# Complete JavaScript Mastery Guide

A comprehensive JavaScript learning resource from basics to advanced concepts with real-world applications and interview preparation.

# About This Guide

This guide is designed for learners who want to master JavaScript from the ground up, with a focus on:

- Deep Understanding: Not just syntax, but how and why JavaScript works
- Real-World Skills: Practical applications you'll use in actual projects
- Interview Readiness: Common patterns and questions asked in technical interviews
- Modern Best Practices: ES6+ features and current industry standards

### ☐ Table of Contents

- Part I: Fundamentals Refresher
  - 1. Variables & Data Types
  - 2. Operators & Expressions
  - 3. Control Flow
  - 4. Functions Deep Dive
  - 5. Scope, Hoisting & TDZ
- Part II: Intermediate Concepts
  - 6. Closures Explained
  - 7. Arrays & Array Methods
  - 8. Objects & Object Access
  - 9. The 'this' Keyword
  - 10. Prototypes & Inheritance
  - 11. Classes & OOP in JavaScript
  - 12. Asynchronous JavaScript
- Part III: Advanced & Interview Topics
  - 13. JavaScript Engine & Compilation
  - 14. Proxy & Reflect
  - 15. Generators & Iterators
  - 16. Memory Management & Optimization
  - 17. Modules: ESM & CommonJS
  - 18. Error Handling & Debugging
  - 19. Deep Copy & Immutability
- Part IV: Professional JavaScript
  - 20. Interview Patterns
  - 21. Common Interview Questions

- 22. Writing Maintainable Code
- 23. JavaScript Quiz



24. Smart Task Manager Project

### How to Use This Guide

- 1. Sequential Learning: Follow chapters in order for structured learning
- 2. Reference Mode: Jump to specific topics as needed
- 3. Practice First: Try examples before reading explanations
- 4. Interview Prep: Focus on "Interview Notes" sections

# **Thapter Structure**

Each chapter includes:

- III Plain English Explanation
- **E** Code Examples with Comments
- A Common Pitfalls
- **@** When & Why to Use
- Mini Practice Problems
- | Interview Notes
- III Visual Aids & Diagrams

# **Y** Learning Outcomes

After completing this guide, you'll be able to:

- Understand JavaScript's core concepts deeply
- Write clean, efficient, and maintainable code
- Handle complex scenarios with confidence
- Ace JavaScript technical interviews
- Build real-world applications

### **S** Contributing

Found an error or want to improve something? Feel free to contribute!

### Happy Learning! Ø

"The best way to learn JavaScript is to understand it, not just memorize it."

# Chapter 1: Variables & Data Types

Understanding the foundation of JavaScript: how to store and work with different types of data.

### Plain English Explanation

Variables are like labeled boxes where you store information. Data types are the different kinds of information you can store - like numbers, text, true/false values, and more complex structures.

Think of it like organizing your desk:

- Variables = labeled drawers
- Data Types = what kind of stuff goes in each drawer (pens, papers, etc.)

# Variable Declarations

Modern Way (ES6+)

```
// const - for values that won't change
const userName = "Alice";
const PI = 3.14159;

// let - for values that might change
let age = 25;
let isLoggedIn = false;

// X Avoid var in modern JavaScript
var oldStyle = "Don't use this";
```

### **W** Key Differences

```
// const - Cannot be reassigned
const name = "John";
// name = "Jane"; // X TypeError!

// let - Can be reassigned
let score = 100;
score = 150; // Works fine

// var - Function scoped (causes issues)
if (true) {
  var x = 1;
  let y = 2;
}
console.log(x); // V 1 (var leaks out)
// console.log(y); // X ReferenceError (let is block-scoped)
```

# JavaScript Data Types

Primitive Types (7 types)

#### 1. Number

```
const integer = 42;
const decimal = 3.14;
const negative = -10;
const scientific = 2e3; // 2000
const infinity = Infinity;
const notANumber = NaN;

// Type checking
console.log(typeof 42); // "number"
console.log(Number.isInteger(42)); // true
console.log(Number.isNaN(NaN)); // true
```

#### 2. String

```
const singleQuotes = "Hello";
const doubleQuotes = "World";
const templateLiteral = `Hello ${singleQuotes}!`; // Template literals

// String methods
const text = "JavaScript";
console.log(text.length); // 10
console.log(text.toUpperCase()); // "JAVASCRIPT"
console.log(text.slice(0, 4)); // "Java"
```

#### 3. Boolean

```
const isTrue = true;
const isFalse = false;

// Truthy and Falsy values
console.log(Boolean(1)); // true
console.log(Boolean(0)); // false
console.log(Boolean("")); // false
console.log(Boolean("hello")); // true
```

#### 4. Undefined

```
let notAssigned;
console.log(notAssigned); // undefined
console.log(typeof notAssigned); // "undefined"
```

#### 5. Null

```
const empty = null;
console.log(empty); // null
console.log(typeof null); // "object" (this is a known bug!)
```

#### 6. Symbol (ES6)

```
const sym1 = Symbol("description");
const sym2 = Symbol("description");
console.log(sym1 === sym2); // false (always unique)
```

#### 7. BigInt (ES2020)

```
const bigNumber = 123456789012345678901234567890n;
const anotherBig = BigInt("123456789012345678901234567890");
console.log(typeof bigNumber); // "bigint"
```

Non-Primitive Types (Reference Types)

#### **Object**

```
const person = {
  name: "Alice",
  age: 30,
  isStudent: false,
};

// Arrays are objects too!
const numbers = [1, 2, 3, 4, 5];
console.log(typeof numbers); // "object"
console.log(Array.isArray(numbers)); // true

// Functions are objects too!
function greet() {
  return "Hello!";
}
console.log(typeof greet); // "function"
```

# Type Checking & Conversion

### **Checking Types**

```
const value = 42;

// typeof operator
console.log(typeof value); // "number"

// More specific checks
console.log(Array.isArray([])); // true
console.log(Number.isInteger(42)); // true
console.log(Object.prototype.toString.call(value)); // "[object Number]"
```

#### Type Conversion

```
// Explicit conversion
const str = "123";
const num = Number(str); // 123
const bool = Boolean(1); // true

// Implicit conversion (coercion)
console.log("5" + 3); // "53" (string concatenation)
console.log("5" - 3); // 2 (numeric subtraction)
console.log("5" * 3); // 15 (numeric multiplication)
```

### 

#### 1. typeof null Bug

```
console.log(typeof null); // "object" (not "null"!)

// Correct way to check for null

const value = null;

if (value === null) {
   console.log("It's null!");
}
```

#### 2. NaN Comparison

```
const result = 0 / 0; // NaN
console.log(result === NaN); // false (NaN is not equal to anything!)
console.log(Number.isNaN(result)); // true (correct way)
```

#### 3. Floating Point Precision

#### 4. Variable Hoisting with var

```
console.log(x); // undefined (not error!)
var x = 5;

// What actually happens:
// var x; // hoisted to top
// console.log(x); // undefined
// x = 5;
```

# **6** When & Why to Use

#### Use const by default

```
// For values that won't change
const API_URL = "https://api.example.com";
const user = { name: "Alice" }; // Object reference won't change
```

#### Use let when you need to reassign

```
// For counters, flags, etc.
let counter = 0;
for (let i = 0; i < 5; i++) {
   counter += i;
}</pre>
```

#### Choose appropriate data types

```
// Use numbers for calculations
const price = 29.99;
const quantity = 3;
const total = price * quantity;

// Use strings for text
const message = `Total: $${total.toFixed(2)}`;
```

```
// Use booleans for flags
const isDiscountApplied = total > 50;
```

# Mini Practice Problems

#### Problem 1: Type Detective

```
// What will these log?
console.log(typeof "42");
console.log(typeof 42);
console.log(typeof true);
console.log(typeof undefined);
console.log(typeof null);
console.log(typeof []);
console.log(typeof {});

// Try to guess before running!
```

#### Problem 2: Fix the Bug

```
// This code has issues. Can you fix them?
var name = "John";
if (true) {
  var name = "Jane";
}
console.log(name); // Should this be "John" or "Jane"?
// Rewrite using const/let appropriately
```

#### Problem 3: Type Conversion Challenge

```
// Predict the output
console.log("5" + 3 + 2);
console.log(3 + 2 + "5");
console.log("5" - 3);
console.log("5" * "3");
console.log("hello" - 3);
```

### Interview Notes

#### **Common Questions:**

#### Q: What's the difference between let, const, and var?

var: Function-scoped, hoisted, can be redeclared

- let: Block-scoped, hoisted but in TDZ, cannot be redeclared
- const: Block-scoped, hoisted but in TDZ, cannot be reassigned

#### Q: Why does typeof null return "object"?

It's a historical bug in JavaScript that can't be fixed due to backward compatibility

#### Q: What are falsy values in JavaScript?

• false, 0, -0, 0n, "", null, undefined, NaN

#### Q: How do you check if a variable is an array?

Use Array.isArray(variable), not typeof (which returns "object")

### **Asked at Companies:**

- Google: "Explain type coercion with examples"
- Facebook: "What happens when you add a string and a number?"
- Amazon: "How would you safely check if a variable is null?"

# **Ⅲ** Visual Memory Aid

```
JavaScript Data Types

├── Primitive (Stored by Value)

├── Number (42, 3.14, NaN, Infinity)

├── String ("hello", 'world', `template`)

├── Boolean (true, false)

├── Undefined (declared but not assigned)

├── Null (intentionally empty)

├── Symbol (unique identifier)

├── BigInt (large integers)

├── Non-Primitive (Stored by Reference)

├── Object (objects, arrays, functions)
```

# **@** Key Takeaways

- 1. Always use const by default, let when you need to reassign
- 2. JavaScript has 7 primitive types + objects
- 3. typeof has quirks know them for interviews
- 4. Type coercion can be tricky be explicit when possible
- 5. Understand the difference between primitive and reference types

**Next Chapter**: Operators & Expressions →

**Practice**: Try the problems above and experiment with different data types in your browser console!

# 4 Chapter 2: Operators & Expressions

Master JavaScript operators and learn how to build complex expressions that power your applications.

### Plain English Explanation

Operators are like tools that perform actions on your data. Think of them as:

- Arithmetic operators = calculator buttons (+, -, \*, /)
- **Comparison operators** = judges that compare things (>, <, ===)
- Logical operators = decision makers (&&, ||, !)
- Assignment operators = ways to store results (=, +=, -=)

Expressions are combinations of values, variables, and operators that produce a result.

### Arithmetic Operators

#### **Basic Math Operations**

```
const a = 10;
console.log(a + b); // 13 (Addition)
console.log(a - b); // 7 (Subtraction)
console.log(a * b); // 30 (Multiplication)
console.log(a / b); // 3.333... (Division)
console.log(a % b); // 1 (Modulus - remainder)
console.log(a ** b); // 1000 (Exponentiation - ES2016)
```

#### Increment & Decrement

```
let counter = 5;

// Pre-increment (increment first, then use)
console.log(++counter); // 6 (counter is now 6)

// Post-increment (use first, then increment)
console.log(counter++); // 6 (shows 6, but counter becomes 7)
console.log(counter); // 7

// Pre-decrement
console.log(--counter); // 6 (counter is now 6)

// Post-decrement
console.log(counter--); // 6 (shows 6, but counter becomes 5)
console.log(counter); // 5
```

#### Unary Plus & Minus

```
const str = "42";
const num = +str; // 42 (converts string to number)
const negative = -num; // -42

console.log(typeof str); // "string"
console.log(typeof num); // "number"
```

### **M** Comparison Operators

#### **Equality Operators**

```
// Strict equality (recommended)
console.log(5 === 5); // true
console.log(5 === "5"); // false (different types)
console.log(null === undefined); // false

// Loose equality (avoid in most cases)
console.log(5 == "5"); // true (type coercion happens)
console.log(null == undefined); // true
console.log(0 == false); // true
```

#### **Inequality Operators**

```
console.log(5 !== "5"); // true (strict inequality)
console.log(5 != "5"); // false (loose inequality)
```

#### **Relational Operators**

```
const x = 10;
const y = 5;

console.log(x > y); // true
console.log(x < y); // false
console.log(x >= 10); // true
console.log(y <= 5); // true

// String comparison (lexicographical)
console.log("apple" < "banana"); // true
console.log("10" < "9"); // true (string comparison!)</pre>
```

# Logical Operators

AND (&&) - All conditions must be true

```
const age = 25;
const hasLicense = true;

// Both conditions must be true
if (age >= 18 && hasLicense) {
   console.log("Can drive!");
}

// Short-circuit evaluation
const user = { name: "Alice" };
user.profile && console.log(user.profile.bio); // Won't error if profile is
undefined
```

#### OR (||) - At least one condition must be true

```
const isWeekend = false;
const isHoliday = true;

// Either condition can be true
if (isWeekend || isHoliday) {
   console.log("No work today!");
}

// Default values using ||
const username = user.name || "Guest";
const config = userConfig || defaultConfig;
```

#### NOT (!) - Inverts boolean value

```
const isLoggedIn = false;

if (!isLoggedIn) {
   console.log("Please log in");
}

// Double NOT for boolean conversion
   console.log(!!"hello"); // true
   console.log(!!0); // false
   console.log(!!""); // false
```

#### Nullish Coalescing (??) - ES2020

```
const value1 = null;
const value2 = undefined;
const value3 = 0;
const value4 = "";
```

2025-07-22 javascript.md

```
const fallback = "default";
// ?? only checks for null/undefined (not other falsy values)
console.log(value1 ?? fallback); // "default"
console.log(value2 ?? fallback); // "default"
console.log(value3 ?? fallback); // 0 (not "default"!)
console.log(value4 ?? fallback); // "" (not "default"!)
// Compare with ||
console.log(value3 || fallback); // "default"
console.log(value4 || fallback); // "default"
```

### **Assignment Operators**

#### **Basic Assignment**

```
let x = 10;
let y = x; // y gets the value of x
```

#### **Compound Assignment**

```
let score = 100;
score += 50; // score = score + 50 (150)
score -= 20; // score = score - 20 (130)
score *= 2; // score = score * 2 (260)
score /= 4; // score = score / 4 (65)
score %= 10; // score = score % 10 (5)
score **= 2; // score = score ** 2 (25)
// String concatenation
let message = "Hello";
message += " World"; // "Hello World"
```

#### Logical Assignment (ES2021)

```
let user = { name: "Alice" };
// Logical AND assignment
user.name &&= user.name.toUpperCase(); // Only if name exists
// Logical OR assignment
user.email ||= "default@example.com"; // Only if email is falsy
// Nullish coalescing assignment
user.age ??= 18; // Only if age is null/undefined
```

# 

#### **Basic Ternary**

```
const age = 20;
const status = age >= 18 ? "adult" : "minor";
console.log(status); // "adult"

// Equivalent to:
let status2;
if (age >= 18) {
    status2 = "adult";
} else {
    status2 = "minor";
}
```

#### Nested Ternary (use sparingly)

```
const score = 85;
const grade =
    score >= 90
    ? "A"
    : score >= 80
    ? "B"
    : score >= 70
    ? "C"
    : score >= 60
    ? "D"
    : "F";
console.log(grade); // "B"
```

#### **Ternary for Function Calls**

```
const isLoggedIn = true;
const action = isLoggedIn ? showDashboard() : showLoginForm();

// Or for conditional execution
isLoggedIn ? console.log("Welcome back!") : console.log("Please log in");
```

# **©** Operator Precedence & Associativity

Precedence (Order of Operations)

```
// Multiplication before addition
console.log(2 + 3 * 4); // 14 (not 20)

// Use parentheses to override
console.log((2 + 3) * 4); // 20

// Complex expression
const result = 10 + (5 * 2 ** 3) / 4 - 1;
// Order: 2**3 = 8, 5*8 = 40, 40/4 = 10, 10+10 = 20, 20-1 = 19
console.log(result); // 19
```

#### Associativity (Left-to-Right vs Right-to-Left)

```
// Left-to-right (most operators)
console.log(10 - 5 - 2); // 3 (not 7)
// Evaluated as: (10 - 5) - 2 = 5 - 2 = 3

// Right-to-left (assignment, exponentiation)
let a, b, c;
a = b = c = 5; // All get value 5
// Evaluated as: a = (b = (c = 5))

console.log(2 ** (3 ** 2)); // 512 (not 64)
// Evaluated as: 2 ** (3 ** 2) = 2 ** 9 = 512
```

### 

#### 1. Floating Point Arithmetic

#### 2. Type Coercion with ==

```
// Unexpected results with ==
console.log("" == 0); // true
console.log(false == 0); // true
console.log(null == 0); // false (but null == undefined is true)
```

```
console.log(" " == 0); // true

// Always use === for safety
console.log("" === 0); // false
console.log(false === 0); // false
```

#### 3. Increment/Decrement Confusion

```
let i = 5;
const a = i++; // a = 5, i = 6
const b = ++i; // i = 7, b = 7

console.log(a, b, i); // 5, 7, 7
```

### 4. Logical Operator Short-Circuiting

```
const user = null;

// This will throw an error!

// console.log(user.name && user.name.length);

// Correct way:
console.log(user && user.name && user.name.length);

// Or use optional chaining (ES2020)
console.log(user?.name?.length);
```

# **6** When & Why to Use

Use Strict Equality (===)

```
// Always prefer === over ==
if (userInput === "admin") {
   // Safe comparison
}
```

#### Use Logical Operators for Defaults

```
// Set default values
const theme = userPreference || "light";
const timeout = config.timeout ?? 5000;
```

```
// Good: Simple condition
const message = isOnline ? "Online" : "Offline";

// Bad: Complex logic (use if/else instead)
const result = condition1
    ? condition2
    ? value1
         : value2
         : condition3
         ? value3
         : value4;
```

# Mini Practice Problems

#### Problem 1: Operator Precedence

```
// What will these expressions evaluate to?
console.log(2 + 3 * 4);
console.log((2 + 3) * 4);
console.log(2 ** (3 ** 2));
console.log(10 - 5 - 2);
console.log(true + false);
console.log("5" + 3 + 2);
console.log(3 + 2 + "5");
```

#### Problem 2: Comparison Challenges

```
// Predict the output
console.log("10" > "9");
console.log("10" > 9);
console.log(null == undefined);
console.log(null === undefined);
console.log(NaN === NaN);
console.log(0 === -0);
```

#### **Problem 3: Logical Operators**

```
// What gets logged?
const a = "hello";
const b = "";
const c = null;
const d = 0;

console.log(a || b || c);
console.log(a && b && c);
```

```
console.log(c ?? d ?? a);
console.log(!a && !b);
console.log(!!a && !!b);
```

#### Problem 4: Build a Grade Calculator

```
// Create a function that takes a score and returns a grade
// Use ternary operators and comparison operators
// 90+ = A, 80-89 = B, 70-79 = C, 60-69 = D, <60 = F

function calculateGrade(score) {
    // Your code here
}

// Test cases
console.log(calculateGrade(95)); // "A"
console.log(calculateGrade(85)); // "B"
console.log(calculateGrade(75)); // "C"
console.log(calculateGrade(65)); // "D"
console.log(calculateGrade(55)); // "F"</pre>
```

### Interview Notes

#### **Common Questions:**

#### Q: What's the difference between == and ===?

- == performs type coercion, === checks both value and type
- Always use === unless you specifically need type coercion

#### Q: Explain operator precedence in JavaScript

- Operators have different priorities (\*, / before +, -)
- Use parentheses to make intentions clear
- · Assignment is right-to-left associative

#### Q: What is short-circuit evaluation?

- && stops at first falsy value
- || stops at first truthy value
- · Useful for conditional execution and default values

#### Q: What's the difference between | | and ???

- | checks for any falsy value (0, "", false, null, undefined)
- ?? only checks for null and undefined

### Asked at Companies:

Microsoft: "What does 0.1 + 0.2 === 0.3 return and why?"

- Netflix: "Explain the difference between ++i and i++"
- Spotify: "How would you safely access a nested property?"

### ☐ Operator Precedence Table (High to Low)

```
1. () [] .
2. ++ -- ! + -
                    // Grouping, Member access
                   // Unary operators
                    // Exponentiation
3. **
4. * / %
                    // Multiplication, Division, Modulus
5. + -
                    // Addition, Subtraction
6. < <= >>= // Addition, ...
7. == != === !== // Equality
8. &&
                     // Logical AND
9. ||
                    // Logical OR
10. ? :
                    // Conditional (ternary)
11. = += -= *= /= // Assignment
```

# **6** Key Takeaways

- 1. Always use === instead of == for comparisons
- 2. Understand operator precedence to avoid bugs
- 3. Use logical operators for defaults and short-circuiting
- 4. Be careful with floating-point arithmetic
- 5. Ternary operator is great for simple conditions
- 6. ?? is better than | | for null/undefined checks

**Previous Chapter**: ← Variables & Data Types

**Next Chapter**: Control Flow →

**Practice**: Try the problems above and experiment with different operator combinations!

# 

Master the art of making decisions and repeating actions in your JavaScript programs.

### Plain English Explanation

Control flow is how your program makes decisions and repeats actions. Think of it like:

- **Conditional statements** = traffic lights (if green, go; if red, stop)
- **Loops** = assembly lines (repeat the same action until done)
- **Switch statements** = elevator buttons (different action for each floor)

Your code doesn't just run top to bottom - it can branch, loop, and jump based on conditions!

### Conditional Statements

#### if Statement

```
const age = 18;

if (age >= 18) {
   console.log("You can vote!");
}

// Single line (no braces needed, but not recommended)
if (age >= 18) console.log("You can vote!");
```

#### if...else Statement

```
const temperature = 25;

if (temperature > 30) {
   console.log("It's hot outside!");
} else {
   console.log("It's not too hot.");
}
```

#### if...else if...else Chain

```
const score = 85;

if (score >= 90) {
    console.log("Grade: A");
} else if (score >= 80) {
    console.log("Grade: B");
} else if (score >= 70) {
    console.log("Grade: C");
} else if (score >= 60) {
    console.log("Grade: D");
} else {
    console.log("Grade: F");
}
```

#### **Nested if Statements**

```
const weather = "sunny";
const temperature = 25;

if (weather === "sunny") {
  if (temperature > 20) {
    console.log("Perfect day for a picnic!");
}
```

```
} else {
    console.log("Sunny but a bit cold.");
}
else {
    console.log("Maybe stay indoors.");
}
```

### Switch Statement

#### **Basic Switch**

```
const day = "monday";
switch (day) {
 case "monday":
    console.log("Start of the work week");
   break;
 case "tuesday":
    console.log("Tuesday blues");
    break;
 case "wednesday":
    console.log("Hump day!");
    break;
 case "thursday":
    console.log("Almost there");
    break;
 case "friday":
   console.log("TGIF!");
    break;
 case "saturday":
  case "sunday":
    console.log("Weekend!");
    break;
 default:
    console.log("Invalid day");
}
```

#### Switch with Fall-through

```
const month = "february";
let days;

switch (month) {
  case "january":
   case "march":
   case "may":
   case "july":
   case "august":
   case "october":
```

```
case "december":
   days = 31;
   break;
 case "april":
 case "june":
 case "september":
 case "november":
   days = 30;
   break;
 case "february":
   days = 28; // Simplified (not accounting for leap years)
   break;
 default:
   days = ∅;
   console.log("Invalid month");
}
console.log(`${month} has ${days} days`);
```

#### Switch with Expressions (Modern Pattern)

```
const getSeasonMessage = (month) => {
 switch (month) {
   case "december":
   case "january":
   case "february":
      return "Winter is here! 🛞 ";
    case "march":
    case "april":
   case "may":
      return "Spring has arrived! @#";
   case "june":
   case "july":
   case "august":
      return "Summer time! 📛 ";
   case "september":
   case "october":
    case "november":
      return "Autumn leaves! 🙈";
   default:
      return "Invalid month";
 }
};
console.log(getSeasonMessage("july")); // "Summer time! ₡"
```

# Loops

for Loop

```
// Basic for loop
for (let i = 0; i < 5; i++) {
 console.log(`Iteration ${i}`);
}
// Counting backwards
for (let i = 10; i >= 0; i--) {
 console.log(`Countdown: ${i}`);
}
// Custom increment
for (let i = 0; i <= 20; i += 2) {
 console.log(`Even number: ${i}`);
}
// Multiple variables
for (let i = 0, j = 10; i < 5; i++, j--) {
 console.log(`i: ${i}, j: ${j}`);
}
```

#### while Loop

```
let count = 0;
while (count < 5) {
   console.log(`Count is: ${count}`);
   count++;
}

// Reading user input (conceptual)
let userInput = "";
while (userInput !== "quit") {
   // userInput = prompt("Enter command (or 'quit' to exit):");
   console.log(`You entered: ${userInput}`);
   break; // Prevent infinite loop in this example
}</pre>
```

#### do...while Loop

```
let number;

do {
    // This runs at least once
    number = Math.floor(Math.random() * 10) + 1;
    console.log(`Generated number: ${number}`);
} while (number !== 7);

console.log("Found lucky number 7!");
```

#### for...in Loop (Objects)

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
};

// Iterate over object properties
for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}

// With arrays (not recommended - use for...of instead)
  const colors = ["red", "green", "blue"];
  for (const index in colors) {
    console.log(`Index ${index}: ${colors[index]}`);
}
```

#### for...of Loop (Iterables)

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

// Iterate over array values
for (const fruit of fruits) {
    console.log(`I like ${fruit}`);
}

// With strings
const word = "hello";
for (const letter of word) {
    console.log(letter.toUpperCase());
}

// With index using entries()
for (const [index, fruit] of fruits.entries()) {
    console.log(`${index}: ${fruit}`);
}
```

# Loop Control Statements

#### break Statement

```
// Exit loop early
for (let i = 0; i < 10; i++) {
  if (i === 5) {</pre>
```

```
break; // Exit the loop when i equals 5
}
console.log(i); // Prints 0, 1, 2, 3, 4
}

// Break from nested loops
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) {
      break outer; // Break from outer loop
    }
    console.log(`i: ${i}, j: ${j}`);
}</pre>
```

#### continue Statement

```
// Skip current iteration
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue; // Skip even numbers
    }
    console.log(i); // Prints 1, 3, 5, 7, 9
}

// Continue with labeled loops
outer: for (let i = 0; i < 3; i++) {
    for (let j = 0; j < 3; j++) {
        if (j === 1) {
            continue outer; // Continue outer loop
        }
        console.log(`i: ${i}, j: ${j}`);
    }
}</pre>
```

### **6** Modern Control Flow Patterns

#### Array Methods for Control Flow

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Instead of for loop with if
const evenNumbers = numbers.filter((num) => num % 2 === 0);
console.log(evenNumbers); // [2, 4, 6, 8, 10]

// Instead of for loop with transformation
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

```
// Instead of for loop with accumulation
const sum = numbers.reduce((acc, num) => acc + num, 0);
console.log(sum); // 55

// Check conditions
const hasEven = numbers.some((num) => num % 2 === 0);
const allPositive = numbers.every((num) => num > 0);
console.log(hasEven, allPositive); // true, true
```

#### Optional Chaining & Nullish Coalescing

```
const user = {
 profile: {
    social: {
      twitter: "@alice",
    },
 },
};
// Old way (verbose)
if (
 user &&
 user.profile &&
 user.profile.social &&
 user.profile.social.twitter
  console.log(user.profile.social.twitter);
}
// Modern way (ES2020)
console.log(user?.profile?.social?.twitter ?? "No Twitter");
// With function calls
user?.getName?.(); // Only calls if getName exists
```

#### Guard Clauses (Early Returns)

```
// Instead of nested if statements
function processUser(user) {
    // Guard clauses
    if (!user) {
        console.log("No user provided");
        return;
    }

    if (!user.email) {
        console.log("User has no email");
        return;
    }
}
```

```
if (!user.isActive) {
    console.log("User is not active");
    return;
}

// Main logic (not nested)
console.log(`Processing user: ${user.email}`);
// ... rest of the function
}
```

### 

#### 1. Missing break in Switch

```
const grade = "B";

switch (grade) {
  case "A":
    console.log("Excellent!");

// Missing break - falls through!
  case "B":
    console.log("Good job!");
    break;
  case "C":
    console.log("Not bad");
    break;
}

// If grade is "A", both "Excellent!" and "Good job!" will print
```

### 2. Infinite Loops

```
// X Infinite loop - forgot to increment
let i = 0;
while (i < 5) {
   console.log(i);
   // i++; // Forgot this!
}

// X Wrong condition
for (let i = 0; i > -5; i++) {
   // Should be i < 5
   console.log(i);
}</pre>
```

#### 3. Off-by-One Errors

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

// X Goes beyond array length
for (let i = 0; i <= arr.length; i++) {
   console.log(arr[i]); // arr[5] is undefined!
}

// Correct
for (let i = 0; i < arr.length; i++) {
   console.log(arr[i]);
}</pre>
```

#### 4. Variable Scope in Loops

```
// X Common mistake with var
for (var i = 0; i < 3; i++) {
    setTimeout(() => {
        console.log(i); // Prints 3, 3, 3
      }, 100);
}

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

# **@** When & Why to Use

#### Choose the Right Loop

```
// Use for...of for arrays when you need values
const items = ["a", "b", "c"];
for (const item of items) {
   console.log(item);
}

// Use for...in for objects when you need keys
const obj = { x: 1, y: 2 };
for (const key in obj) {
   console.log(`${key}: ${obj[key]}`);
}

// Use traditional for when you need index control
for (let i = 0; i < items.length; i++) {
   console.log(`${i}: ${items[i]}`);
}</pre>
```

```
// Use while when you don't know iteration count
while (condition) {
   // Do something until condition becomes false
}
```

#### Choose if vs Switch

```
// Use if for complex conditions
if (age >= 18 && hasLicense && !isSuspended) {
   allowDriving();
}

// Use switch for multiple discrete values
switch (userRole) {
   case "admin":
      showAdminPanel();
      break;
   case "user":
      showUserDashboard();
      break;
   case "guest":
      showPublicContent();
      break;
}
```

# Mini Practice Problems

#### Problem 1: FizzBuzz Classic

```
// Print numbers 1-100, but:
// - "Fizz" for multiples of 3
// - "Buzz" for multiples of 5
// - "FizzBuzz" for multiples of both

function fizzBuzz() {
   // Your code here
}

fizzBuzz();
```

#### Problem 2: Grade Calculator

```
// Convert numeric scores to letter grades
// Use both if/else and switch approaches
```

```
function getGrade(score) {
    // Your code here using if/else
}

function getGradeSwitch(score) {
    // Your code here using switch
    // Hint: Use Math.floor(score/10)
}

console.log(getGrade(95)); // "A"
console.log(getGrade(85)); // "B"
console.log(getGrade(75)); // "C"
```

#### **Problem 3: Pattern Printing**

```
// Print this pattern:
// *
// **
// ***
// ****

function printTriangle(height) {
    // Your code here
}

printTriangle(5);
```

#### **Problem 4: Find Prime Numbers**

```
// Find all prime numbers up to n
function findPrimes(n) {
  const primes = [];
  // Your code here
  return primes;
}

console.log(findPrimes(20)); // [2, 3, 5, 7, 11, 13, 17, 19]
```

#### Problem 5: Nested Loop Challenge

```
// Create a multiplication table
function multiplicationTable(size) {
   // Print a size x size multiplication table
   // Example for size 3:
   // 1 2 3
```

```
// 2 4 6
// 3 6 9
}
multiplicationTable(5);
```

## Interview Notes

#### **Common Questions:**

#### Q: What's the difference between for...in and for...of?

- for...in iterates over enumerable properties (keys)
- for...of iterates over iterable values
- Use for...in for objects, for...of for arrays

#### Q: When would you use a while loop vs a for loop?

- for loop: when you know the number of iterations
- while loop: when you iterate until a condition is met

#### Q: What happens if you forget break in a switch statement?

- Fall-through behavior execution continues to next case
- Sometimes intentional, but usually a bug

#### Q: How do you break out of nested loops?

- Use labeled statements with break labelName
- Or use a function and return
- Or use a flag variable

#### **Asked at Companies:**

- Google: "Implement FizzBuzz without using if statements"
- Amazon: "Find the first duplicate in an array using loops"
- Facebook: "Explain the difference between break and continue"
- Microsoft: "How would you avoid infinite loops?"

### Control Flow Decision Tree

```
Need to make decisions?

├── Simple condition → if/else
├── Multiple discrete values → switch
└── Complex conditions → if/else if/else

Need to repeat actions?
├── Known iterations → for loop
├── Unknown iterations → while loop
├── At least once → do...while
```

```
Array values → for...of
Object properties → for...in
```

# **@** Key Takeaways

- 1. Use === in conditions, not ==
- 2. Always include break in switch cases (unless fall-through is intended)
- 3. Prefer for...of for arrays, for...in for objects
- 4. Use guard clauses to reduce nesting
- 5. Consider array methods instead of manual loops
- 6. Be careful with loop conditions to avoid infinite loops
- 7. Use let instead of var in loops

**Previous Chapter**: ← Operators & Expressions

Next Chapter: Functions Deep Dive →

**Practice**: Try the FizzBuzz problem and experiment with different loop types!



# Chapter 4: Functions Deep Dive

Master JavaScript functions: the building blocks of reusable, modular code.

### Plain English Explanation

Functions are like recipes or machines:

- **Input** = ingredients (parameters)
- **Process** = cooking steps (function body)
- **Output** = finished dish (return value)

They let you write code once and use it many times, making your programs organized and efficient.



### Function Declaration

**Basic Function Declaration** 

```
// Function declaration - hoisted to top
function greet(name) {
  return `Hello, ${name}!`;
}
// Can be called before declaration due to hoisting
console.log(sayHi("Alice")); // Works!
function sayHi(name) {
  return `Hi there, ${name}!`;
```

#### **Function with Multiple Parameters**

```
function calculateArea(length, width) {
  return length * width;
}
const area = calculateArea(10, 5);
console.log(area); // 50
```

#### Function with Default Parameters (ES6)

```
function greetUser(name = "Guest", greeting = "Hello") {
 return `${greeting}, ${name}!`;
}
console.log(greetUser()); // "Hello, Guest!"
console.log(greetUser("Alice")); // "Hello, Alice!"
console.log(greetUser("Bob", "Hi")); // "Hi, Bob!"
```

#### Rest Parameters (...args)

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3)); // 6
console.log(sum(1, 2, 3, 4, 5)); // 15
// Mix regular params with rest
function introduce(firstName, lastName, ...hobbies) {
  console.log(`I'm ${firstName} ${lastName}`);
  console.log(`My hobbies: ${hobbies.join(", ")}`);
}
introduce("John", "Doe", "reading", "coding", "gaming");
```

### Function Expressions

#### **Anonymous Function Expression**

```
// Function expression - not hoisted
const multiply = function (a, b) {
 return a * b;
};
```

```
console.log(multiply(4, 5)); // 20

// This would cause an error:
   // console.log(divide(10, 2)); // ReferenceError!

const divide = function (a, b) {
   return a / b;
};
```

#### Named Function Expression

```
const factorial = function fact(n) {
  if (n <= 1) return 1;
  return n * fact(n - 1); // Can reference itself by name
  };

console.log(factorial(5)); // 120
  // console.log(fact(5)); // ReferenceError - name only available inside</pre>
```

#### Immediately Invoked Function Expression (IIFE)

```
// IIFE - runs immediately
(function () {
    console.log("This runs right away!");
})();

// IIFE with parameters
(function (name) {
    console.log(`Hello, ${name}!`);
})("World");

// IIFE with return value
const result = (function (a, b) {
    return a + b;
})(5, 3);

console.log(result); // 8
```

# Arrow Functions (ES6)

#### **Basic Arrow Function Syntax**

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

```
// Arrow function - concise
const addArrow = (a, b) => a + b;

// Both work the same
console.log(add(2, 3)); // 5
console.log(addArrow(2, 3)); // 5
```

#### **Arrow Function Variations**

```
// No parameters
const sayHello = () => "Hello!";

// One parameter (parentheses optional)
const square = (x) => x * x;
const squareExplicit = (x) => x * x; // Same thing

// Multiple parameters
const multiply = (a, b) => a * b;

// Block body (need explicit return)
const complexFunction = (x, y) => {
   const sum = x + y;
   const product = x * y;
   return { sum, product };
};

console.log(complexFunction(3, 4)); // { sum: 7, product: 12 }
```

#### Arrow Functions with Arrays

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

// Traditional way
const doubled = numbers.map(function (num) {
    return num * 2;
});

// Arrow function way
const doubledArrow = numbers.map((num) => num * 2);
const evens = numbers.filter((num) => num % 2 === 0);
const sum = numbers.reduce((acc, num) => acc + num, 0);

console.log(doubledArrow); // [2, 4, 6, 8, 10]
console.log(evens); // [2, 4]
console.log(sum); // 15
```

# Sunction Scope & Closures Preview

#### Local vs Global Scope

```
const globalVar = "I'm global";

function testScope() {
   const localVar = "I'm local";
   console.log(globalVar); // Can access global
   console.log(localVar); // Can access local
}

testScope();
console.log(globalVar); // Works
// console.log(localVar); // ReferenceError!
```

#### Function Parameters Shadow Global Variables

```
const name = "Global Alice";

function greet(name) {
   console.log(`Hello, ${name}!`); // Uses parameter, not global
}

greet("Local Bob"); // "Hello, Local Bob!"
   console.log(name); // Still "Global Alice"
```

# Advanced Function Concepts

#### **Functions as First-Class Citizens**

```
// Functions can be assigned to variables
const myFunc = function () {
   return "Hello!";
};

// Functions can be passed as arguments
function executeFunction(fn) {
   return fn();
}

console.log(executeFunction(myFunc)); // "Hello!"

// Functions can be returned from other functions
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
```

### **Higher-Order Functions**

```
// Function that takes other functions as arguments
function processArray(arr, callback) {
  const result = [];
  for (let i = 0; i < arr.length; i++) {
    result.push(callback(arr[i], i));
  }
  return result;
}

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

// Different callbacks for different behaviors
  const squared = processArray(numbers, (x) => x * x);
  const withIndex = processArray(numbers, (x, i) => `${i}: ${x}`);

console.log(squared); // [1, 4, 9, 16, 25]
  console.log(withIndex); // ["0: 1", "1: 2", "2: 3", "3: 4", "4: 5"]
```

#### Callback Functions

```
// Simulating asynchronous operation
function fetchData(callback) {
   setTimeout(() => {
      const data = { id: 1, name: "Alice" };
      callback(data);
   }, 1000);
}

// Using the callback
fetchData(function (data) {
   console.log("Received data:", data);
});

// Arrow function callback
fetchData((data) => console.log("Data:", data));
```

## call() Method

```
function introduce() {
    return `Hi, I'm ${this.name} and I'm ${this.age} years old.`;
}

const person1 = { name: "Alice", age: 25 };
const person2 = { name: "Bob", age: 30 };

// Call function with different 'this' context
console.log(introduce.call(person1)); // "Hi, I'm Alice and I'm 25 years old."
console.log(introduce.call(person2)); // "Hi, I'm Bob and I'm 30 years old."

// call() with arguments
function greetWithTitle(title) {
    return `${title} ${this.name}`;
}

console.log(greetWithTitle.call(person1, "Dr.")); // "Dr. Alice"
```

### apply() Method

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

const numbers = [1, 2, 3];

// apply() takes array of arguments
   console.log(sum.apply(null, numbers)); // 6

// Equivalent to:
   console.log(sum.call(null, 1, 2, 3)); // 6

// Modern way with spread operator
   console.log(sum(...numbers)); // 6
```

#### bind() Method

```
const person = {
  name: "Alice",
  greet: function (greeting) {
    return `${greeting}, I'm ${this.name}!`;
  },
};

// bind() creates new function with fixed 'this'
const boundGreet = person.greet.bind(person);
```

```
console.log(boundGreet("Hello")); // "Hello, I'm Alice!"

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

# 

#### 1. Arrow Functions and 'this'

```
const obj = {
  name: "Alice",

// Regular function - 'this' refers to obj
  regularMethod: function () {
    console.log(this.name); // "Alice"
  },

// Arrow function - 'this' refers to outer scope
  arrowMethod: () => {
    console.log(this.name); // undefined (or global)
  },
};

obj.regularMethod(); // "Alice"
  obj.arrowMethod(); // undefined
```

## 2. Hoisting Differences

```
// Function declarations are fully hoisted
console.log(declared()); // "Works!"

function declared() {
   return "Works!";
}

// Function expressions are not hoisted
// console.log(expressed()); // TypeError!

const expressed = function () {
   return "Works!";
};
```

### 3. Modifying Parameters

```
// Primitive parameters are passed by value
function changeNumber(num) {
   num = 100;
}

let x = 5;
changeNumber(x);
console.log(x); // Still 5

// Object parameters are passed by reference
function changeObject(obj) {
   obj.name = "Changed";
}

const person = { name: "Alice" };
changeObject(person);
console.log(person.name); // "Changed"
```

### 4. Closure Memory Leaks

```
// Potential memory leak
function createHandler() {
  const largeData = new Array(1000000).fill("data");
 return function () {
   // Even if we don't use largeData, it's kept in memory
    console.log("Handler called");
 };
}
// Better: only close over what you need
function createHandlerBetter() {
  const largeData = new Array(1000000).fill("data");
  const needed = largeData.length; // Extract only what's needed
  return function () {
    console.log(`Handler called, data size: ${needed}`);
  };
}
```

# **@** When & Why to Use

#### Function Declaration vs Expression vs Arrow

```
// Use function declarations for main functions
function calculateTax(amount, rate) {
  return amount * rate;
}
```

```
// Use function expressions when assigning conditionally
const operation = isAddition
  ? function (a, b) {
      return a + b;
  : function (a, b) {
     return a - b;
    };
// Use arrow functions for short callbacks
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((n) \Rightarrow n * 2);
const evens = numbers.filter((n) => n % 2 === 0);
// Use regular functions when you need 'this' binding
const button = {
 text: "Click me",
  click: function () {
    console.log(this.text); // Need 'this' to refer to button
 },
};
```

#### When to Use Each Pattern

```
// Pure functions (no side effects) - preferred
function add(a, b) {
   return a + b; // Only depends on inputs
}

// Functions with side effects - be careful
function logAndAdd(a, b) {
   console.log(`Adding ${a} + ${b}`); // Side effect
   return a + b;
}

// Higher-order functions for reusability
function createValidator(rule) {
   return function (value) {
     return rule(value);
   };
}

const isEmail = createValidator((val) => val.includes("@"));
const isLongEnough = createValidator((val) => val.length >= 8);
```

# Mini Practice Problems

#### **Problem 1: Function Variations**

```
// Create the same function using all three methods:
// 1. Function declaration
// 2. Function expression
// 3. Arrow function
// Function should calculate compound interest
// Formula: A = P(1 + r/n)^{nt}
// P = principal, r = rate, n = compounds per year, t = time
// Your implementations here:
function calculateInterest1(principal, rate, compounds, time) {
  // Function declaration
}
const calculateInterest2 = function (principal, rate, compounds, time) {
 // Function expression
};
const calculateInterest3 = (principal, rate, compounds, time) => {
  // Arrow function
};
```

#### Problem 2: Higher-Order Function

```
// Create a function that takes an array and a callback
// and returns a new array with the callback applied to each element
// but only if the element passes a test function

function mapIf(array, testFn, mapFn) {
    // Your code here
}

// Test it:
const numbers = [1, 2, 3, 4, 5, 6];
const result = mapIf(
    numbers,
    (n) => n % 2 === 0, // test: only even numbers
    (n) => n * n // map: square them
);
console.log(result); // Should output: [4, 16, 36]
```

#### **Problem 3: Function Factory**

```
// Create a function that returns customized greeting functions

function createGreeter(greeting, punctuation = "!") {
   // Return a function that greets with the given style
}
```

```
// Test it:
const casualGreet = createGreeter("Hey");
const formalGreet = createGreeter("Good morning", ".");
const excitedGreet = createGreeter("Wow", "!!!");

console.log(casualGreet("Alice")); // "Hey, Alice!"
console.log(formalGreet("Mr. Smith")); // "Good morning, Mr. Smith."
console.log(excitedGreet("everyone")); // "Wow, everyone!!!"
```

#### Problem 4: Callback Pattern

```
// Create a function that processes an array with different strategies

function processNumbers(numbers, strategy) {
    // Your code here
}

// Test with different strategies:
const nums = [1, 2, 3, 4, 5];

const sum = processNumbers(nums, (acc, curr) => acc + curr);
const product = processNumbers(nums, (acc, curr) => acc * curr);
const max = processNumbers(nums, (acc, curr) => Math.max(acc, curr));

console.log(sum, product, max); // 15, 120, 5
```

# Interview Notes

#### **Common Questions:**

#### Q: What's the difference between function declaration and function expression?

- Function declarations are hoisted completely
- Function expressions are not hoisted
- Function declarations create a named function in the current scope

### Q: When should you use arrow functions vs regular functions?

- Arrow functions: short callbacks, when you don't need this binding
- Regular functions: methods, constructors, when you need this context

#### Q: What is a closure?

- A function that has access to variables from its outer scope
- Even after the outer function has returned
- Creates a "persistent scope"

#### Q: Explain call(), apply(), and bind()

- call(): invokes function with specific this and individual arguments
- apply(): invokes function with specific this and array of arguments
- bind(): creates new function with specific this (doesn't invoke)

### **Asked at Companies:**

- Google: "Implement a function that can be called with any number of arguments"
- **Facebook**: "What's the difference between function and =>?"
- Amazon: "Create a function that remembers its previous calls"
- Netflix: "Explain function hoisting with examples"

# **Ⅲ** Function Types Summary

```
Function Types

Declaration

Hoisted

Named

Block scoped (in strict mode)

Expression

Not hoisted X

Can be anonymous

Can be assigned conditionally

Arrow

Concise syntax

No 'this' binding X

No 'arguments' object X

Cannot be constructor X
```

# **©** Key Takeaways

- 1. Use function declarations for main functions (hoisting benefit)
- 2. Use arrow functions for short callbacks and array methods
- 3. Regular functions when you need this context
- 4. Functions are first-class citizens can be passed around
- 5. Understand the difference between call, apply, and bind
- 6. Be careful with arrow functions and this binding
- 7. Use default parameters instead of checking for undefined

Previous Chapter: ← Control Flow
Next Chapter: Scope, Hoisting & TDZ →

Practice: Try creating functions using all three methods and experiment with callbacks!

# Q

# Chapter 5: Scope, Hoisting & TDZ

Understand how JavaScript manages variable access, memory, and execution context.

# Plain English Explanation

Think of scope like rooms in a house:

- **Global scope** = the main living area (everyone can access)
- Function scope = private bedrooms (only family members can enter)
- **Block scope** = closets (only the person in that room can access)

Hoisting is like JavaScript "reading ahead" - it sees all your variable declarations before running your code, but not their values.

# **1** Understanding Scope

#### Global Scope

```
// Global scope - accessible everywhere
const globalVar = "I'm global!";
var alsoGlobal = "Me too!";

function testGlobal() {
   console.log(globalVar); // ✓ Can access global variables
   console.log(alsoGlobal); // ✓ Can access global variables
}

testGlobal();
console.log(globalVar); // ✓ Accessible outside functions

// Be careful - implicit globals (avoid!)
function createImplicitGlobal() {
   implicitGlobal = "Oops, I'm global!"; // No var/let/const = global!
}

createImplicitGlobal();
console.log(implicitGlobal); // "Oops, I'm global!"
```

#### **Function Scope**

```
outerFunction();
// console.log(outerVar); // X ReferenceError!
```

### Block Scope (ES6+)

```
// let and const are block-scoped
if (true) {
 let blockScoped = "I'm block scoped";
 const alsoBlockScoped = "Me too!";
 var functionScoped = "I'm function scoped";
 console.log(blockScoped); //  Works inside block
}
// console.log(blockScoped); // X ReferenceError!
// console.log(alsoBlockScoped); // X ReferenceError!
console.log(functionScoped); //  var leaks out of blocks!
// Same with loops
for (let i = 0; i < 3; i++) {
 // i is only accessible here
}
// console.log(i); // X ReferenceError!
// But var leaks:
for (var j = 0; j < 3; j++) {
 // j will be accessible outside
}
console.log(j); //  3 (leaked out!)
```

### Lexical Scope (Static Scope)

```
const globalName = "Global Alice";

function outer() {
  const outerName = "Outer Bob";

function inner() {
   const innerName = "Inner Charlie";

  // Scope chain: inner → outer → global
   console.log(innerName); // "Inner Charlie"
   console.log(outerName); // "Outer Bob"
   console.log(globalName); // "Global Alice"
}

return inner;
```

```
const innerFunc = outer();
innerFunc(); // Still has access to outer scope!
```

# **Q** Hoisting Explained

### Variable Hoisting with var

```
// What you write:
console.log(myVar); // undefined (not error!)
var myVar = "Hello";
console.log(myVar); // "Hello"

// What JavaScript actually does:
// var myVar; // Declaration hoisted to top
// console.log(myVar); // undefined
// myVar = "Hello"; // Assignment stays in place
// console.log(myVar); // "Hello"
```

### **Function Hoisting**

```
// Function declarations are fully hoisted
console.log(hoistedFunction()); // "I'm hoisted!"

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

// Function expressions are NOT hoisted
// console.log(notHoisted()); // TypeError!

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

#### let and const Hoisting (Temporal Dead Zone)

```
// let and const are hoisted but in "Temporal Dead Zone"
console.log(typeof myVar); // "undefined"
console.log(typeof myLet); // ReferenceError!

var myVar = "var variable";
let myLet = "let variable";
// Example of TDZ
```

```
function demonstrateTDZ() {
   console.log(x); // ReferenceError! (TDZ)

let x = "I'm in TDZ until this line";
   console.log(x); // "I'm in TDZ until this line"
}
```

# ← Temporal Dead Zone (TDZ)

#### What is TDZ?

