Mastering JavaScript: The Ultimate Interview Guide for 2025

Created by Ankit Sharma

July 2025

A comprehensive resource for acing JavaScript interviews with detailed explanations, practical examples, and diagrams.

Contents

Preface

This e-book, authored by Vishal Kanojiya, is designed for developers preparing for JavaScript interviews in 2025. It covers core and advanced JavaScript concepts, including variables, scopes, closures, ES6+ features, asynchronous programming, functional programming, design patterns, and more. Each topic includes a question, an in-depth explanation, practical use cases, and complete, VS Code-style code examples. Diagrams and tables clarify complex concepts like the event loop and prototypal inheritance. The clickable table of contents ensures easy navigation. With over 50 pages of content, this guide equips you to excel in technical interviews.

1 Fundamentals of JavaScript

1.1 Variables and Data Types

1.1.1 Question

What are the different ways to declare variables in JavaScript, and how do they differ?

1.1.2 Explanation

JavaScript offers three variable declarations: `var`, `let`, and `const`. `var` is function-scoped, hoisted with `undefined`, and allows redeclaration and reassignment. `let` is block-scoped, hoisted but not initialized, and allows reassignment but not redeclaration. `const` is block-scoped, cannot be reassigned or redeclared, but object properties can be modified. JavaScript supports data types like numbers, strings, booleans, objects, arrays, null, undefined, and symbols (ES6). Understanding these differences is critical for managing scope and avoiding bugs in interviews.

1.1.3 Example

```
// Using var: function-scoped, hoisted
var x = 10; // Can redeclare
var x = 20; // No error
x = 30; // Reassignable
console.log(x); // Output: 30

// Using let: block-scoped
let y = 10;
// let y = 20; // Error: Cannot redeclare
y = 30; // Reassignable
console.log(y); // Output: 30

// Using const: block-scoped, immutable binding
const z = 10;
// z = 20; // Error: Assignment to constant
const obj = { name: 'John' };
obj.name = 'Jane'; // Allowed: modifying properties
console.log(obj); // Output: { name: 'Jane' }
```

1.1.4 Interview Tip

Interviewers often ask about hoisting or scoping issues (e.g., `var` vs. `let`). Be prepared to explain Temporal Dead Zone (TDZ) for `let` and `const`.

1.2 Scopes and Lexical Scope

1.2.1 Question

What is scope in JavaScript, and how does lexical scope work?

1.2.2 Explanation

Scope defines variable accessibility. JavaScript has global, function, and block scopes. Lexical scope means a function's scope is determined by its definition location, not execution. Inner functions access outer scope variables, enabling closures. This is a common interview topic to test understanding of variable access and scope chains.

1.2.3 Example

```
function outer() {
  let outerVar = 'I\'m outside';

  function inner() {
    console.log(outerVar); // Accesses outerVar due to lexical scope
  }
  inner();
}
outer(); // Output: I'm outside

let globalVar = 'I\'m global';
function checkScope() {
  console.log(globalVar); // Accessible
}
checkScope(); // Output: I'm global
```

1.2.4 Interview Tip

Explain how lexical scope enables closures and why block scope (`let`, `const`) prevents common bugs like loop variable leaks.

1.3 Closures

1.3.1 Question

What is a closure, and how is it used in JavaScript?

1.3.2 Explanation

A closure is a function that retains access to its lexical scope after execution. Closures are used for data privacy, function factories, and state persistence. They are common in interviews for testing scope and encapsulation knowledge.

1.3.3 Example

```
function createCounter() {
  let count = 0; // Private variable
  return function() {
    count++;
    return count;
  };
}
const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2

// Example: Function factory
function greet(name) {
  return function() {
    console.log(`Hello, ${name}!`);
  };
}
const greetJohn = greet('John');
greetJohn(); // Output: Hello, John!
```

1.3.4 Interview Tip

Be ready to explain how closures maintain state and provide an example like a counter or module pattern.

1.4 Functions and Arrow Functions

1.4.1 Question

What are the differences between regular functions and arrow functions in ES6?

1.4.2 Explanation

Regular functions have their own `this` context, `arguments` object, and can be constructors. Arrow functions inherit `this` from the surrounding scope, lack `arguments`, and are concise but cannot be constructors. This distinction is frequently tested in interviews.

