Frontend Interview JavaScript Questions - Quick Reference List

Table of Contents

Core JavaScript Concepts

   - `call`, `apply`, and `bind`

3. Scope and Closures - Questions 15-19
   - JavaScript scope types
   - Closures and their importance
   - Global, function, and block scope
   - Temporal dead zone
   - Variable shadowing

4. Objects and Arrays - Questions 20-25
   - Object creation and manipulation
   - Common array methods
   - Shallow vs deep copy
   - Object cloning techniques
   - Destructuring assignment
   - Spread operator and rest parameters

5. Prototypes and Inheritance - Questions 26-30
   - JavaScript prototypes
   - Prototypal inheritance
   - `__proto__` vs `prototype`
   - ES6 classes and prototypes
   - Prototype chain

Asynchronous Programming

6. Asynchronous JavaScript - Questions 31-35
   - Callbacks and callback hell

- Promises and their methods
- async/await syntax
- Event Loop mechanism
- Microtasks vs macrotasks

DOM and Browser APIs

7. Event Handling - Questions 36-40
    - JavaScript event handling
    - Event bubbling and capturing
    - Event delegation
    - Preventing default behavior
    - addEventListener vs onclick
8. DOM Manipulation - Questions 41-45
    - DOM element selection
    - Creating, modifying, removing elements
    - innerHTML vs textContent vs innerText
    - Form validation
    - document.ready vs window.onload

Modern JavaScript

9. ES6+ Features - Questions 46-52
    - Template literals
    - Default parameters
    - Modules and import/export
    - Sets and Maps

- Symbols
- Generators and iterators
- for...of vs for...in loops

Error Handling and Performance

10. Error Handling - Questions 53-56
    - JavaScript error handling
    - Throwing and catching errors
    - Error types in JavaScript
    - Custom error types
11. Performance and Optimization - Questions 57-61
    - Performance optimization techniques
    - Memory leak prevention
    - Debouncing and throttling
    - DOM optimization
    - Lazy loading

Advanced Topics

12. Advanced Concepts - Questions 62-67
    - Memoization
    - Design patterns (Singleton, Observer, Module)
    - Functional programming
    - Immutability
    - Pure functions
    - Recursion

13. Browser APIs - Questions 68-72
    - localStorage vs sessionStorage vs cookies
    - Fetch API
    - Web Workers
    - Geolocation API
    - History API
14. Security - Questions 73-76
    - Cross-Site Scripting (XSS)
    - Cross-Site Request Forgery (CSRF)
    - Input sanitization
    - Content Security Policy (CSP)
15. Testing - Questions 77-80
    - Unit testing in JavaScript
    - Test-driven development (TDD)
    - Mocks and stubs
    - Testing asynchronous code

Data Handling

16. Regular Expressions - Questions 81-84
    - Using regex in JavaScript
    - Common validation patterns
    - match, search, replace, test methods
17. JSON and Data Handling - Questions 85-88
    - Working with JSON

- JSON.parse() vs JSON.stringify()
- API response handling
- CORS handling

Modern JavaScript Tools

18. Modern JavaScript Tools - Questions 89-93
    - npm and package.json
    - Build tools (Webpack, Vite)
    - Babel transpiler
    - ESLint linting
    - Development vs production builds

Practical Applications

19. Miscellaneous - Questions 94-99
    - Synchronous vs asynchronous code
    - Array checking methods
    - typeof operator limitations
    - String to number conversion
    - slice, splice, split differences
    - Array and object sorting
20. Coding Challenges - Questions 100-109
    - String reversal
    - Palindrome checking
    - Array duplicate finding
    - Array flattening

    - Calculator implementation
    - Function debouncing
    - Promise implementation
    - Array method implementations

21. Framework-Agnostic Frontend Concepts - Questions 110-119
    - Virtual DOM
    - Component-based architecture
    - State management
    - Client-side routing
    - SSR vs CSR
    - Progressive Web Apps
    - Responsive design
    - Accessibility (a11y)
    - Micro-frontends
    - SPA vs MPA vs SSG

JavaScript Fundamentals

1. What is JavaScript and what are its key features?

JavaScript is a high-level, interpreted, dynamically-typed programming language that enables interactive web pages and is an essential part of web applications. Originally created for client-side scripting, it now runs on servers, mobile apps, and desktop applications.

Key Features:

- **Dynamic Typing**: Variables don't need explicit type declarations

- **First-class Functions**: Functions can be assigned to variables, passed as arguments, and returned from other functions
- **Prototype-based Object Orientation**: Objects can inherit directly from other objects
- **Event-driven Programming**: Responds to user interactions and system events
- **Interpreted Language**: No compilation step required
- **Weak Typing**: Automatic type conversion when needed
- **Multi-paradigm**: Supports procedural, object-oriented, and functional programming

Example:

```
// Dynamic typing
let variable = "Hello"; // String
variable = 42;          // Now a number
variable = true;        // Now a boolean

// First-class functions
const greet = function(name) {
    return `Hello, ${name}!`;
};

// Event-driven
document.addEventListener('click', function() {
    console.log('Page clicked!');
});
```

Related Questions: Data Types, Type Coercion

⬆️ Back to Top

---

2. What's the difference between `==` and `===`?

The main difference lies in **type checking** and **type coercion**:

`==` (Loose Equality/Abstract Equality):

- Performs type coercion before comparison
- Converts operands to the same type if they're different
- Can lead to unexpected results

`===` (Strict Equality):

- No type coercion
- Compares both value and type
- Recommended for most comparisons

Examples:

```
// Loose equality (==) - with type coercion
console.log(5 == '5');        // true (string '5' converted
console.log(true == 1);       // true (boolean converted to
console.log(false == 0);      // true
console.log(null == undefined); // true (special case)
console.log(0 == '');         // true (empty string convert

// Strict equality (===) - no type coercion
console.log(5 === '5');       // false (different types)
console.log(true === 1);      // false
console.log(false === 0);     // false
console.log(null === undefined); // false
console.log(0 === '');        // false

// Safe comparisons
const userInput = '42';
if (parseInt(userInput) === 42) {
```

```
    console.log('Valid number!');
}
```

Best Practice: Always use `===` unless you specifically need type coercion.

Related Questions: Type Coercion, Data Types

Back to Top

---

3. What are the different data types in JavaScript?

JavaScript has **8 data types** divided into two categories:

Primitive Types (7):

1. `undefined` - Variable declared but not assigned
2. `null` - Intentional absence of value
3. `boolean` - true or false
4. `number` - Integers and floating-point numbers
5. `string` - Text data
6. `symbol` - Unique identifiers (ES6+)
7. `bigint` - Large integers (ES2020+)

Non-Primitive Type (1):

8. `object` - Collections of key-value pairs (including arrays, functions, dates)

Examples:

```
// Primitive types
let undefinedVar;                    // undefined
let nullVar = null;                  // null
let boolVar = true;                  // boolean
let numVar = 42;                     // number
```

```
let floatVar = 3.14;                    // number
let strVar = "Hello World";             // string
let symVar = Symbol('id');              // symbol
let bigIntVar = 12345678901234567890123456789n; // bigint

// Non-primitive type
let objVar = { name: "John", age: 30 }; // object
let arrVar = [1, 2, 3];                 // object (array)
let funcVar = function() {};            // object (function)
let dateVar = new Date();               // object

// Type checking
console.log(typeof undefinedVar);   // "undefined"
console.log(typeof nullVar);        // "object" (known quir
console.log(typeof boolVar);        // "boolean"
console.log(typeof numVar);         // "number"
console.log(typeof strVar);         // "string"
console.log(typeof symVar);         // "symbol"
console.log(typeof bigIntVar);      // "bigint"
console.log(typeof objVar);         // "object"
console.log(typeof arrVar);         // "object"
console.log(typeof funcVar);        // "function"

// Better type checking for arrays
console.log(Array.isArray(arrVar)); // true
console.log(Array.isArray(objVar)); // false
```

Important Notes:

- `typeof null` returns `"object"` (historical bug that can't be fixed)
- Arrays are objects in JavaScript
- Functions are objects but `typeof` returns `"function"`
- Use `Array.isArray()` to check for arrays

Related Questions: null vs undefined, Type Coercion

⬆️ Back to Top

---

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

The three keywords for variable declaration have different behaviors regarding **scope**, **hoisting**, **redeclaration**, and **reassignment**:

Comparison Table:

Feature	var	let	const
Scope	Function/Global	Block	Block
Hoisting	Yes (initialized with undefined)	Yes (Temporal Dead Zone)	Yes (Temporal Dead Zone)
Redeclaration	Allowed	Not allowed	Not allowed
Reassignment	Allowed	Allowed	Not allowed*
Initialization	Optional	Optional	Required

*Objects and arrays declared with `const` can be mutated, but not reassigned.

Examples:

Scope Differences:

```
function scopeExample() {
    if (true) {
        var varVariable = 'I am var';
        let letVariable = 'I am let';
        const constVariable = 'I am const';
    }
```

```
    console.log(varVariable);    // 'I am var' - accessible
    // console.log(letVariable); // ReferenceError - not ac
    // console.log(constVariable); // ReferenceError - not
}

// Global scope pollution with var
var globalVar = 'global';
console.log(window.globalVar); // 'global' (in browsers)

let globalLet = 'global';
console.log(window.globalLet); // undefined
```

Best Practices:

1. Use `const` by default
2. Use `let` when you need to reassign the variable
3. Avoid `var` in modern JavaScript (ES6+)

Related Questions: Hoisting, Scope

⬆️ Back to Top

---

5. What is hoisting in JavaScript?

Hoisting is JavaScript's behavior of moving variable and function declarations to the top of their containing scope during the compilation phase, before code execution.

Key Points:

- Only **declarations** are hoisted, not **initializations**
- Variables are hoisted but initialized with `undefined`
- Function declarations are fully hoisted

- `let` and `const` are hoisted but in a "Temporal Dead Zone"

Variable Hoisting:

```
// What you write:
console.log(myVar); // undefined (not ReferenceError)
var myVar = 5;
console.log(myVar); // 5

// How JavaScript interprets it:
var myVar; // Declaration hoisted
console.log(myVar); // undefined
myVar = 5; // Initialization stays in place
console.log(myVar); // 5
```

Function Hoisting:

```
// Function declarations are fully hoisted
console.log(sayHello()); // "Hello!" - works before declara

function sayHello() {
    return "Hello!";
}

// Function expressions are not hoisted
// console.log(sayGoodbye()); // TypeError: sayGoodbye is n
var sayGoodbye = function() {
    return "Goodbye!";
};
```

Best Practices:

1. Declare variables at the top of their scope
2. Use `let` and `const` instead of `var`

3. Declare functions before using them
4. Initialize variables when declaring them

Related Questions: var, let, const, Scope

⬆️ Back to Top

---

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

Both `null` and `undefined` represent "no value," but they have different meanings and use cases:

`undefined`:

- **Meaning**: Variable has been declared but not assigned a value
- **Type**: `undefined`
- **Automatic**: JavaScript assigns this automatically
- **Default**: Default value for uninitialized variables, missing function parameters, and missing object properties

`null`:

- **Meaning**: Intentional absence of any value
- **Type**: `object` (this is a known bug in JavaScript)
- **Intentional**: Explicitly assigned by the programmer
- **Purpose**: Represents "nothing," "empty," or "value unknown"

Examples:

```
// undefined scenarios
let declaredButNotAssigned;
console.log(declaredButNotAssigned); // undefined
```

```
function testFunction(param) {
    console.log(param); // undefined if no argument passed
}
testFunction(); // undefined

const obj = { name: 'John' };
console.log(obj.age); // undefined (property doesn't exist)

// null scenarios
let intentionallyEmpty = null;
console.log(intentionallyEmpty); // null

// API responses often use null
const user = {
    name: 'John',
    avatar: null, // User has no avatar
    lastLogin: null // User never logged in
};
```

Type Checking:

```
console.log(typeof undefined); // "undefined"
console.log(typeof null);      // "object" (historical bug)

// Better null checking
console.log(null === null);           // true
console.log(undefined === undefined); // true
console.log(null === undefined);      // false
console.log(null == undefined);       // true (loose equali
```

Best Practices:

1. Use `undefined` for uninitialized variables (JavaScript default)
2. Use `null` to explicitly represent "no value" or "empty"

3. Use `==` to check for both null and undefined: `if (value == null)`
4. Use `===` for strict checking: `if (value === null)`

Related Questions: Type Coercion, Data Types

Back to Top

---

7. What is type coercion in JavaScript?

Type coercion is the automatic or implicit conversion of values from one data type to another. JavaScript performs coercion when operators or functions expect a certain type but receive a different type.

Types of Coercion:

1. **Implicit Coercion** - Automatic conversion by JavaScript
2. **Explicit Coercion** - Manual conversion by the programmer

String Coercion:

```
// Implicit string coercion (+ operator with strings)
console.log(5 + '3');         // '53' (number 5 becomes stri
console.log('Hello' + 42);    // 'Hello42'
console.log(true + ' story'); // 'true story'

// Explicit string coercion
console.log(String(123));     // '123'
console.log(String(true));    // 'true'
console.log(String(null));    // 'null'
console.log(String(undefined)); // 'undefined'
```

Numeric Coercion:

```
// Implicit numeric coercion
console.log('5' - 2);        // 3 (string '5' becomes numbe
console.log('10' * 2);       // 20
console.log('15' / 3);       // 5
console.log(+'42');          // 42 (unary + converts to num

// Explicit numeric coercion
console.log(Number('123'));  // 123
console.log(Number(''));     // 0
console.log(Number('abc'));  // NaN
console.log(Number(true));   // 1
console.log(Number(false));  // 0
```

Boolean Coercion:

```
// Falsy values (convert to false)
console.log(Boolean(0));         // false
console.log(Boolean(''));        // false
console.log(Boolean(null));      // false
console.log(Boolean(undefined)); // false
console.log(Boolean(NaN));       // false

// Everything else is truthy (converts to true)
console.log(Boolean(1));         // true
console.log(Boolean('hello'));   // true
console.log(Boolean([]));        // true (empty array is tr
console.log(Boolean({}));        // true (empty object is t
```

Best Practices:

```
// Avoid implicit coercion - be explicit
// Instead of:
if (value) { /* ... */ }
```

```
// Be explicit about what you're checking:
if (value !== null && value !== undefined) { /* ... */ }
if (value !== '') { /* ... */ }
if (value !== 0) { /* ... */ }

// Use strict equality
if (x === y) { /* ... */ } // Instead of x == y

// Explicit conversions
const num = Number(stringValue);
const str = String(numberValue);
const bool = Boolean(value);
```

Related Questions: Equality Operators, Data Types

⬆️ Back to Top

Functions

8. What are the different ways to create functions in JavaScript?

JavaScript provides several ways to create functions, each with different characteristics:

1. Function Declaration:

```
function add(a, b) {
    return a + b;
}
```

2. Function Expression:

```
const multiply = function(a, b) {
    return a * b;
};
```

3. Arrow Function (ES6+):

```
const divide = (a, b) => a / b;
const square = x => x * x;
const greet = () => "Hello!";
```

4. Method Definition (in objects):

```
const calculator = {
    add(a, b) { return a + b; },
    multiply: function(a, b) { return a * b; }
};
```

5. Function Constructor (rarely used):

```
const subtract = new Function('a', 'b', 'return a - b');
```

6. Generator Function:

```
function* fibonacci() {
    let a = 0, b = 1;
    while (true) {
        yield a;
        [a, b] = [b, a + b];
    }
}
```

7. Async Function:

```
async function fetchData() {
    const response = await fetch('/api/data');
    return response.json();
}
```

Key Differences:

- **Hoisting**: Function declarations are fully hoisted, expressions are not
- `this` **binding**: Arrow functions inherit `this`, others have dynamic `this`
- **Constructor**: Arrow functions cannot be used as constructors

Related Questions: Function Declarations vs Expressions, Arrow Functions

⬆️ Back to Top

---

9. What's the difference between function declarations and function expressions?

The key differences lie in **hoisting behavior**, **naming**, and **when they can be called**:

Function Declaration:

```
function declarationFunction() {
    return "I'm a declaration";
}
```

Function Expression:

```
const expressionFunction = function() {
    return "I'm an expression";
};
```

Key Differences:

1. Hoisting Behavior:

```
// This works - function declaration is hoisted
console.log(hoistedDeclaration()); // "I'm hoisted!"

function hoistedDeclaration() {
    return "I'm hoisted!";
}

// This doesn't work - function expression is not hoisted
console.log(notHoisted()); // TypeError: notHoisted is not

var notHoisted = function() {
    return "I'm not hoisted";
};
```

2. Conditional Creation:

```
let condition = true;

if (condition) {
    // Function declaration - behavior varies by environmen
    function conditionalDeclaration() {
        return "Declaration in block";
    }

    // Function expression - safer approach
    var conditionalExpression = function() {
```

```
        return "Expression in block";
    };
}
```

3. Recursion:

```
// Function declaration - can call itself by name
function factorial(n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

// Named function expression - can use internal name
const factorialExpr = function factorial(n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
};
```

When to Use Each:

- **Function Declarations**: Main functions, utilities, when you need hoisting
- **Function Expressions**: Conditional functions, callbacks, when you want to prevent hoisting

Related Questions: Hoisting, Arrow Functions

⬆️ Back to Top

---

10. What are arrow functions and how do they differ from regular functions?

Arrow functions (introduced in ES6) provide a concise syntax for writing functions, but they have important behavioral differences from regular functions.

Basic Syntax:

```
// Regular function
function regularFunction(a, b) {
    return a + b;
}

// Arrow function - full syntax
const arrowFunction = (a, b) => {
    return a + b;
};

// Arrow function - concise syntax (implicit return)
const conciseArrow = (a, b) => a + b;

// Single parameter (parentheses optional)
const singleParam = x => x * 2;

// No parameters (parentheses required)
const noParams = () => "Hello World";
```

Key Differences:

1. `this` Binding - Most Important Difference:

```
const obj = {
    name: 'MyObject',

    // Regular function - has its own 'this'
    regularMethod: function() {
        console.log(this.name); // 'MyObject'

        setTimeout(function() {
            console.log(this.name); // undefined (global ob
        }, 100);
```

```
    },

    // Arrow function - inherits 'this' from enclosing scop
    arrowMethod: () => {
        console.log(this.name); // undefined (inherits from
    },

    // Regular function with arrow function inside
    mixedMethod: function() {
        console.log(this.name); // 'MyObject'

        setTimeout(() => {
            console.log(this.name); // 'MyObject' (inherits
        }, 100);
    }
};
```

2. Arguments Object:

```
// Regular function - has 'arguments' object
function regularFunction() {
    console.log(arguments); // [1, 2, 3]
}
regularFunction(1, 2, 3);

// Arrow function - no 'arguments' object
const arrowFunction = (...args) => {
    console.log(args); // [1, 2, 3] - use rest parameters i
};
arrowFunction(1, 2, 3);
```

3. Constructor Usage:

```
// Regular function - can be used as constructor
function RegularConstructor(name) {
    this.name = name;
}
const instance1 = new RegularConstructor('John'); // Works

// Arrow function - cannot be used as constructor
const ArrowConstructor = (name) => {
    this.name = name;
};
// const instance2 = new ArrowConstructor('Jane'); // TypeE
```

When to Use Arrow Functions:

```
// ✅ Good use cases:
// 1. Short utility functions
const double = x => x * 2;

// 2. Array methods
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(n => n * 2);

// 3. Callbacks where you want to preserve 'this'
class Timer {
    constructor() {
        this.seconds = 0;
        setInterval(() => {
            this.seconds++; // 'this' refers to Timer insta
        }, 1000);
    }
}

// ❌ Avoid arrow functions for:
// 1. Object methods
const obj = {
```

```
    name: 'Object',
    getName: () => this.name // 'this' won't be the object
};

// 2. Event handlers when you need the element as 'this'
// button.addEventListener('click', () => {
//     this.style.color = 'red'; // 'this' won't be the but
// });
```

Best Practices:

1. Use arrow functions for short, simple functions
2. Use arrow functions in callbacks when you need to preserve `this`
3. Use regular functions for object methods
4. Use regular functions for constructors

Related Questions: this keyword, Function Expressions

⬆️ Back to Top

---

11. What is the `this` keyword and how does it work?

The `this` keyword refers to the context object that a function is called with. Its value depends on **how** a function is called, not where it's defined.

The Four Binding Rules:

1. Default Binding (Global Context):

```
function globalFunction() {
    console.log(this); // Window object (browser) or global
}

globalFunction(); // 'this' is the global object
```

```
// In strict mode
'use strict';
function strictFunction() {
    console.log(this); // undefined
}
strictFunction();
```

2. Implicit Binding (Object Method):

```
const person = {
    name: 'John',
    greet: function() {
        console.log(this.name); // 'John' - 'this' is the p
    }
};

person.greet(); // 'this' refers to person object

// Lost binding
const greetFunction = person.greet;
greetFunction(); // 'this' is global object (or undefined i
```

3. Explicit Binding (call, apply, bind):

```
const person1 = { name: 'Alice' };
const person2 = { name: 'Bob' };

function introduce() {
    console.log(`Hi, I'm ${this.name}`);
}

// call() - arguments passed individually
introduce.call(person1); // "Hi, I'm Alice"
```

```
introduce.call(person2); // "Hi, I'm Bob"

// apply() - arguments passed as array
function greet(greeting, punctuation) {
    console.log(`${greeting}, I'm ${this.name}${punctuation
}

greet.apply(person1, ['Hello', '!']); // "Hello, I'm Alice!

// bind() - creates new function with bound 'this'
const boundGreet = greet.bind(person2);
boundGreet('Hi', '.'); // "Hi, I'm Bob."
```

4. New Binding (Constructor):

```
function Person(name) {
    this.name = name;
    this.greet = function() {
        console.log(`Hello, I'm ${this.name}`);
    };
}

const john = new Person('John');
john.greet(); // "Hello, I'm John" - 'this' refers to new i
```

Arrow Functions and `this`:

```
const obj = {
    name: 'Object',

    regularMethod() {
        console.log(this.name); // 'Object'

        // Regular function - loses 'this'
```

```
        setTimeout(function() {
            console.log(this.name); // undefined
        }, 100);

        // Arrow function - preserves 'this'
        setTimeout(() => {
            console.log(this.name); // 'Object'
        }, 200);
    }
};
```

Common Pitfalls:

```
// 1. Method assignment
const obj = {
    name: 'Test',
    getName() { return this.name; }
};

const getName = obj.getName;
console.log(getName()); // undefined - lost binding

// Solution: bind the method
const boundGetName = obj.getName.bind(obj);
console.log(boundGetName()); // 'Test'

// 2. Event handlers
class Button {
    constructor(element) {
        this.element = element;
        this.clickCount = 0;

        // Wrong - 'this' will be the DOM element
        this.element.addEventListener('click', this.handleC

        // Right - bind the method
```

```
        this.element.addEventListener('click', this.handleC

        // Or use arrow function
        this.element.addEventListener('click', () => this.h
    }

    handleClick() {
        this.clickCount++;
        console.log(`Clicked ${this.clickCount} times`);
    }
}
```

Related Questions: Arrow Functions, call, apply, bind

Back to Top

---

12. What are higher-order functions?

Higher-order functions are functions that either:

1. Take one or more functions as arguments, OR
2. Return a function as their result

They enable **functional programming** patterns and code reusability.

Functions as Arguments:

```
// Basic example
function greet(name) {
    return `Hello, ${name}!`;
}

function processUserInput(callback) {
    const name = 'John';
    return callback(name);
```

```
}

console.log(processUserInput(greet)); // "Hello, John!"

// Array methods are higher-order functions
const numbers = [1, 2, 3, 4, 5];

// map() takes a function as argument
const doubled = numbers.map(x => x * 2); // [2, 4, 6, 8, 10

// filter() takes a function as argument
const evens = numbers.filter(x => x % 2 === 0); // [2, 4]

// reduce() takes a function as argument
const sum = numbers.reduce((acc, curr) => acc + curr, 0); /
```

Functions Returning Functions:

```
// Function factory
function createMultiplier(factor) {
    return function(number) {
        return number * factor;
    };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15

// Practical example: Event handler creator
function createEventHandler(eventType) {
    return function(element, callback) {
        element.addEventListener(eventType, callback);
    };
```

```
}

const addClickHandler = createEventHandler('click');
const addHoverHandler = createEventHandler('mouseenter');

// Usage
addClickHandler(button, () => console.log('Clicked!'));
addHoverHandler(div, () => console.log('Hovered!'));
```

Common Higher-Order Functions:

1. Array Methods:

```
const users = [
    { name: 'Alice', age: 25, active: true },
    { name: 'Bob', age: 30, active: false },
    { name: 'Charlie', age: 35, active: true }
];

// map - transform each element
const names = users.map(user => user.name); // ['Alice', 'B

// filter - select elements based on condition
const activeUsers = users.filter(user => user.active);

// find - get first matching element
const alice = users.find(user => user.name === 'Alice');

// some - check if any element matches
const hasActiveUsers = users.some(user => user.active); //

// every - check if all elements match
const allActive = users.every(user => user.active); // fals
```

```
// sort - sort with custom comparator
const sortedByAge = users.sort((a, b) => a.age - b.age);
```

2. Custom Higher-Order Functions:

```
// Retry function
function retry(fn, maxAttempts) {
    return function(...args) {
        let attempts = 0;

        while (attempts < maxAttempts) {
            try {
                return fn.apply(this, args);
            } catch (error) {
                attempts++;
                if (attempts >= maxAttempts) {
                    throw error;
                }
            }
        }
    };
}

// Usage
const unreliableFunction = () => {
    if (Math.random() < 0.7) throw new Error('Failed');
    return 'Success!';
};

const reliableFunction = retry(unreliableFunction, 3);

// Debounce function
function debounce(func, delay) {
    let timeoutId;

    return function(...args) {
```

```
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => func.apply(this, args)
    };
}

// Usage
const debouncedSearch = debounce((query) => {
    console.log(`Searching for: ${query}`);
}, 300);

// Memoization
function memoize(fn) {
    const cache = new Map();

    return function(...args) {
        const key = JSON.stringify(args);

        if (cache.has(key)) {
            return cache.get(key);
        }

        const result = fn.apply(this, args);
        cache.set(key, result);
        return result;
    };
}

// Usage
const expensiveFunction = (n) => {
    console.log(`Computing for ${n}`);
    return n * n;
};

const memoizedFunction = memoize(expensiveFunction);
console.log(memoizedFunction(5)); // Computing for 5, retur
console.log(memoizedFunction(5)); // Returns 25 (from cache
```

Function Composition:

```
// Compose functions together
function compose(...functions) {
    return function(value) {
        return functions.reduceRight((acc, fn) => fn(acc),
    };
}

// Individual functions
const addOne = x => x + 1;
const double = x => x * 2;
const square = x => x * x;

// Compose them
const composedFunction = compose(square, double, addOne);
console.log(composedFunction(3)); // ((3 + 1) * 2)^2 = 64

// Pipe (left-to-right composition)
function pipe(...functions) {
    return function(value) {
        return functions.reduce((acc, fn) => fn(acc), value
    };
}

const pipedFunction = pipe(addOne, double, square);
console.log(pipedFunction(3)); // ((3 + 1) * 2)^2 = 64
```

Benefits of Higher-Order Functions:

1. **Code Reusability**: Write generic functions that work with different behaviors
2. **Abstraction**: Hide complex logic behind simple interfaces
3. **Composition**: Build complex functionality from simple parts
4. **Functional Programming**: Enable declarative programming style

Related Questions: Function Currying, Array Methods

Back to Top

---

13. What is function currying?

Currying is a functional programming technique that transforms a function with multiple arguments into a sequence of functions, each taking a single argument.

Basic Concept:

```
// Regular function
function add(a, b, c) {
    return a + b + c;
}

console.log(add(1, 2, 3)); // 6

// Curried version
function curriedAdd(a) {
    return function(b) {
        return function(c) {
            return a + b + c;
        };
    };
}

console.log(curriedAdd(1)(2)(3)); // 6

// Arrow function version
const curriedAddArrow = a => b => c => a + b + c;
console.log(curriedAddArrow(1)(2)(3)); // 6
```

Manual Currying Examples:

```
// Multiplication example
function multiply(a, b, c) {
    return a * b * c;
}

// Curried version
function curriedMultiply(a) {
    return function(b) {
        return function(c) {
            return a * b * c;
        };
    };
}

// Partial application
const multiplyBy2 = curriedMultiply(2);
const multiplyBy2And3 = multiplyBy2(3);

console.log(multiplyBy2And3(4)); // 24
console.log(multiplyBy2(5)(6)); // 60

// String formatting example
function formatString(prefix) {
    return function(suffix) {
        return function(content) {
            return `${prefix}${content}${suffix}`;
        };
    };
}

const addBrackets = formatString('[')(']');
const addParens = formatString('(')(')');

console.log(addBrackets('Hello')); // [Hello]
console.log(addParens('World')); // (World)
```

Generic Curry Function:

```
// Curry function that works with any function
function curry(fn) {
    return function curried(...args) {
        if (args.length >= fn.length) {
            return fn.apply(this, args);
        } else {
            return function(...nextArgs) {
                return curried.apply(this, args.concat(next
            };
        }
    };
}

// Usage
function sum(a, b, c, d) {
    return a + b + c + d;
}

const curriedSum = curry(sum);

// All these work:
console.log(curriedSum(1)(2)(3)(4)); // 10
console.log(curriedSum(1, 2)(3)(4)); // 10
console.log(curriedSum(1)(2, 3, 4)); // 10
console.log(curriedSum(1, 2, 3, 4)); // 10
```

Practical Applications:

1. Event Handling:

```
function addEventListener(element) {
    return function(eventType) {
        return function(handler) {
```

```
            element.addEventListener(eventType, handler);
        };
    };
}

const button = document.getElementById('myButton');
const addButtonEvent = addEventListener(button);
const addClickEvent = addButtonEvent('click');

addClickEvent(() => console.log('Button clicked!'));
addClickEvent(() => console.log('Another click handler!'));

// Or chain them
addEventListener(button)('mouseover')(() => console.log('Ho
```

2. API Configuration:

```
function apiCall(baseUrl) {
    return function(endpoint) {
        return function(method) {
            return function(data) {
                return fetch(`${baseUrl}${endpoint}`, {
                    method: method,
                    headers: { 'Content-Type': 'application
                    body: JSON.stringify(data)
                });
            };
        };
    };
}

// Create specialized functions
const myApi = apiCall('https://api.example.com');
const usersEndpoint = myApi('/users');
const postToUsers = usersEndpoint('POST');
```

```
// Use it
postToUsers({ name: 'John', email: 'john@example.com' })
    .then(response => response.json())
    .then(data => console.log(data));

// Or create even more specific functions
const getUsersApi = myApi('/users')('GET');
const createUserApi = myApi('/users')('POST');
const updateUserApi = myApi('/users')('PUT');
```

3. Validation:

```
function validate(rule) {
    return function(errorMessage) {
        return function(value) {
            if (rule(value)) {
                return { valid: true, value };
            } else {
                return { valid: false, error: errorMessage
            }
        };
    };
}

// Create validation rules
const isRequired = validate(val => val !== null && val !==
const isEmail = validate(val => /^[^\s@]+@[^\s@]+\.[^\s@]+$
const minLength = min => validate(val => val.length >= min)

// Create specific validators
const requiredField = isRequired('This field is required');
const validEmail = isEmail('Please enter a valid email');
const minPassword = minLength(8)('Password must be at least

// Usage
console.log(requiredField('John')); // { valid: true, value
```

```
console.log(validEmail('invalid')); // { valid: false, erro
console.log(minPassword('12345678')); // { valid: true, val
```

4. Functional Composition:

```
const compose = (f, g) => x => f(g(x));
const pipe = (...fns) => x => fns.reduce((acc, fn) => fn(ac

// Curried utility functions
const add = a => b => a + b;
const multiply = a => b => a * b;
const subtract = a => b => a - b;

// Create specialized functions
const add10 = add(10);
const multiplyBy3 = multiply(3);
const subtract5 = subtract(5);

// Compose them
const complexOperation = pipe(
    add10,      // x + 10
    multiplyBy3, // (x + 10) * 3
    subtract5   // ((x + 10) * 3) - 5
);

console.log(complexOperation(5)); // ((5 + 10) * 3) - 5 = 4
```

Benefits of Currying:

1. **Partial Application**: Create specialized functions from general ones
2. **Function Composition**: Easier to compose functions together
3. **Code Reusability**: Create configurable, reusable functions
4. **Functional Programming**: Enables point-free programming style

When to Use Currying:

- When you frequently call a function with some of the same arguments
- When building configurable functions or APIs
- When working with functional programming patterns
- When you need to create specialized versions of generic functions

Related Questions: Higher-Order Functions, Function Composition

Back to Top

---

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

These three methods allow you to **explicitly set the value of** `this` in a function call, but they work differently:

`call()` - Invoke immediately with individual arguments:

```
function greet(greeting, punctuation) {
    return `${greeting}, I'm ${this.name}${punctuation}`;
}

const person = { name: 'Alice' };

// call(thisArg, arg1, arg2, ...)
const result = greet.call(person, 'Hello', '!');
console.log(result); // "Hello, I'm Alice!"
```

`apply()` - Invoke immediately with arguments array:

```
function greet(greeting, punctuation) {
    return `${greeting}, I'm ${this.name}${punctuation}`;
}
```

```
const person = { name: 'Bob' };

// apply(thisArg, [argsArray])
const result = greet.apply(person, ['Hi', '.']);
console.log(result); // "Hi, I'm Bob."

// Useful with arrays
const numbers = [1, 5, 3, 9, 2];
const max = Math.max.apply(null, numbers); // 9
// Modern equivalent: Math.max(...numbers)
```

`bind()` - Create new function with bound `this`:

```
function greet(greeting, punctuation) {
    return `${greeting}, I'm ${this.name}${punctuation}`;
}

const person = { name: 'Charlie' };

// bind(thisArg, arg1, arg2, ...) - returns new function
const boundGreet = greet.bind(person);
console.log(boundGreet('Hey', '?')); // "Hey, I'm Charlie?"

// Partial application with bind
const boundGreetWithHello = greet.bind(person, 'Hello');
console.log(boundGreetWithHello('!!!')); // "Hello, I'm Cha
```

Comparison Table:

Method	Invocation	Arguments	Returns	Use Case
`call`	Immediate	Individual	Function result	Direct function call

Method	Invocation	Arguments	Returns	Use Case
`apply`	Immediate	Array	Function result	Arguments in array form
`bind`	Deferred	Individual	New function	Event handlers, callbacks

Practical Examples:

1. Borrowing Methods:

```
const person1 = {
    name: 'John',
    greet: function() {
        return `Hello, I'm ${this.name}`;
    }
};

const person2 = { name: 'Jane' };

// Borrow method from person1 for person2
console.log(person1.greet.call(person2)); // "Hello, I'm Ja
console.log(person1.greet.apply(person2)); // "Hello, I'm J

const boundGreet = person1.greet.bind(person2);
console.log(boundGreet()); // "Hello, I'm Jane"
```

2. Array-like Objects:

```
function showArguments() {
    // arguments is array-like but not a real array
    console.log(arguments); // [Arguments] { '0': 1, '1': 2
```

```
    // Convert to real array using call
    const argsArray = Array.prototype.slice.call(arguments)
    console.log(argsArray); // [1, 2, 3]

    // Or using apply
    const argsArray2 = Array.prototype.slice.apply(argument
    console.log(argsArray2); // [1, 2, 3]

    // Modern way
    const argsArray3 = Array.from(arguments);
    const argsArray4 = [...arguments];
}

showArguments(1, 2, 3);

// NodeList example
const divs = document.querySelectorAll('div');
// divs is NodeList, not Array

// Convert to array to use array methods
const divsArray = Array.prototype.slice.call(divs);
// Or: const divsArray = Array.from(divs);
// Or: const divsArray = [...divs];
```

3. Event Handlers:

```
class Button {
    constructor(element) {
        this.element = element;
        this.clickCount = 0;

        // Problem: 'this' will be the DOM element in the h
        // this.element.addEventListener('click', this.hand

        // Solution 1: bind
        this.element.addEventListener('click', this.handleC
```

```
        // Solution 2: arrow function (preserves 'this')
        // this.element.addEventListener('click', () => thi
    }

    handleClick() {
        this.clickCount++;
        console.log(`Clicked ${this.clickCount} times`);
    }
}

// Removing event listeners with bind
class Component {
    constructor() {
        this.boundHandleClick = this.handleClick.bind(this)
    }

    addListener() {
        document.addEventListener('click', this.boundHandle
    }

    removeListener() {
        document.removeEventListener('click', this.boundHan
    }

    handleClick() {
        console.log('Component clicked');
    }
}
```

4. Function Composition and Partial Application:

```
// Partial application with bind
function multiply(a, b, c) {
    return a * b * c;
}
```

```
const multiplyBy2 = multiply.bind(null, 2);
const multiplyBy2And3 = multiply.bind(null, 2, 3);

console.log(multiplyBy2(5, 6)); // 60
console.log(multiplyBy2And3(4)); // 24

// Creating utility functions
function log(level, message) {
    console.log(`[${level}] ${message}`);
}

const logError = log.bind(null, 'ERROR');
const logWarning = log.bind(null, 'WARNING');
const logInfo = log.bind(null, 'INFO');

logError('Something went wrong'); // [ERROR] Something went
logWarning('Be careful'); // [WARNING] Be careful
logInfo('All good'); // [INFO] All good
```

5. Constructor Borrowing:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

function Student(name, age, grade) {
    // Borrow Person constructor
    Person.call(this, name, age);
    this.grade = grade;
}

const student = new Student('John', 20, 'A');
console.log(student); // { name: 'John', age: 20, grade: 'A
```

```
// Modern equivalent with classes
class PersonClass {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}

class StudentClass extends PersonClass {
    constructor(name, age, grade) {
        super(name, age); // calls parent constructor
        this.grade = grade;
    }
}
```

Performance Considerations:

```
// bind creates a new function each time
class Component {
    constructor() {
        // ❌ Creates new function on every render
        this.onClick = this.handleClick.bind(this);
    }

    render() {
        // ❌ Creates new function on every render
        return `<button onclick="${this.handleClick.bind(th
    }

    handleClick() {
        console.log('Clicked');
    }
}

// ✅ Better approach
class BetterComponent {
```

```
    constructor() {
        // ✅ Create bound function once
        this.boundHandleClick = this.handleClick.bind(this)
    }

    render() {
        return `<button onclick="${this.boundHandleClick}">
    }

    handleClick() {
        console.log('Clicked');
    }
}
```

When to Use Each:

- `call`: When you know the exact arguments and want immediate execution
- `apply`: When arguments are in an array or you're working with variable arguments
- `bind`: When you need a reusable function with a specific `this` context (event handlers, callbacks)

Related Questions: this keyword, Arrow Functions

⬆️ Back to Top

Scope and Closures

15. What is scope in JavaScript?

Scope determines the accessibility and visibility of variables, functions, and objects in different parts of your code during runtime.

Types of Scope:

1. Global Scope:

```
var globalVar = 'I am global';
let globalLet = 'I am also global';

function testGlobal() {
    console.log(globalVar); // Accessible
}
```

2. Function Scope:

```
function functionScope() {
    var functionVar = 'Function scoped';
    console.log(functionVar); // Accessible
}
// console.log(functionVar); // ReferenceError
```

3. Block Scope (ES6+):

```
function blockScopeExample() {
    if (true) {
        var blockVar = 'Function scoped'; // var ignores bl
        let blockLet = 'Block scoped';
        const blockConst = 'Also block scoped';
    }

    console.log(blockVar); // Accessible (var ignores block
    // console.log(blockLet); // ReferenceError
}
```

Scope Chain:

```
const globalVar = 'Global';

function outerFunction() {
    const outerVar = 'Outer';

    function innerFunction() {
        const innerVar = 'Inner';
        console.log(innerVar);  // 'Inner'
        console.log(outerVar);  // 'Outer' - from enclosing
        console.log(globalVar); // 'Global' - from global s
    }

    innerFunction();
}
```

Common Issues:

```
// Loop variable problem with var
for (var i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 100); // 3, 3, 3
}

// Solution with let
for (let j = 0; j < 3; j++) {
    setTimeout(() => console.log(j), 100); // 0, 1, 2
}
```

Related Questions: var, let, const, Closures

Back to Top

---

16. What are closures and why are they important?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives a function access to its outer scope. In JavaScript, closures are created every time a function is created, at function creation time.

Basic Example:

```
function outerFunction(x) {
    const outerVariable = x;

    function innerFunction(y) {
        return outerVariable + y; // Access to outer variab
    }

    return innerFunction;
}

const closure = outerFunction(10);
console.log(closure(5)); // 15
```

Practical Applications:

1. Data Privacy:

```
function createCounter() {
    let count = 0; // Private variable

    return function() {
        count++;
        return count;
    };
}

const counter1 = createCounter();
const counter2 = createCounter();
```

```
console.log(counter1()); // 1
console.log(counter1()); // 2
console.log(counter2()); // 1 (separate closure)
```

2. Module Pattern:

```
const Calculator = (function() {
    let result = 0; // Private variable

    return {
        add(x) { result += x; return this; },
        subtract(x) { result -= x; return this; },
        getResult() { return result; },
        clear() { result = 0; return this; }
    };
})();

const finalResult = Calculator.add(10).subtract(5).getResul
```

3. Function Factories:

```
function createMultiplier(multiplier) {
    return function(x) {
        return x * multiplier;
    };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Common Pitfall - Loops:

```
// Problem: All functions reference the same variable
const functions = [];
for (var i = 0; i < 3; i++) {
    functions.push(function() {
        return i; // All closures reference the same 'i'
    });
}
functions.forEach(fn => console.log(fn())); // 3, 3, 3

// Solution: Use let or IIFE
for (let i = 0; i < 3; i++) {
    functions.push(function() {
        return i; // Each closure gets its own 'i'
    });
}
```

Benefits:

1. **Data Privacy**: Create private variables
2. **Function Factories**: Generate specialized functions
3. **Module Pattern**: Organize code
4. **State Preservation**: Maintain state in callbacks

Related Questions: Scope, Higher-Order Functions

Back to Top

---

17. What is the difference between global, function, and block scope?

The three main types of scope in JavaScript have different rules for variable accessibility and lifetime:

Global Scope:

Variables declared outside any function or block have global scope.

```
// Global scope
var globalVar = 'Global with var';
let globalLet = 'Global with let';
const globalConst = 'Global with const';

// Global variables are accessible everywhere
function anyFunction() {
    console.log(globalVar);   // Accessible
    console.log(globalLet);   // Accessible
    console.log(globalConst); // Accessible
}

// In browsers, var creates properties on window object
console.log(window.globalVar); // 'Global with var'
console.log(window.globalLet); // undefined (let/const don'
```

Function Scope:

Variables declared inside a function are only accessible within that function.

```
function functionScopeExample() {
    // Function scoped variables
    var functionVar = 'Function scoped var';
    let functionLet = 'Function scoped let';
    const functionConst = 'Function scoped const';

    // All are accessible within the function
    console.log(functionVar);   // Works
    console.log(functionLet);   // Works
    console.log(functionConst); // Works
```

```
    // Nested function can access parent function's variabl
    function nestedFunction() {
        console.log(functionVar); // Accessible from parent
    }

    nestedFunction();
}

functionScopeExample();

// Not accessible outside the function
// console.log(functionVar); // ReferenceError
// console.log(functionLet); // ReferenceError
// console.log(functionConst); // ReferenceError
```

Block Scope (ES6+):

Variables declared with `let` and `const` inside a block `{}` are only accessible within that block.

```
function blockScopeExample() {
    // Block scope with if statement
    if (true) {
        var blockVar = 'var ignores block scope';
        let blockLet = 'let respects block scope';
        const blockConst = 'const respects block scope';

        console.log(blockVar);   // Accessible
        console.log(blockLet);   // Accessible
        console.log(blockConst); // Accessible
    }

    // Outside the block
    console.log(blockVar);   // Accessible (var ignores blo
    // console.log(blockLet);   // ReferenceError
```

```
    // console.log(blockConst); // ReferenceError

    // Block scope with for loop
    for (let i = 0; i < 3; i++) {
        let loopVar = `Iteration ${i}`;
        console.log(loopVar); // Accessible within loop
    }

    // console.log(i);       // ReferenceError (let is bloc
    // console.log(loopVar); // ReferenceError

    // Compare with var in loop
    for (var j = 0; j < 3; j++) {
        var loopVarVar = `Iteration ${j}`;
    }

    console.log(j);          // 3 (var is function scoped)
    console.log(loopVarVar); // 'Iteration 2' (var is funct
}

blockScopeExample();
```

Scope Comparison Table:

Scope Type	Declaration	Accessible From	Hoisted	Creates Window Property
Global	Outside functions/blocks	Everywhere	Yes	var: Yes, let/const: No

Scope Type	Declaration	Accessible From	Hoisted	Creates Window Property
Function	Inside functions	Within function and nested functions	Yes	N/A
Block	Inside blocks `{}`	Within block only (let/const)	let/const: TDZ, var: ignores blocks	N/A

Practical Examples:

1. Loop Scope Issues:

```
// Problem with var (function scoped)
console.log('--- var in loop ---');
for (var i = 0; i < 3; i++) {
    setTimeout(() => {
        console.log(`var i: ${i}`); // 3, 3, 3 (all referen
    }, 100);
}

// Solution with let (block scoped)
console.log('--- let in loop ---');
for (let i = 0; i < 3; i++) {
    setTimeout(() => {
        console.log(`let i: ${i}`); // 0, 1, 2 (each iterat
    }, 100);
}

// Solution with var using IIFE
```

```
console.log('--- var with IIFE ---');
for (var i = 0; i < 3; i++) {
    (function(index) {
        setTimeout(() => {
            console.log(`IIFE i: ${index}`); // 0, 1, 2
        }, 100);
    })(i);
}
```

2. Switch Statement Block Scope:

```
function switchScopeExample(value) {
    switch (value) {
        case 1: {
            let caseVar = 'Case 1';
            console.log(caseVar); // Accessible
            break;
        }
        case 2: {
            let caseVar = 'Case 2'; // Different variable (
            console.log(caseVar); // Accessible
            break;
        }
        default: {
            // console.log(caseVar); // ReferenceError - no
        }
    }
}

switchScopeExample(1);
switchScopeExample(2);
```

3. Try-Catch Block Scope:

```
function tryCatchScope() {
    try {
        let tryVar = 'In try block';
        throw new Error('Test error');
    } catch (error) {
        let catchVar = 'In catch block';
        console.log(catchVar); // Accessible
        // console.log(tryVar); // ReferenceError - not acc
    } finally {
        let finallyVar = 'In finally block';
        console.log(finallyVar); // Accessible
        // console.log(catchVar); // ReferenceError - not a
    }

    // None of the block variables are accessible here
    // console.log(tryVar);     // ReferenceError
    // console.log(catchVar);   // ReferenceError
    // console.log(finallyVar); // ReferenceError
}

tryCatchScope();
```

4. Object and Array Destructuring Scope:

```
function destructuringScope() {
    const obj = { a: 1, b: 2 };
    const arr = [3, 4, 5];

    if (true) {
        // Destructuring creates block-scoped variables
        const { a, b } = obj;
        const [first, second] = arr;

        console.log(a, b);        // 1, 2 - accessible in b
        console.log(first, second); // 3, 4 - accessible in
```

```
    }

    // console.log(a);     // ReferenceError - not accessib
    // console.log(first); // ReferenceError - not accessib
}

destructuringScope();
```

Best Practices:

1. **Use `let` and `const`** instead of `var` to avoid scope confusion
2. **Minimize global variables** to avoid pollution and conflicts
3. **Use block scope** to limit variable lifetime and improve readability
4. **Declare variables as close to their usage** as possible
5. **Use IIFE or modules** to create private scope when needed

Common Mistakes:

```
// Mistake 1: Accidental global
function accidentalGlobal() {
    myVar = 'I am accidentally global'; // Missing declarat
}
accidentalGlobal();
console.log(myVar); // 'I am accidentally global' - oops!

// Mistake 2: Expecting block scope with var
if (true) {
    var shouldBeBlockScoped = 'But I am not';
}
console.log(shouldBeBlockScoped); // 'But I am not' - var i

// Mistake 3: Loop variable confusion
const clickHandlers = [];
for (var i = 0; i < 3; i++) {
```

```
    clickHandlers.push(() => console.log(i)); // All will l
}
clickHandlers[0](); // 3 (not 0 as expected)
```

Related Questions: var, let, const, Hoisting, Closures

Back to Top

---

18. What is the temporal dead zone?

The **Temporal Dead Zone (TDZ)** is the period between entering scope and being declared where `let` and `const` variables exist but cannot be accessed.

Basic Concept:

```
console.log(typeof myVar);   // 'undefined' - var is hoiste
console.log(typeof myLet);   // ReferenceError - let is in
console.log(typeof myConst); // ReferenceError - const is i

var myVar = 'var variable';
let myLet = 'let variable';
const myConst = 'const variable';
```

How TDZ Works:

```
function temporalDeadZoneExample() {
    // TDZ starts here for 'letVar' and 'constVar'

    console.log(varVar);    // undefined (hoisted and initi
    // console.log(letVar);    // ReferenceError: Cannot ac
    // console.log(constVar);  // ReferenceError: Cannot ac

    var varVar = 'var value';
    let letVar = 'let value';     // TDZ ends here for letV
```

```
    const constVar = 'const value'; // TDZ ends here for co

    console.log(letVar);    // 'let value' - now accessible
    console.log(constVar);  // 'const value' - now accessib
}

temporalDeadZoneExample();
```

TDZ in Different Contexts:

1. Function Parameters:

```
// TDZ with default parameters
function defaultParamTDZ(a = b, b = 2) {
    return [a, b];
}

// defaultParamTDZ(); // ReferenceError: Cannot access 'b'

// Correct order
function correctDefaultParam(a = 1, b = a) {
    return [a, b];
}

console.log(correctDefaultParam()); // [1, 1]
```

2. Block Scope:

```
function blockTDZ() {
    let x = 'outer';

    if (true) {
        // TDZ starts here for inner 'x'
        // console.log(x); // ReferenceError - inner 'x' is
```

```
        let x = 'inner'; // TDZ ends here
        console.log(x); // 'inner'
    }

    console.log(x); // 'outer'
}

blockTDZ();
```

3. Class Declarations:

```
// Classes are also subject to TDZ
// const instance = new MyClass(); // ReferenceError

class MyClass {
    constructor(name) {
        this.name = name;
    }
}

const instance = new MyClass('Test'); // Works
```

4. Import Statements:

```
// This would cause TDZ error
// console.log(importedFunction()); // ReferenceError

import { importedFunction } from './module.js';

console.log(importedFunction()); // Works after import
```

TDZ with typeof:

```
// typeof with undeclared variables
console.log(typeof undeclaredVar); // 'undefined' - safe

// typeof with TDZ variables
// console.log(typeof letInTDZ); // ReferenceError - not sa
let letInTDZ = 'value';

// This is why checking for variable existence changed in E
function safeCheck() {
    try {
        return typeof someLetVariable !== 'undefined';
    } catch (e) {
        return false; // Variable is in TDZ or doesn't exis
    }
}
```

Practical Examples:

1. Loop Variables:

```
// TDZ in for loop
for (let i = 0; i < 3; i++) {
    // Each iteration creates new binding
    setTimeout(() => {
        console.log(i); // 0, 1, 2 - each closure has its o
    }, 100);
}

// TDZ prevents this common mistake
// for (let i = 0; i < i; i++) { // ReferenceError: Cannot
//     console.log(i);
// }
```

2. Function Declarations vs Expressions:

```
// Function declaration - no TDZ
console.log(declaredFunction()); // Works - 'I am declared'

function declaredFunction() {
    return 'I am declared';
}

// Function expression with let/const - TDZ applies
// console.log(expressionFunction()); // ReferenceError

const expressionFunction = function() {
    return 'I am an expression';
};

console.log(expressionFunction()); // Works after declarati
```

3. Switch Statement TDZ:

```
function switchTDZ(value) {
    switch (value) {
        case 1:
            // console.log(switchVar); // ReferenceError -
            let switchVar = 'Case 1';
            console.log(switchVar); // Works
            break;
        case 2:
            // switchVar is still in TDZ here because let i
            // console.log(switchVar); // ReferenceError
            switchVar = 'Case 2'; // ReferenceError - canno
            break;
    }
}

// Better approach with separate blocks
function betterSwitchTDZ(value) {
```

```
    switch (value) {
        case 1: {
            let switchVar = 'Case 1';
            console.log(switchVar);
            break;
        }
        case 2: {
            let switchVar = 'Case 2'; // Different variable
            console.log(switchVar);
            break;
        }
    }
}
```

Why TDZ Exists:

1. **Catch errors early**: Prevents using variables before they're properly initialized
2. **Consistent behavior**: Makes `let` and `const` behavior more predictable
3. **Avoid confusion**: Prevents the weird `undefined` behavior of `var` hoisting
4. **Better debugging**: Errors occur at the line where variable is accessed, not declared

TDZ vs Hoisting Comparison:

```
function hoistingComparison() {
    // var: Hoisted and initialized with undefined
    console.log(varVariable); // undefined
    var varVariable = 'var value';

    // let: Hoisted but not initialized (TDZ)
    // console.log(letVariable); // ReferenceError
    let letVariable = 'let value';

    // const: Hoisted but not initialized (TDZ)
```

```
    // console.log(constVariable); // ReferenceError
    const constVariable = 'const value';

    // Function declaration: Fully hoisted
    console.log(hoistedFunction()); // 'I work!'

    function hoistedFunction() {
        return 'I work!';
    }
}
```

Best Practices:

1. **Declare variables before use** - even though hoisting exists
2. **Use** `let` **and** `const` instead of `var` for better error catching
3. **Initialize variables at declaration** when possible
4. **Group declarations** at the top of their scope for clarity
5. **Use linters** that catch TDZ violations

Related Questions: var, let, const, Hoisting, Block Scope

⬆️ Back to Top

---

19. How does variable shadowing work?

Variable shadowing occurs when a variable declared in an inner scope has the same name as a variable in an outer scope, effectively "hiding" or "shadowing" the outer variable.

Basic Example:

```
let name = 'Global'; // Outer scope
```

```
function shadowExample() {
    let name = 'Function'; // Shadows global 'name'
    console.log(name); // 'Function' - inner variable wins

    if (true) {
        let name = 'Block'; // Shadows function 'name'
        console.log(name); // 'Block' - innermost variable
    }

    console.log(name); // 'Function' - back to function sco
}

shadowExample();
console.log(name); // 'Global' - original variable unchange
```

Shadowing with Different Declaration Types:

```
var globalVar = 'Global var';

function mixedShadowing() {
    console.log(globalVar); // 'Global var' - accessible

    if (true) {
        var globalVar = 'Shadowed var'; // Shadows global
        let blockLet = 'Block let';

        console.log(globalVar); // 'Shadowed var'
        console.log(blockLet);  // 'Block let'

        if (true) {
            const globalVar = 'Shadowed const'; // Shadows
            const blockLet = 'Nested const';   // Shadows b

            console.log(globalVar); // 'Shadowed const'
            console.log(blockLet);  // 'Nested const'
        }
```

```
        console.log(globalVar); // 'Shadowed var' - const d
        console.log(blockLet);  // 'Block let'
    }

    console.log(globalVar); // 'Shadowed var' - var is func
}

mixedShadowing();
console.log(globalVar); // 'Global var' - function didn't a
```

Function Parameter Shadowing:

```
let x = 'Global x';
let y = 'Global y';

function parameterShadowing(x, y) {
    // Parameters shadow global variables
    console.log(x); // Parameter value, not global
    console.log(y); // Parameter value, not global

    // Can still create more shadows
    if (true) {
        let x = 'Block x';
        console.log(x); // 'Block x' - shadows parameter
    }

    console.log(x); // Back to parameter value
}

parameterShadowing('Param x', 'Param y');
console.log(x); // 'Global x' - unchanged
```

Shadowing in Loops:

```
let counter = 'Global counter';

function loopShadowing() {
    console.log(counter); // 'Global counter'

    for (let counter = 0; counter < 3; counter++) {
        console.log(`Loop counter: ${counter}`); // 0, 1, 2

        if (counter === 1) {
            let counter = 'Nested loop counter';
            console.log(`Nested: ${counter}`); // 'Nested l
        }
    }

    console.log(counter); // 'Global counter'
}

loopShadowing();
```

Accidental Shadowing:

```
let config = {
    apiUrl: 'https://api.example.com',
    timeout: 5000
};

function processData(data) {
    // Accidentally shadowing global config
    let config = validateData(data); // Oops! This shadows

    // This won't work as expected
    // fetch(config.apiUrl); // TypeError: config.apiUrl is

    function validateData(data) {
        return { isValid: true, errors: [] };
```

```
    }
}

// Better approach - use different names
function processDataBetter(data) {
    let validationResult = validateData(data);

    // Now can access global config
    fetch(config.apiUrl, { timeout: config.timeout });

    function validateData(data) {
        return { isValid: true, errors: [] };
    }
}
```

Intentional Shadowing for Encapsulation:

```
const DatabaseConnection = (function() {
    let connection = null; // Private variable

    return {
        connect(connectionString) {
            // Shadow outer connection intentionally
            let connection = establishConnection(connection

            if (connection.isValid) {
                // Update outer connection
                connection = connection;
                return true;
            }
            return false;

            function establishConnection(str) {
                return { isValid: true, connectionString: s
            }
        },
```

```
        getConnection() {
            return connection;
        }
    };
})();
```

Shadowing with Arrow Functions:

```
let value = 'Global';

const outerFunction = () => {
    let value = 'Outer';

    const innerFunction = () => {
        let value = 'Inner';
        console.log(value); // 'Inner'

        // Arrow functions don't have their own 'arguments'
        // so they don't shadow it
        const showArgs = (...args) => {
            console.log(args); // Rest parameters, not argu
        };

        showArgs(1, 2, 3);
    };

    innerFunction();
    console.log(value); // 'Outer'
};

outerFunction();
console.log(value); // 'Global'
```

Shadowing with Destructuring:

```
const user = { name: 'John', age: 30 };
let name = 'Global name';

function destructuringShadowing() {
    console.log(name); // 'Global name'

    // Destructuring creates new variables that shadow
    const { name, age } = user;
    console.log(name); // 'John' - shadows global name
    console.log(age);  // 30

    if (true) {
        // Can shadow destructured variables too
        const name = 'Block name';
        console.log(name); // 'Block name'
    }

    console.log(name); // 'John' - back to destructured val
}

destructuringShadowing();
console.log(name); // 'Global name' - unchanged
```

Shadowing Best Practices:

1. Avoid Accidental Shadowing:

```
// Bad - accidental shadowing
let data = 'Important global data';

function processItems(items) {
    let data = items.map(item => item.value); // Accidental

    // Later in function, expecting global data
    // console.log(data.length); // Error if global data do
```

```
}

// Good - use descriptive names
function processItemsBetter(items) {
    let processedData = items.map(item => item.value);

    // Can still access global data
    console.log(data); // 'Important global data'
}
```

2. Use Linting Rules:

```
// ESLint rule: no-shadow
// This would be flagged by linter
function shadowingWarning() {
    let x = 1;

    if (true) {
        let x = 2; // ESLint warning: 'x' is already declar
        console.log(x);
    }
}
```

3. Intentional Shadowing for Privacy:

```
const Module = (function() {
    let privateVar = 'Private';

    return {
        publicMethod() {
            // Intentionally shadow to create local scope
            let privateVar = 'Local private';

            function helperFunction() {
```

```
                // This privateVar is the local one
                console.log(privateVar); // 'Local private'
            }

            helperFunction();

            // Can still access outer private if needed
            console.log(privateVar); // Still 'Local privat
        }
    };
})();
```

Common Pitfalls:

1. **Unintended behavior** when expecting outer variable
2. **Debugging confusion** when wrong variable is accessed
3. **Code maintenance issues** when shadowed variables are renamed
4. **Performance implications** in some JavaScript engines

Benefits of Understanding Shadowing:

1. **Better debugging** - know which variable is being accessed
2. **Intentional encapsulation** - create private scopes
3. **Avoid naming conflicts** - understand scope resolution
4. **Code clarity** - write more predictable code

Related Questions: Scope, var, let, const, Closures

⬆️ Back to Top

Objects and Arrays

20. How do you create and manipulate objects in JavaScript?

JavaScript provides multiple ways to create and manipulate objects:

Object Creation Methods:

1. Object Literal (Most Common):

```
const person = {
    name: 'John',
    age: 30,
    greet() { return `Hello, I'm ${this.name}`; }
};

// Computed property names
const prop = 'dynamicProperty';
const obj = {
    [prop]: 'dynamic value',
    [`${prop}2`]: 'another value'
};
```

2. Object Constructor:

```
const person = new Object();
person.name = 'John';
person.age = 30;

// Constructor function
function Person(name, age) {
    this.name = name;
    this.age = age;
}
const john = new Person('John', 30);
```

3. Object.create():

```
const personPrototype = {
    greet() { return `Hello, I'm ${this.name}`; }
};

const person = Object.create(personPrototype);
person.name = 'John';
```

Object Manipulation:

Adding/Modifying Properties:

```
const obj = {};

// Dot and bracket notation
obj.name = 'John';
obj['age'] = 30;

// Object.assign
Object.assign(obj, { city: 'NYC', country: 'USA' });

// Spread operator
const updated = { ...obj, age: 31, job: 'Developer' };
```

Property Access:

```
// Safe access with optional chaining
console.log(person.address?.street);

// Default values
const city = person.city || 'Unknown';
const country = person.country ?? 'Unknown';
```

Object Iteration:

```
const person = { name: 'John', age: 30 };

// Object.keys(), Object.values(), Object.entries()
Object.keys(person).forEach(key => console.log(key));
Object.values(person).forEach(value => console.log(value));
Object.entries(person).forEach(([key, value]) => {
    console.log(`${key}: ${value}`);
});
```

Related Questions: Destructuring, Spread Operator

Back to Top

---

21. What are the most common array methods?

JavaScript arrays come with many built-in methods for manipulation and iteration:

Mutating Methods (Modify Original Array):

Adding/Removing Elements:

```
const fruits = ['apple', 'banana'];

// push/pop - end of array
fruits.push('orange'); // ['apple', 'banana', 'orange']
const last = fruits.pop(); // 'orange'

// unshift/shift - beginning of array
fruits.unshift('mango'); // ['mango', 'apple', 'banana']
const first = fruits.shift(); // 'mango'

// splice - remove/insert anywhere
fruits.splice(1, 1, 'grape', 'kiwi'); // Remove 1, add 2 at
```

Non-Mutating Methods:

Transformation:

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

// map - transform each element
const doubled = numbers.map(x => x * 2); // [2, 4, 6, 8, 10

// filter - select elements
const evens = numbers.filter(x => x % 2 === 0); // [2, 4]

// reduce - aggregate to single value
const sum = numbers.reduce((acc, curr) => acc + curr, 0); /
```

Search Methods:

```
const fruits = ['apple', 'banana', 'orange'];

// find/findIndex - first match
const found = fruits.find(f => f.startsWith('b')); // 'bana
const index = fruits.findIndex(f => f.startsWith('b')); //

// includes - check existence
console.log(fruits.includes('apple')); // true

// indexOf/lastIndexOf - get index
console.log(fruits.indexOf('banana')); // 1
```

Testing Methods:

```
const numbers = [2, 4, 6, 8];

// every - all elements pass test
```

```
console.log(numbers.every(x => x % 2 === 0)); // true

// some - at least one passes test
console.log(numbers.some(x => x > 5)); // true
```

Array Utilities:

```
// slice - extract portion
const slice = numbers.slice(1, 3); // [2, 3] (from index 1

// concat - join arrays
const combined = [1, 2].concat([3, 4]); // [1, 2, 3, 4]

// join - convert to string
const str = ['Hello', 'World'].join(' '); // 'Hello World'

// flat - flatten nested arrays
const nested = [1, [2, 3], [4, 5]];
const flattened = nested.flat(); // [1, 2, 3, 4, 5]
```

Practical Examples:

```
// Method chaining
const result = [1, 2, 3, 4, 5, 6]
    .filter(x => x % 2 === 0)  // [2, 4, 6]
    .map(x => x * x)           // [4, 16, 36]
    .reduce((sum, x) => sum + x, 0); // 56

// Remove duplicates
const unique = [...new Set([1, 2, 2, 3, 3])]; // [1, 2, 3]
```

Related Questions: Higher-Order Functions, Spread Operator

Back to Top

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

Shallow Copy creates a new object but inserts references to the objects found in the original. **Deep Copy** creates a new object and recursively copies all nested objects.

Shallow Copy:

```
const original = {
    name: 'John',
    age: 30,
    address: {
        city: 'New York',
        country: 'USA'
    },
    hobbies: ['reading', 'swimming']
};

// Shallow copy methods
const shallow1 = { ...original };
const shallow2 = Object.assign({}, original);
const shallow3 = Object.create(original);

// Modifying nested objects affects original
shallow1.address.city = 'Boston';
console.log(original.address.city); // 'Boston' - original

shallow1.hobbies.push('cycling');
console.log(original.hobbies); // ['reading', 'swimming', '

// But primitive values are independent
shallow1.name = 'Jane';
console.log(original.name); // 'John' - original unchanged
```

Deep Copy Methods:

1. JSON.parse(JSON.stringify()) - Simple but Limited:

```
const original = {
    name: 'John',
    age: 30,
    address: { city: 'NYC' },
    hobbies: ['reading']
};

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

deepCopy.address.city = 'Boston';
console.log(original.address.city); // 'NYC' - original unc

// Limitations:
const problematic = {
    date: new Date(),
    func: () => 'hello',
    undef: undefined,
    symbol: Symbol('test'),
    regex: /test/g
};

const copied = JSON.parse(JSON.stringify(problematic));
console.log(copied);
// {
//   date: "2023-01-01T00:00:00.000Z", // becomes string
//   // func: missing (functions are ignored)
//   // undef: missing (undefined is ignored)
//   // symbol: missing (symbols are ignored)
//   regex: {} // becomes empty object
// }
```

2. structuredClone() - Modern Native Method (ES2022):

```
const original = {
    name: 'John',
    date: new Date(),
    regex: /test/g,
    map: new Map([['key', 'value']]),
    set: new Set([1, 2, 3]),
    buffer: new ArrayBuffer(8)
};

const deepCopy = structuredClone(original);

// Works with most built-in types
deepCopy.date.setFullYear(2024);
console.log(original.date.getFullYear()); // Original year

// Limitations: functions, symbols, DOM nodes not supported
```

3. Custom Deep Copy Function:

```
function deepCopy(obj, visited = new WeakMap()) {
    // Handle primitives and null
    if (obj === null || typeof obj !== 'object') {
        return obj;
    }

    // Handle circular references
    if (visited.has(obj)) {
        return visited.get(obj);
    }

    // Handle Date
    if (obj instanceof Date) {
        return new Date(obj.getTime());
    }
```

```
    // Handle RegExp
    if (obj instanceof RegExp) {
        return new RegExp(obj.source, obj.flags);
    }

    // Handle Arrays
    if (Array.isArray(obj)) {
        const copy = [];
        visited.set(obj, copy);
        for (let i = 0; i < obj.length; i++) {
            copy[i] = deepCopy(obj[i], visited);
        }
        return copy;
    }

    // Handle Objects
    const copy = {};
    visited.set(obj, copy);

    for (const key in obj) {
        if (obj.hasOwnProperty(key)) {
            copy[key] = deepCopy(obj[key], visited);
        }
    }

    return copy;
}

// Usage
const original = {
    name: 'John',
    date: new Date(),
    nested: { value: 42 }
};

const copy = deepCopy(original);
```

```
copy.nested.value = 100;
console.log(original.nested.value); // 42 - unchanged
```

4. Using Lodash:

```
const _ = require('lodash');

const original = {
    name: 'John',
    address: { city: 'NYC' },
    hobbies: ['reading']
};

const deepCopy = _.cloneDeep(original);
deepCopy.address.city = 'Boston';
console.log(original.address.city); // 'NYC' - unchanged
```

Array Copying:

```
const originalArray = [1, [2, 3], { a: 4 }];

// Shallow copy methods
const shallow1 = [...originalArray];
const shallow2 = originalArray.slice();
const shallow3 = Array.from(originalArray);

// Deep copy methods
const deep1 = JSON.parse(JSON.stringify(originalArray));
const deep2 = structuredClone(originalArray);

// Test shallow copy
shallow1[1][0] = 999;
console.log(originalArray[1][0]); // 999 - original changed
```

```
// Test deep copy
deep1[1][0] = 888;
console.log(originalArray[1][0]); // Still 999 - original u
```

Performance Comparison:

```
const largeObject = {
    // ... large nested object
};

console.time('JSON method');
const copy1 = JSON.parse(JSON.stringify(largeObject));
console.timeEnd('JSON method');

console.time('structuredClone');
const copy2 = structuredClone(largeObject);
console.timeEnd('structuredClone');

console.time('custom deep copy');
const copy3 = deepCopy(largeObject);
console.timeEnd('custom deep copy');
```

When to Use Each:

Shallow Copy:

- When you only need to copy the top-level properties
- When nested objects are immutable
- For performance reasons with large objects
- When you want to preserve references to shared objects

Deep Copy:

- When you need complete independence from the original

- When dealing with nested mutable objects
- When implementing undo/redo functionality
- When creating templates or default configurations

Best Practices:

1. Use `structuredClone()` for modern browsers
2. Use `JSON.parse(JSON.stringify())` for simple objects
3. Use libraries like Lodash for complex scenarios
4. Be aware of performance implications with large objects
5. Consider immutable data structures for frequent copying

Related Questions: Object Cloning, Spread Operator

⬆️ Back to Top

---

23. How do you clone an object in JavaScript?

Object cloning can be **shallow** (copying only the first level) or **deep** (copying all nested levels). Here are the most common methods:

Shallow Cloning Methods:

1. Spread Operator (ES2018) - Recommended:

```
const original = { name: 'John', age: 30, city: 'NYC' };
const clone = { ...original };

clone.name = 'Jane';
console.log(original.name); // 'John' - unchanged
```

2. Object.assign():

```
const original = { name: 'John', age: 30 };
const clone = Object.assign({}, original);

// Can also merge multiple objects
const clone2 = Object.assign({}, original, { country: 'USA'
```

3. Object.create() with Property Descriptors:

```
const original = { name: 'John', age: 30 };
const clone = Object.create(
    Object.getPrototypeOf(original),
    Object.getOwnPropertyDescriptors(original)
);
```

Deep Cloning Methods:

1. structuredClone() - Modern Native (ES2022):

```
const original = {
    name: 'John',
    address: { city: 'NYC', zip: 10001 },
    hobbies: ['reading', 'swimming'],
    date: new Date(),
    regex: /test/gi
};

const deepClone = structuredClone(original);
deepClone.address.city = 'Boston';
console.log(original.address.city); // 'NYC' - unchanged

// Supports most built-in types
console.log(deepClone.date instanceof Date); // true
console.log(deepClone.regex instanceof RegExp); // true
```

2. JSON Methods - Simple but Limited:

```
const original = {
    name: 'John',
    address: { city: 'NYC' },
    numbers: [1, 2, 3]
};

const deepClone = JSON.parse(JSON.stringify(original));

// Limitations - these will be lost or changed:
const problematic = {
    func: () => 'hello',        // Functions ignored
    undef: undefined,           // undefined ignored
    symbol: Symbol('test'),     // Symbols ignored
    date: new Date(),          // Becomes string
    regex: /test/g,            // Becomes {}
    infinity: Infinity,        // Becomes null
    nan: NaN                   // Becomes null
};
```

3. Custom Deep Clone Function:

```
function deepClone(obj, visited = new WeakMap()) {
    // Handle null and primitives
    if (obj === null || typeof obj !== 'object') {
        return obj;
    }

    // Handle circular references
    if (visited.has(obj)) {
        return visited.get(obj);
    }

    // Handle built-in object types
```

```
    if (obj instanceof Date) return new Date(obj);
    if (obj instanceof RegExp) return new RegExp(obj);
    if (obj instanceof Map) {
        const clonedMap = new Map();
        visited.set(obj, clonedMap);
        for (const [key, value] of obj) {
            clonedMap.set(deepClone(key, visited), deepClon
        }
        return clonedMap;
    }
    if (obj instanceof Set) {
        const clonedSet = new Set();
        visited.set(obj, clonedSet);
        for (const value of obj) {
            clonedSet.add(deepClone(value, visited));
        }
        return clonedSet;
    }

    // Handle arrays
    if (Array.isArray(obj)) {
        const clonedArray = [];
        visited.set(obj, clonedArray);
        for (let i = 0; i < obj.length; i++) {
            clonedArray[i] = deepClone(obj[i], visited);
        }
        return clonedArray;
    }

    // Handle plain objects
    const clonedObj = {};
    visited.set(obj, clonedObj);

    for (const key in obj) {
        if (obj.hasOwnProperty(key)) {
            clonedObj[key] = deepClone(obj[key], visited);
        }
```

```
    }

    return clonedObj;
}
```

Specialized Cloning Scenarios:

1. Array Cloning:

```
const originalArray = [1, 2, [3, 4], { a: 5 }];

// Shallow cloning
const shallow1 = [...originalArray];
const shallow2 = originalArray.slice();
const shallow3 = Array.from(originalArray);

// Deep cloning
const deep1 = structuredClone(originalArray);
const deep2 = JSON.parse(JSON.stringify(originalArray));
```

2. Function Cloning (Advanced):

```
function cloneFunction(fn) {
    // For named functions
    if (fn.name) {
        return eval(`(${fn.toString()})`);
    }

    // For anonymous functions
    return new Function('return ' + fn.toString())();
}

const original = function multiply(a, b) { return a * b; };
```

```
const cloned = cloneFunction(original);
console.log(cloned(3, 4)); // 12
```

3. Class Instance Cloning:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hello, I'm ${this.name}`;
    }
}

function cloneInstance(instance) {
    const cloned = Object.create(Object.getPrototypeOf(inst
    return Object.assign(cloned, instance);
}

const john = new Person('John', 30);
const johnClone = cloneInstance(john);
console.log(johnClone.greet()); // "Hello, I'm John"
```

Performance Considerations:

```
const testObject = {
    // Large nested object for testing
    data: Array.from({length: 1000}, (_, i) => ({
        id: i,
        nested: { value: i * 2 }
    }))
};
```

```
// Benchmark different methods
console.time('Spread (shallow)');
const spread = { ...testObject };
console.timeEnd('Spread (shallow)');

console.time('JSON method');
const jsonClone = JSON.parse(JSON.stringify(testObject));
console.timeEnd('JSON method');

console.time('structuredClone');
const structuredClone = structuredClone(testObject);
console.timeEnd('structuredClone');
```

Library Solutions:

```
// Lodash
const _ = require('lodash');
const deepClone = _.cloneDeep(original);

// Ramda
const R = require('ramda');
const clone = R.clone(original); // shallow
const deepClone2 = R.clone(original); // Ramda's clone is a
```

Choosing the Right Method:

Method	Use Case	Pros	Cons
`{...obj}`	Simple shallow copy	Fast, readable	Only shallow

Method	Use Case	Pros	Cons
`Object.assign()`	Shallow with merge	Can merge objects	Only shallow
`JSON.parse(JSON.stringify())`	Simple deep copy	Works everywhere	Limited types
`structuredClone()`	Modern deep copy	Handles most types	New API
Custom function	Complex requirements	Full control	More code
Libraries	Production apps	Battle-tested	Extra dependency

Best Practices:

1. Use spread operator for shallow copying
2. Use `structuredClone()` for deep copying in modern environments
3. Be aware of the limitations of JSON methods
4. Consider performance implications for large objects
5. Handle circular references in custom implementations
6. Use libraries for complex production scenarios

Related Questions: Shallow vs Deep Copy, Spread Operator

⬆️ Back to Top

---

24. What is destructuring assignment?

Destructuring assignment is a JavaScript expression that makes it possible to unpack values from arrays or properties from objects into distinct variables.

