

JavaScript Interview Questions and Answers

Intern to senior level

JavaScript interview questions and answers from beginner to senior: var vs let vs const, closures, hoisting, this, the event loop, promises, async/await, prototypes and more, with code examples.

13 questions Read online: <https://www.freedevtemplates.com/interview-questions/javascript>

1. What is the difference between var, let and const in JavaScript?

Level: intern JavaScript

var is function-scoped and can be redeclared; let and const are block-scoped, and const cannot be reassigned after it is set.

var is function-scoped, is hoisted and initialised to undefined, and can be redeclared in the same scope. let and const are block-scoped: they only exist inside the nearest pair of curly braces.

let and const are also hoisted, but they stay in the temporal dead zone until the line where they are declared, so using them earlier throws a ReferenceError instead of silently giving undefined.

const prevents reassigning the variable, not mutating the value. A const array or object can still be changed by pushing to it or setting a property. Use const by default, let when you need to reassign, and avoid var in new code.

Key points

var: function scope, hoisted as undefined, can be redeclared

let: block scope, temporal dead zone, cannot be redeclared in the same scope

const: like let, but the binding cannot be reassigned

```
if (true) {  
  var a = 1;  
  let b = 2;  
}  
console.log(a); // 1 (var ignores the block)  
console.log(b); // ReferenceError: b is not defined  
  
const list = [1, 2];  
list.push(3); // allowed, the array is mutated  
list = []; // TypeError: Assignment to constant variable
```

2. What is the difference between == and === in JavaScript?

Level: intern JavaScript

== compares values after type coercion, while === compares both value and type with no conversion, so it is the safer default.

The loose equality operator == converts the operands to a common type before comparing, which leads to surprising results such as 0 == "" and null == undefined both being true. The strict equality operator === returns false immediately if the types differ.

Most teams enforce === with a lint rule. The one commonly accepted use of == is value == null, which checks for both null and undefined in one expression.

Neither operator compares objects by content. Two separate objects or arrays with identical contents are never equal, because both operators compare references.

Key points

=== checks type and value, == coerces first
Prefer === everywhere except deliberate null checks
Objects are compared by reference, not by content

```
0 == ""; // true (both coerced to 0)
0 === ""; // false (number vs string)
null == undefined; // true
null === undefined; // false
NaN === NaN; // false (use Number.isNaN)
[1] === [1]; // false (different references)
```

3. What is hoisting in JavaScript?

Level: intern JavaScript

Hoisting is JavaScript moving declarations to the top of their scope during compilation, so functions and var variables can be referenced before the line where they are written.

Before running code, the engine processes declarations in each scope. Function declarations are hoisted with their full body, so you can call them before they appear. var variables are hoisted and set to undefined, but their assignment stays where you wrote it.

let, const and class declarations are hoisted too, but not initialised. Accessing them before the declaration throws a ReferenceError (the temporal dead zone). Function expressions assigned to a variable follow the variable's rules, not the function declaration rules.

Key points

Declarations are hoisted, assignments are not

Function declarations are available before their definition

let and const exist but cannot be used until declared

```
sayHi(); // "hi" - function declaration is fully hoisted
console.log(x); // undefined - var is hoisted, value is not
console.log(y); // ReferenceError - let is in the temporal dead zone

function sayHi() { console.log("hi"); }
var x = 5;
let y = 10;

notYet(); // TypeError: notYet is not a function
var notYet = function () {};
```

4. What is the difference between null and undefined in JavaScript?

Level: intern JavaScript

undefined means a variable has been declared but no value was assigned; null is an explicit value meaning 'no value' that you assign on purpose.

JavaScript uses undefined automatically: for uninitialised variables, missing function arguments, missing object properties and functions with no return statement. null is something a developer sets deliberately to say that a value is intentionally empty.

typeof undefined is 'undefined', while typeof null is 'object', a long-standing bug in the language that cannot be fixed without breaking the web. Use value == null to test for both, or optional chaining and nullish coalescing (user?.name ?? 'Guest') to handle them safely.

```
let a; // undefined
const b = null; // null
typeof a; // "undefined"
typeof b; // "object" (historical bug)
a == b; // true
a === b; // false
const name = user?.name ?? "Guest";
```

5. What is the difference between map, filter and reduce in JavaScript?

Level: intern JavaScript

map transforms every item into a new array of the same length, filter keeps only items that pass a test, and reduce folds the whole array into a single value.

All three are array methods that do not mutate the original array (when your callback is pure). map returns an array with the same number of items. filter returns a shorter or equal array. reduce takes an accumulator and returns anything: a number, an object, even another array.