```
// TDZ exists from start of scope until declaration
function showTDZ() {
    // TDZ starts here for 'temporal'

    console.log(typeof temporal); // ReferenceError!

    // Still in TDZ
    if (true) {
        console.log(typeof temporal); // ReferenceError!

        let temporal = "Now I exist!"; // TDZ ends here
        console.log(temporal); // "Now I exist!"
    }
}

showTDZ();
```

#### TDZ with const

```
// const must be initialized when declared
// const uninitializedConst; // SyntaxError!

const properConst = "I'm properly initialized";

// const cannot be reassigned
// properConst = "New value"; // TypeError!
```

### TDZ in Loops

```
// Classic var problem
for (var i = 0; i < 3; i++) {
   setTimeout(() => {
      console.log("var:", i); // 3, 3, 3 (all reference same i)
   }, 100);
}
```

```
// let creates new binding each iteration
for (let j = 0; j < 3; j++) {
   setTimeout(() => {
      console.log("let:", j); // 0, 1, 2 (each has own j)
   }, 100);
}
```

# Scope Chain

## How Scope Chain Works

```
const global = "global";
function level1() {
  const level1Var = "level1";
 function level2() {
   const level2Var = "level2";
   function level3() {
      const level3Var = "level3";
      // Scope chain: level3 → level2 → level1 → global
      console.log(level3Var); // Found in level3
      console.log(level2Var); // Found in level2
      console.log(level1Var); // Found in level1
      console.log(global); // Found in global
      // console.log(nonExistent); // ReferenceError!
    }
   level3();
 level2();
}
level1();
```

### Variable Shadowing

```
const name = "Global";

function outer() {
  const name = "Outer";

function inner() {
   const name = "Inner";
   console.log(name); // "Inner" (shadows outer scopes)
```

```
inner();
console.log(name); // "Outer" (shadows global)
}

outer();
console.log(name); // "Global"
```

# **©** Practical Examples

### Module Pattern with Scope

```
// Using IIFE to create private scope
const calculator = (function () {
  // Private variables
  let result = 0;
  // Private function
  function log(operation, value) {
   console.log(`${operation}: ${value}, result: ${result}`);
  }
  // Public API
  return {
    add(value) {
      result += value;
      log("add", value);
      return this; // For chaining
    },
    subtract(value) {
      result -= value;
      log("subtract", value);
      return this;
    },
    getResult() {
      return result;
    },
    reset() {
      result = 0;
      log("reset", 0);
      return this;
    },
  };
})();
// Usage
calculator.add(10).subtract(3).add(5);
```

```
console.log(calculator.getResult()); // 12

// Private variables are not accessible

// console.log(calculator.result); // undefined
```

### Closure with Scope

```
function createCounter(initialValue = 0) {
  let count = initialValue;
  return {
    increment() {
      return ++count;
    },
    decrement() {
      return --count;
    },
    getValue() {
      return count;
    },
    reset() {
      count = initialValue;
      return count;
    },
 };
const counter1 = createCounter(5);
const counter2 = createCounter(10);
console.log(counter1.increment()); // 6
console.log(counter2.increment()); // 11
console.log(counter1.getValue()); // 6
console.log(counter2.getValue()); // 11
```

# 

#### 1. var in Loops

```
// Problem: All callbacks reference same variable
for (var i = 0; i < 3; i++) {
   setTimeout(function () {
      console.log("var i:", i); // 3, 3, 3
   }, 100);
}</pre>
```

```
// Solution 1: Use let
for (let i = 0; i < 3; i++) {
   setTimeout(function () {
      console.log("let i:", i); // 0, 1, 2
    }, 100);
}

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

### 2. Hoisting Confusion

```
// Confusing hoisting behavior
var x = 1;
function confusing() {
  console.log(x); // undefined (not 1!)
 if (false) {
   var x = 2; // This declaration is hoisted!
}
confusing();
// What actually happens:
function confusing() {
 var x; // Hoisted declaration
 console.log(x); // undefined
 if (false) {
    x = 2; // Assignment (never executed)
  }
}
```

#### 3. TDZ Gotchas

```
// TDZ with default parameters
function tdz(a = b, b = 2) {
  return a + b;
}

// tdz(); // ReferenceError! b is in TDZ when a is initialized
```

```
// Correct order
function fixed(b = 2, a = b) {
  return a + b;
}

console.log(fixed()); // 4
```

#### 4. Block Scope Confusion

```
// Mixing let and var
function mixedScope() {
   if (true) {
      var varVariable = "var";
      let letVariable = "let";
   }

   console.log(varVariable); // "var" (function scoped)
   // console.log(letVariable); // ReferenceError! (block scoped)
}

mixedScope();
```

# **6** When & Why to Use

#### Choose the Right Declaration

```
// Use const by default
const API_URL = "https://api.example.com";
const users = []; // Reference won't change

// Use let when you need to reassign
let currentUser = null;
let counter = 0;

// Avoid var in modern JavaScript
// var oldStyle = "Don't use this";
```

### **Scope Best Practices**

```
// Keep variables in smallest possible scope
function processData(data) {
   // Don't declare everything at the top

if (data.length === 0) {
   return [];
```

```
}

// Declare variables when needed
const processed = [];

for (let i = 0; i < data.length; i++) {
   const item = data[i]; // Block scoped

   if (item.isValid) {
      const transformed = transform(item); // Even smaller scope
      processed.push(transformed);
    }
}

return processed;
}</pre>
```

# Mini Practice Problems

### Problem 1: Hoisting Quiz

```
// Predict the output of each console.log

console.log(a); // ?
console.log(b); // ?

// console.log(c); // ?

var a = 1;
let b = 2;
const c = 3;

function hoistingTest() {
   console.log(a); // ?
   console.log(b); // ?

   var a = 10;
   let b = 20;
}

hoistingTest();
```

## Problem 2: Scope Chain Challenge

```
// What will this output?
const x = "global";

function outer() {
  const x = "outer";
```

```
function inner() {
    console.log(x); // ?
}

return inner;
}

const fn = outer();
fn();
```

### Problem 3: Loop Scope Fix

```
// Fix this code so it prints 0, 1, 2 instead of 3, 3, 3
for (var i = 0; i < 3; i++) {
   setTimeout(function () {
      console.log(i);
   }, 100);
}
// Provide at least 2 different solutions</pre>
```

### Problem 4: Create a Private Counter

```
// Create a function that returns an object with methods
// to increment, decrement, and get the current count
// The count should be private (not accessible from outside)

function createPrivateCounter(start = 0) {
    // Your code here
}

const counter = createPrivateCounter(5);
console.log(counter.increment()); // 6
console.log(counter.increment()); // 7
console.log(counter.decrement()); // 6
console.log(counter.getCount()); // 6
// console.log(counter.count); // Should be undefined
```

#### Problem 5: TDZ Understanding

```
// Explain why this code throws an error and how to fix it

function example() {
  console.log(typeof myVar); // Works
  console.log(typeof myLet); // ReferenceError - why?
```

```
var myVar = "var";
let myLet = "let";
}
example();
```

# Interview Notes

#### **Common Questions:**

#### Q: What is hoisting?

- JavaScript moves variable and function declarations to the top of their scope
- Only declarations are hoisted, not initializations
- var is hoisted and initialized with undefined
- let/const are hoisted but in Temporal Dead Zone

#### Q: What is the Temporal Dead Zone?

- Period between entering scope and variable declaration
- Accessing variable in TDZ throws ReferenceError
- Exists for let, const, and class declarations

#### Q: Difference between function and block scope?

- Function scope: variables accessible throughout entire function
- Block scope: variables accessible only within {} block
- var is function-scoped, let/const are block-scoped

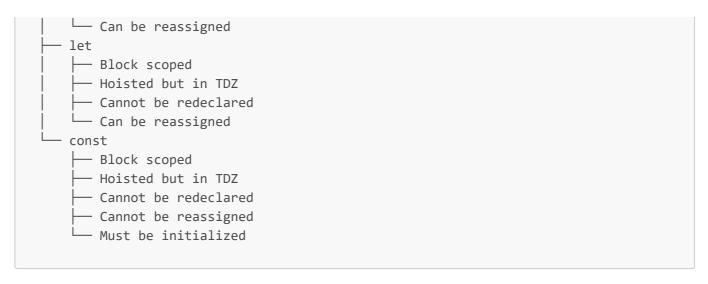
#### Q: What is the scope chain?

- JavaScript looks for variables starting from current scope
- If not found, looks in outer scope, then outer's outer, etc.
- Continues until global scope or ReferenceError

## **Asked at Companies:**

- Google: "Explain what happens when you declare a variable with var vs let"
- Facebook: "What is the Temporal Dead Zone and why does it exist?"
- Amazon: "Fix this loop so each timeout prints a different number"
- Microsoft: "Explain the difference between function and block scope"
- Netflix: "What is hoisting and how does it work with functions?"

# 



# **©** Key Takeaways

- 1. Use const by default, let when you need to reassign
- 2. Avoid var in modern JavaScript
- 3. Understand hoisting to avoid bugs
- 4. Be aware of Temporal Dead Zone with let/const
- 5. Keep variables in the smallest possible scope
- 6. Use block scope to prevent variable leaking
- 7. Understand scope chain for debugging

**Previous Chapter**: ← Functions Deep Dive **Next Chapter**: Closures Explained →

Practice: Try the hoisting quiz and experiment with different scoping scenarios!



# Chapter 6: Closures Explained

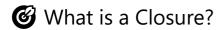
Master one of JavaScript's most powerful and misunderstood features: closures.

# Plain English Explanation

A closure is like a backpack that a function carries around. When a function is created inside another function, it "packs" all the variables from its parent scope into this backpack. Even when the function travels far from home (gets called elsewhere), it still has access to everything in its backpack.

Think of it like:

- **Function** = a person
- Closure = their backpack with memories/tools
- Outer variables = items packed in the backpack
- Inner function = the person who can always access their backpack



**Basic Closure Example** 

```
function outerFunction(x) {
    // This is the outer function's scope

function innerFunction(y) {
    // Inner function has access to outer function's variables
    return x + y; // x is from outer scope!
    }

    return innerFunction;
}

const addFive = outerFunction(5);
console.log(addFive(3)); // 8

// Even though outerFunction has finished executing,
// innerFunction still remembers x = 5!
```

#### Closure Visualization

```
function createGreeting(greeting) {
    // Outer scope variable
    const message = greeting;

    // Inner function "closes over" the message variable
    return function (name) {
        return `${message}, ${name}!`;
    };
}

const sayHello = createGreeting("Hello");
const sayHi = createGreeting("Hi");

console.log(sayHello("Alice")); // "Hello, Alice!"
console.log(sayHi("Bob")); // "Hi, Bob!"

// Each closure maintains its own copy of the outer variables
```

# Practical Closure Patterns

1. Private Variables (Data Encapsulation)

```
function createBankAccount(initialBalance) {
  let balance = initialBalance; // Private variable

  return {
    deposit(amount) {
      if (amount > 0) {
```

```
balance += amount;
        return balance;
      }
     throw new Error("Deposit amount must be positive");
    withdraw(amount) {
      if (amount > 0 && amount <= balance) {
        balance -= amount;
        return balance;
      }
     throw new Error("Invalid withdrawal amount");
    },
    getBalance() {
      return balance;
    },
  };
}
const account = createBankAccount(100);
console.log(account.deposit(50)); // 150
console.log(account.withdraw(30)); // 120
console.log(account.getBalance()); // 120
// balance is private - cannot be accessed directly
// console.log(account.balance); // undefined
```

#### 2. Function Factories

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

const double = createMultiplier(2);
const triple = createMultiplier(3);
const quadruple = createMultiplier(4);

console.log(double(5)); // 10
  console.log(triple(5)); // 15
  console.log(quadruple(5)); // 20

// Each function remembers its own multiplier value
```

#### 3. Event Handlers with State

```
function createClickCounter() {
 let count = 0;
 return function () {
    count++;
   console.log(`Button clicked ${count} times`);
   return count;
 };
}
const buttonHandler = createClickCounter();
// Simulate button clicks
buttonHandler(); // "Button clicked 1 times"
buttonHandler(); // "Button clicked 2 times"
buttonHandler(); // "Button clicked 3 times"
// Each button can have its own counter
const anotherButton = createClickCounter();
anotherButton(); // "Button clicked 1 times" (separate counter)
```

#### 4. Module Pattern

```
const calculator = (function () {
 // Private variables and functions
 let result = ∅;
 function log(operation, value) {
   console.log(`${operation}(${value}) = ${result}`);
 }
 // Public API (returned object)
 return {
   add(value) {
     result += value;
     log("add", value);
     return this; // For method chaining
   },
   subtract(value) {
     result -= value;
     log("subtract", value);
     return this;
   },
   multiply(value) {
     result *= value;
     log("multiply", value);
     return this;
   },
```

```
divide(value) {
      if (value !== 0) {
        result /= value;
        log("divide", value);
      } else {
        console.log("Cannot divide by zero!");
      }
     return this;
    },
    getResult() {
     return result;
    },
    clear() {
      result = 0;
      console.log("Calculator cleared");
      return this;
    },
 };
})();
// Usage
calculator.add(10).multiply(2).subtract(5);
console.log(calculator.getResult()); // 15
```

# Closures in Loops

#### The Classic Loop Problem

```
// ★ Common mistake with var
for (var i = 0; i < 3; i++) {
  setTimeout(function () {
    console.log("var i:", i); // 3, 3, 3 (all reference same i)
 }, 100);
}
// Solution 1: Use let (block scope)
for (let i = 0; i < 3; i++) {
  setTimeout(function () {
   console.log("let i:", i); // 0, 1, 2 (each has own i)
  }, 100);
}
// Solution 2: IIFE to create closure
for (var i = 0; i < 3; i++) {
  (function (index) {
    setTimeout(function () {
      console.log("IIFE i:", index); // 0, 1, 2
    }, 100);
```

```
})(i);
}

// Solution 3: bind method
for (var i = 0; i < 3; i++) {
    setTimeout(
      function (index) {
        console.log("bind i:", index); // 0, 1, 2
      }.bind(null, i),
      100
      );
}</pre>
```

### Creating Multiple Closures in Loops

```
function createFunctions() {
  const functions = [];

  for (let i = 0; i < 3; i++) {
     functions.push(function () {
       return i * i; // Each function closes over its own i
      });
   }

  return functions;
}

const funcs = createFunctions();
console.log(funcs[0]()); // 0
  console.log(funcs[1]()); // 1
  console.log(funcs[2]()); // 4</pre>
```

# Advanced Closure Concepts

#### Closure with Multiple Nested Functions

```
function grandparent(g) {
  const grandparentVar = g;

  return function parent(p) {
    const parentVar = p;

    return function child(c) {
      const childVar = c;

      // Child has access to all ancestor scopes
      return {
          grandparent: grandparentVar,
          parent: parentVar,
    }
}
```

```
child: childVar,
    sum: grandparentVar + parentVar + childVar,
    };
  };
};

const family = grandparent(1)(2)(3);
console.log(family); // { grandparent: 1, parent: 2, child: 3, sum: 6 }
```

#### Closure with Shared State

```
function createSharedCounter() {
  let count = 0;
  return {
    increment() {
      return ++count;
    },
    decrement() {
      return --count;
    },
    getCount() {
     return count;
    },
    // Create a new function that shares the same count
    createIncrementer() {
      return () => ++count;
    },
 };
}
const counter = createSharedCounter();
const incrementer = counter.createIncrementer();
console.log(counter.increment()); // 1
console.log(incrementer()); // 2 (shares same count!)
console.log(counter.getCount()); // 2
```

#### Closure with Dynamic Behavior

```
function createConfigurableFunction(config) {
  const { prefix, suffix, transform } = config;

  return function (input) {
    let result = input;
```

```
if (transform) {
      result = transform(result);
   }
   return `${prefix || ""}${result}${suffix || ""}`;
 };
}
const upperCaseFormatter = createConfigurableFunction({
 prefix: "[",
 suffix: "]",
 transform: (str) => str.toUpperCase(),
});
const numberFormatter = createConfigurableFunction({
 prefix: "$",
 transform: (num) => num.toFixed(2),
});
console.log(upperCaseFormatter("hello")); // "[HELLO]"
console.log(numberFormatter(42.567)); // "$42.57"
```

# **@** Real-World Applications

#### 1. Debounce Function

```
function debounce(func, delay) {
  let timeoutId;
  return function (...args) {
    // Clear previous timeout
    clearTimeout(timeoutId);
    // Set new timeout
    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
 };
// Usage: Debounce search input
const searchInput = document.getElementById("search");
const debouncedSearch = debounce(function (event) {
  console.log("Searching for:", event.target.value);
 // Perform search...
}, 300);
// searchInput.addEventListener('input', debouncedSearch);
```

#### 2. Throttle Function

```
function throttle(func, limit) {
 let inThrottle;
 return function (...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => {
        inThrottle = false;
      }, limit);
 };
}
// Usage: Throttle scroll events
const throttledScroll = throttle(function () {
 console.log("Scroll event fired");
}, 100);
// window.addEventListener('scroll', throttledScroll);
```

#### 3. 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 result...");
    const result = fn.apply(this, args);
    cache.set(key, result);
    return result;
  };
}
// Expensive function
function fibonacci(n) {
  if (n \le 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
```

```
const memoizedFib = memoize(fibonacci);

console.log(memoizedFib(10)); // Computing result... 55
console.log(memoizedFib(10)); // Cache hit! 55
```

# 

### 1. Memory Leaks

```
// X Potential memory leak
function createLeakyFunction() {
 const largeData = new Array(1000000).fill("data");
 return function () {
   // Even if we don't use largeData, it's kept in memory!
   console.log("Function called");
 };
}
// ✓ Better: only close over what you need
function createEfficientFunction() {
  const largeData = new Array(1000000).fill("data");
 const dataLength = largeData.length; // Extract only what's needed
 return function () {
   console.log(`Function called, data size: ${dataLength}`);
 };
}
```

#### 2. Unexpected Behavior with Loops

```
// X All functions will alert 3
const functions = [];
for (var i = 0; i < 3; i++) {
   functions.push(function () {
     alert(i); // All reference the same i
   });
}

// ✓ Each function gets its own copy
const functionsFixed = [];
for (let i = 0; i < 3; i++) {
   functionsFixed.push(function () {
     alert(i); // Each has its own i
   });
}</pre>
```

#### 3. Closure Performance

```
// X Creating closure in every call
function inefficient(arr) {
 return arr.map(function (item) {
    return function () {
      return item * 2; // New closure for each item
    };
 });
// Reuse function when possible
function createDoubler(item) {
 return function () {
   return item * 2;
 };
}
function efficient(arr) {
 return arr.map(createDoubler);
}
```

# Mini Practice Problems

## **Problem 1: Counter Factory**

```
// Create a function that returns an object with increment, decrement, and value
methods
// Each counter should be independent

function createCounter(initialValue = 0) {
    // Your code here
}

const counter1 = createCounter(5);
const counter2 = createCounter(10);

console.log(counter1.increment()); // 6
console.log(counter2.increment()); // 11
console.log(counter1.value()); // 6
console.log(counter2.value()); // 11
```

### Problem 2: Function Multiplier

```
// Create a function that returns a new function that multiplies its input by a
factor
```

```
function createMultiplier(factor) {
   // Your code here
}

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

console.log(double(5)); // 10
console.log(triple(4)); // 12
```

### Problem 3: Private Variable Challenge

```
// Create a function that manages a private array
// Return methods to add, remove, and get items
// The array should not be directly accessible

function createPrivateArray() {
    // Your code here
}

const arr = createPrivateArray();
arr.add("apple");
arr.add("banana");
console.log(arr.getItems()); // ['apple', 'banana']
arr.remove("apple");
console.log(arr.getItems()); // ['banana']
// console.log(arr.items); // Should be undefined
```

## Problem 4: Loop Closure Fix

```
// Fix this code so it prints 0, 1, 2 after 1 second
// Provide multiple solutions

for (var i = 0; i < 3; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
}

// Currently prints: 3, 3, 3
// Should print: 0, 1, 2</pre>
```

#### Problem 5: Advanced Closure

```
// Create a function that returns an object with methods to:
// 1. Set a value
```

```
// 2. Get the value
// 3. Get the history of all values that were set
// 4. Reset to initial value

function createValueTracker(initialValue) {
    // Your code here
}

const tracker = createValueTracker(0);
tracker.setValue(5);
tracker.setValue(10);
console.log(tracker.getValue()); // 10
console.log(tracker.getValue()); // [0, 5, 10]
tracker.reset();
console.log(tracker.getValue()); // 0
```

# Interview Notes

#### **Common Questions:**

#### Q: What is a closure?

- A function that has access to variables from its outer (enclosing) scope
- Even after the outer function has returned
- Creates a "persistent scope" that the inner function can access

#### Q: How do closures work in JavaScript?

- When a function is created, it "closes over" variables from its lexical scope
- JavaScript engine keeps these variables alive as long as the closure exists
- Each closure maintains its own copy of the outer variables

#### Q: What are practical uses of closures?

- Data privacy/encapsulation
- Function factories
- Callbacks with state
- Module pattern
- Debouncing/throttling

#### Q: What are potential issues with closures?

- Memory leaks if large objects are unnecessarily closed over
- Performance impact if overused
- · Can make debugging more difficult

## **Asked at Companies:**

- Google: "Implement a function that can only be called once using closures"
- Facebook: "Fix this loop so each timeout prints a different number"
- Amazon: "Create a private counter using closures"

- Microsoft: "Explain how closures can cause memory leaks"
- Netflix: "Implement debounce function using closures"

# **III** Closure Mental Model

```
Closure Components:

Function Definition

Inner function code

Reference to outer scope

Lexical Environment

Local variables

Parameters

Reference to parent environment

Scope Chain

Current scope

Outer scope(s)

Global scope
```

# **©** Key Takeaways

- 1. Closures give functions access to their outer scope
- 2. Use closures for data privacy and encapsulation
- 3. Each closure maintains its own copy of outer variables
- 4. Be careful with memory leaks don't close over unnecessary data
- 5. Closures are created every time a function is created
- 6. Understanding closures is crucial for advanced JavaScript patterns
- 7. Use let instead of var in loops to avoid closure issues

Previous Chapter: ← Scope, Hoisting & TDZ

Next Chapter: Arrays & Array Methods →

**Practice**: Try creating your own closure examples and experiment with the loop problems!

# Chapter 7: Arrays & Array Methods

Master JavaScript arrays and their powerful built-in methods for data manipulation.

# Plain English Explanation

Arrays are like organized lists or containers that hold multiple items in order. Think of them as:

- Shopping list = array of items to buy
- Playlist = array of songs in order
- Photo album = array of pictures
- **To-do list** = array of tasks

Array methods are like tools that help you work with these lists - adding items, removing items, finding items, transforming items, etc.

# Creating Arrays

### Array Literal (Preferred)

```
// Empty array
const emptyArray = [];

// Array with initial values
const fruits = ["apple", "banana", "orange"];
const numbers = [1, 2, 3, 4, 5];
const mixed = ["hello", 42, true, null, { name: "Alice" }];

// Multi-line for readability
const colors = ["red", "green", "blue", "yellow"];
```

### **Array Constructor**

```
// Empty array
const arr1 = new Array();

// Array with specific length (filled with undefined)
const arr2 = new Array(5); // [undefined, undefined, undefined,
undefined]

// Array with initial values
const arr3 = new Array("a", "b", "c"); // ['a', 'b', 'c']

// A Gotcha: Single number creates array with that length
const arr4 = new Array(3); // [undefined, undefined]
const arr5 = [3]; // [3]
```

#### Array.from() and Array.of()

```
// Array.from() - creates array from iterable
const str = "hello";
const charArray = Array.from(str); // ['h', 'e', 'l', 'l', 'o']

// Array.from() with mapping function
const numbers = Array.from({ length: 5 }, (_, i) => i + 1); // [1, 2, 3, 4, 5]

// Array.of() - creates array from arguments
const arr6 = Array.of(1, 2, 3); // [1, 2, 3]
const arr7 = Array.of(3); // [3] (not array with length 3!)
```

# Accessing Array Elements

#### **Basic Access**

```
const fruits = ["apple", "banana", "orange", "grape"];

// Access by index (0-based)
console.log(fruits[0]); // 'apple'
console.log(fruits[1]); // 'banana'
console.log(fruits[-1]); // undefined (no negative indexing)

// Get last element
console.log(fruits[fruits.length - 1]); // 'grape'

// Array length
console.log(fruits.length); // 4

// Check if index exists
if (fruits[10] !== undefined) {
   console.log(fruits[10]);
} else {
   console.log("Index 10 does not exist");
}
```

### **Destructuring Assignment**

```
const colors = ["red", "green", "blue", "yellow"];

// Basic destructuring
const [first, second] = colors;
console.log(first, second); // 'red', 'green'

// Skip elements
const [, , third] = colors;
console.log(third); // 'blue'

// Rest operator
const [primary, ...secondary] = colors;
console.log(primary); // 'red'
console.log(secondary); // ['green', 'blue', 'yellow']

// Default values
const [a, b, c, d, e = "default"] = colors;
console.log(e); // 'default'
```

# + Adding Elements

push() - Add to End

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

// Add single element
fruits.push("orange");
console.log(fruits); // ['apple', 'banana', 'orange']

// Add multiple elements
fruits.push("grape", "kiwi");
console.log(fruits); // ['apple', 'banana', 'orange', 'grape', 'kiwi']

// push() returns new length
const newLength = fruits.push("mango");
console.log(newLength); // 6
```

## unshift() - Add to Beginning

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

// Add to beginning
numbers.unshift(1);
console.log(numbers); // [1, 2, 3, 4]

// Add multiple to beginning
numbers.unshift(-1, 0);
console.log(numbers); // [-1, 0, 1, 2, 3, 4]
```

## splice() - Add at Specific Position

```
const letters = ["a", "c", "d"];

// Insert 'b' at index 1
letters.splice(1, 0, "b");
console.log(letters); // ['a', 'b', 'c', 'd']

// Insert multiple elements
letters.splice(2, 0, "x", "y");
console.log(letters); // ['a', 'b', 'x', 'y', 'c', 'd']
```

## Removing Elements

### pop() - Remove from End

```
const fruits = ["apple", "banana", "orange"];
// Remove and return last element
```

```
const removed = fruits.pop();
console.log(removed); // 'orange'
console.log(fruits); // ['apple', 'banana']
```

## shift() - Remove from Beginning

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

// Remove and return first element
const removed = numbers.shift();
console.log(removed); // 1
console.log(numbers); // [2, 3, 4]
```

## splice() - Remove from Specific Position

```
const colors = ["red", "green", "blue", "yellow"];

// Remove 1 element at index 1
const removed = colors.splice(1, 1);
console.log(removed); // ['green']
console.log(colors); // ['red', 'blue', 'yellow']

// Remove multiple elements
const moreRemoved = colors.splice(1, 2);
console.log(moreRemoved); // ['blue', 'yellow']
console.log(colors); // ['red']
```

## delete Operator (Not Recommended)

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

// delete creates "holes" in array
delete arr[2];
console.log(arr); // [1, 2, undefined, 4, 5]
console.log(arr.length); // 5 (length unchanged!)

// Use splice() instead
const arr2 = [1, 2, 3, 4, 5];
arr2.splice(2, 1);
console.log(arr2); // [1, 2, 4, 5]
console.log(arr2.length); // 4
```

## Iteration Methods

forEach() - Execute Function for Each Element

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

// Basic forEach
numbers.forEach(function (number) {
   console.log(number * 2);
});

// Arrow function with index
numbers.forEach((number, index) => {
   console.log(`Index ${index}: ${number}`);
});

// With array parameter
numbers.forEach((number, index, array) => {
   console.log(
   `${number} is at index ${index} of array with length ${array.length}`);
});

});
```

### map() - Transform Each Element

```
const numbers = [1, 2, 3, 4, 5];
// Double each number
const doubled = numbers.map((num) => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]
// Transform objects
const users = [
 { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 35 },
1;
const names = users.map((user) => user.name);
console.log(names); // ['Alice', 'Bob', 'Charlie']
// More complex transformation
const userInfo = users.map((user, index) => ({
  id: index + 1,
  displayName: `${user.name} (${user.age} years old)`,
  isAdult: user.age >= 18,
}));
console.log(userInfo);
```

## filter() - Filter Elements

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Filter even numbers
const evens = numbers.filter((num) => num % 2 === 0);
console.log(evens); // [2, 4, 6, 8, 10]

// Filter objects
const products = [
    { name: "Laptop", price: 1000, inStock: true },
    { name: "Phone", price: 500, inStock: false },
    { name: "Tablet", price: 300, inStock: true },
];

const availableProducts = products.filter((product) => product.inStock);
const expensiveProducts = products.filter((product) => product.price > 400);

console.log(availableProducts);
console.log(expensiveProducts);
```

## reduce() - Reduce to Single Value

```
const numbers = [1, 2, 3, 4, 5];
// Sum all numbers
const sum = numbers.reduce((accumulator, current) => {
  return accumulator + current;
}, 0);
console.log(sum); // 15
// Find maximum
const max = numbers.reduce((max, current) => {
 return current > max ? current : max;
console.log(max); // 5
// Count occurrences
const fruits = ["apple", "banana", "apple", "orange", "banana", "apple"];
const count = fruits.reduce((acc, fruit) => {
 acc[fruit] = (acc[fruit] || 0) + 1;
 return acc;
console.log(count); // { apple: 3, banana: 2, orange: 1 }
// Group by property
const people = [
 { name: "Alice", age: 25, city: "New York" },
 { name: "Bob", age: 30, city: "London" },
  { name: "Charlie", age: 35, city: "New York" },
];
```

```
const groupedByCity = people.reduce((acc, person) => {
  const city = person.city;
  if (!acc[city]) {
    acc[city] = [];
  }
  acc[city].push(person);
  return acc;
}, {});
console.log(groupedByCity);
```

## Search Methods

## find() - Find First Matching Element

```
const users = [
    { id: 1, name: "Alice", active: true },
    { id: 2, name: "Bob", active: false },
    { id: 3, name: "Charlie", active: true },
];

// Find first active user
const activeUser = users.find((user) => user.active);
console.log(activeUser); // { id: 1, name: 'Alice', active: true }

// Find by ID
const userById = users.find((user) => user.id === 2);
console.log(userById); // { id: 2, name: 'Bob', active: false }

// Returns undefined if not found
const notFound = users.find((user) => user.id === 999);
console.log(notFound); // undefined
```

### findIndex() - Find Index of First Match

```
const numbers = [10, 20, 30, 40, 50];

// Find index of first number > 25
const index = numbers.findIndex((num) => num > 25);
console.log(index); // 2 (index of 30)

// Returns -1 if not found
const notFoundIndex = numbers.findIndex((num) => num > 100);
console.log(notFoundIndex); // -1
```

### indexOf() and lastIndexOf()

```
const fruits = ["apple", "banana", "orange", "banana", "grape"];

// Find first occurrence
console.log(fruits.indexOf("banana")); // 1
console.log(fruits.indexOf("kiwi")); // -1 (not found)

// Find last occurrence
console.log(fruits.lastIndexOf("banana")); // 3

// With start position
console.log(fruits.indexOf("banana", 2)); // 3 (start searching from index 2)
```

### includes() - Check if Element Exists

```
const colors = ["red", "green", "blue"];

console.log(colors.includes("red")); // true
console.log(colors.includes("yellow")); // false

// Case sensitive
console.log(colors.includes("Red")); // false

// With start position
console.log(colors.includes("green", 2)); // false (start from index 2)
```

# Testing Methods

## some() - Test if Any Element Passes

```
const numbers = [1, 3, 5, 8, 9];

// Check if any number is even
const hasEven = numbers.some((num) => num % 2 === 0);
console.log(hasEven); // true (8 is even)

// Check if any number is > 10
const hasLarge = numbers.some((num) => num > 10);
console.log(hasLarge); // false
```

## every() - Test if All Elements Pass

```
const numbers = [2, 4, 6, 8, 10];

// Check if all numbers are even
const allEven = numbers.every((num) => num % 2 === 0);
```

```
console.log(allEven); // true

// Check if all numbers are > 5
const allLarge = numbers.every((num) => num > 5);
console.log(allLarge); // false (2 and 4 are not > 5)
```

# Utility Methods

## join() - Convert to String

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

// Default separator (comma)
console.log(fruits.join()); // "apple,banana,orange"

// Custom separator
console.log(fruits.join(" - ")); // "apple - banana - orange"
console.log(fruits.join("")); // "applebananaorange"
console.log(fruits.join(" and ")); // "apple and banana and orange"
```

## reverse() - Reverse Array (Mutates Original)

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

// Reverse in place
numbers.reverse();
console.log(numbers); // [5, 4, 3, 2, 1]

// To avoid mutation, create copy first
const original = [1, 2, 3, 4, 5];
const reversed = [...original].reverse();
console.log(original); // [1, 2, 3, 4, 5] (unchanged)
console.log(reversed); // [5, 4, 3, 2, 1]
```

### sort() - Sort Array (Mutates Original)

```
const fruits = ["banana", "apple", "orange", "grape"];

// Default sort (alphabetical)
fruits.sort();
console.log(fruits); // ['apple', 'banana', 'grape', 'orange']

// Numbers need custom compare function
const numbers = [10, 5, 40, 25, 1000, 1];

// X Wrong - converts to strings first
```

```
numbers.sort();
console.log(numbers); // [1, 10, 1000, 25, 40, 5]
// // Correct - numeric sort
const nums = [10, 5, 40, 25, 1000, 1];
nums.sort((a, b) => a - b); // Ascending
console.log(nums); // [1, 5, 10, 25, 40, 1000]
// Descending
nums.sort((a, b) \Rightarrow b - a);
console.log(nums); // [1000, 40, 25, 10, 5, 1]
// Sort objects
const people = [
 { name: "Alice", age: 30 },
  { name: "Bob", age: 25 },
 { name: "Charlie", age: 35 },
1;
// Sort by age
people.sort((a, b) => a.age - b.age);
console.log(people);
// Sort by name
people.sort((a, b) => a.name.localeCompare(b.name));
console.log(people);
```

## slice() - Extract Portion (Non-Mutating)

```
const fruits = ["apple", "banana", "orange", "grape", "kiwi"];

// Extract from index 1 to 3 (exclusive)
const sliced = fruits.slice(1, 3);
console.log(sliced); // ['banana', 'orange']
console.log(fruits); // Original unchanged

// From index to end
const fromIndex = fruits.slice(2);
console.log(fromIndex); // ['orange', 'grape', 'kiwi']

// Negative indices (from end)
const lastTwo = fruits.slice(-2);
console.log(lastTwo); // ['grape', 'kiwi']

// Copy entire array
const copy = fruits.slice();
console.log(copy); // ['apple', 'banana', 'orange', 'grape', 'kiwi']
```

# Chaining Methods

### Method Chaining Examples

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
// Chain multiple operations
const result = numbers
  .filter((num) => num % 2 === 0) // Get even numbers: [2, 4, 6, 8, 10]
  .map((num) => num * num) // Square them: [4, 16, 36, 64, 100]
  .reduce((sum, num) => sum + num, 0); // Sum them: 220
console.log(result); // 220
// More complex example
const users = [
 { name: "Alice", age: 25, active: true, score: 85 },
  { name: "Bob", age: 30, active: false, score: 92 },
  { name: "Charlie", age: 35, active: true, score: 78 },
  { name: "Diana", age: 28, active: true, score: 95 },
1;
const topActiveUsers = users
  .filter((user) => user.active) // Only active users
  .filter((user) => user.score > 80) // High scores only
  .sort((a, b) => b.score - a.score) // Sort by score (descending)
  .map((user) \Rightarrow (\{
   // Transform to simpler object
    name: user.name,
    score: user.score,
    grade: user.score >= 90 ? "A" : "B",
  }));
console.log(topActiveUsers);
// [{ name: 'Diana', score: 95, grade: 'A' }, { name: 'Alice', score: 85, grade:
'B' }]
```

## 

### 1. Mutating vs Non-Mutating Methods

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

// X These methods mutate the original array
original.push(4); // [1, 2, 3, 4]
original.pop(); // [1, 2, 3]
original.reverse(); // [3, 2, 1]
original.sort(); // [1, 2, 3]
original.splice(1, 1); // [1, 3]

// V These methods return new arrays
const arr = [1, 2, 3];
```

```
const mapped = arr.map((x) => x * 2); // [2, 4, 6], arr unchanged const filtered = arr.filter((x) => x > 1); // [2, 3], arr unchanged const sliced = arr.slice(1); // [2, 3], arr unchanged
```

## 2. Array Reference vs Copy

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

// X This creates a reference, not a copy
const reference = original;
reference.push(4);
console.log(original); // [1, 2, 3, 4] - original is modified!

// ✓ Create actual copies
const copy1 = [...original]; // Spread operator
const copy2 = Array.from(original); // Array.from()
const copy3 = original.slice(); // slice() with no arguments
```

## 3. Sparse Arrays

```
// Creating sparse arrays
const sparse = new Array(3); // [undefined, undefined, undefined]
const sparse2 = [1, , 3]; // [1, undefined, 3]

// Some methods skip holes
console.log(sparse2.map((x) => x * 2)); // [2, undefined, 6]

// Others don't
console.log(sparse2.forEach((x) => console.log(x))); // Only logs 1 and 3
```

### 4. Sort Gotchas

```
// X Default sort converts to strings
const numbers = [1, 10, 2, 20];
numbers.sort();
console.log(numbers); // [1, 10, 2, 20] - Wrong!

// V Use compare function for numbers
numbers.sort((a, b) => a - b);
console.log(numbers); // [1, 2, 10, 20] - Correct!
```

# Mini Practice Problems

### **Problem 1: Array Manipulation**

```
// Given an array of numbers, create a new array with:
// 1. Only even numbers
// 2. Each number squared
// 3. Sorted in descending order

const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Your solution here
const result = numbers;
// Your code here

console.log(result); // Should output: [100, 64, 36, 16, 4]
```

## **Problem 2: Object Array Processing**

## Problem 3: Custom Array Methods

```
// Implement these array methods from scratch:

// 1. Custom map
Array.prototype.myMap = function (callback) {
    // Your implementation
};

// 2. Custom filter
Array.prototype.myFilter = function (callback) {
    // Your implementation
};

// 3. Custom reduce
```

```
Array.prototype.myReduce = function (callback, initialValue) {
    // Your implementation
};

// Test your implementations
const nums = [1, 2, 3, 4, 5];
console.log(nums.myMap((x) => x * 2)); // [2, 4, 6, 8, 10]
console.log(nums.myFilter((x) => x % 2 === 0)); // [2, 4]
console.log(nums.myReduce((a, b) => a + b, 0)); // 15
```

## Problem 4: Array Flattening

```
// Flatten a nested array (one level deep)
const nested = [1, [2, 3], [4, [5, 6]], 7];

// Your solution here (don't use flat())
function flattenArray(arr) {
   // Your code here
}

console.log(flattenArray(nested)); // [1, 2, 3, 4, [5, 6], 7]
```

## **Problem 5: Remove Duplicates**

```
// Remove duplicates from an array while preserving order
const withDuplicates = [1, 2, 2, 3, 4, 4, 5, 1];
// Provide multiple solutions:
// 1. Using Set
// 2. Using filter + indexOf
// 3. Using reduce
function removeDuplicates1(arr) {
 // Using Set
}
function removeDuplicates2(arr) {
 // Using filter + indexOf
}
function removeDuplicates3(arr) {
 // Using reduce
}
console.log(removeDuplicates1(withDuplicates)); // [1, 2, 3, 4, 5]
```

## Interview Notes

#### **Common Questions:**

#### Q: What's the difference between map() and forEach()?

- map() returns a new array with transformed elements
- forEach() executes a function for each element but returns undefined
- Use map() when you need to transform data, forEach() for side effects

#### Q: How do you remove duplicates from an array?

```
• [...new Set(array)] - using Set (ES6)
```

- array.filter((item, index) => array.indexOf(item) === index)
- Using reduce() to build unique array

### Q: What's the difference between slice() and splice()?

- slice() returns a new array (non-mutating)
- splice() modifies the original array (mutating)
- slice() for extraction, splice() for insertion/deletion

### Q: How do you flatten an array?

- array.flat() ES2019 method
- [].concat(...array) spread with concat
- array.reduce((acc, val) => acc.concat(val), [])

## Asked at Companies:

- Google: "Implement array methods from scratch (map, filter, reduce)"
- Facebook: "Find the intersection of two arrays"
- Amazon: "Remove duplicates from an array without using Set"
- Microsoft: "Sort an array of objects by multiple properties"
- Netflix: "Implement a function to group array elements by a property"

## Array Methods Cheat Sheet

```
reduce() - Reduce to single value
find() - Find first match
some()/every() - Test elements
```

# **@** Key Takeaways

- 1. Understand mutating vs non-mutating methods
- 2. Use method chaining for readable data transformations
- 3. Choose the right method for the task (map vs forEach)
- 4. Be careful with array references vs copies
- 5. Use spread operator for array copying and concatenation
- 6. Remember that sort() converts to strings by default
- 7. Practice implementing array methods from scratch

Previous Chapter: ← Closures Explained
Next Chapter: Objects & Object Access →

**Practice**: Try the array manipulation problems and experiment with method chaining!

# 🔁 Chapter 8: Objects & Object Access

Master JavaScript objects, the foundation of everything in JavaScript.

## Plain English Explanation

Objects are like containers that hold related information and actions together. Think of them as:

- **Person** = object with properties (name, age, height) and actions (walk, talk, eat)
- Car = object with properties (color, model, year) and actions (start, stop, accelerate)
- **Book** = object with properties (title, author, pages) and actions (open, close, bookmark)
- Bank Account = object with properties (balance, owner) and actions (deposit, withdraw)

In JavaScript, almost everything is an object (except primitives), and objects are the building blocks of complex applications.

# Creating Objects

Object Literal (Most Common)

```
// Empty object
const emptyObj = {};

// Object with properties
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
```

```
isEmployed: true,
};
// Object with methods
const calculator = {
  result: 0,
  add: function (num) {
    this.result += num;
    return this;
  },
  subtract: function (num) {
    this.result -= num;
   return this;
  },
  getValue: function () {
    return this.result;
  },
};
// ES6 method shorthand
const modernCalculator = {
  result: 0,
  add(num) {
    this.result += num;
   return this;
  },
  subtract(num) {
   this.result -= num;
    return this;
  },
  getValue() {
    return this.result;
  },
};
```

## **Object Constructor**

```
// Using Object constructor
const obj1 = new Object();
obj1.name = "Alice";
obj1.age = 30;

// Equivalent to object literal
const obj2 = {
   name: "Alice",
   age: 30,
};
```

## Object.create()

```
// Create object with specific prototype
const personPrototype = {
  greet: function () {
    return `Hello, I'm ${this.name}`;
  },
};

const alice = Object.create(personPrototype);
alice.name = "Alice";
alice.age = 30;

console.log(alice.greet()); // "Hello, I'm Alice"

// Create object with null prototype (no inherited properties)
const pureObject = Object.create(null);
pureObject.name = "Pure";
console.log(pureObject.toString); // undefined (no inherited methods)
```

### **Constructor Functions**

```
// Constructor function (before ES6 classes)
function Person(name, age, city) {
    this.name = name;
    this.age = age;
    this.city = city;

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

// Create instances
const alice = new Person("Alice", 30, "New York");
const bob = new Person("Bob", 25, "London");

console.log(alice.greet()); // "Hello, I'm Alice from New York"
console.log(bob.greet()); // "Hello, I'm Bob from London"
```

### **ES6 Classes**

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

greet() {
```

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

getAge() {
    return this.age;
}

// Static method
static createAnonymous() {
    return new Person("Anonymous", 0, "Unknown");
}

const alice = new Person("Alice", 30, "New York");
console.log(alice.greet());

const anon = Person.createAnonymous();
console.log(anon.name); // 'Anonymous'
```

# Accessing Object Properties

#### **Dot Notation**

```
const person = {
 name: "Alice",
 age: 30,
 address: {
   street: "123 Main St",
    city: "New York",
    zipCode: "10001",
 },
};
// Reading properties
console.log(person.name); // 'Alice'
console.log(person.age); // 30
console.log(person.address.city); // 'New York'
// Setting properties
person.name = "Alice Smith";
person.email = "alice@example.com"; // Adding new property
person.address.country = "USA"; // Adding nested property
console.log(person.email); // 'alice@example.com'
```

### **Bracket Notation**

```
const person = {
  name: "Alice",
```

```
age: 30,
  "favorite color": "blue", // Property with space
 123: "numeric key", // Numeric string key
};
// Reading with bracket notation
console.log(person["name"]); // 'Alice'
console.log(person["favorite color"]); // 'blue'
console.log(person["123"]); // 'numeric key'
// Dynamic property access
const propertyName = "age";
console.log(person[propertyName]); // 30
// Setting with bracket notation
person["last name"] = "Smith";
person[propertyName] = 31;
// Computed property names
const prefix = "user";
const obj = {
  [prefix + "Name"]: "Alice", // userName: 'Alice'
  [prefix + "Age"]: 30, // userAge: 30
  [`${prefix}Email`]: "alice@example.com", // userEmail: 'alice@example.com'
};
console.log(obj.userName); // 'Alice'
console.log(obj.userAge); // 30
console.log(obj.userEmail); // 'alice@example.com'
```

#### When to Use Dot vs Bracket Notation

```
const obj = {
 name: "Alice",
  "first-name": "Alice",
 123: "number",
 "user name": "alice123",
};
// - Property name is a valid identifier
// - Property name is known at write time
console.log(obj.name);
// ✓ Use bracket notation when:
// - Property name has spaces, hyphens, or special characters
console.log(obj["first-name"]);
console.log(obj["user name"]);
// - Property name starts with a number
console.log(obj["123"]);
```

```
// - Property name is dynamic/computed
const propName = "name";
console.log(obj[propName]);

// - Property name comes from a variable
function getValue(object, key) {
   return object[key];
}

console.log(getValue(obj, "name")); // 'Alice'
```

# Object Methods and this

#### Method Definition

```
const person = {
 name: "Alice",
 age: 30,
 // Method using function keyword
 greet: function () {
  return `Hello, I'm ${this.name}`;
 },
 // ES6 method shorthand
 introduce() {
   return `I'm ${this.name} and I'm ${this.age} years old`;
 },
 // Arrow function (be careful with 'this'!)
 arrowMethod: () => {
   // 'this' doesn't refer to the object!
   return `Hello from ${this.name}`; // undefined
 },
 // Method that modifies object
 haveBirthday() {
   this.age++;
   return `Happy birthday! Now I'm ${this.age}`;
 },
};
console.log(person.greet()); // "Hello, I'm Alice"
console.log(person.introduce()); // "I'm Alice and I'm 30 years old"
console.log(person.arrowMethod()); // "Hello from undefined"
console.log(person.haveBirthday()); // "Happy birthday! Now I'm 31"
```

### Understanding this Context

```
const person = {
 name: "Alice",
 greet() {
   return `Hello, I'm ${this.name}`;
 },
};
// Method called on object - 'this' refers to person
console.log(person.greet()); // "Hello, I'm Alice"
// Method assigned to variable - 'this' context lost
const greetFunction = person.greet;
console.log(greetFunction()); // "Hello, I'm undefined"
// Binding 'this' context
const boundGreet = person.greet.bind(person);
console.log(boundGreet()); // "Hello, I'm Alice"
// Using call() and apply()
const anotherPerson = { name: "Bob" };
console.log(person.greet.call(anotherPerson)); // "Hello, I'm Bob"
console.log(person.greet.apply(anotherPerson)); // "Hello, I'm Bob"
```

## Method Chaining

```
const calculator = {
 value: ∅,
 add(num) {
   this.value += num;
   return this; // Return 'this' for chaining
 },
 subtract(num) {
   this.value -= num;
   return this;
 },
 multiply(num) {
   this.value *= num;
   return this;
 },
 divide(num) {
   if (num !== 0) {
     this.value /= num;
   return this;
 },
```

```
getValue() {
    return this.value;
},

reset() {
    this.value = 0;
    return this;
},
};

// Method chaining
const result = calculator.add(10).multiply(2).subtract(5).divide(3).getValue();

console.log(result); // 5

// Reset and chain again
calculator.reset().add(100).subtract(50).multiply(2);
console.log(calculator.getValue()); // 100
```

# **Q** Object Property Descriptors

### **Property Descriptors**

```
const person = {
 name: "Alice",
 age: 30,
};
// Get property descriptor
const nameDescriptor = Object.getOwnPropertyDescriptor(person, "name");
console.log(nameDescriptor);
// {
// value: 'Alice',
   writable: true,
// enumerable: true,
// configurable: true
// }
// Define property with custom descriptor
Object.defineProperty(person, "id", {
 value: 12345,
 writable: false, // Cannot be changed
 enumerable: false, // Won't show in for...in or Object.keys()
  configurable: false, // Cannot be deleted or reconfigured
});
console.log(person.id); // 12345
person.id = 54321; // Silently fails (or throws in strict mode)
console.log(person.id); // 12345 (unchanged)
console.log(Object.keys(person)); // ['name', 'age'] (id not enumerable)
```

```
// Define multiple properties
Object.defineProperties(person, {
    firstName: {
        value: "Alice",
        writable: true,
        enumerable: true,
        configurable: true,
},
lastName: {
        value: "Smith",
        writable: true,
        enumerable: true,
        configurable: true,
        configurable: true,
},
```

#### **Getters and Setters**

