises and how do they differ from callbacks?

A Promise is an object representing the eventual completion or failure of an async operation. Unlike callbacks, Promises avoid deeply nested "callback hell", support chaining via `.then()` and `.catch()`, and allow **Promise.all()** for concurrent operations.

## 10. What is `async/await`?

**async/await** is syntactic sugar built on top of Promises. An **async** function always returns a Promise. The **await** keyword pauses execution inside the async function until the Promise settles, making asynchronous code read like synchronous code.

# ■ Intermediate Level (Q11–Q25)

## 11. What does the `this` keyword refer to?

`this` refers to the execution context. In a method, it refers to the owning object. In a plain function, it is `window` in non-strict mode or `undefined` in strict mode. Arrow functions do not have their own `this` — they inherit it from the surrounding lexical scope.

## 12. What is prototypal inheritance?

In JavaScript, objects inherit properties and methods from their prototype via the internal `[[Prototype]]` chain. You can set up inheritance using `Object.create()`, class syntax, or by directly setting `__proto__`. Every function has a `prototype` property used when creating instances with `new`.

## 13. What is the difference between `call`, `apply`, and `bind`?

All three explicitly set the value of `this`. **`call(thisArg, arg1, arg2)`** invokes the function immediately with individual arguments. **`apply(thisArg, [args])`** does the same but takes an array of arguments. **`bind(thisArg)`** returns a new function permanently bound to `thisArg`, not invoked immediately.

## 14. What are `WeakMap` and `WeakSet`?

They hold **weak references** to objects, meaning their entries do not prevent garbage collection. Unlike `Map` and `Set`, their keys (`WeakMap`) or values (`WeakSet`) must be objects. They are non-iterable and ideal for caching without causing memory leaks.

## 15. What is debouncing vs throttling?

**Debounce** delays a function's execution until after a specified pause in events — useful for search inputs. **Throttle** limits a function to run at most once per specified time interval — useful for scroll or resize handlers.

## 16. What is a generator function?

A generator function (declared with `function*`) can pause its execution using **yield** and resume later. It returns an iterator object with a `.next()` method. Generators are useful for lazy evaluation and managing async flows.

```
function* gen() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const g = gen();  
g.next(); // { value: 1, done: false }
```

## 17. What is Symbol and what are its use cases?

Symbol is a primitive type that creates a **unique, immutable** identifier. It is used for unique object property keys (avoiding collisions), defining well-known behaviours like `Symbol.iterator` for custom iterators, and creating private-like properties on objects.

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

A **shallow copy** copies only the top-level properties; nested objects are still shared by reference (e.g., `spread { ...obj }`). A **deep copy** recursively duplicates all nested objects so they share no references (e.g., `structuredClone()` or `JSON.parse(JSON.stringify(obj))`).

## 19. What are microtasks vs macrotasks?

**Microtasks** (Promise callbacks, `queueMicrotask`) are processed immediately after the current task, before the browser renders or picks the next macrotask. **Macrotasks** (`setTimeout`, `setInterval`, I/O events) are queued and processed one per event loop tick.

## 20. What is memoization?

Memoization is an optimisation technique where a function caches the results of previous calls indexed by their inputs. On subsequent calls with the same arguments, the cached result is returned instead of recomputing, greatly improving performance for expensive pure functions.

## 21. What is the difference between `slice()` and `splice()`?

**`slice(start, end)`** returns a new array from `start` up to (not including) `end` without mutating the original.

**`splice(start, deleteCount, ...items)`** mutates the original array by removing and/or inserting elements in place.

## 22. What does `Array.from()` do?

`Array.from()` converts any array-like (e.g., arguments, NodeList) or iterable (e.g., Set, Map, string) object into a real Array. It optionally accepts a mapping function as the second argument.

```
Array.from('hello'); // ['h','e','l','l','o']
Array.from({length:3}, (_,i)=>i); // [0,1,2]
```

### 23. What is the difference between `forEach` and `map`?

`forEach` iterates over an array and always returns undefined — it is used for side effects. `map` iterates and returns a **new array** containing the transformed values. Use `map` when you need the resulting array; use `forEach` when you just need to execute something.

### 24. What is short-circuit evaluation?

Logical operators `&&` and `||` stop evaluating as soon as the result is certain. With `&&`, if the left side is falsy the right side is never evaluated. With `||`, if the left side is truthy the right side is skipped. This is used for guard clauses and default values.