They chain well: filter first to reduce the work, then map, then reduce. Forgetting the initial value in reduce is a common bug: on an empty array it throws a `TypeError`.

Key points

map: one output per input

filter: keeps items where the callback returns truthy

reduce: always pass an initial value

```
const orders = [{ price: 20, paid: true }, { price: 35, paid: false }, { price: 15, paid: true }];

const prices = orders.map(o => o.price); // [20, 35, 15]
const paid = orders.filter(o => o.paid); // 2 orders
const total = orders
  .filter(o => o.paid)
  .reduce((sum, o) => sum + o.price, 0); // 35
```

6. What is a closure in JavaScript and why is it useful?

Level: junior JavaScript

A closure is a function that keeps access to the variables of the scope where it was created, even after that outer function has finished running.

Every function in JavaScript carries a reference to the lexical environment it was defined in. When an inner function is returned or passed elsewhere, that environment stays alive for as long as the function can be called, so the inner function can still read and update the outer variables.

Closures give you private state without classes, power patterns such as memoization, debounce and once-only functions, and make callbacks and React hooks work. The trade-off is memory: anything captured by a closure cannot be garbage collected while the closure is reachable.

Key points

A closure = a function plus its surrounding scope

Used for private state, factories, callbacks and memoization

Captured variables stay in memory while the closure is reachable

```
function createCounter() {
  let count = 0; // private, not reachable from outside
  return function () {
    count += 1;
    return count;
  };
}

const next = createCounter();
next(); // 1
next(); // 2
```

7. How does the this keyword work in JavaScript?

Level: junior JavaScript

this is decided by how a function is called, not where it is written: as a method it is the object before the dot, with new it is the new object, and arrow functions inherit it from the enclosing scope.

For a normal function there are four rules, applied in this order: new Foo() makes this the newly created object; call, apply or bind set it explicitly; obj.method() makes this the object before the dot; otherwise it is undefined in strict mode (or the global object in sloppy mode).

Arrow functions do not have their own this. They use the this of the scope they were defined in, which is why they are handy for callbacks inside methods. The classic bug is passing obj.method as a callback: the call site changes, so this is lost.

Key points

this depends on the call site

call, apply and bind set this explicitly

Arrow functions capture this lexically and cannot be rebound

```
const user = {
  name: "Asha",
  regular() { return this.name; },
  arrow: () => typeof this, // "this" is NOT user here
};

user.regular(); // "Asha"
const detached = user.regular;
detached(); // undefined (or TypeError) - lost context
detached.call({ name: "Ravi" }) // "Ravi"
user.regular.bind(user)(); // "Asha"
```

8. What is the difference between promises and async/await in JavaScript?

Level: junior JavaScript

async/await is syntax built on top of promises that lets you write asynchronous code that reads like synchronous code; under the hood every async function returns a promise.

A promise represents a value that will be available later and is in one of three states: pending, fulfilled or rejected. You consume it with .then() and .catch(). async/await lets you pause an async function with await until a promise settles, and handle errors with ordinary try/catch.

They are equivalent in power, but await makes sequential logic and error handling much easier to read. A common mistake is awaiting independent calls one after another. Use Promise.all to run them in parallel.

Key points

async functions always return a promise

await only pauses the async function, not the whole thread

Use Promise.all for independent work

```
// promises
fetch("/api/user").then(r => r.json()).then(showUser).catch(handleError);

// async/await
async function load() {
  try {
    const res = await fetch("/api/user");
    showUser(await res.json());
  } catch (err) {
    handleError(err);
  }
}

// run independent calls in parallel
const [user, posts] = await Promise.all([getUser(), getPosts()]);
```

9. What is event delegation in JavaScript?

Level: junior JavaScript

Event delegation means attaching one listener to a parent element and using event bubbling to handle events from its children, instead of adding a listener to each child.

When you click a child element, the event bubbles up through its ancestors. A single listener on the parent can inspect `event.target` (or use `event.target.closest(selector)`) to find which child was clicked and react accordingly.

This uses less memory for long lists and also works for elements added later, because the listener lives on the parent. It does not work for events that do not bubble, such as focus and blur (use `focusin` and `focusout` instead).

```
document.querySelector("#todo-list").addEventListener("click", (event) => {
  const item = event.target.closest("li[data-id]");
  if (!item) return;
  removeTodo(item.dataset.id);
});
```

10. What is the event loop in JavaScript and how does it work?

Level: mid JavaScript

The event loop is the mechanism that lets single-threaded JavaScript handle asynchronous work: it runs the call stack to empty, then runs all queued microtasks (promises), then takes the next task (timers, events) from the queue.