```
const person = {
 firstName: "Alice",
 lastName: "Smith",
 // Getter - computed property
 get fullName() {
  return `${this.firstName} ${this.lastName}`;
 },
 // Setter - validates and sets value
 set fullName(value) {
   const parts = value.split(" ");
   if (parts.length === 2) {
     this.firstName = parts[0];
     this.lastName = parts[1];
   } else {
     throw new Error('Full name must be in format "First Last"');
   }
 },
 // Private-like property with getter/setter
 _age: 30,
 get age() {
  return this._age;
 },
 set age(value) {
   if (value < 0 || value > 150) {
     throw new Error("Age must be between 0 and 150");
   this._age = value;
```

```
},
};
// Using getters and setters
console.log(person.fullName); // "Alice Smith"
person.fullName = "Bob Johnson";
console.log(person.firstName); // "Bob"
console.log(person.lastName); // "Johnson"
console.log(person.age); // 30
person.age = 25;
console.log(person.age); // 25
// This will throw an error
// person.age = -5; // Error: Age must be between 0 and 150
// Define getter/setter with Object.defineProperty
const obj = {};
Object.defineProperty(obj, "temperature", {
  get() {
    return this._celsius;
 },
  set(celsius) {
   this._celsius = celsius;
   this._fahrenheit = (celsius * 9) / 5 + 32;
  },
 enumerable: true,
  configurable: true,
});
obj.temperature = 25;
console.log(obj.temperature); // 25
console.log(obj._fahrenheit); // 77
```

# Object Iteration

## for...in Loop

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
};

// Iterate over enumerable properties
for (const key in person) {
  console.log(`${key}: ${person[key]}`);
}

// Output:
// name: Alice
// age: 30
```

```
// city: New York

// Check if property belongs to object (not inherited)
for (const key in person) {
  if (person.hasOwnProperty(key)) {
    console.log(`${key}: ${person[key]}`);
  }
}
```

## Object.keys(), Object.values(), Object.entries()

```
const person = {
 name: "Alice",
 age: 30,
 city: "New York",
};
// Get all keys
const keys = Object.keys(person);
console.log(keys); // ['name', 'age', 'city']
// Get all values
const values = Object.values(person);
console.log(values); // ['Alice', 30, 'New York']
// Get key-value pairs
const entries = Object.entries(person);
console.log(entries); // [['name', 'Alice'], ['age', 30], ['city', 'New York']]
// Iterate using these methods
Object.keys(person).forEach((key) => {
 console.log(`${key}: ${person[key]}`);
});
Object.entries(person).forEach(([key, value]) => {
 console.log(`${key}: ${value}`);
});
// Convert back to object
const newObj = Object.fromEntries(entries);
console.log(newObj); // { name: 'Alice', age: 30, city: 'New York' }
```

# **%** Object Utility Methods

Object.assign() - Copying and Merging

```
const target = { a: 1, b: 2 };
const source1 = { b: 3, c: 4 };
const source2 = { c: 5, d: 6 };
```

```
// Merge objects (modifies target)
Object.assign(target, source1, source2);
console.log(target); // { a: 1, b: 3, c: 5, d: 6 }
// Create new object without modifying original
const original = { name: "Alice", age: 30 };
const copy = Object.assign({}, original, { city: "New York" });
console.log(copy); // { name: 'Alice', age: 30, city: 'New York' }
console.log(original); // { name: 'Alice', age: 30 } (unchanged)
// Shallow copy limitation
const obj1 = {
 name: "Alice",
 address: { city: "New York", zip: "10001" },
};
const obj2 = Object.assign({}, obj1);
obj2.address.city = "Boston"; // Modifies original!
console.log(obj1.address.city); // 'Boston' (not 'New York'!)
```

### Spread Operator (ES6)

```
const obj1 = { a: 1, b: 2 };
const obj2 = \{ b: 3, c: 4 \};
const obj3 = \{ c: 5, d: 6 \};
// Merge objects with spread
const merged = { ...obj1, ...obj2, ...obj3 };
console.log(merged); // { a: 1, b: 3, c: 5, d: 6 }
// Add properties while copying
const person = { name: "Alice", age: 30 };
const employee = {
  ...person,
 id: 12345,
 department: "Engineering",
 age: 31, // Override existing property
console.log(employee); // { name: 'Alice', age: 31, id: 12345, department:
'Engineering' }
// Conditional properties
const includeEmail = true;
const user = {
 name: "Alice",
 age: 30,
  ...(includeEmail && { email: "alice@example.com" }),
};
console.log(user); // { name: 'Alice', age: 30, email: 'alice@example.com' }
```

## Object.freeze(), Object.seal(), Object.preventExtensions()

```
const obj = { name: "Alice", age: 30 };
// Object.freeze() - completely immutable
const frozen = Object.freeze({ ...obj });
frozen.name = "Bob"; // Silently fails
frozen.city = "New York"; // Silently fails
delete frozen.age; // Silently fails
console.log(frozen); // { name: 'Alice', age: 30 } (unchanged)
console.log(Object.isFrozen(frozen)); // true
// Object.seal() - can modify existing properties, but can't add/delete
const sealed = Object.seal({ ...obj });
sealed.name = "Bob"; // 
Works
sealed.city = "New York"; // X Silently fails
delete sealed.age; // ★ Silently fails
console.log(sealed); // { name: 'Bob', age: 30 }
console.log(Object.isSealed(sealed)); // true
// Object.preventExtensions() - can modify and delete, but can't add
const nonExtensible = Object.preventExtensions({ ...obj });
nonExtensible.name = "Bob"; // Works
nonExtensible.city = "New York"; // X Silently fails
delete nonExtensible.age; // ✓ Works
console.log(nonExtensible); // { name: 'Bob' }
console.log(Object.isExtensible(nonExtensible)); // false
```

# Object Comparison and Checking

## **Checking Properties**

```
const person = {
  name: "Alice",
  age: 30,
  address: {
    city: "New York",
  },
};

// Check if property exists
console.log("name" in person); // true
console.log("email" in person); // false
console.log("toString" in person); // true (inherited)

// Check own property (not inherited)
console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("toString")); // false

// Modern alternative to hasOwnProperty
```

```
console.log(Object.hasOwn(person, "name")); // true (ES2022)
console.log(Object.hasOwn(person, "toString")); // false

// Check if property is undefined
console.log(person.name !== undefined); // true
console.log(person.email !== undefined); // false

// Nested property checking
console.log(person.address && person.address.city); // 'New York'
console.log(person.address && person.address.zip); // undefined

// Optional chaining (ES2020)
console.log(person.address?.city); // 'New York'
console.log(person.address?.zip); // undefined
console.log(person.address?.zip); // undefined
console.log(person.contact?.email); // undefined (no error)
```

### **Object Comparison**

```
// Objects are compared by reference, not value
const obj1 = { name: "Alice" };
const obj2 = { name: "Alice" };
const obj3 = obj1;
console.log(obj1 === obj2); // false (different objects)
console.log(obj1 === obj3); // true (same reference)
// Shallow equality check
function shallowEqual(obj1, obj2) {
  const keys1 = Object.keys(obj1);
  const keys2 = Object.keys(obj2);
 if (keys1.length !== keys2.length) {
   return false;
  }
 for (let key of keys1) {
   if (obj1[key] !== obj2[key]) {
      return false;
    }
  return true;
}
console.log(shallowEqual(obj1, obj2)); // true
// Deep equality check (simplified)
function deepEqual(obj1, obj2) {
  if (obj1 === obj2) return true;
  if (obj1 == null || obj2 == null) return false;
```

```
if (typeof obj1 !== "object" || typeof obj2 !== "object") {
    return obj1 === obj2;
}

const keys1 = Object.keys(obj1);
const keys2 = Object.keys(obj2);

if (keys1.length !== keys2.length) return false;

for (let key of keys1) {
    if (!keys2.includes(key)) return false;
    if (!deepEqual(obj1[key], obj2[key])) return false;
}

return true;
}

const deep1 = { a: 1, b: { c: 2 } };
const deep2 = { a: 1, b: { c: 2 } };
console.log(deepEqual(deep1, deep2)); // true
```

## Destructuring Objects

## **Basic Destructuring**

```
const person = {
 name: "Alice",
  age: 30,
 city: "New York",
 country: "USA",
};
// Basic destructuring
const { name, age } = person;
console.log(name, age); // 'Alice', 30
// Rename variables
const { name: fullName, age: years } = person;
console.log(fullName, years); // 'Alice', 30
// Default values
const { name, email = "No email" } = person;
console.log(name, email); // 'Alice', 'No email'
// Rest operator
const { name: personName, ...rest } = person;
console.log(personName); // 'Alice'
console.log(rest); // { age: 30, city: 'New York', country: 'USA' }
```

### **Nested Destructuring**

```
const user = {
  id: 1,
  name: "Alice",
  address: {
    street: "123 Main St",
    city: "New York",
    coordinates: {
     lat: 40.7128,
     lng: -74.006,
    },
  },
  preferences: {
   theme: "dark",
   language: "en",
  },
};
// Nested destructuring
const {
  name,
  address: {
   city,
    coordinates: { lat, lng },
  },
  preferences: { theme },
} = user;
console.log(name, city, lat, lng, theme);
// 'Alice', 'New York', 40.7128, -74.0060, 'dark'
// Destructuring with default values for nested properties
const { address: { zipCode = "Unknown" } = {} } = user;
console.log(zipCode); // 'Unknown'
```

## **Function Parameter Destructuring**

```
// Function with object parameter
function greetUser({ name, age, city = "Unknown" }) {
   return `Hello ${name}, age ${age} from ${city}`;
}

const user = { name: "Alice", age: 30 };
console.log(greetUser(user)); // "Hello Alice, age 30 from Unknown"

// Nested destructuring in parameters
function processOrder({
   id,
```

```
customer: { name, email },
  items = [],
  shipping: { address, method = "standard" } = {},
}) {
 return {
   orderId: id,
    customerName: name,
    customerEmail: email,
    itemCount: items.length,
    shippingAddress: address,
    shippingMethod: method,
 };
}
const order = {
 id: "ORD-123",
  customer: {
    name: "Alice",
    email: "alice@example.com",
 },
 items: ["item1", "item2"],
 shipping: {
   address: "123 Main St",
 },
};
console.log(processOrder(order));
```

## 

#### 1. Reference vs Value

```
// Objects are passed by reference
const original = { name: "Alice", age: 30 };
const reference = original;

reference.age = 31;
console.log(original.age); // 31 (original is modified!)

// Create actual copy
const copy = { ...original };
copy.age = 32;
console.log(original.age); // 31 (original unchanged)

// Shallow copy limitation
const obj = {
   name: "Alice",
   hobbies: ["reading", "swimming"],
};

const shallowCopy = { ...obj };
```

```
shallowCopy.hobbies.push("cooking");
console.log(obj.hobbies); // ['reading', 'swimming', 'cooking'] (modified!)

// Deep copy (simple approach)
const deepCopy = JSON.parse(JSON.stringify(obj));
// Note: This doesn't work with functions, dates, undefined, etc.
```

### 2. this Context Issues

```
const person = {
 name: "Alice",
  greet: function () {
   return `Hello, I'm ${this.name}`;
 },
  arrowGreet: () => {
    return `Hello, I'm ${this.name}`; // 'this' is not the object!
 },
};
console.log(person.greet()); // "Hello, I'm Alice"
console.log(person.arrowGreet()); // "Hello, I'm undefined"
// Method assignment loses context
const greetFunc = person.greet;
console.log(greetFunc()); // "Hello, I'm undefined"
// Solutions:
// 1. Bind the method
const boundGreet = person.greet.bind(person);
console.log(boundGreet()); // "Hello, I'm Alice"
// 2. Use arrow function wrapper
const wrappedGreet = () => person.greet();
console.log(wrappedGreet()); // "Hello, I'm Alice"
```

## 3. Property Existence Checking

```
const obj = {
  name: "Alice",
  age: 0,
  active: false,
  data: null,
};

// X Wrong ways to check property existence
if (obj.age) {
  /* Won't execute because 0 is falsy */
}
if (obj.active) {
```

## 4. Deleting Properties

## Mini Practice Problems

## Problem 1: Object Manipulation

```
// Create a function that takes an array of objects and returns
// a new object with specific transformations
```

```
const users = [
    { id: 1, name: "Alice", age: 30, active: true },
    { id: 2, name: "Bob", age: 25, active: false },
    { id: 3, name: "Charlie", age: 35, active: true },
];

// Your function should return:
// {
    // activeUsers: ['Alice', 'Charlie'],
    // averageAge: 30,
    // userById: { 1: 'Alice', 2: 'Bob', 3: 'Charlie' }

// Your code here
}

console.log(processUsers(users));
```

## Problem 2: Deep Object Merging

```
// Implement a deep merge function that combines objects recursively
function deepMerge(target, source) {
 // Your implementation here
const obj1 = {
 a: 1,
 b: {
  c: 2,
  d: 3,
 },
 e: [1, 2],
};
const obj2 = {
 b: {
  d: 4,
  f: 5,
 },
 e: [3, 4],
 g: 6,
};
// Should result in:
// {
// a: 1,
// b: { c: 2, d: 4, f: 5 },
// e: [3, 4],
// g: 6
```

```
// }
console.log(deepMerge(obj1, obj2));
```

## Problem 3: Object Path Access

```
// Create a function that safely accesses nested object properties
// using a string path like 'user.address.city'
function getNestedValue(obj, path, defaultValue = undefined) {
 // Your implementation here
const data = {
 user: {
    name: "Alice",
    address: {
      city: "New York",
      coordinates: {
       lat: 40.7128,
     },
   },
  },
};
console.log(getNestedValue(data, "user.name")); // 'Alice'
console.log(getNestedValue(data, "user.address.city")); // 'New York'
console.log(getNestedValue(data, "user.address.coordinates.lat")); // 40.7128
console.log(getNestedValue(data, "user.email", "No email")); // 'No email'
console.log(getNestedValue(data, "user.address.zip")); // undefined
```

## Problem 4: Object Validation

```
// Create a validation system for objects

class ObjectValidator {
   constructor(schema) {
     this.schema = schema;
   }

   validate(obj) {
        // Your implementation here
        // Return { valid: boolean, errors: string[] }
   }
}

// Usage example:
const userSchema = {
   name: { type: "string", required: true },
```

```
age: { type: "number", required: true, min: 0, max: 150 },
    email: { type: "string", required: false, pattern: /^[^@]+@[^@]+\.[^@]+$/ },
};

const validator = new ObjectValidator(userSchema);

const validUser = { name: "Alice", age: 30, email: "alice@example.com" };

const invalidUser = { name: "", age: -5, email: "invalid-email" };

console.log(validator.validate(validUser)); // { valid: true, errors: [] }

console.log(validator.validate(invalidUser)); // { valid: false, errors: [...] }
```

## Problem 5: Object Proxy

```
// Create a proxy that logs all property access and modifications

function createLoggingProxy(obj, logFunction = console.log) {
    // Your implementation using Proxy
}

const person = { name: "Alice", age: 30 };
const loggedPerson = createLoggingProxy(person);

// These operations should log to console:
loggedPerson.name; // "GET: name = Alice"
loggedPerson.age = 31; // "SET: age = 31"
loggedPerson.city = "NYC"; // "SET: city = NYC"
delete loggedPerson.age; // "DELETE: age"
```

## Interview Notes

**Common Questions:** 

#### Q: What's the difference between dot notation and bracket notation?

- Dot notation: obj.property for valid identifiers, known at write time
- Bracket notation: obj['property'] for dynamic access, special characters, computed properties

### Q: How do you check if a property exists in an object?

- 'property' in obj checks own and inherited properties
- obj.hasOwnProperty('property') checks only own properties
- Object.hasOwn(obj, 'property') modern alternative (ES2022)
- obj.property !== undefined checks if value is not undefined

#### Q: What's the difference between Object.assign() and spread operator?

- Both perform shallow copying/merging
- Spread operator is more concise and readable

- Object.assign() modifies the target object, spread creates new object
- Spread operator doesn't work with getters/setters the same way

### Q: How do you deep copy an object?

- JSON.parse(JSON.stringify(obj)) simple but limited (no functions, dates, etc.)
- Lodash cloneDeep() robust library solution
- Custom recursive function for specific needs
- structuredClone() new native method (limited browser support)

## Asked at Companies:

- Google: "Implement Object.assign() from scratch"
- Facebook: "Create a function to flatten nested objects"
- Amazon: "Implement deep equality comparison for objects"
- Microsoft: "Design a system for object validation and transformation"
- Netflix: "Create a proxy that tracks object property access patterns"

# **6** Key Takeaways

- 1. Objects are reference types understand copying vs referencing
- 2. **Use appropriate property access method** dot vs bracket notation
- 3. Be careful with this context especially with arrow functions
- 4. **Understand property descriptors** writable, enumerable, configurable
- 5. Master object iteration methods for...in, Object.keys(), Object.entries()
- 6. Use destructuring for cleaner code especially in function parameters
- 7. Know the difference between shallow and deep operations
- 8. Practice object manipulation patterns merging, filtering, transforming

Previous Chapter: ← Arrays & Array Methods Next Chapter: Prototypes & Inheritance →

**Practice**: Try the object manipulation problems and experiment with different property access patterns!

# Chapter 8: Objects & Object Access

Master JavaScript objects, the foundation of everything in JavaScript.

## Plain English Explanation

Objects are like containers that hold related information and actions together. Think of them as:

- **Person** = object with properties (name, age, height) and actions (walk, talk, eat)
- Car = object with properties (color, model, year) and actions (start, stop, accelerate)
- **Book** = object with properties (title, author, pages) and actions (open, close, bookmark)
- Bank Account = object with properties (balance, owner) and actions (deposit, withdraw)

In JavaScript, almost everything is an object (except primitives), and objects are the building blocks of complex applications.

## Creating Objects

Object Literal (Most Common)

```
// Empty object
const emptyObj = {};
// Object with properties
const person = {
 name: "Alice",
 age: 30,
 city: "New York",
  isEmployed: true,
};
// Object with methods
const calculator = {
 result: 0,
 add: function (num) {
    this.result += num;
    return this;
  },
  subtract: function (num) {
    this.result -= num;
    return this;
  },
  getValue: function () {
    return this.result;
 },
};
// ES6 method shorthand
const modernCalculator = {
  result: 0,
  add(num) {
    this.result += num;
    return this;
  },
  subtract(num) {
    this.result -= num;
    return this;
  },
  getValue() {
    return this.result;
  },
};
```

```
// Using Object constructor
const obj1 = new Object();
obj1.name = "Alice";
obj1.age = 30;

// Equivalent to object literal
const obj2 = {
   name: "Alice",
   age: 30,
};
```

## Object.create()

```
// Create object with specific prototype
const personPrototype = {
  greet: function () {
    return `Hello, I'm ${this.name}`;
  },
};

const alice = Object.create(personPrototype);
alice.name = "Alice";
alice.age = 30;

console.log(alice.greet()); // "Hello, I'm Alice"

// Create object with null prototype (no inherited properties)
const pureObject = Object.create(null);
pureObject.name = "Pure";
console.log(pureObject.toString); // undefined (no inherited methods)
```

#### **Constructor Functions**

```
// Constructor function (before ES6 classes)
function Person(name, age, city) {
    this.name = name;
    this.age = age;
    this.city = city;

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

// Create instances
const alice = new Person("Alice", 30, "New York");
const bob = new Person("Bob", 25, "London");
```

```
console.log(alice.greet()); // "Hello, I'm Alice from New York"
console.log(bob.greet()); // "Hello, I'm Bob from London"
```

#### **ES6 Classes**

```
class Person {
 constructor(name, age, city) {
   this.name = name;
   this.age = age;
   this.city = city;
  }
 greet() {
   return `Hello, I'm ${this.name} from ${this.city}`;
 getAge() {
  return this.age;
  }
 // Static method
 static createAnonymous() {
    return new Person("Anonymous", 0, "Unknown");
 }
}
const alice = new Person("Alice", 30, "New York");
console.log(alice.greet());
const anon = Person.createAnonymous();
console.log(anon.name); // 'Anonymous'
```

# **Accessing Object Properties**

### **Dot Notation**

```
const person = {
  name: "Alice",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
    zipCode: "10001",
    },
};

// Reading properties
console.log(person.name); // 'Alice'
console.log(person.age); // 30
```

```
console.log(person.address.city); // 'New York'

// Setting properties
person.name = "Alice Smith";
person.email = "alice@example.com"; // Adding new property
person.address.country = "USA"; // Adding nested property

console.log(person.email); // 'alice@example.com'
```

#### **Bracket Notation**

```
const person = {
 name: "Alice",
  age: 30,
  "favorite color": "blue", // Property with space
  123: "numeric key", // Numeric string key
};
// Reading with bracket notation
console.log(person["name"]); // 'Alice'
console.log(person["favorite color"]); // 'blue'
console.log(person["123"]); // 'numeric key'
// Dynamic property access
const propertyName = "age";
console.log(person[propertyName]); // 30
// Setting with bracket notation
person["last name"] = "Smith";
person[propertyName] = 31;
// Computed property names
const prefix = "user";
const obj = {
  [prefix + "Name"]: "Alice", // userName: 'Alice'
  [prefix + "Age"]: 30, // userAge: 30
  [`${prefix}Email`]: "alice@example.com", // userEmail: 'alice@example.com'
};
console.log(obj.userName); // 'Alice'
console.log(obj.userAge); // 30
console.log(obj.userEmail); // 'alice@example.com'
```

#### When to Use Dot vs Bracket Notation

```
const obj = {
  name: "Alice",
  "first-name": "Alice",
  123: "number",
```

```
"user name": "alice123",
};
// 
Use dot notation when:
// - Property name is a valid identifier
// - Property name is known at write time
console.log(obj.name);
// Use bracket notation when:
// - Property name has spaces, hyphens, or special characters
console.log(obj["first-name"]);
console.log(obj["user name"]);
// - Property name starts with a number
console.log(obj["123"]);
// - Property name is dynamic/computed
const propName = "name";
console.log(obj[propName]);
// - Property name comes from a variable
function getValue(object, key) {
 return object[key];
}
console.log(getValue(obj, "name")); // 'Alice'
```

# Object Methods and this

#### Method Definition

```
const person = {
 name: "Alice",
 age: 30,
 // Method using function keyword
 greet: function () {
  return `Hello, I'm ${this.name}`;
 },
 // ES6 method shorthand
 introduce() {
   return `I'm ${this.name} and I'm ${this.age} years old`;
 },
 // Arrow function (be careful with 'this'!)
 arrowMethod: () => {
   // 'this' doesn't refer to the object!
   return `Hello from ${this.name}`; // undefined
 },
```

```
// Method that modifies object
haveBirthday() {
    this.age++;
    return `Happy birthday! Now I'm ${this.age}`;
    },
};

console.log(person.greet()); // "Hello, I'm Alice"
console.log(person.introduce()); // "I'm Alice and I'm 30 years old"
console.log(person.arrowMethod()); // "Hello from undefined"
console.log(person.haveBirthday()); // "Happy birthday! Now I'm 31"
```

## Understanding this Context

```
const person = {
 name: "Alice",
 greet() {
  return `Hello, I'm ${this.name}`;
 },
};
// Method called on object - 'this' refers to person
console.log(person.greet()); // "Hello, I'm Alice"
// Method assigned to variable - 'this' context lost
const greetFunction = person.greet;
console.log(greetFunction()); // "Hello, I'm undefined"
// Binding 'this' context
const boundGreet = person.greet.bind(person);
console.log(boundGreet()); // "Hello, I'm Alice"
// Using call() and apply()
const anotherPerson = { name: "Bob" };
console.log(person.greet.call(anotherPerson)); // "Hello, I'm Bob"
console.log(person.greet.apply(anotherPerson)); // "Hello, I'm Bob"
```

#### Method Chaining

```
const calculator = {
  value: 0,

add(num) {
   this.value += num;
   return this; // Return 'this' for chaining
  },

subtract(num) {
   this.value -= num;
}
```

```
return this;
  },
  multiply(num) {
   this.value *= num;
   return this;
  },
  divide(num) {
   if (num !== 0) {
     this.value /= num;
   return this;
  },
  getValue() {
  return this.value;
 },
 reset() {
   this.value = 0;
  return this;
 },
};
// Method chaining
const result = calculator.add(10).multiply(2).subtract(5).divide(3).getValue();
console.log(result); // 5
// Reset and chain again
calculator.reset().add(100).subtract(50).multiply(2);
console.log(calculator.getValue()); // 100
```

## Object Property Descriptors

## **Property Descriptors**

```
const person = {
  name: "Alice",
  age: 30,
};

// Get property descriptor
  const nameDescriptor = Object.getOwnPropertyDescriptor(person, "name");
  console.log(nameDescriptor);

// {
  // value: 'Alice',
  // writable: true,
  // enumerable: true,
  // configurable: true
```

```
// }
// Define property with custom descriptor
Object.defineProperty(person, "id", {
  value: 12345,
 writable: false, // Cannot be changed
  enumerable: false, // Won't show in for...in or Object.keys()
  configurable: false, // Cannot be deleted or reconfigured
});
console.log(person.id); // 12345
person.id = 54321; // Silently fails (or throws in strict mode)
console.log(person.id); // 12345 (unchanged)
console.log(Object.keys(person)); // ['name', 'age'] (id not enumerable)
// Define multiple properties
Object.defineProperties(person, {
  firstName: {
    value: "Alice",
    writable: true,
    enumerable: true,
    configurable: true,
  },
  lastName: {
    value: "Smith",
    writable: true,
    enumerable: true,
    configurable: true,
  },
});
```

#### **Getters and Setters**

```
const person = {
  firstName: "Alice",
  lastName: "Smith",

  // Getter - computed property
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },

  // Setter - validates and sets value
  set fullName(value) {
    const parts = value.split(" ");
    if (parts.length === 2) {
        this.firstName = parts[0];
        this.lastName = parts[1];
    } else {
        throw new Error('Full name must be in format "First Last"');
```

```
},
 // Private-like property with getter/setter
  _age: 30,
 get age() {
   return this._age;
  },
  set age(value) {
    if (value < 0 || value > 150) {
     throw new Error("Age must be between 0 and 150");
   this._age = value;
  },
};
// Using getters and setters
console.log(person.fullName); // "Alice Smith"
person.fullName = "Bob Johnson";
console.log(person.firstName); // "Bob"
console.log(person.lastName); // "Johnson"
console.log(person.age); // 30
person.age = 25;
console.log(person.age); // 25
// This will throw an error
// person.age = -5; // Error: Age must be between 0 and 150
// Define getter/setter with Object.defineProperty
const obj = {};
Object.defineProperty(obj, "temperature", {
  get() {
    return this._celsius;
 },
  set(celsius) {
   this._celsius = celsius;
   this._fahrenheit = (celsius * 9) / 5 + 32;
  enumerable: true,
  configurable: true,
});
obj.temperature = 25;
console.log(obj.temperature); // 25
console.log(obj._fahrenheit); // 77
```

## Object Iteration

for...in Loop

```
const person = {
 name: "Alice",
 age: 30,
 city: "New York",
};
// Iterate over enumerable properties
for (const key in person) {
 console.log(`${key}: ${person[key]}`);
// Output:
// name: Alice
// age: 30
// city: New York
// Check if property belongs to object (not inherited)
for (const key in person) {
 if (person.hasOwnProperty(key)) {
    console.log(`${key}: ${person[key]}`);
  }
}
```

## Object.keys(), Object.values(), Object.entries()

```
const person = {
 name: "Alice",
 age: 30,
 city: "New York",
};
// Get all keys
const keys = Object.keys(person);
console.log(keys); // ['name', 'age', 'city']
// Get all values
const values = Object.values(person);
console.log(values); // ['Alice', 30, 'New York']
// Get key-value pairs
const entries = Object.entries(person);
console.log(entries); // [['name', 'Alice'], ['age', 30], ['city', 'New York']]
// Iterate using these methods
Object.keys(person).forEach((key) => {
 console.log(`${key}: ${person[key]}`);
});
Object.entries(person).forEach(([key, value]) => {
 console.log(`${key}: ${value}`);
});
```

```
// Convert back to object
const newObj = Object.fromEntries(entries);
console.log(newObj); // { name: 'Alice', age: 30, city: 'New York' }
```

## **%** Object Utility Methods

### Object.assign() - Copying and Merging

```
const target = { a: 1, b: 2 };
const source1 = { b: 3, c: 4 };
const source2 = { c: 5, d: 6 };
// Merge objects (modifies target)
Object.assign(target, source1, source2);
console.log(target); // { a: 1, b: 3, c: 5, d: 6 }
// Create new object without modifying original
const original = { name: "Alice", age: 30 };
const copy = Object.assign({}, original, { city: "New York" });
console.log(copy); // { name: 'Alice', age: 30, city: 'New York' }
console.log(original); // { name: 'Alice', age: 30 } (unchanged)
// Shallow copy limitation
const obj1 = {
 name: "Alice",
 address: { city: "New York", zip: "10001" },
};
const obj2 = Object.assign({}, obj1);
obj2.address.city = "Boston"; // Modifies original!
console.log(obj1.address.city); // 'Boston' (not 'New York'!)
```

### Spread Operator (ES6)

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const obj3 = { c: 5, d: 6 };

// Merge objects with spread
const merged = { ...obj1, ...obj2, ...obj3 };
console.log(merged); // { a: 1, b: 3, c: 5, d: 6 }

// Add properties while copying
const person = { name: "Alice", age: 30 };
const employee = {
    ...person,
    id: 12345,
    department: "Engineering",
```

```
age: 31, // Override existing property
};
console.log(employee); // { name: 'Alice', age: 31, id: 12345, department:
'Engineering' }

// Conditional properties
const includeEmail = true;
const user = {
  name: "Alice",
  age: 30,
  ...(includeEmail && { email: "alice@example.com" }),
};
console.log(user); // { name: 'Alice', age: 30, email: 'alice@example.com' }
```

### Object.freeze(), Object.seal(), Object.preventExtensions()

```
const obj = { name: "Alice", age: 30 };
// Object.freeze() - completely immutable
const frozen = Object.freeze({ ...obj });
frozen.name = "Bob"; // Silently fails
frozen.city = "New York"; // Silently fails
delete frozen.age; // Silently fails
console.log(frozen); // { name: 'Alice', age: 30 } (unchanged)
console.log(Object.isFrozen(frozen)); // true
// Object.seal() - can modify existing properties, but can't add/delete
const sealed = Object.seal({ ...obj });
sealed.name = "Bob"; // 
Works
sealed.city = "New York"; // X Silently fails
delete sealed.age; // X Silently fails
console.log(sealed); // { name: 'Bob', age: 30 }
console.log(Object.isSealed(sealed)); // true
// Object.preventExtensions() - can modify and delete, but can't add
const nonExtensible = Object.preventExtensions({ ...obj });
nonExtensible.name = "Bob"; // Works
nonExtensible.city = "New York"; // X Silently fails
delete nonExtensible.age; // ✓ Works
console.log(nonExtensible); // { name: 'Bob' }
console.log(Object.isExtensible(nonExtensible)); // false
```

## Object Comparison and Checking

#### **Checking Properties**

```
const person = {
  name: "Alice",
  age: 30,
```

```
address: {
    city: "New York",
 },
};
// Check if property exists
console.log("name" in person); // true
console.log("email" in person); // false
console.log("toString" in person); // true (inherited)
// Check own property (not inherited)
console.log(person.hasOwnProperty("name")); // true
console.log(person.hasOwnProperty("toString")); // false
// Modern alternative to hasOwnProperty
console.log(Object.hasOwn(person, "name")); // true (ES2022)
console.log(Object.hasOwn(person, "toString")); // false
// Check if property is undefined
console.log(person.name !== undefined); // true
console.log(person.email !== undefined); // false
// Nested property checking
console.log(person.address && person.address.city); // 'New York'
console.log(person.address && person.address.zip); // undefined
// Optional chaining (ES2020)
console.log(person.address?.city); // 'New York'
console.log(person.address?.zip); // undefined
console.log(person.contact?.email); // undefined (no error)
```

## **Object Comparison**

```
// Objects are compared by reference, not value
const obj1 = { name: "Alice" };
const obj2 = { name: "Alice" };
const obj3 = obj1;

console.log(obj1 === obj2); // false (different objects)
console.log(obj1 === obj3); // true (same reference)

// Shallow equality check
function shallowEqual(obj1, obj2) {
  const keys1 = Object.keys(obj1);
  const keys2 = Object.keys(obj2);

  if (keys1.length !== keys2.length) {
    return false;
  }

  for (let key of keys1) {
```

```
if (obj1[key] !== obj2[key]) {
      return false;
    }
  }
 return true;
}
console.log(shallowEqual(obj1, obj2)); // true
// Deep equality check (simplified)
function deepEqual(obj1, obj2) {
  if (obj1 === obj2) return true;
 if (obj1 == null || obj2 == null) return false;
  if (typeof obj1 !== "object" || typeof obj2 !== "object") {
    return obj1 === obj2;
  const keys1 = Object.keys(obj1);
 const keys2 = Object.keys(obj2);
 if (keys1.length !== keys2.length) return false;
 for (let key of keys1) {
   if (!keys2.includes(key)) return false;
   if (!deepEqual(obj1[key], obj2[key])) return false;
  }
 return true;
}
const deep1 = { a: 1, b: { c: 2 } };
const deep2 = { a: 1, b: { c: 2 } };
console.log(deepEqual(deep1, deep2)); // true
```

# **Destructuring Objects**

#### **Basic Destructuring**

```
const person = {
  name: "Alice",
  age: 30,
  city: "New York",
  country: "USA",
};

// Basic destructuring
const { name, age } = person;
console.log(name, age); // 'Alice', 30
```

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

// Default values
const { name, email = "No email" } = person;
console.log(name, email); // 'Alice', 'No email'

// Rest operator
const { name: personName, ...rest } = person;
console.log(personName); // 'Alice'
console.log(rest); // { age: 30, city: 'New York', country: 'USA' }
```

#### **Nested Destructuring**

```
const user = {
 id: 1,
 name: "Alice",
  address: {
   street: "123 Main St",
   city: "New York",
    coordinates: {
     lat: 40.7128,
     lng: -74.006,
   },
  },
 preferences: {
   theme: "dark",
   language: "en",
 },
};
// Nested destructuring
const {
 name,
  address: {
   city,
   coordinates: { lat, lng },
 },
 preferences: { theme },
} = user;
console.log(name, city, lat, lng, theme);
// 'Alice', 'New York', 40.7128, -74.0060, 'dark'
// Destructuring with default values for nested properties
const { address: { zipCode = "Unknown" } = {} } = user;
console.log(zipCode); // 'Unknown'
```

#### **Function Parameter Destructuring**

```
// Function with object parameter
function greetUser({ name, age, city = "Unknown" }) {
 return `Hello ${name}, age ${age} from ${city}`;
}
const user = { name: "Alice", age: 30 };
console.log(greetUser(user)); // "Hello Alice, age 30 from Unknown"
// Nested destructuring in parameters
function processOrder({
  id,
  customer: { name, email },
 items = [],
 shipping: { address, method = "standard" } = {},
}) {
 return {
   orderId: id,
    customerName: name,
    customerEmail: email,
    itemCount: items.length,
    shippingAddress: address,
    shippingMethod: method,
 };
}
const order = {
 id: "ORD-123",
 customer: {
    name: "Alice",
    email: "alice@example.com",
 },
 items: ["item1", "item2"],
  shipping: {
    address: "123 Main St",
 },
};
console.log(processOrder(order));
```

## 

#### 1. Reference vs Value

```
// Objects are passed by reference
const original = { name: "Alice", age: 30 };
const reference = original;
reference.age = 31;
```

```
console.log(original.age); // 31 (original is modified!)
// Create actual copy
const copy = { ...original };
copy.age = 32;
console.log(original.age); // 31 (original unchanged)
// Shallow copy limitation
const obj = {
 name: "Alice",
 hobbies: ["reading", "swimming"],
};
const shallowCopy = { ...obj };
shallowCopy.hobbies.push("cooking");
console.log(obj.hobbies); // ['reading', 'swimming', 'cooking'] (modified!)
// Deep copy (simple approach)
const deepCopy = JSON.parse(JSON.stringify(obj));
// Note: This doesn't work with functions, dates, undefined, etc.
```

#### 2. this Context Issues

```
const person = {
 name: "Alice",
  greet: function () {
   return `Hello, I'm ${this.name}`;
 },
  arrowGreet: () => {
   return `Hello, I'm ${this.name}`; // 'this' is not the object!
 },
};
console.log(person.greet()); // "Hello, I'm Alice"
console.log(person.arrowGreet()); // "Hello, I'm undefined"
// Method assignment loses context
const greetFunc = person.greet;
console.log(greetFunc()); // "Hello, I'm undefined"
// Solutions:
// 1. Bind the method
const boundGreet = person.greet.bind(person);
console.log(boundGreet()); // "Hello, I'm Alice"
// 2. Use arrow function wrapper
const wrappedGreet = () => person.greet();
console.log(wrappedGreet()); // "Hello, I'm Alice"
```

#### 3. Property Existence Checking

```
const obj = {
 name: "Alice",
 age: ∅,
 active: false,
 data: null,
};
// X Wrong ways to check property existence
if (obj.age) {
  /* Won't execute because 0 is falsy */
}
if (obj.active) {
 /* Won't execute because false is falsy */
if (obj.data) {
 /* Won't execute because null is falsy */
}
// Correct ways
if ("age" in obj) {
 /* Correct */
}
if (obj.hasOwnProperty("age")) {
 /* Correct */
}
if (obj.age !== undefined) {
 /* Correct */
}
if (typeof obj.age !== "undefined") {
 /* Correct */
}
```

## 4. Deleting Properties

```
// Alternative: destructuring to exclude
const { temp, ...newObj } = obj;
console.log(newObj); // { name: 'Alice', age: 30 }
```

## Mini Practice Problems

## Problem 1: Object Manipulation

```
// Create a function that takes an array of objects and returns
// a new object with specific transformations
const users = [
 { id: 1, name: "Alice", age: 30, active: true },
 { id: 2, name: "Bob", age: 25, active: false },
 { id: 3, name: "Charlie", age: 35, active: true },
];
// Your function should return:
// {
// activeUsers: ['Alice', 'Charlie'],
// averageAge: 30,
// userById: { 1: 'Alice', 2: 'Bob', 3: 'Charlie' }
// }
function processUsers(users) {
 // Your code here
console.log(processUsers(users));
```

## Problem 2: Deep Object Merging

```
// Implement a deep merge function that combines objects recursively

function deepMerge(target, source) {
    // Your implementation here
}

const obj1 = {
    a: 1,
    b: {
        c: 2,
        d: 3,
    },
    e: [1, 2],
};

const obj2 = {
    b: {
```

```
d: 4,
    f: 5,
},
e: [3, 4],
g: 6,
};

// Should result in:
// {
    // a: 1,
    // b: { c: 2, d: 4, f: 5 },
    // e: [3, 4],
    // g: 6
    // }

console.log(deepMerge(obj1, obj2));
```

### Problem 3: Object Path Access

```
// Create a function that safely accesses nested object properties
// using a string path like 'user.address.city'
function getNestedValue(obj, path, defaultValue = undefined) {
 // Your implementation here
}
const data = {
 user: {
    name: "Alice",
    address: {
      city: "New York",
     coordinates: {
       lat: 40.7128,
      },
   },
 },
};
console.log(getNestedValue(data, "user.name")); // 'Alice'
console.log(getNestedValue(data, "user.address.city")); // 'New York'
console.log(getNestedValue(data, "user.address.coordinates.lat")); // 40.7128
console.log(getNestedValue(data, "user.email", "No email")); // 'No email'
console.log(getNestedValue(data, "user.address.zip")); // undefined
```

### Problem 4: Object Validation

```
// Create a validation system for objects
class ObjectValidator {
```

```
constructor(schema) {
    this.schema = schema;
  }
  validate(obj) {
   // Your implementation here
   // Return { valid: boolean, errors: string[] }
 }
}
// Usage example:
const userSchema = {
  name: { type: "string", required: true },
  age: { type: "number", required: true, min: 0, max: 150 },
 email: { type: "string", required: false, pattern: /^[^@]+@[^@]+\.[^@]+$/ },
};
const validator = new ObjectValidator(userSchema);
const validUser = { name: "Alice", age: 30, email: "alice@example.com" };
const invalidUser = { name: "", age: -5, email: "invalid-email" };
console.log(validator.validate(validUser)); // { valid: true, errors: [] }
console.log(validator.validate(invalidUser)); // { valid: false, errors: [...] }
```

## Problem 5: Object Proxy

```
// Create a proxy that logs all property access and modifications

function createLoggingProxy(obj, logFunction = console.log) {
    // Your implementation using Proxy
}

const person = { name: "Alice", age: 30 };

const loggedPerson = createLoggingProxy(person);

// These operations should log to console:
loggedPerson.name; // "GET: name = Alice"
loggedPerson.age = 31; // "SET: age = 31"
loggedPerson.city = "NYC"; // "SET: city = NYC"
delete loggedPerson.age; // "DELETE: age"
```

## Interview Notes

#### **Common Questions:**

#### Q: What's the difference between dot notation and bracket notation?

- Dot notation: obj.property for valid identifiers, known at write time
- Bracket notation: obj['property'] for dynamic access, special characters, computed properties

#### Q: How do you check if a property exists in an object?

- 'property' in obj checks own and inherited properties
- obj.hasOwnProperty('property') checks only own properties
- Object.hasOwn(obj, 'property') modern alternative (ES2022)
- obj.property !== undefined checks if value is not undefined

#### Q: What's the difference between Object.assign() and spread operator?

- Both perform shallow copying/merging
- Spread operator is more concise and readable
- Object.assign() modifies the target object, spread creates new object
- Spread operator doesn't work with getters/setters the same way

#### Q: How do you deep copy an object?

- JSON.parse(JSON.stringify(obj)) simple but limited (no functions, dates, etc.)
- Lodash cloneDeep() robust library solution
- Custom recursive function for specific needs
- structuredClone() new native method (limited browser support)

## **Asked at Companies:**

- Google: "Implement Object.assign() from scratch"
- Facebook: "Create a function to flatten nested objects"
- Amazon: "Implement deep equality comparison for objects"
- Microsoft: "Design a system for object validation and transformation"
- Netflix: "Create a proxy that tracks object property access patterns"

# **©** Key Takeaways

- 1. Objects are reference types understand copying vs referencing
- 2. Use appropriate property access method dot vs bracket notation
- 3. Be careful with this context especially with arrow functions
- 4. Understand property descriptors writable, enumerable, configurable
- 5. Master object iteration methods for...in, Object.keys(), Object.entries()
- 6. **Use destructuring for cleaner code** especially in function parameters
- 7. Know the difference between shallow and deep operations
- 8. Practice object manipulation patterns merging, filtering, transforming

Previous Chapter: ← Arrays & Array Methods Next Chapter: Prototypes & Inheritance →

**Practice**: Try the object manipulation problems and experiment with different property access patterns!

# Chapter 9: Prototypes & Inheritance

Master JavaScript's prototype-based inheritance system and understand how objects inherit from other objects.

## Plain English Explanation

Prototypes are like "blueprints" or "templates" that objects can inherit properties and methods from. Think of them as:

- Family traits = children inherit characteristics from parents
- **Recipe templates** = specific recipes inherit basic cooking methods
- Car models = all Honda Civics share common features from the Civic "prototype"
- Employee types = all managers inherit basic employee properties plus management-specific ones

Unlike class-based languages, JavaScript uses prototype-based inheritance where objects can directly inherit from other objects.