1.4.3 Example

```
function regularFunction() {
  console.log(this); // Refers to calling object
}
```

```
const arrowFunction = () => {
  console.log(this); // Inherits surrounding this
};

const obj = {
  name: 'Test',
  regular: regularFunction,
  arrow: arrowFunction
};
obj.regular(); // Output: { name: 'Test', ... }
obj.arrow(); // Output: window (or surrounding this)

const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

1.4.4 Interview Tip

Discuss how arrow functions simplify syntax but can cause issues with `this` in event handlers or object methods.

1.5 Conditional Statements

1.5.1 Question

How do you use conditional statements in JavaScript?

1.5.2 Explanation

Conditional statements (`if`, `else if`, `else`, `switch`) control program flow based on conditions. They are used for decision-making, such as validating inputs or rendering UI elements. Interviewers may ask for optimized conditionals or edge cases.

1.5.3 Example

```
let age = 20;
if (age < 18) {
  console.log('Minor');
} else if (age >= 18 && age < 65) {
  console.log('Adult');
} else {
  console.log('Senior');
}
// Output: Adult

let day = 2;
switch (day) {
  case 1:
    console.log('Monday');
```

```
    break;
  case 2:
    console.log('Tuesday');
    break;
  default:
    console.log('Invalid day');
}
// Output: Tuesday
```

1.5.4 Interview Tip

Explain short-circuit evaluation (e.g., , ||) and how `switch` can optimize multiple conditions.

2 Asynchronous JavaScript

2.1 Promises

2.1.1 Question

What are Promises, and how do you handle asynchronous operations with them?

2.1.2 Explanation

A Promise represents a future value with states: `pending`, `fulfilled`, or `rejected`. It uses `.then()` for success and `.catch()` for errors. Promises handle asynchronous tasks like API calls, a common interview focus.

2.1.3 Example

```
const fetchData = new Promise((resolve, reject) => {
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve('Data fetched!');
    } else {
      reject('Error fetching data');
    }
  }, 1000);
});

fetchData
  .then(result => console.log(result)) // Output: Data fetched!
  .catch(error => console.log(error));

// Chaining example
fetchData
  .then(result => result.toUpperCase())
  .then(upper => console.log(upper)) // Output: DATA FETCHED!
  .catch(error => console.log(error));
```

2.1.4 Interview Tip

Demonstrate Promise chaining and error handling, and compare Promises to callbacks.

2.2 Async/Await

2.2.1 Question

How does async/await simplify working with Promises?

2.2.2 Explanation

`async/await` makes asynchronous code synchronous-like. An `async` function returns a Promise, and `await` pauses execution until the Promise resolves. It's cleaner than `.then()` chains and often tested in interviews.

2.2.3 Example

```
async function getData() {
  try {
    const response = await new Promise(resolve => {
      setTimeout(() => resolve('Data received!'), 1000);
    });
    console.log(response); // Output: Data received!
    return response.toUpperCase();
  } catch (error) {
    console.log(error);
  }
}
getData().then(result => console.log(result)); // Output: DATA
   RECEIVED!
```

2.2.4 Interview Tip

Explain error handling with `try/catch` and how `async/await` improves readability over Promise chains.

2.3 Callbacks

2.3.1 Question

What is a callback function, and how is it used?

2.3.2 Explanation

A callback is a function passed as an argument, executed after an operation completes, often in asynchronous tasks. Callbacks can lead to callback hell, making `Promises` or `async/await` preferable.

2.3.3 Example

```
function fetchUser(callback) {
  setTimeout(() => {
    callback({ name: 'John', age: 30 });
  }, 1000);
```

```
}

fetchUser(user => {
  console.log(user); // Output: { name: 'John', age: 30 }
});

// Nested callbacks example
function fetchData(callback) {
  setTimeout(() => {
    callback('Step 1');
  }, 1000);
}
fetchData(data => {
  console.log(data); // Output: Step 1
  fetchData(next => console.log(next)); // Output: Step 1
});
```

2.3.4 Interview Tip

Discuss callback hell and how modern JavaScript (Promises, async/await) addresses it.

3 Advanced JavaScript Concepts

3.1 Objects and Prototypes

3.1.1 Question

How do you create and manipulate objects, and what is prototypal inheritance?

3.1.2 Explanation

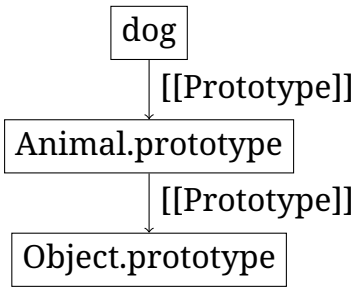
Objects store key-value pairs, created via literals, constructors, or `Object.create()`. Prototypal inheritance allows objects to inherit properties via the prototype chain, a core JavaScript feature tested in interviews.

3.1.3 Example