Array Destructuring:

Basic Array Destructuring:

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

// Traditional way
const first = numbers[0];
const second = numbers[1];

// Destructuring way
const [a, b, c] = numbers;
console.log(a, b, c); // 1, 2, 3

// Skip elements
const [first, , third] = numbers;
console.log(first, third); // 1, 3

// Rest elements
const [head, ...tail] = numbers;
console.log(head); // 1
console.log(tail); // [2, 3, 4, 5]
```

Object Destructuring:

Basic Object Destructuring:

```
const person = {
    name: 'John',
    age: 30,
    city: 'New York',
    country: 'USA'
```

```
};

// Traditional way
const name = person.name;
const age = person.age;

// Destructuring way
const { name, age, city } = person;
console.log(name, age, city); // 'John', 30, 'New York'

// Rename variables
const { name: fullName, age: years } = person;
console.log(fullName, years); // 'John', 30

// Default values
const { name, profession = 'Unknown' } = person;
console.log(profession); // 'Unknown'
```

Advanced Destructuring Patterns:

1. Nested Destructuring:

```
const user = {
    id: 1,
    name: 'John',
    address: {
        street: '123 Main St',
        city: 'New York',
        coordinates: {
            lat: 40.7128,
            lng: -74.0060
        }
    },
    hobbies: ['reading', 'swimming', 'coding']
};
```

```
// Nested object destructuring
const {
    name,
    address: {
        city,
        coordinates: { lat, lng }
    },
    hobbies: [firstHobby, secondHobby]
} = user;

console.log(name);        // 'John'
console.log(city);        // 'New York'
console.log(lat, lng);    // 40.7128, -74.0060
console.log(firstHobby);  // 'reading'
```

2. Function Parameter Destructuring:

```
// Object parameter destructuring
function greetUser({ name, age, city = 'Unknown' }) {
    return `Hello ${name}, you are ${age} years old from ${
}

const user = { name: 'John', age: 30 };
console.log(greetUser(user)); // "Hello John, you are 30 ye

// Array parameter destructuring
function calculateTotal([price, tax, discount = 0]) {
    return price + tax - discount;
}

console.log(calculateTotal([100, 10, 5])); // 105

// Mixed destructuring
function processOrder({ items: [firstItem], customer: { nam
    return `Processing ${firstItem} for ${name}`;
}
```

```
const order = {
    items: ['laptop', 'mouse'],
    customer: { name: 'John', email: 'john@example.com' }
};

console.log(processOrder(order)); // "Processing laptop for
```

3. Swapping Variables:

```
let a = 1;
let b = 2;

// Traditional swap
let temp = a;
a = b;
b = temp;

// Destructuring swap
[a, b] = [b, a];
console.log(a, b); // 2, 1

// Multiple variable swap
let x = 1, y = 2, z = 3;
[x, y, z] = [z, x, y];
console.log(x, y, z); // 3, 1, 2
```

4. Extracting Multiple Values from Functions:

```
function getCoordinates() {
    return [40.7128, -74.0060];
}

function getUserInfo() {
```

```
    return {
        name: 'John',
        age: 30,
        email: 'john@example.com'
    };
}

// Array destructuring from function return
const [latitude, longitude] = getCoordinates();

// Object destructuring from function return
const { name, email } = getUserInfo();
```

5. Rest Pattern in Destructuring:

```
// Array rest
const numbers = [1, 2, 3, 4, 5];
const [first, second, ...rest] = numbers;
console.log(first);  // 1
console.log(second); // 2
console.log(rest);   // [3, 4, 5]

// Object rest
const person = {
    name: 'John',
    age: 30,
    city: 'NYC',
    country: 'USA',
    profession: 'Developer'
};

const { name, age, ...otherInfo } = person;
console.log(name, age);    // 'John', 30
console.log(otherInfo);    // { city: 'NYC', country: 'USA'
```

Practical Use Cases:

1. API Response Handling:

```
// Typical API response
const apiResponse = {
    data: {
        users: [
            { id: 1, name: 'John', email: 'john@example.com
            { id: 2, name: 'Jane', email: 'jane@example.com
        ]
    },
    status: 'success',
    message: 'Data retrieved successfully'
};

// Extract what you need
const {
    data: { users },
    status,
    message
} = apiResponse;

// Process first user
const [{ name: firstName, email: firstEmail }] = users;
console.log(firstName, firstEmail); // 'John', 'john@exampl
```

2. Configuration Objects:

```
function createServer(options = {}) {
    const {
        port = 3000,
        host = 'localhost',
        ssl = false,
        middleware = [],
```

```
        routes = {}
    } = options;

    console.log(`Server starting on ${host}:${port}`);
    console.log(`SSL: ${ssl ? 'enabled' : 'disabled'}`);

    return { port, host, ssl, middleware, routes };
}

// Usage
const server = createServer({
    port: 8080,
    ssl: true
});
```

3. Module Imports:

```
// ES6 modules
import { useState, useEffect, useCallback } from 'react';
import { map, filter, reduce } from 'lodash';

// CommonJS
const { readFile, writeFile } = require('fs').promises;
const { join, resolve } = require('path');
```

4. Event Handling:

```
// DOM event destructuring
document.addEventListener('click', ({ target, clientX, clie
    console.log(`Clicked on ${target.tagName} at (${clientX
});

// Custom event data
function handleUserAction({ type, payload: { userId, action
```

```
    console.log(`User ${userId} performed ${action} of type
}

const eventData = {
    type: 'USER_ACTION',
    payload: {
        userId: 123,
        action: 'login'
    }
};

handleUserAction(eventData);
```

5. Array Processing:

```
const users = [
    ['John', 'Doe', 30],
    ['Jane', 'Smith', 25],
    ['Bob', 'Johnson', 35]
];

// Process each user
users.forEach(([firstName, lastName, age]) => {
    console.log(`${firstName} ${lastName} is ${age} years o
});

// Transform data
const userObjects = users.map(([firstName, lastName, age])
    firstName,
    lastName,
    age,
    fullName: `${firstName} ${lastName}`
}));
```

Common Patterns and Tricks:

1. Computed Property Names:

```
const key = 'dynamicKey';
const obj = {
    [key]: 'value',
    staticKey: 'static'
};

const { [key]: dynamicValue, staticKey } = obj;
console.log(dynamicValue); // 'value'
```

2. Conditional Destructuring:

```
const data = { name: 'John', age: 30 };

// Only destructure if object exists
const { name, age } = data || {};

// With default object
function processUser(user = {}) {
    const { name = 'Anonymous', age = 0 } = user;
    return `${name} (${age})`;
}
```

3. Aliasing with Defaults:

```
const config = {
    api: {
        baseUrl: 'https://api.example.com'
    }
};

const {
```

```
    api: { baseUrl: apiUrl = 'http://localhost:3000' } = {},
    timeout = 5000
} = config;

console.log(apiUrl);  // 'https://api.example.com'
console.log(timeout); // 5000
```

Best Practices:

1. Use destructuring for cleaner, more readable code
2. Provide default values to prevent undefined errors
3. Use meaningful variable names when aliasing
4. Don't over-nest destructuring (keep it readable)
5. Use rest patterns to collect remaining properties
6. Combine with other ES6+ features for powerful patterns

Related Questions: Spread Operator, Object Manipulation

⬆️ Back to Top

---

25. What is the spread operator and rest parameters?

The **spread operator (`...`)** and **rest parameters** use the same syntax but serve opposite purposes: spread expands elements, while rest collects them.

Spread Operator:

1. Array Spreading:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

// Combine arrays
```

```
const combined = [...arr1, ...arr2];
console.log(combined); // [1, 2, 3, 4, 5, 6]

// Insert elements
const inserted = [...arr1, 'middle', ...arr2];
console.log(inserted); // [1, 2, 3, 'middle', 4, 5, 6]

// Copy array (shallow)
const copy = [...arr1];
copy.push(4);
console.log(arr1); // [1, 2, 3] - original unchanged

// Convert string to array
const letters = [...'hello'];
console.log(letters); // ['h', 'e', 'l', 'l', 'o']

// Find max/min
const numbers = [1, 5, 3, 9, 2];
const max = Math.max(...numbers);
const min = Math.min(...numbers);
console.log(max, min); // 9, 1
```

2. Object Spreading:

```
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };

// Combine objects
const combined = { ...obj1, ...obj2 };
console.log(combined); // { a: 1, b: 2, c: 3, d: 4 }

// Override properties
const overridden = { ...obj1, b: 'new value', e: 5 };
console.log(overridden); // { a: 1, b: 'new value', e: 5 }

// Copy object (shallow)
```

```
const copy = { ...obj1 };
copy.a = 10;
console.log(obj1.a); // 1 - original unchanged

// Conditional properties
const includeOptional = true;
const result = {
    required: 'value',
    ...(includeOptional && { optional: 'included' })
};
console.log(result); // { required: 'value', optional: 'inc
```

3. Function Call Spreading:

```
function sum(a, b, c) {
    return a + b + c;
}

const numbers = [1, 2, 3];

// Traditional way
const result1 = sum.apply(null, numbers);

// Spread way
const result2 = sum(...numbers);
console.log(result2); // 6

// With mixed arguments
const result3 = sum(...numbers.slice(0, 2), 10);
console.log(result3); // 1 + 2 + 10 = 13
```

Rest Parameters:

1. Function Rest Parameters:

```
// Collect remaining parameters
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // 15
console.log(sum(10, 20));         // 30
console.log(sum());               // 0

// Mixed parameters
function greet(greeting, ...names) {
    return `${greeting} ${names.join(', ')}!`;
}

console.log(greet('Hello', 'John', 'Jane', 'Bob'));
// "Hello John, Jane, Bob!"

// Rest must be last parameter
function invalidFunction(...rest, last) {} // SyntaxError
```

2. Array Rest in Destructuring:

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

// Collect remaining elements
const [first, second, ...rest] = numbers;
console.log(first);  // 1
console.log(second); // 2
console.log(rest);   // [3, 4, 5]

// Skip elements
const [, , ...remaining] = numbers;
console.log(remaining); // [3, 4, 5]

// Empty rest
```

```
const [a, b, c, d, e, ...empty] = numbers;
console.log(empty); // []
```

3. Object Rest in Destructuring:

```
const person = {
    name: 'John',
    age: 30,
    city: 'NYC',
    country: 'USA',
    profession: 'Developer'
};

// Collect remaining properties
const { name, age, ...otherInfo } = person;
console.log(name, age);    // 'John', 30
console.log(otherInfo);    // { city: 'NYC', country: 'USA'

// Exclude specific properties
const { profession, ...personalInfo } = person;
console.log(personalInfo); // { name: 'John', age: 30, city
```

Practical Applications:

1. Function Overloading Pattern:

```
function createUser(name, ...options) {
    const [age, email, role = 'user'] = options;

    return {
        name,
        age: age || null,
        email: email || null,
        role
```

```
    };
}

// Different ways to call
console.log(createUser('John'));                    // Basi
console.log(createUser('Jane', 25));                // With
console.log(createUser('Bob', 30, 'bob@example.com')); // W
console.log(createUser('Alice', 28, 'alice@example.com', 'a
```

2. Array Manipulation:

```
const originalArray = [1, 2, 3];

// Add elements immutably
const addToStart = [0, ...originalArray];
const addToEnd = [...originalArray, 4];
const addToMiddle = [...originalArray.slice(0, 2), 2.5, ...

console.log(addToStart);  // [0, 1, 2, 3]
console.log(addToEnd);    // [1, 2, 3, 4]
console.log(addToMiddle); // [1, 2, 2.5, 3]

// Remove duplicates
const withDuplicates = [1, 2, 2, 3, 3, 4];
const unique = [...new Set(withDuplicates)];
console.log(unique); // [1, 2, 3, 4]

// Flatten one level
const nested = [[1, 2], [3, 4], [5, 6]];
const flattened = [].concat(...nested);
console.log(flattened); // [1, 2, 3, 4, 5, 6]
```

3. Object Updates (Immutable):

```
const user = {
    id: 1,
    name: 'John',
    settings: {
        theme: 'dark',
        notifications: true
    }
};

// Update top-level property
const updatedUser = {
    ...user,
    name: 'John Doe',
    lastLogin: new Date()
};

// Update nested property (careful - need deep spread)
const updatedSettings = {
    ...user,
    settings: {
        ...user.settings,
        theme: 'light'
    }
};

// Remove property
const { settings, ...userWithoutSettings } = user;
console.log(userWithoutSettings); // { id: 1, name: 'John'
```

4. Component Props (React-style):

```
function Button({ children, className, ...otherProps }) {
    return {
        type: 'button',
        className: `btn ${className || ''}`,
```

```
        children,
        ...otherProps // Pass through all other props
    };
}

// Usage
const buttonProps = Button({
    children: 'Click me',
    className: 'primary',
    onClick: () => console.log('clicked'),
    disabled: false,
    'data-testid': 'submit-button'
});
```

5. API Data Processing:

```
function processApiResponse(response) {
    const { data, meta, ...otherFields } = response;

    return {
        items: data || [],
        pagination: meta?.pagination || {},
        additionalInfo: otherFields
    };
}

// Transform user data
function transformUsers(users) {
    return users.map(({ password, ...safeUser }) => ({
        ...safeUser,
        displayName: `${safeUser.firstName} ${safeUser.last
    }));
}
```

Advanced Patterns:

1. Conditional Spreading:

```
const baseConfig = { host: 'localhost', port: 3000 };
const isDevelopment = true;

const config = {
    ...baseConfig,
    ...(isDevelopment && { debug: true, verbose: true }),
    ...(process.env.SSL && { ssl: true, port: 443 })
};
```

2. Function Composition:

```
const pipe = (...functions) => (value) =>
    functions.reduce((acc, fn) => fn(acc), value);

const addOne = x => x + 1;
const double = x => x * 2;
const square = x => x * x;

const transform = pipe(addOne, double, square);
console.log(transform(3)); // ((3 + 1) * 2)^2 = 64
```

3. Dynamic Object Creation:

```
function createObject(keys, values) {
    return keys.reduce((obj, key, index) => ({
        ...obj,
        [key]: values[index]
    }), {});
}

const keys = ['name', 'age', 'city'];
```

```
const values = ['John', 30, 'NYC'];
const person = createObject(keys, values);
console.log(person); // { name: 'John', age: 30, city: 'NYC
```

Performance Considerations:

```
// Spread creates new objects/arrays - consider performance
const largeArray = Array.from({length: 100000}, (_, i) => i

console.time('spread');
const spreadCopy = [...largeArray];
console.timeEnd('spread');

console.time('slice');
const sliceCopy = largeArray.slice();
console.timeEnd('slice');

// For objects, spread is generally fast for shallow copyin
const largeObject = Object.fromEntries(
    Array.from({length: 1000}, (_, i) => [`key${i}`, i])
);

console.time('object spread');
const objectCopy = { ...largeObject };
console.timeEnd('object spread');
```

Best Practices:

1. Use spread for immutable updates
2. Use rest parameters for flexible function signatures
3. Be careful with nested objects - spread is shallow
4. Use meaningful names for rest parameters
5. Consider performance with very large data structures

6. Combine with destructuring for powerful patterns

Related Questions: Destructuring, Array Methods, Object Manipulation

⬆️ Back to Top

Prototypes and Inheritance

26. What are prototypes in JavaScript?

Prototypes are the mechanism by which JavaScript objects inherit features from one another. Every object in JavaScript has a prototype, which is another object that the original object inherits properties and methods from.

Understanding Prototypes:

1. Every Object Has a Prototype:

```
const obj = {};
console.log(obj.__proto__); // Object.prototype
console.log(Object.getPrototypeOf(obj)); // Object.prototyp

// Functions have Function.prototype
function myFunc() {}
console.log(myFunc.__proto__); // Function.prototype

// Arrays have Array.prototype
const arr = [];
console.log(arr.__proto__); // Array.prototype
```

2. Prototype Chain:

```
const person = {
    name: 'John',
```

```
    age: 30
};

// person inherits from Object.prototype
console.log(person.toString()); // "[object Object]" - inhe
console.log(person.hasOwnProperty('name')); // true - inher

// The chain: person -> Object.prototype -> null
console.log(Object.getPrototypeOf(person) === Object.protot
console.log(Object.getPrototypeOf(Object.prototype)); // nu
```

3. Function Prototypes:

```
function Person(name) {
    this.name = name;
}

// Functions have a special 'prototype' property
console.log(Person.prototype); // Person.prototype object
console.log(typeof Person.prototype); // "object"

// This prototype is used when creating instances with 'new
const john = new Person('John');
console.log(john.__proto__ === Person.prototype); // true
```

4. Adding Methods to Prototypes:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// Add methods to the prototype
Person.prototype.greet = function() {
```

```
    return `Hello, I'm ${this.name}`;
};

Person.prototype.getAge = function() {
    return this.age;
};

// All instances share the same prototype methods
const john = new Person('John', 30);
const jane = new Person('Jane', 25);

console.log(john.greet()); // "Hello, I'm John"
console.log(jane.greet()); // "Hello, I'm Jane"

// Methods are shared, not duplicated
console.log(john.greet === jane.greet); // true
```

5. Prototype Properties vs Instance Properties:

```
function Person(name) {
    this.name = name; // Instance property
}

Person.prototype.species = 'Homo sapiens'; // Prototype pro

const john = new Person('John');

console.log(john.name); // "John" - instance property
console.log(john.species); // "Homo sapiens" - prototype pr

// hasOwnProperty checks instance properties only
console.log(john.hasOwnProperty('name')); // true
console.log(john.hasOwnProperty('species')); // false
```

```
// Check if property exists in prototype chain
console.log('species' in john); // true
```

6. Built-in Object Prototypes:

```
// Array prototype
const arr = [1, 2, 3];
console.log(arr.push); // function - from Array.prototype
console.log(arr.toString); // function - from Object.protot

// String prototype
const str = 'hello';
console.log(str.toUpperCase); // function - from String.pro
console.log(str.charAt); // function - from String.prototyp

// Number prototype
const num = 42;
console.log(num.toFixed); // function - from Number.prototy
```

7. Creating Objects with Specific Prototypes:

```
// Using Object.create()
const animal = {
    type: 'animal',
    makeSound() {
        return 'Some sound';
    }
};

const dog = Object.create(animal);
dog.breed = 'Golden Retriever';
dog.makeSound = function() {
    return 'Woof!';
};
```

```
console.log(dog.type); // "animal" - inherited
console.log(dog.makeSound()); // "Woof!" - overridden
console.log(dog.breed); // "Golden Retriever" - own propert
```

8. Prototype Inspection:

```
function Person(name) {
    this.name = name;
}

Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};

const john = new Person('John');

// Check prototype
console.log(Object.getPrototypeOf(john) === Person.prototyp

// List prototype properties
console.log(Object.getOwnPropertyNames(Person.prototype));
// ['constructor', 'greet']

// Check if property exists in prototype chain
console.log('greet' in john); // true
console.log('toString' in john); // true (from Object.proto
```

Best Practices:

1. Use `Object.getPrototypeOf()` instead of `__proto__`
2. Add methods to prototypes, not instances
3. Use `hasOwnProperty()` to check instance properties
4. Be careful when modifying built-in prototypes

5. Use `Object.create()` for clean inheritance

Related Questions: Prototypal Inheritance, Prototype Chain

⬆️ Back to Top

---

27. How does prototypal inheritance work?

Prototypal inheritance is JavaScript's mechanism for object inheritance where objects can inherit properties and methods from other objects through the prototype chain.

How It Works:

1. Basic Prototype Chain:

```
// Parent object
const animal = {
    type: 'animal',
    makeSound() {
        return 'Some generic sound';
    },
    eat() {
        return 'Eating food';
    }
};

// Child object inherits from animal
const dog = Object.create(animal);
dog.breed = 'Golden Retriever';
dog.makeSound = function() {
    return 'Woof! Woof!';
};

console.log(dog.type); // "animal" - inherited
console.log(dog.eat()); // "Eating food" - inherited
```

```
console.log(dog.makeSound()); // "Woof! Woof!" - overridden
console.log(dog.breed); // "Golden Retriever" - own propert
```

2. Constructor Function Inheritance:

```
// Parent constructor
function Animal(name) {
    this.name = name;
    this.type = 'animal';
}

Animal.prototype.makeSound = function() {
    return 'Some generic sound';
};

Animal.prototype.eat = function() {
    return `${this.name} is eating`;
};

// Child constructor
function Dog(name, breed) {
    Animal.call(this, name); // Call parent constructor
    this.breed = breed;
}

// Set up inheritance
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Add child-specific methods
Dog.prototype.makeSound = function() {
    return 'Woof! Woof!';
};

Dog.prototype.fetch = function() {
    return `${this.name} is fetching the ball`;
```

```
};

// Create instances
const buddy = new Dog('Buddy', 'Golden Retriever');
console.log(buddy.name); // "Buddy"
console.log(buddy.type); // "animal" - inherited
console.log(buddy.makeSound()); // "Woof! Woof!" - overridd
console.log(buddy.eat()); // "Buddy is eating" - inherited
console.log(buddy.fetch()); // "Buddy is fetching the ball"
```

3. ES6 Class Inheritance:

```
// Parent class
class Animal {
    constructor(name) {
        this.name = name;
        this.type = 'animal';
    }

    makeSound() {
        return 'Some generic sound';
    }

    eat() {
        return `${this.name} is eating`;
    }
}

// Child class
class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Call parent constructor
        this.breed = breed;
    }

    makeSound() {
```

```
        return 'Woof! Woof!';
    }

    fetch() {
        return `${this.name} is fetching the ball`;
    }
}

const buddy = new Dog('Buddy', 'Golden Retriever');
console.log(buddy instanceof Dog); // true
console.log(buddy instanceof Animal); // true
console.log(buddy instanceof Object); // true
```

4. Property Lookup Process:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';
Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};

const john = new Person('John');

// Property lookup process:
// 1. Check instance properties
console.log(john.name); // "John" - found in instance

// 2. Check prototype properties
console.log(john.species); // "Homo sapiens" - found in Per

// 3. Check prototype's prototype (Object.prototype)
console.log(john.toString); // function - found in Object.p
```

```
// 4. Continue up the chain until null
console.log(john.nonExistent); // undefined - not found any
```

5. Method Overriding:

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.makeSound = function() {
    return 'Some generic sound';
};

function Dog(name) {
    Animal.call(this, name);
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Override parent method
Dog.prototype.makeSound = function() {
    return 'Woof! Woof!';
};

// Call parent method from child
Dog.prototype.makeSoundAndEat = function() {
    return `${this.makeSound()} and ${Animal.prototype.eat.
};

const buddy = new Dog('Buddy');
console.log(buddy.makeSound()); // "Woof! Woof!" - overridd
```

6. Multiple Inheritance (Mixins):

```
// Mixin objects
const Flyable = {
    fly() {
        return `${this.name} is flying`;
    }
};

const Swimmable = {
    swim() {
        return `${this.name} is swimming`;
    }
};

// Base class
function Animal(name) {
    this.name = name;
}

// Mix in capabilities
Object.assign(Animal.prototype, Flyable);

function Duck(name) {
    Animal.call(this, name);
}

Duck.prototype = Object.create(Animal.prototype);
Duck.prototype.constructor = Duck;

// Mix in additional capabilities
Object.assign(Duck.prototype, Swimmable);

const donald = new Duck('Donald');
console.log(donald.fly()); // "Donald is flying"
console.log(donald.swim()); // "Donald is swimming"
```

7. Prototype Chain Inspection:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';

const john = new Person('John');

// Check the prototype chain
let current = john;
let level = 0;

while (current !== null) {
    console.log(`Level ${level}:`, current.constructor.name
    console.log('Properties:', Object.getOwnPropertyNames(c
    current = Object.getPrototypeOf(current);
    level++;
}

// Output:
// Level 0: Person
// Properties: ['name']
// Level 1: Object
// Properties: ['constructor', 'species', 'greet', ...]
// Level 2: null
```

Key Concepts:

1. **Delegation**: Objects delegate property access to their prototype
2. **Dynamic**: Prototype chain is checked at runtime
3. **Mutable**: Prototypes can be modified after creation
4. **Flexible**: Multiple inheritance through mixins

5. **Efficient**: Methods are shared, not duplicated

Best Practices:

1. Use `Object.create()` for clean inheritance
2. Always set `constructor` property when overriding prototypes
3. Use `call()` or `apply()` to invoke parent constructors
4. Prefer composition over deep inheritance chains
5. Use ES6 classes for cleaner syntax
6. Be careful with prototype pollution

Related Questions: Prototypes, ES6 Classes, Prototype Chain

Back to Top

---

28. What's the difference between `__proto__` and `prototype`?

The key difference is that `__proto__` is a property of object instances that points to their prototype, while `prototype` is a property of constructor functions that defines what prototype new instances will have.

`__proto__` **- Instance Property:**

```
const obj = {};
console.log(obj.__proto__); // Object.prototype

const arr = [];
console.log(arr.__proto__); // Array.prototype

function Person(name) {
    this.name = name;
}

const john = new Person('John');
```

```
console.log(john.__proto__); // Person.prototype
console.log(john.__proto__ === Person.prototype); // true
```

`prototype` - Constructor Property:

```
function Person(name) {
    this.name = name;
}

// prototype is a property of the constructor function
console.log(Person.prototype); // Person.prototype object
console.log(typeof Person.prototype); // "object"

// Only functions have prototype property
console.log(Person.prototype.constructor === Person); // tr
```

Key Differences:

1. What They Are:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';

const john = new Person('John');

// __proto__ is on the instance
console.log(john.__proto__); // Person.prototype

// prototype is on the constructor
console.log(Person.prototype); // Person.prototype
```

```
// They point to the same object
console.log(john.__proto__ === Person.prototype); // true
```

2. When They're Used:

```
function Animal(name) {
    this.name = name;
}

// prototype is used when creating new instances
Animal.prototype.makeSound = function() {
    return 'Some sound';
};

const dog = new Animal('Buddy');

// __proto__ is used for property lookup
console.log(dog.makeSound()); // "Some sound" - found via _
console.log(dog.__proto__.makeSound()); // "Some sound" - s
```

3. Modifying Prototypes:

```
function Person(name) {
    this.name = name;
}

const john = new Person('John');

// Modify constructor's prototype
Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};

// This affects all instances (existing and new)
```

```
console.log(john.greet()); // "Hello, I'm John"

// Modify instance's __proto__ (not recommended)
john.__proto__.sayGoodbye = function() {
    return `Goodbye, ${this.name}`;
};

// This affects all instances of the same constructor
const jane = new Person('Jane');
console.log(jane.sayGoodbye()); // "Goodbye, Jane"
```

4. Modern vs Legacy:

```
const obj = {};

// Legacy way (deprecated)
console.log(obj.__proto__); // Object.prototype

// Modern way (recommended)
console.log(Object.getPrototypeOf(obj)); // Object.prototyp

// Setting prototype
const newProto = { newMethod: () => 'new method' };

// Legacy way (deprecated)
obj.__proto__ = newProto;

// Modern way (recommended)
Object.setPrototypeOf(obj, newProto);
```

5. Function vs Object:

```
function Person(name) {
    this.name = name;
```

```
}

// Functions have BOTH __proto__ and prototype
console.log(Person.__proto__); // Function.prototype
console.log(Person.prototype); // Person.prototype object

// Regular objects only have __proto__
const obj = {};
console.log(obj.__proto__); // Object.prototype
console.log(obj.prototype); // undefined
```

6. Inheritance Setup:

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.eat = function() {
    return `${this.name} is eating`;
};

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

// Set up inheritance using prototype
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
    return 'Woof!';
};

const buddy = new Dog('Buddy', 'Golden Retriever');
```

```
// __proto__ chain: buddy -> Dog.prototype -> Animal.protot
console.log(buddy.__proto__ === Dog.prototype); // true
console.log(buddy.__proto__.__proto__ === Animal.prototype)
```

7. Property Lookup:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';

const john = new Person('John');

// Property lookup process:
// 1. Check instance properties
console.log(john.name); // "John" - found in instance

// 2. Check __proto__ (Person.prototype)
console.log(john.species); // "Homo sapiens" - found in __p

// 3. Check __proto__.__proto__ (Object.prototype)
console.log(john.toString); // function - found in __proto_

// Manual lookup using __proto__
console.log(john.__proto__.species); // "Homo sapiens"
console.log(john.__proto__.__proto__.toString); // function
```

8. Common Mistakes:

```
function Person(name) {
    this.name = name;
}
```

```
const john = new Person('John');

// WRONG: Trying to access prototype on instance
console.log(john.prototype); // undefined

// CORRECT: Access prototype via __proto__ or Object.getPro
console.log(john.__proto__); // Person.prototype
console.log(Object.getPrototypeOf(john)); // Person.prototy

// WRONG: Modifying __proto__ directly
john.__proto__ = { newMethod: () => 'new' }; // Not recomme

// CORRECT: Modify constructor's prototype
Person.prototype.newMethod = () => 'new';
```

9. ES6 Classes:

```
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        return `Hello, I'm ${this.name}`;
    }
}

const john = new Person('John');

// ES6 classes still use prototypes under the hood
console.log(john.__proto__ === Person.prototype); // true
console.log(Person.prototype.greet); // function

// The class syntax is just syntactic sugar
console.log(typeof Person); // "function"
```

10. Prototype Chain Visualization:

```
function Animal(name) {
    this.name = name;
}

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

const buddy = new Dog('Buddy', 'Golden Retriever');

// Visualize the chain
console.log('buddy.__proto__ === Dog.prototype:', buddy.__p
console.log('Dog.prototype.__proto__ === Animal.prototype:'
console.log('Animal.prototype.__proto__ === Object.prototyp
console.log('Object.prototype.__proto__ === null:', Object.
```

Summary Table:

Aspect	`__proto__`	`prototype`
Location	On object instances	On constructor functions
Purpose	Points to object's prototype	Defines prototype for new instances
Access	`obj.__proto__`	`Constructor.prototype`

Aspect	`__proto__`	`prototype`
Modification	`obj.__proto__ = newProto`	`Constructor.prototype = newProto`
Modern Alternative	`Object.getPrototypeOf(obj)`	Still `Constructor.prototype`
Available On	All objects	Only functions
Used For	Property lookup	Inheritance setup

Best Practices:

1. Use `Object.getPrototypeOf()` instead of `__proto__`
2. Use `Object.setPrototypeOf()` instead of `__proto__ =`
3. Modify `prototype` for inheritance setup
4. Don't modify `__proto__` directly
5. Understand that ES6 classes use prototypes under the hood

Related Questions: Prototypes, Prototypal Inheritance, ES6 Classes

⬆️ Back to Top

---

29. How do ES6 classes relate to prototypes?

ES6 classes are **syntactic sugar** over JavaScript's existing prototype-based inheritance. They provide a cleaner syntax but still use prototypes under the hood.

Class Syntax vs Prototype:

1. Basic Class Declaration:

```
// ES6 Class
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hello, I'm ${this.name}`;
    }

    getAge() {
        return this.age;
    }
}

// Equivalent Constructor Function
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};

Person.prototype.getAge = function() {
    return this.age;
};

// Both create the same prototype structure
const john1 = new Person('John', 30); // Class
const john2 = new Person('John', 30); // Constructor

console.log(john1.__proto__ === Person.prototype); // true
```

```
console.log(john2.__proto__ === Person.prototype); // true
console.log(john1.greet === john2.greet); // true
```

2. Class Inheritance:

```
// ES6 Class Inheritance
class Animal {
    constructor(name) {
        this.name = name;
    }

    makeSound() {
        return 'Some generic sound';
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }

    makeSound() {
        return 'Woof! Woof!';
    }

    fetch() {
        return `${this.name} is fetching`;
    }
}

// Equivalent Prototype Inheritance
function Animal(name) {
    this.name = name;
}
```

```
Animal.prototype.makeSound = function() {
    return 'Some generic sound';
};

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.makeSound = function() {
    return 'Woof! Woof!';
};

Dog.prototype.fetch = function() {
    return `${this.name} is fetching`;
};
```

3. Static Methods:

```
// ES6 Class with Static Methods
class MathUtils {
    static add(a, b) {
        return a + b;
    }

    static multiply(a, b) {
        return a * b;
    }
}

// Equivalent Constructor Function
function MathUtils() {}
```

```
MathUtils.add = function(a, b) {
    return a + b;
};

MathUtils.multiply = function(a, b) {
    return a * b;
};

// Usage is identical
console.log(MathUtils.add(2, 3)); // 5
console.log(MathUtils.multiply(4, 5)); // 20
```

4. Getters and Setters:

```
// ES6 Class with Getters/Setters
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }

    set fullName(name) {
        [this.firstName, this.lastName] = name.split(' ');
    }
}

// Equivalent with Object.defineProperty
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
```

```
Object.defineProperty(Person.prototype, 'fullName', {
    get: function() {
        return `${this.firstName} ${this.lastName}`;
    },
    set: function(name) {
        [this.firstName, this.lastName] = name.split(' ');
    },
    enumerable: true,
    configurable: true
});

const person = new Person('John', 'Doe');
console.log(person.fullName); // "John Doe"
person.fullName = 'Jane Smith';
console.log(person.firstName); // "Jane"
```

5. Private Fields (ES2022):

```
// ES6 Class with Private Fields
class BankAccount {
    #balance = 0;
    #accountNumber;

    constructor(accountNumber, initialBalance = 0) {
        this.#accountNumber = accountNumber;
        this.#balance = initialBalance;
    }

    deposit(amount) {
        this.#balance += amount;
        return this.#balance;
    }

    getBalance() {
        return this.#balance;
    }
```

```
}

// Private fields are not accessible outside the class
const account = new BankAccount('12345', 1000);
console.log(account.getBalance()); // 1000
// console.log(account.#balance); // SyntaxError: Private f
```

6. Class Methods vs Prototype Methods:

```
class Person {
    constructor(name) {
        this.name = name;
    }

    // Instance method (goes to prototype)
    greet() {
        return `Hello, I'm ${this.name}`;
    }

    // Static method (goes to constructor)
    static createAnonymous() {
        return new Person('Anonymous');
    }
}

const john = new Person('John');

// Instance method is on prototype
console.log(john.greet); // function - from Person.prototyp
console.log(john.__proto__.greet); // function - same thing

// Static method is on constructor
console.log(Person.createAnonymous); // function - from Per
console.log(john.createAnonymous); // undefined - not on in
```

7. Class Hoisting:

```
// Classes are NOT hoisted like functions
// console.log(MyClass); // ReferenceError: Cannot access '

class MyClass {
    constructor() {
        this.value = 42;
    }
}

// Functions ARE hoisted
console.log(MyFunction); // function MyFunction() { ... }

function MyFunction() {
    this.value = 42;
}
```

8. Class vs Constructor Function:

```
// Class
class Person {
    constructor(name) {
        this.name = name;
    }
}

// Constructor Function
function Person(name) {
    this.name = name;
}

// Both create similar objects, but classes have stricter b
const classInstance = new Person('John');
const funcInstance = new Person('John');
```

```
console.log(classInstance instanceof Person); // true
console.log(funcInstance instanceof Person); // true

// Classes must be called with 'new'
// Person('John'); // TypeError: Class constructor Person c

// Constructor functions can be called without 'new' (thoug
const funcInstance2 = Person('Jane'); // Works, but 'this'
```

9. Class Inheritance Chain:

```
class Animal {
    constructor(name) {
        this.name = name;
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }
}

const buddy = new Dog('Buddy', 'Golden Retriever');

// Prototype chain: buddy -> Dog.prototype -> Animal.protot
console.log(buddy.__proto__ === Dog.prototype); // true
console.log(Dog.prototype.__proto__ === Animal.prototype);
console.log(Animal.prototype.__proto__ === Object.prototype

// instanceof works with classes
console.log(buddy instanceof Dog); // true
```

```
console.log(buddy instanceof Animal); // true
console.log(buddy instanceof Object); // true
```

10. Mixins with Classes:

```
// Mixin functions
const Flyable = {
    fly() {
        return `${this.name} is flying`;
    }
};

const Swimmable = {
    swim() {
        return `${this.name} is swimming`;
    }
};

// Mixin helper function
function mixin(target, ...sources) {
    Object.assign(target.prototype, ...sources);
}

class Animal {
    constructor(name) {
        this.name = name;
    }
}

class Duck extends Animal {
    constructor(name) {
        super(name);
    }
}

// Apply mixins
```

```
mixin(Duck, Flyable, Swimmable);

const donald = new Duck('Donald');
console.log(donald.fly()); // "Donald is flying"
console.log(donald.swim()); // "Donald is swimming"
```

Key Differences:

Feature	ES6 Classes	Constructor Functions
Syntax	Clean, familiar	Verbose
Hoisting	Not hoisted	Hoisted
Strict Mode	Always in strict mode	Depends on context
Call without `new`	Throws error	Works (but problematic)
Private Fields	Supported (ES2022)	Not supported
Super	Built-in `super` keyword	Manual `Parent.call(this)`
Prototype	Still uses prototypes	Uses prototypes

Best Practices:

1. Use classes for cleaner syntax
2. Remember that classes are still prototypes under the hood
3. Use `super()` in child constructors
4. Prefer composition over deep inheritance
5. Use private fields for encapsulation

6. Don't forget that classes are not hoisted

Related Questions: Prototypes, Prototypal Inheritance, Prototype Chain

⬆️ Back to Top

---

30. What is the prototype chain?

The **prototype chain** is the mechanism by which JavaScript objects inherit properties and methods from other objects. When a property is accessed on an object, JavaScript searches up the prototype chain until it finds the property or reaches the end of the chain.

How the Prototype Chain Works:

1. Basic Chain Structure:

```
const obj = {};
console.log(obj.__proto__); // Object.prototype
console.log(obj.__proto__.__proto__); // null

// Chain: obj -> Object.prototype -> null
```

2. Property Lookup Process:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';
Person.prototype.greet = function() {
    return `Hello, I'm ${this.name}`;
};
```

```
const john = new Person('John');

// Property lookup follows this order:
// 1. Check instance properties
console.log(john.name); // "John" - found in instance

// 2. Check Person.prototype
console.log(john.species); // "Homo sapiens" - found in pro

// 3. Check Object.prototype
console.log(john.toString); // function - found in Object.p

// 4. Continue until null
console.log(john.nonExistent); // undefined - not found any
```

3. Inheritance Chain:

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.eat = function() {
    return `${this.name} is eating`;
};

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {
    return 'Woof!';
};
```

```
const buddy = new Dog('Buddy', 'Golden Retriever');

// Chain: buddy -> Dog.prototype -> Animal.prototype -> Obj
console.log(buddy.breed); // "Golden Retriever" - instance
console.log(buddy.bark()); // "Woof!" - Dog.prototype
console.log(buddy.eat()); // "Buddy is eating" - Animal.pro
console.log(buddy.toString()); // "[object Object]" - Objec
```

4. Chain Visualization:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';

const john = new Person('John');

// Visualize the entire chain
let current = john;
let level = 0;

while (current !== null) {
    console.log(`Level ${level}:`, current.constructor.name
    console.log('Own properties:', Object.getOwnPropertyNam
    current = Object.getPrototypeOf(current);
    level++;
}

// Output:
// Level 0: Person
// Own properties: ['name']
// Level 1: Object
```

```
// Own properties: ['constructor', 'species', 'greet', ...]
// Level 2: null
```

5. Method Overriding:

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.makeSound = function() {
    return 'Some generic sound';
};

function Dog(name) {
    Animal.call(this, name);
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Override parent method
Dog.prototype.makeSound = function() {
    return 'Woof! Woof!';
};

const buddy = new Dog('Buddy');

// JavaScript finds the method in Dog.prototype first
console.log(buddy.makeSound()); // "Woof! Woof!" - from Dog

// Access parent method explicitly
console.log(Animal.prototype.makeSound.call(buddy)); // "So
```

6. Property Shadowing:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';

const john = new Person('John');

// Instance property shadows prototype property
john.species = 'Alien';

console.log(john.species); // "Alien" - instance property
console.log(john.__proto__.species); // "Homo sapiens" - pr

// Check if property is own or inherited
console.log(john.hasOwnProperty('species')); // true - own
console.log(john.hasOwnProperty('name')); // true - own pro
console.log(john.hasOwnProperty('toString')); // false - in
```

7. Chain Modification:

```
const obj = { a: 1 };
const proto = { b: 2 };
const grandProto = { c: 3 };

// Set up chain: obj -> proto -> grandProto -> Object.proto
Object.setPrototypeOf(obj, proto);
Object.setPrototypeOf(proto, grandProto);

console.log(obj.a); // 1 - own property
console.log(obj.b); // 2 - from proto
console.log(obj.c); // 3 - from grandProto
console.log(obj.toString); // function - from Object.protot
```

8. Built-in Object Chains:

```
const arr = [1, 2, 3];

// Array prototype chain
console.log(arr.__proto__ === Array.prototype); // true
console.log(Array.prototype.__proto__ === Object.prototype)
console.log(Object.prototype.__proto__ === null); // true

// String prototype chain
const str = 'hello';
console.log(str.__proto__ === String.prototype); // true
console.log(String.prototype.__proto__ === Object.prototype

// Function prototype chain
function fn() {}
console.log(fn.__proto__ === Function.prototype); // true
console.log(Function.prototype.__proto__ === Object.prototy
```

9. Chain Inspection Methods:

```
function Person(name) {
    this.name = name;
}

Person.prototype.species = 'Homo sapiens';

const john = new Person('John');

// Check if property exists in chain
console.log('name' in john); // true - own property
console.log('species' in john); // true - inherited
console.log('toString' in john); // true - inherited from O

// Get property descriptor
```

```
console.log(Object.getOwnPropertyDescriptor(john, 'name'));
// { value: 'John', writable: true, enumerable: true, confi

// List all properties in chain
function getAllProperties(obj) {
    const props = [];
    let current = obj;

    while (current !== null) {
        props.push(...Object.getOwnPropertyNames(current));
        current = Object.getPrototypeOf(current);
    }

    return [...new Set(props)]; // Remove duplicates
}

console.log(getAllProperties(john));
// ['name', 'constructor', 'species', 'greet', 'toString',
```

10. Performance Considerations:

```
// Deep prototype chains can impact performance
function createDeepChain(depth) {
    let current = {};

    for (let i = 0; i < depth; i++) {
        const newObj = { [`level${i}`]: i };
        Object.setPrototypeOf(newObj, current);
        current = newObj;
    }

    return current;
}

const deepObj = createDeepChain(1000);
```

```
// Accessing properties deep in the chain is slower
console.time('deep property access');
for (let i = 0; i < 10000; i++) {
    deepObj.level999; // Access property deep in chain
}
console.timeEnd('deep property access');

// Shallow chains are faster
const shallowObj = { a: 1, b: 2, c: 3 };
console.time('shallow property access');
for (let i = 0; i < 10000; i++) {
    shallowObj.a; // Access own property
}
console.timeEnd('shallow property access');
```

11. Common Chain Patterns:

Single Inheritance:

```
function Vehicle(make, model) {
    this.make = make;
    this.model = model;
}

Vehicle.prototype.start = function() {
    return `${this.make} ${this.model} is starting`;
};

function Car(make, model, doors) {
    Vehicle.call(this, make, model);
    this.doors = doors;
}

Car.prototype = Object.create(Vehicle.prototype);
Car.prototype.constructor = Car;
```

```
Car.prototype.honk = function() {
    return 'Beep! Beep!';
};
```

Multiple Inheritance (Mixins):

```
const Flyable = {
    fly() {
        return `${this.name} is flying`;
    }
};

const Swimmable = {
    swim() {
        return `${this.name} is swimming`;
    }
};

function Duck(name) {
    this.name = name;
}

// Mix in capabilities
Object.assign(Duck.prototype, Flyable, Swimmable);

const donald = new Duck('Donald');
console.log(donald.fly()); // "Donald is flying"
console.log(donald.swim()); // "Donald is swimming"
```

12. Chain Debugging:

```
function debugPrototypeChain(obj, name = 'Object') {
    console.log(`\n=== ${name} Prototype Chain ===`);
```

```
    let current = obj;
    let level = 0;

    while (current !== null) {
        console.log(`Level ${level}:`, current.constructor.
        console.log('Properties:', Object.getOwnPropertyNam

        if (level > 10) { // Prevent infinite loops
            console.log('... (chain too deep, stopping)');
            break;
        }

        current = Object.getPrototypeOf(current);
        level++;
    }

    console.log('=== End Chain ===\n');
}

const testObj = { test: 'value' };
debugPrototypeChain(testObj, 'Test Object');
```

Key Concepts:

1. **Delegation**: Objects delegate property access to their prototype
2. **Search Order**: Instance → Prototype → Prototype's prototype → ... → null
3. **Shadowing**: Instance properties override prototype properties
4. **Dynamic**: Chain is checked at runtime
5. **Mutable**: Chain can be modified (though not recommended)

Best Practices:

1. Keep prototype chains shallow for better performance
2. Use `hasOwnProperty()` to check for own properties

3. Avoid modifying built-in prototype chains
4. Use `Object.getPrototypeOf()` instead of `__proto__`
5. Prefer composition over deep inheritance
6. Be careful with circular references

Related Questions: Prototypes, Prototypal Inheritance, **proto** vs prototype

Back to Top

Asynchronous JavaScript

31. What are callbacks and what is callback hell?

Callbacks are functions that are passed as arguments to other functions and are executed at a later time, typically after an asynchronous operation completes. **Callback hell** refers to the deeply nested, hard-to-read code that results from chaining multiple asynchronous operations using callbacks.

Understanding Callbacks:

1. Basic Callback Pattern:

```
// Synchronous callback
function greet(name, callback) {
    const message = `Hello, ${name}!`;
    callback(message);
}

greet('John', function(message) {
    console.log(message); // "Hello, John!"
});

// Asynchronous callback
```

```
function fetchData(callback) {
    setTimeout(() => {
        const data = { id: 1, name: 'John' };
        callback(data);
    }, 1000);
}

fetchData(function(data) {
    console.log(data); // { id: 1, name: 'John' } (after 1
});
```

2. Common Callback Examples:

```
// Array methods with callbacks
const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(num) {
    console.log(num * 2);
});

const doubled = numbers.map(function(num) {
    return num * 2;
});

// DOM event callbacks
document.addEventListener('click', function(event) {
    console.log('Clicked:', event.target);
});

// Timer callbacks
setTimeout(function() {
    console.log('This runs after 2 seconds');
}, 2000);
```

3. Error-First Callback Pattern:

```
function readFile(filename, callback) {
    setTimeout(() => {
        if (filename === 'error.txt') {
            callback(new Error('File not found'), null);
        } else {
            callback(null, `Content of ${filename}`);
        }
    }, 1000);
}

// Usage with error handling
readFile('data.txt', function(error, data) {
    if (error) {
        console.error('Error:', error.message);
        return;
    }
    console.log('Data:', data);
});
```

4. Callback Hell Problem:

```
// Nested callbacks become hard to read and maintain
getUserData(function(user) {
    getProfileData(user.id, function(profile) {
        getSettingsData(profile.id, function(settings) {
            getNotificationsData(settings.id, function(noti
                getMessagesData(notifications.id, function(
                    // Deep nesting - hard to read and debu
                    console.log('All data loaded:', {
                        user, profile, settings, notificati
                    });
                });
            });
        });
```

```
    });
});
```

5. Solutions to Callback Hell:

Named Functions:

```
function handleUserData(user) {
    getProfileData(user.id, handleProfileData);
}

function handleProfileData(profile) {
    getSettingsData(profile.id, handleSettingsData);
}

function handleSettingsData(settings) {
    getNotificationsData(settings.id, handleMessagesData);
}

function handleMessagesData(messages) {
    console.log('All data loaded:', messages);
}

// Much cleaner!
getUserData(handleUserData);
```

Promise-based Solution:

```
getUserData()
    .then(user => getProfileData(user.id))
    .then(profile => getSettingsData(profile.id))
    .then(settings => getNotificationsData(settings.id))
    .then(notifications => getMessagesData(notifications.id
```

```
    .then(messages => console.log('All data loaded:', messa
    .catch(error => console.error('Error:', error));
```

Best Practices:

1. Use named functions instead of anonymous callbacks
2. Implement error-first callback pattern
3. Consider using Promises or async/await for complex flows
4. Use Promise.all() for parallel operations
5. Avoid deep nesting by breaking functions into smaller pieces

Related Questions: Promises, Async/Await

Back to Top

---

32. What are Promises and how do they work?

Promises are objects that represent the eventual completion or failure of an asynchronous operation. They provide a cleaner way to handle asynchronous code compared to callbacks.

Promise States:

- **Pending**: Initial state, neither fulfilled nor rejected
- **Fulfilled**: Operation completed successfully
- **Rejected**: Operation failed

1. Creating Promises:

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        const success = Math.random() > 0.5;
        if (success) {
```

```
            resolve('Operation successful!');
        } else {
            reject(new Error('Operation failed!'));
        }
    }, 1000);
});

promise
    .then(result => console.log(result))
    .catch(error => console.error(error));
```

2. Promise Methods:

```
const fetchData = () => {
    return new Promise((resolve) => {
        setTimeout(() => resolve({ id: 1, name: 'John' }),
    });
};

fetchData()
    .then(data => {
        console.log('Data received:', data);
        return data.id;
    })
    .then(id => console.log('ID:', id))
    .catch(error => console.error('Error:', error))
    .finally(() => console.log('Operation completed'));
```

3. Chaining Promises:

```
getUser(1)
    .then(user => {
        console.log('User:', user);
        return getProfile(user.id);
```

```
    })
    .then(profile => {
        console.log('Profile:', profile);
        return getSettings(profile.userId);
    })
    .then(settings => console.log('Settings:', settings))
    .catch(error => console.error('Error in chain:', error));
```

4. Promise.all() - Parallel Execution:

```
Promise.all([promise1, promise2, promise3])
    .then(results => console.log('All completed:', results))
    .catch(error => console.error('One failed:', error));
```

5. Promise.race() - First to Complete:

```
const timeoutPromise = new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Timeout')), 5000)
);

Promise.race([dataPromise, timeoutPromise])
    .then(data => console.log('Data received:', data))
    .catch(error => console.error('Error:', error.message));
```

6. Converting Callbacks to Promises:

```
function promisify(fn) {
    return function(...args) {
        return new Promise((resolve, reject) => {
            fn(...args, (error, result) => {
                if (error) reject(error);
                else resolve(result);
            });
```

```
        });
    };
}

const readFileAsync = promisify(readFile);
readFileAsync('data.txt')
    .then(data => console.log(data))
    .catch(error => console.error(error.message));
```

Best Practices:

1. Always handle errors with .catch() or try-catch
2. Use Promise.all() for parallel operations
3. Use Promise.race() for timeout patterns
4. Prefer async/await for complex flows
5. Convert callbacks to Promises for consistency

Related Questions: Callbacks, Async/Await

⬆️ Back to Top

---

33. What is async/await and how does it differ from Promises?

async/await is syntactic sugar built on top of Promises that makes asynchronous code look and behave more like synchronous code. It provides a cleaner, more readable way to work with Promises.