## Understanding Prototypes

### The Prototype Chain

```
// Every object has a prototype (except Object.prototype)
const person = {
 name: "Alice",
  age: 30,
};
// Check the prototype
console.log(Object.getPrototypeOf(person)); // Object.prototype
console.log(person.__proto__); // Object.prototype (deprecated way)
// The prototype chain
console.log(person.toString); // Inherited from Object.prototype
console.log(person.hasOwnProperty); // Inherited from Object.prototype
// Prototype chain visualization:
// person -> Object.prototype -> null
// Arrays have their own prototype chain
const numbers = [1, 2, 3];
console.log(Object.getPrototypeOf(numbers)); // Array.prototype
console.log(numbers.push); // Inherited from Array.prototype
console.log(numbers.toString); // Inherited from Object.prototype
// Array prototype chain:
// numbers -> Array.prototype -> Object.prototype -> null
```

## Creating Objects with Specific Prototypes

```
// Method 1: Object.create()
const animalPrototype = {
 makeSound: function () {
   return `${this.name} makes a sound`;
 },
 eat: function () {
  return `${this.name} is eating`;
 },
};
const dog = Object.create(animalPrototype);
dog.name = "Buddy";
dog.breed = "Golden Retriever";
console.log(dog.makeSound()); // "Buddy makes a sound"
console.log(dog.eat()); // "Buddy is eating"
// Check prototype relationship
console.log(Object.getPrototypeOf(dog) === animalPrototype); // true
console.log(animalPrototype.isPrototypeOf(dog)); // true
// Method 2: Constructor functions
function Animal(name) {
 this.name = name;
}
Animal.prototype.makeSound = function () {
 return `${this.name} makes a sound`;
};
Animal.prototype.eat = function () {
 return `${this.name} is eating`;
};
const cat = new Animal("Whiskers");
console.log(cat.makeSound()); // "Whiskers makes a sound"
console.log(Object.getPrototypeOf(cat) === Animal.prototype); // true
```

#### Prototype Property vs proto

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

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

const alice = new Person("Alice");
```

```
// Constructor function has 'prototype' property
console.log(Person.prototype); // { greet: function, constructor: Person }

// Instance has '__proto__' (or use Object.getPrototypeOf)
console.log(alice.__proto__ === Person.prototype); // true
console.log(Object.getPrototypeOf(alice) === Person.prototype); // true

// The relationship:
// Person.prototype === alice.__proto__
// alice inherits from Person.prototype
```

## **R** Constructor Functions

#### **Basic Constructor Pattern**

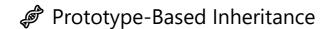
```
// Constructor function (capitalized by convention)
function Person(name, age, city) {
 // Instance properties
 this.name = name;
 this.age = age;
 this.city = city;
}
// Methods on prototype (shared by all instances)
Person.prototype.greet = function () {
 return `Hello, I'm ${this.name} from ${this.city}`;
};
Person.prototype.getAge = function () {
  return this.age;
};
Person.prototype.haveBirthday = function () {
  this.age++;
  return `Happy birthday! Now I'm ${this.age}`;
};
// Create instances
const alice = new Person("Alice", 30, "New York");
const bob = new Person("Bob", 25, "London");
console.log(alice.greet()); // "Hello, I'm Alice from New York"
console.log(bob.greet()); // "Hello, I'm Bob from London"
// Methods are shared (same reference)
console.log(alice.greet === bob.greet); // true
// But instance properties are separate
console.log(alice.name === bob.name); // false
```

#### Constructor Function Patterns

```
// Pattern 1: All methods in constructor (not recommended)
function BadPerson(name) {
 this.name = name;
 // X Each instance gets its own copy of this method
 this.greet = function () {
   return `Hello, I'm ${this.name}`;
 };
}
// Pattern 2: Methods on prototype (recommended)
function GoodPerson(name) {
 this.name = name;
}
// // All instances share this method
GoodPerson.prototype.greet = function () {
 return `Hello, I'm ${this.name}`;
};
// Pattern 3: Prototype object replacement
function BestPerson(name) {
  this.name = name;
}
BestPerson.prototype = {
  constructor: BestPerson, // Important: restore constructor reference
  greet: function () {
    return `Hello, I'm ${this.name}`;
  },
  introduce: function () {
    return `My name is ${this.name}`;
 },
};
// Test memory efficiency
const bad1 = new BadPerson("Alice");
const bad2 = new BadPerson("Bob");
console.log(bad1.greet === bad2.greet); // false (different functions)
const good1 = new GoodPerson("Alice");
const good2 = new GoodPerson("Bob");
console.log(good1.greet === good2.greet); // true (same function)
```

#### The new Operator

```
function Person(name, age) {
 this.name = name;
 this.age = age;
}
Person.prototype.greet = function () {
 return `Hello, I'm ${this.name}`;
};
// What happens when you use 'new':
// 1. Create a new empty object
// 2. Set the object's prototype to the constructor's prototype
// 3. Call the constructor with 'this' bound to the new object
// 4. Return the new object (unless constructor returns an object)
const alice = new Person("Alice", 30);
// Manual simulation of 'new' operator
function createPerson(name, age) {
 // Step 1: Create new object
 const obj = {};
 // Step 2: Set prototype
 Object.setPrototypeOf(obj, Person.prototype);
 // Step 3: Call constructor
  Person.call(obj, name, age);
 // Step 4: Return object
 return obj;
}
const bob = createPerson("Bob", 25);
console.log(bob.greet()); // "Hello, I'm Bob"
// Forgetting 'new' keyword
function SafePerson(name, age) {
 // Guard against missing 'new'
 if (!(this instanceof SafePerson)) {
   return new SafePerson(name, age);
  }
 this.name = name;
 this.age = age;
}
const charlie1 = new SafePerson("Charlie", 35); // With 'new'
const charlie2 = SafePerson("Charlie", 35); // Without 'new'
console.log(charlie1 instanceof SafePerson); // true
console.log(charlie2 instanceof SafePerson); // true
```



#### Basic Inheritance Pattern

```
// Parent constructor
function Animal(name, species) {
 this.name = name;
 this.species = species;
}
Animal.prototype.makeSound = function () {
  return `${this.name} makes a sound`;
};
Animal.prototype.eat = function () {
 return `${this.name} is eating`;
};
Animal.prototype.sleep = function () {
  return `${this.name} is sleeping`;
};
// Child constructor
function Dog(name, breed) {
 // Call parent constructor
 Animal.call(this, name, "Canine");
 this.breed = breed;
}
// Set up inheritance
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog; // Restore constructor reference
// Add child-specific methods
Dog.prototype.bark = function () {
  return `${this.name} barks: Woof!`;
};
// Override parent method
Dog.prototype.makeSound = function () {
 return `${this.name} barks loudly`;
};
// Create instance
const buddy = new Dog("Buddy", "Golden Retriever");
console.log(buddy.name); // 'Buddy'
console.log(buddy.species); // 'Canine'
console.log(buddy.breed); // 'Golden Retriever'
console.log(buddy.eat()); // "Buddy is eating" (inherited)
console.log(buddy.bark()); // "Buddy barks: Woof!" (own method)
console.log(buddy.makeSound()); // "Buddy barks loudly" (overridden)
```

```
// Check inheritance chain
console.log(buddy instanceof Dog); // true
console.log(buddy instanceof Animal); // true
console.log(buddy instanceof Object); // true

// Prototype chain: buddy -> Dog.prototype -> Animal.prototype -> Object.prototype
-> null
```

### Multiple Levels of Inheritance

```
// Grandparent
function LivingBeing(name) {
 this.name = name;
 this.alive = true;
}
LivingBeing.prototype.breathe = function () {
  return `${this.name} is breathing`;
};
// Parent
function Animal(name, species) {
  LivingBeing.call(this, name);
 this.species = species;
}
Animal.prototype = Object.create(LivingBeing.prototype);
Animal.prototype.constructor = Animal;
Animal.prototype.move = function () {
 return `${this.name} is moving`;
};
// Child
function Mammal(name, species, furColor) {
  Animal.call(this, name, species);
  this.furColor = furColor;
 this.warmBlooded = true;
Mammal.prototype = Object.create(Animal.prototype);
Mammal.prototype.constructor = Mammal;
Mammal.prototype.produceMilk = function () {
  return `${this.name} produces milk`;
};
// Grandchild
function Dog(name, breed, furColor) {
  Mammal.call(this, name, "Canine", furColor);
```

```
this.breed = breed;
}
Dog.prototype = Object.create(Mammal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.bark = function () {
 return `${this.name} barks`;
};
// Create instance
const max = new Dog("Max", "German Shepherd", "brown");
console.log(max.breathe()); // "Max is breathing" (from LivingBeing)
console.log(max.move()); // "Max is moving" (from Animal)
console.log(max.produceMilk()); // "Max produces milk" (from Mammal)
console.log(max.bark()); // "Max barks" (from Dog)
// Complex inheritance chain:
// max -> Dog.prototype -> Mammal.prototype -> Animal.prototype ->
LivingBeing.prototype -> Object.prototype -> null
```

#### Mixin Pattern

```
// Mixins for adding functionality
const Flyable = {
  fly: function () {
    return `${this.name} is flying`;
  },
  land: function () {
   return `${this.name} has landed`;
 },
};
const Swimmable = {
  swim: function () {
    return `${this.name} is swimming`;
  },
  dive: function () {
   return `${this.name} is diving`;
 },
};
// Mixin function
function mixin(target, ...sources) {
  sources.forEach((source) => {
    Object.getOwnPropertyNames(source).forEach((name) => {
      if (name !== "constructor") {
        target[name] = source[name];
      }
    });
```

```
});
 return target;
}
// Base class
function Bird(name, species) {
 this.name = name;
 this.species = species;
}
Bird.prototype.chirp = function () {
 return `${this.name} chirps`;
};
// Add flying ability
mixin(Bird.prototype, Flyable);
// Duck can fly and swim
function Duck(name) {
 Bird.call(this, name, "Duck");
}
Duck.prototype = Object.create(Bird.prototype);
Duck.prototype.constructor = Duck;
// Add swimming ability to Duck
mixin(Duck.prototype, Swimmable);
Duck.prototype.quack = function () {
 return `${this.name} quacks`;
};
// Create duck instance
const donald = new Duck("Donald");
console.log(donald.chirp()); // "Donald chirps" (from Bird)
console.log(donald.fly()); // "Donald is flying" (from Flyable mixin)
console.log(donald.swim()); // "Donald is swimming" (from Swimmable mixin)
console.log(donald.quack()); // "Donald quacks" (own method)
```

# **©** ES6 Classes (Syntactic Sugar)

### **Basic Class Syntax**

```
// ES6 class (syntactic sugar over prototypes)
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
```

```
greet() {
    return `Hello, I'm ${this.name}`;
  getAge() {
   return this.age;
 haveBirthday() {
   this.age++;
   return `Happy birthday! Now I'm ${this.age}`;
  }
 // Static method
 static createAnonymous() {
    return new Person("Anonymous", 0);
 }
}
// Equivalent to constructor function approach
const alice = new Person("Alice", 30);
console.log(alice.greet()); // "Hello, I'm Alice"
// Static method usage
const anon = Person.createAnonymous();
console.log(anon.name); // 'Anonymous'
// Classes are still functions under the hood
console.log(typeof Person); // 'function'
console.log(Person.prototype.greet); // function
```

#### Class Inheritance with extends

```
// Parent class
class Animal {
  constructor(name, species) {
    this.name = name;
    this.species = species;
}

makeSound() {
    return `${this.name} makes a sound`;
}

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

static getKingdom() {
    return "Animalia";
}
```

```
// Child class
class Dog extends Animal {
  constructor(name, breed) {
    super(name, "Canine"); // Call parent constructor
    this.breed = breed;
  }
  // Override parent method
  makeSound() {
   return `${this.name} barks: Woof!`;
  }
 // New method
  fetch() {
    return `${this.name} fetches the ball`;
  }
  // Call parent method from child
  describe() {
   return `${super.eat()} and ${this.makeSound()}`;
  }
 // Static method
 static getSpecies() {
    return "Canis lupus";
  }
}
// Create instance
const buddy = new Dog("Buddy", "Golden Retriever");
console.log(buddy.name); // 'Buddy'
console.log(buddy.species); // 'Canine'
console.log(buddy.breed); // 'Golden Retriever'
console.log(buddy.makeSound()); // "Buddy barks: Woof!"
console.log(buddy.fetch()); // "Buddy fetches the ball"
console.log(buddy.describe()); // "Buddy is eating and Buddy barks: Woof!"
// Static methods
console.log(Animal.getKingdom()); // 'Animalia'
console.log(Dog.getSpecies()); // 'Canis lupus'
// Inheritance check
console.log(buddy instanceof Dog); // true
console.log(buddy instanceof Animal); // true
```

#### Private Fields and Methods (ES2022)

```
class BankAccount {
 // Private fields (start with #)
  #balance = ∅;
  #accountNumber;
 constructor(accountNumber, initialBalance = 0) {
   this.#accountNumber = accountNumber;
   this.#balance = initialBalance;
  }
  // Public methods
  deposit(amount) {
   if (amount > 0) {
     this.#balance += amount;
      return this.#formatTransaction("deposit", amount);
   throw new Error("Deposit amount must be positive");
  }
  withdraw(amount) {
    if (amount > 0 && amount <= this.#balance) {</pre>
     this.#balance -= amount;
      return this.#formatTransaction("withdrawal", amount);
   throw new Error("Invalid withdrawal amount");
  }
  getBalance() {
   return this.#balance;
  }
  // Private method
  #formatTransaction(type, amount) {
    return `${type.toUpperCase()}: $${amount}. New balance: $${this.#balance}`;
  }
 // Getter
  get accountInfo() {
   return `Account ${this.#accountNumber}: $${this.#balance}`;
  }
}
const account = new BankAccount("12345", 1000);
console.log(account.deposit(500)); // "DEPOSIT: $500. New balance: $1500"
console.log(account.withdraw(200)); // "WITHDRAWAL: $200. New balance: $1300"
console.log(account.accountInfo); // "Account 12345: $1300"
// Private fields are not accessible
// console.log(account.#balance); // SyntaxError
// account.#formatTransaction(); // SyntaxError
```

#### Getters and Setters in Classes

```
class Temperature {
 constructor(celsius = 0) {
   this._celsius = celsius;
  }
 // Getter for Fahrenheit
 get fahrenheit() {
   return (this._celsius * 9) / 5 + 32;
  }
 // Setter for Fahrenheit
  set fahrenheit(value) {
   this._celsius = ((value - 32) * 5) / 9;
  // Getter for Celsius
  get celsius() {
   return this._celsius;
  }
  // Setter for Celsius with validation
  set celsius(value) {
    if (value < -273.15) {
     throw new Error("Temperature cannot be below absolute zero");
   this._celsius = value;
 // Getter for Kelvin
  get kelvin() {
   return this._celsius + 273.15;
  }
 // Setter for Kelvin
 set kelvin(value) {
   this.celsius = value - 273.15; // Uses the celsius setter for validation
  }
}
const temp = new Temperature(25);
console.log(temp.celsius); // 25
console.log(temp.fahrenheit); // 77
console.log(temp.kelvin); // 298.15
// Set temperature using different scales
temp.fahrenheit = 86;
console.log(temp.celsius); // 30
temp.kelvin = 300;
```

```
console.log(temp.celsius); // 26.85

// Validation works
// temp.celsius = -300; // Error: Temperature cannot be below absolute zero
```

## Prototype Manipulation

### **Checking Prototypes**

```
function Animal(name) {
  this.name = name;
Animal.prototype.speak = function () {
 return `${this.name} makes a sound`;
};
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", "Labrador");
// Check prototype relationships
console.log(Object.getPrototypeOf(buddy) === Dog.prototype); // true
console.log(Dog.prototype.isPrototypeOf(buddy)); // true
console.log(Animal.prototype.isPrototypeOf(buddy)); // true
console.log(Object.prototype.isPrototypeOf(buddy)); // true
// Check constructor
console.log(buddy.constructor === Dog); // true
console.log(buddy instanceof Dog); // true
console.log(buddy instanceof Animal); // true
console.log(buddy instanceof Object); // true
// Get prototype chain
function getPrototypeChain(obj) {
  const chain = [];
  let current = obj;
  while (current) {
    chain.push(current.constructor?.name || "Object");
    current = Object.getPrototypeOf(current);
    if (current === Object.prototype) {
      chain.push("Object.prototype");
      break;
    }
```

```
return chain;
}

console.log(getPrototypeChain(buddy)); // ['Dog', 'Animal', 'Object.prototype']
```

### **Dynamic Prototype Modification**

```
function Person(name) {
 this.name = name;
Person.prototype.greet = function () {
 return `Hello, I'm ${this.name}`;
};
const alice = new Person("Alice");
const bob = new Person("Bob");
console.log(alice.greet()); // "Hello, I'm Alice"
// Add method to prototype after instances are created
Person.prototype.introduce = function () {
  return `My name is ${this.name}`;
};
// All instances get the new method
console.log(alice.introduce()); // "My name is Alice"
console.log(bob.introduce()); // "My name is Bob"
// Modify existing method
Person.prototype.greet = function () {
  return `Hi there, I'm ${this.name}!`;
};
console.log(alice.greet()); // "Hi there, I'm Alice!"
// Add property to prototype
Person.prototype.species = "Homo sapiens";
console.log(alice.species); // 'Homo sapiens'
console.log(bob.species); // 'Homo sapiens'
// Instance property shadows prototype property
alice.species = "Human";
console.log(alice.species); // 'Human' (own property)
console.log(bob.species); // 'Homo sapiens' (prototype property)
// Delete instance property to reveal prototype property
delete alice.species;
console.log(alice.species); // 'Homo sapiens' (prototype property again)
```

### Prototype Pollution (Security Concern)

```
// ♠ Prototype pollution example (DON'T DO THIS)
function vulnerableFunction(obj) {
 // Dangerous: modifying Object.prototype
 if (obj.constructor && obj.constructor.prototype) {
    obj.constructor.prototype.isAdmin = true;
 }
}
// This affects ALL objects
const user = {};
vulnerableFunction({ constructor: Object });
console.log(user.isAdmin); // true (BAD!)
console.log({}.isAdmin); // true (BAD!)
// ✓ Safe alternatives:
// 1. Use Object.create(null) for data objects
const safeData = Object.create(null);
safeData.name = "Alice";
console.log(safeData.toString); // undefined (no inherited methods)
// 2. Use Map for key-value storage
const dataMap = new Map();
dataMap.set("name", "Alice");
dataMap.set("age", 30);
// 3. Validate input and use Object.hasOwnProperty
function safeFunction(obj) {
 if (obj && typeof obj === "object" && obj.hasOwnProperty("name")) {
   return obj.name;
  }
 return null;
```

### 

### 1. Forgetting to Set Constructor

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

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

### 2. Modifying Built-in Prototypes

```
// X Don't modify built-in prototypes (can break other code)
Array.prototype.last = function () {
 return this[this.length - 1];
};
// This affects ALL arrays in your application
const numbers = [1, 2, 3];
console.log(numbers.last()); // 3
// ✓ Better: create utility functions or extend in your own classes
class ExtendedArray extends Array {
  last() {
    return this[this.length - 1];
 first() {
    return this[0];
  }
}
const extNumbers = new ExtendedArray(1, 2, 3);
console.log(extNumbers.last()); // 3
```

### 3. Prototype vs Instance Properties

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

// X Wrong: array is shared among all instances
Person.prototype.hobbies = [];
```

```
const alice = new Person("Alice");
const bob = new Person("Bob");
alice.hobbies.push("reading");
console.log(bob.hobbies); // ['reading'] (shared!)
// ✓ Correct: initialize arrays in constructor
function BetterPerson(name) {
 this.name = name;
 this.hobbies = []; // Each instance gets its own array
}
BetterPerson.prototype.addHobby = function (hobby) {
 this.hobbies.push(hobby);
};
const charlie = new BetterPerson("Charlie");
const diana = new BetterPerson("Diana");
charlie.addHobby("swimming");
console.log(diana.hobbies); // [] (separate arrays)
```

#### 4. Arrow Functions and this

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

// X Wrong: arrow function doesn't bind 'this'
Person.prototype.greet = () => {
   return `Hello, I'm ${this.name}`; // 'this' is not the instance!
};

const alice = new Person("Alice");
console.log(alice.greet()); // "Hello, I'm undefined"

// ✓ Correct: use regular function
Person.prototype.greet = function () {
   return `Hello, I'm ${this.name}`;
};

console.log(alice.greet()); // "Hello, I'm Alice"
```

# Mini Practice Problems

### Problem 1: Shape Hierarchy

```
// Create a shape hierarchy with inheritance
// Base Shape class with area() and perimeter() methods
```

```
// Rectangle and Circle classes that extend Shape
// Square class that extends Rectangle
// Your implementation here:
class Shape {
 // Base implementation
class Rectangle extends Shape {
 // Rectangle implementation
class Circle extends Shape {
 // Circle implementation
}
class Square extends Rectangle {
 // Square implementation
// Test your implementation:
const rect = new Rectangle(5, 3);
const circle = new Circle(4);
const square = new Square(4);
console.log(rect.area()); // 15
console.log(circle.area()); // ~50.27
console.log(square.area()); // 16
console.log(square.perimeter()); // 16
```

### **Problem 2: Mixin Implementation**

```
// Implement a mixin system that allows multiple inheritance
const Flyable = {
 fly() {
   return `${this.name} is flying`;
  },
 land() {
   return `${this.name} has landed`;
 },
};
const Swimmable = {
  swim() {
   return `${this.name} is swimming`;
 },
  dive() {
    return `${this.name} is diving`;
 },
};
```

```
const Walkable = {
 walk() {
   return `${this.name} is walking`;
  },
 run() {
  return `${this.name} is running`;
 },
};
// Implement mixin function
function mixin(target, ...sources) {
 // Your implementation here
}
// Base class
class Animal {
 constructor(name) {
   this.name = name;
}
// Create Duck class that can fly, swim, and walk
class Duck extends Animal {
 constructor(name) {
   super(name);
 }
}
// Apply mixins
mixin(Duck.prototype, Flyable, Swimmable, Walkable);
const donald = new Duck("Donald");
console.log(donald.fly()); // "Donald is flying"
console.log(donald.swim()); // "Donald is swimming"
console.log(donald.walk()); // "Donald is walking"
```

### Problem 3: Custom instanceof

```
// Implement your own version of instanceof operator

function myInstanceof(obj, constructor) {
    // Your implementation here
    // Should return true if obj is an instance of constructor
    // Should work with inheritance chain
}

// Test cases:
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", "Labrador");

console.log(myInstanceof(buddy, Dog)); // true
   console.log(myInstanceof(buddy, Animal)); // true
   console.log(myInstanceof(buddy, Object)); // true
   console.log(myInstanceof(buddy, Array)); // false
```

### Problem 4: Prototype Chain Visualizer

```
// Create a function that visualizes the prototype chain
function visualizePrototypeChain(obj) {
 // Your implementation here
  // Should return an array of constructor names in the chain
 // Example: ['Dog', 'Animal', 'Object']
}
// Test with complex inheritance
class LivingBeing {
 constructor(name) {
    this.name = name;
  }
}
class Animal extends LivingBeing {
  constructor(name, species) {
    super(name);
    this.species = species;
}
class Mammal extends Animal {
  constructor(name, species, furColor) {
   super(name, species);
   this.furColor = furColor;
 }
}
class Dog extends Mammal {
  constructor(name, breed) {
    super(name, "Canine", "brown");
    this.breed = breed;
```

```
}
}
const max = new Dog("Max", "German Shepherd");
console.log(visualizePrototypeChain(max));
// Should output: ['Dog', 'Mammal', 'Animal', 'LivingBeing', 'Object']
```

### Problem 5: Safe Object Creation

```
// Create a factory function that creates objects with safe prototypes
// to prevent prototype pollution
function createSafeObject(properties = {}) {
 // Your implementation here
 // Should create object without inheriting from Object.prototype
  // Should still allow setting and getting properties
 // Should prevent prototype pollution
// Test your implementation:
const safeObj = createSafeObject({ name: "Alice", age: 30 });
console.log(safeObj.name); // 'Alice'
console.log(safeObj.age); // 30
console.log(safeObj.toString); // undefined (no inherited methods)
console.log(safeObj.hasOwnProperty); // undefined
// Should not affect other objects
const normalObj = {};
console.log(normalObj.toString); // function (should still work)
```

### Interview Notes

### Common Questions:

### Q: What is the prototype chain?

- Chain of objects linked through their prototypes
- When property is accessed, JavaScript looks up the chain
- Ends at Object.prototype (which has null prototype)
- Enables inheritance in JavaScript

#### Q: Difference between proto and prototype?

- \_\_proto\_\_: property of instances, points to constructor's prototype
- prototype: property of constructor functions, template for instances
- obj.\_\_proto\_\_ === Constructor.prototype

### Q: How does new operator work?

- 1. Creates new empty object
- 2. Sets object's prototype to constructor's prototype
- 3. Calls constructor with this bound to new object
- 4. Returns the object (unless constructor returns object)

### Q: What's the difference between classical and prototypal inheritance?

- Classical: classes inherit from classes (Java, C++)
- Prototypal: objects inherit from objects (JavaScript)
- JavaScript ES6 classes are syntactic sugar over prototypes

#### Q: How do you implement inheritance in JavaScript?

- Constructor functions + Object.create()
- ES6 classes with extends
- Object.create() for direct object inheritance
- Mixins for multiple inheritance

### Asked at Companies:

- Google: "Implement your own version of Object.create()"
- Facebook: "Explain prototype pollution and how to prevent it"
- Amazon: "Create a mixin system for multiple inheritance"
- Microsoft: "Implement classical inheritance using prototypes"
- Netflix: "Design a safe object creation system"

# **6** Key Takeaways

- 1. **Understand the prototype chain** foundation of JavaScript inheritance
- 2. Know constructor functions vs ES6 classes classes are syntactic sugar
- 3. Master inheritance patterns Object.create(), extends, mixins
- 4. Be aware of prototype pollution security and stability concerns
- 5. **Use appropriate inheritance method** composition over inheritance when possible
- 6. **Understand this binding** especially with arrow functions
- 7. Know when to use prototypes vs instances shared vs individual properties
- 8. Practice inheritance hierarchies real-world object modeling

Previous Chapter: ← Objects & Object Access Next Chapter: Asynchronous JavaScript →

**Practice**: Try implementing the inheritance problems and experiment with different prototype patterns!

# // Chapter 11: Error Handling & Debugging

Master error handling, debugging techniques, and building robust JavaScript applications.

### Plain English Explanation

Error handling is like having safety nets and emergency procedures:

- **Smoke detectors** = catching errors before they cause damage
- First aid kit = having tools ready to handle problems
- **Emergency exits** = graceful ways to handle failures
- Security cameras = logging and monitoring to understand what went wrong
- **Building codes** = following best practices to prevent errors

In programming, errors are inevitable. Good error handling makes your applications resilient, user-friendly, and easier to debug.

### Types of Errors

### **Syntax Errors**

```
// X Syntax errors - code won't run at all
function badSyntax() {
   console.log('Hello World'; // Missing closing parenthesis

   if (true {
      console.log('Missing closing parenthesis');
   }

   const obj = {
      name: 'Alice'
      age: 30 // Missing comma
   };
}

// These are caught by the JavaScript parser before execution
```

### Reference Errors

```
// X ReferenceError - using undefined variables
function referenceErrors() {
   console.log(undefinedVariable); // ReferenceError: undefinedVariable is not
   defined

   someFunction(); // ReferenceError: someFunction is not defined

   // Accessing variables before declaration (temporal dead zone)
   console.log(myLet); // ReferenceError
   let myLet = "Hello";
}

// Menction avoidReferenceErrors() {
   // Check if variable exists
   if (typeof someVariable !== "undefined") {
```

```
console.log(someVariable);
}

// Or use optional chaining for object properties
console.log(window.someProperty?.value);

// Declare variables before use
let myVar = "Hello";
console.log(myVar);
}
```

### Type Errors

```
// ★ TypeError - wrong type operations
function typeErrors() {
  const num = 42;
 num.toUpperCase(); // TypeError: num.toUpperCase is not a function
 const str = "Hello";
 str.push("World"); // TypeError: str.push is not a function
 null.property; // TypeError: Cannot read property 'property' of null
 const obj = {};
 obj.method(); // TypeError: obj.method is not a function
}
// How to avoid
function avoidTypeErrors() {
  const num = 42;
 // Check type before operation
 if (typeof num === "string") {
   console.log(num.toUpperCase());
 }
 // Check if method exists
 const obj = {};
 if (typeof obj.method === "function") {
   obj.method();
 }
 // Use optional chaining
 const user = null;
 console.log(user?.name?.toUpperCase?.()); // undefined, no error
 // Type checking with instanceof
 const arr = [];
 if (arr instanceof Array) {
    arr.push("item");
```

```
}
}
```

### Range Errors

```
// X RangeError - values outside valid range
function rangeErrors() {
 // Array with invalid length
 new Array(-1); // RangeError: Invalid array length
 // Number methods with invalid parameters
 const num = 123.456;
 num.toFixed(-1); // RangeError: toFixed() digits argument must be between 0 and
100
 // Stack overflow (infinite recursion)
 function infiniteRecursion() {
   return infiniteRecursion(); // RangeError: Maximum call stack size exceeded
 }
 infiniteRecursion();
}
// // How to avoid
function avoidRangeErrors() {
 // Validate array length
 function createArray(length) {
    if (length < 0 || length > Number.MAX_SAFE_INTEGER) {
      throw new Error("Invalid array length");
    return new Array(length);
  }
 // Validate number method parameters
 function safeToFixed(num, digits) {
   if (digits < 0 || digits > 100) {
      digits = 2; // Default value
    return num.toFixed(digits);
  }
 // Prevent infinite recursion
  function safeRecursion(n, depth = 0) {
   if (depth > 1000) {
     // Safety limit
     throw new Error("Maximum recursion depth exceeded");
    }
   if (n <= 1) return 1;
    return n * safeRecursion(n - 1, depth + 1);
 }
}
```

#### **Custom Errors**

```
// Creating custom error types
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;
 }
}
class BusinessLogicError extends Error {
  constructor(message, code) {
    super(message);
   this.name = "BusinessLogicError";
   this.code = code;
  }
}
// Using custom errors
function validateUser(user) {
  if (!user.email) {
    throw new ValidationError("Email is required", "email");
  }
  if (!user.email.includes("@")) {
    throw new ValidationError("Invalid email format", "email");
  }
 if (user.age < 0 || user.age > 150) {
   throw new ValidationError("Age must be between 0 and 150", "age");
  }
}
async function fetchUserData(userId) {
  try {
    const response = await fetch(`/api/users/${userId}`);
    if (!response.ok) {
      throw new NetworkError(
        `Failed to fetch user: ${response.statusText}`,
        response.status
      );
```

```
return await response.json();
 } catch (error) {
    if (error instanceof NetworkError) {
      throw error; // Re-throw network errors
   throw new NetworkError("Network request failed", 0);
}
function processPayment(amount, balance) {
 if (amount <= ∅) {
   throw new BusinessLogicError(
      "Payment amount must be positive",
      "INVALID_AMOUNT"
   );
 }
 if (amount > balance) {
   throw new BusinessLogicError("Insufficient funds", "INSUFFICIENT_FUNDS");
  // Process payment...
}
```

# Try-Catch-Finally

### Basic Try-Catch

```
// Basic error handling
function basicTryCatch() {
 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);
 }
}
// Catching specific error types
function specificErrorHandling() {
 try {
   const user = { name: "Alice" };
   validateUser(user);
    processUser(user);
 } catch (error) {
    if (error instanceof ValidationError) {
      console.error(`Validation failed for ${error.field}: ${error.message}`);
```

```
// Show user-friendly validation message
} else if (error instanceof NetworkError) {
    console.error(`Network error (${error.statusCode}): ${error.message}`);
    // Show network error message, maybe retry
} else if (error instanceof BusinessLogicError) {
    console.error(`Business logic error (${error.code}): ${error.message}`);
    // Handle business logic errors
} else {
    console.error("Unexpected error:", error);
    // Log unexpected errors for debugging
}
}
```

### Finally Block

```
// Finally block always executes
function tryFinallyExample() {
 let resource = null;
 try {
   resource = acquireResource();
    processResource(resource);
   return "Success";
  } catch (error) {
   console.error("Error processing resource:", error);
    return "Error";
 } finally {
   // This always runs, regardless of success or error
   if (resource) {
      releaseResource(resource);
      console.log("Resource cleaned up");
   }
 }
}
// Practical example: File handling
async function processFile(filename) {
 let fileHandle = null;
 try {
   fileHandle = await openFile(filename);
    const data = await readFile(fileHandle);
    const processed = processData(data);
    await writeFile(fileHandle, processed);
   return processed;
  } catch (error) {
   console.error(`Error processing file ${filename}:`, error);
   throw error;
  } finally {
    // Always close the file, even if an error occurred
```

```
if (fileHandle) {
      await closeFile(fileHandle);
    }
 }
}
// Database transaction example
async function transferMoney(fromAccount, toAccount, amount) {
 const transaction = await db.beginTransaction();
 try {
    await db.debit(fromAccount, amount);
    await db.credit(toAccount, amount);
   await transaction.commit();
    return { success: true, transactionId: transaction.id };
  } catch (error) {
   await transaction.rollback();
    console.error("Transfer failed:", error);
   throw new BusinessLogicError("Transfer failed", "TRANSFER_ERROR");
 } finally {
   // Clean up transaction resources
   await transaction.close();
 }
}
```

### **Nested Try-Catch**

```
// Multiple levels of error handling
async function complexOperation() {
 try {
   // Outer try-catch for general errors
   const user = await fetchUser();
   try {
     // Inner try-catch for specific operation
     const preferences = await fetchUserPreferences(user.id);
      return { user, preferences };
   } catch (prefError) {
     // Handle preference errors gracefully
     console.warn("Could not load preferences:", prefError.message);
      return { user, preferences: getDefaultPreferences() };
 } catch (userError) {
   // Handle user fetch errors
   if (userError instanceof NetworkError && userError.statusCode === 404) {
     throw new BusinessLogicError("User not found", "USER_NOT_FOUND");
   throw userError; // Re-throw other errors
}
```

```
// Error recovery with fallbacks
async function robustDataFetch() {
 const sources = [
    () => fetchFromPrimaryAPI(),
    () => fetchFromSecondaryAPI(),
    () => fetchFromCache(),
    () => getDefaultData(),
 1;
 for (let i = 0; i < sources.length; i++) {
   try {
      const data = await sources[i]();
      if (i > 0) {
        console.warn(`Used fallback source ${i}`);
      }
      return data;
    } catch (error) {
      console.error(`Source ${i} failed:`, error.message);
      if (i === sources.length - 1) {
       throw new Error("All data sources failed");
      // Continue to next source
 }
}
```

# Q Debugging Techniques

#### Console Methods

```
// Different console methods for debugging
function debuggingWithConsole() {
  const user = { name: "Alice", age: 30, hobbies: ["reading", "swimming"] };

  // Basic logging
  console.log("User data:", user);

  // Different log levels
  console.info("Information message");
  console.warn("Warning message");
  console.error("Error message");

  // Formatted output
  console.log("User %s is %d years old", user.name, user.age);

  // Table format for arrays/objects
  console.table(user.hobbies);
  console.table([user]);

  // Grouping related logs
```

```
console.group("User Processing");
 console.log("Validating user...");
 console.log("User is valid");
 console.log("Processing complete");
 console.groupEnd();
 // Timing operations
 console.time("Operation");
 // ... some operation
 console.timeEnd("Operation");
 // Counting occurrences
 for (let i = 0; i < 5; i++) {
   console.count("Loop iteration");
 }
 // Stack trace
 console.trace("Execution path");
 // Conditional logging
 console.assert(user.age > 0, "Age should be positive");
 // Clear console
 // console.clear();
}
```

### **Debugger Statement**

```
// Using debugger statement
function debuggerExample(data) {
 console.log("Starting processing...");
 // Execution will pause here when dev tools are open
 debugger;
 const processed = data.map((item) => {
   // Another breakpoint
   debugger;
   return item * 2;
 });
 return processed;
}
// Conditional debugging
function conditionalDebugging(items) {
  items.forEach((item, index) => {
   // Only break on specific conditions
   if (item.error && process.env.NODE_ENV === "development") {
      debugger;
```

```
processItem(item);
});
}
```

### Error Boundaries (React-style pattern)

```
// Error boundary pattern for JavaScript
class ErrorBoundary {
 constructor(fallbackUI) {
   this.fallbackUI = fallbackUI;
   this.hasError = false;
   this.error = null;
 }
 wrap(fn) {
    return (...args) => {
     try {
       this.hasError = false;
       this.error = null;
       return fn(...args);
      } catch (error) {
        this.hasError = true;
        this.error = error;
        console.error("Error caught by boundary:", error);
        if (this.fallbackUI) {
         return this.fallbackUI(error);
       throw error;
      }
   };
 wrapAsync(fn) {
    return async (...args) => {
      try {
       this.hasError = false;
        this.error = null;
        return await fn(...args);
      } catch (error) {
        this.hasError = true;
        this.error = error;
        console.error("Async error caught by boundary:", error);
        if (this.fallbackUI) {
          return this.fallbackUI(error);
        throw error;
```

```
}
};
}
}

// Usage
const errorBoundary = new ErrorBoundary((error) => {
  return { error: true, message: "Something went wrong" };
});

const safeFunction = errorBoundary.wrap((data) => {
  // This function is now protected
  return riskyOperation(data);
});

const safeAsyncFunction = errorBoundary.wrapAsync(async (url) => {
  const response = await fetch(url);
  return response.json();
});
```

### Logging and Monitoring

```
// Advanced logging system
class Logger {
 constructor(level = "info") {
   this.level = level;
   this.levels = {
      error: 0,
      warn: 1,
      info: 2,
      debug: 3,
   };
 }
 log(level, message, data = {}) {
    if (this.levels[level] <= this.levels[this.level]) {</pre>
      const timestamp = new Date().toISOString();
      const logEntry = {
        timestamp,
        level: level.toUpperCase(),
       message,
        data,
        stack: new Error().stack,
      };
      // In production, send to logging service
      if (process.env.NODE_ENV === "production") {
       this.sendToLoggingService(logEntry);
      } else {
        console[level](logEntry);
```

```
}
 error(message, data) {
   this.log("error", message, data);
  }
 warn(message, data) {
   this.log("warn", message, data);
 }
 info(message, data) {
   this.log("info", message, data);
  }
 debug(message, data) {
   this.log("debug", message, data);
 }
 sendToLoggingService(logEntry) {
   // Send to external logging service
   // fetch('/api/logs', {
   // method: 'POST',
         headers: { 'Content-Type': 'application/json' },
       body: JSON.stringify(logEntry)
   // });
 }
}
const logger = new Logger("debug");
// Usage throughout application
function processUser(user) {
 logger.info("Processing user", { userId: user.id });
 try {
   validateUser(user);
    logger.debug("User validation passed", { userId: user.id });
    const result = performComplexOperation(user);
    logger.info("User processing completed", {
      userId: user.id,
      result: result.id,
    });
    return result;
  } catch (error) {
    logger.error("User processing failed", {
      userId: user.id,
      error: error.message,
      stack: error.stack,
    });
    throw error;
```

```
}
```

# Error Handling Patterns

### Result Pattern

```
// Result pattern for explicit error handling
class Result {
 constructor(success, value, error) {
   this.success = success;
   this.value = value;
   this.error = error;
 }
 static ok(value) {
   return new Result(true, value, null);
 static error(error) {
   return new Result(false, null, error);
 }
 is0k() {
   return this.success;
 }
 isError() {
   return !this.success;
  }
 map(fn) {
   if (this.isOk()) {
     try {
       return Result.ok(fn(this.value));
     } catch (error) {
        return Result.error(error);
      }
   return this;
 flatMap(fn) {
   if (this.is0k()) {
      try {
       return fn(this.value);
     } catch (error) {
        return Result.error(error);
    return this;
```

```
getOrElse(defaultValue) {
    return this.isOk() ? this.value : defaultValue;
}
// Using Result pattern
function safeParseJSON(jsonString) {
 try {
   const parsed = JSON.parse(jsonString);
   return Result.ok(parsed);
 } catch (error) {
   return Result.error(error);
 }
}
function processUserData(jsonString) {
  return safeParseJSON(jsonString)
    .map((data) => data.user)
    .map((user) => ({ ...user, processed: true }))
    .flatMap((user) => {
     if (!user.email) {
       return Result.error(new Error("Email is required"));
      }
      return Result.ok(user);
    });
}
// Usage
const result = processUserData(
  '{"user": {"name": "Alice", "email": "alice@example.com"}}'
);
if (result.is0k()) {
 console.log("Processed user:", result.value);
} else {
  console.error("Processing failed:", result.error.message);
}
```

### Circuit Breaker Pattern

```
// Circuit breaker for handling repeated failures
class CircuitBreaker {
  constructor(threshold = 5, timeout = 60000) {
    this.threshold = threshold;
    this.timeout = timeout;
    this.failureCount = 0;
    this.lastFailureTime = null;
    this.state = "CLOSED"; // CLOSED, OPEN, HALF_OPEN
}
```

```
async call(fn, ...args) {
    if (this.state === "OPEN") {
      if (Date.now() - this.lastFailureTime > this.timeout) {
        this.state = "HALF_OPEN";
      } else {
        throw new Error("Circuit breaker is OPEN");
      }
    }
    try {
      const result = await fn(...args);
     this.onSuccess();
     return result;
    } catch (error) {
      this.onFailure();
      throw error;
    }
  }
  onSuccess() {
   this.failureCount = 0;
    this.state = "CLOSED";
  }
 onFailure() {
    this.failureCount++;
    this.lastFailureTime = Date.now();
    if (this.failureCount >= this.threshold) {
      this.state = "OPEN";
    }
  }
  getState() {
    return this.state;
  }
}
// Usage
const apiCircuitBreaker = new CircuitBreaker(3, 30000); // 3 failures, 30s timeout
async function callAPI(endpoint) {
  return apiCircuitBreaker.call(async () => {
    const response = await fetch(endpoint);
    if (!response.ok) {
      throw new Error(`API call failed: ${response.status}`);
    return response.json();
 });
}
// The circuit breaker will prevent calls after 3 failures
// and allow them again after 30 seconds
```

### Retry Pattern

```
// Retry pattern with exponential backoff
class RetryHandler {
  constructor(maxRetries = 3, baseDelay = 1000, maxDelay = 10000) {
    this.maxRetries = maxRetries;
    this.baseDelay = baseDelay;
    this.maxDelay = maxDelay;
  }
  async retry(fn, ...args) {
    let lastError;
    for (let attempt = 1; attempt <= this.maxRetries; attempt++) {</pre>
      try {
        return await fn(...args);
      } catch (error) {
        lastError = error;
        if (attempt === this.maxRetries) {
          break;
        }
        // Don't retry on certain errors
        if (this.shouldNotRetry(error)) {
          break;
        const delay = this.calculateDelay(attempt);
        console.warn(
          `Attempt ${attempt} failed, retrying in ${delay}ms:`,
          error.message
        );
        await this.sleep(delay);
      }
    }
    throw lastError;
  }
  shouldNotRetry(error) {
    // Don't retry on client errors (4xx)
      error instanceof NetworkError &&
      error.statusCode >= 400 &&
      error.statusCode < 500
    ) {
      return true;
```

```
// Don't retry on validation errors
   if (error instanceof ValidationError) {
      return true;
   return false;
 calculateDelay(attempt) {
   // Exponential backoff with jitter
   const exponentialDelay = this.baseDelay * Math.pow(2, attempt - 1);
   const jitter = Math.random() * 0.1 * exponentialDelay;
   return Math.min(exponentialDelay + jitter, this.maxDelay);
 }
 sleep(ms) {
   return new Promise((resolve) => setTimeout(resolve, ms));
 }
}
// Usage
const retryHandler = new RetryHandler(3, 1000, 10000);
async function reliableAPICall(endpoint) {
 return retryHandler.retry(async () => {
    const response = await fetch(endpoint);
   if (!response.ok) {
     throw new NetworkError(
        `API call failed: ${response.statusText}`,
        response.status
      );
   return response.json();
 });
```

# Performance Debugging

### Memory Leak Detection

```
// Memory leak detection utilities
class MemoryMonitor {
  constructor() {
    this.snapshots = [];
    this.listeners = new Set();
}

takeSnapshot(label) {
```

```
if (performance.memory) {
    const snapshot = {
      label,
      timestamp: Date.now(),
      usedJSHeapSize: performance.memory.usedJSHeapSize,
     totalJSHeapSize: performance.memory.totalJSHeapSize,
      jsHeapSizeLimit: performance.memory.jsHeapSizeLimit,
   };
   this.snapshots.push(snapshot);
   return snapshot;
 }
 console.warn("performance.memory not available");
 return null;
}
compareSnapshots(label1, label2) {
 const snap1 = this.snapshots.find((s) => s.label === label1);
 const snap2 = this.snapshots.find((s) => s.label === label2);
 if (!snap1 || !snap2) {
   console.error("Snapshots not found");
   return null;
 }
 const diff = {
   timeDiff: snap2.timestamp - snap1.timestamp,
    memoryDiff: snap2.usedJSHeapSize - snap1.usedJSHeapSize,
   percentageIncrease:
      ((snap2.usedJSHeapSize - snap1.usedJSHeapSize) / snap1.usedJSHeapSize) *
      100,
 };
 console.log(`Memory change from ${label1} to ${label2}:`, diff);
 return diff;
}
startMonitoring(interval = 5000) {
 this.monitoringInterval = setInterval(() => {
    this.takeSnapshot(`auto-${Date.now()}`);
   // Alert if memory usage is growing rapidly
    if (this.snapshots.length >= 2) {
      const recent = this.snapshots.slice(-2);
      const growth = recent[1].usedJSHeapSize - recent[0].usedJSHeapSize;
      if (growth > 10 * 1024 * 1024) {
        // 10MB growth
        console.warn(
          "Rapid memory growth detected:",
          growth / 1024 / 1024,
          "MB"
        );
```

```
}, interval);
 stopMonitoring() {
    if (this.monitoringInterval) {
      clearInterval(this.monitoringInterval);
 }
}
// Usage
const memoryMonitor = new MemoryMonitor();
function potentiallyLeakyFunction() {
 memoryMonitor.takeSnapshot("before-operation");
 // Simulate memory-intensive operation
 const largeArray = new Array(1000000).fill("data");
 // Potential leak: not cleaning up event listeners
 const element = document.createElement("div");
 element.addEventListener("click", () => {
    console.log("Clicked");
 });
 memoryMonitor.takeSnapshot("after-operation");
 memoryMonitor.compareSnapshots("before-operation", "after-operation");
 // Clean up to prevent leaks
 element.removeEventListener("click", () => {});
```

### Performance Profiling

```
// Performance profiling utilities
class Profiler {
  constructor() {
    this.marks = new Map();
    this.measures = new Map();
}

mark(name) {
  const timestamp = performance.now();
  this.marks.set(name, timestamp);

  if (performance.mark) {
    performance.mark(name);
  }
}
```

```
measure(name, startMark, endMark) {
    const startTime = this.marks.get(startMark);
    const endTime = this.marks.get(endMark);
    if (startTime && endTime) {
      const duration = endTime - startTime;
      this.measures.set(name, duration);
      if (performance.measure) {
        performance.measure(name, startMark, endMark);
      }
     return duration;
    }
    console.error("Start or end mark not found");
    return null;
  }
  time(label, fn) {
    return async (...args) => {
      this.mark(`${label}-start`);
     try {
        const result = await fn(...args);
        this.mark(`${label}-end`);
        const duration = this.measure(label, `${label}-start`, `${label}-end`);
        console.log(`${label} took ${duration.toFixed(2)}ms`);
        return result;
      } catch (error) {
        this.mark(`${label}-end`);
        this.measure(label, `${label}-start`, `${label}-end`);
        throw error;
      }
   };
  getReport() {
    const report = {
      marks: Object.fromEntries(this.marks),
      measures: Object.fromEntries(this.measures),
    };
    console.table(report.measures);
    return report;
  }
}
// Usage
const profiler = new Profiler();
const timedFunction = profiler.time("data-processing", async (data) => {
```

```
// Simulate processing
 await new Promise((resolve) => setTimeout(resolve, 100));
 return data.map((item) => item * 2);
});
// Function will automatically be timed
timedFunction([1, 2, 3, 4, 5]).then((result) => {
 console.log("Result:", result);
 profiler.getReport();
});
```

### 

### 1. Swallowing Errors

```
// X Swallowing errors (hiding problems)
function badErrorHandling() {
 try {
   riskyOperation();
 } catch (error) {
   // Silent failure - very bad!
   // Error is lost, making debugging impossible
  }
}
// X Generic error handling
function genericErrorHandling() {
 try {
   riskyOperation();
  } catch (error) {
    console.log("Something went wrong"); // Not helpful!
 }
}
// Proper error handling
function goodErrorHandling() {
 try {
   riskyOperation();
 } catch (error) {
   // Log the actual error
    console.error("Risk operation failed:", error);
    // Provide context
    logger.error("Risk operation failed", {
      operation: "riskyOperation",
     timestamp: new Date().toISOString(),
      stack: error.stack,
    });
    // Re-throw if caller should handle it
    throw error;
```

```
} }
```

### 2. Not Handling Async Errors

```
// X Unhandled promise rejections
function unhandledPromiseRejection() {
  fetch("/api/data"); // No .catch() - unhandled rejection!
 Promise.resolve().then(() => {
   throw new Error("Oops");
  }); // No .catch() - unhandled rejection!
}
// ✓ Proper async error handling
function properAsyncErrorHandling() {
  fetch("/api/data")
    .then((response) => response.json())
    .then((data) => console.log(data))
    .catch((error) => console.error("Fetch failed:", error));
 Promise.resolve()
    .then(() => {
      throw new Error("Oops");
    .catch((error) => console.error("Promise failed:", error));
}
// With async/await
async function asyncErrorHandling() {
 try {
    const response = await fetch("/api/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Async operation failed:", error);
  }
}
// Global unhandled rejection handler
window.addEventListener("unhandledrejection", (event) => {
  console.error("Unhandled promise rejection:", event.reason);
  // Log to monitoring service
  logger.error("Unhandled promise rejection", {
    reason: event.reason,
   stack: event.reason?.stack,
  });
  // Prevent default browser behavior
```

```
event.preventDefault();
});
```

### 3. Improper Error Propagation

```
// X Converting errors to different types unnecessarily
function badErrorPropagation() {
 try {
   const result = validateUser(user);
   return result;
 } catch (validationError) {
   // Don't do this - loses original error information
   throw new Error("User validation failed");
 }
}
// ✓ Proper error propagation
function goodErrorPropagation() {
 try {
   const result = validateUser(user);
   return result;
 } catch (validationError) {
    // Add context but preserve original error
   validationError.context = { userId: user.id, timestamp: Date.now() };
   throw validationError;
 }
}
// // Wrapping with additional context
function errorWrapping() {
 try {
   const result = validateUser(user);
   return result;
  } catch (originalError) {
    const wrappedError = new Error(
      `User validation failed for user ${user.id}`
    wrappedError.originalError = originalError;
   wrappedError.userId = user.id;
   throw wrappedError;
 }
```

# Mini Practice Problems

### Problem 1: Error-Safe JSON Parser

```
// Create a safe JSON parser that handles various error cases
function safeJSONParse(jsonString, defaultValue = null) {
```

```
// Your implementation here
// Should handle:
// - Invalid JSON syntax
// - null/undefined input
// - Non-string input
// - Return defaultValue on any error
// - Log errors appropriately
}

// Test cases:
console.log(safeJSONParse('{"name": "Alice"}')); // { name: "Alice" }
console.log(safeJSONParse("invalid json")); // null
console.log(safeJSONParse(null)); // null
console.log(safeJSONParse(undefined, {})); // {}
console.log(safeJSONParse(123)); // null
```

### Problem 2: Async Operation with Timeout

```
// Create a function that adds timeout to any async operation
function withTimeout(asyncFn, timeoutMs) {
    // Your implementation here
    // Should return a function that:
    // - Calls the original async function
    // - Rejects with timeout error if operation takes too long
    // - Properly cleans up resources
}

// Test:
const slowOperation = () =>
    new Promise((resolve) => setTimeout(() => resolve("Done"), 5000));

const fastOperation = withTimeout(slowOperation, 2000);

fastOperation()
    .then((result) => console.log("Result:", result))
    .catch((error) => console.error("Error:", error.message)); // Should timeout
```

### Problem 3: Error Aggregator

```
// Create a utility that collects multiple errors and reports them together
class ErrorAggregator {
  constructor() {
    // Your implementation here
  }
  add(error, context = {}) {
    // Add an error with optional context
  }
}
```

```
hasErrors() {
   // Return true if any errors were added
  }
  getErrors() {
   // Return array of all errors with context
  throwIfAny() {
   // Throw aggregated error if any errors exist
  }
 clear() {
   // Clear all errors
 }
}
// Usage:
const errors = new ErrorAggregator();
// Collect multiple validation errors
if (!user.name) errors.add(new Error("Name required"), { field: "name" });
if (!user.email) errors.add(new Error("Email required"), { field: "email" });
if (user.age < 0) errors.add(new Error("Invalid age"), { field: "age" });</pre>
if (errors.hasErrors()) {
 console.log("Validation errors:", errors.getErrors());
 errors.throwIfAny(); // Throw combined error
}
```

### Problem 4: Retry with Circuit Breaker

```
// Combine retry logic with circuit breaker pattern
class ResilientCaller {
 constructor(maxRetries = 3, circuitThreshold = 5, circuitTimeout = 30000) {
   // Your implementation here
   // Should combine RetryHandler and CircuitBreaker functionality
  }
 async call(fn, ...args) {
   // Your implementation here
   // Should:
   // - Check circuit breaker state
   // - Retry on failures (with exponential backoff)
   // - Update circuit breaker on success/failure
   // - Respect circuit breaker open state
  }
  getStats() {
   // Return statistics about calls, failures, circuit state
```

```
// Test:
const resilientCaller = new ResilientCaller(3, 5, 10000);

const unreliableAPI = () => {
    if (Math.random() < 0.7) {
        throw new Error("API temporarily unavailable");
    }
    return "Success";
};

// Should retry failures and eventually open circuit
for (let i = 0; i < 10; i++) {
    resilientCaller
        .call(unreliableAPI)
        .then((result) => console.log(`Call ${i}:`, result))
        .catch((error) => console.error(`Call ${i}:`, error.message));
}
```

### Problem 5: Error Reporting Service

```
// Create an error reporting service that batches and sends errors
class ErrorReporter {
 constructor(options = {}) {
   this.endpoint = options.endpoint || "/api/errors";
   this.batchSize = options.batchSize | 10;
   this.flushInterval = options.flushInterval | 30000;
   // Your implementation here
  }
 report(error, context = {}) {
   // Add error to batch queue
   // Auto-flush if batch is full
  }
 flush() {
   // Send all queued errors to server
   // Return promise that resolves when sent
  startAutoFlush() {
   // Start automatic flushing on interval
  }
 stopAutoFlush() {
   // Stop automatic flushing
 }
}
// Usage:
```