### 25. What is the nullish coalescing operator `??`?

`??` returns the right-hand value only when the left side is **null or undefined**. Unlike `||`, it does not trigger on other falsy values such as 0, false, or an empty string, making it safer for default values when 0 or "" are valid inputs.

```
0 ?? "default"; // 0
0 || "default"; // "default"
```

## ■ Advanced Level (Q26–Q40)

### 26. What is optional chaining (`?.`)?

Optional chaining lets you safely access deeply nested object properties without throwing a `TypeError` if any reference in the chain is null or undefined. It short-circuits and returns undefined instead.

```
user?.address?.city; // undefined if user or address is nullish
user?.getAge?.(); // only calls if getAge exists
```

### 27. What is the difference between `Object.freeze()` and `Object.seal()`?

`Object.freeze()` prevents adding, removing, or modifying any properties — the object becomes completely immutable (shallowly). `Object.seal()` prevents adding or removing properties, but still allows modification of existing property values.

### 28. What are tagged template literals?

A tag is a function placed directly before a template literal. It receives an array of string parts and the interpolated values as separate arguments, giving full control over how the string is assembled. Used in libraries like styled-components and GraphQL query builders.

```
function highlight(strings, ...vals) {
  return strings.reduce((acc, str, i) =>
    acc + str + (vals[i] ? `<b>${vals[i]}</b>` : ''), '');
}
```

```
}  
highlight`Hello ${name}!`;
```

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

`for...in` iterates over the **enumerable keys** of an object (including inherited keys). `for...of` iterates over the **iterable values** of objects that implement the iterable protocol — such as arrays, strings, Maps, and Sets.

### 30. What is a pure function?

A pure function always produces the **same output for the same inputs** and has **no side effects** — it does not modify external state, make network calls, or mutate its arguments. Pure functions are predictable, testable, and safe to memoize.

### 31. What is `Object.assign()` and when would you use it?

`Object.assign(target, ...sources)` copies enumerable own properties from source objects into the target object (shallow). It is commonly used for shallow-merging objects or cloning. For deep merges, use `structuredClone()` or a library like `lodash`.

```
const config = Object.assign({}, defaults, userOptions);
```

### 32. What are truthy and falsy values?

Falsy values in JavaScript: **false, 0, -0, 0n, "", null, undefined, NaN**. Every other value is truthy. Understanding this is critical for writing correct conditionals and using short-circuit evaluation safely.

### 33. What is the difference between `localStorage`, `sessionStorage`, and cookies?

All three persist data in the browser. **localStorage** persists indefinitely until cleared by the user or code. **sessionStorage** is cleared when the browser tab closes. **Cookies** are sent to the server on every request, support expiry dates, and can be secured with `HttpOnly` and `Secure` flags.

### 34. What is the Temporal Dead Zone (TDZ)?

The TDZ is the period between when a `let` or `const` variable is hoisted to the top of its block and when its declaration line is actually executed. Accessing the variable during this window throws a `ReferenceError`. `var` does not have a TDZ.

### 35. How does garbage collection work in JavaScript?

JavaScript engines (like V8) use a **mark-and-sweep** algorithm. The GC starts from root objects (e.g., global, call stack) and marks all reachable objects. Anything not marked is considered unreachable and its memory is freed. This handles circular references that reference-counting cannot.

### 36. What is function currying?

Currying transforms a function that takes multiple arguments into a series of unary functions (each taking one argument). It enables partial application — you can pre-fill some arguments and reuse the resulting specialised function.

```
const add = a => b => a + b;
const add5 = add(5);
add5(3); // 8
add5(10); // 15
```

### 37. What are Proxy and Reflect?

**Proxy** wraps an object and lets you intercept and redefine fundamental operations (get, set, delete, has, etc.) via handler traps. **Reflect** provides the default implementations of those operations, commonly used inside Proxy traps to preserve default behaviour while adding custom logic.

```
const proxy = new Proxy(obj, {
  get(target, key) {
    return key in target ? target[key] : 'N/A';
  }
});
```

### 38. What is event delegation and why is it useful?

Instead of attaching event listeners to each child element, you attach a single listener to a common parent and use **event.target** to determine which child triggered the event. This reduces memory usage and automatically handles dynamically added child elements.

### 39. What is the difference between `setTimeout(fn, 0)` and a resolved Promise?

Both schedule async work, but they use different queues. A resolved Promise callback is a **microtask** and runs before the next macrotask. `setTimeout(fn, 0)` schedules a **macrotask**. So `Promise.resolve().then(fn)` always executes before `setTimeout(fn, 0)`, even with a 0ms delay.

### 40. What are Web Workers?

Web Workers run JavaScript in a separate background thread, completely independent of the main UI thread. They allow CPU-intensive computations without blocking user interaction. Workers cannot directly access the DOM; they communicate with the main thread via **postMessage()** and the **message** event.