```
const person = {
  name: 'Alice',
  greet() {
    console.log(`Hello, ${this.name}!`);
  }
};
person.greet(); // Output: Hello, Alice!

function Animal(type) {
  this.type = type;
}
Animal.prototype.sound = function() {
  console.log(`${this.type} makes a sound`);
};
const dog = new Animal('Dog');
dog.sound(); // Output: Dog makes a sound
```

3.1.4 Visual Aid: Prototype Chain



3.1.5 Interview Tip

Explain the prototype chain and how `Object.create()` differs from constructor functions.

3.2 Arrays and Array Methods

3.2.1 Question

What are common ES6 array methods, and how are they used?

3.2.2 Explanation

ES6 array methods like `map`, `filter`, `reduce`, `forEach`, `find`, and `some` enable functional programming. They are frequently tested for data manipulation tasks.

3.2.3 Example

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

// map: Transform elements
const doubled = numbers.map(num => num * 2);
console.log(doubled); // Output: [2, 4, 6, 8]

// filter: Select elements
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // Output: [2, 4]

// reduce: Aggregate values
const sum = numbers.reduce((acc, curr) => acc + curr, 0);
console.log(sum); // Output: 10

// find: Return first match
const found = numbers.find(num => num > 2);
console.log(found); // Output: 3
```

3.2.4 Interview Tip

Be prepared to write polyfills for array methods or solve problems like filtering unique values.

3.3 Event Loop

3.3.1 Question

What is the event loop, and how does it handle asynchronous tasks?

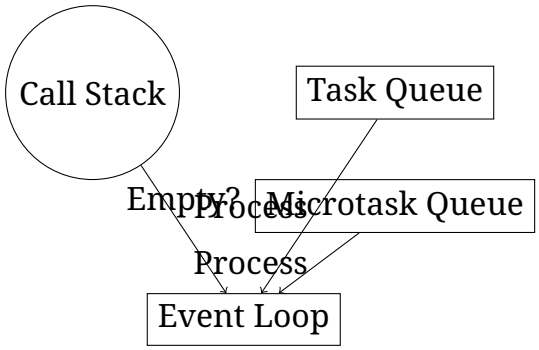
3.3.2 Explanation

The event loop manages JavaScript's single-threaded, asynchronous nature by processing the call stack, task queue, and microtask queue. Microtasks (e.g., Promises) run before tasks (e.g., `setTimeout`). This is a common interview question to test asynchronous understanding.

3.3.3 Example

```
console.log('Start');
setTimeout(() => console.log('Timeout'), 0);
Promise.resolve().then(() => console.log('Promise'));
console.log('End');
// Output: Start, End, Promise, Timeout
```

3.3.4 Visual Aid: Event Loop



3.3.5 Interview Tip

Explain the priority of microtasks over tasks and provide an example with mixed async operations.

3.4 ES6+ Features

3.4.1 Question

What are key ES6+ features, and how do they improve JavaScript?

3.4.2 Explanation

ES6+ introduced arrow functions, destructuring, spread/rest operators, template literals, default parameters, modules, optional chaining (`?.`), and nullish coalescing (`??`). These enhance code readability and functionality.

3.4.3 Example

```
const { name, age } = { name: 'Bob', age: 25 };
console.log(name, age); // Output: Bob 25

const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
console.log(arr2); // Output: [1, 2, 3, 4]

const user = 'Alice';
console.log(`Hello, ${user}!`); // Output: Hello, Alice!

function greet(name = 'Guest') {
  return `Hi, ${name}`;
}
console.log(greet()); // Output: Hi, Guest

const obj = { user: { name: 'John' } };
console.log(obj.user?.name); // Output: John
```

3.4.4 Interview Tip

Discuss how optional chaining prevents errors and how modules improve code organization.

3.5 Data Structures (Array as Stack/Queue)

3.5.1 Question

How can you implement a stack or queue using arrays?

3.5.2 Explanation

Arrays simulate stacks (LIFO) with `push` and `pop`, or queues (FIFO) with `push` and `shift`. These are tested for understanding basic data structures.