```
const errorReporter = new ErrorReporter({
  endpoint: "/api/errors",
 batchSize: 5,
  flushInterval: 10000,
});
errorReporter.startAutoFlush();
// Report errors throughout the application
try {
  riskyOperation();
} catch (error) {
  errorReporter.report(error, {
    userId: currentUser.id,
    page: window.location.pathname,
    userAgent: navigator.userAgent,
 });
}
```

### Interview Notes

### **Common Questions:**

### Q: What are the different types of errors in JavaScript?

- Syntax errors: Invalid code structure, caught at parse time
- Reference errors: Using undefined variables or functions
- Type errors: Wrong type operations (calling non-function, accessing null properties)
- Range errors: Values outside valid range (array length, recursion depth)
- Custom errors: Application-specific error types

#### Q: What's the difference between throw and return in error handling?

- throw: Stops execution, propagates up the call stack until caught
- return: Normal function exit, doesn't indicate error to caller
- Use throw for exceptional conditions, return for normal flow

### Q: How do you handle errors in async/await vs Promises?

- Async/await: Use try-catch blocks
- Promises: Use .catch() method
- Both can be combined: async functions return promises

#### Q: What are unhandled promise rejections and how do you prevent them?

- Promises that reject without .catch() handler
- Can crash Node.js applications
- Prevent with global handlers and proper error handling

### Q: What's the purpose of the finally block?

Always executes regardless of try/catch outcome

- Used for cleanup (closing files, releasing resources)
- Executes even if return statement in try/catch

### **Asked at Companies:**

- Google: "Design an error handling system for a large-scale application"
- Facebook: "Implement a circuit breaker pattern from scratch"
- Amazon: "How would you debug a memory leak in a JavaScript application?"
- Microsoft: "Create a logging system that handles different error severities"
- Netflix: "Design error boundaries for a component-based architecture"

# **©** Key Takeaways

- 1. Handle errors explicitly don't let them fail silently
- 2. Use appropriate error types built-in and custom error classes
- 3. Provide meaningful error messages include context and actionable information
- 4. Log errors properly with sufficient detail for debugging
- 5. Handle async errors use try-catch with async/await or .catch() with promises
- 6. Implement error boundaries prevent single failures from crashing entire application
- 7. Use debugging tools effectively console methods, debugger, performance profiling
- 8. Plan for failure circuit breakers, retries, fallbacks

**Previous Chapter**: ← Asynchronous JavaScript

**Next Chapter**: ES6+ Features →

Practice: Try the error handling problems and experiment with different debugging techniques!

# Chapter 12: ES6+ Features

Master modern JavaScript syntax and features that make code more elegant and powerful.

### Plain English Explanation

ES6+ features are like upgrading from an old car to a modern one:

- **Arrow functions** = automatic transmission (simpler, more intuitive)
- **Template literals** = GPS navigation (easier string formatting)
- **Destructuring** = organized toolbox (extract what you need easily)
- Modules = standardized parts (import/export functionality)
- Classes = blueprint system (cleaner object-oriented programming)
- Promises = delivery tracking (better async handling)

These features make JavaScript more readable, maintainable, and powerful while reducing common programming errors.



**Basic Syntax** 

```
// Traditional function
function traditionalAdd(a, b) {
  return a + b;
}
// Arrow function - basic
const arrowAdd = (a, b) \Rightarrow \{
 return a + b;
};
// Arrow function - concise (implicit return)
const conciseAdd = (a, b) \Rightarrow a + b;
// Single parameter (parentheses optional)
const square = (x) => x * x;
const squareExplicit = (x) => x * x;
// No parameters
const greet = () => "Hello World!";
const getCurrentTime = () => new Date();
// Multiple statements
const processUser = (user) => {
  console.log(`Processing ${user.name}`);
  const processed = { ...user, processed: true };
  return processed;
};
// Returning object literals (wrap in parentheses)
const createUser = (name, age) => ({ name, age, id: Date.now() });
// Array methods with arrow functions
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((n) \Rightarrow n * 2);
const evens = numbers.filter((n) => n % 2 === 0);
const sum = numbers.reduce((acc, n) => acc + n, 0);
```

#### Lexical this Binding

```
// Traditional function - `this` depends on how it's called
function TraditionalTimer() {
  this.seconds = 0;

setInterval(function () {
   this.seconds++; // `this` refers to global object, not TraditionalTimer
   console.log(this.seconds); // NaN or undefined
  }, 1000);
}

// Solution with traditional function
```

```
function TraditionalTimerFixed() {
  this.seconds = 0;
  const self = this; // Capture reference
  setInterval(function () {
   self.seconds++; // Use captured reference
   console.log(self.seconds);
 }, 1000);
// Arrow function - `this` is lexically bound
function ArrowTimer() {
 this.seconds = 0;
 setInterval(() => {
    this.seconds++; // `this` refers to ArrowTimer instance
    console.log(this.seconds); // Works correctly!
 }, 1000);
// Practical example: Event handlers
class Button {
  constructor(element) {
    this.element = element;
   this.clickCount = 0;
    // Arrow function preserves `this`
    this.element.addEventListener("click", () => {
     this.clickCount++;
      console.log(`Clicked ${this.clickCount} times`);
    });
    // Traditional function would need .bind(this)
    // this.element.addEventListener('click', function() {
    // this.clickCount++; // `this` would be the element, not Button
    // }.bind(this));
  }
}
// React component example
class TodoList extends React.Component {
  constructor(props) {
   super(props);
   this.state = { todos: [] };
  }
  // Arrow function method - automatically bound
  addTodo = (text) => {
    this.setState({
      todos: [...this.state.todos, { id: Date.now(), text }],
   });
  };
  render() {
```

#### When NOT to Use Arrow Functions

```
// X Object methods (loses `this` context)
const person = {
 name: "Alice",
 greet: () => {
   console.log(`Hello, I'm ${this.name}`); // `this` is not person
 },
};
// ✓ Use regular function for object methods
const personFixed = {
 name: "Alice",
  greet() {
  console.log(`Hello, I'm ${this.name}`); // Works correctly
 },
};
// X Constructor functions
const Person = (name) => {
  this.name = name; // Error: arrow functions can't be constructors
};
// ✓ Use regular function or class
function PersonConstructor(name) {
 this.name = name;
}
class PersonClass {
 constructor(name) {
   this.name = name;
  }
}
// X When you need `arguments` object
const sumAll = () => {
  console.log(arguments); // ReferenceError: arguments is not defined
};
// ✓ Use rest parameters instead
const sumAllFixed = (...numbers) => {
 return numbers.reduce((sum, num) => sum + num, 0);
```

```
};
// X Dynamic context methods
const calculator = {
 value: ∅,
 add: (n) => (this.value += n), // `this` doesn't refer to calculator
 multiply: (n) => (this.value *= n),
};
// ✓ Use regular methods
const calculatorFixed = {
 value: ∅,
  add(n) {
   return (this.value += n);
 },
  multiply(n) {
  return (this.value *= n);
 },
};
```

# Template Literals

#### **Basic String Interpolation**

```
// Old way - string concatenation
const name = "Alice";
const age = 30;
const oldWay = "Hello, my name is " + name + " and I am " + age + " years old.";
// Template literals - much cleaner
const newWay = `Hello, my name is ${name} and I am ${age} years old.`;
// Expressions in template literals
const a = 5;
const b = 10;
const mathResult = `The sum of \{a\} and \{b\} is \{a + b\}.`;
// Function calls
const getCurrentTime = () => new Date().toLocaleTimeString();
const timeMessage = `Current time: ${getCurrentTime()}`;
// Object properties
const user = { name: "Bob", role: "admin" };
const userInfo = `User: ${user.name} (${user.role.toUpperCase()})`;
// Conditional expressions
const score = 85;
const result = `You ${
  score >= 60 ? "passed" : "failed"
} the test with ${score}%.`;
```

```
// Complex expressions
const items = ["apple", "banana", "orange"];
const itemList = `You have ${items.length} items: ${items.join(", ")}.`;
```

#### Multiline Strings

```
// Old way - concatenation or escaping
const oldMultiline = "This is line 1\n" + "This is line 2\n" + "This is line 3";
// Template literals - natural multiline
const newMultiline = `This is line 1
This is line 2
This is line 3';
// HTML templates
const htmlTemplate = `
    <div class="user-card">
        <h2>${user.name}</h2>
        Email: ${user.email}
        Role: ${user.role}
        <button onclick="editUser(${user.id})">
            Edit User
        </button>
    </div>
` ;
// SQL queries
const sqlQuery = `
    SELECT users.name, users.email, profiles.bio
    FROM users
    JOIN profiles ON users.id = profiles.user_id
    WHERE users.active = true
    AND users.created_at > '${startDate}'
    ORDER BY users.name
`;
// Configuration files
const configFile = `
    {
        "name": "${projectName}",
        "version": "${version}",
        "description": "${description}",
        "main": "${entryPoint}"
    }
```

### Tagged Template Literals

```
// Custom template tag functions
function highlight(strings, ...values) {
 return strings.reduce((result, string, i) => {
    const value = values[i] ? `<mark>${values[i]}</mark>` : "";
    return result + string + value;
 }, "");
const searchTerm = "JavaScript";
const text = highlight`Learn ${searchTerm} programming with ease!`;
// Result: "Learn <mark>JavaScript</mark> programming with ease!"
// SQL template tag (prevents injection)
function sql(strings, ...values) {
  const escapedValues = values.map((value) => {
    if (typeof value === "string") {
     return `'${value.replace(/'/g, "''")}'`; // Escape single quotes
    }
   return value;
 });
 return strings.reduce((result, string, i) => {
   const value = escapedValues[i] | "";
   return result + string + value;
  }, "");
const userId = 123;
const userName = "O'Connor";
const query = sql`SELECT * FROM users WHERE id = ${userId} AND name =
${userName}`;
// Result: "SELECT * FROM users WHERE id = 123 AND name = '0''Connor'"
// Internationalization template tag
function i18n(strings, ...values) {
  const key = strings.join("{}");
  const translation = translations[key] | key;
 return values.reduce((result, value, i) => {
   return result.replace("{}", value);
  }, translation);
const translations = {
 "Hello, {}! You have {} messages.": "Hola, {}! Tienes {} mensajes.",
};
const greeting = i18n`Hello, ${userName}! You have ${messageCount} messages.`;
// Styled components (React)
const Button = styled.button`
  background-color: ${(props) => (props.primary ? "blue" : "gray")};
  color: white;
```

```
padding: ${(props) => (props.size === "large" ? "12px 24px" : "8px 16px")};
border: none;
border-radius: 4px;
cursor: pointer;

&:hover {
    opacity: 0.8;
}
`;
```

# **@** Destructuring

#### Array Destructuring

```
// Basic array destructuring
const colors = ["red", "green", "blue"];
const [first, second, third] = colors;
console.log(first); // 'red'
console.log(second); // 'green'
console.log(third); // 'blue'
// Skipping elements
const [primary, , tertiary] = colors;
console.log(primary); // 'red'
console.log(tertiary); // 'blue'
// Default values
const [a, b, c, d = "yellow"] = colors;
console.log(d); // 'yellow' (default value)
// Rest operator
const numbers = [1, 2, 3, 4, 5];
const [head, ...tail] = numbers;
console.log(head); // 1
console.log(tail); // [2, 3, 4, 5]
// Swapping variables
let x = 1;
let y = 2;
[x, y] = [y, x];
console.log(x); // 2
console.log(y); // 1
// Function return values
function getCoordinates() {
 return [10, 20];
}
const [x, y] = getCoordinates();
// Nested arrays
```

```
const matrix = [
  [1, 2],
  [3, 4],
];
const [[a, b], [c, d]] = matrix;
console.log(a, b, c, d); // 1 2 3 4
// Practical examples
const csvLine = "John,Doe,30,Engineer";
const [firstName, lastName, age, profession] = csvLine.split(",");
// React hooks
const [count, setCount] = useState(∅);
const [loading, setLoading] = useState(false);
// Array methods
const entries = Object.entries({ name: "Alice", age: 30 });
entries.forEach(([key, value]) => {
 console.log(`${key}: ${value}`);
});
```

#### **Object Destructuring**

```
// Basic object destructuring
const person = {
 name: "Alice",
  age: 30,
 email: "alice@example.com",
 address: {
   street: "123 Main St",
   city: "New York",
   country: "USA",
 },
};
const { name, age, email } = person;
console.log(name); // 'Alice'
console.log(age); // 30
// Renaming variables
const { name: fullName, age: years } = person;
console.log(fullName); // 'Alice'
console.log(years); // 30
// Default values
const { name, age, phone = "Not provided" } = person;
console.log(phone); // 'Not provided'
// Nested destructuring
const {
  address: { street, city },
```

```
} = person;
console.log(street); // '123 Main St'
console.log(city); // 'New York'
// Renaming nested properties
const {
 address: { street: streetAddress, city: cityName },
} = person;
// Rest operator with objects
const { name, ...otherInfo } = person;
console.log(otherInfo); // { age: 30, email: '...', address: {...} }
// Function parameters
function greetUser({ name, age, email = "No email" }) {
  console.log(`Hello ${name}, you are ${age} years old.`);
 console.log(`Email: ${email}`);
}
greetUser(person);
// API response handling
function handleApiResponse({ data, status, message = "Success" }) {
  if (status === "success") {
   console.log(message);
   return data;
  } else {
    throw new Error(message);
  }
}
// React props destructuring
function UserCard({ user: { name, email, avatar }, onEdit, className = "" }) {
  return (
    <div className={`user-card ${className}`}>
      <img src={avatar} alt={name} />
      <h3>{name}</h3>
      {email}
      <button onClick={() => onEdit(name)}>Edit
    </div>
 );
}
// Configuration objects
function createServer({
 port = 3000,
 host = "localhost",
 ssl = false,
 middleware = [],
 routes = {},
}) {
 console.log(`Starting server on ${host}:${port}`);
  // Server setup logic...
```

```
createServer({
  port: 8080,
  ssl: true,
  middleware: [cors(), helmet()],
});
```

#### **Advanced Destructuring Patterns**

```
// Computed property names
const key = "dynamicKey";
const obj = { [key]: "value", other: "data" };
const { [key]: dynamicValue, other } = obj;
console.log(dynamicValue); // 'value'
// Destructuring in loops
const users = [
 { name: "Alice", age: 30 },
 { name: "Bob", age: 25 },
 { name: "Charlie", age: 35 },
];
// Array of objects
for (const { name, age } of users) {
 console.log(`${name} is ${age} years old`);
}
// Map entries
const userMap = new Map([
  ["alice", { name: "Alice", role: "admin" }],
  ["bob", { name: "Bob", role: "user" }],
]);
for (const [username, { name, role }] of userMap) {
console.log(`${username}: ${name} (${role})`);
}
// Mixed destructuring
const response = {
 data: {
    users: [
     { id: 1, name: "Alice" },
     { id: 2, name: "Bob" },
    ],
   meta: { total: 2, page: 1 },
 },
 status: "success",
};
const {
  data: {
```

```
users: [firstUser, ...otherUsers],
   meta: { total },
 },
 status,
} = response;
console.log(firstUser); // { id: 1, name: 'Alice' }
console.log(total); // 2
console.log(status); // 'success'
// Destructuring with validation
function processOrder({
 items = [],
 customer: { name, email } = {},
 shipping: { address, method = "standard" } = {},
 if (!name || !email) {
   throw new Error("Customer information is required");
 if (!address) {
   throw new Error("Shipping address is required");
 }
 console.log(`Processing order for ${name} (${email})`);
 console.log(`${items.length} items, shipping to ${address}`);
 console.log(`Shipping method: ${method}`);
}
```

### Spread and Rest Operators

#### Spread Operator (...)

```
// Array spreading
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Adding elements
const withExtra = [0, ...arr1, 3.5, ...arr2, 7]; // [0, 1, 2, 3, 3.5, 4, 5, 6, 7]

// Array copying (shallow)
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
copiedArray.push(4); // originalArray remains [1, 2, 3]

// Converting iterables to arrays
const nodeList = document.querySelectorAll(".item");
const elementsArray = [...nodeList];

const str = "hello";
```

```
const chars = [...str]; // ['h', 'e', 'l', 'l', 'o']
const set = new Set([1, 2, 3, 2, 1]);
const uniqueArray = [...set]; // [1, 2, 3]
// Function arguments
function sum(a, b, c) {
 return a + b + c;
}
const numbers = [1, 2, 3];
const result = sum(...numbers); // Same as sum(1, 2, 3)
// Math functions
const scores = [85, 92, 78, 96, 88];
const highest = Math.max(...scores);
const lowest = Math.min(...scores);
// Object spreading
const person = { name: "Alice", age: 30 };
const employee = { ...person, role: "developer", salary: 75000 };
// { name: 'Alice', age: 30, role: 'developer', salary: 75000 }
// Overriding properties
const updated = { ...person, age: 31 }; // age is updated to 31
// Merging objects
const defaults = { theme: "light", language: "en" };
const userPrefs = { theme: "dark" };
const settings = { ...defaults, ...userPrefs };
// { theme: 'dark', language: 'en' }
// Conditional spreading
const includeExtra = true;
const config = {
 base: "value",
  ...(includeExtra && { extra: "data" }),
};
// React props spreading
function Button({ children, ...props }) {
 return <button {...props}>{children}</button>;
}
// Usage: <Button className="primary" onClick={handleClick}>Click me</Button>
```

#### Rest Operator (...)

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

```
sum(1, 2, 3, 4, 5); // 15
sum(10, 20); // 30
sum(); // 0
// Mixed parameters
function greet(greeting, ...names) {
 return `${greeting} ${names.join(", ")}!`;
}
greet("Hello", "Alice", "Bob", "Charlie"); // "Hello Alice, Bob, Charlie!"
// Array destructuring with rest
const [first, second, ...rest] = [1, 2, 3, 4, 5];
console.log(first); // 1
console.log(second); // 2
console.log(rest); // [3, 4, 5]
// Object destructuring with rest
const user = {
 name: "Alice",
 age: 30,
 email: "alice@example.com",
 role: "admin",
};
const { name, age, ...otherProps } = user;
console.log(name); // 'Alice'
console.log(age); // 30
console.log(otherProps); // { email: 'alice@example.com', role: 'admin' }
// Function that accepts options
function createUser(name, email, ...options) {
  const [age, role = "user", department] = options;
 return {
    name,
    email,
    age,
    role,
    department,
  };
}
const newUser = createUser("Bob", "bob@example.com", 25, "admin", "IT");
// Flexible API functions
function apiCall(endpoint, method = "GET", ...middlewares) {
 console.log(`${method} ${endpoint}`);
 middlewares.forEach((middleware) => middleware());
}
apiCall(
  "/users",
```

```
"POST",
authMiddleware,
validationMiddleware,
loggingMiddleware
);

// Event handler with rest
function handleFormSubmit(event, ...validators) {
  event.preventDefault();

  const isValid = validators.every((validator) => validator());

  if (isValid) {
    console.log("Form is valid, submitting...");
  }
}
```

#### **Practical Combinations**

```
// Array manipulation utilities
function removeItem(array, index) {
 return [...array.slice(0, index), ...array.slice(index + 1)];
}
function insertItem(array, index, item) {
 return [...array.slice(∅, index), item, ...array.slice(index)];
}
function updateItem(array, index, newItem) {
  return array.map((item, i) => (i === index ? newItem : item));
}
// Object utilities
function omit(obj, ...keys) {
  const { [keys[0]]: omitted, ...rest } = obj;
 return keys.length > 1 ? omit(rest, ...keys.slice(1)) : rest;
}
function pick(obj, ...keys) {
 return keys.reduce((result, key) => {
    if (key in obj) {
      result[key] = obj[key];
    }
   return result;
  }, {});
// Usage
const user = {
  name: "Alice",
  age: 30,
```

```
email: "alice@example.com",
  password: "secret",
};
const publicUser = omit(user, "password"); // { name: 'Alice', age: 30, email:
'alice@example.com' }
const basicInfo = pick(user, "name", "age"); // { name: 'Alice', age: 30 }
// Redux-style state updates
function updateUserState(state, action) {
  switch (action.type) {
    case "UPDATE_USER":
      return {
        ...state,
        user: {
          ...state.user,
          ...action.payload,
        },
      };
    case "ADD_ITEM":
      return {
        ...state,
        items: [...state.items, action.payload],
      };
    case "REMOVE_ITEM":
      return {
        ...state,
        items: state.items.filter((item) => item.id !== action.payload.id),
      };
    default:
      return state;
  }
```

# **Enhanced Object Literals**

#### **Shorthand Properties and Methods**

```
// Old way
const name = "Alice";
const age = 30;
const email = "alice@example.com";

const oldUser = {
  name: name,
  age: age,
  email: email,
  greet: function () {
    return "Hello!";
```

```
},
};
// New way - shorthand properties
const newUser = {
 name, // Same as name: name
 age, // Same as age: age
 email, // Same as email: email
 greet() {
   // Same as greet: function()
  return "Hello!";
 },
};
// Method definitions
const calculator = {
  value: ∅,
  add(n) {
   this.value += n;
   return this;
 },
  multiply(n) {
   this.value *= n;
   return this;
  },
  get result() {
  return this.value;
  },
 set reset(val) {
   this.value = val | 0;
 },
};
// Usage: calculator.add(5).multiply(2).result; // 10
// Async methods
const apiService = {
  baseUrl: "https://api.example.com",
  async fetchUser(id) {
    const response = await fetch(`${this.baseUrl}/users/${id}`);
    return response.json();
  },
  async createUser(userData) {
    const response = await fetch(`${this.baseUrl}/users`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(userData),
    });
```

```
return response.json();
},
};
```

### **Computed Property Names**

```
// Dynamic property names
const propertyName = "dynamicKey";
const value = "dynamicValue";
const obj = {
  [propertyName]: value,
  [`${propertyName}_modified`]: value.toUpperCase(),
  [propertyName + "_count"]: 1,
};
// { dynamicKey: 'dynamicValue', dynamicKey_modified: 'DYNAMICVALUE',
dynamicKey_count: 1 }
// Function-based property names
function getPropertyName(prefix, suffix) {
 return `${prefix}_${suffix}`;
}
const config = {
  [getPropertyName("api", "url")]: "https://api.example.com",
  [getPropertyName("api", "key")]: "secret-key",
};
// { api_url: 'https://api.example.com', api_key: 'secret-key' }
// Symbol properties
const uniqueId = Symbol("id");
const metadata = Symbol("metadata");
const user = {
 name: "Alice",
  [uniqueId]: 12345,
  [metadata]: { created: new Date(), version: 1 },
};
// Computed method names
const actions = ["create", "read", "update", "delete"];
const api = {};
actions.forEach((action) => {
  api[`${action}User`] = function (data) {
    console.log(`${action.toUpperCase()} user:`, data);
 };
});
// api.createUser, api.readUser, api.updateUser, api.deleteUser
// React component with dynamic handlers
```

2025-07-22 javascript.md

```
class FormComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {};
    // Create handlers for each field
    props.fields.forEach((field) => {
      this[`handle${field.charAt(0).toUpperCase() + field.slice(1)}Change`] = (
        е
      ) => {
       this.setState({ [field]: e.target.value });
      };
    });
  }
}
// Practical example: Form validation
function createValidator(rules) {
  return {
    [Symbol.toStringTag]: "Validator",
    validate(data) {
      const errors = {};
      Object.keys(rules).forEach((field) => {
        const rule = rules[field];
        const value = data[field];
        if (rule.required && !value) {
          errors[field] = `${field} is required`;
        } else if (rule.minLength && value.length < rule.minLength) {</pre>
          errors[
            field
          ] = `${field} must be at least ${rule.minLength} characters`;
      });
      return {
        isValid: Object.keys(errors).length === 0,
        errors,
      };
    },
 };
}
const userValidator = createValidator({
  name: { required: true, minLength: 2 },
  email: { required: true },
  password: { required: true, minLength: 8 },
});
```

#### Named Exports and Imports

```
// math.js - Named exports
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;
export class Calculator {
 constructor() {
    this.value = 0;
  }
  add(n) {
   this.value += n;
   return this;
  }
}
// Alternative export syntax
const subtract = (a, b) \Rightarrow a - b;
const divide = (a, b) => a / b;
export { subtract, divide };
// Exporting with different names
const power = (base, exponent) => Math.pow(base, exponent);
export { power as pow };
// main.js - Named imports
import { PI, add, multiply, Calculator } from "./math.js";
import { subtract, divide, pow } from "./math.js";
// Using imported functions
console.log(add(5, 3)); // 8
console.log(multiply(4, 7)); // 28
console.log(PI); // 3.14159
const calc = new Calculator();
calc.add(10).add(5);
// Importing with different names
import { pow as power } from "./math.js";
console.log(power(2, 3)); // 8
// Importing all named exports
```

```
import * as MathUtils from "./math.js";
console.log(MathUtils.PI);
console.log(MathUtils.add(1, 2));

// Selective imports
import { add, multiply } from "./math.js";
```

#### **Default Exports and Imports**

```
// user.js - Default export
class User {
    constructor(name, email) {
       this.name = name;
       this.email = email;
    }
    greet() {
       return `Hello, I'm ${this.name}`;
    }
}
export default User;
// Alternative default export syntax
const createUser = (name, email) => new User(name, email);
export { createUser as default };
// Or inline default export
export default function validateEmail(email) {
    return email.includes('@');
}
// main.js - Default imports
import User from './user.js';
                                         // Import default export
import MyUser from './user.js';
                                         // Can use any name
import validateEmail from './validator.js'; // Import default function
const user = new User('Alice', 'alice@example.com');
console.log(user.greet());
// Mixed imports (default + named)
// utils.js
export default function log(message) {
   console.log(`[LOG]: ${message}`);
}
export const version = '1.0.0';
export const author = 'John Doe';
// main.js
import log, { version, author } from './utils.js';
```

```
log(`App version: ${version} by ${author}`);
```

#### Re-exports and Module Aggregation

```
// api/users.js
export async function getUsers() {
 const response = await fetch("/api/users");
 return response.json();
}
export async function createUser(userData) {
  const response = await fetch("/api/users", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(userData),
 });
 return response.json();
}
// api/posts.js
export async function getPosts() {
 const response = await fetch("/api/posts");
 return response.json();
}
export async function createPost(postData) {
  const response = await fetch("/api/posts", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(postData),
 });
  return response.json();
}
// api/index.js - Aggregating exports
export * from "./users.js"; // Re-export all named exports
export * from "./posts.js";
// Or selective re-exports
export { getUsers, createUser } from "./users.js";
export { getPosts, createPost } from "./posts.js";
// Re-export with renaming
export { getUsers as fetchUsers } from "./users.js";
// Re-export default as named
export { default as UserClass } from "./user.js";
// main.js - Clean imports
import { getUsers, createUser, getPosts, createPost } from "./api/index.js";
```

```
// Or namespace import
import * as API from "./api/index.js";
API.getUsers().then((users) => console.log(users));
```

#### **Dynamic Imports**

```
// Dynamic imports for code splitting
async function loadModule() {
 try {
    const module = await import("./heavy-module.js");
    module.doSomething();
 } catch (error) {
    console.error("Failed to load module:", error);
  }
}
// Conditional loading
if (condition) {
  import("./feature-module.js")
    .then((module) => {
     module.initializeFeature();
    })
    .catch((error) => {
      console.error("Feature not available:", error);
    });
}
// React lazy loading
const LazyComponent = React.lazy(() => import("./LazyComponent.js"));
function App() {
 return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
 );
}
// Dynamic import with destructuring
async function loadUtilities() {
  const { add, multiply, PI } = await import("./math.js");
 console.log(add(2, 3));
 console.log(PI);
}
// Module loading based on environment
const isDevelopment = process.env.NODE_ENV === "development";
const logger = isDevelopment
  ? await import("./dev-logger.js")
```

```
: await import("./prod-logger.js");
logger.log("Application started");
// Plugin system with dynamic imports
class PluginManager {
 constructor() {
    this.plugins = new Map();
  }
 async loadPlugin(name) {
   try {
      const plugin = await import(`./plugins/${name}.js`);
      this.plugins.set(name, plugin.default || plugin);
      console.log(`Plugin ${name} loaded successfully`);
    } catch (error) {
      console.error(`Failed to load plugin ${name}:`, error);
   }
  }
 getPlugin(name) {
   return this.plugins.get(name);
 }
}
const pluginManager = new PluginManager();
await pluginManager.loadPlugin("analytics");
await pluginManager.loadPlugin("authentication");
```

### **Classes**

### Basic Class Syntax

```
// ES6 Class
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    return `Hello, I'm ${this.name} and I'm ${this.age} years old.`;
  }
  haveBirthday() {
    this.age++;
    return `Happy birthday! I'm now ${this.age}.`;
  }
  // Static method
  static createAdult(name) {
```

```
return new Person(name, 18);
  }
 // Getter
  get description() {
  return `${this.name} (${this.age} years old)`;
 // Setter
 set fullName(name) {
  this.name = name;
 }
}
// Usage
const person = new Person("Alice", 30);
console.log(person.greet());
console.log(person.haveBirthday());
console.log(person.description);
person.fullName = "Alice Smith";
console.log(person.description);
const adult = Person.createAdult("Bob");
console.log(adult.age); // 18
```

#### Inheritance

```
// Base class
class Animal {
 constructor(name, species) {
   this.name = name;
   this.species = species;
 }
 makeSound() {
  return `${this.name} makes a sound`;
 move() {
   return `${this.name} moves`;
 }
 toString() {
   return `${this.name} the ${this.species}`;
 }
}
// Derived class
class Dog extends Animal {
 constructor(name, breed) {
```

```
super(name, "Dog"); // Call parent constructor
    this.breed = breed;
  }
  makeSound() {
   return `${this.name} barks: Woof!`;
  }
  fetch() {
    return `${this.name} fetches the ball`;
 // Override toString
 toString() {
    return `${super.toString()} (${this.breed})`;
}
class Cat extends Animal {
  constructor(name, isIndoor = true) {
   super(name, "Cat");
   this.isIndoor = isIndoor;
  }
 makeSound() {
   return `${this.name} meows: Meow!`;
  }
 climb() {
    return `${this.name} climbs the tree`;
}
// Usage
const dog = new Dog("Buddy", "Golden Retriever");
const cat = new Cat("Whiskers");
console.log(dog.makeSound()); // "Buddy barks: Woof!"
console.log(cat.makeSound()); // "Whiskers meows: Meow!"
console.log(dog.toString()); // "Buddy the Dog (Golden Retriever)"
// Polymorphism
const animals = [dog, cat];
animals.forEach((animal) => {
  console.log(animal.makeSound()); // Different behavior for each
});
// instanceof checks
console.log(dog instanceof Dog); // true
console.log(dog instanceof Animal); // true
console.log(cat instanceof Dog); // false
```

#### Private Fields and Methods (ES2022)

```
class BankAccount {
 // Private fields
 #balance = ∅;
 #accountNumber;
 #transactions = [];
 constructor(accountNumber, initialBalance = 0) {
   this.#accountNumber = accountNumber;
   this.#balance = initialBalance;
   this.#addTransaction("Initial deposit", initialBalance);
 // Private method
 #addTransaction(type, amount) {
   this.#transactions.push({
      type,
      amount,
     timestamp: new Date(),
     balance: this.#balance,
   });
  }
 // Public methods
 deposit(amount) {
   if (amount <= 0) {
     throw new Error("Deposit amount must be positive");
   this.#balance += amount;
   this.#addTransaction("Deposit", amount);
   return this. #balance;
  }
 withdraw(amount) {
   if (amount <= 0) {
      throw new Error("Withdrawal amount must be positive");
    }
   if (amount > this.#balance) {
     throw new Error("Insufficient funds");
    }
   this.#balance -= amount;
   this.#addTransaction("Withdrawal", -amount);
   return this. #balance;
 }
 // Getter for balance (read-only access)
  get balance() {
   return this. #balance;
```

```
get accountNumber() {
   return this.#accountNumber;
 getTransactionHistory() {
   // Return copy to prevent external modification
   return [...this.#transactions];
 }
}
// Usage
const account = new BankAccount("12345", 1000);
console.log(account.balance); // 1000
account.deposit(500);
console.log(account.balance); // 1500
account.withdraw(200);
console.log(account.balance); // 1300
// These would throw errors:
// console.log(account.#balance); // SyntaxError: Private field '#balance'
must be declared in an enclosing class
// account.#addTransaction('test', 100); // SyntaxError
console.log(account.getTransactionHistory());
```

#### Static Fields and Methods

```
class MathUtils {
 // Static fields
 static PI = 3.14159;
 static E = 2.71828;
 static #precision = 10; // Private static field
 // Static methods
 static add(a, b) {
   return a + b;
 }
 static multiply(a, b) {
  return a * b;
 }
 static round(number, decimals = 2) {
   return Math.round(number * Math.pow(10, decimals)) / Math.pow(10, decimals);
 }
 // Private static method
```

```
static #validateNumber(num) {
    if (typeof num !== "number" || isNaN(num)) {
     throw new Error("Invalid number");
   }
  }
 static circleArea(radius) {
   this.#validateNumber(radius);
    return this.round(this.PI * radius * radius);
  }
 // Static getter
 static get precision() {
   return this. #precision;
  }
  // Static setter
  static set precision(value) {
   if (value > 0 && value <= 20) {
     this.#precision = value;
    }
 }
}
// Usage
console.log(MathUtils.PI); // 3.14159
console.log(MathUtils.add(5, 3)); // 8
console.log(MathUtils.circleArea(5)); // 78.54
console.log(MathUtils.precision); // 10
MathUtils.precision = 5;
console.log(MathUtils.precision); // 5
// Counter example with static fields
class Counter {
 static #count = 0;
 static #instances = [];
 constructor(name) {
   this.name = name;
   this.id = ++Counter.#count;
    Counter.#instances.push(this);
  }
  static getCount() {
    return Counter.#count;
  }
  static getAllInstances() {
   return [...Counter.#instances];
  }
  static reset() {
    Counter.#count = 0;
```

```
Counter.#instances = [];
}

const counter1 = new Counter("First");
const counter2 = new Counter("Second");

console.log(Counter.getCount()); // 2
console.log(Counter.getAllInstances()); // [Counter, Counter]
```

#### Mixins and Composition

```
// Mixin functions
const Flyable = {
 fly() {
   return `${this.name} is flying`;
 },
 land() {
   return `${this.name} has landed`;
 },
};
const Swimmable = {
  swim() {
   return `${this.name} is swimming`;
 },
 dive() {
   return `${this.name} dives underwater`;
 },
};
const Walkable = {
 walk() {
   return `${this.name} is walking`;
 },
  return `${this.name} is running`;
 },
};
// Mixin helper function
function mixin(target, ...sources) {
  sources.forEach((source) => {
   Object.getOwnPropertyNames(source).forEach((name) => {
      if (name !== "constructor") {
        target.prototype[name] = source[name];
      }
    });
```

```
});
 return target;
}
// Base class
class Animal {
 constructor(name) {
   this.name = name;
 }
}
// Classes with mixins
class Bird extends Animal {
 constructor(name, wingspan) {
    super(name);
    this.wingspan = wingspan;
 }
}
class Duck extends Animal {
 constructor(name) {
   super(name);
 }
}
class Human extends Animal {
 constructor(name, age) {
   super(name);
   this.age = age;
 }
}
// Apply mixins
mixin(Bird, Flyable, Walkable);
mixin(Duck, Flyable, Swimmable, Walkable);
mixin(Human, Walkable, Swimmable);
// Usage
const eagle = new Bird("Eagle", 200);
const mallard = new Duck("Mallard");
const person = new Human("Alice", 30);
console.log(eagle.fly()); // "Eagle is flying"
console.log(eagle.walk()); // "Eagle is walking"
console.log(mallard.fly()); // "Mallard is flying"
console.log(mallard.swim()); // "Mallard is swimming"
console.log(mallard.walk()); // "Mallard is walking"
console.log(person.walk()); // "Alice is walking"
console.log(person.swim()); // "Alice is swimming"
// console.log(person.fly()); // Error: person.fly is not a function
// Factory function approach
```

```
function createFlyingAnimal(name, type) {
  const animal = {
    name,
    type,
    ...Flyable,
    ...Walkable,
  };

  return animal;
}

const bat = createFlyingAnimal("Bat", "Mammal");
  console.log(bat.fly()); // "Bat is flying"
```

# 

#### 1. Arrow Function Context Issues

```
// X Using arrow functions where you need dynamic `this`
const button = {
 text: "Click me",
 click: () => {
    console.log(this.text); // `this` is not the button object
 },
};
// ✓ Use regular function for object methods
const buttonFixed = {
 text: "Click me",
 click() {
    console.log(this.text); // Works correctly
 },
};
// X Arrow functions as constructors
const Person = (name) => {
 this.name = name; // Error: Arrow functions cannot be constructors
};
// ✓ Use regular function or class
function PersonConstructor(name) {
 this.name = name;
}
class PersonClass {
 constructor(name) {
   this.name = name;
 }
}
```

#### 2. Destructuring with Default Values

```
// X Incorrect default value syntax
function processUser({ name = "Anonymous", age = 0, email }) {
 // This works, but what if the entire object is undefined?
}
processUser(); // TypeError: Cannot destructure property 'name' of 'undefined'
// Provide default for the entire parameter
function processUserFixed({ name = "Anonymous", age = 0, email } = {}) {
  console.log(name, age, email);
}
processUserFixed(); // Works: "Anonymous 0 undefined"
// ★ Nested destructuring without defaults
function getAddress({
 user: {
   address: { street },
 },
}) {
 return street;
getAddress({ user: {} }); // TypeError: Cannot destructure property 'street' of
'undefined'
// ✓ Provide defaults at each level
function getAddressFixed({ user: { address: { street } = {} } = {} ) {
 return street;
}
getAddressFixed({ user: {} }); // undefined (no error)
```

#### 3. Module Import/Export Issues

```
// X Mixing default and named exports incorrectly
// utils.js
export default function log(message) {
   console.log(message);
}

export const version = "1.0.0";

// main.js - Wrong import
import { log, version } from "./utils.js"; // Error: log is default export

// Correct import
import log, { version } from "./utils.js";
```

```
// X Circular dependencies
// a.js
import { b } from "./b.js";
export const a = "A";

// b.js
import { a } from "./a.js"; // Circular dependency
export const b = "B";

// Avoid circular dependencies or use dynamic imports
// b.js
export const b = "B";

// Later, when needed:
async function useA() {
  const { a } = await import("./a.js");
  console.log(a);
}
```

#### 4. Class Inheritance Pitfalls

```
// X Forgetting to call super() in constructor
class Animal {
 constructor(name) {
   this.name = name;
}
class Dog extends Animal {
 constructor(name, breed) {
   // Missing super(name) call
   this.breed = breed; // ReferenceError: Must call super constructor
 }
}
// ✓ Always call super() first
class DogFixed extends Animal {
 constructor(name, breed) {
    super(name); // Call parent constructor first
    this.breed = breed;
 }
}
// X Incorrect method overriding
class Calculator {
 add(a, b) {
   return a + b;
 }
}
```

```
class ScientificCalculator extends Calculator {
  add(a, b, c = 0) {
   // Different signature
   return super.add(a, b) + c;
}
// This can cause confusion when used polymorphically
const calc = new ScientificCalculator();
console.log(calc.add(1, 2)); // Works, but third parameter is ignored
// ✓ Maintain consistent method signatures
class ScientificCalculatorFixed extends Calculator {
  add(a, b) {
   return super.add(a, b);
 addThree(a, b, c) {
   return this.add(a, b) + c;
  }
}
```

#### 5. Template Literal Gotchas

```
// X Unintended expression evaluation
const userInput = '${alert("XSS")}';
const message = `Hello ${userInput}`; // Safe - treated as string
// But be careful with eval-like functions
const dangerousTemplate = `Hello ${eval(userInput)}`; // Dangerous!
// ✓ Always sanitize user input
function sanitize(input) {
  return input.replace(/[<>"'&]/g, (char) => {
    const entities = {
      "<": "&lt;",
      ">": ">",
      '"': """,
      "'": "'",
      "&": "&",
   };
   return entities[char];
 });
}
const safeMessage = `Hello ${sanitize(userInput)}`;
// X Performance issues with large templates
function generateLargeHTML(items) {
  let html = "";
  items.forEach((item) => {
```

```
html += `<div>${item.name}</div>`; // String concatenation in loop
  });
 return html;
}
// ✓ Use array join for better performance
function generateLargeHTMLFixed(items) {
  return items.map((item) => `<div>${item.name}</div>`).join("");
}
```

### Mini Practice Problems

#### Problem 1: Advanced Destructuring

```
// Create a function that extracts and transforms data from a complex API response
function processApiResponse(response) {
 // Your implementation here
 // Extract: user name, email, first hobby, remaining hobbies, and address city
  // Handle cases where data might be missing
 // Return an object with: userName, userEmail, primaryHobby, otherHobbies, city
// Test data:
const apiResponse = {
  data: {
    user: {
      profile: {
        name: "Alice Johnson",
        contact: {
          email: "alice@example.com",
          phone: "123-456-7890",
       },
      },
      preferences: {
        hobbies: ["reading", "swimming", "photography"],
        settings: {
          theme: "dark",
        },
      },
      location: {
        address: {
          street: "123 Main St",
          city: "New York",
          country: "USA",
        },
      },
    },
  },
  meta: {
    timestamp: "2023-01-01T00:00:00Z",
  },
```

```
};

// Expected output:

// {

// userName: 'Alice Johnson',

// userEmail: 'alice@example.com',

// primaryHobby: 'reading',

// otherHobbies: ['swimming', 'photography'],

// city: 'New York'

// }
```

#### Problem 2: Class with Mixins

```
// Create a Vehicle class system with mixins for different capabilities
// Base Vehicle class
class Vehicle {
 constructor(make, model, year) {
   // Your implementation
 }
}
// Mixins
const Drivable = {
 // Add methods: start(), stop(), accelerate(speed), brake()
};
const Flyable = {
// Add methods: takeOff(), land(), setAltitude(altitude)
};
const Floatable = {
// Add methods: launch(), dock(), setDepth(depth)
};
// Create specific vehicle types:
// Car (Drivable)
// Airplane (Drivable, Flyable)
// Boat (Drivable, Floatable)
// AmphibiousVehicle (Drivable, Flyable, Floatable)
// Test your implementation:
const car = new Car("Toyota", "Camry", 2023);
const plane = new Airplane("Boeing", "737", 2022);
const boat = new Boat("Yamaha", "WaveRunner", 2023);
const amphibious = new AmphibiousVehicle("DUKW", "Amphibian", 1944);
// All should work:
car.start();
car.accelerate(60);
```

```
plane.start();
plane.takeOff();
plane.setAltitude(30000);

boat.launch();
boat.accelerate(25);

amphibious.start();
amphibious.accelerate(30);
amphibious.takeOff();
amphibious.launch();
```

# Problem 3: Module System

```
// Create a plugin system using ES6 modules
// plugin-manager.js
// Create a PluginManager class that can:
// - Register plugins dynamically
// - Load plugins from files
// - Execute plugin hooks
// - Handle plugin dependencies
class PluginManager {
    constructor() {
        // Your implementation
    async loadPlugin(pluginPath) {
        // Load plugin from file path
    }
    registerPlugin(name, plugin) {
        // Register plugin manually
    }
    executeHook(hookName, ...args) {
        // Execute all plugins that have this hook
    }
    getPlugin(name) {
        // Get specific plugin
    }
}
// Example plugins:
// plugins/logger.js
export default {
    name: 'logger',
    hooks: {
        beforeRequest: (url) => console.log(`Making request to: ${url}`),
```

```
afterRequest: (response) => console.log(`Response status:
${response.status}`)
   }
};
// plugins/cache.js
export default {
    name: 'cache',
    dependencies: ['logger'],
    hooks: {
        beforeRequest: (url) => {
            // Check cache
        },
        afterRequest: (response) => {
            // Store in cache
    }
};
// Usage:
const pm = new PluginManager();
await pm.loadPlugin('./plugins/logger.js');
await pm.loadPlugin('./plugins/cache.js');
pm.executeHook('beforeRequest', 'https://api.example.com/users');
```

#### Problem 4: Advanced Template Literals

```
// Create a template engine using tagged template literals
// Your implementation should support:
// - Variable interpolation
// - Conditional rendering
// - Loop rendering
// - Nested templates
// - Escaping for security
function template(strings, ...values) {
    // Your implementation here
    // Should handle special syntax like:
   // ${if condition}...${endif}
    // ${for item in items}...${endfor}
   // ${include 'template-name'}
}
// Register sub-templates
template.register = function(name, templateFn) {
   // Register named templates
};
// Example usage:
```

```
const users = [
    { name: 'Alice', active: true, role: 'admin' },
    { name: 'Bob', active: false, role: 'user' },
    { name: 'Charlie', active: true, role: 'user' }
];
const userListTemplate = template`
    <div class="user-list">
        <h2>Users (${users.length})</h2>
        ${for user in users}
            <div class="user ${user.active ? 'active' : 'inactive'}">
                <h3>${user.name}</h3>
                Role: ${user.role}
                ${if user.role === 'admin'}
                    <span class="badge">Administrator</span>
                ${endif}
            </div>
        ${endfor}
    </div>
console.log(userListTemplate);
```

## Problem 5: Async Class with Error Handling

```
// Create a DataService class that handles async operations with proper error
handling
class DataService {
 constructor(baseUrl, options = {}) {
   // Your implementation
   // Should support: timeout, retries, caching, rate limiting
 }
 async get(endpoint, options = {}) {
   // GET request with error handling
  }
 async post(endpoint, data, options = {}) {
   // POST request with error handling
  }
 async put(endpoint, data, options = {}) {
   // PUT request with error handling
  }
 async delete(endpoint, options = {}) {
   // DELETE request with error handling
  }
  // Private methods for:
```

```
// - Retry logic
 // - Rate limiting
 // - Caching
 // - Request/response transformation
// Usage should support:
const api = new DataService("https://api.example.com", {
 timeout: 5000,
 retries: 3,
 cache: true,
 rateLimit: { requests: 100, per: "minute" },
});
// All methods should return promises and handle errors gracefully
 const users = await api.get("/users");
 const newUser = await api.post("/users", {
   name: "Alice",
   email: "alice@example.com",
 });
 const updatedUser = await api.put(`/users/${newUser.id}`, {
   name: "Alice Smith",
 });
 await api.delete(`/users/${newUser.id}`);
} catch (error) {
  console.error("API operation failed:", error);
}
```

# Interview Notes

#### **Common Questions:**

# Q: What's the difference between let, const, and var?

- var: Function-scoped, hoisted, can be redeclared
- let: Block-scoped, hoisted but in temporal dead zone, cannot be redeclared
- const: Block-scoped, hoisted but in temporal dead zone, cannot be reassigned or redeclared

### Q: When should you use arrow functions vs regular functions?

- Arrow functions: Callbacks, array methods, when you want lexical this
- Regular functions: Object methods, constructors, when you need dynamic this

#### Q: What are the benefits of destructuring?

- Cleaner code, easier data extraction, default values, swapping variables
- Reduces repetitive property access, makes function parameters more readable

#### Q: How do ES6 modules differ from CommonJS?

• ES6: Static analysis, tree shaking, top-level await, import/export syntax

• CommonJS: Dynamic loading, require/module.exports, synchronous

#### Q: What are the advantages of ES6 classes over function constructors?

- Cleaner syntax, built-in inheritance with extends, static methods, private fields
- Better error messages, no accidental function calls without new

# Asked at Companies:

- Google: "Implement a module bundler that handles ES6 imports"
- Facebook: "Create a React-like component system using ES6 classes"
- Amazon: "Design a plugin architecture using ES6 modules and dynamic imports"
- Microsoft: "Implement a template engine using tagged template literals"
- Netflix: "Create a data fetching library with classes and async/await"

# **©** Key Takeaways

- 1. Arrow functions Great for callbacks and when you need lexical this
- 2. **Template literals** Much cleaner than string concatenation
- 3. **Destructuring** Powerful way to extract data from arrays and objects
- 4. Spread/Rest Flexible operators for arrays, objects, and function parameters
- 5. **Modules** Proper way to organize and share code
- 6. Classes Clean syntax for object-oriented programming
- 7. Enhanced object literals Shorthand properties and computed names
- 8. **Default parameters** Cleaner function definitions

**Previous Chapter**: ← Error Handling & Debugging

**Next Chapter**: DOM Manipulation →

Practice: Try the ES6+ problems and experiment with modern JavaScript features!

# Chapter 12: ES6+ Features

Master modern JavaScript syntax and features that make code more elegant and powerful.

# Plain English Explanation

ES6+ features are like upgrading from an old car to a modern one:

- **Arrow functions** = automatic transmission (simpler, more intuitive)
- **Template literals** = GPS navigation (easier string formatting)
- **Destructuring** = organized toolbox (extract what you need easily)
- **Modules** = standardized parts (import/export functionality)
- Classes = blueprint system (cleaner object-oriented programming)
- **Promises** = delivery tracking (better async handling)

These features make JavaScript more readable, maintainable, and powerful while reducing common programming errors.



### **Basic Syntax**

```
// Traditional function
function traditionalAdd(a, b) {
  return a + b;
}
// Arrow function - basic
const arrowAdd = (a, b) \Rightarrow \{
 return a + b;
};
// Arrow function - concise (implicit return)
const conciseAdd = (a, b) \Rightarrow a + b;
// Single parameter (parentheses optional)
const square = (x) \Rightarrow x * x;
const squareExplicit = (x) \Rightarrow x * x;
// No parameters
const greet = () => "Hello World!";
const getCurrentTime = () => new Date();
// Multiple statements
const processUser = (user) => {
  console.log(`Processing ${user.name}`);
  const processed = { ...user, processed: true };
  return processed;
};
// Returning object literals (wrap in parentheses)
const createUser = (name, age) => ({ name, age, id: Date.now() });
// Array methods with arrow functions
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((n) \Rightarrow n * 2);
const evens = numbers.filter((n) => n % 2 === 0);
const sum = numbers.reduce((acc, n) => acc + n, 0);
```

### Lexical this Binding

```
// Traditional function - `this` depends on how it's called
function TraditionalTimer() {
  this.seconds = 0;

setInterval(function () {
   this.seconds++; // `this` refers to global object, not TraditionalTimer
   console.log(this.seconds); // NaN or undefined
```