JavaScript runs one piece of code at a time on the call stack. Async operations such as timers, network requests and DOM events are handled by the browser (or Node.js), which places a callback in a queue when they finish. The event loop waits until the call stack is empty and then moves the next callback onto it.

There are two queues that matter. The macrotask queue holds timers, I/O and UI events. The microtask queue holds promise callbacks and `queueMicrotask`. After every task, the engine drains the entire microtask queue before taking the next macrotask, which is why a resolved promise runs before a `setTimeout` of 0.

A long synchronous task blocks everything, including rendering, because nothing else can run until the stack is empty. Splitting heavy work into chunks, using web workers, or yielding to the event loop keeps the page responsive.

Key points

One call stack, many queues

Microtasks (promises) run before the next macrotask (`setTimeout`)

Blocking the stack blocks rendering and input

Synchronous code first, then microtasks, then timers.

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
// Output: 1, 4, 3, 2
```

11. What is prototypal inheritance in JavaScript?

Level: mid JavaScript

In JavaScript every object has an internal link to another object called its prototype, and property lookups walk up this prototype chain until the property is found or the chain ends.

When you read `obj.name`, the engine checks `obj` first, then `obj`'s prototype, then that object's prototype, and so on until it reaches `null`. Methods shared by all instances live on the prototype instead of being copied onto each object, which saves memory.

The `class` keyword is syntax over this mechanism. `class B extends A` sets up `B.prototype` to inherit from `A.prototype`. You can create an object with a chosen prototype using `Object.create(proto)`, and inspect it with `Object.getPrototypeOf(obj)`.

Key points

Lookups follow the prototype chain

`class` is syntactic sugar over prototypes

Shared methods live on the prototype, not on each instance

```
const animal = { speak() { return this.name + " makes a sound"; } };
const dog = Object.create(animal);
dog.name = "Rex";
dog.speak(); // "Rex makes a sound"
Object.getPrototypeOf(dog) === animal; // true

class Cat {
  constructor(name) { this.name = name; }
  speak() { return this.name + " meows"; } // lives on Cat.prototype
}
```

12. What is the difference between debounce and throttle in JavaScript?

Level: mid JavaScript

Debounce waits until the user stops triggering an event for a set time before running the function; throttle runs the function at most once per time interval no matter how often the event fires.

Use `debounce` when only the final value matters, such as a search box that should call the API after the user pauses typing. Use `throttle` when you want regular updates during continuous activity, such as scroll position tracking or window resize handling.

Both are built with closures and timers. In `React`, make sure the debounced function is created once (for example with `useMemo` or `useRef`), otherwise every render creates a new timer and the debounce does nothing.

Key points

Debounce: run after the burst ends

Throttle: run at a steady maximum rate

Both rely on closures to keep the timer between calls

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}

function throttle(fn, limit) {
  let waiting = false;
  return (...args) => {
    if (waiting) return;
    fn(...args);
    waiting = true;
    setTimeout(() => { waiting = false; }, limit);
  };
}
```

13. How do you find and fix memory leaks in JavaScript?

Level: senior JavaScript

Take heap snapshots in Chrome DevTools before and after the suspect action, compare what grew, and fix the retaining reference, which is usually a forgotten listener, timer, global variable or unbounded cache.

JavaScript uses garbage collection based on reachability: an object is freed only when nothing reachable still references it. A leak is memory that your app no longer needs but something still references.

The usual causes are event listeners that are never removed, setInterval callbacks that are never cleared, detached DOM nodes held in a variable, accidental globals, closures that capture large objects, and caches or maps that only grow. In React, a missing cleanup function in useEffect is the most common source.

To diagnose, open the DevTools Memory tab, take a snapshot, repeat the action several times, take another and use the comparison view to see which object types keep growing. The retainers panel shows the chain of references keeping an object alive. Fixes are to remove listeners and timers in cleanup code, use WeakMap or WeakRef for caches keyed by objects, and put a size limit or eviction policy on caches.

Key points

Leak = unneeded but still reachable

Compare heap snapshots to see what keeps growing

Always pair listeners and timers with cleanup

```
useEffect(() => {
  const onResize = () => setWidth(window.innerWidth);
  window.addEventListener("resize", onResize);
  const id = setInterval(refresh, 5000);

  return () => { // cleanup prevents the leak
    window.removeEventListener("resize", onResize);
    clearInterval(id);
  };
}, []);
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/javascript>