Key Differences:

1. Syntax Comparison:

```
// Promise-based approach
function fetchUserData(userId) {
    return fetch(`/api/users/${userId}`)
        .then(response => response.json())
        .then(user => {
            return fetch(`/api/profiles/${user.id}`)
                .then(response => response.json())
                .then(profile => ({ user, profile }));
        })
        .catch(error => {
            console.error('Error:', error);
            throw error;
        });
}

// async/await approach
async function fetchUserData(userId) {
    try {
        const userResponse = await fetch(`/api/users/${user
        const user = await userResponse.json();

        const profileResponse = await fetch(`/api/profiles/
        const profile = await profileResponse.json();

        return { user, profile };
    } catch (error) {
        console.error('Error:', error);
        throw error;
    }
}
```

2. Basic async/await Usage:

```
// async function always returns a Promise
async function getData() {
```

```
    return 'Hello World';
}

// Using await
async function processData() {
    const data = await getData();
    console.log(data); // "Hello World"
    return data.toUpperCase();
}

processData().then(result => console.log(result)); // "HELL
```

3. Error Handling:

```
// Promise error handling
riskyOperation()
    .then(result => console.log(result))
    .catch(error => console.error('Error:', error.message))

// async/await error handling
async function handleRiskyOperation() {
    try {
        const result = await riskyOperation();
        console.log(result);
    } catch (error) {
        console.error('Error:', error.message);
    }
}
```

4. Sequential vs Parallel Execution:

```
// Sequential execution (slower)
async function sequentialExample() {
    const user = await fetchUser();
```

```
    const profile = await fetchProfile(user.id);
    const settings = await fetchSettings(profile.id);
    return { user, profile, settings };
}

// Parallel execution (faster)
async function parallelExample() {
    const [user, profile, settings] = await Promise.all([
        fetchUser(),
        fetchProfile(1),
        fetchSettings(1)
    ]);
    return { user, profile, settings };
}
```

5. Common Patterns:

```
// Retry pattern
async function retryOperation(fn, maxAttempts = 3) {
    for (let attempt = 1; attempt <= maxAttempts; attempt++
        try {
            return await fn();
        } catch (error) {
            if (attempt === maxAttempts) throw error;
            await new Promise(resolve => setTimeout(resolve
        }
    }
}

// Timeout pattern
async function withTimeout(promise, timeoutMs) {
    const timeoutPromise = new Promise((_, reject) =>
        setTimeout(() => reject(new Error('Timeout')), time
    );
```

```
    return Promise.race([promise, timeoutPromise]);
}
```

Best Practices:

1. Use async/await for complex asynchronous flows
2. Always handle errors with try-catch
3. Use Promise.all() for parallel operations
4. Don't forget the await keyword
5. Use Promise.race() for timeout patterns

Related Questions: Promises, Event Loop

Back to Top

---

34. What is the Event Loop?

The **Event Loop** is the mechanism that allows JavaScript to handle asynchronous operations despite being single-threaded. It manages the execution of code, handling events, and processing callbacks.

How the Event Loop Works:

1. JavaScript Runtime Components:

```
// JavaScript runtime consists of:
// 1. Call Stack - where function calls are executed
// 2. Web APIs - browser APIs (setTimeout, DOM, etc.)
// 3. Task Queue (Macrotask Queue) - for callbacks
// 4. Microtask Queue - for Promises and other microtasks
// 5. Event Loop - coordinates everything

console.log('1. Start');
setTimeout(() => console.log('2. Timeout'), 0);
```

```
Promise.resolve().then(() => console.log('3. Promise'));
console.log('4. End');

// Output:
// 1. Start
// 4. End
// 3. Promise
// 2. Timeout
```

2. Microtasks vs Macrotasks:

```
console.log('1. Start');

// Macrotask
setTimeout(() => console.log('2. Timeout (Macrotask)'), 0);

// Microtask
Promise.resolve().then(() => console.log('3. Promise (Micro

// Microtask
queueMicrotask(() => console.log('4. queueMicrotask (Microt

console.log('5. End');

// Output:
// 1. Start
// 5. End
// 3. Promise (Microtask)
// 4. queueMicrotask (Microtask)
// 2. Timeout (Macrotask)
```

3. Event Loop Phases:

```
// 1. Execute all synchronous code
console.log('Sync code');

// 2. Execute all microtasks
Promise.resolve().then(() => console.log('Microtask 1'));
queueMicrotask(() => console.log('Microtask 2'));

// 3. Execute one macrotask
setTimeout(() => console.log('Macrotask 1'), 0);

// 4. Repeat: microtasks → one macrotask → microtasks → one
```

Key Concepts:

1. **Single-threaded**: JavaScript has one call stack
2. **Non-blocking**: Uses callbacks and promises for async operations
3. **Event-driven**: Responds to events and callbacks
4. **Microtasks have priority**: Execute before macrotasks
5. **Call stack must be empty**: Before processing queues

Best Practices:

1. Avoid blocking the call stack with long-running operations
2. Use setTimeout(0) to yield control back to the event loop
3. Use requestAnimationFrame for smooth animations
4. Be aware of microtask vs macrotask priority
5. Use Web Workers for CPU-intensive tasks

Related Questions: Microtasks vs Macrotasks, Promises

⬆️ Back to Top

---

35. What's the difference between microtasks and macrotasks?

Microtasks and **macrotasks** are two different queues in the JavaScript event loop that handle different types of asynchronous operations. Microtasks have higher priority and execute before macrotasks.

Key Differences:

Aspect	Microtasks	Macrotasks
Priority	Higher (execute first)	Lower (execute after microtasks)
Queue	Microtask Queue	Task Queue (Macrotask Queue)
Examples	Promises, queueMicrotask(), MutationObserver	setTimeout, setInterval, DOM events, I/O
Execution	All microtasks in queue	One macrotask at a time

1. Basic Example:

```
console.log('1. Start');

// Macrotask
setTimeout(() => console.log('2. setTimeout (Macrotask)'),

// Microtask
Promise.resolve().then(() => console.log('3. Promise (Micro

// Microtask
queueMicrotask(() => console.log('4. queueMicrotask (Microt
```

```
console.log('5. End');

// Output:
// 1. Start
// 5. End
// 3. Promise (Microtask)
// 4. queueMicrotask (Microtask)
// 2. setTimeout (Macrotask)
```

2. Microtask Examples:

```
// Promise.then()
Promise.resolve().then(() => console.log('Promise microtask

// queueMicrotask()
queueMicrotask(() => console.log('queueMicrotask'));

// async/await (creates microtasks)
async function asyncFunction() {
    await Promise.resolve();
    console.log('async/await microtask');
}
```

3. Macrotask Examples:

```
// setTimeout
setTimeout(() => console.log('setTimeout macrotask'), 0);

// DOM events
document.addEventListener('click', () => {
    console.log('Click event macrotask');
});
```

```
// I/O operations
fetch('/api/data').then(() => {
    console.log('Fetch response macrotask');
});
```

4. Execution Order:

```
console.log('1. Synchronous');

// Macrotasks
setTimeout(() => console.log('2. setTimeout 1'), 0);
setTimeout(() => console.log('3. setTimeout 2'), 0);

// Microtasks
Promise.resolve().then(() => console.log('4. Promise 1'));
queueMicrotask(() => console.log('5. queueMicrotask 1'));

console.log('6. Synchronous end');

// Output:
// 1. Synchronous
// 6. Synchronous end
// 4. Promise 1
// 5. queueMicrotask 1
// 2. setTimeout 1
// 3. setTimeout 2
```

Best Practices:

1. Use microtasks for immediate, high-priority operations
2. Use macrotasks for non-critical, background operations
3. Be careful not to create microtask storms
4. Use setTimeout(0) to yield control back to the event loop

5. Understand that microtasks can block macrotasks

Related Questions: Event Loop, Promises, Async/Await

⬆️ Back to Top

Event Handling

36. How does event handling work in JavaScript?

Event handling in JavaScript allows you to respond to user interactions and browser events. It involves registering event listeners that execute functions when specific events occur.

Basic Event Handling:

1. addEventListener Method (Recommended):

```
// Basic event listener
const button = document.getElementById('myButton');

button.addEventListener('click', function(event) {
    console.log('Button clicked!');
    console.log('Event object:', event);
});

// Arrow function syntax
button.addEventListener('click', (event) => {
    console.log('Button clicked with arrow function!');
});

// Named function
function handleClick(event) {
    console.log('Button clicked with named function!');
}
```

```
button.addEventListener('click', handleClick);
```

2. Event Object:

```
button.addEventListener('click', function(event) {
    console.log('Event type:', event.type); // 'click'
    console.log('Target element:', event.target); // The bu
    console.log('Current target:', event.currentTarget); //
    console.log('Event phase:', event.eventPhase); // 2 (bu
    console.log('Time stamp:', event.timeStamp); // When th
    console.log('Default prevented:', event.defaultPrevente
});
```

3. Different Event Types:

```
// Mouse events
button.addEventListener('click', handleClick);
button.addEventListener('dblclick', handleDoubleClick);
button.addEventListener('mousedown', handleMouseDown);
button.addEventListener('mouseup', handleMouseUp);
button.addEventListener('mouseover', handleMouseOver);
button.addEventListener('mouseout', handleMouseOut);

// Keyboard events
document.addEventListener('keydown', handleKeyDown);
document.addEventListener('keyup', handleKeyUp);
document.addEventListener('keypress', handleKeyPress);

// Form events
const form = document.getElementById('myForm');
form.addEventListener('submit', handleSubmit);
form.addEventListener('reset', handleReset);
```

```
const input = document.getElementById('myInput');
input.addEventListener('input', handleInput);
input.addEventListener('change', handleChange);
input.addEventListener('focus', handleFocus);
input.addEventListener('blur', handleBlur);

// Window events
window.addEventListener('load', handleLoad);
window.addEventListener('resize', handleResize);
window.addEventListener('scroll', handleScroll);
```

4. Event Listener Options:

```
// Third parameter options
button.addEventListener('click', handleClick, {
    capture: false,        // Use capturing phase (default:
    once: true,           // Remove listener after first ex
    passive: false,       // Never call preventDefault() (d
    signal: abortSignal   // AbortController signal to remo
});

// Once option - removes listener after first execution
button.addEventListener('click', handleClick, { once: true

// Capture phase
button.addEventListener('click', handleClick, { capture: tr

// Passive option for performance
window.addEventListener('scroll', handleScroll, { passive:
```

5. Removing Event Listeners:

```
// Remove with named function
button.addEventListener('click', handleClick);
```

```
button.removeEventListener('click', handleClick); // Must b

// Remove with AbortController
const controller = new AbortController();
button.addEventListener('click', handleClick, {
    signal: controller.signal
});
controller.abort(); // Removes the listener
```

6. Event Handler Functions:

```
function handleClick(event) {
    console.log('Button clicked!');
    console.log('Event details:', {
        type: event.type,
        target: event.target,
        clientX: event.clientX,
        clientY: event.clientY,
        button: event.button,
        ctrlKey: event.ctrlKey,
        shiftKey: event.shiftKey,
        altKey: event.altKey
    });
}

function handleKeyDown(event) {
    console.log('Key pressed:', event.key);
    console.log('Key code:', event.code);
    console.log('Modifier keys:', {
        ctrl: event.ctrlKey,
        shift: event.shiftKey,
        alt: event.altKey,
        meta: event.metaKey
    });
}
```

7. Event Delegation:

```
// Event delegation for dynamic content
document.addEventListener('click', function(event) {
    if (event.target.matches('.delete-button')) {
        handleDelete(event);
    } else if (event.target.matches('.edit-button')) {
        handleEdit(event);
    }
});
```

8. Custom Events:

```
// Create custom event
const customEvent = new CustomEvent('myCustomEvent', {
    detail: { message: 'Hello from custom event!' },
    bubbles: true,
    cancelable: true
});

// Listen for custom event
button.addEventListener('myCustomEvent', function(event) {
    console.log('Custom event received:', event.detail.mess
});

// Dispatch custom event
button.dispatchEvent(customEvent);
```

Best Practices:

1. Use `addEventListener` instead of `onclick` attributes
2. Use event delegation for dynamic content
3. Clean up event listeners to prevent memory leaks

4. Use passive listeners for scroll and touch events
5. Use named functions for easier debugging
6. Consider using AbortController for cleanup

Related Questions: Event Bubbling and Capturing, Event Delegation

Back to Top

---

37. What is event bubbling and capturing?

Event bubbling and **capturing** are two phases of event propagation in the DOM. They determine the order in which event handlers are executed when an event occurs on an element that has parent elements.

Event Propagation Phases:

1. Capturing Phase (Phase 1):

- Events travel from the root element down to the target element
- Parent elements receive the event before their children
- Use `capture: true` in addEventListener to listen during this phase

2. Target Phase (Phase 2):

- The event reaches the target element
- Event handlers on the target element are executed

3. Bubbling Phase (Phase 3):

- Events bubble up from the target element to the root
- Child elements' events bubble up to their parents
- This is the default behavior for most events

Basic Example:

```
<div id="grandparent">
    <div id="parent">
        <div id="child">Click me!</div>
    </div>
</div>
```

```
const grandparent = document.getElementById('grandparent');
const parent = document.getElementById('parent');
const child = document.getElementById('child');

// Bubbling phase (default)
child.addEventListener('click', () => console.log('Child cl
parent.addEventListener('click', () => console.log('Parent
grandparent.addEventListener('click', () => console.log('Gr

// Capturing phase
child.addEventListener('click', () => console.log('Child cl
parent.addEventListener('click', () => console.log('Parent
grandparent.addEventListener('click', () => console.log('Gr

// Clicking the child element outputs:
// Grandparent clicked (capturing)
// Parent clicked (capturing)
// Child clicked (capturing)
// Child clicked (bubbling)
// Parent clicked (bubbling)
// Grandparent clicked (bubbling)
```

2. Event Object Properties:

```
function handleEvent(event) {
    console.log('Event phase:', event.eventPhase);
    // 1 = CAPTURING_PHASE
```

```
    // 2 = AT_TARGET
    // 3 = BUBBLING_PHASE

    console.log('Target:', event.target); // Element that t
    console.log('Current target:', event.currentTarget); //
    console.log('Bubbles:', event.bubbles); // Whether even
    console.log('Cancelable:', event.cancelable); // Whethe
}
```

3. Stopping Event Propagation:

```
// stopPropagation() - stops event from propagating further
child.addEventListener('click', function(event) {
    console.log('Child clicked');
    event.stopPropagation(); // Stops bubbling/capturing
});

// stopImmediatePropagation() - stops event and other liste
child.addEventListener('click', function(event) {
    console.log('First listener');
    event.stopImmediatePropagation(); // Stops other listen
});

child.addEventListener('click', function(event) {
    console.log('Second listener'); // This won't execute
});
```

4. Events That Don't Bubble:

```
// These events don't bubble by default
const input = document.getElementById('myInput');

input.addEventListener('focus', handleFocus); // Doesn't bu
input.addEventListener('blur', handleBlur);   // Doesn't bu
```

```
input.addEventListener('load', handleLoad);   // Doesn't bu
input.addEventListener('unload', handleUnload); // Doesn't

// Check if event bubbles
function handleEvent(event) {
    console.log('Event bubbles:', event.bubbles);
}
```

5. Practical Examples:

Event Delegation with Bubbling:

```
// Instead of adding listeners to each button
document.addEventListener('click', function(event) {
    if (event.target.matches('.delete-btn')) {
        handleDelete(event);
    } else if (event.target.matches('.edit-btn')) {
        handleEdit(event);
    }
});

// Modal close on backdrop click
modal.addEventListener('click', function(event) {
    if (event.target === modal) {
        closeModal();
    }
});
```

Capturing for Early Intervention:

```
// Prevent all clicks on a specific container during captur
container.addEventListener('click', function(event) {
    event.stopPropagation();
```

```
    console.log('Click prevented on container');
}, { capture: true });
```

Best Practices:

1. Use event delegation for dynamic content
2. Understand which events bubble and which don't
3. Use `stopPropagation()` carefully - it can break other functionality
4. Use capturing phase for early intervention
5. Consider performance implications of event propagation

Related Questions: Event Delegation, Preventing Default Behavior

⬆️ Back to Top

---

38. What is event delegation?

Event delegation is a technique where you attach a single event listener to a parent element to handle events for multiple child elements. This is particularly useful for dynamic content and improves performance.

How Event Delegation Works:

1. Basic Event Delegation:

```
<ul id="todo-list">
    <li class="todo-item" data-id="1">Buy groceries</li>
    <li class="todo-item" data-id="2">Walk the dog</li>
    <li class="todo-item" data-id="3">Read a book</li>
</ul>
```

```
// Instead of adding listeners to each <li>
const todoList = document.getElementById('todo-list');

todoList.addEventListener('click', function(event) {
    // Check if the clicked element is a todo item
    if (event.target.classList.contains('todo-item')) {
        const todoId = event.target.dataset.id;
        console.log('Todo clicked:', todoId);
        toggleTodo(todoId);
    }
});

// Add new todo items dynamically
function addTodo(text) {
    const li = document.createElement('li');
    li.className = 'todo-item';
    li.dataset.id = Date.now();
    li.textContent = text;
    todoList.appendChild(li);
    // No need to add event listener - delegation handles i
}
```

2. Using matches() Method:

```
document.addEventListener('click', function(event) {
    // Handle delete buttons
    if (event.target.matches('.delete-btn')) {
        handleDelete(event);
    }

    // Handle edit buttons
    if (event.target.matches('.edit-btn')) {
        handleEdit(event);
    }
```

```
    // Handle any button with data-action
    if (event.target.matches('button[data-action]')) {
        const action = event.target.dataset.action;
        handleAction(action, event);
    }

    // Handle links with specific class
    if (event.target.matches('a.nav-link')) {
        event.preventDefault();
        handleNavigation(event);
    }
});
```

3. Event Delegation with Closest():

```
document.addEventListener('click', function(event) {
    // Find the closest element with a specific class
    const card = event.target.closest('.card');
    if (card) {
        handleCardClick(card, event);
    }

    // Find the closest form
    const form = event.target.closest('form');
    if (form) {
        handleFormInteraction(form, event);
    }

    // Find the closest item with data-id
    const item = event.target.closest('[data-id]');
    if (item) {
        const id = item.dataset.id;
        handleItemClick(id, event);
    }
});
```

4. Performance Benefits:

```
// ❌ Inefficient - one listener per item
function addListenersToItems() {
    const items = document.querySelectorAll('.item');
    items.forEach(item => {
        item.addEventListener('click', handleItemClick);
    });
}

// ✅ Efficient - one listener for all items
document.addEventListener('click', function(event) {
    if (event.target.matches('.item')) {
        handleItemClick(event);
    }
});
```

5. Event Delegation for Dynamic Content:

```
class TodoApp {
    constructor() {
        this.todos = [];
        this.setupEventDelegation();
    }

    setupEventDelegation() {
        document.addEventListener('click', (event) => {
            if (event.target.matches('.add-todo-btn')) {
                this.addTodo();
            } else if (event.target.matches('.delete-todo-b
                this.deleteTodo(event.target.dataset.id);
            } else if (event.target.matches('.toggle-todo-b
                this.toggleTodo(event.target.dataset.id);
            }
        });
```

```
        document.addEventListener('change', (event) => {
            if (event.target.matches('.todo-checkbox')) {
                this.toggleTodo(event.target.dataset.id);
            }
        });
    }

    addTodo() {
        const input = document.getElementById('todo-input')
        const text = input.value.trim();
        if (text) {
            const todo = { id: Date.now(), text, completed:
            this.todos.push(todo);
            this.renderTodo(todo);
            input.value = '';
        }
    }

    renderTodo(todo) {
        const container = document.getElementById('todo-con
        const todoElement = document.createElement('div');
        todoElement.className = 'todo-item';
        todoElement.dataset.id = todo.id;
        todoElement.innerHTML = `
            <input type="checkbox" class="todo-checkbox" da
            <span class="todo-text">${todo.text}</span>
            <button class="delete-todo-btn" data-id="${todo
        `;
        container.appendChild(todoElement);
    }
}
```

Best Practices:

1. Use event delegation for dynamic content

2. Use specific selectors to avoid conflicts
3. Use `closest()` for complex DOM traversal
   traversal - a step-by-step search of tree elements by connections between ancestor nodes and descendant nodes is called tree traversal. It's assumed that each node will only be visited once during the traversal. Basically, it's the same as traversing any collection using a loop or recursion.
4. Consider performance implications
5. Test with different DOM structures
6. Use `matches()` for flexible element selection

Related Questions: Event Bubbling and Capturing, Preventing Default Behavior

⬆️ Back to Top

---

39. How do you prevent default behavior and stop event propagation?

Preventing default behavior and **stopping event propagation** are two different concepts that control how events behave in the DOM.

Preventing Default Behavior:

1. preventDefault() Method:

```
// Prevent form submission
const form = document.getElementById('myForm');
form.addEventListener('submit', function(event) {
    event.preventDefault();
    console.log('Form submission prevented');
    // Handle form data manually
});
```

```
// Prevent link navigation
const link = document.getElementById('myLink');
link.addEventListener('click', function(event) {
    event.preventDefault();
    console.log('Link navigation prevented');
    // Handle navigation manually
});

// Prevent context menu
document.addEventListener('contextmenu', function(event) {
    event.preventDefault();
    console.log('Right-click menu prevented');
});
```

2. Common Use Cases for preventDefault():

```
// Form validation
form.addEventListener('submit', function(event) {
    const input = document.getElementById('email');
    if (!isValidEmail(input.value)) {
        event.preventDefault();
        showError('Please enter a valid email');
    }
});

// Custom keyboard shortcuts
document.addEventListener('keydown', function(event) {
    if (event.ctrlKey && event.key === 's') {
        event.preventDefault();
        saveDocument();
    }
});

// Prevent text selection
document.addEventListener('selectstart', function(event) {
```

```
    event.preventDefault();
});
```

Stopping Event Propagation:

3. stopPropagation() Method:

```
const parent = document.getElementById('parent');
const child = document.getElementById('child');

// Child stops propagation
child.addEventListener('click', function(event) {
    console.log('Child clicked');
    event.stopPropagation(); // Prevents parent from receiv
});

parent.addEventListener('click', function(event) {
    console.log('Parent clicked'); // This won't execute wh
});

// Only "Child clicked" will be logged
```

4. stopImmediatePropagation() Method:

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

// First listener
button.addEventListener('click', function(event) {
    console.log('First listener');
    event.stopImmediatePropagation(); // Stops other listen
});

// Second listener - won't execute
button.addEventListener('click', function(event) {
```

```
    console.log('Second listener'); // This won't execute
});
```

5. Practical Examples:

Modal Implementation:

```
const modal = document.getElementById('modal');
const closeBtn = document.getElementById('close-btn');

// Close modal when clicking close button
closeBtn.addEventListener('click', function(event) {
    event.preventDefault();
    event.stopPropagation();
    closeModal();
});

// Close modal when clicking backdrop
modal.addEventListener('click', function(event) {
    if (event.target === modal) {
        closeModal();
    }
});

// Prevent modal from closing when clicking content
const modalContent = document.getElementById('modal-content
modalContent.addEventListener('click', function(event) {
    event.stopPropagation();
});
```

Best Practices:

1. Use `preventDefault()` to stop default browser behavior
2. Use `stopPropagation()` to prevent event bubbling

3. Use `stopImmediatePropagation()` to stop other listeners on the same element
4. Be careful with `stopPropagation()` - it can break other functionality
5. Test event handling thoroughly
6. Consider accessibility when preventing default behavior

Related Questions: Event Bubbling and Capturing, Event Delegation

⬆️ Back to Top

---

40. What's the difference between `addEventListener` and `onclick`?

The main difference is that `addEventListener` allows multiple event listeners on the same element, while `onclick` only allows one. `addEventListener` also provides more control and options.

Key Differences:

Aspect	`addEventListener`	`onclick`
Multiple Listeners	✅ Supports multiple	❌ Only one
Event Options	✅ Capture, passive, once	❌ No options
Event Object	✅ Full event object	✅ Basic event object
Removal	✅ `removeEventListener`	✅ Set to `null`

Aspect	`addEventListener`	`onclick`
Performance	✅ Better for many listeners	⚠️ Good for single listener
Standards	✅ Modern standard	⚠️ Legacy but still used

1. Basic Usage:

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

// addEventListener - can add multiple listeners
button.addEventListener('click', function(event) {
    console.log('First click handler');
});

button.addEventListener('click', function(event) {
    console.log('Second click handler');
});

// onclick - only one handler
button.onclick = function(event) {
    console.log('onclick handler');
};

// Adding another onclick overwrites the previous one
button.onclick = function(event) {
    console.log('New onclick handler'); // Previous handler
};
```

2. Multiple Event Listeners:

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

```
// addEventListener - multiple listeners work
button.addEventListener('click', handleClick1);
button.addEventListener('click', handleClick2);
button.addEventListener('click', handleClick3);

// All three functions will execute when button is clicked

// onclick - only the last one works
button.onclick = handleClick1;
button.onclick = handleClick2; // This overwrites handleCli
button.onclick = handleClick3; // This overwrites handleCli

// Only handleClick3 will execute
```

3. Event Options:

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

// addEventListener with options
button.addEventListener('click', handleClick, {
    capture: true,        // Use capturing phase
    once: true,           // Remove after first execution
    passive: true         // Never call preventDefault()
});

// onclick has no options
button.onclick = handleClick; // No additional options
```

4. Event Removal:

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

// addEventListener - need exact function reference to remo
function handleClick(event) {
```

```
    console.log('Clicked');
}

button.addEventListener('click', handleClick);
button.removeEventListener('click', handleClick); // Remove

// onclick - set to null
button.onclick = handleClick;
button.onclick = null; // Removed
```

5. Event Delegation:

```
// addEventListener - perfect for delegation
document.addEventListener('click', function(event) {
    if (event.target.matches('.delete-btn')) {
        handleDelete(event);
    } else if (event.target.matches('.edit-btn')) {
        handleEdit(event);
    }
});

// onclick - not suitable for delegation
// Would need individual onclick handlers
```

Best Practices:

1. Use `addEventListener` for new code
2. Use `onclick` only for simple, single-handler cases
3. Prefer `addEventListener` for event delegation
4. Use `addEventListener` when you need event options
5. Consider performance implications
6. Use `addEventListener` for better maintainability

Related Questions: Event Handling, Event Delegation

Back to Top

DOM Manipulation

41. How do you select DOM elements?

DOM element selection is the process of finding and accessing HTML elements in the Document Object Model. JavaScript provides several methods to select elements based on different criteria.

1. getElementById():

```
// Select element by ID
const element = document.getElementById('myId');
console.log(element); // Returns the element or null

// Example
const header = document.getElementById('main-header');
header.style.color = 'blue';
```

2. getElementsByClassName():

```
// Select elements by class name
const elements = document.getElementsByClassName('myClass')
console.log(elements); // Returns HTMLCollection (live coll

// Access individual elements
const firstElement = elements[0];
const allElements = Array.from(elements); // Convert to arr

// Example
```

```
const buttons = document.getElementsByClassName('btn');
Array.from(buttons).forEach(button => {
    button.addEventListener('click', handleClick);
});
```

3. getElementsByTagName():

```
// Select elements by tag name
const elements = document.getElementsByTagName('div');
console.log(elements); // Returns HTMLCollection

// Example
const paragraphs = document.getElementsByTagName('p');
console.log(`Found ${paragraphs.length} paragraphs`);
```

4. querySelector():

```
// Select first matching element
const element = document.querySelector('#myId');
const element2 = document.querySelector('.myClass');
const element3 = document.querySelector('div.myClass');
const element4 = document.querySelector('[data-id="123"]');

// Complex selectors
const element5 = document.querySelector('div > p:first-chil
const element6 = document.querySelector('input[type="email"
const element7 = document.querySelector('.parent .child:not
```

5. querySelectorAll():

```
// Select all matching elements
const elements = document.querySelectorAll('.myClass');
console.log(elements); // Returns NodeList (static collecti
```

```
// Convert to array for easier manipulation
const elementsArray = Array.from(elements);

// Example
const buttons = document.querySelectorAll('button[data-acti
buttons.forEach(button => {
    button.addEventListener('click', handleAction);
});
```

6. Modern Selection Methods:

```
// Select by data attributes
const element = document.querySelector('[data-id="123"]');
const elements = document.querySelectorAll('[data-category=

// Select by attribute values
const emailInputs = document.querySelectorAll('input[type="
const requiredFields = document.querySelectorAll('[required

// Select by pseudo-classes
const firstChild = document.querySelector('p:first-child');
const lastChild = document.querySelector('li:last-child');
const evenRows = document.querySelectorAll('tr:nth-child(ev
```

7. Selection within Elements:

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

// Select within specific element
const child = container.querySelector('.child');
const children = container.querySelectorAll('.child');
```

```
// Select direct children only
const directChildren = container.querySelectorAll(':scope >
```

8. Performance Considerations:

```
// Cache frequently used elements
const header = document.getElementById('header');
const nav = document.querySelector('nav');
const main = document.querySelector('main');

// Use specific selectors for better performance
const specificElements = document.querySelectorAll('div.myC
const allDivs = document.querySelectorAll('div'); // Less e

// Use getElementById for single elements (fastest)
const element = document.getElementById('myId'); // Fastest

// Use querySelector for complex selectors
const complexElement = document.querySelector('div.containe
```

9. Common Selection Patterns:

```
// Select form elements
const form = document.querySelector('form');
const inputs = form.querySelectorAll('input');
const textInputs = form.querySelectorAll('input[type="text"
const checkboxes = form.querySelectorAll('input[type="check

// Select navigation elements
const navLinks = document.querySelectorAll('nav a');
const activeLink = document.querySelector('nav a.active');

// Select table elements
const table = document.querySelector('table');
```

```
const rows = table.querySelectorAll('tbody tr');
const cells = table.querySelectorAll('td');

// Select modal elements
const modal = document.querySelector('.modal');
const modalContent = modal.querySelector('.modal-content');
const closeButton = modal.querySelector('.close-btn');
```

10. Error Handling:

```
// Check if element exists
const element = document.getElementById('myId');
if (element) {
    element.style.display = 'none';
} else {
    console.error('Element with ID "myId" not found');
}

// Safe selection with fallback
function safeSelect(selector) {
    const element = document.querySelector(selector);
    if (!element) {
        console.warn(`Element with selector "${selector}" n
        return null;
    }
    return element;
}

const element = safeSelect('#myId');
if (element) {
    // Use element safely
    element.classList.add('active');
}
```

Best Practices:

1. Use `getElementById()` for single elements by ID (fastest)
2. Use `querySelector()` for complex selectors
3. Use `querySelectorAll()` for multiple elements
4. Cache frequently used elements
5. Check if elements exist before using them
6. Use specific selectors for better performance
7. Consider using `closest()` for finding parent elements

Related Questions: Creating and Modifying DOM Elements, Form Validation

⬆️ Back to Top

---

42. How do you create, modify, and remove DOM elements?

DOM manipulation involves creating, modifying, and removing HTML elements dynamically using JavaScript. This is essential for building interactive web applications.

Creating DOM Elements:

1. createElement():

```
// Create new element
const div = document.createElement('div');
const button = document.createElement('button');
const input = document.createElement('input');

// Set attributes
div.id = 'myDiv';
div.className = 'container';
button.textContent = 'Click me';
input.type = 'email';
input.placeholder = 'Enter email';
```

```
// Add to DOM
document.body.appendChild(div);
div.appendChild(button);
div.appendChild(input);
```

2. createTextNode():

```
// Create text node
const textNode = document.createTextNode('Hello World');
const paragraph = document.createElement('p');
paragraph.appendChild(textNode);

// Or use textContent (preferred)
const paragraph2 = document.createElement('p');
paragraph2.textContent = 'Hello World';
```

3. createDocumentFragment():

```
// Create document fragment for better performance
const fragment = document.createDocumentFragment();

// Add multiple elements to fragment
for (let i = 0; i < 1000; i++) {
    const li = document.createElement('li');
    li.textContent = `Item ${i}`;
    fragment.appendChild(li);
}

// Add fragment to DOM (single reflow)
const list = document.getElementById('myList');
list.appendChild(fragment);
```

Modifying DOM Elements:

4. Content Modification:

```
const element = document.getElementById('myElement');

// Text content
element.textContent = 'New text content';
element.innerText = 'New inner text'; // Respects styling
element.innerHTML = '<strong>Bold text</strong>';

// HTML content
element.innerHTML = `
    <h2>Title</h2>
    <p>Paragraph with <em>emphasis</em></p>
    <button>Click me</button>
`;
```

5. Attribute Modification:

```
const element = document.getElementById('myElement');

// Set attributes
element.setAttribute('id', 'newId');
element.setAttribute('class', 'newClass');
element.setAttribute('data-value', '123');

// Get attributes
const id = element.getAttribute('id');
const className = element.getAttribute('class');

// Remove attributes
element.removeAttribute('data-value');

// Check if attribute exists
```

```
if (element.hasAttribute('data-value')) {
    console.log('Attribute exists');
}

// Direct property access
element.id = 'newId';
element.className = 'newClass';
element.dataset.value = '123';
```

6. Class List Manipulation:

```
const element = document.getElementById('myElement');

// Add classes
element.classList.add('active', 'highlighted');
element.classList.add('new-class');

// Remove classes
element.classList.remove('active');
element.classList.remove('old-class', 'another-class');

// Toggle classes
element.classList.toggle('active');
element.classList.toggle('hidden', true); // Force add
element.classList.toggle('visible', false); // Force remove

// Check if class exists
if (element.classList.contains('active')) {
    console.log('Element has active class');
}

// Replace classes
element.classList.replace('old-class', 'new-class');
```

7. Style Modification:

```
const element = document.getElementById('myElement');

// Set individual styles
element.style.color = 'red';
element.style.backgroundColor = 'blue';
element.style.fontSize = '16px';
element.style.display = 'none';

// Set multiple styles
element.style.cssText = 'color: red; background-color: blue

// Get computed styles
const computedStyle = window.getComputedStyle(element);
const color = computedStyle.color;
const fontSize = computedStyle.fontSize;
```

Removing DOM Elements:

8. removeChild():

```
const parent = document.getElementById('parent');
const child = document.getElementById('child');

// Remove child element
parent.removeChild(child);

// Check if child exists before removing
if (parent.contains(child)) {
    parent.removeChild(child);
}
```

9. remove():

```
const element = document.getElementById('myElement');

// Remove element (modern method)
element.remove();

// Remove with error handling
if (element && element.parentNode) {
    element.remove();
} else {
    console.warn('Element not found or has no parent');
}
```

10. replaceChild():

```
const parent = document.getElementById('parent');
const oldChild = document.getElementById('oldChild');
const newChild = document.createElement('div');
newChild.textContent = 'New content';

// Replace child element
parent.replaceChild(newChild, oldChild);
```

Best Practices:

1. Use `createElement()` for new elements
2. Use `textContent` instead of `innerHTML` for plain text
3. Use `createDocumentFragment()` for multiple elements
4. Cache frequently accessed elements
5. Use `classList` for class manipulation
6. Remove event listeners to prevent memory leaks
7. Check if elements exist before manipulating them

Related Questions: DOM Element Selection, innerHTML vs textContent

⬆️ Back to Top

---

43. What's the difference between `innerHTML`, `textContent`, and `innerText`?

These three properties are used to get or set the content of HTML elements, but they behave differently in terms of what content they include and how they handle HTML.

Key Differences:

Property	HTML Content	Text Content	Performance	Security	Whitespace
`innerHTML`	✅ Includes HTML	✅ Includes text	⚠️ Slower	❌ XSS risk	✅ Preserved
`textContent`	❌ Strips HTML	✅ Includes text	✅ Faster	✅ Safe	✅ Preserved
`innerText`	❌ Strips HTML	✅ Includes text	⚠️ Slower	✅ Safe	❌ Collapsed

1. innerHTML:

```
const element = document.getElementById('myElement');

// Get HTML content
```

```
const htmlContent = element.innerHTML;
console.log(htmlContent); // "<p>Hello <strong>World</stron

// Set HTML content
element.innerHTML = '<p>New <em>content</em></p>';