```
}, 1000);
// Solution with traditional function
function TraditionalTimerFixed() {
 this.seconds = 0;
 const self = this; // Capture reference
  setInterval(function () {
    self.seconds++; // Use captured reference
   console.log(self.seconds);
  }, 1000);
}
// Arrow function - `this` is lexically bound
function ArrowTimer() {
  this.seconds = 0;
  setInterval(() => {
    this.seconds++; // `this` refers to ArrowTimer instance
    console.log(this.seconds); // Works correctly!
 }, 1000);
}
// Practical example: Event handlers
class Button {
  constructor(element) {
    this.element = element;
    this.clickCount = 0;
    // Arrow function preserves `this`
    this.element.addEventListener("click", () => {
     this.clickCount++;
      console.log(`Clicked ${this.clickCount} times`);
    });
    // Traditional function would need .bind(this)
    // this.element.addEventListener('click', function() {
         this.clickCount++; // `this` would be the element, not Button
    // }.bind(this));
  }
}
// React component example
class TodoList extends React.Component {
  constructor(props) {
   super(props);
   this.state = { todos: [] };
  }
  // Arrow function method - automatically bound
  addTodo = (text) => {
    this.setState({
      todos: [...this.state.todos, { id: Date.now(), text }],
```

#### When NOT to Use Arrow Functions

```
// X Object methods (loses `this` context)
const person = {
 name: "Alice",
 greet: () => {
  console.log(`Hello, I'm ${this.name}`); // `this` is not person
 },
};
// ✓ Use regular function for object methods
const personFixed = {
 name: "Alice",
  greet() {
   console.log(`Hello, I'm ${this.name}`); // Works correctly
 },
};
// X Constructor functions
const Person = (name) => {
 this.name = name; // Error: arrow functions can't be constructors
};
// ✓ Use regular function or class
function PersonConstructor(name) {
 this.name = name;
class PersonClass {
 constructor(name) {
   this.name = name;
 }
}
// X When you need `arguments` object
const sumAll = () \Rightarrow \{
  console.log(arguments); // ReferenceError: arguments is not defined
};
```

```
// // Use rest parameters instead
const sumAllFixed = (...numbers) => {
 return numbers.reduce((sum, num) => sum + num, 0);
};
// X Dynamic context methods
const calculator = {
 value: ∅,
 add: (n) => (this.value += n), // `this` doesn't refer to calculator
 multiply: (n) => (this.value *= n),
};
// ✓ Use regular methods
const calculatorFixed = {
 value: ∅,
 add(n) {
   return (this.value += n);
 },
  multiply(n) {
   return (this.value *= n);
 },
};
```

# Template Literals

## **Basic String Interpolation**

```
// Old way - string concatenation
const name = "Alice";
const age = 30;
const oldWay = "Hello, my name is " + name + " and I am " + age + " years old.";
// Template literals - much cleaner
const newWay = `Hello, my name is ${name} and I am ${age} years old.`;
// Expressions in template literals
const a = 5;
const b = 10;
const mathResult = `The sum of \{a\} and \{b\} is \{a + b\}.`;
// Function calls
const getCurrentTime = () => new Date().toLocaleTimeString();
const timeMessage = `Current time: ${getCurrentTime()}`;
// Object properties
const user = { name: "Bob", role: "admin" };
const userInfo = `User: ${user.name} (${user.role.toUpperCase()})`;
// Conditional expressions
const score = 85;
```

```
const result = `You ${
   score >= 60 ? "passed" : "failed"
} the test with ${score}%.`;

// Complex expressions
const items = ["apple", "banana", "orange"];
const itemList = `You have ${items.length} items: ${items.join(", ")}.`;
```

### Multiline Strings

```
// Old way - concatenation or escaping
const oldMultiline = "This is line 1\n" + "This is line 2\n" + "This is line 3";
// Template literals - natural multiline
const newMultiline = `This is line 1
This is line 2
This is line 3';
// HTML templates
const htmlTemplate = `
    <div class="user-card">
        <h2>${user.name}</h2>
        Email: ${user.email}
        Role: ${user.role}
        <button onclick="editUser(${user.id})">
            Edit User
        </button>
    </div>
`;
// SQL queries
const sqlQuery = `
    SELECT users.name, users.email, profiles.bio
    FROM users
    JOIN profiles ON users.id = profiles.user_id
    WHERE users.active = true
    AND users.created at > '${startDate}'
    ORDER BY users.name
`;
// Configuration files
const configFile = `
    {
        "name": "${projectName}",
        "version": "${version}",
        "description": "${description}",
        "main": "${entryPoint}"
```

#### **Tagged Template Literals**

```
// Custom template tag functions
function highlight(strings, ...values) {
 return strings.reduce((result, string, i) => {
   const value = values[i] ? `<mark>${values[i]}</mark>` : "";
   return result + string + value;
 }, "");
}
const searchTerm = "JavaScript";
const text = highlight`Learn ${searchTerm} programming with ease!`;
// Result: "Learn <mark>JavaScript</mark> programming with ease!"
// SQL template tag (prevents injection)
function sql(strings, ...values) {
  const escapedValues = values.map((value) => {
    if (typeof value === "string") {
     return `'${value.replace(/'/g, "''")}'`; // Escape single quotes
    }
   return value;
  });
  return strings.reduce((result, string, i) => {
    const value = escapedValues[i] || "";
   return result + string + value;
 }, "");
}
const userId = 123;
const userName = "O'Connor";
const query = sql`SELECT * FROM users WHERE id = ${userId} AND name =
${userName}`;
// Result: "SELECT * FROM users WHERE id = 123 AND name = '0''Connor'"
// Internationalization template tag
function i18n(strings, ...values) {
 const key = strings.join("{}");
  const translation = translations[key] | key;
 return values.reduce((result, value, i) => {
   return result.replace("{}", value);
  }, translation);
}
const translations = {
  "Hello, {}! You have {} messages.": "Hola, {}! Tienes {} mensajes.",
};
const greeting = i18n`Hello, ${userName}! You have ${messageCount} messages.`;
// Styled components (React)
```

```
const Button = styled.button`
  background-color: ${(props) => (props.primary ? "blue" : "gray")};
  color: white;
  padding: ${(props) => (props.size === "large" ? "12px 24px" : "8px 16px")};
  border: none;
  border-radius: 4px;
  cursor: pointer;

  &:hover {
    opacity: 0.8;
  }
  `;
```

# **@** Destructuring

#### Array Destructuring

```
// Basic array destructuring
const colors = ["red", "green", "blue"];
const [first, second, third] = colors;
console.log(first); // 'red'
console.log(second); // 'green'
console.log(third); // 'blue'
// Skipping elements
const [primary, , tertiary] = colors;
console.log(primary); // 'red'
console.log(tertiary); // 'blue'
// Default values
const [a, b, c, d = "yellow"] = colors;
console.log(d); // 'yellow' (default value)
// Rest operator
const numbers = [1, 2, 3, 4, 5];
const [head, ...tail] = numbers;
console.log(head); // 1
console.log(tail); // [2, 3, 4, 5]
// Swapping variables
let x = 1;
let y = 2;
[x, y] = [y, x];
console.log(x); // 2
console.log(y); // 1
// Function return values
function getCoordinates() {
 return [10, 20];
}
```

```
const [x, y] = getCoordinates();
// Nested arrays
const matrix = [
  [1, 2],
 [3, 4],
];
const [[a, b], [c, d]] = matrix;
console.log(a, b, c, d); // 1 2 3 4
// Practical examples
const csvLine = "John,Doe,30,Engineer";
const [firstName, lastName, age, profession] = csvLine.split(",");
// React hooks
const [count, setCount] = useState(∅);
const [loading, setLoading] = useState(false);
// Array methods
const entries = Object.entries({ name: "Alice", age: 30 });
entries.forEach(([key, value]) => {
 console.log(`${key}: ${value}`);
});
```

### **Object Destructuring**

```
// Basic object destructuring
const person = {
 name: "Alice",
 age: 30,
 email: "alice@example.com",
 address: {
   street: "123 Main St",
   city: "New York",
   country: "USA",
 },
};
const { name, age, email } = person;
console.log(name); // 'Alice'
console.log(age); // 30
// Renaming variables
const { name: fullName, age: years } = person;
console.log(fullName); // 'Alice'
console.log(years); // 30
// Default values
const { name, age, phone = "Not provided" } = person;
console.log(phone); // 'Not provided'
```

```
// Nested destructuring
const {
  address: { street, city },
} = person;
console.log(street); // '123 Main St'
console.log(city); // 'New York'
// Renaming nested properties
const {
 address: { street: streetAddress, city: cityName },
} = person;
// Rest operator with objects
const { name, ...otherInfo } = person;
console.log(otherInfo); // { age: 30, email: '...', address: {...} }
// Function parameters
function greetUser({ name, age, email = "No email" }) {
 console.log(`Hello ${name}, you are ${age} years old.`);
 console.log(`Email: ${email}`);
}
greetUser(person);
// API response handling
function handleApiResponse({ data, status, message = "Success" }) {
  if (status === "success") {
   console.log(message);
   return data;
  } else {
    throw new Error(message);
  }
}
// React props destructuring
function UserCard({ user: { name, email, avatar }, onEdit, className = "" }) {
  return (
    <div className={`user-card ${className}`}>
      <img src={avatar} alt={name} />
      <h3>{name}</h3>
      {email}
      <button onClick={() => onEdit(name)}>Edit
    </div>
  );
}
// Configuration objects
function createServer({
 port = 3000,
 host = "localhost",
 ssl = false,
 middleware = [],
  routes = {},
}) {
```

```
console.log(`Starting server on ${host}:${port}`);
  // Server setup logic...
}

createServer({
  port: 8080,
   ssl: true,
  middleware: [cors(), helmet()],
});
```

## Advanced Destructuring Patterns

```
// Computed property names
const key = "dynamicKey";
const obj = { [key]: "value", other: "data" };
const { [key]: dynamicValue, other } = obj;
console.log(dynamicValue); // 'value'
// Destructuring in loops
const users = [
 { name: "Alice", age: 30 },
 { name: "Bob", age: 25 },
 { name: "Charlie", age: 35 },
];
// Array of objects
for (const { name, age } of users) {
 console.log(`${name} is ${age} years old`);
}
// Map entries
const userMap = new Map([
  ["alice", { name: "Alice", role: "admin" }],
  ["bob", { name: "Bob", role: "user" }],
]);
for (const [username, { name, role }] of userMap) {
 console.log(`${username}: ${name} (${role})`);
}
// Mixed destructuring
const response = {
  data: {
    users: [
     { id: 1, name: "Alice" },
     { id: 2, name: "Bob" },
    1,
   meta: { total: 2, page: 1 },
  status: "success",
};
```

```
const {
 data: {
    users: [firstUser, ...otherUsers],
   meta: { total },
 },
  status,
} = response;
console.log(firstUser); // { id: 1, name: 'Alice' }
console.log(total); // 2
console.log(status); // 'success'
// Destructuring with validation
function processOrder({
  items = [],
 customer: { name, email } = {},
  shipping: { address, method = "standard" } = {},
  if (!name || !email) {
   throw new Error("Customer information is required");
  if (!address) {
   throw new Error("Shipping address is required");
  }
 console.log(`Processing order for ${name} (${email})`);
 console.log(`${items.length} items, shipping to ${address}`);
  console.log(`Shipping method: ${method}`);
}
```

# Spread and Rest Operators

# Spread Operator (...)

```
// Array spreading
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Adding elements
const withExtra = [0, ...arr1, 3.5, ...arr2, 7]; // [0, 1, 2, 3, 3.5, 4, 5, 6, 7]

// Array copying (shallow)
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
copiedArray.push(4); // originalArray remains [1, 2, 3]

// Converting iterables to arrays
const nodeList = document.querySelectorAll(".item");
```

```
const elementsArray = [...nodeList];
const str = "hello";
const chars = [...str]; // ['h', 'e', 'l', 'l', 'o']
const set = new Set([1, 2, 3, 2, 1]);
const uniqueArray = [...set]; // [1, 2, 3]
// Function arguments
function sum(a, b, c) {
 return a + b + c;
}
const numbers = [1, 2, 3];
const result = sum(...numbers); // Same as sum(1, 2, 3)
// Math functions
const scores = [85, 92, 78, 96, 88];
const highest = Math.max(...scores);
const lowest = Math.min(...scores);
// Object spreading
const person = { name: "Alice", age: 30 };
const employee = { ...person, role: "developer", salary: 75000 };
// { name: 'Alice', age: 30, role: 'developer', salary: 75000 }
// Overriding properties
const updated = { ...person, age: 31 }; // age is updated to 31
// Merging objects
const defaults = { theme: "light", language: "en" };
const userPrefs = { theme: "dark" };
const settings = { ...defaults, ...userPrefs };
// { theme: 'dark', language: 'en' }
// Conditional spreading
const includeExtra = true;
const config = {
 base: "value",
  ...(includeExtra && { extra: "data" }),
};
// React props spreading
function Button({ children, ...props }) {
 return <button {...props}>{children}</button>;
}
// Usage: <Button className="primary" onClick={handleClick}>Click me</Button>
```

#### Rest Operator (...)

```
// Function parameters
function sum(...numbers) {
 return numbers.reduce((total, num) => total + num, 0);
}
sum(1, 2, 3, 4, 5); // 15
sum(10, 20); // 30
sum(); // 0
// Mixed parameters
function greet(greeting, ...names) {
 return `${greeting} ${names.join(", ")}!`;
}
greet("Hello", "Alice", "Bob", "Charlie"); // "Hello Alice, Bob, Charlie!"
// Array destructuring with rest
const [first, second, ...rest] = [1, 2, 3, 4, 5];
console.log(first); // 1
console.log(second); // 2
console.log(rest); // [3, 4, 5]
// Object destructuring with rest
const user = {
 name: "Alice",
 age: 30,
  email: "alice@example.com",
 role: "admin",
};
const { name, age, ...otherProps } = user;
console.log(name); // 'Alice'
console.log(age); // 30
console.log(otherProps); // { email: 'alice@example.com', role: 'admin' }
// Function that accepts options
function createUser(name, email, ...options) {
  const [age, role = "user", department] = options;
 return {
    name,
    email,
    age,
    role,
    department,
 };
}
const newUser = createUser("Bob", "bob@example.com", 25, "admin", "IT");
// Flexible API functions
function apiCall(endpoint, method = "GET", ...middlewares) {
  console.log(`${method} ${endpoint}`);
  middlewares.forEach((middleware) => middleware());
```

```
apiCall(
   "/users",
   "POST",
   authMiddleware,
   validationMiddleware,
   loggingMiddleware
);

// Event handler with rest
function handleFormSubmit(event, ...validators) {
   event.preventDefault();

   const isValid = validators.every((validator) => validator());

   if (isValid) {
      console.log("Form is valid, submitting...");
   }
}
```

#### **Practical Combinations**

```
// Array manipulation utilities
function removeItem(array, index) {
  return [...array.slice(∅, index), ...array.slice(index + 1)];
}
function insertItem(array, index, item) {
  return [...array.slice(∅, index), item, ...array.slice(index)];
}
function updateItem(array, index, newItem) {
  return array.map((item, i) => (i === index ? newItem : item));
}
// Object utilities
function omit(obj, ...keys) {
 const { [keys[0]]: omitted, ...rest } = obj;
  return keys.length > 1 ? omit(rest, ...keys.slice(1)) : rest;
}
function pick(obj, ...keys) {
 return keys.reduce((result, key) => {
    if (key in obj) {
      result[key] = obj[key];
    return result;
  }, {});
}
```

```
// Usage
const user = {
  name: "Alice",
  age: 30,
  email: "alice@example.com",
  password: "secret",
};
const publicUser = omit(user, "password"); // { name: 'Alice', age: 30, email:
'alice@example.com' }
const basicInfo = pick(user, "name", "age"); // { name: 'Alice', age: 30 }
// Redux-style state updates
function updateUserState(state, action) {
  switch (action.type) {
    case "UPDATE_USER":
      return {
        ...state,
        user: {
          ...state.user,
         ...action.payload,
        },
      };
    case "ADD_ITEM":
      return {
        ...state,
        items: [...state.items, action.payload],
      };
    case "REMOVE_ITEM":
      return {
        ...state,
        items: state.items.filter((item) => item.id !== action.payload.id),
      };
    default:
      return state;
  }
}
```

# **a** Enhanced Object Literals

### **Shorthand Properties and Methods**

```
// Old way
const name = "Alice";
const age = 30;
const email = "alice@example.com";

const oldUser = {
   name: name,
```

```
age: age,
 email: email,
 greet: function () {
  return "Hello!";
 },
};
// New way - shorthand properties
const newUser = {
 name, // Same as name: name
 age, // Same as age: age
 email, // Same as email: email
 greet() {
  // Same as greet: function()
  return "Hello!";
 },
};
// Method definitions
const calculator = {
 value: ∅,
 add(n) {
  this.value += n;
   return this;
 },
 multiply(n) {
   this.value *= n;
   return this;
  },
 get result() {
   return this.value;
  },
 set reset(val) {
   this.value = val | 0;
 },
};
// Usage: calculator.add(5).multiply(2).result; // 10
// Async methods
const apiService = {
  baseUrl: "https://api.example.com",
 async fetchUser(id) {
   const response = await fetch(`${this.baseUrl}/users/${id}`);
  return response.json();
 },
  async createUser(userData) {
    const response = await fetch(`${this.baseUrl}/users`, {
```

```
method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(userData),
    });
    return response.json();
    },
};
```

#### **Computed Property Names**

```
// Dynamic property names
const propertyName = "dynamicKey";
const value = "dynamicValue";
const obj = {
  [propertyName]: value,
  [`${propertyName}_modified`]: value.toUpperCase(),
  [propertyName + "_count"]: 1,
};
// { dynamicKey: 'dynamicValue', dynamicKey_modified: 'DYNAMICVALUE',
dynamicKey_count: 1 }
// Function-based property names
function getPropertyName(prefix, suffix) {
  return `${prefix}_${suffix}`;
}
const config = {
  [getPropertyName("api", "url")]: "https://api.example.com",
  [getPropertyName("api", "key")]: "secret-key",
};
// { api_url: 'https://api.example.com', api_key: 'secret-key' }
// Symbol properties
const uniqueId = Symbol("id");
const metadata = Symbol("metadata");
const user = {
 name: "Alice",
  [uniqueId]: 12345,
  [metadata]: { created: new Date(), version: 1 },
};
// Computed method names
const actions = ["create", "read", "update", "delete"];
const api = {};
actions.forEach((action) => {
  api[`${action}User`] = function (data) {
    console.log(`${action.toUpperCase()} user:`, data);
  };
```

```
});
// api.createUser, api.readUser, api.updateUser, api.deleteUser
// React component with dynamic handlers
class FormComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {};
    // Create handlers for each field
    props.fields.forEach((field) => {
      this[`handle${field.charAt(0).toUpperCase() + field.slice(1)}Change`] = (
        е
      ) => {
       this.setState({ [field]: e.target.value });
    });
  }
}
// Practical example: Form validation
function createValidator(rules) {
  return {
    [Symbol.toStringTag]: "Validator",
    validate(data) {
      const errors = {};
      Object.keys(rules).forEach((field) => {
        const rule = rules[field];
        const value = data[field];
        if (rule.required && !value) {
          errors[field] = `${field} is required`;
        } else if (rule.minLength && value.length < rule.minLength) {</pre>
          errors[
            field
          ] = `${field} must be at least ${rule.minLength} characters`;
      });
      return {
        isValid: Object.keys(errors).length === 0,
        errors,
      };
    },
 };
}
const userValidator = createValidator({
 name: { required: true, minLength: 2 },
  email: { required: true },
  password: { required: true, minLength: 8 },
});
```

2025-07-22 javascript.md



# Modules (Import/Export)

### Named Exports and Imports

```
// math.js - Named exports
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;
}
export class Calculator {
  constructor() {
    this.value = 0;
  }
  add(n) {
   this.value += n;
    return this;
  }
}
// Alternative export syntax
const subtract = (a, b) \Rightarrow a - b;
const divide = (a, b) => a / b;
export { subtract, divide };
// Exporting with different names
const power = (base, exponent) => Math.pow(base, exponent);
export { power as pow };
// main.js - Named imports
import { PI, add, multiply, Calculator } from "./math.js";
import { subtract, divide, pow } from "./math.js";
// Using imported functions
console.log(add(5, 3)); // 8
console.log(multiply(4, 7)); // 28
console.log(PI); // 3.14159
const calc = new Calculator();
calc.add(10).add(5);
// Importing with different names
```

```
import { pow as power } from "./math.js";
console.log(power(2, 3)); // 8

// Importing all named exports
import * as MathUtils from "./math.js";
console.log(MathUtils.PI);
console.log(MathUtils.add(1, 2));

// Selective imports
import { add, multiply } from "./math.js";
```

### **Default Exports and Imports**

```
// user.js - Default export
class User {
   constructor(name, email) {
       this.name = name;
       this.email = email;
    }
    greet() {
        return `Hello, I'm ${this.name}`;
}
export default User;
// Alternative default export syntax
const createUser = (name, email) => new User(name, email);
export { createUser as default };
// Or inline default export
export default function validateEmail(email) {
    return email.includes('@');
}
// main.js - Default imports
import User from './user.js';
                                  // Import default export
// Can use any name
import MyUser from './user.js';
import validateEmail from './validator.js'; // Import default function
const user = new User('Alice', 'alice@example.com');
console.log(user.greet());
// Mixed imports (default + named)
// utils.js
export default function log(message) {
   console.log(`[LOG]: ${message}`);
}
export const version = '1.0.0';
```

```
export const author = 'John Doe';

// main.js
import log, { version, author } from './utils.js';

log(`App version: ${version} by ${author}`);
```

#### Re-exports and Module Aggregation

```
// api/users.js
export async function getUsers() {
 const response = await fetch("/api/users");
 return response.json();
}
export async function createUser(userData) {
  const response = await fetch("/api/users", {
   method: "POST",
   headers: { "Content-Type": "application/json" },
   body: JSON.stringify(userData),
 });
 return response.json();
}
// api/posts.js
export async function getPosts() {
 const response = await fetch("/api/posts");
 return response.json();
}
export async function createPost(postData) {
 const response = await fetch("/api/posts", {
   method: "POST",
   headers: { "Content-Type": "application/json" },
   body: JSON.stringify(postData),
 });
 return response.json();
}
// api/index.js - Aggregating exports
export * from "./users.js"; // Re-export all named exports
export * from "./posts.js";
// Or selective re-exports
export { getUsers, createUser } from "./users.js";
export { getPosts, createPost } from "./posts.js";
// Re-export with renaming
export { getUsers as fetchUsers } from "./users.js";
// Re-export default as named
```

```
export { default as UserClass } from "./user.js";

// main.js - Clean imports
import { getUsers, createUser, getPosts, createPost } from "./api/index.js";

// Or namespace import
import * as API from "./api/index.js";

API.getUsers().then((users) => console.log(users));
```

# **Dynamic Imports**

```
// Dynamic imports for code splitting
async function loadModule() {
 try {
    const module = await import("./heavy-module.js");
   module.doSomething();
 } catch (error) {
   console.error("Failed to load module:", error);
  }
}
// Conditional loading
if (condition) {
  import("./feature-module.js")
    .then((module) => {
      module.initializeFeature();
    })
    .catch((error) => {
      console.error("Feature not available:", error);
    });
}
// React lazy loading
const LazyComponent = React.lazy(() => import("./LazyComponent.js"));
function App() {
 return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
 );
// Dynamic import with destructuring
async function loadUtilities() {
 const { add, multiply, PI } = await import("./math.js");
 console.log(add(2, 3));
 console.log(PI);
}
// Module loading based on environment
```

```
const isDevelopment = process.env.NODE_ENV === "development";
const logger = isDevelopment
  ? await import("./dev-logger.js")
  : await import("./prod-logger.js");
logger.log("Application started");
// Plugin system with dynamic imports
class PluginManager {
 constructor() {
   this.plugins = new Map();
 }
 async loadPlugin(name) {
   try {
      const plugin = await import(`./plugins/${name}.js`);
      this.plugins.set(name, plugin.default || plugin);
      console.log(`Plugin ${name} loaded successfully`);
    } catch (error) {
      console.error(`Failed to load plugin ${name}:`, error);
   }
 }
 getPlugin(name) {
   return this.plugins.get(name);
 }
}
const pluginManager = new PluginManager();
await pluginManager.loadPlugin("analytics");
await pluginManager.loadPlugin("authentication");
```

# **a** Classes

### **Basic Class Syntax**

```
// ES6 Class
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

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

  haveBirthday() {
    this.age++;
    return `Happy birthday! I'm now ${this.age}.`;
}
```

```
// Static method
  static createAdult(name) {
    return new Person(name, 18);
  }
 // Getter
  get description() {
    return `${this.name} (${this.age} years old)`;
 // Setter
 set fullName(name) {
  this.name = name;
}
// Usage
const person = new Person("Alice", 30);
console.log(person.greet());
console.log(person.haveBirthday());
console.log(person.description);
person.fullName = "Alice Smith";
console.log(person.description);
const adult = Person.createAdult("Bob");
console.log(adult.age); // 18
```

#### Inheritance

```
// Base class
class Animal {
  constructor(name, species) {
    this.name = name;
    this.species = species;
}

makeSound() {
  return `${this.name} makes a sound`;
}

move() {
  return `${this.name} moves`;
}

toString() {
  return `${this.name} the ${this.species}`;
}
```

```
// Derived class
class Dog extends Animal {
  constructor(name, breed) {
    super(name, "Dog"); // Call parent constructor
   this.breed = breed;
  }
  makeSound() {
   return `${this.name} barks: Woof!`;
  }
  fetch() {
   return `${this.name} fetches the ball`;
  }
 // Override toString
 toString() {
    return `${super.toString()} (${this.breed})`;
 }
}
class Cat extends Animal {
 constructor(name, isIndoor = true) {
   super(name, "Cat");
   this.isIndoor = isIndoor;
  }
 makeSound() {
   return `${this.name} meows: Meow!`;
 climb() {
    return `${this.name} climbs the tree`;
  }
}
// Usage
const dog = new Dog("Buddy", "Golden Retriever");
const cat = new Cat("Whiskers");
console.log(dog.makeSound()); // "Buddy barks: Woof!"
console.log(cat.makeSound()); // "Whiskers meows: Meow!"
console.log(dog.toString()); // "Buddy the Dog (Golden Retriever)"
// Polymorphism
const animals = [dog, cat];
animals.forEach((animal) => {
 console.log(animal.makeSound()); // Different behavior for each
});
// instanceof checks
console.log(dog instanceof Dog); // true
```

```
console.log(dog instanceof Animal); // true
console.log(cat instanceof Dog); // false
```

### Private Fields and Methods (ES2022)

```
class BankAccount {
 // Private fields
 #balance = ∅;
 #accountNumber;
 #transactions = [];
 constructor(accountNumber, initialBalance = 0) {
   this.#accountNumber = accountNumber;
   this.#balance = initialBalance;
   this.#addTransaction("Initial deposit", initialBalance);
 }
 // Private method
 #addTransaction(type, amount) {
   this.#transactions.push({
     type,
      amount,
     timestamp: new Date(),
     balance: this.#balance,
   });
  }
 // Public methods
 deposit(amount) {
   if (amount <= 0) {
     throw new Error("Deposit amount must be positive");
    }
   this.#balance += amount;
   this.#addTransaction("Deposit", amount);
   return this. #balance;
 }
 withdraw(amount) {
   if (amount <= 0) {
      throw new Error("Withdrawal amount must be positive");
    }
   if (amount > this.#balance) {
     throw new Error("Insufficient funds");
    }
   this.#balance -= amount;
   this.#addTransaction("Withdrawal", -amount);
    return this. #balance;
```

```
// Getter for balance (read-only access)
  get balance() {
   return this.#balance;
 get accountNumber() {
   return this. #accountNumber;
 getTransactionHistory() {
   // Return copy to prevent external modification
   return [...this.#transactions];
 }
}
// Usage
const account = new BankAccount("12345", 1000);
console.log(account.balance); // 1000
account.deposit(500);
console.log(account.balance); // 1500
account.withdraw(200);
console.log(account.balance); // 1300
// These would throw errors:
// console.log(account.#balance); // SyntaxError: Private field '#balance'
must be declared in an enclosing class
// account.#addTransaction('test', 100); // SyntaxError
console.log(account.getTransactionHistory());
```

#### Static Fields and Methods

```
class MathUtils {
   // Static fields
   static PI = 3.14159;
   static E = 2.71828;
   static #precision = 10; // Private static field

   // Static methods
   static add(a, b) {
     return a + b;
   }

   static multiply(a, b) {
     return a * b;
   }

   static round(number, decimals = 2) {
```

```
return Math.round(number * Math.pow(10, decimals)) / Math.pow(10, decimals);
  }
 // Private static method
  static #validateNumber(num) {
   if (typeof num !== "number" || isNaN(num)) {
     throw new Error("Invalid number");
    }
  }
 static circleArea(radius) {
   this.#validateNumber(radius);
   return this.round(this.PI * radius * radius);
  }
 // Static getter
  static get precision() {
   return this. #precision;
  }
 // Static setter
  static set precision(value) {
    if (value > 0 && value <= 20) {
     this.#precision = value;
   }
 }
}
// Usage
console.log(MathUtils.PI); // 3.14159
console.log(MathUtils.add(5, 3)); // 8
console.log(MathUtils.circleArea(5)); // 78.54
console.log(MathUtils.precision); // 10
MathUtils.precision = 5;
console.log(MathUtils.precision); // 5
// Counter example with static fields
class Counter {
  static #count = 0;
  static #instances = [];
 constructor(name) {
   this.name = name;
   this.id = ++Counter.#count;
    Counter.#instances.push(this);
  }
  static getCount() {
   return Counter.#count;
  }
  static getAllInstances() {
    return [...Counter.#instances];
```

```
static reset() {
    Counter.#count = 0;
    Counter.#instances = [];
}

const counter1 = new Counter("First");
const counter2 = new Counter("Second");

console.log(Counter.getCount()); // 2
console.log(Counter.getAllInstances()); // [Counter, Counter]
```

## Mixins and Composition

```
// Mixin functions
const Flyable = {
  fly() {
   return `${this.name} is flying`;
 },
 land() {
   return `${this.name} has landed`;
 },
};
const Swimmable = {
  swim() {
   return `${this.name} is swimming`;
 },
 dive() {
   return `${this.name} dives underwater`;
 },
};
const Walkable = {
 walk() {
   return `${this.name} is walking`;
 },
 run() {
  return `${this.name} is running`;
 },
};
// Mixin helper function
function mixin(target, ...sources) {
  sources.forEach((source) => {
    Object.getOwnPropertyNames(source).forEach((name) => {
```

```
if (name !== "constructor") {
        target.prototype[name] = source[name];
   });
  });
 return target;
// Base class
class Animal {
 constructor(name) {
   this.name = name;
 }
}
// Classes with mixins
class Bird extends Animal {
 constructor(name, wingspan) {
    super(name);
    this.wingspan = wingspan;
 }
}
class Duck extends Animal {
 constructor(name) {
    super(name);
 }
}
class Human extends Animal {
 constructor(name, age) {
   super(name);
   this.age = age;
  }
}
// Apply mixins
mixin(Bird, Flyable, Walkable);
mixin(Duck, Flyable, Swimmable, Walkable);
mixin(Human, Walkable, Swimmable);
// Usage
const eagle = new Bird("Eagle", 200);
const mallard = new Duck("Mallard");
const person = new Human("Alice", 30);
console.log(eagle.fly()); // "Eagle is flying"
console.log(eagle.walk()); // "Eagle is walking"
console.log(mallard.fly()); // "Mallard is flying"
console.log(mallard.swim()); // "Mallard is swimming"
console.log(mallard.walk()); // "Mallard is walking"
console.log(person.walk()); // "Alice is walking"
```

```
console.log(person.swim()); // "Alice is swimming"
// console.log(person.fly()); // Error: person.fly is not a function

// Factory function approach
function createFlyingAnimal(name, type) {
   const animal = {
      name,
      type,
      ...Flyable,
      ...Walkable,
   };

   return animal;
}

const bat = createFlyingAnimal("Bat", "Mammal");
console.log(bat.fly()); // "Bat is flying"
```

## 

#### 1. Arrow Function Context Issues

```
// X Using arrow functions where you need dynamic `this`
const button = {
 text: "Click me",
 click: () => {
    console.log(this.text); // `this` is not the button object
 },
};
// Use regular function for object methods
const buttonFixed = {
 text: "Click me",
  click() {
    console.log(this.text); // Works correctly
 },
};
// X Arrow functions as constructors
const Person = (name) => {
 this.name = name; // Error: Arrow functions cannot be constructors
};
// ✓ Use regular function or class
function PersonConstructor(name) {
 this.name = name;
}
class PersonClass {
  constructor(name) {
    this.name = name;
```

```
} }
```

## 2. Destructuring with Default Values

```
// X Incorrect default value syntax
function processUser({ name = "Anonymous", age = 0, email }) {
 // This works, but what if the entire object is undefined?
}
processUser(); // TypeError: Cannot destructure property 'name' of 'undefined'
// Provide default for the entire parameter
function processUserFixed({ name = "Anonymous", age = 0, email } = {}) {
  console.log(name, age, email);
}
processUserFixed(); // Works: "Anonymous 0 undefined"
// X Nested destructuring without defaults
function getAddress({
 user: {
   address: { street },
 },
}) {
 return street;
getAddress({ user: {} }); // TypeError: Cannot destructure property 'street' of
'undefined'
// ✓ Provide defaults at each level
function getAddressFixed({ user: { address: { street } = {} } = {} } = {}) {
  return street;
}
getAddressFixed({ user: {} }); // undefined (no error)
```

## 3. Module Import/Export Issues

```
// X Mixing default and named exports incorrectly
// utils.js
export default function log(message) {
   console.log(message);
}

export const version = "1.0.0";

// main.js - Wrong import
```

```
import { log, version } from "./utils.js"; // Error: log is default export
// Correct import
import log, { version } from "./utils.js";
// X Circular dependencies
// a.js
import { b } from "./b.js";
export const a = "A";
// b.js
import { a } from "./a.js"; // Circular dependency
export const b = "B";
// Avoid circular dependencies or use dynamic imports
export const b = "B";
// Later, when needed:
async function useA() {
 const { a } = await import("./a.js");
 console.log(a);
}
```

#### 4. Class Inheritance Pitfalls

```
// X Forgetting to call super() in constructor
class Animal {
 constructor(name) {
   this.name = name;
 }
}
class Dog extends Animal {
 constructor(name, breed) {
    // Missing super(name) call
   this.breed = breed; // ReferenceError: Must call super constructor
}
// // Always call super() first
class DogFixed extends Animal {
 constructor(name, breed) {
    super(name); // Call parent constructor first
    this.breed = breed;
 }
}
// X Incorrect method overriding
class Calculator {
  add(a, b) {
```

```
return a + b;
 }
}
class ScientificCalculator extends Calculator {
 add(a, b, c = 0) {
    // Different signature
    return super.add(a, b) + c;
 }
}
// This can cause confusion when used polymorphically
const calc = new ScientificCalculator();
console.log(calc.add(1, 2)); // Works, but third parameter is ignored
// ✓ Maintain consistent method signatures
class ScientificCalculatorFixed extends Calculator {
  add(a, b) {
    return super.add(a, b);
 addThree(a, b, c) {
   return this.add(a, b) + c;
 }
}
```

#### 5. Template Literal Gotchas

```
// X Unintended expression evaluation
const userInput = '${alert("XSS")}';
const message = `Hello ${userInput}`; // Safe - treated as string
// But be careful with eval-like functions
const dangerousTemplate = `Hello ${eval(userInput)}`; // Dangerous!
// ✓ Always sanitize user input
function sanitize(input) {
  return input.replace(/[<>"'&]/g, (char) => {
    const entities = {
      "<": "&lt;",
      ">": ">",
      '"': """,
      "'": "'",
      "&": "&",
   return entities[char];
 });
}
const safeMessage = `Hello ${sanitize(userInput)}`;
```

```
// X Performance issues with large templates
function generateLargeHTML(items) {
    let html = "";
    items.forEach((item) => {
        html += `<div>${item.name}</div>`; // String concatenation in loop
    });
    return html;
}

// Use array join for better performance
function generateLargeHTMLFixed(items) {
    return items.map((item) => `<div>${item.name}</div>`).join("");
}
```

# Mini Practice Problems

## Problem 1: Advanced Destructuring

```
// Create a function that extracts and transforms data from a complex API response
function processApiResponse(response) {
 // Your implementation here
 // Extract: user name, email, first hobby, remaining hobbies, and address city
 // Handle cases where data might be missing
 // Return an object with: userName, userEmail, primaryHobby, otherHobbies, city
// Test data:
const apiResponse = {
 data: {
    user: {
      profile: {
        name: "Alice Johnson",
        contact: {
          email: "alice@example.com",
          phone: "123-456-7890",
       },
      },
      preferences: {
        hobbies: ["reading", "swimming", "photography"],
        settings: {
          theme: "dark",
        },
      },
      location: {
        address: {
         street: "123 Main St",
         city: "New York",
         country: "USA",
        },
     },
    },
```

```
meta: {
    timestamp: "2023-01-01T00:00:00Z",
    },
};

// Expected output:
// {
    userName: 'Alice Johnson',
    userEmail: 'alice@example.com',
    primaryHobby: 'reading',
    otherHobbies: ['swimming', 'photography'],
    city: 'New York'
// }
```

#### Problem 2: Class with Mixins

```
// Create a Vehicle class system with mixins for different capabilities
// Base Vehicle class
class Vehicle {
 constructor(make, model, year) {
   // Your implementation
 }
}
// Mixins
const Drivable = {
 // Add methods: start(), stop(), accelerate(speed), brake()
};
const Flyable = {
 // Add methods: takeOff(), land(), setAltitude(altitude)
};
const Floatable = {
// Add methods: launch(), dock(), setDepth(depth)
};
// Create specific vehicle types:
// Car (Drivable)
// Airplane (Drivable, Flyable)
// Boat (Drivable, Floatable)
// AmphibiousVehicle (Drivable, Flyable, Floatable)
// Test your implementation:
const car = new Car("Toyota", "Camry", 2023);
const plane = new Airplane("Boeing", "737", 2022);
const boat = new Boat("Yamaha", "WaveRunner", 2023);
const amphibious = new AmphibiousVehicle("DUKW", "Amphibian", 1944);
```

```
// All should work:
car.start();
car.accelerate(60);

plane.start();
plane.takeOff();
plane.setAltitude(30000);

boat.launch();
boat.accelerate(25);

amphibious.start();
amphibious.accelerate(30);
amphibious.takeOff();
amphibious.launch();
```

## Problem 3: Module System

```
// Create a plugin system using ES6 modules
// plugin-manager.js
// Create a PluginManager class that can:
// - Register plugins dynamically
// - Load plugins from files
// - Execute plugin hooks
// - Handle plugin dependencies
class PluginManager {
    constructor() {
        // Your implementation
    }
    async loadPlugin(pluginPath) {
        // Load plugin from file path
    }
    registerPlugin(name, plugin) {
        // Register plugin manually
    }
    executeHook(hookName, ...args) {
        // Execute all plugins that have this hook
    }
    getPlugin(name) {
        // Get specific plugin
}
// Example plugins:
// plugins/logger.js
```

```
export default {
    name: 'logger',
    hooks: {
        beforeRequest: (url) => console.log(`Making request to: ${url}`),
        afterRequest: (response) => console.log(`Response status:
${response.status}`)
   }
};
// plugins/cache.js
export default {
    name: 'cache',
    dependencies: ['logger'],
    hooks: {
        beforeRequest: (url) => {
            // Check cache
        },
        afterRequest: (response) => {
            // Store in cache
    }
};
// Usage:
const pm = new PluginManager();
await pm.loadPlugin('./plugins/logger.js');
await pm.loadPlugin('./plugins/cache.js');
pm.executeHook('beforeRequest', 'https://api.example.com/users');
```

## **Problem 4: Advanced Template Literals**

```
// Create a template engine using tagged template literals
// Your implementation should support:
// - Variable interpolation
// - Conditional rendering
// - Loop rendering
// - Nested templates
// - Escaping for security
function template(strings, ...values) {
   // Your implementation here
    // Should handle special syntax like:
    // ${if condition}...${endif}
   // ${for item in items}...${endfor}
    // ${include 'template-name'}
}
// Register sub-templates
template.register = function(name, templateFn) {
```

```
// Register named templates
};
// Example usage:
const users = [
   { name: 'Alice', active: true, role: 'admin' },
   { name: 'Bob', active: false, role: 'user' },
    { name: 'Charlie', active: true, role: 'user' }
];
const userListTemplate = template`
    <div class="user-list">
        <h2>Users (${users.length})</h2>
        ${for user in users}
            <div class="user ${user.active ? 'active' : 'inactive'}">
                <h3>${user.name}</h3>
                Role: ${user.role}
                ${if user.role === 'admin'}
                    <span class="badge">Administrator</span>
                ${endif}
            </div>
        ${endfor}
   </div>
console.log(userListTemplate);
```

## Problem 5: Async Class with Error Handling

```
// Create a DataService class that handles async operations with proper error
handling
class DataService {
 constructor(baseUrl, options = {}) {
   // Your implementation
   // Should support: timeout, retries, caching, rate limiting
 }
 async get(endpoint, options = {}) {
  // GET request with error handling
  }
 async post(endpoint, data, options = {}) {
   // POST request with error handling
  }
 async put(endpoint, data, options = {}) {
  // PUT request with error handling
  }
  async delete(endpoint, options = {}) {
```

```
// DELETE request with error handling
  }
 // Private methods for:
 // - Retry logic
 // - Rate limiting
 // - Caching
 // - Request/response transformation
// Usage should support:
const api = new DataService("https://api.example.com", {
 timeout: 5000,
 retries: 3,
 cache: true,
  rateLimit: { requests: 100, per: "minute" },
});
// All methods should return promises and handle errors gracefully
try {
 const users = await api.get("/users");
 const newUser = await api.post("/users", {
   name: "Alice",
   email: "alice@example.com",
 });
 const updatedUser = await api.put(`/users/${newUser.id}`, {
   name: "Alice Smith",
 });
 await api.delete(`/users/${newUser.id}`);
} catch (error) {
  console.error("API operation failed:", error);
}
```

## Interview Notes

#### **Common Questions:**

#### Q: What's the difference between let, const, and var?

- var: Function-scoped, hoisted, can be redeclared
- let: Block-scoped, hoisted but in temporal dead zone, cannot be redeclared
- const: Block-scoped, hoisted but in temporal dead zone, cannot be reassigned or redeclared

## Q: When should you use arrow functions vs regular functions?

- Arrow functions: Callbacks, array methods, when you want lexical this
- Regular functions: Object methods, constructors, when you need dynamic this

#### Q: What are the benefits of destructuring?

- Cleaner code, easier data extraction, default values, swapping variables
- Reduces repetitive property access, makes function parameters more readable

#### Q: How do ES6 modules differ from CommonJS?

- ES6: Static analysis, tree shaking, top-level await, import/export syntax
- CommonJS: Dynamic loading, require/module.exports, synchronous

#### Q: What are the advantages of ES6 classes over function constructors?

- Cleaner syntax, built-in inheritance with extends, static methods, private fields
- Better error messages, no accidental function calls without new

## Asked at Companies:

- Google: "Implement a module bundler that handles ES6 imports"
- Facebook: "Create a React-like component system using ES6 classes"
- Amazon: "Design a plugin architecture using ES6 modules and dynamic imports"
- Microsoft: "Implement a template engine using tagged template literals"
- Netflix: "Create a data fetching library with classes and async/await"

# **6** Key Takeaways

- 1. Arrow functions Great for callbacks and when you need lexical this
- 2. Template literals Much cleaner than string concatenation
- 3. **Destructuring** Powerful way to extract data from arrays and objects
- 4. Spread/Rest Flexible operators for arrays, objects, and function parameters
- 5. Modules Proper way to organize and share code
- 6. Classes Clean syntax for object-oriented programming
- 7. Enhanced object literals Shorthand properties and computed names
- 8. **Default parameters** Cleaner function definitions

**Previous Chapter**: ← Error Handling & Debugging

**Next Chapter**: DOM Manipulation →

**Practice**: Try the ES6+ problems and experiment with modern JavaScript features!

# **邑** Chapter 13: DOM Manipulation

Master the art of dynamically controlling web page content and user interactions.

## Plain English Explanation

DOM manipulation is like being a stage director for a theater production:

- **Selecting elements** = finding specific actors on stage
- Changing content = giving actors new lines to say
- Modifying styles = changing costumes and lighting
- Adding/removing elements = bringing actors on/off stage
- **Event handling** = responding to audience reactions
- Animation = choreographing smooth movements

The DOM (Document Object Model) is the browser's representation of your HTML page as a tree of objects that JavaScript can manipulate.

# **©** Selecting Elements

#### **Basic Selection Methods**

```
// By ID (returns single element or null)
const header = document.getElementById("main-header");
const loginForm = document.getElementById("login-form");
// By class name (returns HTMLCollection - array-like)
const buttons = document.getElementsByClassName("btn");
const cards = document.getElementsByClassName("card");
// By tag name (returns HTMLCollection)
const paragraphs = document.getElementsByTagName("p");
const images = document.getElementsByTagName("img");
// By name attribute (returns NodeList)
const radioButtons = document.getElementsByName("gender");
const checkboxes = document.getElementsByName("interests");
// Modern selectors (CSS-style)
const firstButton = document.querySelector(".btn"); // First match
const allButtons = document.querySelectorAll(".btn"); // All matches (NodeList)
const specificButton = document.querySelector("#submit-btn"); // By ID
const nestedElement = document.querySelector(".container .card .title");
// Advanced CSS selectors
const evenRows = document.querySelectorAll("tr:nth-child(even)");
const firstChild = document.querySelector(".menu > li:first-child");
const lastInput = document.querySelector("input:last-of-type");
const checkedBoxes = document.querySelectorAll(
  'input[type="checkbox"]:checked'
);
const externalLinks = document.querySelectorAll('a[href^="http"]');
// Attribute selectors
const requiredFields = document.querySelectorAll("input[required]");
const dataElements = document.querySelectorAll('[data-category="electronics"]');
const partialMatch = document.querySelectorAll('[class*="btn"]');
```

#### **Element Relationships and Navigation**

```
// Parent/child relationships
const element = document.querySelector(".target");

// Parent navigation
const parent = element.parentElement; // Direct parent
```

```
const parentNode = element.parentNode; // Parent node (includes text nodes)
const closestContainer = element.closest(".container"); // Nearest ancestor
matching selector
// Child navigation
const children = element.children; // HTMLCollection of child elements
const childNodes = element.childNodes; // NodeList including text nodes
const firstChild = element.firstElementChild; // First child element
const lastChild = element.lastElementChild; // Last child element
// Sibling navigation
const nextSibling = element.nextElementSibling; // Next sibling element
const prevSibling = element.previousElementSibling; // Previous sibling element
const allSiblings = Array.from(element.parentElement.children).filter(
  (child) => child !== element
);
// Practical examples
function highlightSiblings(element) {
  const siblings = Array.from(element.parentElement.children);
 siblings.forEach((sibling) => {
    if (sibling !== element) {
      sibling.classList.add("highlighted");
 });
}
function findFormContainer(inputElement) {
  return (
    inputElement.closest("form") || inputElement.closest(".form-container")
  );
}
function getTableRow(cellElement) {
  return cellElement.closest("tr");
}
// Walking the DOM tree
function walkDOM(node, callback) {
  callback(node);
 for (let child of node.children) {
    walkDOM(child, callback);
  }
}
// Usage: Find all elements with specific data attribute
walkDOM(document.body, (element) => {
  if (element.dataset && element.dataset.trackable) {
    console.log("Trackable element:", element);
  }
});
```

#### Modern Selection Patterns

```
// Utility functions for common selections
const $ = (selector) => document.querySelector(selector);
const $$ = (selector) => document.querySelectorAll(selector);
// Usage
const header = $("#main-header");
const buttons = $$(".btn");
// Converting NodeList to Array for array methods
const buttonArray = Array.from($$(".btn"));
const activeButtons = [...$$(".btn")].filter((btn) => !btn.disabled);
// Chaining selections
const menuItems = Array.from($(".navigation").children)
  .filter((item) => item.tagName === "LI")
  .map((item) => item.querySelector("a"))
  .filter((link) => link !== null);
// Conditional selection
function getElement(selector, fallback = null) {
  const element = document.querySelector(selector);
  return element || (fallback && document.querySelector(fallback));
}
const mainContent = getElement("#main-content", ".content");
// Selection with error handling
function safeSelect(selector) {
 try {
   const elements = document.querySelectorAll(selector);
   return Array.from(elements);
  } catch (error) {
    console.error("Invalid selector:", selector, error);
    return [];
 }
}
// Scoped selection (within a container)
function selectWithin(container, selector) {
  if (typeof container === "string") {
    container = document.querySelector(container);
  }
 return container ? Array.from(container.querySelectorAll(selector)) : [];
}
const formInputs = selectWithin("#user-form", "input, select, textarea");
const cardButtons = selectWithin(".card-container", ".btn");
```



## **Tontent Manipulation**

#### **Text Content**

```
// Getting and setting text content
const heading = document.querySelector("h1");
// textContent - gets/sets text only (no HTML)
console.log(heading.textContent); // "Welcome to Our Site"
heading.textContent = "New Heading"; // Safe - no HTML injection
// innerText - respects styling (hidden elements ignored)
console.log(heading.innerText); // Visible text only
heading.innerText = "Styled Heading";
// innerHTML - gets/sets HTML content (dangerous with user input)
const container = document.querySelector(".content");
console.log(container.innerHTML); // "Hello <strong>World</strong>"
container.innerHTML = "New <em>content</em>";
// Safe HTML insertion
function safeSetHTML(element, htmlString) {
  // Create a temporary element to parse HTML
  const temp = document.createElement("div");
 temp.innerHTML = htmlString;
 // Clear existing content
 element.innerHTML = "";
 // Move parsed nodes to target element
 while (temp.firstChild) {
    element.appendChild(temp.firstChild);
 }
}
// Text manipulation utilities
function appendText(element, text) {
  element.textContent += text;
}
function prependText(element, text) {
  element.textContent = text + element.textContent;
function replaceText(element, oldText, newText) {
  element.textContent = element.textContent.replace(oldText, newText);
}
// Practical examples
function updateCounter(element, count) {
  element.textContent = `Count: ${count}`;
```