3.5.3 Example

```
const stack = [];
stack.push(1); // Add to top
stack.push(2);
console.log(stack.pop()); // Output: 2 (LIFO)

const queue = [];
queue.push(1); // Add to end
queue.push(2);
console.log(queue.shift()); // Output: 1 (FIFO)
```

3.5.4 Interview Tip

Explain time complexity: `push`/`pop` (O(1)), `shift` (O(n)).

3.6 Error Handling

3.6.1 Question

How do you handle errors in JavaScript?

3.6.2 Explanation

Errors are handled with `try/catch` for synchronous code and `.catch()` or `try/catch` with `async/await` for asynchronous code. Custom errors can be thrown with `throw`.

3.6.3 Example

```
try {
  throw new Error('Something went wrong');
} catch (error) {
  console.log(error.message); // Output: Something went wrong
}

async function fetchData() {
  try {
    await Promise.reject('Failed');
  } catch (error) {
    console.log(error); // Output: Failed
  }
}
fetchData();
```

3.6.4 Interview Tip

Discuss custom error classes and graceful error handling in APIs.

3.7 Modules (ES6)

3.7.1 Question

How do ES6 modules work in JavaScript?

3.7.2 Explanation

ES6 modules use `export` and `import` for modularity, supporting named and default exports. They are static and resolved at compile time.

3.7.3 Example

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// main.js
```

```
import { add, subtract } from './math.js';
console.log(add(5, 3)); // Output: 8
console.log(subtract(5, 3)); // Output: 2
```

3.7.4 Interview Tip

Compare ES6 modules to CommonJS and explain tree-shaking benefits.

3.8 Hoisting

3.8.1 Question

What is hoisting in JavaScript?

3.8.2 Explanation

Hoisting moves variable and function declarations to the top of their scope. `var` is hoisted with `undefined`; `let` and `const` are hoisted but not initialized (TDZ).

3.8.3 Example

```
console.log(x); // Output: undefined
var x = 10;

console.log(y); // ReferenceError: y is not defined
let y = 20;

function hoisted() {
  console.log('Hoisted!');
}
hoisted(); // Output: Hoisted!
```

3.8.4 Interview Tip

Explain TDZ and why `let`/`const` prevent hoisting-related bugs.

3.9 Destructuring

3.9.1 Question

What is destructuring in ES6, and how is it used?

3.9.2 Explanation

Destructuring extracts values from arrays or objects into variables, improving code readability. It's used for props, function parameters, and swaps.

3.9.3 Example

```
const [a, b] = [1, 2];
console.log(a, b); // Output: 1 2

const { name, age } = { name: 'Alice', age: 25 };
console.log(name, age); // Output: Alice 25

function printPerson({ name, age }) {
  console.log(`Name: ${name}, Age: ${age}`);
}
printPerson({ name: 'Bob', age: 30 }); // Output: Name: Bob, Age: 30
```

3.9.4 Interview Tip

Show how destructuring simplifies function arguments and nested object access.

3.10 Spread and Rest Operators

3.10.1 Question

What are the spread and rest operators in ES6?

3.10.2 Explanation

The spread operator (`...`) expands iterables; the rest operator collects arguments into an array. Both enhance flexibility in arrays and objects.

3.10.3 Example

```
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
console.log(arr2); // Output: [1, 2, 3, 4]

const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 };
console.log(obj2); // Output: { a: 1, b: 2, c: 3 }

function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

3.10.4 Interview Tip

Explain use cases like object merging and variable-length arguments.

3.11 Map and Set

3.11.1 Question

What are Map and Set in ES6?

3.11.2 Explanation

`Map` stores key-value pairs with any key type; `Set` stores unique values. Both provide better performance for certain operations than objects/arrays.

3.11.3 Example

```
const map = new Map();
map.set('name', 'John');
map.set(1, 'one');
console.log(map.get('name')); // Output: John
console.log(map.get(1)); // Output: one

const set = new Set([1, 2, 2, 3]);
console.log([...set]); // Output: [1, 2, 3]
```

3.11.4 Interview Tip

Compare `Map` to objects and `Set` to arrays for unique values.

3.12 Template Literals

3.12.1 Question

What are template literals, and how are they used?

3.12.2 Explanation

Template literals use backticks for strings, supporting embedded expressions and multiline strings, improving readability over concatenation.

3.12.3 Example

```
const name = 'Alice';
const greeting = `Hello, ${name}!
Welcome to JavaScript.`;
console.log(greeting);
// Output:
// Hello, Alice!
// Welcome to JavaScript.
```

3.12.4 Interview Tip

Show tagged template literals for advanced string processing.