// Complex HTML
element.innerHTML = `
    <div class="card">
        <h3>Title</h3>
        <p>Description with <a href="#">link</a></p>
        <button onclick="handleClick()">Click me</button>
    </div>
`;
```

2. textContent:

```
const element = document.getElementById('myElement');

// Get text content (strips HTML)
const textContent = element.textContent;
console.log(textContent); // "Hello World" (HTML tags remov

// Set text content (escapes HTML)
element.textContent = '<p>This will be displayed as text</p
// Result: "<p>This will be displayed as text</p>" (not ren

// Safe for user input
const userInput = '<script>alert("XSS")</script>';
element.textContent = userInput; // Safe - displays as text
```

3. innerText:

```
const element = document.getElementById('myElement');

// Get visible text content
const innerText = element.innerText;
console.log(innerText); // "Hello World" (respects CSS styl

// Set text content
element.innerText = 'New text content';

// Respects CSS display properties
element.innerHTML = '<p>Visible</p><p style="display:none">
console.log(element.innerText); // "Visible" (hidden text e
```

Best Practices:

1. Use `textContent` for user input to prevent XSS
2. Use `innerHTML` only when you need HTML markup
3. Use `innerText` when you need visible text only
4. Sanitize HTML content before using `innerHTML`
5. Consider performance implications
6. Be aware of whitespace differences

Related Questions: Creating and Modifying DOM Elements, Form Validation

Back to Top

---

44. How do you handle form validation in JavaScript?

Form validation ensures that user input meets specific requirements before submission. JavaScript provides various methods to validate forms both on the client side and in real-time.

1. Basic Form Validation:

```
const form = document.getElementById('myForm');

form.addEventListener('submit', function(event) {
    event.preventDefault(); // Prevent default form submiss

    const formData = new FormData(form);
    const data = Object.fromEntries(formData);

    // Validate form data
    if (validateForm(data)) {
        submitForm(data);
    }
});

function validateForm(data) {
    const errors = [];

    // Required field validation
    if (!data.name || data.name.trim() === '') {
        errors.push('Name is required');
    }

    // Email validation
    if (!data.email || !isValidEmail(data.email)) {
        errors.push('Valid email is required');
    }

    // Password validation
    if (!data.password || data.password.length < 8) {
        errors.push('Password must be at least 8 characters
    }

    // Display errors
    if (errors.length > 0) {
        displayErrors(errors);
        return false;
```

```
    }

    return true;
}
```

2. Real-time Validation:

```
// Validate on input
const nameInput = document.getElementById('name');
nameInput.addEventListener('input', function(event) {
    validateField(event.target);
});

const emailInput = document.getElementById('email');
emailInput.addEventListener('blur', function(event) {
    validateField(event.target);
});

function validateField(field) {
    const value = field.value.trim();
    const fieldName = field.name;
    let isValid = true;
    let errorMessage = '';

    switch (fieldName) {
        case 'name':
            if (value === '') {
                isValid = false;
                errorMessage = 'Name is required';
            } else if (value.length < 2) {
                isValid = false;
                errorMessage = 'Name must be at least 2 cha
            }
            break;

        case 'email':
```

```
            if (value === '') {
                isValid = false;
                errorMessage = 'Email is required';
            } else if (!isValidEmail(value)) {
                isValid = false;
                errorMessage = 'Please enter a valid email'
            }
            break;
    }

    showFieldValidation(field, isValid, errorMessage);
}
```

3. Validation Functions:

```
// Email validation
function isValidEmail(email) {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(email);
}

// Phone validation
function isValidPhone(phone) {
    const phoneRegex = /^[\+]?[1-9][\d]{0,15}$/;
    return phoneRegex.test(phone.replace(/\s/g, ''));
}

// Password strength validation
function validatePassword(password) {
    const errors = [];

    if (password.length < 8) {
        errors.push('At least 8 characters');
    }

    if (!/[A-Z]/.test(password)) {
```

```
        errors.push('At least one uppercase letter');
    }

    if (!/[a-z]/.test(password)) {
        errors.push('At least one lowercase letter');
    }

    if (!/\d/.test(password)) {
        errors.push('At least one number');
    }

    return errors;
}
```

4. Visual Feedback:

```
function showFieldValidation(field, isValid, errorMessage)
    const fieldContainer = field.closest('.field-container'
    const errorElement = fieldContainer.querySelector('.err

    // Remove existing validation classes
    field.classList.remove('valid', 'invalid');

    if (isValid) {
        field.classList.add('valid');
        if (errorElement) {
            errorElement.textContent = '';
            errorElement.style.display = 'none';
        }
    } else {
        field.classList.add('invalid');
        if (errorElement) {
            errorElement.textContent = errorMessage;
            errorElement.style.display = 'block';
        }
```

```
        }
    }
```

Best Practices:

1. Always validate on both client and server side
2. Provide immediate feedback to users
3. Use HTML5 validation attributes when possible
4. Sanitize and escape user input
5. Provide clear, specific error messages
6. Consider accessibility in validation feedback

Related Questions: Creating and Modifying DOM Elements, Document Ready vs Window Load

Back to Top

---

45. What is the difference between `document.ready` and `window.onload`?

The main difference is **when** they fire during the page loading process. `document.ready` fires when the DOM is fully constructed, while `window.onload` fires when all resources (images, stylesheets, etc.) have finished loading.

Key Differences:

Aspect	`document.ready`	`window.onload`
Timing	DOM ready (faster)	All resources loaded (slower)

Aspect	`document.ready`	`window.onload`
Resources	❌ Images, CSS may not be loaded	✅ All resources loaded
Performance	✅ Faster execution	⚠️ Slower execution
Use Case	DOM manipulation	Resource-dependent operations

1. document.ready (jQuery):

```
// jQuery document ready
$(document).ready(function() {
    console.log('DOM is ready');
    // Safe to manipulate DOM elements
    $('#myElement').addClass('loaded');
});

// Shorthand
$(function() {
    console.log('DOM is ready');
});
```

2. Native JavaScript Equivalent:

```
// DOMContentLoaded event (equivalent to document.ready)
document.addEventListener('DOMContentLoaded', function() {
    console.log('DOM is ready');
    // Safe to manipulate DOM elements
    document.getElementById('myElement').classList.add('loa
});
```

3. window.onload:

```
// window.onload
window.onload = function() {
    console.log('All resources loaded');
    // Safe to access images, stylesheets, etc.
    const img = document.getElementById('myImage');
    console.log('Image dimensions:', img.offsetWidth, img.o
};

// Or using addEventListener
window.addEventListener('load', function() {
    console.log('All resources loaded');
});
```

4. Timing Comparison:

```
console.log('Script start');

document.addEventListener('DOMContentLoaded', function() {
    console.log('DOM ready');
});

window.addEventListener('load', function() {
    console.log('Window loaded');
});

console.log('Script end');

// Output order:
// Script start
// Script end
// DOM ready
// Window loaded
```

5. When to Use Each:

Use `DOMContentLoaded` for:

- DOM manipulation
- Event listener attachment
- Form handling
- Element selection and modification

Use `window.onload` for:

- Image dimension calculations
- Layout-dependent operations
- Resource-heavy initializations
- Operations requiring all assets

Best Practices:

1. Use `DOMContentLoaded` for DOM manipulation
2. Use `window.onload` for resource-dependent operations
3. Check `document.readyState` for immediate execution
4. Use `addEventListener` instead of `onload` property
5. Consider performance implications

Related Questions: DOM Element Selection, Creating and Modifying DOM Elements

⬆️ Back to Top

ES6+ Features

46. What are template literals?

Template literals are string literals that allow embedded expressions and multi-line strings. They use backticks (`) instead of quotes and support string interpolation.

Basic Syntax:

```
// Template literal with backticks
const name = 'John';
const age = 30;
const message = `Hello, my name is ${name} and I am ${age}
console.log(message); // "Hello, my name is John and I am 3
```

1. String Interpolation:

```
// Variables and expressions
const price = 19.99;
const tax = 0.08;
const total = `Total: $${(price * (1 + tax)).toFixed(2)}`;
console.log(total); // "Total: $21.59"

// Function calls
const user = { firstName: 'Jane', lastName: 'Doe' };
const greeting = `Welcome, ${user.firstName.toUpperCase()}
console.log(greeting); // "Welcome, JANE Doe!"

// Complex expressions
const items = ['apple', 'banana', 'orange'];
const list = `Items: ${items.map(item => item.toUpperCase()
console.log(list); // "Items: APPLE, BANANA, ORANGE"
```

2. Multi-line Strings:

```
// Traditional approach (concatenation)
const oldWay = 'This is line 1\n' +
               'This is line 2\n' +
               'This is line 3';

// Template literals (cleaner)
const newWay = `This is line 1
This is line 2
This is line 3`;

console.log(newWay);
// This is line 1
// This is line 2
// This is line 3
```

3. HTML Templates:

```
function createUserCard(user) {
    return `
        <div class="user-card">
            <h3>${user.name}</h3>
            <p>Email: ${user.email}</p>
            <p>Age: ${user.age}</p>
            <p>Status: ${user.isActive ? 'Active' : 'Inacti
        </div>
    `;
}

const user = {
    name: 'John Doe',
    email: 'john@example.com',
    age: 30,
    isActive: true
};
```

```
const cardHTML = createUserCard(user);
console.log(cardHTML);
```

4. Tagged Template Literals:

```
// Tagged template function
function highlight(strings, ...values) {
    return strings.reduce((result, string, i) => {
        const value = values[i] ? `<mark>${values[i]}</mark
        return result + string + value;
    }, '');
}

const name = 'John';
const age = 30;
const highlighted = highlight`Hello ${name}, you are ${age}
console.log(highlighted); // "Hello <mark>John</mark>, you
```

Best Practices:

1. Use template literals for string interpolation
2. Use tagged templates for complex string processing
3. Be careful with user input in template literals (XSS prevention)
4. Use `String.raw` for raw string content
5. Consider performance for frequently called functions
6. Use template literals for multi-line strings
7. Leverage tagged templates for domain-specific languages

Related Questions: Default Parameters, Modules and Import/Export

⬆️ Back to Top

---

47. What are default parameters?

Default parameters allow you to set default values for function parameters when they are not provided or are `undefined`. This makes functions more flexible and reduces the need for parameter validation.

Basic Syntax:

```
// Basic default parameter
function greet(name = 'World') {
    return `Hello, ${name}!`;
}

console.log(greet()); // "Hello, World!"
console.log(greet('John')); // "Hello, John!"
```

1. Multiple Default Parameters:

```
function createUser(name = 'Anonymous', age = 0, isActive =
    return {
        name,
        age,
        isActive,
        createdAt: new Date()
    };
}

console.log(createUser()); // { name: 'Anonymous', age: 0,
console.log(createUser('John')); // { name: 'John', age: 0,
console.log(createUser('Jane', 25)); // { name: 'Jane', age
console.log(createUser('Bob', 30, false)); // { name: 'Bob'
```

2. Expressions as Default Values:

```
// Function calls as defaults
function getCurrentTime() {
    return new Date().toISOString();
}

function logMessage(message, timestamp = getCurrentTime())
    console.log(`[${timestamp}] ${message}`);
}

logMessage('User logged in'); // [2024-01-15T10:30:00.000Z]

// Complex expressions
function calculateArea(length = 10, width = length) {
    return length * width;
}

console.log(calculateArea()); // 100 (10 * 10)
console.log(calculateArea(5)); // 25 (5 * 5)
console.log(calculateArea(5, 8)); // 40 (5 * 8)
```

3. Object and Array Defaults:

```
// Object default
function createConfig(options = {}) {
    return {
        timeout: 5000,
        retries: 3,
        debug: false,
        ...options
    };
}

console.log(createConfig()); // { timeout: 5000, retries: 3
console.log(createConfig({ timeout: 10000, debug: true }));
```

```
// Array default
function processItems(items = []) {
    return items.map(item => item.toUpperCase());
}

console.log(processItems()); // []
console.log(processItems(['apple', 'banana'])); // ['APPLE'
```

4. Destructuring with Defaults:

```
// Object destructuring with defaults
function createUser({ name = 'Anonymous', age = 0, email =
    return { name, age, email };
}

console.log(createUser()); // { name: 'Anonymous', age: 0,
console.log(createUser({ name: 'John' })); // { name: 'John

// Array destructuring with defaults
function getFirstAndLast([first = 'first', , last = 'last']
    return { first, last };
}

console.log(getFirstAndLast()); // { first: 'first', last:
console.log(getFirstAndLast(['a', 'b', 'c'])); // { first:
```

Best Practices:

1. Use default parameters instead of checking for `undefined`
2. Place default parameters at the end of the parameter list
3. Use simple values for defaults to avoid side effects
4. Use destructuring with defaults for object parameters
5. Be careful with mutable default values (use functions)

6. Document default parameter behavior
7. Consider using default parameters in class constructors

Related Questions: Template Literals, Modules and Import/Export

Back to Top

---

48. What are modules and how do you use import/export?

Modules are reusable pieces of code that can be imported and exported between different files. They help organize code, avoid global namespace pollution, and enable better code splitting and tree shaking.

1. Export Types:

Named Exports:

```
// math.js
export const PI = 3.14159;
export const E = 2.71828;

export function add(a, b) {
    return a + b;
}

export function multiply(a, b) {
    return a * b;
}

// Alternative syntax
const subtract = (a, b) => a - b;
const divide = (a, b) => a / b;

export { subtract, divide };
```

Default Export:

```
// calculator.js
class Calculator {
    constructor() {
        this.result = 0;
    }

    add(value) {
        this.result += value;
        return this;
    }

    multiply(value) {
        this.result *= value;
        return this;
    }

    getResult() {
        return this.result;
    }
}

export default Calculator;
```

2. Import Types:

Named Imports:

```
// main.js
import { add, multiply, PI } from './math.js';

console.log(add(5, 3)); // 8
console.log(multiply(4, 6)); // 24
console.log(PI); // 3.14159
```

```
// Import with aliases
import { add as addition, multiply as multiplication } from

// Import all named exports
import * as math from './math.js';
console.log(math.add(2, 3)); // 5
console.log(math.PI); // 3.14159
```

Default Import:

```
// main.js
import Calculator from './calculator.js';

const calc = new Calculator();
calc.add(5).multiply(2);
console.log(calc.getResult()); // 10

// Import with alias
import Calculator as Calc from './calculator.js';
```

3. Mixed Imports:

```
// utils.js
export const VERSION = '1.0.0';
export function formatDate(date) {
    return date.toISOString();
}

export default class Utils {
    static log(message) {
        console.log(`[${new Date().toISOString()}] ${messag
    }
}
```

```
// main.js
import Utils, { VERSION, formatDate } from './utils.js';

Utils.log('Application started');
console.log('Version:', VERSION);
console.log('Date:', formatDate(new Date()));
```

4. Re-exporting:

```
// index.js (barrel export)
export { add, multiply, PI } from './math.js';
export { Calculator } from './calculator.js';
export { Utils, VERSION, formatDate } from './utils.js';

// main.js
import { add, Calculator, Utils } from './index.js';
```

5. Dynamic Imports:

```
// Dynamic import (returns a Promise)
async function loadModule() {
    const { add, multiply } = await import('./math.js');
    console.log(add(5, 3)); // 8
}

// Conditional loading
if (user.isAdmin) {
    const { AdminPanel } = await import('./admin.js');
    new AdminPanel();
}

// Lazy loading with error handling
async function loadFeature() {
```

```
    try {
        const module = await import('./feature.js');
        module.default();
    } catch (error) {
        console.error('Failed to load feature:', error);
    }
}
```

6. Module Patterns:

Utility Module:

```
// helpers.js
export const formatCurrency = (amount, currency = 'USD') =>
    return new Intl.NumberFormat('en-US', {
        style: 'currency',
        currency: currency
    }).format(amount);
};

export const debounce = (func, wait) => {
    let timeout;
    return function executedFunction(...args) {
        const later = () => {
            clearTimeout(timeout);
            func(...args);
        };
        clearTimeout(timeout);
        timeout = setTimeout(later, wait);
    };
};

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

```
            func.apply(this, args);
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    };
};
```

Service Module:

```
// api.js
class ApiService {
    constructor(baseURL) {
        this.baseURL = baseURL;
    }

    async get(endpoint) {
        const response = await fetch(`${this.baseURL}${endp
        return response.json();
    }

    async post(endpoint, data) {
        const response = await fetch(`${this.baseURL}${endp
            method: 'POST',
            headers: { 'Content-Type': 'application/json' }
            body: JSON.stringify(data)
        });
        return response.json();
    }
}

export default ApiService;
```

7. Module Configuration:

```
// config.js
const config = {
    apiUrl: process.env.API_URL || 'https://api.example.com
    timeout: 5000,
    retries: 3
};

export default config;

// constants.js
export const HTTP_STATUS = {
    OK: 200,
    CREATED: 201,
    BAD_REQUEST: 400,
    UNAUTHORIZED: 401,
    NOT_FOUND: 404,
    INTERNAL_SERVER_ERROR: 500
};

export const API_ENDPOINTS = {
    USERS: '/users',
    POSTS: '/posts',
    COMMENTS: '/comments'
};
```

8. Common Patterns:

```
// Event emitter module
// events.js
class EventEmitter {
    constructor() {
        this.events = {};
    }

    on(event, listener) {
```

```
        if (!this.events[event]) {
            this.events[event] = [];
        }
        this.events[event].push(listener);
    }

    emit(event, ...args) {
        if (this.events[event]) {
            this.events[event].forEach(listener => listener
        }
    }

    off(event, listener) {
        if (this.events[event]) {
            this.events[event] = this.events[event].filter(
        }
    }
}

export default EventEmitter;
```

Best Practices:

1. Use named exports for utilities and constants
2. Use default exports for main classes or functions
3. Keep modules focused and single-purpose
4. Use barrel exports for clean imports
5. Use dynamic imports for code splitting
6. Handle import errors gracefully
7. Use consistent naming conventions

Related Questions: Template Literals, Sets and Maps

⬆️ Back to Top

49. What are Sets and Maps?

Sets and **Maps** are new data structures introduced in ES6 that provide better alternatives to arrays and objects for specific use cases.

Sets:
A Set is a collection of unique values where each value can only occur once.

1. Basic Set Operations:

```
// Creating a Set
const mySet = new Set();
const mySet2 = new Set([1, 2, 3, 4, 5]);
const mySet3 = new Set('hello'); // {'h', 'e', 'l', 'o'}

// Adding values
mySet.add(1);
mySet.add(2);
mySet.add(2); // Duplicate, won't be added
mySet.add('hello');
mySet.add({ name: 'John' });

console.log(mySet); // Set { 1, 2, 'hello', { name: 'John'
console.log(mySet.size); // 4

// Checking if value exists
console.log(mySet.has(1)); // true
console.log(mySet.has(3)); // false

// Deleting values
mySet.delete(1);
console.log(mySet.has(1)); // false

// Clearing the Set
```

```
mySet.clear();
console.log(mySet.size); // 0
```

2. Set Iteration:

```
const mySet = new Set([1, 2, 3, 4, 5]);

// for...of loop
for (const value of mySet) {
    console.log(value);
}

// forEach method
mySet.forEach(value => {
    console.log(value);
});

// Converting to Array
const arrayFromSet = Array.from(mySet);
const arrayFromSet2 = [...mySet];

console.log(arrayFromSet); // [1, 2, 3, 4, 5]
```

3. Set Use Cases:

```
// Removing duplicates from array
const numbers = [1, 2, 2, 3, 3, 4, 5, 5];
const uniqueNumbers = [...new Set(numbers)];
console.log(uniqueNumbers); // [1, 2, 3, 4, 5]

// Tracking unique values
const visitedPages = new Set();
visitedPages.add('/home');
visitedPages.add('/about');
```

```
visitedPages.add('/contact');
visitedPages.add('/home'); // Duplicate

console.log(visitedPages.size); // 3

// Set operations
const setA = new Set([1, 2, 3, 4]);
const setB = new Set([3, 4, 5, 6]);

// Union
const union = new Set([...setA, ...setB]);
console.log(union); // Set { 1, 2, 3, 4, 5, 6 }

// Intersection
const intersection = new Set([...setA].filter(x => setB.has
console.log(intersection); // Set { 3, 4 }

// Difference
const difference = new Set([...setA].filter(x => !setB.has(
console.log(difference); // Set { 1, 2 }
```

Maps:

A Map is a collection of key-value pairs where keys can be of any type.

4. Basic Map Operations:

```
// Creating a Map
const myMap = new Map();
const myMap2 = new Map([
    ['key1', 'value1'],
    ['key2', 'value2'],
    [1, 'number key']
]);

// Setting values
myMap.set('name', 'John');
```

```
myMap.set('age', 30);
myMap.set(1, 'one');
myMap.set({ id: 1 }, 'object key');

console.log(myMap); // Map { 'name' => 'John', 'age' => 30,
console.log(myMap.size); // 4

// Getting values
console.log(myMap.get('name')); // 'John'
console.log(myMap.get('age')); // 30
console.log(myMap.get('nonexistent')); // undefined

// Checking if key exists
console.log(myMap.has('name')); // true
console.log(myMap.has('email')); // false

// Deleting entries
myMap.delete('age');
console.log(myMap.has('age')); // false

// Clearing the Map
myMap.clear();
console.log(myMap.size); // 0
```

5. Map Iteration:

```
const myMap = new Map([
    ['name', 'John'],
    ['age', 30],
    ['city', 'New York']
]);

// for...of loop
for (const [key, value] of myMap) {
    console.log(`${key}: ${value}`);
}
```

```
// forEach method
myMap.forEach((value, key) => {
    console.log(`${key}: ${value}`);
});

// Getting keys and values
console.log([...myMap.keys()]); // ['name', 'age', 'city']
console.log([...myMap.values()]); // ['John', 30, 'New York
console.log([...myMap.entries()]); // [['name', 'John'], ['
```

6. Map Use Cases:

```
// Caching function results
const cache = new Map();

function expensiveOperation(n) {
    if (cache.has(n)) {
        console.log('Cache hit!');
        return cache.get(n);
    }

    console.log('Computing...');
    const result = n * n * n; // Expensive operation
    cache.set(n, result);
    return result;
}

console.log(expensiveOperation(5)); // Computing... 125
console.log(expensiveOperation(5)); // Cache hit! 125

// Counting occurrences
const text = 'hello world';
const charCount = new Map();

for (const char of text) {
```

```
    if (char !== ' ') {
        charCount.set(char, (charCount.get(char) || 0) + 1)
    }
}

console.log(charCount); // Map { 'h' => 1, 'e' => 1, 'l' =>

// Object metadata
const userMetadata = new Map();
const user1 = { id: 1, name: 'John' };
const user2 = { id: 2, name: 'Jane' };

userMetadata.set(user1, { lastLogin: new Date(), loginCount
userMetadata.set(user2, { lastLogin: new Date(), loginCount

console.log(userMetadata.get(user1)); // { lastLogin: Date,
```

7. WeakSet and WeakMap:

```
// WeakSet - holds only objects, allows garbage collection
const weakSet = new WeakSet();
const obj1 = { name: 'John' };
const obj2 = { name: 'Jane' };

weakSet.add(obj1);
weakSet.add(obj2);

console.log(weakSet.has(obj1)); // true

// WeakMap - keys must be objects, allows garbage collectio
const weakMap = new WeakMap();
const key1 = { id: 1 };
const key2 = { id: 2 };

weakMap.set(key1, 'value1');
weakMap.set(key2, 'value2');
```

```
console.log(weakMap.get(key1)); // 'value1'
```

8. Performance Comparison:

```
// Set vs Array for unique values
const numbers = Array.from({ length: 10000 }, (_, i) => i);
const duplicates = [...numbers, ...numbers];

// Using Set (faster)
console.time('Set');
const uniqueSet = new Set(duplicates);
const uniqueArray = [...uniqueSet];
console.timeEnd('Set');

// Using Array methods (slower)
console.time('Array');
const uniqueArray2 = duplicates.filter((item, index) =>
    duplicates.indexOf(item) === index
);
console.timeEnd('Array');

// Map vs Object for key-value pairs
const map = new Map();
const obj = {};

// Map performance
console.time('Map');
for (let i = 0; i < 10000; i++) {
    map.set(i, i * 2);
}
console.timeEnd('Map');

// Object performance
console.time('Object');
for (let i = 0; i < 10000; i++) {
```

```
    obj[i] = i * 2;
}
console.timeEnd('Object');
```

Best Practices:

1. Use Set for unique value collections
2. Use Map for key-value pairs with non-string keys
3. Use WeakSet/WeakMap for object references that can be garbage collected
4. Use Set for removing duplicates from arrays
5. Use Map for caching and metadata storage
6. Consider performance implications for large datasets
7. Use appropriate iteration methods for your use case

Related Questions: Modules and Import/Export, Symbols

Back to Top

---

50. What are Symbols?

Symbols are a primitive data type introduced in ES6 that represents a unique identifier. They are immutable and can be used as object property keys.

Basic Syntax:

```
// Creating symbols
const sym1 = Symbol();
const sym2 = Symbol('description');
const sym3 = Symbol('description');

console.log(sym1); // Symbol()
```

```
console.log(sym2); // Symbol(description)
console.log(sym2 === sym3); // false (each symbol is unique
```

1. Symbol Properties:

```
// Using symbols as object keys
const id = Symbol('id');
const name = Symbol('name');

const user = {
    [id]: 123,
    [name]: 'John Doe',
    age: 30
};

console.log(user[id]); // 123
console.log(user[name]); // 'John Doe'
console.log(user.age); // 30

// Symbols don't appear in for...in loops
for (const key in user) {
    console.log(key); // Only 'age'
}

// Symbols don't appear in Object.keys()
console.log(Object.keys(user)); // ['age']

// But they appear in Object.getOwnPropertySymbols()
console.log(Object.getOwnPropertySymbols(user)); // [Symbol
```

2. Symbol.for() and Symbol.keyFor():

```
// Symbol.for() - creates or retrieves a symbol from global
const sym1 = Symbol.for('key');
```

```
const sym2 = Symbol.for('key');

console.log(sym1 === sym2); // true (same symbol from regis

// Symbol.keyFor() - gets the key for a symbol from registr
console.log(Symbol.keyFor(sym1)); // 'key'

// Regular symbols are not in registry
const sym3 = Symbol('key');
console.log(Symbol.keyFor(sym3)); // undefined
```

3. Well-known Symbols:

```
// Symbol.iterator - makes object iterable
const iterable = {
    [Symbol.iterator]: function* () {
        yield 1;
        yield 2;
        yield 3;
    }
};

for (const value of iterable) {
    console.log(value); // 1, 2, 3
}

// Symbol.toPrimitive - custom type conversion
const obj = {
    [Symbol.toPrimitive](hint) {
        if (hint === 'number') return 42;
        if (hint === 'string') return 'hello';
        return 'default';
    }
};

console.log(+obj); // 42
```

```
console.log(String(obj)); // 'hello'
console.log(obj + ''); // 'default'

// Symbol.toStringTag - custom toString behavior
const customObj = {
    [Symbol.toStringTag]: 'CustomObject'
};

console.log(Object.prototype.toString.call(customObj)); //
```

4. Symbol Use Cases:

```
// Private properties
const _private = Symbol('private');

class MyClass {
    constructor() {
        this[_private] = 'secret';
        this.public = 'visible';
    }

    getPrivate() {
        return this[_private];
    }
}

const instance = new MyClass();
console.log(instance.public); // 'visible'
console.log(instance[_private]); // 'secret'
console.log(instance.getPrivate()); // 'secret'

// Metadata
const metadata = Symbol('metadata');

function addMetadata(obj, data) {
    obj[metadata] = data;
```

```
}

function getMetadata(obj) {
    return obj[metadata];
}

const user = { name: 'John' };
addMetadata(user, { created: new Date(), version: 1 });
console.log(getMetadata(user)); // { created: Date, version
```

5. Symbol Constants:

```
// Using symbols as constants
const LOG_LEVELS = {
    DEBUG: Symbol('debug'),
    INFO: Symbol('info'),
    WARN: Symbol('warn'),
    ERROR: Symbol('error')
};

function log(level, message) {
    switch (level) {
        case LOG_LEVELS.DEBUG:
            console.debug(message);
            break;
        case LOG_LEVELS.INFO:
            console.info(message);
            break;
        case LOG_LEVELS.WARN:
            console.warn(message);
            break;
        case LOG_LEVELS.ERROR:
            console.error(message);
            break;
    }
}
```

```
log(LOG_LEVELS.INFO, 'User logged in');
log(LOG_LEVELS.ERROR, 'Database connection failed');
```

6. Symbol in Classes:

```
// Private methods using symbols
const _calculate = Symbol('calculate');

class Calculator {
    constructor() {
        this.result = 0;
    }

    add(value) {
        this.result = this[_calculate](this.result, value,
        return this;
    }

    multiply(value) {
        this.result = this[_calculate](this.result, value,
        return this;
    }

    [_calculate](a, b, operation) {
        switch (operation) {
            case '+': return a + b;
            case '*': return a * b;
            default: return a;
        }
    }

    getResult() {
        return this.result;
    }
}
```

```
const calc = new Calculator();
calc.add(5).multiply(2);
console.log(calc.getResult()); // 10
```

7. Symbol with Mixins:

```
// Mixin pattern using symbols
const mixin = Symbol('mixin');

function withLogging(target) {
    target[mixin] = {
        log(message) {
            console.log(`[${this.constructor.name}] ${messa
        }
    };

    return target;
}

class User {
    constructor(name) {
        this.name = name;
    }
}

// Apply mixin
withLogging(User);

// Use mixin
const user = new User('John');
user[mixin].log('User created'); // [User] User created
```

8. Symbol Performance:

```
// Symbol vs string keys performance
const symbolKey = Symbol('key');
const stringKey = 'key';

const obj = {};
obj[symbolKey] = 'value';
obj[stringKey] = 'value';

// Symbol access is slightly faster
console.time('Symbol access');
for (let i = 0; i < 1000000; i++) {
    const value = obj[symbolKey];
}
console.timeEnd('Symbol access');

console.time('String access');
for (let i = 0; i < 1000000; i++) {
    const value = obj[stringKey];
}
console.timeEnd('String access');
```

Best Practices:

1. Use symbols for private properties
2. Use symbols for metadata storage
3. Use symbols for constants to avoid naming conflicts
4. Use Symbol.for() for global symbols
5. Use well-known symbols for custom behavior
6. Consider performance implications
7. Use symbols sparingly - they can make debugging harder

Related Questions: Sets and Maps, Generators and Iterators

⬆️ Back to Top

---

51. What are generators and iterators?

Generators are functions that can be paused and resumed, allowing you to control the execution flow. **Iterators** are objects that provide a way to access elements sequentially.

1. Generator Functions:

```
// Generator function syntax
function* generatorFunction() {
    yield 1;
    yield 2;
    yield 3;
}

const generator = generatorFunction();
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: false }
console.log(generator.next()); // { value: undefined, done:
```

2. Generator with Parameters:

```
function* counter(start = 0, step = 1) {
    let current = start;
    while (true) {
        const reset = yield current;
        if (reset) {
            current = start;
        } else {
            current += step;
        }
```

```
    }
}

const counterGen = counter(10, 5);
console.log(counterGen.next()); // { value: 10, done: false
console.log(counterGen.next()); // { value: 15, done: false
console.log(counterGen.next()); // { value: 20, done: false
console.log(counterGen.next(true)); // { value: 10, done: f
```

3. Generator with Return:

```
function* generatorWithReturn() {
    yield 1;
    yield 2;
    return 'finished';
    yield 3; // This won't execute
}

const gen = generatorWithReturn();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 'finished', done: true
console.log(gen.next()); // { value: undefined, done: true
```

4. Generator with Error Handling:

```
function* errorGenerator() {
    try {
        yield 1;
        yield 2;
        throw new Error('Something went wrong');
        yield 3;
    } catch (error) {
        yield `Error caught: ${error.message}`;
```

```
    }
}

const gen = errorGenerator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 'Error caught: Somethi
console.log(gen.next()); // { value: undefined, done: true
```

5. Generator Delegation:

```
function* generator1() {
    yield 1;
    yield 2;
}

function* generator2() {
    yield 3;
    yield 4;
}

function* combinedGenerator() {
    yield* generator1();
    yield* generator2();
    yield 5;
}

const gen = combinedGenerator();
console.log([...gen]); // [1, 2, 3, 4, 5]
```

6. Custom Iterator:

```
// Custom iterable object
const iterableObject = {
```

```
    data: [1, 2, 3, 4, 5],
    [Symbol.iterator]() {
        let index = 0;
        const data = this.data;

        return {
            next() {
                if (index < data.length) {
                    return { value: data[index++], done: fa
                } else {
                    return { done: true };
                }
            }
        };
    }
};

// Using the custom iterator
for (const value of iterableObject) {
    console.log(value); // 1, 2, 3, 4, 5
}

// Or using spread operator
console.log([...iterableObject]); // [1, 2, 3, 4, 5]
```

7. Generator Use Cases:

```
// Infinite sequence
function* fibonacci() {
    let a = 0, b = 1;
    while (true) {
        yield a;
        [a, b] = [b, a + b];
    }
}
```

```
const fib = fibonacci();
console.log(fib.next().value); // 0
console.log(fib.next().value); // 1
console.log(fib.next().value); // 1
console.log(fib.next().value); // 2
console.log(fib.next().value); // 3

// Lazy evaluation
function* range(start, end) {
    for (let i = start; i < end; i++) {
        yield i;
    }
}

const numbers = range(1, 1000000);
console.log(numbers.next().value); // 1 (only generates whe

// Async generator
async function* asyncGenerator() {
    yield await fetch('/api/data1');
    yield await fetch('/api/data2');
    yield await fetch('/api/data3');
}

// Using async generator
async function processData() {
    for await (const response of asyncGenerator()) {
        const data = await response.json();
        console.log(data);
    }
}
```

8. Generator with State:

```
function* stateMachine() {
    let state = 'idle';
```

```
    while (true) {
        const action = yield state;

        switch (state) {
            case 'idle':
                if (action === 'start') state = 'running';
                break;
            case 'running':
                if (action === 'pause') state = 'paused';
                else if (action === 'stop') state = 'stoppe
                break;
            case 'paused':
                if (action === 'resume') state = 'running';
                else if (action === 'stop') state = 'stoppe
                break;
            case 'stopped':
                if (action === 'reset') state = 'idle';
                break;
        }
    }
}

const machine = stateMachine();
console.log(machine.next()); // { value: 'idle', done: fals
console.log(machine.next('start')); // { value: 'running',
console.log(machine.next('pause')); // { value: 'paused', d
console.log(machine.next('resume')); // { value: 'running',
console.log(machine.next('stop')); // { value: 'stopped', d
```

Best Practices:

1. Use generators for lazy evaluation
2. Use generators for infinite sequences
3. Use generators for state machines

4. Use generators for async operations
5. Use custom iterators for complex data structures
6. Consider memory usage with large datasets
7. Use generators for data transformation pipelines

Related Questions: Symbols, for...of vs for...in

⬆️ Back to Top

---

52. What is the `for...of` vs `for...in` loop?

The main difference is that `for...of` iterates over values, while `for...in` iterates over property names (keys).

Key Differences:

Aspect	`for...of`	`for...in`
Iterates over	Values	Property names (keys)
Works with	Iterables (arrays, strings, maps, sets)	Objects (and arrays)
Order	Guaranteed order	Not guaranteed order
Prototype	Only own properties	Includes inherited properties
Type	ES6+	ES5+

1. Basic Usage:

```
const arr = ['a', 'b', 'c'];

// for...of - iterates over values
for (const value of arr) {
    console.log(value); // 'a', 'b', 'c'
}

// for...in - iterates over indices
for (const index in arr) {
    console.log(index); // '0', '1', '2'
    console.log(arr[index]); // 'a', 'b', 'c'
}
```

2. Object Iteration:

```
const obj = { a: 1, b: 2, c: 3 };

// for...of - doesn't work with plain objects
// for (const value of obj) { // TypeError: obj is not iter
//     console.log(value);
// }

// for...in - works with objects
for (const key in obj) {
    console.log(key); // 'a', 'b', 'c'
    console.log(obj[key]); // 1, 2, 3
}
```

3. Array Iteration:

```
const arr = [10, 20, 30];

// for...of - values
```

```
for (const value of arr) {
    console.log(value); // 10, 20, 30
}

// for...in - indices (as strings)
for (const index in arr) {
    console.log(index); // '0', '1', '2'
    console.log(typeof index); // 'string'
    console.log(arr[index]); // 10, 20, 30
}
```

4. String Iteration:

```
const str = 'hello';

// for...of - characters
for (const char of str) {
    console.log(char); // 'h', 'e', 'l', 'l', 'o'
}

// for...in - indices
for (const index in str) {
    console.log(index); // '0', '1', '2', '3', '4'
    console.log(str[index]); // 'h', 'e', 'l', 'l', 'o'
}
```

5. Map and Set Iteration:

```
const map = new Map([['a', 1], ['b', 2], ['c', 3]]);
const set = new Set([1, 2, 3, 4, 5]);

// for...of with Map
for (const [key, value] of map) {
    console.log(key, value); // 'a' 1, 'b' 2, 'c' 3
```

```
}

// for...of with Set
for (const value of set) {
    console.log(value); // 1, 2, 3, 4, 5
}

// for...in doesn't work with Map/Set
// for (const key in map) { // Doesn't iterate
//     console.log(key);
// }
```

6. Prototype Chain:

```
// Object with inherited properties
const parent = { inherited: 'parent' };
const child = Object.create(parent);
child.own = 'child';

// for...in - includes inherited properties
for (const key in child) {
    console.log(key); // 'inherited', 'own'
}

// for...in with hasOwnProperty check
for (const key in child) {
    if (child.hasOwnProperty(key)) {
        console.log(key); // 'own'
    }
}

// for...of - only own properties (when iterable)
const arr = [1, 2, 3];
Array.prototype.customMethod = function() {};

for (const value of arr) {
```

```
    console.log(value); // 1, 2, 3 (no customMethod)
}

for (const index in arr) {
    console.log(index); // '0', '1', '2', 'customMethod'
}
```

7. Performance Comparison:

```
const largeArray = Array.from({ length: 1000000 }, (_, i) =

// for...of performance
console.time('for...of');
for (const value of largeArray) {
    // Do something
}
console.timeEnd('for...of');

// for...in performance
console.time('for...in');
for (const index in largeArray) {
    const value = largeArray[index];
    // Do something
}
console.timeEnd('for...in');

// Traditional for loop
console.time('for loop');
for (let i = 0; i < largeArray.length; i++) {
    const value = largeArray[i];
    // Do something
}
console.timeEnd('for loop');
```

8. Common Patterns:

```
// Iterating over array with index
const arr = ['a', 'b', 'c'];

// Using for...of with entries()
for (const [index, value] of arr.entries()) {
    console.log(index, value); // 0 'a', 1 'b', 2 'c'
}

// Using for...in with arrays (not recommended)
for (const index in arr) {
    console.log(parseInt(index), arr[index]); // 0 'a', 1 '
}

// Object iteration with for...of
const obj = { a: 1, b: 2, c: 3 };

// Convert to entries and iterate
for (const [key, value] of Object.entries(obj)) {
    console.log(key, value); // 'a' 1, 'b' 2, 'c' 3
}

// Iterate over keys only
for (const key of Object.keys(obj)) {
    console.log(key); // 'a', 'b', 'c'
}

// Iterate over values only
for (const value of Object.values(obj)) {
    console.log(value); // 1, 2, 3
}
```

9. When to Use Each:

```
// Use for...of for:
// - Array values
```

```
// - String characters
// - Map entries
// - Set values
// - Any iterable object

const iterables = [
    [1, 2, 3],           // Array
    'hello',             // String
    new Map([['a', 1]]), // Map
    new Set([1, 2, 3])   // Set
];

for (const iterable of iterables) {
    for (const value of iterable) {
        console.log(value);
    }
}

// Use for...in for:
// - Object properties
// - When you need the key/index
// - When checking for own properties

const obj = { a: 1, b: 2, c: 3 };
for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
        console.log(key, obj[key]);
    }
}
```

Best Practices:

1. Use `for...of` for iterating over values
2. Use `for...in` for iterating over object properties
3. Always check `hasOwnProperty` with `for...in`

4. Use `for...of` with `entries()` for index and value
5. Prefer `for...of` for arrays over `for...in`
6. Use `Object.entries()` with `for...of` for objects
7. Consider performance implications for large datasets

Related Questions: Generators and Iterators, Error Handling

⬆️ Back to Top

Error Handling

53. How do you handle errors in JavaScript?

Error handling in JavaScript involves catching and managing errors that occur during program execution. JavaScript provides several mechanisms for error handling, with `try...catch` being the most common.