```
function formatPrice(element, price) {
  element.textContent = `$${price.toFixed(2)}`;
}
function truncateText(element, maxLength) {
  const text = element.textContent;
  if (text.length > maxLength) {
    element.textContent = text.substring(0, maxLength) + "...";
    element.title = text; // Show full text on hover
 }
}
// Dynamic content updates
function updateStatus(statusElement, status, message) {
  statusElement.textContent = message;
  statusElement.className = `status status-${status}`;
}
// Usage
updateStatus($(".status"), "success", "Operation completed successfully!");
updateStatus($(".status"), "error", "Something went wrong.");
```

#### **HTML Content and Templates**

```
// Template-based content creation
function createUserCard(user) {
  return `
        <div class="user-card" data-user-id="${user.id}">
           <img src="${user.avatar}" alt="${user.name}" class="avatar">
           <h3 class="name">${user.name}</h3>
           ${user.email}
           <div class="actions">
               <button class="btn btn-primary" onclick="editUser(${user.id})">
                   Edit
               </button>
               <button class="btn btn-danger" onclick="deleteUser(${user.id})">
                   Delete
               </button>
           </div>
       </div>
}
// Safe template rendering
function renderTemplate(container, template, data) {
 if (typeof container === "string") {
   container = document.querySelector(container);
  }
  const html = typeof template === "function" ? template(data) : template;
```

```
container.innerHTML = html;
}
// List rendering
function renderUserList(users) {
  const container = document.querySelector("#user-list");
 const html = users.map((user) => createUserCard(user)).join("");
  container.innerHTML = html;
}
// Conditional rendering
function renderContent(container, data) {
  if (!data || data.length === 0) {
   container.innerHTML = 'No data available';
   return;
  }
 const html = data.map((item) => createItemTemplate(item)).join("");
  container.innerHTML = html;
}
// Template with escaping for security
function escapeHtml(text) {
  const div = document.createElement("div");
 div.textContent = text;
 return div.innerHTML;
}
function createSafeUserCard(user) {
  return `
        <div class="user-card">
            <h3>${escapeHtml(user.name)}</h3>
            ${escapeHtml(user.email)}
            ${escapeHtml(user.bio)}
        </div>
}
// Using HTML templates (modern approach)
function createElementFromTemplate(templateId, data) {
  const template = document.querySelector(`#${templateId}`);
  const clone = template.content.cloneNode(true);
  // Fill in data
  Object.keys(data).forEach((key) => {
   const element = clone.querySelector(`[data-field="${key}"]`);
   if (element) {
      element.textContent = data[key];
   }
 });
  return clone;
}
```

## Style Manipulation

#### **CSS Classes**

```
const element = document.querySelector(".target");
// Class manipulation methods
element.classList.add("active"); // Add class
element.classList.remove("hidden"); // Remove class
element.classList.toggle("expanded"); // Toggle class
element.classList.replace("old", "new"); // Replace class
// Check if class exists
if (element.classList.contains("active")) {
 console.log("Element is active");
}
// Multiple classes
element.classList.add("class1", "class2", "class3");
element.classList.remove("class1", "class2");
// Conditional class manipulation
function setActiveState(element, isActive) {
  element.classList.toggle("active", isActive);
  element.classList.toggle("inactive", !isActive);
}
// Class utilities
function hasAnyClass(element, classes) {
  return classes.some((className) => element.classList.contains(className));
}
function hasAllClasses(element, classes) {
  return classes.every((className) => element.classList.contains(className));
```

```
function replaceClasses(element, oldClasses, newClasses) {
  oldClasses.forEach((cls) => element.classList.remove(cls));
  newClasses.forEach((cls) => element.classList.add(cls));
// State management with classes
class ElementState {
 constructor(element) {
   this.element = element;
   this.states = new Set();
  }
  setState(state, active = true) {
    if (active) {
      this.states.add(state);
      this.element.classList.add(`state-${state}`);
    } else {
     this.states.delete(state);
      this.element.classList.remove(`state-${state}`);
    }
  }
  hasState(state) {
   return this.states.has(state);
  }
  clearStates() {
    this.states.forEach((state) => {
     this.element.classList.remove(`state-${state}`);
    });
   this.states.clear();
 }
}
// Usage
const buttonState = new ElementState(document.querySelector("#my-button"));
buttonState.setState("loading", true);
buttonState.setState("disabled", true);
// Practical examples
function showElement(element) {
  element.classList.remove("hidden", "fade-out");
  element.classList.add("visible", "fade-in");
}
function hideElement(element) {
  element.classList.remove("visible", "fade-in");
  element.classList.add("hidden", "fade-out");
}
function setLoadingState(button, isLoading) {
  button.classList.toggle("loading", isLoading);
  button.disabled = isLoading;
```

```
const text = button.querySelector(".text");
const spinner = button.querySelector(".spinner");

if (isLoading) {
    text.style.display = "none";
    spinner.style.display = "inline-block";
} else {
    text.style.display = "inline-block";
    spinner.style.display = "none";
}
}
```

## Inline Styles

```
const element = document.querySelector(".target");
// Setting individual styles
element.style.color = "red";
element.style.backgroundColor = "blue";
element.style.fontSize = "16px";
element.style.marginTop = "10px";
// CSS property names (camelCase)
element.style.borderRadius = "5px";
element.style.textAlign = "center";
element.style.zIndex = "1000";
// Setting multiple styles
function setStyles(element, styles) {
  Object.assign(element.style, styles);
}
// Usage
setStyles(element, {
  color: "white",
  backgroundColor: "navy",
  padding: "10px",
  borderRadius: "5px",
});
// CSS custom properties (CSS variables)
element.style.setProperty("--primary-color", "#007bff");
element.style.setProperty("--border-width", "2px");
// Getting computed styles
const computedStyle = window.getComputedStyle(element);
const color = computedStyle.getPropertyValue("color");
const fontSize = computedStyle.fontSize;
// Style utilities
```

```
function getNumericStyle(element, property) {
  const value = window.getComputedStyle(element).getPropertyValue(property);
 return parseFloat(value) | 0;
}
function setOpacity(element, opacity) {
  element.style.opacity = Math.max(0, Math.min(1, opacity));
}
function fadeIn(element, duration = 300) {
  element.style.opacity = "0";
  element.style.display = "block";
  const start = performance.now();
  function animate(currentTime) {
    const elapsed = currentTime - start;
    const progress = Math.min(elapsed / duration, 1);
    element.style.opacity = progress;
    if (progress < 1) {</pre>
      requestAnimationFrame(animate);
    }
  }
  requestAnimationFrame(animate);
}
function fadeOut(element, duration = 300) {
  const start = performance.now();
  const startOpacity = parseFloat(element.style.opacity) || 1;
  function animate(currentTime) {
    const elapsed = currentTime - start;
    const progress = Math.min(elapsed / duration, 1);
    element.style.opacity = startOpacity * (1 - progress);
    if (progress >= 1) {
      element.style.display = "none";
    } else {
      requestAnimationFrame(animate);
    }
  }
  requestAnimationFrame(animate);
}
// Responsive style adjustments
function adjustForScreenSize(element) {
  const width = window.innerWidth;
 if (width < 768) {
```

```
setStyles(element, {
      fontSize: "14px",
      padding: "5px",
    });
  } else if (width < 1024) {</pre>
    setStyles(element, {
      fontSize: "16px",
      padding: "10px",
    });
  } else {
    setStyles(element, {
      fontSize: "18px",
      padding: "15px",
    });
  }
}
// Theme switching
function applyTheme(themeName) {
  const themes = {
    light: {
      "--bg-color": "#ffffff",
      "--text-color": "#333333",
      "--border-color": "#cccccc",
    },
    dark: {
      "--bg-color": "#333333",
      "--text-color": "#ffffff",
      "--border-color": "#555555",
   },
  };
  const theme = themes[themeName];
 if (theme) {
    Object.entries(theme).forEach(([property, value]) => {
      document.documentElement.style.setProperty(property, value);
    });
  }
}
```

# Table Creating and Modifying Elements

## **Creating Elements**

```
// Basic element creation
const div = document.createElement("div");
const paragraph = document.createElement("p");
const button = document.createElement("button");
const image = document.createElement("img");

// Setting attributes and content
```

```
div.className = "container";
div.id = "main-container";
paragraph.textContent = "Hello, World!";
button.textContent = "Click me";
button.type = "button";
image.src = "image.jpg";
image.alt = "Description";
// Creating complex elements
function createElement(tag, attributes = {}, children = []) {
  const element = document.createElement(tag);
  // Set attributes
  Object.entries(attributes).forEach(([key, value]) => {
    if (key === "className") {
      element.className = value;
    } else if (key === "textContent") {
      element.textContent = value;
    } else if (key === "innerHTML") {
      element.innerHTML = value;
    } else {
      element.setAttribute(key, value);
    }
 });
  // Add children
  children.forEach((child) => {
    if (typeof child === "string") {
      element.appendChild(document.createTextNode(child));
    } else {
      element.appendChild(child);
    }
  });
 return element;
}
// Usage
const card = createElement(
  "div",
    className: "card",
    "data-id": "123",
  },
    createElement("h3", { textContent: "Card Title" }),
    createElement("p", { textContent: "Card description" }),
    createElement("button", {
      className: "btn btn-primary",
      textContent: "Action",
    }),
  ]
);
```

```
// Factory functions for common elements
function createButton(text, className = "btn", onClick = null) {
  const button = createElement("button", {
    className,
    textContent: text,
   type: "button",
  });
  if (onClick) {
    button.addEventListener("click", onClick);
  }
  return button;
}
function createInput(type, name, placeholder = "") {
  return createElement("input", {
    type,
    name,
    placeholder,
    className: "form-control",
  });
}
function createSelect(name, options = []) {
  const select = createElement("select", {
    className: "form-control",
  });
  options.forEach((option) => {
    const optionElement = createElement("option", {
      value: option.value,
      textContent: option.text,
    });
    select.appendChild(optionElement);
  });
 return select;
}
// Creating from HTML strings
function createFromHTML(htmlString) {
  const template = document.createElement("template");
  template.innerHTML = htmlString.trim();
  return template.content.firstChild;
}
// Usage
const element = createFromHTML()
    <div class="alert alert-success">
        <strong>Success!</strong> Operation completed.
    </div>
`);
```

```
// Document fragments for efficient DOM manipulation
function createMultipleElements(count, elementFactory) {
  const fragment = document.createDocumentFragment();
 for (let i = 0; i < count; i++) {
   fragment.appendChild(elementFactory(i));
  }
 return fragment;
}
// Usage
const listItems = createMultipleElements(10, (index) => {
 return createElement("li", {
   textContent: `Item ${index + 1}`,
    "data-index": index,
 });
});
document.querySelector("#list").appendChild(listItems);
```

## Adding and Removing Elements

```
// Adding elements
const container = document.querySelector("#container");
const newElement = document.createElement("div");
// Append to end
container.appendChild(newElement);
// Insert at beginning
container.insertBefore(newElement, container.firstChild);
// Insert at specific position
const referenceElement = container.children[2];
container.insertBefore(newElement, referenceElement);
// Modern insertion methods
const targetElement = document.querySelector(".target");
// Insert adjacent to element
targetElement.insertAdjacentElement("beforebegin", newElement); // Before target
targetElement.insertAdjacentElement("afterbegin", newElement); // First child of
targetElement.insertAdjacentElement("beforeend", newElement); // Last child of
target
targetElement.insertAdjacentElement("afterend", newElement); // After target
// Insert HTML strings
targetElement.insertAdjacentHTML("beforebegin", "<div>Before</div>");
```

```
targetElement.insertAdjacentHTML("afterbegin", "<div>First child</div>");
targetElement.insertAdjacentHTML("beforeend", "<div>Last child</div>");
targetElement.insertAdjacentHTML("afterend", "<div>After</div>");
// Removing elements
const elementToRemove = document.querySelector(".remove-me");
// Modern way
elementToRemove.remove();
// Traditional way
elementToRemove.parentNode.removeChild(elementToRemove);
// Remove all children
function clearElement(element) {
  while (element.firstChild) {
    element.removeChild(element.firstChild);
 }
 // Or simply:
  // element.innerHTML = '';
}
// Conditional removal
function removeIf(selector, condition) {
  const elements = document.querySelectorAll(selector);
  elements.forEach((element) => {
    if (condition(element)) {
      element.remove();
    }
 });
}
// Usage: Remove all empty paragraphs
removeIf("p", (p) => p.textContent.trim() === "");
// Replace elements
function replaceElement(oldElement, newElement) {
  oldElement.parentNode.replaceChild(newElement, oldElement);
}
// Clone elements
const original = document.querySelector(".original");
const clone = original.cloneNode(true); // true = deep clone (includes children)
const shallowClone = original.cloneNode(false); // false = shallow clone
// Move elements
function moveElement(element, newParent) {
  newParent.appendChild(element); // Automatically removes from old parent
}
// Batch operations with DocumentFragment
function batchInsert(container, elements) {
  const fragment = document.createDocumentFragment();
  elements.forEach((element) => fragment.appendChild(element));
```

```
container.appendChild(fragment); // Single DOM operation
}
// List management utilities
class ListManager {
 constructor(containerSelector) {
   this.container = document.querySelector(containerSelector);
   this.items = [];
 }
 addItem(data, template) {
   const element = template(data);
   this.container.appendChild(element);
   this.items.push({ data, element });
   return element;
 removeItem(index) {
   if (this.items[index]) {
     this.items[index].element.remove();
     this.items.splice(index, 1);
   }
  }
 updateItem(index, newData, template) {
   if (this.items[index]) {
      const newElement = template(newData);
     this.items[index].element.replaceWith(newElement);
     this.items[index] = { data: newData, element: newElement };
   }
  }
 clear() {
   this.container.innerHTML = "";
   this.items = [];
 }
 getItems() {
   return this.items.map((item) => item.data);
 }
}
// Usage
const todoList = new ListManager("#todo-list");
todoList.addItem({ id: 1, text: "Buy groceries", completed: false }, (data) =>
 createElement("li", {
   textContent: data.text,
   className: data.completed ? "completed" : "",
 })
);
```

## Attributes and Properties

### Working with Attributes

```
const element = document.querySelector(".target");
// Getting attributes
const id = element.getAttribute("id");
const className = element.getAttribute("class");
const dataValue = element.getAttribute("data-value");
// Setting attributes
element.setAttribute("id", "new-id");
element.setAttribute("class", "new-class");
element.setAttribute("data-value", "123");
element.setAttribute("aria-label", "Close button");
// Removing attributes
element.removeAttribute("data-old");
element.removeAttribute("disabled");
// Checking if attribute exists
if (element.hasAttribute("data-value")) {
 console.log("Element has data-value attribute");
}
// Getting all attributes
const attributes = Array.from(element.attributes);
attributes.forEach((attr) => {
 console.log(`${attr.name}: ${attr.value}`);
});
// Data attributes (modern approach)
const element = document.querySelector('[data-user-id="123"]');
// Access via dataset property
console.log(element.dataset.userId); // "123"
console.log(element.dataset.userName); // Gets data-user-name
// Setting data attributes
element.dataset.status = "active";
element.dataset.lastModified = new Date().toISOString();
// Data attribute utilities
function setDataAttributes(element, data) {
 Object.entries(data).forEach(([key, value]) => {
    element.dataset[key] = value;
 });
function getDataAttributes(element) {
 return { ...element.dataset };
```

```
// Usage
setDataAttributes(element, {
  userId: "456",
 role: "admin",
 permissions: "read,write,delete",
});
const data = getDataAttributes(element);
console.log(data); // { userId: '456', role: 'admin', permissions:
'read,write,delete' }
// Boolean attributes
function setBooleanAttribute(element, attribute, value) {
  if (value) {
   element.setAttribute(attribute, "");
  } else {
    element.removeAttribute(attribute);
}
// Usage
setBooleanAttribute(button, "disabled", true);
setBooleanAttribute(input, "required", false);
setBooleanAttribute(details, "open", true);
```

#### **Properties vs Attributes**

```
const input = document.querySelector('input[type="text"]');
// Properties (JavaScript object properties)
input.value = "Hello"; // Current value
input.disabled = true; // Current state
input.checked = false; // For checkboxes/radios
// Attributes (HTML attributes)
input.setAttribute("value", "Hello"); // Default value
input.setAttribute("disabled", ""); // Presence indicates disabled
input.setAttribute("checked", ""); // Default checked state
// Key differences:
// 1. Properties reflect current state, attributes reflect initial state
// 2. Properties are typed (boolean, string, number), attributes are always
strings
// 3. Some properties don't have corresponding attributes
// Form element properties
const form = document.querySelector("form");
const select = document.querySelector("select");
const checkbox = document.querySelector('input[type="checkbox"]');
```

```
// Input properties
console.log(input.value); // Current value
console.log(input.defaultValue); // Original value attribute
console.log(input.validity); // Validation state
console.log(input.files); // For file inputs
// Select properties
console.log(select.selectedIndex); // Index of selected option
console.log(select.selectedOptions); // Selected option elements
console.log(select.options); // All option elements
// Checkbox/radio properties
console.log(checkbox.checked); // Current checked state
console.log(checkbox.defaultChecked); // Original checked attribute
// Form properties
console.log(form.elements); // All form controls
console.log(form.length); // Number of form controls
// Property utilities
function getFormData(form) {
 const data = {};
 const formData = new FormData(form);
 for (const [key, value] of formData.entries()) {
   if (data[key]) {
     // Handle multiple values (checkboxes, multi-select)
      if (Array.isArray(data[key])) {
       data[key].push(value);
      } else {
        data[key] = [data[key], value];
      }
    } else {
      data[key] = value;
   }
  }
 return data;
}
function setFormData(form, data) {
 Object.entries(data).forEach(([name, value]) => {
    const element = form.elements[name];
   if (element) {
      if (element.type === "checkbox" || element.type === "radio") {
        element.checked = value;
      } else {
        element.value = value;
      }
    }
 });
```

```
// Element state management
class ElementState {
 constructor(element) {
    this.element = element;
    this.originalAttributes = new Map();
    this.originalProperties = new Map();
  saveState() {
    // Save current attributes
    Array.from(this.element.attributes).forEach((attr) => {
      this.originalAttributes.set(attr.name, attr.value);
    });
    // Save important properties
    const props = ["value", "checked", "selected", "disabled"];
    props.forEach((prop) => {
      if (prop in this.element) {
        this.originalProperties.set(prop, this.element[prop]);
    });
  restoreState() {
    // Restore attributes
    this.originalAttributes.forEach((value, name) => {
      this.element.setAttribute(name, value);
    });
    // Restore properties
    this.originalProperties.forEach((value, prop) => {
      this.element[prop] = value;
    });
  }
}
// Usage
const elementState = new ElementState(input);
elementState.saveState();
// Make changes...
input.value = "Modified";
input.disabled = true;
// Restore original state
elementState.restoreState();
```

## 

#### 1. NodeList vs HTMLCollection

```
// X Assuming live collections are static
const divs = document.getElementsByTagName("div"); // HTMLCollection (live)
console.log(divs.length); // 5
// Adding a new div
document.body.appendChild(document.createElement("div"));
console.log(divs.length); // 6 (automatically updated!)
// This can cause infinite loops:
for (let i = 0; i < divs.length; i++) {
 document.body.appendChild(document.createElement("div")); // Infinite loop!
}
// Convert to static array when needed
const staticDivs = Array.from(document.getElementsByTagName("div"));
for (let i = 0; i < staticDivs.length; i++) {
  document.body.appendChild(document.createElement("div")); // Safe
}
// < Or use querySelectorAll (returns static NodeList)</pre>
const divs = document.querySelectorAll("div");
```

### 2. innerHTML Security Issues

```
// X Dangerous - XSS vulnerability
const userInput = '<img src="x" onerror="alert(\'XSS\')">';
element.innerHTML = userInput; // Executes malicious script!
// Safe alternatives
// Use textContent for plain text
element.textContent = userInput; // Safe - no HTML execution
// Use createElement for dynamic content
const img = document.createElement("img");
img.src = userSrc; // Validate userSrc first
img.alt = userAlt; // Validate userAlt first
element.appendChild(img);
// Sanitize HTML if you must use innerHTML
function sanitizeHTML(html) {
  const temp = document.createElement("div");
 temp.textContent = html;
 return temp.innerHTML;
}
element.innerHTML = sanitizeHTML(userInput);
```

### 3. Event Delegation Issues

```
// X Adding event listeners to many elements
const buttons = document.guerySelectorAll(".btn");
buttons.forEach((button) => {
  button.addEventListener("click", handleClick); // Memory intensive
});
// ✓ Use event delegation
document.addEventListener("click", (event) => {
  if (event.target.matches(".btn")) {
    handleClick(event);
  }
});
// X Not checking if element exists
document.guerySelector(".non-existent").addEventListener("click", handler);
// TypeError: Cannot read property 'addEventListener' of null
// Always check existence
const element = document.querySelector(".maybe-exists");
if (element) {
 element.addEventListener("click", handler);
}
// Or use optional chaining (modern browsers)
document.querySelector(".maybe-exists")?.addEventListener("click", handler);
```

#### 4. Style Manipulation Issues

```
// X Setting styles that cause layout thrashing
function animateElements(elements) {
  elements.forEach((element, index) => {
    element.style.left = `${index * 100}px`; // Causes reflow for each element
 });
}
// ✓ Batch DOM operations
function animateElementsEfficiently(elements) {
 // Use DocumentFragment or CSS transforms
 elements.forEach((element, index) => {
    element.style.transform = `translateX(${index * 100}px)`; // No reflow
 });
}
// X Reading and writing styles in a loop
for (let i = 0; i < elements.length; i++) {
 const height = elements[i].offsetHeight; // Read (causes reflow)
 elements[i].style.height = `${height + 10}px`; // Write (causes reflow)
}
// ✓ Separate reads and writes
```

```
const heights = elements.map((el) => el.offsetHeight); // Batch reads
elements.forEach((el, i) => {
   el.style.height = `${heights[i] + 10}px`; // Batch writes
});
```

## 5. Memory Leaks

```
// X Not removing event listeners
function setupComponent() {
  const button = document.createElement("button");
  const handler = () => console.log("clicked");
 button.addEventListener("click", handler);
 document.body.appendChild(button);
 // Later, removing the button but not the listener
 button.remove(); // Memory leak - handler still referenced
}
// ✓ Proper cleanup
function setupComponentProperly() {
  const button = document.createElement("button");
  const handler = () => console.log("clicked");
 button.addEventListener("click", handler);
 document.body.appendChild(button);
 // Cleanup function
 return () => {
    button.removeEventListener("click", handler);
    button.remove();
 };
}
const cleanup = setupComponentProperly();
// Later...
cleanup(); // Proper cleanup
// X Circular references
function createCircularReference() {
  const element = document.createElement("div");
 element.customProperty = {
   element: element, // Circular reference
   data: "some data",
 };
 return element;
}
// ✓ Avoid circular references
function createProperReference() {
  const element = document.createElement("div");
```

```
const data = {
    elementId: element.id || `element-${Date.now()}`,
    data: "some data",
};
element.dataset.objectId = data.elementId;
return { element, data };
}
```

# Mini Practice Problems

## Problem 1: Dynamic Table Generator

```
// Create a function that generates a sortable, filterable table
function createDataTable(container, data, options = {}) {
 // Your implementation should:
 // - Generate table with headers and data rows
 // - Add sorting functionality (click headers to sort)
 // - Add filtering (search input)
 // - Support pagination
 // - Handle empty data gracefully
 // - Allow custom cell renderers
 // Options should support:
 // {
 // sortable: true,
 // filterable: true,
 // pageSize: 10,
 //
     columns: [
     { key: 'name', title: 'Name', sortable: true },
 //
       { key: 'age', title: 'Age', sortable: true, type: 'number' },
 //
        { key: 'email', title: 'Email', sortable: false },
 //
        { key: 'actions', title: 'Actions', render: (row) => `<button
onclick="edit(${row.id})">Edit</button>` }
 // ]
 // }
}
// Test data
const users = [
 { id: 1, name: "Alice", age: 30, email: "alice@example.com" },
 { id: 2, name: "Bob", age: 25, email: "bob@example.com" },
  { id: 3, name: "Charlie", age: 35, email: "charlie@example.com" },
];
createDataTable("#table-container", users, {
  sortable: true,
 filterable: true,
 pageSize: 5,
 columns: [
   { key: "name", title: "Name" },
   { key: "age", title: "Age", type: "number" },
    { key: "email", title: "Email" },
```

```
{
    key: "actions",
    title: "Actions",
    render: (row) => `<button>Edit</button>`,
    },
    ],
});
```

## Problem 2: Modal Dialog System

```
// Create a reusable modal dialog system
class ModalManager {
 constructor() {
   // Your implementation
   // Should handle:
    // - Multiple modals
   // - Modal stacking (z-index management)
   // - Backdrop clicks
   // - Escape key handling
   // - Focus management
   // - Animations
  }
 create(options) {
   // Create and return a modal instance
   // Options: { title, content, size, closable, backdrop }
 open(modal) {
   // Open a modal
 close(modal) {
   // Close a modal
  }
 closeAll() {
   // Close all open modals
 }
 confirm(message, options = {}) {
   // Show confirmation dialog, return promise
  }
 alert(message, options = {}) {
   // Show alert dialog, return promise
  }
 prompt(message, defaultValue = "", options = {}) {
   // Show prompt dialog, return promise with user input
```

```
// Usage examples:
const modals = new ModalManager();
// Basic modal
const modal = modals.create({
  title: "User Profile",
  content: "<form>...</form>",
  size: "large",
});
modals.open(modal);
// Confirmation
modals
  .confirm("Are you sure you want to delete this item?")
  .then((confirmed) => {
    if (confirmed) {
      console.log("Item deleted");
  });
// Prompt
modals.prompt("Enter your name:", "John Doe").then((name) => {
  if (name) {
   console.log("Hello,", name);
  }
});
```

#### Problem 3: Drag and Drop List

```
// Create a drag-and-drop sortable list
class SortableList {
  constructor(container, options = {}) {
   // Your implementation should:
   // - Make list items draggable
   // - Show visual feedback during drag
    // - Handle drop zones
   // - Animate reordering
   // - Support touch devices
   // - Emit events for reorder
   // - Support disabled items
   // - Handle nested lists (optional)
    // Options: {
    // itemSelector: '.list-item',
       handleSelector: '.drag-handle', // Optional drag handle
    //
    // placeholder: 'drop-placeholder',
    // animation: 300,
       disabled: false,
   //
       onSort: (oldIndex, newIndex) => {}
    // }
```

```
enable() {
  // Enable drag and drop
 disable() {
  // Disable drag and drop
 }
 destroy() {
   // Clean up event listeners
 }
 getOrder() {
   // Return current order of items
 }
 setOrder(order) {
   // Set order of items
 }
}
// HTML structure:
// 
// 
//
    <span class="drag-handle">::</span>
     <span>Item 1</span>
//
// 
// 
     <span class="drag-handle">::</span>
//
     <span>Item 2</span>
// 
// 
// Usage:
const sortable = new SortableList("#sortable-list", {
 handleSelector: ".drag-handle",
 onSort: (oldIndex, newIndex) => {
   console.log(`Moved item from ${oldIndex} to ${newIndex}`);
 },
});
```

#### Problem 4: Virtual Scrolling List

```
// Create a virtual scrolling list for large datasets
class VirtualList {
  constructor(container, options = {}) {
    // Your implementation should:
    // - Only render visible items
    // - Handle scrolling efficiently
```

```
// - Support variable item heights
    // - Maintain scroll position
    // - Support dynamic data updates
    // - Handle resize events
    // Options: {
    // itemHeight: 50, // Fixed height or function
    // renderItem: (item, index) => string, // Item renderer
    // overscan: 5, // Extra items to render outside viewport
       data: [], // Initial data
   // estimatedItemHeight: 50 // For variable heights
   // }
  }
  setData(data) {
  // Update the data and re-render
  scrollToIndex(index) {
  // Scroll to specific item
  }
  scrollToTop() {
   // Scroll to top
  }
 refresh() {
   // Force re-render
  }
  destroy() {
   // Clean up
  }
}
// Usage:
const virtualList = new VirtualList("#list-container", {
  itemHeight: 60,
  renderItem: (item, index) => `
        <div class="list-item">
            <h4>${item.title}</h4>
            ${item.description}
        </div>
  overscan: 10,
});
// Large dataset
const largeData = Array.from({ length: 100000 }, (_, i) => ({
  id: i,
 title: `Item ${i}`,
 description: `Description for item ${i}`,
}));
virtualList.setData(largeData);
```

#### Problem 5: Form Validation System

```
// Create a comprehensive form validation system
class FormValidator {
  constructor(form, rules = {}) {
    // Your implementation should:
   // - Validate on input, blur, and submit
    // - Show/hide error messages
   // - Support custom validation rules
   // - Handle different input types
   // - Support async validation
    // - Prevent form submission if invalid
   // - Highlight invalid fields
   // Rules format:
   // {
       fieldName: [
         { rule: 'required', message: 'This field is required' },
   //
           { rule: 'email', message: 'Invalid email format' },
    //
    //
           { rule: 'minLength', value: 8, message: 'Minimum 8 characters' },
          { rule: 'custom', validate: (value) => boolean, message: 'Custom error'
    //
},
          { rule: 'async', validate: async (value) => boolean, message: 'Async
   //
error' }
  // 1
   // }
 addRule(fieldName, rule) {
   // Add validation rule to field
  }
 removeRule(fieldName, ruleType) {
   // Remove validation rule from field
  }
 validate(fieldName = null) {
   // Validate specific field or entire form
   // Return { isValid: boolean, errors: {} }
  }
  showError(fieldName, message) {
   // Show error message for field
  }
 hideError(fieldName) {
   // Hide error message for field
  }
  reset() {
   // Reset form and clear all errors
```

```
destroy() {
   // Clean up event listeners
}
// Built-in validation rules
const validationRules = {
  required: (value) => value.trim() !== "",
  email: (value) \Rightarrow /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value),
  minLength: (value, length) => value.length >= length,
  maxLength: (value, length) => value.length <= length,</pre>
  pattern: (value, regex) => regex.test(value),
  number: (value) => !isNaN(value) && !isNaN(parseFloat(value)),
  url: (value) => {
   try {
      new URL(value);
      return true;
    } catch {
      return false;
    }
 },
};
// Usage:
const validator = new FormValidator("#user-form", {
  username: [
    { rule: "required", message: "Username is required" },
      rule: "minLength",
      value: 3,
      message: "Username must be at least 3 characters",
    },
  ],
  email: [
    { rule: "required", message: "Email is required" },
    { rule: "email", message: "Please enter a valid email" },
    {
      rule: "async",
      validate: async (email) => {
        const response = await fetch(`/api/check-email?email=${email}`);
        const result = await response.json();
        return !result.exists;
      },
      message: "Email already exists",
    },
  ],
  password: [
    { rule: "required", message: "Password is required" },
    {
      rule: "minLength",
      value: 8,
      message: "Password must be at least 8 characters",
```

```
},
{
    rule: "custom",
    validate: (value) => /(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/.test(value),
    message: "Password must contain uppercase, lowercase, and number",
    },
],
});
```

### Interview Notes

#### **Common Questions:**

#### Q: What's the difference between innerHTML, textContent, and innerText?

- innerHTML: Gets/sets HTML content, can execute scripts (security risk)
- textContent: Gets/sets text content, ignores HTML tags, includes hidden text
- innerText: Gets/sets visible text content, respects styling (hidden elements ignored)

#### Q: How do you efficiently add many elements to the DOM?

- Use DocumentFragment to batch operations
- Use insertAdjacentHTML for HTML strings
- Avoid adding elements one by one in a loop

#### Q: What's the difference between querySelector and getElementById?

- getElementById: Faster, returns single element by ID
- querySelector: More flexible (CSS selectors), returns first match
- querySelectorAll: Returns all matches as static NodeList

#### Q: How do you prevent XSS when inserting user content?

- Use textContent instead of innerHTML for plain text
- Sanitize HTML content before insertion
- Use createElement and set properties instead of HTML strings
- Validate and escape user input

#### Q: What's event delegation and when should you use it?

- Attaching event listener to parent element instead of individual children
- Use when you have many similar elements or dynamic content
- More memory efficient, handles dynamically added elements

### Asked at Companies:

- Google: "Implement a virtual scrolling component for large lists"
- Facebook: "Create a drag-and-drop interface with React-like virtual DOM"
- Amazon: "Build a form validation system with real-time feedback"
- Microsoft: "Design a modal system that handles focus management and accessibility"
- Netflix: "Create a responsive grid layout that adapts to different screen sizes"

# **©** Key Takeaways

- 1. Use modern selectors querySelector and querySelectorAll are more flexible
- 2. Batch DOM operations Use DocumentFragment for multiple insertions
- 3. Prefer textContent over innerHTML Safer and often faster
- 4. Use event delegation More efficient for many elements
- 5. Check element existence Always verify elements exist before manipulation
- 6. Separate reads and writes Avoid layout thrashing
- 7. Clean up event listeners Prevent memory leaks
- 8. Use CSS classes over inline styles Better performance and maintainability

Previous Chapter: ← ES6+ Features
Next Chapter: Event Handling →

**Practice**: Try the DOM manipulation problems and experiment with different selection and modification

techniques!

## 15. Browser APIs

### **Table of Contents**

- Local Storage & Session Storage
- Fetch API
- Geolocation API
- File API
- Canvas API
- Web Workers
- Service Workers
- Intersection Observer
- Mutation Observer
- History API
- Notification API
- Common Pitfalls
- Mini Practice Problems
- Interview Notes

# **@** Learning Objectives

By the end of this chapter, you will:

- Master browser storage APIs (localStorage, sessionStorage)
- Use the Fetch API for HTTP requests
- Work with geolocation and file handling
- Understand Canvas for graphics and animations
- Implement Web Workers for background processing
- Use modern observer APIs for efficient DOM monitoring

- Handle browser history and navigation
- Implement push notifications



### Local Storage & Session Storage

#### Basic Usage

```
// localStorage - persists until manually cleared
localStorage.setItem("username", "john_doe");
localStorage.setItem(
  "preferences",
  JSON.stringify({
   theme: "dark",
    language: "en",
 })
);
// Get items
const username = localStorage.getItem("username");
const preferences = JSON.parse(localStorage.getItem("preferences") || "{}");
// Remove items
localStorage.removeItem("username");
localStorage.clear(); // Remove all items
// sessionStorage - persists only for the session
sessionStorage.setItem("tempData", "session-specific");
const tempData = sessionStorage.getItem("tempData");
```

#### Storage Events

```
// Listen for storage changes (only fires in other tabs/windows)
window.addEventListener("storage", (e) => {
  console.log("Storage changed:", {
    key: e.key,
    oldValue: e.oldValue,
    newValue: e.newValue,
    url: e.url,
  });
  // Sync UI with storage changes
  if (e.key === "theme") {
    updateTheme(e.newValue);
  }
});
```

#### Storage Wrapper Class

```
class StorageManager {
 constructor(storage = localStorage) {
   this.storage = storage;
 }
 set(key, value, expiry = null) {
   const item = {
     value,
     timestamp: Date.now(),
     expiry: expiry ? Date.now() + expiry : null,
   };
   try {
     this.storage.setItem(key, JSON.stringify(item));
     return true;
   } catch (error) {
     console.error("Storage error:", error);
     return false;
   }
 }
 get(key) {
   try {
     const item = JSON.parse(this.storage.getItem(key));
     if (!item) return null;
     // Check expiry
     if (item.expiry && Date.now() > item.expiry) {
      this.remove(key);
       return null;
     }
     return item.value;
   } catch (error) {
     console.error("Storage parse error:", error);
     return null;
   }
 }
 remove(key) {
   this.storage.removeItem(key);
 }
 clear() {
   this.storage.clear();
 }
 keys() {
   return Object.keys(this.storage);
 }
```

```
size() {
    return this.storage.length;
  }
  // Get storage usage in bytes (approximate)
  getUsage() {
    let total = 0;
    for (let key in this.storage) {
      if (this.storage.hasOwnProperty(key)) {
        total += this.storage[key].length + key.length;
      }
   return total;
 }
}
// Usage
const storage = new StorageManager();
storage.set("user", { name: "John", age: 30 }, 24 * 60 * 60 * 1000); // 24 hours
const user = storage.get("user");
```

# Fetch API

#### **Basic Requests**

```
// GET request
fetch("/api/users")
  .then((response) => {
    if (!response.ok) {
     throw new Error(`HTTP error! status: ${response.status}`);
    }
    return response.json();
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
// POST request
fetch("/api/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
   Authorization: "Bearer " + token,
  },
  body: JSON.stringify({
    name: "John Doe",
    email: "john@example.com",
  }),
})
  .then((response) => response.json())
```

```
.then((data) => console.log("Success:", data))
.catch((error) => console.error("Error:", error));
```

#### Advanced Fetch Patterns

```
// Fetch with timeout
function fetchWithTimeout(url, options = {}, timeout = 5000) {
 return Promise.race([
   fetch(url, options),
   new Promise((_, reject) =>
     setTimeout(() => reject(new Error("Request timeout")), timeout)
 ]);
}
// Retry mechanism
async function fetchWithRetry(url, options = {}, maxRetries = 3) {
 for (let i = 0; i <= maxRetries; i++) {
   try {
      const response = await fetch(url, options);
      if (response.ok) return response;
      if (i === maxRetries)
       throw new Error(`Failed after ${maxRetries} retries`);
      // Exponential backoff
      await new Promise((resolve) =>
        setTimeout(resolve, Math.pow(2, i) * 1000)
      );
    } catch (error) {
      if (i === maxRetries) throw error;
   }
 }
}
// Upload with progress
function uploadWithProgress(file, url, onProgress) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    const formData = new FormData();
   formData.append("file", file);
    xhr.upload.addEventListener("progress", (e) => {
      if (e.lengthComputable) {
        const progress = (e.loaded / e.total) * 100;
        onProgress(progress);
      }
    });
    xhr.addEventListener("load", () => {
      if (xhr.status === 200) {
```

```
resolve(JSON.parse(xhr.responseText));
} else {
    reject(new Error(`Upload failed: ${xhr.status}`));
}
});

xhr.addEventListener("error", () => reject(new Error("Upload failed")));

xhr.open("POST", url);
    xhr.send(formData);
});
}
```

#### **HTTP Client Class**

```
class HTTPClient {
 constructor(baseURL = "", defaultHeaders = {}) {
   this.baseURL = baseURL;
   this.defaultHeaders = {
      "Content-Type": "application/json",
      ...defaultHeaders,
   };
   this.interceptors = {
     request: [],
     response: [],
   };
 }
 addRequestInterceptor(interceptor) {
   this.interceptors.request.push(interceptor);
 }
 addResponseInterceptor(interceptor) {
   this.interceptors.response.push(interceptor);
 }
 async request(url, options = {}) {
   // Apply request interceptors
   let config = {
     ...options,
     headers: { ...this.defaultHeaders, ...options.headers },
   };
   for (const interceptor of this.interceptors.request) {
     config = await interceptor(config);
   }
   try {
     let response = await fetch(this.baseURL + url, config);
     // Apply response interceptors
```

```
for (const interceptor of this.interceptors.response) {
        response = await interceptor(response);
      }
      return response;
    } catch (error) {
      throw error;
    }
  }
  get(url, options = {}) {
    return this.request(url, { ...options, method: "GET" });
  }
  post(url, data, options = {}) {
    return this.request(url, {
      ...options,
      method: "POST",
      body: JSON.stringify(data),
    });
  }
  put(url, data, options = {}) {
    return this.request(url, {
      ...options,
      method: "PUT",
      body: JSON.stringify(data),
   });
  }
 delete(url, options = {}) {
    return this.request(url, { ...options, method: "DELETE" });
 }
}
// Usage
const api = new HTTPClient("/api/v1");
// Add auth interceptor
api.addRequestInterceptor(async (config) => {
  const token = localStorage.getItem("authToken");
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
// Add error handling interceptor
api.addResponseInterceptor(async (response) => {
 if (response.status === 401) {
   // Redirect to login
    window.location.href = "/login";
  }
  return response;
```

```
});

// Make requests
api.get("/users").then((response) => response.json());
api.post("/users", { name: "John", email: "john@example.com" });
```

## Geolocation API

#### **Basic Geolocation**

```
// Check if geolocation is supported
if ("geolocation" in navigator) {
 // Get current position
 navigator.geolocation.getCurrentPosition(
    (position) => {
      const { latitude, longitude, accuracy } = position.coords;
      console.log(
       `Lat: ${latitude}, Lng: ${longitude}, Accuracy: ${accuracy}m`
      );
      // Use the coordinates
      showLocationOnMap(latitude, longitude);
    },
    (error) => {
      console.error("Geolocation error:", error.message);
      handleLocationError(error);
    },
      enableHighAccuracy: true,
      timeout: 10000,
      maximumAge: 60000, // Cache for 1 minute
   }
  );
} else {
  console.log("Geolocation not supported");
}
// Watch position changes
const watchId = navigator.geolocation.watchPosition(
  (position) => {
    updateLocationOnMap(position.coords.latitude, position.coords.longitude);
  (error) => {
   console.error("Watch position error:", error);
  { enableHighAccuracy: true }
);
// Stop watching
// navigator.geolocation.clearWatch(watchId);
```

#### **Geolocation Wrapper**

```
class GeolocationService {
  constructor() {
   this.watchId = null;
   this.isSupported = "geolocation" in navigator;
 async getCurrentPosition(options = {}) {
    if (!this.isSupported) {
      throw new Error("Geolocation not supported");
    }
    const defaultOptions = {
      enableHighAccuracy: true,
      timeout: 10000,
      maximumAge: 60000,
    };
    return new Promise((resolve, reject) => {
      navigator.geolocation.getCurrentPosition(resolve, reject, {
        ...defaultOptions,
        ...options,
      });
   });
 watchPosition(callback, errorCallback, options = {}) {
    if (!this.isSupported) {
      throw new Error("Geolocation not supported");
    }
    this.watchId = navigator.geolocation.watchPosition(
      callback,
      errorCallback,
      options
    );
   return this.watchId;
  }
 stopWatching() {
   if (this.watchId !== null) {
      navigator.geolocation.clearWatch(this.watchId);
      this.watchId = null;
   }
  }
  async getAddressFromCoords(lat, lng) {
   try {
      const response = await fetch(
```

```
`https://api.opencagedata.com/geocode/v1/json?
q=${lat}+${lng}&key=YOUR_API_KEY`
      );
      const data = await response.json();
      return data.results[0]?.formatted || "Address not found";
    } catch (error) {
      console.error("Reverse geocoding error:", error);
      return null;
    }
  }
  calculateDistance(lat1, lng1, lat2, lng2) {
    const R = 6371; // Earth's radius in kilometers
    const dLat = this.toRadians(lat2 - lat1);
    const dLng = this.toRadians(lng2 - lng1);
    const a =
      Math.sin(dLat / 2) * Math.sin(dLat / 2) +
      Math.cos(this.toRadians(lat1)) *
        Math.cos(this.toRadians(lat2)) *
        Math.sin(dLng / 2) *
        Math.sin(dLng / 2);
    const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    return R * c; // Distance in kilometers
  }
 toRadians(degrees) {
    return degrees * (Math.PI / 180);
 }
}
// Usage
const geo = new GeolocationService();
try {
 const position = await geo.getCurrentPosition();
  const { latitude, longitude } = position.coords;
 const address = await geo.getAddressFromCoords(latitude, longitude);
  console.log("Current location:", address);
 // Calculate distance to a point of interest
 const distance = geo.calculateDistance(
    latitude,
    longitude,
   40.7128,
    -74.006 // New York City
  );
  console.log(`Distance to NYC: ${distance.toFixed(2)} km`);
} catch (error) {
  console.error("Location error:", error);
}
```

#### File API

#### File Input Handling

```
// HTML: <input type="file" id="fileInput" multiple accept="image/*">
const fileInput = document.getElementById("fileInput");
fileInput.addEventListener("change", (e) => {
  const files = Array.from(e.target.files);
  files.forEach((file) => {
    console.log("File info:", {
      name: file.name,
      size: file.size,
      type: file.type,
      lastModified: new Date(file.lastModified),
    });
    // Process each file
    processFile(file);
 });
});
function processFile(file) {
  const reader = new FileReader();
  // Read as different formats
  if (file.type.startsWith("image/")) {
    reader.onload = (e) => {
      const img = new Image();
      img.onload = () => {
        console.log(`Image dimensions: ${img.width}x${img.height}`);
        displayImage(e.target.result);
      };
      img.src = e.target.result;
    };
    reader.readAsDataURL(file);
  } else if (file.type === "text/plain") {
    reader.onload = (e) => {
      console.log("Text content:", e.target.result);
    };
    reader.readAsText(file);
  } else {
    reader.onload = (e) => {
      console.log("Binary data:", e.target.result);
    };
    reader.readAsArrayBuffer(file);
  }
```

```
reader.onerror = () => {
   console.error("File reading error:", reader.error);
};
}
```

#### File Validation and Processing

```
class FileProcessor {
 constructor(options = {}) {
   this.maxSize = options.maxSize \parallel 5 * 1024 * 1024; // 5MB
   this.allowedTypes = options.allowedTypes || [];
   this.maxFiles = options.maxFiles | 10;
 }
 validateFile(file) {
   const errors = [];
    // Size validation
   if (file.size > this.maxSize) {
      errors.push(
        `File too large. Max size: ${this.formatBytes(this.maxSize)}`
     );
    }
    // Type validation
    if (this.allowedTypes.length > ∅) {
      const isAllowed = this.allowedTypes.some((type) => {
        if (type.endsWith("/*")) {
          return file.type.startsWith(type.slice(∅, -1));
        }
       return file.type === type;
      });
      if (!isAllowed) {
        errors.push(
          `File type not allowed. Allowed: ${this.allowedTypes.join(", ")}`
        );
      }
    }
    return errors;
  }
 validateFiles(files) {
    if (files.length > this.maxFiles) {
      return [`Too many files. Max: ${this.maxFiles}`];
    }
    const allErrors = [];
    files.forEach((file, index) => {
      const errors = this.validateFile(file);
```

```
if (errors.length > 0) {
      allErrors.push(
        `File ${index + 1} (${file.name}): ${errors.join(", ")}`
      );
 });
 return allErrors;
}
async readFileAsDataURL(file) {
 return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => resolve(reader.result);
    reader.onerror = () => reject(reader.error);
    reader.readAsDataURL(file);
 });
}
async readFileAsText(file) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => resolve(reader.result);
    reader.onerror = () => reject(reader.error);
    reader.readAsText(file);
 });
}
async compressImage(file, quality = 0.8, maxWidth = 1920, maxHeight = 1080) {
  return new Promise((resolve) => {
    const canvas = document.createElement("canvas");
    const ctx = canvas.getContext("2d");
    const img = new Image();
    img.onload = () => {
     // Calculate new dimensions
     let { width, height } = img;
      if (width > maxWidth || height > maxHeight) {
        const ratio = Math.min(maxWidth / width, maxHeight / height);
        width *= ratio;
        height *= ratio;
      }
      canvas.width = width;
      canvas.height = height;
      // Draw and compress
      ctx.drawImage(img, 0, 0, width, height);
      canvas.toBlob(resolve, file.type, quality);
    };
    img.src = URL.createObjectURL(file);
  });
```

```
formatBytes(bytes) {
   if (bytes === 0) return "0 Bytes";
    const k = 1024;
   const sizes = ["Bytes", "KB", "MB", "GB"];
   const i = Math.floor(Math.log(bytes) / Math.log(k));
    return parseFloat((bytes / Math.pow(k, i)).toFixed(2)) + " " + sizes[i];
  }
  generateThumbnail(file, size = 150) {
    return new Promise((resolve) => {
      const canvas = document.createElement("canvas");
      const ctx = canvas.getContext("2d");
      const img = new Image();
      img.onload = () => {
        canvas.width = size;
        canvas.height = size;
        // Calculate crop area for square thumbnail
        const minDim = Math.min(img.width, img.height);
        const x = (img.width - minDim) / 2;
        const y = (img.height - minDim) / 2;
        ctx.drawImage(img, x, y, minDim, minDim, 0, 0, size, size);
        canvas.toBlob(resolve, "image/jpeg", 0.8);
      };
      img.src = URL.createObjectURL(file);
    });
 }
}
// Usage
const processor = new FileProcessor({
 maxSize: 10 * 1024 * 1024, // 10MB
 allowedTypes: ["image/*", "application/pdf"],
 maxFiles: 5,
});
fileInput.addEventListener("change", async (e) => {
 const files = Array.from(e.target.files);
 // Validate files
 const errors = processor.validateFiles(files);
 if (errors.length > ∅) {
   console.error("Validation errors:", errors);
    return;
 }
 // Process each file
 for (const file of files) {
   if (file.type.startsWith("image/")) {
```

```
// Compress image
const compressed = await processor.compressImage(file);
console.log(
    `Original: ${processor.formatBytes(
        file.size
    )}, Compressed: ${processor.formatBytes(compressed.size)}`);

// Generate thumbnail
const thumbnail = await processor.generateThumbnail(file);
displayThumbnail(thumbnail);
}