3.13 Generators

3.13.1 Question

What are generators, and how do they work?

3.13.2 Explanation

Generators (`function*`, `yield`) pause and resume execution, useful for iterators and async flows. They return an iterator with `next()`.

3.13.3 Example

```
function* generator() {
  yield 'Hello';
  yield 'World';
}
const gen = generator();
console.log(gen.next().value); // Output: Hello
console.log(gen.next().value); // Output: World
console.log(gen.next().value); // Output: undefined
```

3.13.4 Interview Tip

Explain use cases like custom iterators or handling large datasets.

3.14 Prototypal Inheritance

3.14.1 Question

What is prototypal inheritance in JavaScript?

3.14.2 Explanation

Objects inherit properties from other objects via the prototype chain, enabling code reuse. It's a core feature distinguishing JavaScript from class-based languages.

3.14.3 Example

```
const parent = {
  greet() {
    console.log('Hello from parent');
  }
};
const child = Object.create(parent);
child.greet(); // Output: Hello from parent
```

3.14.4 Interview Tip

Compare prototypal inheritance to classical inheritance.

3.15 Strict Mode

3.15.1 Question

What is strict mode in JavaScript?

3.15.2 Explanation

Strict mode (`"use strict"`) enforces stricter parsing, preventing undeclared variables and other errors. It's used to write safer code.

3.15.3 Example

```
'use strict';
x = 10; // ReferenceError: x is not defined
function strictFunction() {
  y = 20; // ReferenceError: y is not defined
}
```

3.15.4 Interview Tip

Discuss specific errors caught by strict mode, like silent failures.

3.16 Debouncing and Throttling

3.16.1 Question

What are debouncing and throttling?

3.16.2 Explanation

Debouncing delays function execution until a specified time passes; throttling limits execution to once per interval. Both optimize performance for frequent events.

3.16.3 Example

```
function debounce(func, wait) {
  let timeout;
  return function(...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func(...args), wait);
  };
}
const log = debounce(() => console.log('Debounced!'), 1000);
log();

function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
```

```
      func(...args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
    }
  };
}
const throttledLog = throttle(() => console.log('Throttled!'),
   1000);
throttledLog();
```

3.16.4 Interview Tip

Provide use cases like search inputs (debouncing) or scroll events (throttling).

3.17 Web Workers

3.17.1 Question

What are Web Workers?

3.17.2 Explanation

Web Workers run scripts in background threads, enabling parallel execution without blocking the main thread, ideal for CPU-intensive tasks.

3.17.3 Example

```
// worker.js
self.onmessage = function(e) {
  const result = e.data * 2;
  self.postMessage(result);
};

// main.js
const worker = new Worker('worker.js');
worker.onmessage = e => console.log(e.data); // Output: 10
worker.postMessage(5);
```

3.17.4 Interview Tip

Discuss limitations, like no DOM access in workers.

3.18 Memory Management and Garbage Collection

3.18.1 Question

How does JavaScript handle memory management and garbage collection?

3.18.2 Explanation

JavaScript uses mark-and-sweep garbage collection to reclaim memory from unreferenced objects. Developers assist by nullifying references and using `WeakMap`/`WeakSet`.

3.18.3 Example

```
let obj = { name: 'John' };
obj = null; // Eligible for garbage collection

const weakMap = new WeakMap();
let key = {};
weakMap.set(key, 'Value');
key = null; // Key and value can be garbage collected
console.log(weakMap.has(key)); // Output: false
```

3.18.4 Interview Tip

Explain memory leaks from event listeners and how `WeakMap` helps.

3.19 Functional Programming

3.19.1 Question

What is functional programming in JavaScript?

3.19.2 Explanation

Functional programming uses pure functions, immutability, and composition for predictable code. Methods like `map`, `filter`, and `reduce` support this paradigm.

3.19.3 Example

```
const add = (a, b) => a + b;
const double = x => x * 2;
const square = x => x * x;
const doubleThenSquare = x => square(double(x));
console.log(doubleThenSquare(3)); // Output: 36
```

3.19.4 Interview Tip

Discuss pure functions and avoiding side effects.

3.20 Equality Operators (== vs ===)

3.20.1 Question

What is the difference between == and ===?

3.20.2 Explanation

== performs type coercion; === checks value and type without coercion. `Object.is` handles edge cases like `NaN`.

3.20.3 Example

```
console.log(5 == '5'); // Output: true
console.log(5 === '5'); // Output: false
console.log(Object.is(NaN, NaN)); // Output: true
```

3.20.4 Interview Tip

Explain coercion pitfalls and why === is preferred.