1. Basic try...catch:

```
try {
    // Code that might throw an error
    const result = riskyOperation();
    console.log('Success:', result);
} catch (error) {
    // Handle the error
    console.error('Error occurred:', error.message);
} finally {
    // Code that always runs
    console.log('Cleanup completed');
}
```

2. Throwing Errors:

```
function divide(a, b) {
    if (b === 0) {
        throw new Error('Division by zero is not allowed');
    }
    return a / b;
}

try {
    const result = divide(10, 0);
} catch (error) {
    console.error('Error:', error.message); // "Division by
}
```

3. Error Types:

```
// Built-in error types
try {
    throw new Error('Generic error');
} catch (error) {
    console.log(error instanceof Error); // true
}

try {
    throw new TypeError('Type error');
} catch (error) {
    console.log(error instanceof TypeError); // true
}

try {
    throw new ReferenceError('Reference error');
} catch (error) {
    console.log(error instanceof ReferenceError); // true
}

try {
```

```
    throw new SyntaxError('Syntax error');
} catch (error) {
    console.log(error instanceof SyntaxError); // true
}
```

4. Custom Error Classes:

```
class ValidationError extends Error {
    constructor(message, field) {
        super(message);
        this.name = 'ValidationError';
        this.field = field;
    }
}

class NetworkError extends Error {
    constructor(message, statusCode) {
        super(message);
        this.name = 'NetworkError';
        this.statusCode = statusCode;
    }
}

// Using custom errors
function validateUser(user) {
    if (!user.name) {
        throw new ValidationError('Name is required', 'name
    }
    if (!user.email) {
        throw new ValidationError('Email is required', 'ema
    }
}

try {
    validateUser({});
} catch (error) {
```

```
    if (error instanceof ValidationError) {
        console.log(`Validation failed for ${error.field}:
    }
}
```

5. Async Error Handling:

```
// Promise-based error handling
async function fetchData() {
    try {
        const response = await fetch('/api/data');
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response
        }
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Fetch error:', error.message);
        throw error; // Re-throw if needed
    }
}

// Using fetchData
fetchData()
    .then(data => console.log('Data:', data))
    .catch(error => console.error('Error:', error.message))
```

Best Practices:

1. Always handle errors appropriately
2. Use specific error types for different scenarios
3. Provide meaningful error messages
4. Log errors for debugging and monitoring

5. Implement retry mechanisms for transient errors
6. Use fallback strategies when possible
7. Don't ignore errors silently
8. Test error handling paths
9. Use global error handlers for unexpected errors
10. Consider user experience when handling errors

Related Questions: Throwing and Catching Errors, Error Types

Back to Top

---

54. What is the difference between throwing and catching errors?

Throwing errors means creating and raising an error condition, while **catching errors** means handling and responding to errors that have been thrown.

Key Differences:

Aspect	Throwing	Catching
Purpose	Signal that an error occurred	Handle the error
When	When error condition is detected	When error is thrown
Syntax	`throw new Error()`	`try...catch`
Control	Stops execution	Continues execution
Location	Anywhere in code	Only in try block

1. Throwing Errors:

```
// Basic error throwing
function validateAge(age) {
    if (age < 0) {
        throw new Error('Age cannot be negative');
    }
    if (age > 150) {
        throw new Error('Age cannot be greater than 150');
    }
    return true;
}

// Throwing different error types
function processData(data) {
    if (!data) {
        throw new TypeError('Data is required');
    }
    if (typeof data !== 'object') {
        throw new TypeError('Data must be an object');
    }
    if (!data.id) {
        throw new ReferenceError('Data must have an id prop
    }
    return data;
}
```

2. Catching Errors:

```
// Basic error catching
try {
    validateAge(-5);
} catch (error) {
    console.error('Validation failed:', error.message);
}

// Catching specific error types
```

```
try {
    processData('invalid');
} catch (error) {
    if (error instanceof TypeError) {
        console.error('Type error:', error.message);
    } else if (error instanceof ReferenceError) {
        console.error('Reference error:', error.message);
    } else {
        console.error('Unknown error:', error.message);
    }
}
```

3. Error Propagation:

```
// Errors propagate up the call stack
function level1() {
    try {
        level2();
    } catch (error) {
        console.log('Level 1 caught:', error.message);
        throw error; // Re-throw to propagate further
    }
}

function level2() {
    try {
        level3();
    } catch (error) {
        console.log('Level 2 caught:', error.message);
        throw error; // Re-throw to propagate further
    }
}

function level3() {
```

```
    throw new Error('Error from level 3');
}
```

Best Practices:

1. Throw errors early and catch them late
2. Use specific error types for different scenarios
3. Provide meaningful error messages
4. Don't catch errors unless you can handle them
5. Re-throw errors when appropriate
6. Use error boundaries for UI components
7. Implement retry mechanisms for transient errors
8. Log errors for debugging and monitoring
9. Consider error recovery strategies
10. Test error handling paths

Related Questions: Error Handling, Error Types

⬆️ Back to Top

---

55. What are the different types of errors in JavaScript?

JavaScript has several built-in error types, each designed for specific error conditions. Understanding these types helps in proper error handling and debugging.

1. Built-in Error Types:

Error (Base Error):

```
// Generic error
const error = new Error('Something went wrong');
console.log(error.name); // "Error"
console.log(error.message); // "Something went wrong"
console.log(error instanceof Error); // true
```

TypeError:

```
// Type-related errors
try {
    const obj = null;
    obj.property; // TypeError: Cannot read property 'prope
} catch (error) {
    console.log(error instanceof TypeError); // true
    console.log(error.name); // "TypeError"
}
```

ReferenceError:

```
// Undefined variable errors
try {
    console.log(undefinedVariable); // ReferenceError: unde
} catch (error) {
    console.log(error instanceof ReferenceError); // true
    console.log(error.name); // "ReferenceError"
}
```

SyntaxError:

```
// Syntax errors (usually caught at parse time)
try {
    eval('const x = ;'); // SyntaxError: Unexpected token '
```

```
} catch (error) {
    console.log(error instanceof SyntaxError); // true
    console.log(error.name); // "SyntaxError"
}
```

RangeError:

```
// Range-related errors
try {
    const arr = new Array(-1); // RangeError: Invalid array
} catch (error) {
    console.log(error instanceof RangeError); // true
    console.log(error.name); // "RangeError"
}
```

URIError:

```
// URI-related errors
try {
    decodeURIComponent('%'); // URIError: URI malformed
} catch (error) {
    console.log(error instanceof URIError); // true
    console.log(error.name); // "URIError"
}
```

2. Custom Error Types:

```
// Custom error class
class ValidationError extends Error {
    constructor(message, field) {
        super(message);
        this.name = 'ValidationError';
        this.field = field;
```

```
    }
}

class NetworkError extends Error {
    constructor(message, statusCode) {
        super(message);
        this.name = 'NetworkError';
        this.statusCode = statusCode;
    }
}

// Using custom errors
function validateUser(user) {
    if (!user.name) {
        throw new ValidationError('Name is required', 'name
    }
    if (!user.email) {
        throw new ValidationError('Email is required', 'ema
    }
}
```

Best Practices:

1. Use specific error types for different scenarios
2. Create custom error classes for domain-specific errors
3. Include relevant context in error objects
4. Use error type checking for proper handling
5. Document custom error types
6. Follow consistent error naming conventions
7. Consider error inheritance hierarchies
8. Test error type creation and handling
9. Use error factories for consistent error creation

10. Implement error serialization for logging

Related Questions: Error Handling, Throwing and Catching Errors

⬆️ Back to Top

---

56. How do you create custom error types?

Custom error types allow you to create specific error classes for different scenarios in your application. This provides better error handling, debugging, and user experience.

1. Basic Custom Error Class:

```
class CustomError extends Error {
    constructor(message) {
        super(message);
        this.name = 'CustomError';
    }
}

// Usage
try {
    throw new CustomError('Something went wrong');
} catch (error) {
    console.log(error instanceof CustomError); // true
    console.log(error.name); // "CustomError"
    console.log(error.message); // "Something went wrong"
}
```

2. Error with Additional Properties:

```
class ValidationError extends Error {
    constructor(message, field, value) {
```

```
        super(message);
        this.name = 'ValidationError';
        this.field = field;
        this.value = value;
        this.timestamp = new Date().toISOString();
    }
}

// Usage
try {
    throw new ValidationError('Invalid email format', 'emai
} catch (error) {
    console.log(error.field); // "email"
    console.log(error.value); // "invalid-email"
    console.log(error.timestamp); // "2024-01-15T10:30:00.0
}
```

3. Error with Error Codes:

```
class ApiError extends Error {
    constructor(message, code, statusCode = 500) {
        super(message);
        this.name = 'ApiError';
        this.code = code;
        this.statusCode = statusCode;
    }
}

// Usage
try {
    throw new ApiError('User not found', 'USER_NOT_FOUND',
} catch (error) {
    console.log(error.code); // "USER_NOT_FOUND"
    console.log(error.statusCode); // 404
}
```

Best Practices:

1. Extend the base Error class
2. Set a descriptive name property
3. Include relevant context information
4. Implement proper serialization
5. Add useful methods for error handling
6. Use consistent naming conventions
7. Document your custom error types
8. Consider error inheritance hierarchies
9. Implement proper error logging
10. Test your custom error types

Related Questions: Error Handling, Error Types

Back to Top

Performance and Optimization

57. How do you optimize JavaScript performance?

JavaScript performance optimization involves improving the speed, efficiency, and responsiveness of JavaScript code. This includes optimizing algorithms, reducing memory usage, and improving execution time.

1. Algorithm Optimization:

```
// Inefficient O(n²) approach
function findDuplicates(arr) {
    const duplicates = [];
    for (let i = 0; i < arr.length; i++) {
```

```
        for (let j = i + 1; j < arr.length; j++) {
            if (arr[i] === arr[j] && !duplicates.includes(a
                duplicates.push(arr[i]);
            }
        }
    }
    return duplicates;
}

// Efficient O(n) approach
function findDuplicatesOptimized(arr) {
    const seen = new Set();
    const duplicates = new Set();

    for (const item of arr) {
        if (seen.has(item)) {
            duplicates.add(item);
        } else {
            seen.add(item);
        }
    }

    return Array.from(duplicates);
}
```

2. Memory Optimization:

```
// Avoid memory leaks
function createClosure() {
    const largeData = new Array(1000000).fill('data');

    return function() {
        // Only use what you need
        return largeData.length;
    };
}
```

```
// Better approach - don't hold references to large objects
function createClosureOptimized() {
    return function(data) {
        return data.length;
    };
}
```

3. DOM Optimization:

```
// Inefficient DOM manipulation
function updateList(items) {
    const list = document.getElementById('list');
    list.innerHTML = ''; // Clear all

    items.forEach(item => {
        const li = document.createElement('li');
        li.textContent = item.name;
        list.appendChild(li); // Multiple reflows
    });
}

// Efficient DOM manipulation
function updateListOptimized(items) {
    const list = document.getElementById('list');
    const fragment = document.createDocumentFragment();

    items.forEach(item => {
        const li = document.createElement('li');
        li.textContent = item.name;
        fragment.appendChild(li);
    });

    list.innerHTML = '';
```

```
    list.appendChild(fragment); // Single reflow
}
```

Best Practices:

1. Use efficient algorithms and data structures
2. Minimize DOM manipulation and reflows
3. Use event delegation and throttling
4. Avoid memory leaks and circular references
5. Use Web Workers for heavy computations
6. Implement lazy loading and code splitting
7. Use caching and memoization
8. Profile and measure performance
9. Use modern JavaScript features
10. Optimize for the critical rendering path

Related Questions: Memory Leaks, Debouncing and Throttling

⬆️ Back to Top

---

58. What are memory leaks and how do you prevent them?

Memory leaks occur when memory is allocated but never released, causing the application to consume increasing amounts of memory over time. This can lead to performance degradation and eventually crash the application.

1. Common Memory Leak Sources:

Global Variables:

```
// Bad - creates global variable
function createUser() {
    user = { name: 'John', data: new Array(1000000) };
}

// Good - use local variables
function createUser() {
    const user = { name: 'John', data: new Array(1000000) }
    return user;
}
```

Event Listeners:

```
// Bad - event listener not removed
function addClickListener() {
    document.addEventListener('click', function() {
        console.log('Clicked');
    });
}

// Good - remove event listeners
function addClickListener() {
    const handler = function() {
        console.log('Clicked');
    };

    document.addEventListener('click', handler);

    // Remove when no longer needed
    return () => document.removeEventListener('click', hand
}
```

2. Memory Leak Prevention:

Proper Cleanup:

```
class ResourceManager {
    constructor() {
        this.resources = new Set();
        this.eventListeners = new Map();
    }

    addResource(resource) {
        this.resources.add(resource);
    }

    addEventListener(element, event, handler) {
        element.addEventListener(event, handler);

        if (!this.eventListeners.has(element)) {
            this.eventListeners.set(element, []);
        }
        this.eventListeners.get(element).push({ event, hand
    }

    cleanup() {
        // Remove event listeners
        this.eventListeners.forEach((listeners, element) =>
            listeners.forEach(({ event, handler }) => {
                element.removeEventListener(event, handler)
            });
        });
        this.eventListeners.clear();

        // Clear resources
        this.resources.clear();
    }
}
```

Best Practices:

1. Avoid global variables
2. Remove event listeners when no longer needed
3. Use WeakMap and WeakSet for object references
4. Implement proper cleanup in classes
5. Avoid circular references
6. Use object pooling for frequently created objects
7. Monitor memory usage
8. Test for memory leaks
9. Use profiling tools
10. Implement garbage collection hints

Related Questions: Performance Optimization, Debouncing and Throttling

Back to Top

---

59. What is debouncing and throttling?

Debouncing and **throttling** are techniques used to limit the rate at which functions are executed, improving performance and user experience.

Debouncing:
Delays function execution until after a specified time has passed since the last invocation.

Throttling:
Limits function execution to once per specified time period.

1. Debouncing Implementation:

```
function debounce(func, delay) {
    let timeoutId;
    return function(...args) {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => func.apply(this, args)
    };
}

// Usage
const debouncedSearch = debounce(function(query) {
    console.log('Searching for:', query);
}, 300);

// Input field search
document.getElementById('search').addEventListener('input',
    debouncedSearch(e.target.value);
});
```

2. Throttling Implementation:

```
function throttle(func, limit) {
    let inThrottle;
    return function(...args) {
        if (!inThrottle) {
            func.apply(this, args);
            inThrottle = true;
            setTimeout(() => inThrottle = false, limit);
        }
    };
}

// Usage
const throttledScroll = throttle(function() {
    console.log('Scroll event');
}, 100);
```

```
window.addEventListener('scroll', throttledScroll);
```

Best Practices:

1. Use debouncing for search inputs
2. Use throttling for scroll and resize events
3. Choose appropriate delay times
4. Consider immediate execution for debouncing
5. Test performance impact
6. Use modern alternatives like Intersection Observer
7. Implement proper cleanup
8. Consider user experience
9. Use requestAnimationFrame for animations
10. Profile and measure performance

Related Questions: Performance Optimization, Memory Leaks

Back to Top

---

60. How do you optimize DOM operations?

DOM optimization involves improving the performance of Document Object Model operations, which are often the bottleneck in web applications.

1. Minimize DOM Access:

```
// Bad - multiple DOM queries
function updateElements() {
    document.getElementById('title').textContent = 'New Tit
    document.getElementById('title').className = 'updated';
```

```
    document.getElementById('title').style.color = 'red';
}

// Good - single DOM query
function updateElements() {
    const title = document.getElementById('title');
    title.textContent = 'New Title';
    title.className = 'updated';
    title.style.color = 'red';
}
```

2. Use Document Fragments:

```
// Bad - multiple reflows
function createList(items) {
    const list = document.getElementById('list');

    items.forEach(item => {
        const li = document.createElement('li');
        li.textContent = item.name;
        list.appendChild(li); // Causes reflow
    });
}

// Good - single reflow
function createList(items) {
    const list = document.getElementById('list');
    const fragment = document.createDocumentFragment();

    items.forEach(item => {
        const li = document.createElement('li');
        li.textContent = item.name;
        fragment.appendChild(li);
    });
```

```
    list.appendChild(fragment); // Single reflow
}
```

Best Practices:

1. Minimize DOM queries
2. Use document fragments
3. Batch DOM updates
4. Use CSS classes instead of inline styles
5. Implement event delegation
6. Use virtual scrolling for large lists
7. Avoid layout thrashing
8. Use requestAnimationFrame for animations
9. Implement proper cleanup
10. Profile and measure performance

Related Questions: Performance Optimization, Lazy Loading

⬆️ Back to Top

---

61. What is lazy loading and how do you implement it?

Lazy loading is a technique that delays the loading of resources until they are actually needed, improving initial page load performance and reducing bandwidth usage.

1. Image Lazy Loading:

```
// Intersection Observer API
const imageObserver = new IntersectionObserver((entries, ob
    entries.forEach(entry => {
```

```
        if (entry.isIntersecting) {
            const img = entry.target;
            img.src = img.dataset.src;
            img.classList.remove('lazy');
            observer.unobserve(img);
        }
    });
});

// Observe all lazy images
document.querySelectorAll('img[data-src]').forEach(img => {
    imageObserver.observe(img);
});
```

2. Module Lazy Loading:

```
// Dynamic import
async function loadModule(moduleName) {
    try {
        const module = await import(`./modules/${moduleName
        return module;
    } catch (error) {
        console.error('Failed to load module:', error);
        return null;
    }
}

// Usage
document.getElementById('loadModule').addEventListener('cli
    const module = await loadModule('heavyModule');
    if (module) {
        module.init();
    }
});
```

Best Practices:

1. Use Intersection Observer API
2. Implement proper error handling
3. Consider fallbacks for older browsers
4. Optimize loading strategies
5. Use appropriate thresholds
6. Implement proper cleanup
7. Consider user experience
8. Test performance impact
9. Use modern loading techniques
10. Monitor loading performance

Related Questions: Performance Optimization, DOM Optimization

Back to Top

Advanced Concepts

62. What is memoization and how do you implement it?

Memoization is an optimization technique that caches the results of expensive function calls and returns the cached result when the same inputs occur again. This can significantly improve performance for functions with expensive computations.

1. Basic Memoization:

```
function memoize(fn) {
    const cache = new Map();
```

```
    return function(...args) {
        const key = JSON.stringify(args);

        if (cache.has(key)) {
            console.log('Cache hit!');
            return cache.get(key);
        }

        console.log('Computing...');
        const result = fn.apply(this, args);
        cache.set(key, result);
        return result;
    };
}

// Example usage
function expensiveCalculation(n) {
    console.log(`Computing for ${n}`);
    let result = 0;
    for (let i = 0; i < n * 1000000; i++) {
        result += i;
    }
    return result;
}

const memoizedCalculation = memoize(expensiveCalculation);

console.log(memoizedCalculation(5)); // Computing... (takes
console.log(memoizedCalculation(5)); // Cache hit! (instant
```

2. Memoization with Cache Size Limit:

```
function memoizeWithLimit(fn, limit = 100) {
    const cache = new Map();

    return function(...args) {
```

```
        const key = JSON.stringify(args);

        if (cache.has(key)) {
            // Move to end (LRU)
            const value = cache.get(key);
            cache.delete(key);
            cache.set(key, value);
            return value;
        }

        const result = fn.apply(this, args);

        // Remove oldest if at limit
        if (cache.size >= limit) {
            const firstKey = cache.keys().next().value;
            cache.delete(firstKey);
        }

        cache.set(key, result);
        return result;
    };
}
```

3. Memoization for Recursive Functions:

```
function memoizedFibonacci() {
    const cache = new Map();

    function fibonacci(n) {
        if (n <= 1) return n;

        if (cache.has(n)) {
            return cache.get(n);
        }

        const result = fibonacci(n - 1) + fibonacci(n - 2);
```

```
            cache.set(n, result);
            return result;
        }

        return fibonacci;
    }

    const fib = memoizedFibonacci();
    console.log(fib(40)); // Much faster than naive implementat
```

Best Practices:

1. Use memoization for expensive, pure functions
2. Consider cache size limits for memory management
3. Use appropriate key generation strategies
4. Handle async functions carefully
5. Consider TTL for time-sensitive data
6. Use WeakMap for object-based caching
7. Test performance improvements
8. Be aware of memory usage
9. Consider cache invalidation strategies
10. Use memoization judiciously

Related Questions: Design Patterns, Functional Programming

⬆️ Back to Top

---

63. What are some common design patterns in JavaScript?

Design patterns are reusable solutions to commonly occurring problems in software design. They provide templates for solving problems in a consistent way.

1. Singleton Pattern:

```
class Singleton {
    constructor() {
        if (Singleton.instance) {
            return Singleton.instance;
        }

        this.data = {};
        Singleton.instance = this;
    }

    setData(key, value) {
        this.data[key] = value;
    }

    getData(key) {
        return this.data[key];
    }
}

// Usage
const instance1 = new Singleton();
const instance2 = new Singleton();
console.log(instance1 === instance2); // true
```

2. Observer Pattern:

```
class EventEmitter {
    constructor() {
        this.events = {};
    }

    on(event, listener) {
        if (!this.events[event]) {
```

```
            this.events[event] = [];
        }
        this.events[event].push(listener);
    }

    emit(event, ...args) {
        if (this.events[event]) {
            this.events[event].forEach(listener => listener
        }
    }

    off(event, listener) {
        if (this.events[event]) {
            this.events[event] = this.events[event].filter(
        }
    }
}

// Usage
const emitter = new EventEmitter();
emitter.on('user:login', (user) => {
    console.log(`User ${user.name} logged in`);
});
emitter.emit('user:login', { name: 'John', id: 1 });
```

3. Factory Pattern:

```
class AnimalFactory {
    static createAnimal(type, name) {
        switch (type) {
            case 'dog':
                return new Dog(name);
            case 'cat':
                return new Cat(name);
            case 'bird':
                return new Bird(name);
```

```
            default:
                throw new Error('Unknown animal type');
        }
    }
}

class Dog {
    constructor(name) {
        this.name = name;
        this.type = 'dog';
    }

    speak() {
        return 'Woof!';
    }
}

// Usage
const dog = AnimalFactory.createAnimal('dog', 'Buddy');
const cat = AnimalFactory.createAnimal('cat', 'Whiskers');
```

Best Practices:

1. Choose patterns that fit your problem
2. Don't over-engineer simple solutions
3. Understand the trade-offs of each pattern
4. Use patterns consistently across your codebase
5. Consider performance implications
6. Document pattern usage
7. Test pattern implementations
8. Consider modern alternatives
9. Use composition over inheritance

10. Keep patterns simple and readable

Related Questions: Memoization, Functional Programming

⬆️ Back to Top

---

64. What is functional programming?

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It emphasizes immutability, pure functions, and function composition.

1. Pure Functions:

```
// Pure function - same input always produces same output
function add(a, b) {
    return a + b;
}

// Impure function - depends on external state
let counter = 0;
function increment() {
    return ++counter; // Side effect
}

// Pure function version
function incrementPure(counter) {
    return counter + 1;
}
```

2. Immutability:

```
// Mutable approach
const user = { name: 'John', age: 30 };
```

```
user.age = 31; // Mutates original object

// Immutable approach
const user = { name: 'John', age: 30 };
const updatedUser = { ...user, age: 31 }; // Creates new ob
```

3. Higher-Order Functions:

```
// Functions that take other functions as arguments
function map(array, fn) {
    const result = [];
    for (let i = 0; i < array.length; i++) {
        result.push(fn(array[i], i, array));
    }
    return result;
}

function filter(array, predicate) {
    const result = [];
    for (let i = 0; i < array.length; i++) {
        if (predicate(array[i], i, array)) {
            result.push(array[i]);
        }
    }
    return result;
}

// Usage
const numbers = [1, 2, 3, 4, 5];
const doubled = map(numbers, x => x * 2);
const evens = filter(numbers, x => x % 2 === 0);
```

Best Practices:

1. Write pure functions whenever possible

2. Avoid side effects and mutations
3. Use immutable data structures
4. Compose functions instead of nesting
5. Use higher-order functions
6. Implement proper error handling
7. Consider performance implications
8. Use functional libraries when appropriate
9. Test pure functions thoroughly
10. Balance functional and imperative approaches

Related Questions: Memoization, Design Patterns

⬆️ Back to Top

---

65. What is immutability and why is it important?

Immutability is the principle that data should not be modified after it's created. Instead of changing existing data, you create new data with the desired changes.

1. Immutable Objects:

```
// Mutable approach
const user = { name: 'John', age: 30 };
user.age = 31; // Mutates original object

// Immutable approach
const user = { name: 'John', age: 30 };
const updatedUser = { ...user, age: 31 }; // Creates new ob

// Deep immutability for nested objects
const user = {
    name: 'John',
```

```
    age: 30,
    address: {
        street: '123 Main St',
        city: 'New York'
    }
};

// Deep copy
const deepCopy = {
    ...user,
    address: {
        ...user.address,
        city: 'Boston'
    }
};
```

2. Immutable Arrays:

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

// Add element
const newNumbers = [...numbers, 6];

// Remove element
const filteredNumbers = numbers.filter(n => n !== 3);

// Update element
const updatedNumbers = numbers.map(n => n === 3 ? 30 : n);
```

Best Practices:

1. Use spread operator for shallow copies
2. Use libraries like Immer for complex updates
3. Implement structural sharing when possible

4. Use immutable data structures
5. Avoid deep cloning when not necessary
6. Use reference equality for comparisons
7. Consider performance implications
8. Use proper error handling
9. Document immutable patterns
10. Test immutable operations

Related Questions: Functional Programming, Pure Functions

Back to Top

---

66. What are pure functions?

Pure functions are functions that always return the same output for the same input and have no side effects. They don't modify external state or depend on external variables.

1. Characteristics of Pure Functions:

```
// Pure function
function add(a, b) {
    return a + b;
}

// Impure function
let counter = 0;
function increment() {
    return ++counter; // Side effect: modifies external sta
}

// Pure function version
function incrementPure(counter) {
```

```
    return counter + 1;
}
```

2. Benefits of Pure Functions:

```
// Predictable behavior
function calculateTax(amount, rate) {
    return amount * rate;
}

// Always returns same result for same input
console.log(calculateTax(100, 0.1)); // 10
console.log(calculateTax(100, 0.1)); // 10 (always the same

// Easy to test
function testCalculateTax() {
    const result = calculateTax(100, 0.1);
    console.assert(result === 10, 'Tax calculation failed')
}
```

Best Practices:

1. Avoid side effects
2. Don't modify input parameters
3. Don't rely on external state
4. Return consistent outputs
5. Use dependency injection
6. Handle errors gracefully
7. Write comprehensive tests
8. Document function behavior
9. Use pure functions for business logic

10. Consider performance implications

Related Questions: Immutability, Recursion

⬆️ Back to Top

---

67. What is recursion and how do you use it?

Recursion is a programming technique where a function calls itself to solve a problem. It's particularly useful for problems that can be broken down into smaller, similar subproblems.

1. Basic Recursion:

```
// Factorial function
function factorial(n) {
    if (n <= 1) {
        return 1; // Base case
    }
    return n * factorial(n - 1); // Recursive case
}

console.log(factorial(5)); // 120

// Fibonacci sequence
function fibonacci(n) {
    if (n <= 1) {
        return n; // Base case
    }
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursiv
}

console.log(fibonacci(10)); // 55
```

2. Tail Recursion:

```
// Tail recursive factorial
function factorialTail(n, acc = 1) {
    if (n <= 1) {
        return acc; // Base case
    }
    return factorialTail(n - 1, n * acc); // Tail call
}

// Tail recursive fibonacci
function fibonacciTail(n, a = 0, b = 1) {
    if (n === 0) return a;
    if (n === 1) return b;
    return fibonacciTail(n - 1, b, a + b);
}

console.log(fibonacciTail(10)); // 55
```

3. Recursion with Arrays:

```
// Sum of array elements
function sumArray(arr, index = 0) {
    if (index >= arr.length) {
        return 0; // Base case
    }
    return arr[index] + sumArray(arr, index + 1); // Recurs
}

// Find maximum element
function findMax(arr, index = 0, max = -Infinity) {
    if (index >= arr.length) {
        return max; // Base case
    }
    const currentMax = Math.max(max, arr[index]);
```

```
    return findMax(arr, index + 1, currentMax); // Recursiv
}
```

Best Practices:

1. Always define a base case
2. Ensure the recursive case moves toward the base case
3. Use tail recursion when possible
4. Consider memoization for expensive operations
5. Handle stack overflow errors
6. Test with edge cases
7. Consider iterative alternatives
8. Use recursion for tree/graph problems
9. Document recursive functions clearly
10. Consider performance implications

Related Questions: Pure Functions, Memoization

⬆️ Back to Top

Advanced Topics

68. What are Browser APIs and how do you use them?

Browser APIs are interfaces provided by web browsers that allow JavaScript to interact with browser features and system resources. They enable web applications to access device capabilities, storage, and other browser functionality.

1. Local Storage API:

```
// Storing data
localStorage.setItem('username', 'john_doe');
localStorage.setItem('userPreferences', JSON.stringify({
    theme: 'dark',
    language: 'en',
    notifications: true
}));

// Retrieving data
const username = localStorage.getItem('username');
const preferences = JSON.parse(localStorage.getItem('userPr

// Removing data
localStorage.removeItem('username');

// Clearing all data
localStorage.clear();

// Checking if storage is available
function isLocalStorageAvailable() {
    try {
        const test = 'test';
        localStorage.setItem(test, test);
        localStorage.removeItem(test);
        return true;
    } catch (e) {
        return false;
    }
}
```

2. Session Storage API:

```
// Session storage (data persists only for the session)
sessionStorage.setItem('currentPage', 'dashboard');
sessionStorage.setItem('sessionId', 'abc123');
```

```
// Retrieve and use
const currentPage = sessionStorage.getItem('currentPage');
const sessionId = sessionStorage.getItem('sessionId');

// Session storage vs Local storage
console.log('Session storage:', sessionStorage.length);
console.log('Local storage:', localStorage.length);
```

3. Fetch API:

```
// Basic fetch
async function fetchData(url) {
    try {
        const response = await fetch(url);

        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response
        }

        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Fetch error:', error);
        throw error;
    }
}

// Fetch with options
async function postData(url, data) {
    const response = await fetch(url, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Authorization': 'Bearer ' + token
        },
```

```
        body: JSON.stringify(data)
    });

    return response.json();
}
```

Best Practices:

1. Always check for API support before using
2. Handle errors gracefully
3. Use appropriate error messages
4. Consider fallbacks for older browsers
5. Implement proper cleanup
6. Use modern APIs when available
7. Test across different browsers
8. Consider performance implications
9. Follow security best practices
10. Document API usage

Related Questions: Security, Testing

⬆️ Back to Top

---

69. What are common security vulnerabilities?

Security vulnerabilities in web applications can lead to data breaches, unauthorized access, and other security issues. Understanding these vulnerabilities is crucial for building secure applications.

1. Cross-Site Scripting (XSS):

```
// Vulnerable code
function displayUserInput(input) {
    document.getElementById('output').innerHTML = input; //
}

// Safe code
function displayUserInput(input) {
    const output = document.getElementById('output');
    output.textContent = input; // Safe - textContent escap
}

// Or use proper escaping
function escapeHtml(text) {
    const div = document.createElement('div');
    div.textContent = text;
    return div.innerHTML;
}
```

2. Cross-Site Request Forgery (CSRF):

```
CSRF stands for Cross-Site Request Forgery, a type of web s
// CSRF protection with tokens
class CSRFProtection {
    static generateToken() {
        return crypto.randomUUID();
    }

    static setToken() {
        const token = this.generateToken();
        sessionStorage.setItem('csrf-token', token);
        return token;
    }

    static getToken() {
        return sessionStorage.getItem('csrf-token');
```

```
    }

    static validateToken(token) {
        return token === this.getToken();
    }
}
```

3. Input Validation:

```
// Input validation class
class InputValidator {
    static validateEmail(email) {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(email);
    }

    static validatePassword(password) {
        const minLength = 8;
        const hasUpperCase = /[A-Z]/.test(password);
        const hasLowerCase = /[a-z]/.test(password);
        const hasNumbers = /\d/.test(password);
        const hasSpecialChar = /[!@#$%^&*(),.?":{}|<>]/.tes

        return {
            isValid: password.length >= minLength && hasUpp
                    hasLowerCase && hasNumbers && hasSpecia
            errors: [
                password.length < minLength && 'Password mu
                !hasUpperCase && 'Password must contain upp
                !hasLowerCase && 'Password must contain low
                !hasNumbers && 'Password must contain numbe
                !hasSpecialChar && 'Password must contain s
            ].filter(Boolean)
        };
```

```
        }
    }
```

Best Practices:

1. Always validate and sanitize input
2. Use parameterized queries
3. Implement proper authentication
4. Use HTTPS everywhere
5. Set security headers
6. Implement rate limiting
7. Keep dependencies updated
8. Use Content Security Policy
9. Implement proper session management
10. Regular security audits

Related Questions: Browser APIs, Testing

Back to Top

---

70. What is unit testing?

Unit testing is a software testing method where individual units or components of a software application are tested in isolation. It helps ensure that each part of the code works correctly.

1. Basic Unit Testing:

```
// Simple test function
function test(description, testFunction) {
    try {
```

```
        testFunction();
        console.log(`✅ ${description}`);
    } catch (error) {
        console.log(`❌ ${description}: ${error.message}`);
    }
}

// Assertion functions
function assert(condition, message) {
    if (!condition) {
        throw new Error(message || 'Assertion failed');
    }
}

function assertEqual(actual, expected, message) {
    if (actual !== expected) {
        throw new Error(message || `Expected ${expected}, b
    }
}

// Example tests
function add(a, b) {
    return a + b;
}

// Test cases
test('add function works correctly', () => {
    assertEqual(add(2, 3), 5);
    assertEqual(add(-1, 1), 0);
    assertEqual(add(0, 0), 0);
});
```

2. Mocking and Stubbing:

```
// Mock function
function createMock() {
```

```
    const calls = [];
    const mockFn = function(...args) {
        calls.push(args);
        return mockFn.returnValue;
    };

    mockFn.calls = calls;
    mockFn.returnValue = undefined;
    mockFn.mockReturnValue = (value) => {
        mockFn.returnValue = value;
        return mockFn;
    };

    return mockFn;
}

// Usage
const mockFetch = createMock();
mockFetch.mockReturnValue(Promise.resolve({ json: () => ({
```

Best Practices:

1. Write tests before or alongside code
2. Test edge cases and error conditions
3. Keep tests simple and focused
4. Use descriptive test names
5. Mock external dependencies
6. Test async code properly
7. Maintain good test coverage
8. Keep tests fast and reliable
9. Refactor tests when refactoring code
10. Use appropriate assertions

Related Questions: Security, Performance

Back to Top

---

71. What is performance optimization?

Performance optimization involves improving the speed, efficiency, and responsiveness of web applications. It's crucial for providing a good user experience and reducing resource usage.

1. Code Optimization:

```
// Inefficient code
function processLargeArray(arr) {
    let result = [];
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] > 100) {
            result.push(arr[i] * 2);
        }
    }
    return result;
}

// Optimized code
function processLargeArray(arr) {
    return arr
        .filter(item => item > 100)
        .map(item => item * 2);
}
```

2. Memory Optimization:

```
// Memory leak prevention
class DataProcessor {
```

```
    constructor() {
        this.cache = new Map();
        this.maxCacheSize = 1000;
    }

    processData(data) {
        const key = JSON.stringify(data);

        if (this.cache.has(key)) {
            return this.cache.get(key);
        }

        const result = this.expensiveOperation(data);

        // Prevent memory leaks
        if (this.cache.size >= this.maxCacheSize) {
            const firstKey = this.cache.keys().next().value
            this.cache.delete(firstKey);
        }

        this.cache.set(key, result);
        return result;
    }

    cleanup() {
        this.cache.clear();
    }
}
```

3. DOM Optimization:

```
// Inefficient DOM manipulation
function updateList(items) {
    const list = document.getElementById('list');
    list.innerHTML = '';
```

```
    items.forEach(item => {
        const li = document.createElement('li');
        li.textContent = item.name;
        list.appendChild(li);
    });
}

// Optimized DOM manipulation
function updateListOptimized(items) {
    const list = document.getElementById('list');
    const fragment = document.createDocumentFragment();

    items.forEach(item => {
        const li = document.createElement('li');
        li.textContent = item.name;
        fragment.appendChild(li);
    });

    list.innerHTML = '';
    list.appendChild(fragment);
}
```

Best Practices:

1. Profile before optimizing
2. Focus on bottlenecks
3. Use appropriate data structures
4. Minimize DOM manipulation
5. Implement proper caching
6. Use lazy loading
7. Optimize images and assets
8. Minimize HTTP requests
9. Use compression

10. Monitor performance metrics

Related Questions: Testing, Browser APIs

⬆️ Back to Top

---

72. What are Web Workers and how do you use them?

Web Workers allow you to run JavaScript code in background threads, separate from the main UI thread. This prevents blocking the UI during heavy computations.

1. Basic Web Worker:

```
// main.js
const worker = new Worker('worker.js');

worker.postMessage({ command: 'calculate', data: [1, 2, 3,

worker.onmessage = function(e) {
    console.log('Result from worker:', e.data);
};

worker.onerror = function(error) {
    console.error('Worker error:', error);
};

// worker.js
self.onmessage = function(e) {
    const { command, data } = e.data;

    if (command === 'calculate') {
        const result = data.reduce((sum, num) => sum + num,
        self.postMessage({ result });
    }
};
```

2. Shared Workers:

```
// main.js
const sharedWorker = new SharedWorker('shared-worker.js');
sharedWorker.port.start();

sharedWorker.port.postMessage('Hello from main thread');

sharedWorker.port.onmessage = function(e) {
    console.log('Message from shared worker:', e.data);
};

// shared-worker.js
const connections = [];

self.addEventListener('connect', function(e) {
    const port = e.ports[0];
    connections.push(port);

    port.onmessage = function(e) {
        connections.forEach(conn => {
            if (conn !== port) {
                conn.postMessage(e.data);
            }
        });
    };
});
```

Best Practices:

1. Use workers for CPU-intensive tasks
2. Minimize data transfer between threads
3. Handle errors properly
4. Clean up workers when done

5. Use transferable objects for large data
6. Consider worker pools for multiple tasks
7. Test worker functionality
8. Use appropriate worker types
9. Monitor worker performance
10. Implement proper communication patterns

Related Questions: Performance, Browser APIs

Back to Top

Browser APIs

73. What is localStorage vs sessionStorage vs cookies?

Storage mechanisms in web browsers provide different ways to store data on the client side. Each has unique characteristics, use cases, and limitations.

1. localStorage:

Characteristics:

```
// localStorage - Persists indefinitely until explicitly cl
class LocalStorageManager {
    // Set item
    static setItem(key, value) {
        try {
            const serializedValue = JSON.stringify(value);
            localStorage.setItem(key, serializedValue);
            return true;
        } catch (error) {
            console.error('Error saving to localStorage:',
```

```
            return false;
        }
    }

    // Get item
    static getItem(key) {
        try {
            const item = localStorage.getItem(key);
            return item ? JSON.parse(item) : null;
        } catch (error) {
            console.error('Error reading from localStorage:
            return null;
        }
    }

    // Remove item
    static removeItem(key) {
        localStorage.removeItem(key);
    }

    // Clear all
    static clear() {
        localStorage.clear();
    }

    // Get all keys
    static getAllKeys() {
        return Object.keys(localStorage);
    }

    // Check storage availability
    static isAvailable() {
        try {
            const test = '__test__';
            localStorage.setItem(test, test);
            localStorage.removeItem(test);
            return true;
```

```
        } catch (e) {
            return false;
        }
    }
}

// Usage examples
LocalStorageManager.setItem('user', {
    id: 1,
    name: 'John Doe',
    preferences: { theme: 'dark', language: 'en' }
});

const user = LocalStorageManager.getItem('user');
console.log(user.name); // 'John Doe'
```

2. sessionStorage:

Characteristics:

```
// sessionStorage - Persists only for the session (tab/wind
class SessionStorageManager {
    static setItem(key, value) {
        try {
            sessionStorage.setItem(key, JSON.stringify(valu
            return true;
        } catch (error) {
            console.error('Error saving to sessionStorage:'
            return false;
        }
    }

    static getItem(key) {
        try {
            const item = sessionStorage.getItem(key);
            return item ? JSON.parse(item) : null;
```

```
        } catch (error) {
            return null;
        }
    }

    static removeItem(key) {
        sessionStorage.removeItem(key);
    }

    static clear() {
        sessionStorage.clear();
    }
}

// Usage - perfect for temporary data
SessionStorageManager.setItem('currentSession', {
    sessionId: 'abc123',
    startTime: Date.now(),
    currentPage: '/dashboard'
});

// Data is lost when tab/window is closed
```

3. Cookies:

Characteristics:

```
// Cookies - Sent with every HTTP request, can have expirat
class CookieManager {
    static setCookie(name, value, options = {}) {
        let cookie = `${encodeURIComponent(name)}=${encodeU

        if (options.days) {
            const expires = new Date();
            expires.setTime(expires.getTime() + (options.da
            cookie += `; expires=${expires.toUTCString()}`;
```

```
        }

        if (options.path) {
            cookie += `; path=${options.path}`;
        } else {
            cookie += '; path=/';
        }

        if (options.domain) {
            cookie += `; domain=${options.domain}`;
        }

        if (options.secure) {
            cookie += '; secure';
        }

        if (options.sameSite) {
            cookie += `; SameSite=${options.sameSite}`;
        }

        if (options.httpOnly) {
            // Note: httpOnly can only be set server-side
            console.warn('httpOnly flag can only be set ser
        }

        document.cookie = cookie;
    }

    static getCookie(name) {
        const nameEQ = encodeURIComponent(name) + '=';
        const cookies = document.cookie.split(';');

        for (let cookie of cookies) {
            cookie = cookie.trim();
            if (cookie.indexOf(nameEQ) === 0) {
                return decodeURIComponent(cookie.substring(
            }
```

```
        }
        return null;
    }

    static deleteCookie(name, path = '/') {
        document.cookie = `${encodeURIComponent(name)}=; ex
    }

    static getAllCookies() {
        const cookies = {};
        document.cookie.split(';').forEach(cookie => {
            const [name, value] = cookie.trim().split('=');
            if (name) {
                cookies[decodeURIComponent(name)] = decodeU
            }
        });
        return cookies;
    }
}

// Usage
CookieManager.setCookie('authToken', 'xyz789', {
    days: 7,
    path: '/',
    secure: true,
    sameSite: 'Strict'
});

const token = CookieManager.getCookie('authToken');
```

4. Comparison Table:

```
// Storage Comparison
const storageComparison = {
    localStorage: {
        capacity: '5-10MB',
```

```
        expiration: 'Never (until manually cleared)',
        accessibility: 'Any window/tab from same origin',
        sentWithRequests: 'No',
        apiType: 'Synchronous',
        useCase: 'Long-term data storage'
    },
    sessionStorage: {
        capacity: '5-10MB',
        expiration: 'Tab/window close',
        accessibility: 'Same tab/window only',
        sentWithRequests: 'No',
        apiType: 'Synchronous',
        useCase: 'Temporary session data'
    },
    cookies: {
        capacity: '4KB per cookie',
        expiration: 'Configurable',
        accessibility: 'Any window/tab from same origin',
        sentWithRequests: 'Yes (with every HTTP request)',
        apiType: 'Synchronous',
        useCase: 'Authentication tokens, small data'
    }
};
```

5. Practical Usage Examples:

```
// Combined storage strategy
class StorageStrategy {
    // Use localStorage for user preferences
    static saveUserPreferences(preferences) {
        LocalStorageManager.setItem('userPrefs', preference
    }

    // Use sessionStorage for form data
    static saveFormData(formId, data) {
        SessionStorageManager.setItem(`form_${formId}`, dat
```

```
    }

    // Use cookies for authentication
    static saveAuthToken(token, remember = false) {
        if (remember) {
            CookieManager.setCookie('authToken', token, {
                days: 30,
                secure: true,
                sameSite: 'Strict'
            });
        } else {
            // Session cookie (expires when browser closes)
            CookieManager.setCookie('authToken', token, {
                secure: true,
                sameSite: 'Strict'
            });
        }
    }

    // Clear all storage
    static clearAllStorage() {
        localStorage.clear();
        sessionStorage.clear();
        // Delete all cookies
        const cookies = CookieManager.getAllCookies();
        Object.keys(cookies).forEach(name => {
            CookieManager.deleteCookie(name);
        });
    }
}

// Real-world example: Shopping cart
class ShoppingCart {
    constructor() {
        this.storageKey = 'shopping_cart';
    }
```

```
    addItem(item) {
        const cart = LocalStorageManager.getItem(this.stora
        cart.push(item);
        LocalStorageManager.setItem(this.storageKey, cart);
    }

    getItems() {
        return LocalStorageManager.getItem(this.storageKey)
    }

    clear() {
        LocalStorageManager.removeItem(this.storageKey);
    }
}
```

Best Practices:

1. Use localStorage for persistent user preferences
2. Use sessionStorage for temporary form data
3. Use cookies for authentication tokens
4. Always validate and sanitize stored data
5. Implement error handling for quota exceeded
6. Encrypt sensitive data before storing
7. Set appropriate cookie security flags
8. Clear sensitive data on logout
9. Monitor storage usage
10. Provide fallbacks for private browsing mode

Related Questions: Fetch API, Security

⬆️ Back to Top

---

74. What is the Fetch API?

Fetch API provides a modern, promise-based way to make HTTP requests in JavaScript. It's more powerful and flexible than the older XMLHttpRequest.

1. Basic Fetch Usage:

```
// GET request
async function fetchData(url) {
    try {
        const response = await fetch(url);

        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response
        }

        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Fetch error:', error);
        throw error;
    }
}

// Usage
fetchData('https://api.example.com/users')
    .then(users => console.log(users))
    .catch(error => console.error(error));
```

2. Advanced Fetch Operations:

```
// Comprehensive Fetch wrapper
class FetchClient {
    constructor(baseURL, defaultOptions = {}) {
        this.baseURL = baseURL;
```

```
        this.defaultOptions = defaultOptions;
    }

    async request(endpoint, options = {}) {
        const url = `${this.baseURL}${endpoint}`;
        const config = {
            ...this.defaultOptions,
            ...options,
            headers: {
                'Content-Type': 'application/json',
                ...this.defaultOptions.headers,
                ...options.headers
            }
        };

        try {
            const response = await fetch(url, config);

            // Handle different response types
            const contentType = response.headers.get('conte
            let data;

            if (contentType && contentType.includes('applic
                data = await response.json();
            } else if (contentType && contentType.includes(
                data = await response.text();
            } else {
                data = await response.blob();
            }

            if (!response.ok) {
                throw new Error(`HTTP ${response.status}: $
            }

            return {
                data,
                status: response.status,
```

```
                headers: response.headers
            };
        } catch (error) {
            console.error(`Request failed for ${url}:`, err
            throw error;
        }
    }

    get(endpoint, options = {}) {
        return this.request(endpoint, { ...options, method:
    }

    post(endpoint, body, options = {}) {
        return this.request(endpoint, {
            ...options,
            method: 'POST',
            body: JSON.stringify(body)
        });
    }

    put(endpoint, body, options = {}) {
        return this.request(endpoint, {
            ...options,
            method: 'PUT',
            body: JSON.stringify(body)
        });
    }

    patch(endpoint, body, options = {}) {
        return this.request(endpoint, {
            ...options,
            method: 'PATCH',
            body: JSON.stringify(body)
        });
    }

    delete(endpoint, options = {}) {
```

```
        return this.request(endpoint, { ...options, method:
    }
}

// Usage
const api = new FetchClient('https://api.example.com', {
    headers: {
        'Authorization': 'Bearer token123'
    }
});

// GET request
const users = await api.get('/users');

// POST request
const newUser = await api.post('/users', {
    name: 'John Doe',
    email: 'john@example.com'
});
```

3. Fetch with Advanced Features:

```
// Request with timeout
async function fetchWithTimeout(url, options = {}, timeout
    const controller = new AbortController();
    const timeoutId = setTimeout(() => controller.abort(),

    try {
        const response = await fetch(url, {
            ...options,
            signal: controller.signal
        });
        clearTimeout(timeoutId);
        return response;
    } catch (error) {
        clearTimeout(timeoutId);
```

```
        if (error.name === 'AbortError') {
            throw new Error('Request timeout');
        }
        throw error;
    }
}

// Retry logic
async function fetchWithRetry(url, options = {}, maxRetries
    for (let i = 0; i < maxRetries; i++) {
        try {
            const response = await fetch(url, options);
            if (response.ok) {
                return response;
            }

            // Don't retry on client errors (4xx)
            if (response.status >= 400 && response.status <
                throw new Error(`Client error: ${response.s
            }
        } catch (error) {
            if (i === maxRetries - 1) throw error;

            // Exponential backoff
            const delay = Math.pow(2, i) * 1000;
            await new Promise(resolve => setTimeout(resolve
        }
    }
}

// Parallel requests
async function fetchMultiple(urls) {
    try {
        const promises = urls.map(url => fetch(url).then(r
        const results = await Promise.all(promises);
        return results;
    } catch (error) {
```

```
        console.error('Error fetching multiple URLs:', erro
        throw error;
    }
}

// Sequential requests
async function fetchSequential(urls) {
    const results = [];
    for (const url of urls) {
        try {
            const response = await fetch(url);
            const data = await response.json();
            results.push(data);
        } catch (error) {
            console.error(`Error fetching ${url}:`, error);
            results.push(null);
        }
    }
    return results;
}
```

4. File Upload with Fetch:

```
// Upload single file
async function uploadFile(url, file, onProgress) {
    const formData = new FormData();
    formData.append('file', file);
    formData.append('fileName', file.name);

    try {
        const response = await fetch(url, {
            method: 'POST',
            body: formData,
            // Don't set Content-Type header - browser will
        });
```

```
        if (!response.ok) {
            throw new Error(`Upload failed: ${response.stat
        }

        return await response.json();
    } catch (error) {
        console.error('Upload error:', error);
        throw error;
    }
}

// Upload multiple files
async function uploadMultipleFiles(url, files) {
    const formData = new FormData();

    files.forEach((file, index) => {
        formData.append(`file${index}`, file);
    });

    const response = await fetch(url, {
        method: 'POST',
        body: formData
    });

    return await response.json();
}

// Usage
const fileInput = document.getElementById('fileInput');
fileInput.addEventListener('change', async (e) => {
    const file = e.target.files[0];
    if (file) {
        try {
            const result = await uploadFile('/api/upload',
            console.log('Upload successful:', result);
        } catch (error) {
            console.error('Upload failed:', error);
```

```
        }
    }
});
```

5. Fetch with Caching:

```
// Cache-first strategy
class CachedFetch {
    constructor(cacheName = 'api-cache') {
        this.cacheName = cacheName;
    }

    async fetch(url, options = {}, cacheTime = 60000) {
        const cacheKey = `${url}_${JSON.stringify(options)}
        const cached = this.getCachedData(cacheKey);

        if (cached && Date.now() - cached.timestamp < cache
            return cached.data;
        }

        try {
            const response = await fetch(url, options);
            const data = await response.json();

            this.setCachedData(cacheKey, data);
            return data;
        } catch (error) {
            // Return stale cache if available
            if (cached) {
                console.warn('Using stale cache due to fetc
                return cached.data;
            }
            throw error;
        }
    }
```

```
    getCachedData(key) {
        const item = localStorage.getItem(key);
        return item ? JSON.parse(item) : null;
    }

    setCachedData(key, data) {
        localStorage.setItem(key, JSON.stringify({
            data,
            timestamp: Date.now()
        }));
    }

    clearCache() {
        Object.keys(localStorage).forEach(key => {
            if (key.startsWith(this.cacheName)) {
                localStorage.removeItem(key);
            }
        });
    }
}
```

Best Practices:

1. Always handle errors properly
2. Check response.ok before processing
3. Set appropriate timeouts
4. Implement retry logic for transient failures
5. Use AbortController for cancellation
6. Handle different response types
7. Implement proper loading states
8. Use credentials option for cookies
9. Set appropriate CORS headers

10. Implement request caching when appropriate

Related Questions: Async/Await, Promises, Web Workers

⬆️ Back to Top

---

75. What are Web Workers?

Web Workers allow you to run JavaScript code in background threads, separate from the main UI thread. This prevents blocking the UI during heavy computations and improves application performance.

1. Basic Web Worker:

```
// main.js - Main thread
class WorkerManager {
    constructor(workerScript) {
        this.worker = new Worker(workerScript);
        this.messageHandlers = new Map();
        this.messageId = 0;

        this.worker.onmessage = (event) => {
            const { id, result, error } = event.data;
            const handler = this.messageHandlers.get(id);

            if (handler) {
                if (error) {
                    handler.reject(new Error(error));
                } else {
                    handler.resolve(result);
                }
                this.messageHandlers.delete(id);
            }
        };
```

```
        this.worker.onerror = (error) => {
            console.error('Worker error:', error);
        };
    }

    execute(command, data) {
        return new Promise((resolve, reject) => {
            const id = this.messageId++;
            this.messageHandlers.set(id, { resolve, reject
            this.worker.postMessage({ id, command, data });
        });
    }

    terminate() {
        this.worker.terminate();
        this.messageHandlers.clear();
    }
}

// Usage
const workerManager = new WorkerManager('worker.js');

// Execute heavy computation
workerManager.execute('processLargeArray', [1, 2, 3, /*...*
    .then(result => console.log('Result:', result))
    .catch(error => console.error('Error:', error));
```

```
// worker.js - Worker thread
self.onmessage = function(event) {
    const { id, command, data } = event.data;

    try {
        let result;

        switch(command) {
```

```
            case 'processLargeArray':
                result = processLargeArray(data);
                break;
            case 'calculatePrimes':
                result = calculatePrimes(data);
                break;
            case 'sortData':
                result = sortData(data);
                break;
            default:
                throw new Error(`Unknown command: ${command
        }

        self.postMessage({ id, result });
    } catch (error) {
        self.postMessage({ id, error: error.message });
    }
};

function processLargeArray(arr) {
    // Heavy computation
    return arr.map(item => item * 2).filter(item => item >
}

function calculatePrimes(max) {
    const primes = [];
    for (let i = 2; i <= max; i++) {
        let isPrime = true;
        for (let j = 2; j <= Math.sqrt(i); j++) {
            if (i % j === 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) primes.push(i);
    }
    return primes;
```

```
}

function sortData(data) {
    return data.sort((a, b) => a - b);
}
```

2. Shared Workers:

```
// main.js - Multiple tabs can connect
const sharedWorker = new SharedWorker('shared-worker.js');

sharedWorker.port.start();

// Send message
sharedWorker.port.postMessage({
    type: 'register',
    clientId: Date.now()
});

// Receive messages
sharedWorker.port.onmessage = function(event) {
    console.log('Message from shared worker:', event.data);
};

// shared-worker.js
const connections = [];

self.addEventListener('connect', function(e) {
    const port = e.ports[0];
    connections.push(port);

    port.onmessage = function(event) {
        const { type, clientId, message } = event.data;

        if (type === 'register') {
            port.postMessage({
```

```
                type: 'registered',
                totalConnections: connections.length
            });
        } else if (type === 'broadcast') {
            // Broadcast to all connected tabs
            connections.forEach(conn => {
                if (conn !== port) {
                    conn.postMessage({
                        type: 'message',
                        from: clientId,
                        message: message
                    });
                }
            });
        }
    };

    port.start();
});
```

3. Advanced Worker Patterns:

```
// Worker Pool for parallel processing
class WorkerPool {
    constructor(workerScript, poolSize = 4) {
        this.workers = [];
        this.taskQueue = [];
        this.activeWorkers = new Set();

        for (let i = 0; i < poolSize; i++) {
            const worker = new Worker(workerScript);
            worker.onmessage = (event) => this.handleWorker
            worker.onerror = (error) => this.handleWorkerEr
            this.workers.push(worker);
        }
    }
```

```
    execute(task) {
        return new Promise((resolve, reject) => {
            const workerTask = { task, resolve, reject };

            const availableWorker = this.workers.find(w =>

            if (availableWorker) {
                this.assignTask(availableWorker, workerTask
            } else {
                this.taskQueue.push(workerTask);
            }
        });
    }

    assignTask(worker, workerTask) {
        this.activeWorkers.add(worker);
        worker.currentTask = workerTask;
        worker.postMessage(workerTask.task);
    }

    handleWorkerMessage(worker, event) {
        const task = worker.currentTask;
        if (task) {
            task.resolve(event.data);
            this.activeWorkers.delete(worker);
            worker.currentTask = null;

            // Process next task in queue
            if (this.taskQueue.length > 0) {
                const nextTask = this.taskQueue.shift();
                this.assignTask(worker, nextTask);
            }
        }
    }

    handleWorkerError(worker, error) {
```

```
            const task = worker.currentTask;
            if (task) {
                task.reject(error);
                this.activeWorkers.delete(worker);
                worker.currentTask = null;
            }
        }

        terminate() {
            this.workers.forEach(worker => worker.terminate());
            this.workers = [];
            this.taskQueue = [];
            this.activeWorkers.clear();
        }
    }

    // Usage
    const pool = new WorkerPool('worker.js', 4);

    // Process multiple tasks in parallel
    const tasks = [
        { command: 'processData', data: [1,2,3] },
        { command: 'calculatePrimes', data: 1000 },
        { command: 'sortData', data: [5,2,8,1,9] }
    ];

    const results = await Promise.all(
        tasks.map(task => pool.execute(task))
    );
```

4. Transferable Objects:

```
    // Efficient large data transfer using Transferable Objects
    function processLargeDataInWorker(largeArray) {
        const worker = new Worker('processor.js');
```

```
    return new Promise((resolve, reject) => {
        worker.onmessage = (event) => {
            resolve(event.data);
            worker.terminate();
        };

        worker.onerror = reject;

        // Transfer ArrayBuffer instead of copying
        const buffer = new Float64Array(largeArray).buffer;
        worker.postMessage({ buffer }, [buffer]);
        // Note: buffer is now transferred, not accessible
    });
}

// processor.js
self.onmessage = function(event) {
    const { buffer } = event.data;
    const array = new Float64Array(buffer);

    // Process the data
    for (let i = 0; i < array.length; i++) {
        array[i] = array[i] * 2;
    }

    // Transfer back
    self.postMessage({ buffer }, [buffer]);
};
```

5. Real-World Examples:

```
// Image processing in worker
class ImageProcessor {
    constructor() {
        this.worker = new Worker('image-worker.js');
    }
```

```
    async processImage(imageData, filters) {
        return new Promise((resolve, reject) => {
            this.worker.onmessage = (event) => {
                resolve(event.data.imageData);
            };

            this.worker.onerror = reject;

            this.worker.postMessage({
                imageData,
                filters
            }, [imageData.data.buffer]);
        });
    }
}

// image-worker.js
self.onmessage = function(event) {
    const { imageData, filters } = event.data;

    // Apply filters
    if (filters.grayscale) {
        applyGrayscale(imageData);
    }

    if (filters.brightness) {
        applyBrightness(imageData, filters.brightness);
    }

    self.postMessage({ imageData }, [imageData.data.buffer]
};

function applyGrayscale(imageData) {
    const data = imageData.data;
    for (let i = 0; i < data.length; i += 4) {
        const avg = (data[i] + data[i + 1] + data[i + 2]) /
```

```
            data[i] = avg;      // R
            data[i + 1] = avg; // G
            data[i + 2] = avg; // B
        }
    }

    function applyBrightness(imageData, brightness) {
        const data = imageData.data;
        for (let i = 0; i < data.length; i += 4) {
            data[i] += brightness;      // R
            data[i + 1] += brightness; // G
            data[i + 2] += brightness; // B
        }
    }
```

Best Practices:

1. Use workers for CPU-intensive tasks only
2. Minimize data transfer between threads
3. Use Transferable Objects for large data
4. Implement proper error handling
5. Terminate workers when done
6. Use worker pools for multiple tasks
7. Avoid DOM manipulation in workers
8. Keep worker code modular
9. Monitor worker performance
10. Provide fallbacks for unsupported browsers

Related Questions: Performance Optimization, Fetch API, Service Workers

⬆️ Back to Top

---

76. What is the Geolocation API?

Geolocation API allows web applications to access the geographic location of a user's device. It requires user permission and provides latitude, longitude, altitude, and other location data.

1. Basic Geolocation:

```
// Get current position
function getCurrentLocation() {
    return new Promise((resolve, reject) => {
        // Check if geolocation is supported
        if (!navigator.geolocation) {
            reject(new Error('Geolocation is not supported
            return;
        }

        navigator.geolocation.getCurrentPosition(
            // Success callback
            (position) => {
                resolve({
                    latitude: position.coords.latitude,
                    longitude: position.coords.longitude,
                    accuracy: position.coords.accuracy,
                    altitude: position.coords.altitude,
                    altitudeAccuracy: position.coords.altit
                    heading: position.coords.heading,
                    speed: position.coords.speed,
                    timestamp: position.timestamp
                });
            },
            // Error callback
            (error) => {
                reject(error);
            },
            // Options
```

```
            {
                enableHighAccuracy: true,
                timeout: 10000,
                maximumAge: 300000 // 5 minutes
            }
        );
    });
}

// Usage
try {
    const location = await getCurrentLocation();
    console.log(`Latitude: ${location.latitude}`);
    console.log(`Longitude: ${location.longitude}`);
    console.log(`Accuracy: ${location.accuracy} meters`);
} catch (error) {
    console.error('Error getting location:', error.message)
}
```

2. Advanced Geolocation Manager:

```
class GeolocationManager {
    constructor() {
        this.watchId = null;
        this.isSupported = 'geolocation' in navigator;
    }

    // Check if geolocation is supported
    isGeolocationSupported() {
        return this.isSupported;
    }

    // Get current position once
    async getCurrentPosition(options = {}) {
        const defaultOptions = {
            enableHighAccuracy: true,
```

```
            timeout: 10000,
            maximumAge: 0
        };

        const finalOptions = { ...defaultOptions, ...option

        return new Promise((resolve, reject) => {
            if (!this.isSupported) {
                reject(new Error('Geolocation not supported
                return;
            }

            navigator.geolocation.getCurrentPosition(
                (position) => resolve(this.formatPosition(p
                (error) => reject(this.handleError(error)),
                finalOptions
            );
        });
    }

    // Watch position continuously
    watchPosition(onSuccess, onError, options = {}) {
        if (!this.isSupported) {
            onError(new Error('Geolocation not supported'))
            return null;
        }

        const defaultOptions = {
            enableHighAccuracy: true,
            timeout: 10000,
            maximumAge: 0
        };

        const finalOptions = { ...defaultOptions, ...option

        this.watchId = navigator.geolocation.watchPosition(
            (position) => onSuccess(this.formatPosition(pos
```

```
            (error) => onError(this.handleError(error)),
            finalOptions
        );

        return this.watchId;
    }

    // Stop watching position
    stopWatching() {
        if (this.watchId !== null) {
            navigator.geolocation.clearWatch(this.watchId);
            this.watchId = null;
        }
    }

    // Format position data
    formatPosition(position) {
        return {
            coords: {
                latitude: position.coords.latitude,
                longitude: position.coords.longitude,
                accuracy: position.coords.accuracy,
                altitude: position.coords.altitude,
                altitudeAccuracy: position.coords.altitudeA
                heading: position.coords.heading,
                speed: position.coords.speed
            },
            timestamp: position.timestamp,
            formattedTime: new Date(position.timestamp).toI
        };
    }

    // Handle geolocation errors
    handleError(error) {
        const errorMessages = {
            [error.PERMISSION_DENIED]: 'User denied geoloca
            [error.POSITION_UNAVAILABLE]: 'Location informa
```

```
            [error.TIMEOUT]: 'Location request timed out'
        };

        const message = errorMessages[error.code] || 'Unkno
        return new Error(message);
    }

    // Calculate distance between two points (Haversine for
    calculateDistance(lat1, lon1, lat2, lon2) {
        const R = 6371; // Earth's radius in kilometers
        const dLat = this.toRadians(lat2 - lat1);
        const dLon = this.toRadians(lon2 - lon1);

        const a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
                  Math.cos(this.toRadians(lat1)) * Math.cos
                  Math.sin(dLon / 2) * Math.sin(dLon / 2);

        const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1
        const distance = R * c;

        return distance; // Distance in kilometers
    }

    toRadians(degrees) {
        return degrees * (Math.PI / 180);
    }

    // Get formatted address from coordinates (requires rev
    async getAddressFromCoords(latitude, longitude) {
        try {
            const response = await fetch(
                `https://nominatim.openstreetmap.org/revers
            );
            const data = await response.json();
            return data.display_name;
        } catch (error) {
            throw new Error('Failed to get address from coo
```

```
        }
    }
}

// Usage examples
const geoManager = new GeolocationManager();

// Get current position once
try {
    const position = await geoManager.getCurrentPosition();
    console.log('Current position:', position);

    // Get address
    const address = await geoManager.getAddressFromCoords(
        position.coords.latitude,
        position.coords.longitude
    );
    console.log('Address:', address);
} catch (error) {
    console.error('Error:', error.message);
}

// Watch position continuously
const watchId = geoManager.watchPosition(
    (position) => {
        console.log('Position updated:', position);
    },
    (error) => {
        console.error('Watch error:', error.message);
    },
    {
        enableHighAccuracy: true,
        maximumAge: 30000, // 30 seconds
        timeout: 27000
    }
);
```

```
// Stop watching after 60 seconds
setTimeout(() => {
    geoManager.stopWatching();
    console.log('Stopped watching position');
}, 60000);
```

3. Real-World Applications:

```
// Location-based app features
class LocationApp {
    constructor() {
        this.geoManager = new GeolocationManager();
        this.userLocation = null;
    }

    // Initialize and get permission
    async initialize() {
        try {
            this.userLocation = await this.geoManager.getCu
            return true;
        } catch (error) {
            console.error('Failed to get location:', error)
            return false;
        }
    }

    // Find nearby places
    findNearbyPlaces(places) {
        if (!this.userLocation) {
            throw new Error('User location not available');
        }

        return places
            .map(place => ({
                ...place,
                distance: this.geoManager.calculateDistance
```

```
                    this.userLocation.coords.latitude,
                    this.userLocation.coords.longitude,
                    place.latitude,
                    place.longitude
                )
            }))
            .sort((a, b) => a.distance - b.distance);
    }

    // Track user movement
    startTracking(onLocationChange) {
        return this.geoManager.watchPosition(
            (position) => {
                const previousLocation = this.userLocation;
                this.userLocation = position;

                if (previousLocation) {
                    const distance = this.geoManager.calcul
                        previousLocation.coords.latitude,
                        previousLocation.coords.longitude,
                        position.coords.latitude,
                        position.coords.longitude
                    );

                    onLocationChange({
                        position,
                        distanceMoved: distance
                    });
                }
            },
            (error) => {
                console.error('Tracking error:', error);
            }
        );
    }

    // Geofencing - check if user is within area
```

```
    isWithinGeofence(centerLat, centerLon, radiusKm) {
        if (!this.userLocation) return false;

        const distance = this.geoManager.calculateDistance(
            this.userLocation.coords.latitude,
            this.userLocation.coords.longitude,
            centerLat,
            centerLon
        );

        return distance <= radiusKm;
    }
}

// Usage
const app = new LocationApp();

await app.initialize();

// Find nearby restaurants
const restaurants = [
    { name: 'Restaurant A', latitude: 40.7128, longitude: -
    { name: 'Restaurant B', latitude: 40.7589, longitude: -
    { name: 'Restaurant C', latitude: 40.7484, longitude: -
];

const nearbyRestaurants = app.findNearbyPlaces(restaurants)
console.log('Nearby restaurants:', nearbyRestaurants);

// Start tracking movement
app.startTracking((update) => {
    console.log('Location updated:', update.position);
    console.log('Distance moved:', update.distanceMoved, 'k
});

// Check geofence
```

```
const isNearOffice = app.isWithinGeofence(40.7128, -74.0060
console.log('Is near office:', isNearOffice);
```

Best Practices:

1. Always request permission explicitly
2. Provide clear explanation for location access
3. Handle permission denial gracefully
4. Use appropriate accuracy settings
5. Implement timeout handling
6. Cache location data when appropriate
7. Clear watchers when not needed
8. Handle errors comprehensively
9. Respect user privacy
10. Test on real devices

Related Questions: Fetch API, Web Workers, History API

Back to Top

---

77. What is the History API?

History API allows manipulation of the browser session history, enabling single-page applications (SPAs) to update the URL without full page reloads. It provides methods to navigate through history and modify the history stack.

1. Basic History Operations:

```
// Navigate through history
class HistoryManager {
    // Go back one page
```

```
    static goBack() {
        window.history.back();
    }

    // Go forward one page
    static goForward() {
        window.history.forward();
    }

    // Go to specific history entry
    static go(delta) {
        window.history.go(delta);
        // delta: -1 = back, 1 = forward, -2 = back twice,
    }

    // Get history length
    static getHistoryLength() {
        return window.history.length;
    }
}

// Usage
HistoryManager.goBack();     // Go to previous page
HistoryManager.goForward();  // Go to next page
HistoryManager.go(-2);       // Go back 2 pages
```

2. pushState and replaceState:

```
// Modern SPA routing with History API
class Router {
    constructor() {
        this.routes = new Map();
        this.currentPath = window.location.pathname;

        // Listen for popstate event (back/forward buttons)
        window.addEventListener('popstate', (event) => {
```

```
            this.handleRouteChange(window.location.pathname
        });
    }

    // Register a route
    addRoute(path, handler) {
        this.routes.set(path, handler);
    }

    // Navigate to a route (adds to history)
    navigateTo(path, state = {}) {
        if (this.currentPath === path) return;

        // Push new state to history
        window.history.pushState(
            state,
            '', // Title (ignored by most browsers)
            path
        );

        this.currentPath = path;
        this.handleRouteChange(path, state);
    }

    // Replace current history entry
    replaceTo(path, state = {}) {
        // Replace current state without adding new entry
        window.history.replaceState(
            state,
            '',
            path
        );

        this.currentPath = path;
        this.handleRouteChange(path, state);
    }
```

```
    // Handle route changes
    handleRouteChange(path, state) {
        const handler = this.routes.get(path);

        if (handler) {
            handler(state);
        } else {
            // Handle 404
            this.handle404();
        }

        // Update UI
        this.updateActiveLinks();
    }

    // Update active navigation links
    updateActiveLinks() {
        document.querySelectorAll('[data-route]').forEach(l
            const routePath = link.getAttribute('data-route
            if (routePath === this.currentPath) {
                link.classList.add('active');
            } else {
                link.classList.remove('active');
            }
        });
    }

    // Handle 404
    handle404() {
        console.error('Route not found:', this.currentPath)
        document.getElementById('app').innerHTML = '<h1>404
    }

    // Initialize router with click handlers
    initialize() {
        // Handle navigation link clicks
        document.addEventListener('click', (e) => {
```

```
            const link = e.target.closest('[data-route]');
            if (link) {
                e.preventDefault();
                const path = link.getAttribute('data-route'
                const state = link.dataset.state ? JSON.par
                this.navigateTo(path, state);
            }
        });
    }
}

// Usage
const router = new Router();

// Define routes
router.addRoute('/', (state) => {
    document.getElementById('app').innerHTML = '<h1>Home Pa
});

router.addRoute('/about', (state) => {
    document.getElementById('app').innerHTML = '<h1>About P
});

router.addRoute('/user/:id', (state) => {
    const userId = state.userId || 'unknown';
    document.getElementById('app').innerHTML = `<h1>User Pr
});

// Initialize router
router.initialize();

// Programmatic navigation
router.navigateTo('/about', { from: 'home' });
```

3. Advanced History Management:

```
// Comprehensive history manager with state management
class AdvancedHistoryManager {
    constructor() {
        this.listeners = [];
        this.setupPopStateListener();
    }

    // Setup popstate listener
    setupPopStateListener() {
        window.addEventListener('popstate', (event) => {
            this.notifyListeners({
                type: 'popstate',
                path: window.location.pathname,
                state: event.state
            });
        });
    }

    // Add history entry
    push(path, state = {}, title = '') {
        const fullState = {
            ...state,
            timestamp: Date.now(),
            scrollPosition: { x: window.scrollX, y: window.
        };

        window.history.pushState(fullState, title, path);

        this.notifyListeners({
            type: 'push',
            path,
            state: fullState
        });
    }

    // Replace current entry
```

```
    replace(path, state = {}, title = '') {
        const fullState = {
            ...state,
            timestamp: Date.now()
        };

        window.history.replaceState(fullState, title, path)

        this.notifyListeners({
            type: 'replace',
            path,
            state: fullState
        });
    }

    // Get current state
    getState() {
        return window.history.state;
    }

    // Subscribe to history changes
    subscribe(listener) {
        this.listeners.push(listener);

        // Return unsubscribe function
        return () => {
            const index = this.listeners.indexOf(listener);
            if (index > -1) {
                this.listeners.splice(index, 1);
            }
        };
    }

    // Notify all listeners
    notifyListeners(event) {
        this.listeners.forEach(listener => listener(event))
    }
```

```
    // Save and restore scroll position
    saveScrollPosition() {
        const state = this.getState() || {};
        this.replace(window.location.pathname, {
            ...state,
            scrollPosition: { x: window.scrollX, y: window.
        });
    }

    restoreScrollPosition() {
        const state = this.getState();
        if (state && state.scrollPosition) {
            window.scrollTo(state.scrollPosition.x, state.s
        }
    }

    // Confirmation before leaving
    setBeforeUnloadHandler(handler) {
        window.addEventListener('beforeunload', (e) => {
            if (handler()) {
                e.preventDefault();
                e.returnValue = ''; // Required for Chrome
            }
        });
    }
}

// Usage
const historyManager = new AdvancedHistoryManager();

// Subscribe to history changes
const unsubscribe = historyManager.subscribe((event) => {
    console.log('History changed:', event);

    if (event.type === 'popstate') {
        console.log('User clicked back/forward');
```

```
    }
});

// Navigate with state
historyManager.push('/products/123', {
    productId: 123,
    from: 'search',
    filters: { category: 'electronics' }
});

// Replace current entry
historyManager.replace('/products/123', {
    productId: 123,
    viewed: true
});

// Save scroll position before navigating
window.addEventListener('scroll', () => {
    historyManager.saveScrollPosition();
});

// Restore scroll position after navigation
historyManager.restoreScrollPosition();
```

4. Real-World SPA Router:

```
// Complete SPA router implementation
class SPARouter {
    constructor(options = {}) {
        this.routes = [];
        this.notFoundHandler = options.notFoundHandler || t
        this.beforeNavigate = options.beforeNavigate || nul
        this.afterNavigate = options.afterNavigate || null;

        this.init();
    }
```

```
    init() {
        // Handle popstate (browser back/forward)
        window.addEventListener('popstate', (event) => {
            this.handleRoute(window.location.pathname, even
        });

        // Handle link clicks
        document.addEventListener('click', (e) => {
            const link = e.target.closest('a[data-router-li
            if (link && link.hostname === window.location.h
                e.preventDefault();
                this.navigate(link.pathname, { trigger: 'cl
            }
        });

        // Handle initial route
        this.handleRoute(window.location.pathname, window.h
    }

    // Add route with pattern matching
    addRoute(pattern, handler) {
        const paramNames = [];
        const regexPattern = pattern.replace(/:([^/]+)/g, (
            paramNames.push(paramName);
            return '([^/]+)';
        });

        this.routes.push({
            pattern,
            regex: new RegExp(`^${regexPattern}$`),
            paramNames,
            handler
        });
    }

    // Navigate to path
```

```
    async navigate(path, state = {}) {
        // Call beforeNavigate hook
        if (this.beforeNavigate) {
            const shouldNavigate = await this.beforeNavigat
            if (shouldNavigate === false) return;
        }

        // Update history
        window.history.pushState(state, '', path);

        // Handle route
        await this.handleRoute(path, state);

        // Call afterNavigate hook
        if (this.afterNavigate) {
            this.afterNavigate(path, state);
        }
    }

    // Handle route matching
    async handleRoute(path, state) {
        for (const route of this.routes) {
            const match = path.match(route.regex);

            if (match) {
                const params = {};
                route.paramNames.forEach((name, index) => {
                    params[name] = match[index + 1];
                });

                await route.handler({ params, state, path }
                return;
            }
        }

        // No route matched
        this.notFoundHandler(path);
```

```
    }

    // Default 404 handler
    defaultNotFound(path) {
        document.getElementById('app').innerHTML = `
            <h1>404 - Page Not Found</h1>
            <p>The page "${path}" does not exist.</p>
        `;
    }
}

// Usage
const router = new SPARouter({
    beforeNavigate: async (path, state) => {
        // Check if user has unsaved changes
        if (hasUnsavedChanges()) {
            return confirm('You have unsaved changes. Do yo
        }
        return true;
    },
    afterNavigate: (path, state) => {
        // Track page view
        analytics.track('page_view', { path });
    }
});

// Define routes
router.addRoute('/', async ({ params, state, path }) => {
    document.getElementById('app').innerHTML = '<h1>Home</h
});

router.addRoute('/products', async ({ params, state, path }
    const products = await fetchProducts();
    renderProducts(products);
});

router.addRoute('/products/:id', async ({ params, state, pa
```

```
    const product = await fetchProduct(params.id);
    renderProduct(product);
});

router.addRoute('/user/:userId/posts/:postId', async ({ par
    const { userId, postId } = params;
    const post = await fetchUserPost(userId, postId);
    renderPost(post);
});

// Navigate programmatically
router.navigate('/products/123', { from: 'home' });
```

Best Practices:

1. Always use pushState/replaceState for SPAs
2. Handle popstate events properly
3. Update page title with each route
4. Implement proper error handling
5. Save scroll positions
6. Handle browser back/forward buttons
7. Implement route guards for authentication
8. Use meaningful state objects
9. Test with browser navigation
10. Provide fallback for older browsers

Related Questions: Geolocation API, Fetch API, Single Page Applications

⬆️ Back to Top

---

73. What is the Service Worker API?

Service Workers are scripts that run in the background, separate from web pages, enabling features like push notifications, background sync, and offline functionality.

1. Basic Service Worker:

```
// sw.js
self.addEventListener('install', event => {
    console.log('Service Worker installing');
    event.waitUntil(
        caches.open('v1').then(cache => {
            return cache.addAll([
                '/',
                '/index.html',
                '/styles.css',
                '/script.js'
            ]);
        })
    );
});

self.addEventListener('fetch', event => {
    event.respondWith(
        caches.match(event.request)
            .then(response => {
                if (response) {
                    return response;
                }
                return fetch(event.request);
            })
    );
});

// main.js
if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/sw.js')
        .then(registration => {
```

```
            console.log('SW registered: ', registration);
        })
        .catch(registrationError => {
            console.log('SW registration failed: ', registr
        });
}
```

2. Push Notifications:

```
// Request permission
async function requestNotificationPermission() {
    if (!('Notification' in window)) {
        console.log('This browser does not support notifica
        return false;
    }

    if (Notification.permission === 'granted') {
        return true;
    }

    if (Notification.permission !== 'denied') {
        const permission = await Notification.requestPermis
        return permission === 'granted';
    }

    return false;
}

// Show notification
async function showNotification(title, options = {}) {
    const hasPermission = await requestNotificationPermissi

    if (hasPermission) {
        const notification = new Notification(title, {
            body: options.body || 'You have a new notificat
            icon: options.icon || '/icon.png',
```

```
            badge: options.badge || '/badge.png',
            tag: options.tag || 'default'
        });

        notification.onclick = () => {
            window.focus();
            notification.close();
        };

        return notification;
    }
}
```

Best Practices:

1. Implement proper caching strategies
2. Handle offline scenarios
3. Use appropriate cache policies
4. Implement background sync
5. Handle push notifications
6. Test offline functionality
7. Monitor service worker performance
8. Implement proper error handling
9. Use appropriate lifecycle events
10. Consider user experience

Related Questions: Browser APIs, Performance

⬆️ Back to Top

---

74. What are Progressive Web Apps (PWAs)?

Progressive Web Apps are web applications that use modern web capabilities to deliver a native app-like experience to users.

1. PWA Features:

```
// Manifest file (manifest.json)
{
    "name": "My PWA",
    "short_name": "PWA",
    "description": "A Progressive Web App",
    "start_url": "/",
    "display": "standalone",
    "background_color": "#ffffff",
    "theme_color": "#000000",
    "icons": [
        {
            "src": "/icon-192.png",
            "sizes": "192x192",
            "type": "image/png"
        },
        {
            "src": "/icon-512.png",
            "sizes": "512x512",
            "type": "image/png"
        }
    ]
}

// Install prompt
let deferredPrompt;

window.addEventListener('beforeinstallprompt', (e) => {
    e.preventDefault();
    deferredPrompt = e;

    // Show install button
```

```
    const installButton = document.getElementById('install-
    installButton.style.display = 'block';

    installButton.addEventListener('click', async () => {
        if (deferredPrompt) {
            deferredPrompt.prompt();
            const { outcome } = await deferredPrompt.userCh
            console.log(`User response to the install promp
            deferredPrompt = null;
        }
    });
});
```

2. Offline Functionality:

```
// Service Worker for offline support
self.addEventListener('fetch', event => {
    event.respondWith(
        caches.match(event.request)
            .then(response => {
                if (response) {
                    return response;
                }

                return fetch(event.request).then(response =
                    if (!response || response.status !== 20
                        return response;
                    }

                    const responseToCache = response.clone(
                    caches.open('v1')
                        .then(cache => {
                            cache.put(event.request, respon
                        });

                    return response;
```

```
                    });
                })
        );
    });
```

Best Practices:

1. Implement proper manifest file
2. Use service workers for offline support
3. Implement install prompts
4. Use appropriate icons and themes
5. Test on different devices
6. Implement proper caching
7. Consider user experience
8. Use modern web APIs
9. Test offline functionality
10. Monitor performance metrics

Related Questions: Service Workers, Browser APIs

Back to Top

---

75. What is the Intersection Observer API?

Intersection Observer API provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with the top-level document's viewport.

1. Basic Intersection Observer:

```
// Create intersection observer
const observer = new IntersectionObserver((entries) => {
    entries.forEach(entry => {
        if (entry.isIntersecting) {
            entry.target.classList.add('visible');
            // Load content when element comes into view
            loadContent(entry.target);
        }
    });
}, {
    root: null, // Use viewport as root
    rootMargin: '0px',
    threshold: 0.1 // Trigger when 10% visible
});

// Observe elements
document.querySelectorAll('.lazy-load').forEach(el => {
    observer.observe(el);
});
```

2. Lazy Loading Images:

```
// Lazy loading images
function setupLazyLoading() {
    const imageObserver = new IntersectionObserver((entries
        entries.forEach(entry => {
            if (entry.isIntersecting) {
                const img = entry.target;
                img.src = img.dataset.src;
                img.classList.remove('lazy');
                imageObserver.unobserve(img);
            }
        });
    });
```

```
    document.querySelectorAll('img[data-src]').forEach(img
        imageObserver.observe(img);
    });
}
```

Best Practices:

1. Use appropriate thresholds
2. Implement proper cleanup
3. Consider performance implications
4. Use modern alternatives when available
5. Test across different browsers
6. Implement proper error handling
7. Consider user experience
8. Use appropriate root margins
9. Monitor performance metrics
10. Document usage patterns

Related Questions: Performance, Browser APIs

Back to Top

---

76. What are Web APIs and how do you use them?

Web APIs are interfaces provided by web browsers that allow JavaScript to interact with browser features and system resources. They enable web applications to access device capabilities, storage, and other browser functionality.

1. Geolocation API:

```
// Get current position
function getCurrentLocation() {
    return new Promise((resolve, reject) => {
        if (!navigator.geolocation) {
            reject(new Error('Geolocation is not supported'
            return;
        }

        navigator.geolocation.getCurrentPosition(
            (position) => {
                resolve({
                    latitude: position.coords.latitude,
                    longitude: position.coords.longitude,
                    accuracy: position.coords.accuracy
                });
            },
            (error) => {
                reject(error);
            },
            {
                enableHighAccuracy: true,
                timeout: 10000,
                maximumAge: 300000
            }
        );
    });
}
```

2. Notification API:

```
// Request permission
async function requestNotificationPermission() {
    if (!('Notification' in window)) {
        console.log('This browser does not support notifica
        return false;
```

```
    }

    if (Notification.permission === 'granted') {
        return true;
    }

    if (Notification.permission !== 'denied') {
        const permission = await Notification.requestPermis
        return permission === 'granted';
    }

    return false;
}
```

Best Practices:

1. Always check for API support
2. Handle errors gracefully
3. Use appropriate error messages
4. Consider fallbacks for older browsers
5. Implement proper cleanup
6. Use modern APIs when available
7. Test across different browsers
8. Consider performance implications
9. Follow security best practices
10. Document API usage

Related Questions: Browser APIs, Security

Back to Top

Security

81. What is Cross-Site Scripting (XSS)?

Cross-Site Scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. These scripts can steal sensitive information, manipulate page content, or perform actions on behalf of the user.

1. Types of XSS:

Reflected XSS:

```
// Vulnerable code - reflected XSS
function searchResults(query) {
    const results = performSearch(query);
    // Dangerous - directly inserting user input
    document.getElementById('results').innerHTML =
        `<h2>Search results for: ${query}</h2>`;
}

// Safe code - proper escaping
function searchResults(query) {
    const results = performSearch(query);
    const element = document.getElementById('results');
    element.innerHTML = `<h2>Search results for: ${escapeHt
}

function escapeHtml(text) {
    const div = document.createElement('div');
    div.textContent = text;
    return div.innerHTML;
}
```

Stored XSS:

```
// Vulnerable code - stored XSS
function saveComment(comment) {
    // Dangerous - storing raw HTML
    const commentData = {
        text: comment,
        timestamp: Date.now()
    };
    comments.push(commentData);
    displayComments();
}

// Safe code - sanitize before storing
function saveComment(comment) {
    const commentData = {
        text: sanitizeInput(comment),
        timestamp: Date.now()
    };
    comments.push(commentData);
    displayComments();
}

function sanitizeInput(input) {
    return input
        .replace(/</g, '&lt;')
        .replace(/>/g, '&gt;')
        .replace(/"/g, '&quot;')
        .replace(/'/g, '&#x27;')
        .replace(/\//g, '&#x2F;');
}
```

2. XSS Prevention Techniques:

Input Validation:

```
class InputValidator {
    static validateEmail(email) {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(email);
    }

    static validateUsername(username) {
        const usernameRegex = /^[a-zA-Z0-9_-]{3,20}$/;
        return usernameRegex.test(username);
    }

    static sanitizeHTML(html, allowedTags) {
        const div = document.createElement('div');
        div.innerHTML = html;

        const walker = document.createTreeWalker(
            div,
            NodeFilter.SHOW_ELEMENT,
            null,
            false
        );

        const elementsToRemove = [];
        let node;

        while (node = walker.nextNode()) {
            if (!allowedTags.includes(node.tagName.toLowerC
                elementsToRemove.push(node);
            }
        }

        elementsToRemove.forEach(el => el.remove());
        return div.innerHTML;
    }
}
```

Best Practices:

1. Always validate and sanitize user input
2. Use proper output encoding
3. Implement Content Security Policy
4. Use textContent instead of innerHTML
5. Validate data on both client and server
6. Use HTTPS everywhere
7. Implement proper session management
8. Regular security audits
9. Use security headers
10. Keep dependencies updated

Related Questions: CSRF, Input Sanitization

Back to Top

---

82. What is Cross-Site Request Forgery (CSRF)?

Cross-Site Request Forgery (CSRF) is an attack that tricks users into performing unwanted actions on a web application where they are authenticated. The attack exploits the trust that a site has in the user's browser.

1. CSRF Attack Examples:

Basic CSRF Attack:

```
<!-- Malicious website trying to perform CSRF attack -->
<form action="https://bank.com/transfer" method="POST" id="
    <input type="hidden" name="to" value="attacker-account"
    <input type="hidden" name="amount" value="1000">
</form>
```

```
<script>
    document.getElementById('csrf-form').submit();
</script>
```

2. CSRF Protection Techniques:

CSRF Tokens:

```
class CSRFProtection {
    static generateToken() {
        return crypto.randomUUID();
    }

    static setToken() {
        const token = this.generateToken();
        sessionStorage.setItem('csrf-token', token);
        return token;
    }

    static getToken() {
        return sessionStorage.getItem('csrf-token');
    }

    static validateToken(token) {
        return token === this.getToken();
    }

    static addTokenToForm(form) {
        const token = this.getToken();
        if (token) {
            const input = document.createElement('input');
            input.type = 'hidden';
            input.name = 'csrf-token';
            input.value = token;
            form.appendChild(input);
        }
```

```
        }
    }
```

SameSite Cookies:

```
// Set SameSite cookie attribute
function setSecureCookie(name, value, days) {
    const expires = new Date();
    expires.setTime(expires.getTime() + (days * 24 * 60 * 6

    document.cookie = `${name}=${value};expires=${expires.t
}
```

Best Practices:

1. Use CSRF tokens for state-changing operations
2. Implement SameSite cookie attributes
3. Validate Origin and Referer headers
4. Use custom headers for AJAX requests
5. Implement proper session management
6. Use HTTPS everywhere
7. Implement rate limiting
8. Validate tokens on both client and server
9. Use secure random token generation
10. Regular security testing

Related Questions: XSS, Input Sanitization

⬆️ Back to Top

---

83. How do you sanitize user input?

Input sanitization is the process of cleaning and validating user input to prevent security vulnerabilities like XSS, injection attacks, and other malicious activities.

1. HTML Sanitization:

Basic HTML Escaping:

```
class HTMLSanitizer {
    static escapeHtml(text) {
        const div = document.createElement('div');
        div.textContent = text;
        return div.innerHTML;
    }

    static sanitizeHTML(html, allowedTags = []) {
        const div = document.createElement('div');
        div.innerHTML = html;

        const walker = document.createTreeWalker(
            div,
            NodeFilter.SHOW_ELEMENT,
            null,
            false
        );

        const elementsToRemove = [];
        let node;

        while (node = walker.nextNode()) {
            if (!allowedTags.includes(node.tagName.toLowerC
                elementsToRemove.push(node);
            }
        }
```

```
        elementsToRemove.forEach(el => el.remove());
        return div.innerHTML;
    }
}
```

2. Input Validation:

Email Validation:

```
class InputValidator {
    static validateEmail(email) {
        const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        return emailRegex.test(email);
    }

    static validatePassword(password) {
        const errors = [];

        if (!password || password.length < 8) {
            errors.push('Password must be at least 8 charac
        }

        if (!/[A-Z]/.test(password)) {
            errors.push('Password must contain uppercase le
        }

        if (!/[a-z]/.test(password)) {
            errors.push('Password must contain lowercase le
        }

        if (!/\d/.test(password)) {
            errors.push('Password must contain number');
        }

        return {
            isValid: errors.length === 0,
```

```
                errors
            };
        }
    }
```

Best Practices:

1. Validate input on both client and server
2. Use whitelist approach instead of blacklist
3. Escape output based on context
4. Implement proper error handling
5. Use established libraries when possible
6. Regular security testing
7. Keep sanitization rules updated
8. Log suspicious input attempts
9. Implement rate limiting
10. Use Content Security Policy

Related Questions: XSS, CSRF

⬆️ Back to Top

---

84. What is Content Security Policy (CSP)?

Content Security Policy (CSP) is a security feature that helps prevent Cross-Site Scripting (XSS) attacks by allowing developers to specify which resources the browser should be allowed to load and execute.

1. Basic CSP Implementation:

Meta Tag CSP:

```
<!-- Basic CSP via meta tag -->
<meta http-equiv="Content-Security-Policy"
      content="default-src 'self'; script-src 'self' 'unsaf
```

HTTP Header CSP:

```
// Express.js CSP middleware
function setCSPHeaders(req, res, next) {
    const csp = [
        "default-src 'self'",
        "script-src 'self' 'unsafe-inline' https://trusted-
        "style-src 'self' 'unsafe-inline' https://fonts.goo
        "img-src 'self' data: https:",
        "font-src 'self' https://fonts.gstatic.com",
        "connect-src 'self' https://api.example.com",
        "object-src 'none'",
        "base-uri 'self'",
        "form-action 'self'",
        "frame-ancestors 'none'"
    ].join('; ');

    res.setHeader('Content-Security-Policy', csp);
    next();
}
```

2. Advanced CSP Features:

Nonce-based CSP:

```
class CSPNonce {
    static generateNonce() {
        return crypto.randomBytes(16).toString('base64');
    }
```

```
    static setNonceInCSP(nonce) {
        const csp = `script-src 'self' 'nonce-${nonce}'; st
        return csp;
    }

    static addNonceToScript(script, nonce) {
        script.setAttribute('nonce', nonce);
        return script;
    }
}

// Usage
const nonce = CSPNonce.generateNonce();
const script = document.createElement('script');
script.textContent = 'console.log("Safe inline script");';
CSPNonce.addNonceToScript(script, nonce);
document.head.appendChild(script);
```

Best Practices:

1. Start with strict policies and relax as needed
2. Use nonces or hashes for inline scripts/styles
3. Implement CSP reporting for monitoring
4. Test CSP policies thoroughly
5. Use report-only mode for testing
6. Keep policies up to date
7. Monitor violation reports
8. Use HTTPS with CSP
9. Implement proper fallbacks
10. Regular security audits

Related Questions: XSS, Input Sanitization

⬆️ Back to Top

Testing

85. How do you write unit tests in JavaScript?

Unit testing is a software testing method where individual units or components of a software application are tested in isolation. It helps ensure that each part of the code works correctly.

1. Basic Unit Testing Framework:

Simple Test Framework:

```
class TestFramework {
    constructor() {
        this.tests = [];
        this.beforeEach = null;
        this.afterEach = null;
        this.passed = 0;
        this.failed = 0;
    }

    describe(description, fn) {
        console.log(`\n📁 ${description}`);
        fn();
    }

    it(description, fn) {
        this.tests.push({ description, fn });
    }

    beforeEach(fn) {
        this.beforeEach = fn;
    }
```

```
    afterEach(fn) {
        this.afterEach = fn;
    }

    run() {
        this.tests.forEach(test => {
            try {
                if (this.beforeEach) this.beforeEach();
                test.fn();
                console.log(`  ✅ ${test.description}`);
                this.passed++;
            } catch (error) {
                console.log(`  ❌ ${test.description}: ${er
                this.failed++;
            } finally {
                if (this.afterEach) this.afterEach();
            }
        });

        console.log(`\n📊 Results: ${this.passed} passed, $
    }
}

// Usage
const testFramework = new TestFramework();

testFramework.describe('Math Utils', () => {
    let mathUtils;

    testFramework.beforeEach(() => {
        mathUtils = new MathUtils();
    });

    testFramework.it('should add two numbers', () => {
        assertEqual(mathUtils.add(2, 3), 5);
    });
```

```
    testFramework.it('should multiply two numbers', () => {
        assertEqual(mathUtils.multiply(4, 5), 20);
    });
});

testFramework.run();
```

2. Assertion Functions:

Basic Assertions:

```
// Assertion functions
function assert(condition, message) {
    if (!condition) {
        throw new Error(message || 'Assertion failed');
    }
}

function assertEqual(actual, expected, message) {
    if (actual !== expected) {
        throw new Error(message || `Expected ${expected}, b
    }
}

function assertNotEqual(actual, expected, message) {
    if (actual === expected) {
        throw new Error(message || `Expected not to be ${ex
    }
}

function assertTrue(actual, message) {
    if (actual !== true) {
        throw new Error(message || `Expected true, but got
    }
}
```

```
function assertFalse(actual, message) {
    if (actual !== false) {
        throw new Error(message || `Expected false, but got
    }
}

function assertThrows(fn, expectedError, message) {
    try {
        fn();
        throw new Error(message || 'Expected function to th
    } catch (error) {
        if (expectedError && !(error instanceof expectedErr
            throw new Error(message || `Expected ${expected
        }
    }
}
```

3. Testing Different Types of Functions:

Testing Pure Functions:

```
// Pure function to test
function calculateTax(amount, rate) {
    if (amount < 0) throw new Error('Amount cannot be negat
    if (rate < 0 || rate > 1) throw new Error('Rate must be
    return amount * rate;
}

// Tests for pure function
testFramework.describe('calculateTax', () => {
    testFramework.it('should calculate tax correctly', () =
        assertEqual(calculateTax(100, 0.1), 10);
        assertEqual(calculateTax(200, 0.15), 30);
        assertEqual(calculateTax(0, 0.1), 0);
    });
```

```
    testFramework.it('should throw error for negative amoun
        assertThrows(() => calculateTax(-100, 0.1), Error);
    });

    testFramework.it('should throw error for invalid rate',
        assertThrows(() => calculateTax(100, 1.5), Error);
        assertThrows(() => calculateTax(100, -0.1), Error);
    });
});
```

Best Practices:

1. Write tests before or alongside code
2. Test edge cases and error conditions
3. Keep tests simple and focused
4. Use descriptive test names
5. Mock external dependencies
6. Test async code properly
7. Maintain good test coverage
8. Keep tests fast and reliable
9. Refactor tests when refactoring code
10. Use appropriate assertions

Related Questions: TDD, Mocks and Stubs

Back to Top

---

86. What is test-driven development (TDD)?

Test-Driven Development (TDD) is a software development approach where you write tests before writing the actual code. It follows a cycle of Red-Green-Refactor

to ensure code quality and reliability.

1. TDD Cycle (Red-Green-Refactor):

Red Phase - Write Failing Test:

```
// Step 1: Write a failing test
describe('Calculator', () => {
    it('should add two numbers', () => {
        const calculator = new Calculator();
        const result = calculator.add(2, 3);
        assertEqual(result, 5);
    });
});

// This test will fail because Calculator doesn't exist yet
```

Green Phase - Write Minimal Code:

```
// Step 2: Write minimal code to make test pass
class Calculator {
    add(a, b) {
        return a + b;
    }
}

// Test now passes
```

Refactor Phase - Improve Code:

```
// Step 3: Refactor while keeping tests green
class Calculator {
    add(a, b) {
        if (typeof a !== 'number' || typeof b !== 'number')
```

```
            throw new Error('Both arguments must be numbers
        }
        return a + b;
    }
}

// Add more tests
describe('Calculator', () => {
    it('should add two numbers', () => {
        const calculator = new Calculator();
        const result = calculator.add(2, 3);
        assertEqual(result, 5);
    });

    it('should throw error for non-numeric inputs', () => {
        const calculator = new Calculator();
        assertThrows(() => calculator.add('2', 3), Error);
    });
});
```

2. TDD Example - Building a Stack:

Red Phase:

```
describe('Stack', () => {
    let stack;

    beforeEach(() => {
        stack = new Stack();
    });

    it('should be empty when created', () => {
        assertTrue(stack.isEmpty());
        assertEqual(stack.size(), 0);
    });
```

```
    it('should push elements', () => {
        stack.push(1);
        assertFalse(stack.isEmpty());
        assertEqual(stack.size(), 1);
    });

    it('should pop elements', () => {
        stack.push(1);
        stack.push(2);
        assertEqual(stack.pop(), 2);
        assertEqual(stack.size(), 1);
    });

    it('should throw error when popping from empty stack',
        assertThrows(() => stack.pop(), Error);
    });
});
```

Green Phase:

```
class Stack {
    constructor() {
        this.items = [];
    }

    isEmpty() {
        return this.items.length === 0;
    }

    size() {
        return this.items.length;
    }

    push(item) {
        this.items.push(item);
    }
```

```
    pop() {
        if (this.isEmpty()) {
            throw new Error('Cannot pop from empty stack');
        }
        return this.items.pop();
    }
}
```

Best Practices:

1. Write tests before writing code
2. Follow Red-Green-Refactor cycle
3. Write one test at a time
4. Keep tests simple and focused
5. Use descriptive test names
6. Test edge cases and error conditions
7. Refactor frequently
8. Maintain high test coverage
9. Keep tests fast
10. Use proper test structure (AAA pattern)

Related Questions: Unit Testing, Mocks and Stubs

Back to Top

---

87. What are mocks and stubs?

Mocks and stubs are testing techniques that replace real dependencies with fake implementations to isolate the code under test and control its behavior.

1. Stubs:

Basic Stub:

```
// Function to test
function processOrder(order, paymentService) {
    const payment = paymentService.processPayment(order.amo
    if (payment.success) {
        return { status: 'success', orderId: order.id };
    } else {
        return { status: 'failed', error: payment.error };
    }
}

// Stub for payment service
function createPaymentStub(shouldSucceed = true) {
    return {
        processPayment: (amount) => {
            if (shouldSucceed) {
                return { success: true, transactionId: 'txn
            } else {
                return { success: false, error: 'Payment fa
            }
        }
    };
}

// Tests using stub
describe('processOrder', () => {
    it('should process order successfully', () => {
        const order = { id: 1, amount: 100 };
        const paymentStub = createPaymentStub(true);

        const result = processOrder(order, paymentStub);

        assertEqual(result.status, 'success');
        assertEqual(result.orderId, 1);
    });
```

```
    it('should handle payment failure', () => {
        const order = { id: 1, amount: 100 };
        const paymentStub = createPaymentStub(false);

        const result = processOrder(order, paymentStub);

        assertEqual(result.status, 'failed');
        assertEqual(result.error, 'Payment failed');
    });
});
```

2. Mocks:

Basic Mock:

```
// Mock function that tracks calls
function createMock() {
    const calls = [];
    const mockFn = function(...args) {
        calls.push(args);
        return mockFn.returnValue;
    };

    mockFn.calls = calls;
    mockFn.returnValue = undefined;
    mockFn.mockReturnValue = (value) => {
        mockFn.returnValue = value;
        return mockFn;
    };

    mockFn.mockImplementation = (fn) => {
        mockFn.implementation = fn;
        return mockFn;
    };
```

```
    return mockFn;
}

// Usage
describe('UserService', () => {
    it('should call fetch with correct URL', () => {
        const mockFetch = createMock();
        mockFetch.mockReturnValue(Promise.resolve({
            ok: true,
            json: () => Promise.resolve({ id: 1, name: 'Joh
        }));

        global.fetch = mockFetch;

        const userService = new UserService();
        userService.getUser(1);

        assertEqual(mockFetch.calls.length, 1);
        assertEqual(mockFetch.calls[0][0], '/api/users/1');
    });
});
```

3. Spy Functions:

Function Spying:

```
function createSpy(originalFn) {
    const calls = [];

    const spy = function(...args) {
        calls.push(args);
        if (originalFn) {
            return originalFn.apply(this, args);
        }
    };
```

```
    spy.calls = calls;
    spy.callCount = () => calls.length;
    spy.calledWith = (...args) => {
        return calls.some(call =>
            call.length === args.length &&
            call.every((arg, index) => arg === args[index])
        );
    };

    return spy;
}

// Usage
describe('EventEmitter', () => {
    it('should call all listeners', () => {
        const emitter = new EventEmitter();
        const listener1 = createSpy();
        const listener2 = createSpy();

        emitter.on('test', listener1);
        emitter.on('test', listener2);
        emitter.emit('test', 'data');

        assertEqual(listener1.callCount(), 1);
        assertEqual(listener2.callCount(), 1);
        assertTrue(listener1.calledWith('data'));
        assertTrue(listener2.calledWith('data'));
    });
});
```

Best Practices:

1. Use mocks to isolate units under test
2. Mock external dependencies
3. Use stubs for simple replacements

4. Use spies to verify behavior
5. Keep mocks simple and focused
6. Reset mocks between tests
7. Use descriptive mock names
8. Mock at the right level of abstraction
9. Avoid over-mocking
10. Test mock behavior separately

Related Questions: Unit Testing, TDD

Back to Top

---

88. How do you test asynchronous code?

Testing asynchronous code requires special handling because async operations don't complete immediately. You need to wait for promises to resolve or reject, and handle timing issues.

1. Testing Promises:

Basic Promise Testing:

```
// Function to test
async function fetchUserData(userId) {
    if (!userId) throw new Error('User ID is required');

    const response = await fetch(`/api/users/${userId}`);
    if (!response.ok) throw new Error('Failed to fetch user

    return response.json();
}

// Tests for async function
```

```
describe('fetchUserData', () => {
    let mockFetch;

    beforeEach(() => {
        mockFetch = jest.fn();
        global.fetch = mockFetch;
    });

    it('should fetch user data successfully', async () => {
        // Arrange
        const mockUser = { id: 1, name: 'John' };
        mockFetch.mockResolvedValue({
            ok: true,
            json: () => Promise.resolve(mockUser)
        });

        // Act
        const result = await fetchUserData(1);

        // Assert
        assertEqual(result.id, 1);
        assertEqual(result.name, 'John');
        assertEqual(mockFetch).toHaveBeenCalledWith('/api/u
    });

    it('should throw error for missing user ID', async () =
        // Act & Assert
        try {
            await fetchUserData();
            assert(false, 'Expected function to throw');
        } catch (error) {
            assertEqual(error.message, 'User ID is required
        }
    });
});
```

2. Testing Callbacks:

Callback Testing:

```
// Function with callback
function processData(data, callback) {
    setTimeout(() => {
        if (data.error) {
            callback(new Error(data.error));
        } else {
            callback(null, data.result);
        }
    }, 100);
}

// Test with callback
describe('processData', () => {
    it('should call callback with result', (done) => {
        const testData = { result: 'success' };

        processData(testData, (error, result) => {
            try {
                assertEqual(error, null);
                assertEqual(result, 'success');
                done();
            } catch (err) {
                done(err);
            }
        });
    });

    it('should call callback with error', (done) => {
        const testData = { error: 'Something went wrong' };

        processData(testData, (error, result) => {
            try {
```

```
                assertNotEqual(error, null);
                assertEqual(error.message, 'Something went
                assertEqual(result, undefined);
                done();
            } catch (err) {
                done(err);
            }
        });
    });
});
```

3. Testing with Jest:

Jest Async Testing:

```
// Using Jest for async testing
describe('UserService', () => {
    let userService;
    let mockFetch;

    beforeEach(() => {
        userService = new UserService();
        mockFetch = jest.fn();
        global.fetch = mockFetch;
    });

    it('should fetch user data', async () => {
        // Arrange
        const mockUser = { id: 1, name: 'John' };
        mockFetch.mockResolvedValue({
            ok: true,
            json: () => Promise.resolve(mockUser)
        });

        // Act
        const result = await userService.getUser(1);
```

```
        // Assert
        expect(result).toEqual(mockUser);
        expect(mockFetch).toHaveBeenCalledWith('/api/users/
    });

    it('should handle fetch errors', async () => {
        // Arrange
        mockFetch.mockRejectedValue(new Error('Network erro

        // Act & Assert
        await expect(userService.getUser(1)).rejects.toThro
    });
});
```

4. Testing Timeouts and Delays:

Timeout Testing:

```
// Function with timeout
function fetchWithTimeout(url, timeout = 5000) {
    return new Promise((resolve, reject) => {
        const timer = setTimeout(() => {
            reject(new Error('Request timeout'));
        }, timeout);

        fetch(url)
            .then(response => {
                clearTimeout(timer);
                resolve(response);
            })
            .catch(error => {
                clearTimeout(timer);
                reject(error);
            });
    });
```

```
}

// Test with timeout
describe('fetchWithTimeout', () => {
    it('should resolve before timeout', async () => {
        const mockFetch = jest.fn().mockResolvedValue({ ok:
        global.fetch = mockFetch;

        const result = await fetchWithTimeout('/api/data',

        expect(result.ok).toBe(true);
    });

    it('should reject on timeout', async () => {
        const mockFetch = jest.fn().mockImplementation(() =
            new Promise(resolve => setTimeout(resolve, 2000
        );
        global.fetch = mockFetch;

        await expect(fetchWithTimeout('/api/data', 1000))
            .rejects.toThrow('Request timeout');
    });
});
```

Best Practices:

1. Use async/await for cleaner async tests
2. Always await async operations in tests
3. Use proper error handling in async tests
4. Mock external dependencies
5. Test both success and error cases
6. Use timeouts appropriately
7. Test event emitters properly

8. Use done callback for callback-based code
9. Avoid testing implementation details
10. Keep async tests focused and simple

Related Questions: Unit Testing, Mocks and Stubs

⬆️ Back to Top

⬆️ Back to Top

Modern JavaScript Tools

83. What is npm and package.json?

npm (Node Package Manager) is the default package manager for Node.js and the world's largest software registry. It allows developers to install, manage, and share JavaScript packages and dependencies.

1. What is npm:

Package Manager:

```
# Install a package globally
npm install -g create-react-app

# Install a package locally
npm install lodash

# Install as dev dependency
npm install --save-dev webpack

# Install specific version
npm install react@18.2.0
```

```
# Install latest version
npm install react@latest
```

2. package.json:

Basic Structure:

```
{
  "name": "my-javascript-project",
  "version": "1.0.0",
  "description": "A sample JavaScript project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "build": "webpack --mode production",
    "test": "jest",
    "lint": "eslint src/"
  },
  "keywords": ["javascript", "node", "web"],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2",
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "webpack": "^5.88.0",
    "jest": "^29.5.0",
    "eslint": "^8.42.0"
  },
  "engines": {
    "node": ">=14.0.0",
    "npm": ">=6.0.0"
```

```
  }
}
```

3. Scripts and Commands:

Common Scripts:

```
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "build": "webpack --mode production",
    "build:dev": "webpack --mode development",
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage",
    "lint": "eslint src/",
    "lint:fix": "eslint src/ --fix",
    "format": "prettier --write src/",
    "clean": "rimraf dist/",
    "prebuild": "npm run clean",
    "postbuild": "echo 'Build completed'"
  }
}
```

4. Dependency Management:

Semantic Versioning:

```
{
  "dependencies": {
    "react": "^18.2.0",     // Compatible with 18.2.0, <19.
    "lodash": "~4.17.21",   // Compatible with 4.17.21, <4.
    "express": "4.18.2",    // Exact version
    "axios": "*"             // Any version (not recommende
```

```
    }
}
```

Best Practices:

1. Use exact versions for critical dependencies
2. Keep devDependencies separate from dependencies
3. Use npm ci for production builds
4. Regularly update dependencies
5. Use .npmrc for configuration
6. Lock dependency versions with package-lock.json
7. Use semantic versioning
8. Document scripts clearly
9. Use engines field for Node.js version requirements
10. Keep package.json clean and organized

Related Questions: Build Tools, Babel

Back to Top

---

84. What are build tools like Webpack, Vite?

Build tools are essential for modern JavaScript development, helping bundle, transpile, optimize, and manage assets for production deployment.

1. Webpack:

Basic Configuration:

```
// webpack.config.js
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

```
const MiniCssExtractPlugin = require('mini-css-extract-plug

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.[contenthash].js',
    clean: true
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env']
          }
        }
      },
      {
        test: /\.css$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader']
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif)$/i,
        type: 'asset/resource'
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    }),
    new MiniCssExtractPlugin({
      filename: 'styles.[contenthash].css'
```

```
    })
  ],
  devServer: {
    static: './dist',
    hot: true,
    port: 3000
  }
};
```

2. Vite:

Basic Vite Configuration:

```
// vite.config.js
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
import { resolve } from 'path';

export default defineConfig({
  plugins: [react()],
  root: './src',
  build: {
    outDir: '../dist',
    rollupOptions: {
      input: {
        main: resolve(__dirname, 'src/index.html')
      }
    }
  },
  server: {
    port: 3000,
    open: true,
    cors: true
  },
  resolve: {
    alias: {
```

```
      '@': resolve(__dirname, 'src'),
      '@components': resolve(__dirname, 'src/components')
    }
  }
});
```

3. Build Tool Comparison:

Feature Comparison:

```
// Webpack - Most features, complex config
const webpackConfig = {
  // Extensive configuration options
  // Plugin ecosystem
  // Code splitting
  // Hot module replacement
  // Tree shaking
};

// Vite - Fast, modern, simple
const viteConfig = {
  // Fast dev server
  // ES modules in dev
  // Rollup for production
  // Built-in TypeScript support
};

// Rollup - Library focused
const rollupConfig = {
  // Tree shaking
  // ES modules output
  // Plugin system
  // Library bundling
};

// Parcel - Zero config
```

```
const parcelConfig = {
  // Zero configuration
  // Automatic asset handling
  // Built-in optimizations
  // Fast builds
};
```

Best Practices:

1. Choose the right tool for your project size
2. Use code splitting for large applications
3. Optimize bundle size with tree shaking
4. Use source maps for debugging
5. Implement caching strategies
6. Monitor bundle analysis
7. Use environment-specific configurations
8. Implement proper error handling
9. Use plugins judiciously
10. Keep build times fast

Related Questions: npm, Babel

Back to Top

---

85. What is Babel and why is it used?

Babel is a JavaScript compiler that transforms modern JavaScript code into a version that can run in older browsers or environments. It's essential for using cutting-edge JavaScript features while maintaining compatibility.

1. What is Babel:

Core Purpose:

```
// Modern JavaScript (ES2022+)
const user = {
  name: 'John',
  age: 30,
  greet() {
    return `Hello, I'm ${this.name}`;
  }
};

// Transformed to ES5
var user = {
  name: 'John',
  age: 30,
  greet: function greet() {
    return 'Hello, I\'m ' + this.name;
  }
};
```

2. Babel Configuration:

Basic Setup:

```
// package.json
{
  "devDependencies": {
    "@babel/core": "^7.22.0",
    "@babel/cli": "^7.22.0",
    "@babel/preset-env": "^7.22.0"
  },
  "scripts": {
    "build": "babel src -d lib",
    "watch": "babel src -d lib --watch"
```

```
  }
}
```

Babel Configuration File:

```
// babel.config.js
module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        targets: {
          browsers: ['> 1%', 'last 2 versions', 'not dead']
        },
        useBuiltIns: 'usage',
        corejs: 3
      }
    ]
  ],
  plugins: [
    '@babel/plugin-proposal-class-properties',
    '@babel/plugin-proposal-optional-chaining'
  ]
};
```

3. Transform Examples:

Arrow Functions:

```
// Input
const add = (a, b) => a + b;

// Output (ES5)
var add = function add(a, b) {
```

```
  return a + b;
};
```

Template Literals:

```
// Input
const name = 'John';
const message = `Hello, ${name}!`;

// Output (ES5)
var name = 'John';
var message = 'Hello, ' + name + '!';
```

4. Integration with Build Tools:

Webpack Integration:

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-env'],
            plugins: ['@babel/plugin-proposal-class-propert
          }
        }
      }
    ]
```

```
    }
};
```

Best Practices:

1. Use .browserslistrc for target configuration
2. Enable useBuiltIns for polyfills
3. Use @babel/plugin-transform-runtime for libraries
4. Configure environment-specific settings
5. Use babel-loader with webpack
6. Keep Babel updated
7. Use specific plugins instead of presets when possible
8. Configure source maps for debugging
9. Use babel-plugin-import for tree shaking
10. Test transformed code in target environments

Related Questions: Build Tools, ESLint

Back to Top

---

86. What are linters like ESLint?

Linters are tools that analyze code for potential errors, bugs, stylistic issues, and enforce coding standards. ESLint is the most popular JavaScript linter.

1. What is ESLint:

Core Purpose:

```
// Code with issues
var unused = 'not used';
if (true) {
```

```
  console.log('always true');
}
function badFunction() {
  return
    'this will return undefined';
}

// ESLint catches:
// - unused variables
// - unreachable code
// - missing semicolons
// - inconsistent formatting
```

2. ESLint Configuration:

Basic Configuration:

```
// .eslintrc.js
module.exports = {
  env: {
    browser: true,
    es2021: true,
    node: true
  },
  extends: [
    'eslint:recommended',
    '@typescript-eslint/recommended',
    'plugin:react/recommended'
  ],
  parser: '@typescript-eslint/parser',
  parserOptions: {
    ecmaVersion: 'latest',
    sourceType: 'module',
    ecmaFeatures: {
      jsx: true
    }
```

```
  },
  plugins: ['react', '@typescript-eslint'],
  rules: {
    'no-console': 'warn',
    'no-unused-vars': 'error',
    'prefer-const': 'error',
    'no-var': 'error'
  }
};
```

3. Common Rules:

Code Quality Rules:

```
// .eslintrc.js
module.exports = {
  rules: {
    // Possible Errors
    'no-console': 'warn',
    'no-debugger': 'error',
    'no-alert': 'error',
    'no-unused-vars': 'error',

    // Best Practices
    'eqeqeq': 'error',
    'no-eval': 'error',
    'no-implied-eval': 'error',
    'no-new-func': 'error',
    'prefer-const': 'error',
    'no-var': 'error',

    // Stylistic Issues
    'indent': ['error', 2],
    'quotes': ['error', 'single'],
    'semi': ['error', 'always'],
    'comma-dangle': ['error', 'never'],
```

```
    'no-trailing-spaces': 'error',

    // ES6+
    'arrow-spacing': 'error',
    'prefer-arrow-callback': 'error',
    'prefer-template': 'error',
    'template-curly-spacing': 'error'
  }
};
```

4. Scripts and Automation:

```
// package.json
{
  "scripts": {
    "lint": "eslint src/",
    "lint:fix": "eslint src/ --fix",
    "lint:report": "eslint src/ --format html -o lint-repor
    "lint:ci": "eslint src/ --format json -o lint-results.j
  }
}
```

Best Practices:

1. Use consistent configuration across team
2. Enable auto-fix for formatting issues
3. Use pre-commit hooks for linting
4. Configure IDE integration
5. Use specific rules for different file types
6. Regularly update ESLint and plugins
7. Use ignore files appropriately
8. Integrate with CI/CD pipeline

9. Use custom rules for project-specific needs
10. Combine with Prettier for formatting

Related Questions: Build Tools, Development vs Production

Back to Top

---

87. What is the difference between development and production builds?

Development and production builds serve different purposes in the software development lifecycle, with distinct optimizations, configurations, and features.

1. Development Builds:

Characteristics:

```
// webpack.config.js - Development
module.exports = {
  mode: 'development',
  devtool: 'eval-source-map',
  devServer: {
    hot: true,
    liveReload: true,
    port: 3000,
    open: true
  },
  optimization: {
    minimize: false,
    splitChunks: false
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoEmitOnErrorsPlugin()
```

```
  ]
};
```

2. Production Builds:

Characteristics:

```
// webpack.config.js - Production
module.exports = {
  mode: 'production',
  devtool: 'source-map',
  optimization: {
    minimize: true,
    splitChunks: {
      chunks: 'all',
      cacheGroups: {
        vendor: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          chunks: 'all'
        }
      }
    }
  },
  plugins: [
    new TerserPlugin({
      terserOptions: {
        compress: {
          drop_console: true,
          drop_debugger: true
        }
      }
    }),
    new CssMinimizerPlugin()
```

```
  ]
};
```

3. Key Differences:

Bundle Size:

```
// Development
// - Unminified code
// - Source maps included
// - Debug information
// - Larger bundle size

// Production
// - Minified code
// - Tree shaking
// - Dead code elimination
// - Smaller bundle size
```

Performance:

```
// Development
const config = {
  // Fast rebuild times
  // Hot module replacement
  // Source maps for debugging
  // Verbose error messages
};

// Production
const config = {
  // Optimized for runtime performance
  // Code splitting
  // Lazy loading
```

```
  // Caching strategies
};
```

4. Environment Variables:

```
// .env.development
NODE_ENV=development
API_URL=http://localhost:3000/api
DEBUG=true
LOG_LEVEL=debug

// .env.production
NODE_ENV=production
API_URL=https://api.myapp.com
DEBUG=false
LOG_LEVEL=error
```

5. Build Scripts:

```
// package.json
{
  "scripts": {
    "dev": "webpack serve --mode development",
    "build": "webpack --mode production",
    "build:dev": "webpack --mode development",
    "build:analyze": "webpack --mode production --analyze",
    "start": "node dist/server.js",
    "start:dev": "nodemon src/server.js"
  }
}
```

Best Practices:

1. Use different configurations for each environment

2. Enable source maps in development only
3. Implement proper error handling for production
4. Use environment variables for configuration
5. Optimize bundle size for production
6. Implement proper caching strategies
7. Remove debug code in production
8. Use code splitting for large applications
9. Monitor bundle size and performance
10. Test both development and production builds

Related Questions: Build Tools, ESLint

⬆️ Back to Top

⬆️ Back to Top

Regular Expressions

88. How do you use regular expressions in JavaScript?
89. What are common regex patterns for validation?
90. What's the difference between `match`, `search`, `replace`, and `test`?

⬆️ Back to Top

JSON and Data Handling

91. How do you work with JSON in JavaScript?
92. What's the difference between `JSON.parse()` and `JSON.stringify()`?
93. How do you handle API responses?

94. What is CORS and how do you handle it?

⬆️ Back to Top

Miscellaneous

95. What is the difference between synchronous and asynchronous code?
96. How do you check if a variable is an array?
97. What is the `typeof` operator and its limitations?
98. How do you convert strings to numbers and vice versa?
99. What is the difference between `slice`, `splice`, and `split`?
100. How do you sort arrays and objects?

⬆️ Back to Top

Coding Challenges (Common Interview Questions)

101. How do you reverse a string?
102. How do you check if a string is a palindrome?
103. How do you find duplicates in an array?
104. How do you flatten a nested array?
105. How do you implement a simple calculator?
106. How do you debounce a function?
107. How do you implement a simple Promise?
108. How do you find the largest/smallest number in an array?
109. How do you remove duplicates from an array?

110. How do you implement array methods like `map`, `filter`, `reduce` from scratch?

⬆️ Back to Top

Framework-Agnostic Frontend Concepts

111. What is the Virtual DOM?
112. What is component-based architecture?
113. What is state management?
114. What is client-side routing?
115. What is server-side rendering vs client-side rendering?
116. What are Progressive Web Apps (PWAs)?
117. What is responsive design and how do you implement it?
118. What is accessibility (a11y) and why is it important?
119. What are micro-frontends?
120. What is the difference between SPA, MPA, and SSG?

⬆️ Back to Top

This list covers the most frequently asked JavaScript questions in frontend interviews. Each topic builds upon fundamental JavaScript concepts and progresses to more advanced frontend development scenarios.