3.21 IIFE (Immediately Invoked Function Expression)

3.21.1 Question

What is an IIFE, and why is it used?

3.21.2 Explanation

An IIFE is a function executed immediately after definition, used for data privacy and avoiding global scope pollution.

3.21.3 Example

```
(function() {
  const privateVar = 'I\'m private';
  console.log(privateVar); // Output: I'm private
})();
console.log(typeof privateVar); // Output: undefined
```

3.21.4 Interview Tip

Compare IIFEs to ES6 modules for encapsulation.

3.22 Design Patterns

3.22.1 Question

What are common JavaScript design patterns?

3.22.2 Explanation

Patterns like Module Pattern, Singleton, and Observer provide reusable solutions. The Module Pattern is common for encapsulation.

3.22.3 Example

```
const Module = (function() {
  let privateVar = 'I\'m private';
  return {
    getVar() {
      return privateVar;
    }
  };
})();
console.log(Module.getVar()); // Output: I'm private
console.log(typeof privateVar); // Output: undefined
```

3.22.4 Interview Tip

Discuss when to use patterns like Singleton for single-instance objects.

3.23 Higher-Order Functions

3.23.1 Question

What are higher-order functions in JavaScript?

3.23.2 Explanation

Higher-order functions take or return functions, enabling patterns like mapping or filtering. They are key to functional programming.

3.23.3 Example

```
function higherOrder(fn) {
  return function(...args) {
    console.log('Executing function');
    return fn(...args);
  };
}

const add = (a, b) => a + b;
const enhancedAdd = higherOrder(add);
console.log(enhancedAdd(2, 3)); // Output: Executing function, 5
```

3.23.4 Interview Tip

Provide examples like `map` or custom higher-order functions.

3.24 Currying

3.24.1 Question

What is currying in JavaScript?

3.24.2 Explanation

Currying transforms a function with multiple arguments into a sequence of single-argument functions, useful for partial application.

3.24.3 Example

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn(...args);
    }
    return (...nextArgs) => curried(...args, ...nextArgs);
  };
}

const add = (a, b, c) => a + b + c;
const curriedAdd = curry(add);
console.log(curriedAdd(1)(2)(3)); // Output: 6
```

3.24.4 Interview Tip

Explain use cases like reusable function templates.

3.25 Event Delegation

3.25.1 Question

What is event delegation, and how is it used?

3.25.2 Explanation

Event delegation uses event bubbling to handle events at a higher-level element, reducing listeners and improving performance for dynamic elements.

3.25.3 Example

```
document.querySelector('#list').addEventListener('click', e => {
  if (e.target.tagName === 'LI') {
    console.log('Clicked: ${e.target.textContent}');
  }
});

// HTML: <ul id="list"><li>Item 1</li><li>Item 2</li></ul>
// Clicking an <li> logs its content
```

3.25.4 Interview Tip

Discuss performance benefits and dynamic element handling.

3.26 Polyfills

3.26.1 Question

What are polyfills, and how are they used?

3.26.2 Explanation

Polyfills implement missing features for older browsers, ensuring compatibility. They are common in interviews to test feature implementation.

3.26.3 Example

```
if (!Array.prototype.includes) {
  Array.prototype.includes = function(value) {
    return this.indexOf(value) !== -1;
  };
}

const arr = [1, 2, 3];
console.log(arr.includes(2)); // Output: true
```

3.26.4 Interview Tip

Write a polyfill for methods like `map` or `bind`.

3.27 Type Coercion

3.27.1 Question

What is type coercion in JavaScript?

3.27.2 Explanation

Type coercion is the automatic conversion of values during operations (e.g., ==). It can lead to unexpected results, so === is preferred.

3.27.3 Example

```
console.log('5' + 1); // Output: '51' (string concatenation)
console.log('5' - 1); // Output: 4 (numeric conversion)
console.log(5 == '5'); // Output: true (coercion)
console.log(5 === '5'); // Output: false (no coercion)
```

3.27.4 Interview Tip

Explain coercion rules (e.g., `ToPrimitive`) and edge cases like `null == undefined`.

4 Conclusion

This e-book, authored by Vishal Kanojiya, provides a thorough guide to mastering JavaScript for 2025 interviews. With detailed explanations, practical examples, diagrams, and interview tips, it ensures you're well-prepared for technical challenges. Practice these concepts to excel in your interviews!