}
}
}
```

# Canvas API

#### **Basic Canvas Operations**

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
// Set canvas size
canvas.width = 800;
canvas.height = 600;
// Basic shapes
ctx.fillStyle = "#ff0000";
ctx.fillRect(10, 10, 100, 100);
ctx.strokeStyle = "#0000ff";
ctx.lineWidth = 3;
ctx.strokeRect(150, 10, 100, 100);
// Circles
ctx.beginPath();
ctx.arc(200, 200, 50, 0, 2 * Math.PI);
ctx.fillStyle = "#00ff00";
ctx.fill();
// Lines and paths
ctx.beginPath();
ctx.moveTo(300, 100);
ctx.lineTo(400, 200);
ctx.lineTo(300, 300);
ctx.closePath();
ctx.stroke();
// Text
ctx.font = "24px Arial";
```

```
ctx.fillStyle = "#000000";
ctx.fillText("Hello Canvas!", 50, 400);

// Gradients
const gradient = ctx.createLinearGradient(0, 0, 200, 0);
gradient.addColorStop(0, "#ff0000");
gradient.addColorStop(1, "#0000ff");
ctx.fillStyle = gradient;
ctx.fillRect(500, 100, 200, 100);
```

#### Animation with Canvas

```
class CanvasAnimation {
 constructor(canvas) {
   this.canvas = canvas;
   this.ctx = canvas.getContext("2d");
   this.animationId = null;
   this.lastTime = 0;
   this.objects = [];
 }
 start() {
   this.animate(∅);
 }
 stop() {
   if (this.animationId) {
     cancelAnimationFrame(this.animationId);
     this.animationId = null;
   }
 }
 animate(currentTime) {
   const deltaTime = currentTime - this.lastTime;
   this.lastTime = currentTime;
   // Clear canvas
   this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
   // Update and draw objects
   this.objects.forEach((obj) => {
     obj.update(deltaTime);
     obj.draw(this.ctx);
   });
   this.animationId = requestAnimationFrame((time) => this.animate(time));
 }
 addObject(obj) {
   this.objects.push(obj);
```

```
removeObject(obj) {
    const index = this.objects.indexOf(obj);
   if (index > -1) {
      this.objects.splice(index, 1);
   }
 }
}
// Animated ball class
class Ball {
 constructor(x, y, radius, color, vx, vy) {
   this.x = x;
   this.y = y;
   this.radius = radius;
   this.color = color;
   this.vx = vx;
   this.vy = vy;
   this.gravity = 0.5;
   this.bounce = 0.8;
 }
 update(deltaTime) {
   // Apply gravity
   this.vy += this.gravity;
   // Update position
   this.x += this.vx;
   this.y += this.vy;
   // Bounce off walls
   if (this.x + this.radius > canvas.width || this.x - this.radius < 0) {
     this.vx *= -this.bounce;
     this.x = Math.max(
       this.radius,
       Math.min(canvas.width - this.radius, this.x)
     );
    }
   if (this.y + this.radius > canvas.height || this.y - this.radius < 0) {
     this.vy *= -this.bounce;
     this.y = Math.max(
       this.radius,
       Math.min(canvas.height - this.radius, this.y)
      );
    }
  }
 draw(ctx) {
   ctx.beginPath();
    ctx.arc(this.x, this.y, this.radius, 0, 2 * Math.PI);
    ctx.fillStyle = this.color;
    ctx.fill();
    ctx.strokeStyle = "#000";
```

```
ctx.stroke();
 }
}
// Usage
const animation = new CanvasAnimation(canvas);
// Add some bouncing balls
for (let i = 0; i < 10; i++) {
 const ball = new Ball(
    Math.random() * canvas.width,
    Math.random() * canvas.height,
    10 + Math.random() * 20,
    `hsl(${Math.random() * 360}, 70%, 50%)`,
    (Math.random() - 0.5) * 10,
    (Math.random() - 0.5) * 10
 );
 animation.addObject(ball);
animation.start();
```

#### **Canvas Drawing Tools**

```
class DrawingTool {
 constructor(canvas) {
   this.canvas = canvas;
   this.ctx = canvas.getContext("2d");
   this.isDrawing = false;
   this.tool = "pen";
   this.color = "#000000";
   this.lineWidth = 2;
   this.history = [];
   this.historyStep = -1;
   this.setupEventListeners();
   this.saveState();
 }
 setupEventListeners() {
   this.canvas.addEventListener("mousedown", (e) => this.startDrawing(e));
   this.canvas.addEventListener("mousemove", (e) => this.draw(e));
   this.canvas.addEventListener("mouseup", () => this.stopDrawing());
   this.canvas.addEventListener("mouseout", () => this.stopDrawing());
 }
 startDrawing(e) {
   this.isDrawing = true;
   const rect = this.canvas.getBoundingClientRect();
   this.lastX = e.clientX - rect.left;
   this.lastY = e.clientY - rect.top;
```

```
if (this.tool === "pen" || this.tool === "eraser") {
   this.ctx.beginPath();
   this.ctx.moveTo(this.lastX, this.lastY);
}
draw(e) {
 if (!this.isDrawing) return;
  const rect = this.canvas.getBoundingClientRect();
  const currentX = e.clientX - rect.left;
  const currentY = e.clientY - rect.top;
 this.ctx.lineWidth = this.lineWidth;
  this.ctx.lineCap = "round";
  this.ctx.lineJoin = "round";
  switch (this.tool) {
    case "pen":
      this.ctx.globalCompositeOperation = "source-over";
      this.ctx.strokeStyle = this.color;
      this.ctx.lineTo(currentX, currentY);
      this.ctx.stroke();
      break;
    case "eraser":
      this.ctx.globalCompositeOperation = "destination-out";
      this.ctx.lineTo(currentX, currentY);
      this.ctx.stroke();
      break;
    case "line":
      this.redrawCanvas();
      this.ctx.strokeStyle = this.color;
      this.ctx.beginPath();
      this.ctx.moveTo(this.lastX, this.lastY);
      this.ctx.lineTo(currentX, currentY);
      this.ctx.stroke();
      break;
    case "rectangle":
      this.redrawCanvas();
      this.ctx.strokeStyle = this.color;
      this.ctx.strokeRect(
        this.lastX,
        this.lastY,
        currentX - this.lastX,
        currentY - this.lastY
      );
      break;
    case "circle":
      this.redrawCanvas();
```

```
const radius = Math.sqrt(
        Math.pow(currentX - this.lastX, 2) +
          Math.pow(currentY - this.lastY, 2)
      );
      this.ctx.strokeStyle = this.color;
      this.ctx.beginPath();
      this.ctx.arc(this.lastX, this.lastY, radius, 0, 2 * Math.PI);
      this.ctx.stroke();
      break;
  }
}
stopDrawing() {
 if (this.isDrawing) {
   this.isDrawing = false;
    this.saveState();
  }
}
setTool(tool) {
 this.tool = tool;
setColor(color) {
 this.color = color;
}
setLineWidth(width) {
 this.lineWidth = width;
}
clear() {
 this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
  this.saveState();
}
saveState() {
 this.historyStep++;
  if (this.historyStep < this.history.length) {</pre>
   this.history.length = this.historyStep;
  this.history.push(this.canvas.toDataURL());
}
undo() {
 if (this.historyStep > 0) {
   this.historyStep--;
   this.restoreState();
}
redo() {
  if (this.historyStep < this.history.length - 1) {</pre>
   this.historyStep++;
```

```
this.restoreState();
    }
  }
  restoreState() {
    const img = new Image();
    img.onload = () => {
      this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
      this.ctx.drawImage(img, 0, 0);
    };
    img.src = this.history[this.historyStep];
  }
  redrawCanvas() {
    if (this.historyStep >= 0) {
      const img = new Image();
      img.onload = () => {
        this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
        this.ctx.drawImage(img, ∅, ∅);
      };
      img.src = this.history[this.historyStep];
    }
  }
  exportImage(format = "image/png") {
    return this.canvas.toDataURL(format);
  }
  importImage(dataURL) {
    const img = new Image();
    img.onload = () => {
      this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);
      this.ctx.drawImage(img, ∅, ∅);
      this.saveState();
    };
    img.src = dataURL;
  }
}
// Usage
const drawingTool = new DrawingTool(canvas);
// Tool controls
document.getElementById("penTool").addEventListener("click", () => {
  drawingTool.setTool("pen");
});
document.getElementById("colorPicker").addEventListener("change", (e) => {
  drawingTool.setColor(e.target.value);
});
document.getElementById("lineWidth").addEventListener("input", (e) => {
  drawingTool.setLineWidth(e.target.value);
});
```

```
document.getElementById("undo").addEventListener("click", () => {
 drawingTool.undo();
});
document.getElementById("clear").addEventListener("click", () => {
 drawingTool.clear();
});
```

### Web Workers

#### Basic Web Worker

```
// main.js
if (typeof Worker !== "undefined") {
 const worker = new Worker("worker.js");
 // Send data to worker
 worker.postMessage({
   command: "calculate",
   data: [1, 2, 3, 4, 5],
 });
 // Receive data from worker
 worker.onmessage = function (e) {
   console.log("Result from worker:", e.data);
 };
 // Handle errors
 worker.onerror = function (error) {
   console.error("Worker error:", error);
 };
 // Terminate worker when done
 // worker.terminate();
  console.log("Web Workers not supported");
}
```

```
// worker.js
self.onmessage = function (e) {
 const { command, data } = e.data;
 switch (command) {
    case "calculate":
      const result = heavyCalculation(data);
      self.postMessage(result);
      break;
```

```
case "processImage":
      const processedImage = processImageData(data);
      self.postMessage(processedImage);
      break;
    default:
      self.postMessage({ error: "Unknown command" });
 }
};
function heavyCalculation(numbers) {
 // Simulate heavy computation
 let result = 0;
 for (let i = 0; i < 1000000; i++) {
    result += numbers.reduce((sum, num) => sum + num * Math.random(), 0);
 }
 return result;
}
function processImageData(imageData) {
 // Process image data (e.g., apply filters)
 const data = imageData.data;
 // Apply grayscale filter
 for (let i = 0; i < data.length; i += 4) {
   const gray = data[i] * 0.299 + data[i + 1] * 0.587 + data[i + 2] * 0.114;
   data[i] = gray; // Red
   data[i + 1] = gray; // Green
   data[i + 2] = gray; // Blue
   // Alpha channel (i + 3) remains unchanged
 }
 return imageData;
}
```

#### Worker Pool Manager

```
class WorkerPool {
  constructor(workerScript, poolSize = navigator.hardwareConcurrency || 4) {
    this.workerScript = workerScript;
    this.poolSize = poolSize;
    this.workers = [];
    this.queue = [];
    this.activeJobs = new Map();

    this.initializeWorkers();
}

initializeWorkers() {
    for (let i = 0; i < this.poolSize; i++) {</pre>
```

```
const worker = new Worker(this.workerScript);
    worker.id = i;
    worker.busy = false;
    worker.onmessage = (e) => {
     this.handleWorkerMessage(worker, e);
    };
    worker.onerror = (error) => {
     this.handleWorkerError(worker, error);
    };
   this.workers.push(worker);
 }
}
execute(data, transferable = []) {
  return new Promise((resolve, reject) => {
    const job = {
      id: Date.now() + Math.random(),
     data,
     transferable,
      resolve,
     reject,
    };
    const availableWorker = this.workers.find((w) => !w.busy);
    if (availableWorker) {
     this.assignJob(availableWorker, job);
    } else {
     this.queue.push(job);
 });
assignJob(worker, job) {
 worker.busy = true;
 this.activeJobs.set(worker.id, job);
 worker.postMessage(job.data, job.transferable);
}
handleWorkerMessage(worker, e) {
  const job = this.activeJobs.get(worker.id);
 if (job) {
    job.resolve(e.data);
    this.activeJobs.delete(worker.id);
    worker.busy = false;
    // Process next job in queue
    if (this.queue.length > 0) {
      const nextJob = this.queue.shift();
      this.assignJob(worker, nextJob);
```

```
}
  }
  handleWorkerError(worker, error) {
    const job = this.activeJobs.get(worker.id);
    if (job) {
      job.reject(error);
      this.activeJobs.delete(worker.id);
      worker.busy = false;
    }
  }
 terminate() {
    this.workers.forEach((worker) => worker.terminate());
    this.workers = [];
   this.queue = [];
    this.activeJobs.clear();
  }
  getStats() {
    return {
      totalWorkers: this.workers.length,
      busyWorkers: this.workers.filter((w) => w.busy).length,
      queueLength: this.queue.length,
      activeJobs: this.activeJobs.size,
    };
  }
}
// Usage
const workerPool = new WorkerPool("worker.js", 4);
// Process multiple tasks in parallel
const tasks = [
  { command: "calculate", data: [1, 2, 3] },
  { command: "calculate", data: [4, 5, 6] },
 { command: "calculate", data: [7, 8, 9] },
  { command: "calculate", data: [10, 11, 12] },
1;
Promise.all(tasks.map((task) => workerPool.execute(task)))
  .then((results) => {
    console.log("All tasks completed:", results);
    console.log("Worker stats:", workerPool.getStats());
  })
  .catch((error) => {
   console.error("Task error:", error);
  });
```



#### **Basic Service Worker**

```
// main.js - Register service worker
if ("serviceWorker" in navigator) {
 window.addEventListener("load", () => {
    navigator.serviceWorker
      .register("/sw.js")
      .then((registration) => {
        console.log("SW registered:", registration);
        // Listen for updates
        registration.addEventListener("updatefound", () => {
          const newWorker = registration.installing;
          newWorker.addEventListener("statechange", () => {
            if (
              newWorker.state === "installed" &&
              navigator.serviceWorker.controller
            ) {
              // New version available
              showUpdateNotification();
          });
        });
      })
      .catch((error) => {
        console.log("SW registration failed:", error);
      });
 });
 // Listen for messages from service worker
 navigator.serviceWorker.addEventListener("message", (e) => {
   console.log("Message from SW:", e.data);
 });
}
function showUpdateNotification() {
 if (confirm("New version available! Reload to update?")) {
   window.location.reload();
 }
}
```

```
// sw.js - Service Worker
const CACHE_NAME = "my-app-v1";
const urlsToCache = [
   "/",
   "/styles/main.css",
   "/scripts/main.js",
   "/images/logo.png",
```

```
"/offline.html",
];
// Install event - cache resources
self.addEventListener("install", (e) => {
  console.log("Service Worker installing");
  e.waitUntil(
    caches
      .open(CACHE_NAME)
      .then((cache) => {
        console.log("Caching app shell");
        return cache.addAll(urlsToCache);
      })
      .then(() => {
        // Force activation of new service worker
        return self.skipWaiting();
      })
 );
});
// Activate event - clean up old caches
self.addEventListener("activate", (e) => {
  console.log("Service Worker activating");
  e.waitUntil(
    caches
      .keys()
      .then((cacheNames) => {
        return Promise.all(
          cacheNames.map((cacheName) => {
            if (cacheName !== CACHE NAME) {
              console.log("Deleting old cache:", cacheName);
              return caches.delete(cacheName);
          })
        );
      })
      .then(() => {
        // Take control of all pages
        return self.clients.claim();
      })
 );
});
// Fetch event - serve from cache, fallback to network
self.addEventListener("fetch", (e) => {
  e.respondWith(
    caches
      .match(e.request)
      .then((response) => {
        // Return cached version or fetch from network
        if (response) {
          return response;
```

```
return fetch(e.request).then((response) => {
          // Don't cache non-successful responses
          if (
            !response ||
            response.status !== 200 ||
            response.type !== "basic"
          ) {
            return response;
          }
          // Clone response for caching
          const responseToCache = response.clone();
          caches.open(CACHE_NAME).then((cache) => {
            cache.put(e.request, responseToCache);
          });
          return response;
        });
      })
      .catch(() => {
        // Show offline page for navigation requests
        if (e.request.destination === "document") {
          return caches.match("/offline.html");
      })
 );
});
// Background sync
self.addEventListener("sync", (e) => {
  if (e.tag === "background-sync") {
    e.waitUntil(doBackgroundSync());
  }
});
// Push notifications
self.addEventListener("push", (e) => {
  const options = {
    body: e.data ? e.data.text() : "Default notification body",
    icon: "/images/icon-192x192.png",
    badge: "/images/badge-72x72.png",
    vibrate: [100, 50, 100],
    data: {
      dateOfArrival: Date.now(),
      primaryKey: 1,
    },
    actions: [
        action: "explore",
        title: "Explore",
        icon: "/images/checkmark.png",
```

```
},
        action: "close",
        title: "Close",
        icon: "/images/xmark.png",
     },
    ],
  };
  e.waitUntil(
    self.registration.showNotification("My App Notification", options)
 );
});
// Notification click
self.addEventListener("notificationclick", (e) => {
  e.notification.close();
 if (e.action === "explore") {
    // Open app
   e.waitUntil(clients.openWindow("/explore"));
  } else if (e.action === "close") {
   // Just close notification
   return;
 } else {
    // Default action
    e.waitUntil(clients.openWindow("/"));
  }
});
function doBackgroundSync() {
 // Perform background tasks
  return fetch("/api/sync")
    .then((response) => response.json())
    .then((data) => {
      console.log("Background sync completed:", data);
    })
    .catch((error) => {
      console.error("Background sync failed:", error);
    });
}
// Message handling
self.addEventListener("message", (e) => {
  if (e.data && e.data.type === "SKIP WAITING") {
    self.skipWaiting();
  }
});
```

### Intersection Observer

#### **Basic Intersection Observer**

```
// Lazy loading images
const imageObserver = new IntersectionObserver(
  (entries, observer) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        const img = entry.target;
        img.src = img.dataset.src;
        img.classList.remove("lazy");
        observer.unobserve(img);
   });
  },
    rootMargin: "50px 0px", // Load 50px before entering viewport
    threshold: 0.1,
  }
);
// Observe all lazy images
document.querySelectorAll("img[data-src]").forEach((img) => {
  imageObserver.observe(img);
});
```

#### Advanced Intersection Observer

```
class IntersectionManager {
 constructor() {
   this.observers = new Map();
  }
 createObserver(name, callback, options = {}) {
    const defaultOptions = {
      root: null,
      rootMargin: "0px",
     threshold: 0,
   };
    const observer = new IntersectionObserver(callback, {
      ...defaultOptions,
      ...options,
    });
   this.observers.set(name, observer);
    return observer;
  }
 getObserver(name) {
    return this.observers.get(name);
```

```
destroyObserver(name) {
    const observer = this.observers.get(name);
    if (observer) {
      observer.disconnect();
      this.observers.delete(name);
    }
  }
  destroyAll() {
    this.observers.forEach((observer) => observer.disconnect());
    this.observers.clear();
 }
}
const intersectionManager = new IntersectionManager();
// Lazy loading observer
intersectionManager.createObserver(
  "lazyLoad",
  (entries) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        const element = entry.target;
        if (element.dataset.src) {
          element.src = element.dataset.src;
          element.removeAttribute("data-src");
        }
        if (element.dataset.srcset) {
          element.srcset = element.dataset.srcset;
          element.removeAttribute("data-srcset");
        }
        element.classList.add("loaded");
        intersectionManager.getObserver("lazyLoad").unobserve(element);
    });
  },
    rootMargin: "100px 0px",
    threshold: 0.1,
  }
);
// Infinite scroll observer
intersectionManager.createObserver(
  "infiniteScroll",
  (entries) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        loadMoreContent();
```

```
});
  },
    rootMargin: "200px 0px",
    threshold: 1.0,
  }
);
// Animation trigger observer
intersectionManager.createObserver(
  "animations",
  (entries) => {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        entry.target.classList.add("animate-in");
      } else {
        entry.target.classList.remove("animate-in");
    });
  },
    threshold: 0.3,
  }
);
// Progress tracking observer
intersectionManager.createObserver(
  "progress",
  (entries) => {
    entries.forEach((entry) => {
      const progress = Math.round(entry.intersectionRatio * 100);
      updateReadingProgress(progress);
    });
 },
   threshold: Array.from({ length: 101 }, (_, i) => i / 100), // 0, 0.01, 0.02,
..., 1.0
 }
);
// Usage
document.querySelectorAll("[data-src]").forEach((el) => {
  intersectionManager.getObserver("lazyLoad").observe(el);
});
const sentinel = document.querySelector("#scroll-sentinel");
if (sentinel) {
  intersectionManager.getObserver("infiniteScroll").observe(sentinel);
}
document.querySelectorAll(".animate-on-scroll").forEach((el) => {
  intersectionManager.getObserver("animations").observe(el);
});
```

```
const article = document.querySelector("article");
if (article) {
   intersectionManager.getObserver("progress").observe(article);
}

function loadMoreContent() {
   // Load more content implementation
   console.log("Loading more content...");
}

function updateReadingProgress(progress) {
   const progressBar = document.querySelector("#reading-progress");
   if (progressBar) {
      progressBar.style.width = `${progress}%`;
   }
}
```

# **Q** Mutation Observer

#### **Basic Mutation Observer**

```
// Watch for DOM changes
const targetNode = document.getElementById("content");
const observer = new MutationObserver((mutations) => {
 mutations.forEach((mutation) => {
    console.log("Mutation type:", mutation.type);
    switch (mutation.type) {
      case "childList":
        console.log("Children changed:", {
          added: mutation.addedNodes,
          removed: mutation.removedNodes,
        });
        break;
      case "attributes":
        console.log("Attribute changed:", {
          name: mutation.attributeName,
          oldValue: mutation.oldValue,
          newValue: mutation.target.getAttribute(mutation.attributeName),
        });
        break;
      case "characterData":
        console.log("Text content changed:", {
          oldValue: mutation.oldValue,
          newValue: mutation.target.textContent,
        });
```

```
break;
}
});
});

// Start observing
observer.observe(targetNode, {
  childList: true,
   attributes: true,
  attributeOldValue: true,
  characterData: true,
  characterDataOldValue: true,
  subtree: true, // Watch all descendants
});

// Stop observing
// observer.disconnect();
```

#### **Advanced Mutation Observer**

```
class DOMWatcher {
   constructor() {
       this.observers = new Map();
       this.watchers = new Map();
   }
   watch(selector, options = {}, callback) {
        const watchId = Date.now() + Math.random();
        const defaultOptions = {
            childList: true,
            attributes: false,
            attributeOldValue: false,
            characterData: false,
            characterDataOldValue: false,
            subtree: true
       };
        const observer = new MutationObserver((mutations) => {
            const relevantMutations = mutations.filter(mutation => {
                if (selector === '*') return true;
                // Check if mutation affects elements matching selector
                if (mutation.type === 'childList') {
                    const addedElements = Array.from(mutation.addedNodes)
                        .filter(node => node.nodeType === Node.ELEMENT_NODE);
                    const removedElements = Array.from(mutation.removedNodes)
                        .filter(node => node.nodeType === Node.ELEMENT_NODE);
                    return addedElements.some(el => el.matches(selector)) ||
                           removedElements.some(el => el.matches(selector));
```

```
return mutation.target.matches(selector);
            });
            if (relevantMutations.length > ∅) {
                callback(relevantMutations, observer);
            }
        });
        const target = document.body; // Watch entire document
        observer.observe(target, { ...defaultOptions, ...options });
        this.observers.set(watchId, observer);
        this.watchers.set(watchId, { selector, options, callback });
        return watchId;
   }
   unwatch(watchId) {
        const observer = this.observers.get(watchId);
        if (observer) {
            observer.disconnect();
            this.observers.delete(watchId);
           this.watchers.delete(watchId);
       }
   }
   unwatchAll() {
       this.observers.forEach(observer => observer.disconnect());
        this.observers.clear();
       this.watchers.clear();
   }
   // Watch for specific element additions
   onElementAdded(selector, callback) {
        return this.watch(selector, { childList: true }, (mutations) => {
            mutations.forEach(mutation => {
                if (mutation.type === 'childList') {
                    const addedElements = Array.from(mutation.addedNodes)
                        .filter(node => node.nodeType === Node.ELEMENT NODE &&
node.matches(selector));
                    addedElements.forEach(element => callback(element));
                }
            });
        });
   }
   // Watch for specific element removals
   onElementRemoved(selector, callback) {
        return this.watch(selector, { childList: true }, (mutations) => {
            mutations.forEach(mutation => {
                if (mutation.type === 'childList') {
```

```
const removedElements = Array.from(mutation.removedNodes)
                        .filter(node => node.nodeType === Node.ELEMENT_NODE &&
node.matches(selector));
                    removedElements.forEach(element => callback(element));
                }
            });
        });
    }
    // Watch for attribute changes
    onAttributeChange(selector, attributeName, callback) {
        return this.watch(selector, {
            attributes: true,
            attributeOldValue: true,
            attributeFilter: attributeName ? [attributeName] : undefined
        }, (mutations) => {
            mutations.forEach(mutation => {
                if (mutation.type === 'attributes') {
                    callback(mutation.target, {
                        attributeName: mutation.attributeName,
                        oldValue: mutation.oldValue,
                        newValue:
mutation.target.getAttribute(mutation.attributeName)
                    });
                }
            });
        });
    }
    // Watch for text content changes
    onTextChange(selector, callback) {
        return this.watch(selector, {
            characterData: true,
            characterDataOldValue: true,
            subtree: true
        }, (mutations) => {
            mutations.forEach(mutation => {
                if (mutation.type === 'characterData') {
                    callback(mutation.target, {
                        oldValue: mutation.oldValue,
                        newValue: mutation.target.textContent
                    });
                }
            });
        });
    }
}
// Usage
const domWatcher = new DOMWatcher();
// Watch for new buttons being added
const buttonWatchId = domWatcher.onElementAdded('button', (button) => {
```

```
console.log('New button added:', button);
    // Auto-setup event listeners for new buttons
    button.addEventListener('click', handleButtonClick);
});
// Watch for class changes on specific elements
const classWatchId = domWatcher.onAttributeChange('.status-indicator', 'class',
(element, change) => {
    console.log('Class changed:', change);
    if (change.newValue.includes('error')) {
        showErrorNotification();
    }
});
// Watch for text changes in live content
const textWatchId = domWatcher.onTextChange('#live-content', (textNode, change) =>
{
    console.log('Text updated:', change);
    updateLastModified();
});
function handleButtonClick(e) {
    console.log('Button clicked:', e.target);
}
function showErrorNotification() {
    console.log('Error detected!');
}
function updateLastModified() {
    document.querySelector('#last-modified').textContent = new
Date().toLocaleString();
}
## 🙉 History API
### Basic History Management
```javascript
// Push new state
history.pushState({ page: 'about' }, 'About Page', '/about');
// Replace current state
history.replaceState({ page: 'home', section: 'hero' }, 'Home - Hero',
'/home#hero');
// Go back/forward
history.back();
history.forward();
history.go(-2); // Go back 2 pages
// Listen for popstate events (back/forward button)
window.addEventListener('popstate', (e) => {
```

```
console.log('State changed:', e.state);
    handleRouteChange(e.state);
});
function handleRouteChange(state) {
    if (state) {
        // Update page based on state
        switch (state.page) {
            case 'home':
                showHomePage();
                break;
            case 'about':
                showAboutPage();
                break;
            default:
                show404Page();
        }
    }
}
```

## Single Page Application Router

```
class SPARouter {
 constructor() {
   this.routes = new Map();
   this.currentRoute = null;
   this.beforeRouteChange = null;
   this.afterRouteChange = null;
   this.init();
 }
 init() {
   // Handle popstate events
   window.addEventListener("popstate", (e) => {
     this.handleRouteChange(window.location.pathname, e.state, false);
   });
   // Handle initial page load
   this.handleRouteChange(window.location.pathname, null, false);
   // Intercept link clicks
   document.addEventListener("click", (e) => {
     if (e.target.matches('a[href^="/"]')) {
        e.preventDefault();
       this.navigate(e.target.getAttribute("href"));
     }
   });
 }
 addRoute(path, handler, options = {}) {
```

```
this.routes.set(path, {
    handler,
    title: options.title || "",
    meta: options.meta || {},
    guards: options.guards || [],
 });
}
navigate(path, state = {}, replace = false) {
  const route = this.findRoute(path);
  if (!route) {
    console.error("Route not found:", path);
    return;
  }
 this.handleRouteChange(path, state, true, replace);
}
findRoute(path) {
 // Exact match first
  if (this.routes.has(path)) {
    return { path, ...this.routes.get(path) };
  }
  // Pattern matching for dynamic routes
  for (const [routePath, routeConfig] of this.routes) {
    const params = this.matchRoute(routePath, path);
    if (params) {
      return { path: routePath, params, ...routeConfig };
  }
  return null;
}
matchRoute(routePath, actualPath) {
  const routeParts = routePath.split("/");
  const actualParts = actualPath.split("/");
  if (routeParts.length !== actualParts.length) {
    return null;
  }
  const params = {};
  for (let i = 0; i < routeParts.length; i++) {</pre>
    const routePart = routeParts[i];
    const actualPart = actualParts[i];
    if (routePart.startsWith(":")) {
      // Dynamic parameter
      params[routePart.slice(1)] = actualPart;
    } else if (routePart !== actualPart) {
```

```
return null;
   }
 }
 return params;
}
async handleRouteChange(path, state, updateHistory, replace = false) {
 const route = this.findRoute(path);
 if (!route) {
   console.error("Route not found:", path);
   return;
 }
 // Run before route change hook
 if (this.beforeRouteChange) {
   const canProceed = await this.beforeRouteChange(route, this.currentRoute);
   if (!canProceed) return;
 }
 // Run route guards
 for (const guard of route.guards) {
   const canProceed = await guard(route);
   if (!canProceed) return;
 }
 // Update browser history
 if (updateHistory) {
   const method = replace ? "replaceState" : "pushState";
   history[method](state, route.title, path);
 }
 // Update document title
 if (route.title) {
   document.title = route.title;
 }
 // Update meta tags
 this.updateMetaTags(route.meta);
 // Execute route handler
 try {
   await route.handler(route.params || {}, state);
   this.currentRoute = route;
   // Run after route change hook
   if (this.afterRouteChange) {
     this.afterRouteChange(route);
   }
 } catch (error) {
   console.error("Route handler error:", error);
  }
```

```
updateMetaTags(meta) {
    Object.entries(meta).forEach(([name, content]) => {
      let metaTag = document.querySelector(`meta[name="${name}"]`);
      if (!metaTag) {
        metaTag = document.createElement("meta");
        metaTag.name = name;
        document.head.appendChild(metaTag);
      }
      metaTag.content = content;
    });
  }
  setBeforeRouteChange(callback) {
   this.beforeRouteChange = callback;
  }
  setAfterRouteChange(callback) {
    this.afterRouteChange = callback;
  }
  getCurrentRoute() {
    return this.currentRoute;
 }
}
// Usage
const router = new SPARouter();
// Add routes
router.addRoute(
 "/",
  async () => {
    document.getElementById("app").innerHTML = "<h1>Home Page</h1>";
 },
    title: "Home - My App",
    meta: {
      description: "Welcome to our home page",
    },
  }
);
router.addRoute(
 "/about",
  async () => {
    document.getElementById("app").innerHTML = "<h1>About Page</h1>";
  },
    title: "About - My App",
```

```
router.addRoute(
 "/user/:id",
  async (params) => {
    const userId = params.id;
    const user = await fetchUser(userId);
   document.getElementById("app").innerHTML = `<h1>User: ${user.name}</h1>`;
  },
    title: "User Profile - My App",
    guards: [requireAuth],
  }
);
// Route guards
async function requireAuth(route) {
  const isAuthenticated = checkAuthStatus();
  if (!isAuthenticated) {
    router.navigate("/login");
   return false;
 }
 return true;
}
// Hooks
router.setBeforeRouteChange(async (newRoute, oldRoute) => {
  console.log("Navigating from", oldRoute?.path, "to", newRoute.path);
  showLoadingSpinner();
 return true;
});
router.setAfterRouteChange((route) => {
  console.log("Route changed to:", route.path);
  hideLoadingSpinner();
 trackPageView(route.path);
});
function fetchUser(id) {
 return fetch(`/api/users/${id}`).then((r) => r.json());
}
function checkAuthStatus() {
  return localStorage.getItem("authToken") !== null;
}
function showLoadingSpinner() {
  document.getElementById("loading").style.display = "block";
}
function hideLoadingSpinner() {
  document.getElementById("loading").style.display = "none";
}
function trackPageView(path) {
```

```
// Analytics tracking
console.log("Page view:", path);
}
```

## △ Notification API

#### **Basic Notifications**

```
// Check if notifications are supported
if ("Notification" in window) {
  // Request permission
  Notification.requestPermission().then((permission) => {
    if (permission === "granted") {
      showNotification("Welcome!", "Thanks for enabling notifications.");
    } else {
      console.log("Notification permission denied");
  });
} else {
  console.log("Notifications not supported");
function showNotification(title, body, options = {}) {
  if (Notification.permission === "granted") {
    const notification = new Notification(title, {
      body,
      icon: "/images/icon-192x192.png",
      badge: "/images/badge-72x72.png",
      image: "/images/notification-image.jpg",
      vibrate: [200, 100, 200],
      tag: "general", // Prevents duplicate notifications
      renotify: false,
      requireInteraction: false,
      silent: false,
      ...options,
    });
    // Handle notification events
    notification.onclick = () => {
      window.focus();
      notification.close();
    };
    notification.onshow = () => {
      console.log("Notification shown");
    };
    notification.onclose = () => {
      console.log("Notification closed");
    };
```

```
notification.onerror = () => {
    console.error("Notification error");
};

// Auto-close after 5 seconds
setTimeout(() => {
    notification.close();
}, 5000);

return notification;
}
```

## **Advanced Notification Manager**

```
class NotificationManager {
 constructor() {
   this.permission = Notification.permission;
   this.notifications = new Map();
   this.queue = [];
   this.isSupported = "Notification" in window;
 }
 async requestPermission() {
   if (!this.isSupported) {
     throw new Error("Notifications not supported");
   }
   if (this.permission === "default") {
     this.permission = await Notification.requestPermission();
   return this.permission;
 }
 async show(title, options = {}) {
   if (!this.isSupported) {
     console.warn("Notifications not supported");
     return null;
   }
   if (this.permission !== "granted") {
     const permission = await this.requestPermission();
     if (permission !== "granted") {
       console.warn("Notification permission denied");
       return null;
     }
   }
   const defaultOptions = {
```

```
body: "",
  icon: "/images/icon-192x192.png",
  badge: "/images/badge-72x72.png",
  tag: "default",
  renotify: false,
  requireInteraction: false,
  silent: false,
 vibrate: [100, 50, 100],
 data: {},
 actions: [],
};
const finalOptions = { ...defaultOptions, ...options };
// Close existing notification with same tag
if (finalOptions.tag && this.notifications.has(finalOptions.tag)) {
 this.notifications.get(finalOptions.tag).close();
}
const notification = new Notification(title, finalOptions);
// Store notification
if (finalOptions.tag) {
 this.notifications.set(finalOptions.tag, notification);
}
// Setup event handlers
notification.onclick = (e) => {
 if (options.onClick) {
   options.onClick(e, notification);
 } else {
   window.focus();
   notification.close();
 }
};
notification.onclose = (e) => {
 if (finalOptions.tag) {
   this.notifications.delete(finalOptions.tag);
 }
 if (options.onClose) {
    options.onClose(e, notification);
 }
};
notification.onerror = (e) => {
 if (options.onError) {
   options.onError(e, notification);
 }
};
notification.onshow = (e) => {
 if (options.onShow) {
    options.onShow(e, notification);
```

```
};
  // Auto-close if specified
  if (options.autoClose) {
   setTimeout(() => {
      notification.close();
    }, options.autoClose);
 return notification;
}
close(tag) {
  const notification = this.notifications.get(tag);
  if (notification) {
   notification.close();
}
closeAll() {
  this.notifications.forEach((notification) => notification.close());
  this.notifications.clear();
}
// Predefined notification types
success(title, body, options = {}) {
  return this.show(title, {
    body,
    icon: "/images/success-icon.png",
    tag: "success",
    autoClose: 3000,
    ...options,
  });
}
error(title, body, options = {}) {
  return this.show(title, {
    body,
    icon: "/images/error-icon.png",
    tag: "error",
    requireInteraction: true,
    vibrate: [200, 100, 200, 100, 200],
    ...options,
  });
warning(title, body, options = {}) {
  return this.show(title, {
    body,
    icon: "/images/warning-icon.png",
    tag: "warning",
    autoClose: 5000,
    ...options,
```

```
});
  }
  info(title, body, options = {}) {
    return this.show(title, {
      body,
      icon: "/images/info-icon.png",
      tag: "info",
      autoClose: 4000,
      ...options,
   });
  }
 // Queue notifications when permission is not granted
 queueNotification(title, options) {
    this.queue.push({ title, options });
  }
  async processQueue() {
   if (this.permission === "granted" && this.queue.length > 0) {
      for (const { title, options } of this.queue) {
        await this.show(title, options);
      }
     this.queue = [];
   }
  }
 getPermissionStatus() {
   return this.permission;
 isSupported() {
   return this.isSupported;
 }
}
// Usage
const notifications = new NotificationManager();
// Request permission on page load
notifications.requestPermission().then((permission) => {
 if (permission === "granted") {
    notifications.processQueue();
 }
});
// Show different types of notifications
notifications.success("Success!", "Your changes have been saved.");
notifications.error("Error!", "Failed to save changes. Please try again.");
notifications.warning("Warning!", "Your session will expire in 5 minutes.");
notifications.info("Info", "New features are available!");
// Custom notification with actions (requires service worker)
notifications.show("New Message", {
```

```
body: "You have received a new message from John.",
 tag: "message",
 data: { messageId: 123 },
  actions: [
    { action: "reply", title: "Reply", icon: "/images/reply-icon.png" },
      action: "mark-read",
     title: "Mark as Read",
     icon: "/images/read-icon.png",
   },
 ],
 onClick: (e, notification) => {
   // Handle notification click
   window.open("/messages/123");
   notification.close();
 },
});
```

## 

## 1. Storage Quota Exceeded

```
// X Not handling storage quota errors
localStorage.setItem("data", largeDataString); // May throw QuotaExceededError
// Handle storage errors gracefully
function safeSetItem(key, value) {
 try {
   localStorage.setItem(key, value);
   return true;
  } catch (error) {
   if (error.name === "QuotaExceededError") {
      console.warn("Storage quota exceeded");
      // Clear old data or ask user
      clearOldData();
      return false;
   throw error;
 }
}
```

## 2. Fetch API Error Handling

```
// X Not checking response status
fetch("/api/data")
   .then((response) => response.json()) // This will succeed even for 404/500
   .then((data) => console.log(data));
```

```
// Proper error handling
fetch("/api/data")
   .then((response) => {
    if (!response.ok) {
        throw new Error(`HTTP ${response.status}: ${response.statusText}`);
    }
    return response.json();
})
   .then((data) => console.log(data))
   .catch((error) => console.error("Fetch error:", error));
```

#### 3. Geolocation Timeout

```
// X No timeout handling
navigator.geolocation.getCurrentPosition(success, error);

// Set appropriate timeout
navigator.geolocation.getCurrentPosition(success, error, {
   timeout: 10000, // 10 seconds
   maximumAge: 60000, // Cache for 1 minute
   enableHighAccuracy: false, // Faster, less accurate
});
```

## 4. Canvas Memory Leaks

## 5. Web Worker Memory Management

```
// X Not terminating workers
const worker = new Worker("worker.js");
worker.postMessage(data);
// Worker keeps running
```

```
// Iterminate when done
worker.onmessage = (e) => {
  console.log(e.data);
  worker.terminate(); // Clean up
};
```

## 6. Service Worker Update Issues

#### 7. Observer Memory Leaks

# Mini Practice Problems

Problem 1: Offline-First Todo App

```
// Create a todo app that works offline using:
// - Service Worker for caching
// - IndexedDB for data storage
// - Background sync for uploading when online
// - Push notifications for reminders
class OfflineTodoApp {
 constructor() {
   // Your implementation here
   // Should handle:
   // - CRUD operations that work offline
   // - Sync with server when online
   // - Conflict resolution
   // - Push notifications for due dates
   // - Offline indicator
  }
 // Methods to implement:
 // init()
 // addTodo(text, dueDate)
 // updateTodo(id, updates)
 // deleteTodo(id)
 // syncWithServer()
 // handleConflicts(localTodos, serverTodos)
 // scheduleNotification(todo)
 // showOfflineIndicator()
 // registerServiceWorker()
}
```

#### Problem 2: Real-time Collaborative Canvas

```
// Create a collaborative drawing canvas using:
// - Canvas API for drawing
// - WebSockets for real-time sync
// - Web Workers for heavy computations
// - File API for import/export
// - History API for undo/redo
class CollaborativeCanvas {
  constructor(canvas, websocketUrl) {
   // Your implementation
    // Should handle:
   // - Multi-user drawing
    // - Real-time cursor positions
   // - Conflict-free drawing operations
    // - Export to various formats
    // - Undo/redo with collaborative history
  // Methods to implement:
```

```
// setupCanvas()
// setupWebSocket()
// setupWorkers()
// broadcastDrawing(operation)
// receiveDrawing(operation)
// exportCanvas(format)
// importImage(file)
// undo()
// redo()
// showCursors(users)
}
```

## Problem 3: Progressive Web App Dashboard

```
// Create a PWA dashboard with:
// - Service Worker for offline functionality
// - Web App Manifest for installation
// - Push notifications for alerts
// - Background sync for data updates
// - Geolocation for location-based features
// - Local storage for user preferences
class PWADashboard {
  constructor() {
    // Your implementation
    // Should include:
    // - Installable PWA
    // - Offline data visualization
    // - Real-time notifications
   // - Location-based widgets
   // - Customizable layout
   // - Data export/import
  }
 // Methods to implement:
  // registerServiceWorker()
 // setupPushNotifications()
 // handleInstallPrompt()
 // syncData()
  // updateLocation()
 // saveLayout()
  // exportData()
  // showOfflineMessage()
}
```

## Problem 4: Media Processing Workbench

```
// Create a media processing app using:
// - File API for file handling
```

```
// - Canvas API for image processing
// - Web Workers for heavy processing
// - Web Audio API for audio processing
// - Intersection Observer for lazy loading
// - Drag and Drop API
class MediaWorkbench {
 constructor() {
   // Your implementation
   // Should support:
   // - Image filters and effects
   // - Audio visualization
   // - Batch processing
   // - Progress tracking
   // - Preview generation
   // - Format conversion
  }
 // Methods to implement:
 // setupDropZone()
 // processImage(file, filters)
 // processAudio(file, effects)
 // generateThumbnails(files)
 // showProgress(operation)
 // exportProcessed(format)
 // setupWorkerPool()
}
```

### **Problem 5: Smart Notification System**

```
// Create an intelligent notification system using:
// - Notification API for alerts
// - Service Worker for background notifications
// - Geolocation for location-based alerts
// - Intersection Observer for engagement tracking
// - Local Storage for user preferences
// - Machine learning for notification timing
class SmartNotificationSystem {
  constructor() {
    // Your implementation
    // Should include:
    // - Adaptive notification timing
    // - Location-based triggers
   // - User engagement analysis
   // - Notification grouping
   // - Do not disturb modes
    // - A/B testing for effectiveness
  // Methods to implement:
```

```
// analyzeUserBehavior()
// predictOptimalTime(notification)
// scheduleNotification(notification, timing)
// groupNotifications(notifications)
// trackEngagement(notification, action)
// respectDoNotDisturb()
// optimizeDelivery()
}
```

## Interview Notes

#### **Common Questions:**

#### Q: What's the difference between localStorage and sessionStorage?

- localStorage: Persists until manually cleared, shared across tabs
- **sessionStorage**: Persists only for the session, tab-specific
- Both: Synchronous, 5-10MB limit, string-only storage

#### Q: How do you handle CORS in fetch requests?

- CORS is handled by the server, not the client
- Use mode: 'cors' for cross-origin requests
- Server must send appropriate CORS headers
- For credentials: <a href="mailto:redentials">redentials</a>: <a href="mailto:redentials">'include</a> and server allows

#### Q: When would you use Web Workers?

- Heavy computations that block the main thread
- Image/video processing
- Data parsing and transformation
- Background data synchronization
- Cryptographic operations

#### O: What's the difference between Service Workers and Web Workers?

- Service Workers: Network proxy, caching, push notifications, background sync
- Web Workers: Parallel processing, doesn't intercept network requests
- Service Workers: Persist between sessions, control multiple pages
- Web Workers: Tied to the page that created them

#### Q: How do you implement offline functionality?

- Service Worker for caching strategies
- IndexedDB for offline data storage
- Background sync for data synchronization
- Cache API for resource caching
- Network detection for online/offline states

#### Q: What are the different caching strategies?

- Cache First: Check cache, fallback to network
- Network First: Try network, fallback to cache
- Cache Only: Only serve from cache
- **Network Only**: Only fetch from network
- Stale While Revalidate: Serve from cache, update in background

## Asked at Companies:

- Google: "Implement a PWA with offline functionality and push notifications"
- Facebook: "Create a real-time collaborative editor using various browser APIs"
- Amazon: "Build a file upload system with progress tracking and resumable uploads"
- Microsoft: "Design a notification system that respects user preferences and timing"
- **Netflix**: "Implement video streaming with offline download capabilities"

# **©** Key Takeaways

- 1. Always check API support Use feature detection before using APIs
- 2. Handle errors gracefully Network requests, storage limits, permissions
- 3. Respect user preferences Notifications, location, storage usage
- 4. Optimize for performance Use Web Workers for heavy tasks
- 5. Implement offline strategies Service Workers, caching, local storage
- 6. Clean up resources Terminate workers, disconnect observers
- 7. **Progressive enhancement** Apps should work without advanced APIs
- 8. **Security considerations** Validate data, handle permissions properly

Previous Chapter: ← Event Handling

Next Chapter: Testing & Debugging →

**Practice**: Build a complete PWA using multiple browser APIs and test offline functionality!

```
## Chapter 16: Testing & Debugging  ## 個 Table of Contents
- [Testing Fundamentals](#testing-fundamentals)
- [Unit Testing](#unit-testing)
- [Integration Testing](#integration-testing)
- [End-to-End Testing](#end-to-end-testing)
- [Test-Driven Development (TDD)](#test-driven-development-tdd)
- [Debugging Techniques](#debugging-techniques)
- [Browser DevTools](#browser-devtools)
- [Performance Testing](#performance-testing)
- [Common Pitfalls](#common-pitfalls)
- [Practice Problems](#practice-problems)
- [Interview Notes](#interview-notes)
```

```
### What is Testing?
Testing ensures your code works as expected and helps prevent bugs from reaching
production.
### Types of Testing
```javascript
// 1. Unit Testing - Test individual functions/components
function add(a, b) {
    return a + b;
}
// Test
expect(add(2, 3)).toBe(5);
// 2. Integration Testing - Test component interactions
class Calculator {
    constructor() {
        this.result = 0;
    }
    add(num) {
        this.result += num;
        return this;
    }
    getResult() {
        return this.result;
}
// Integration test
const calc = new Calculator();
expect(calc.add(5).add(3).getResult()).toBe(8);
// 3. End-to-End Testing - Test complete user workflows
// (Usually done with tools like Cypress, Playwright)
```

### **Testing Pyramid**



# **4** Unit Testing

**Basic Test Structure** 

```
// Using Jest syntax (most popular)
describe("Math utilities", () => {
  test("should add two numbers correctly", () => {
    // Arrange
    const a = 2;
    const b = 3;
    // Act
    const result = add(a, b);
    // Assert
    expect(result).toBe(5);
  });
  test("should handle negative numbers", () => {
    expect(add(-2, 3)).toBe(1);
    expect(add(-2, -3)).toBe(-5);
  });
});
```

## **Testing Async Code**

```
// Testing Promises
test("should fetch user data", async () => {
 const userData = await fetchUser(1);
  expect(userData.name).toBe("John Doe");
});
// Testing with async/await
test("should handle API errors", async () => {
  await expect(fetchUser(-1)).rejects.toThrow("User not found");
});
// Testing callbacks
test("should call callback with result", (done) => {
  processData("input", (result) => {
    expect(result).toBe("processed: input");
    done();
 });
});
```

## Mocking and Spies

```
// Mocking functions
const mockFetch = jest.fn();
global.fetch = mockFetch;

test("should call API with correct parameters", async () => {
```

```
mockFetch.mockResolvedValue({
   json: () => Promise.resolve({ id: 1, name: "John" }),
 });
 const user = await getUser(1);
 expect(mockFetch).toHaveBeenCalledWith("/api/users/1");
 expect(user.name).toBe("John");
});
// Spying on methods
test("should log user creation", () => {
 const consoleSpy = jest.spyOn(console, "log");
 createUser("John");
 expect(consoleSpy).toHaveBeenCalledWith("User created: John");
 consoleSpy.mockRestore();
});
// Mocking modules
jest.mock("./userService", () => ({
  getUser: jest.fn(() => Promise.resolve({ id: 1, name: "Mock User" })),
}));
```

## **Testing Classes**

```
class UserManager {
 constructor(apiService) {
   this.apiService = apiService;
   this.users = [];
 }
 async addUser(userData) {
   const user = await this.apiService.createUser(userData);
   this.users.push(user);
   return user;
 }
 getUserCount() {
   return this.users.length;
 }
}
describe("UserManager", () => {
 let userManager;
 let mockApiService;
 beforeEach(() => {
   mockApiService = {
```

## Testing DOM Manipulation

```
// Using jsdom (automatically set up in Jest)
test("should create button element", () => {
 document.body.innerHTML = '<div id="container"></div>';
  const button = createButton("Click me", () => console.log("clicked"));
 document.getElementById("container").appendChild(button);
 expect(button.tagName).toBe("BUTTON");
  expect(button.textContent).toBe("Click me");
 expect(document.querySelector("button")).toBeTruthy();
});
// Testing event handlers
test("should handle button click", () => {
  const mockHandler = jest.fn();
  const button = createButton("Test", mockHandler);
 button.click();
  expect(mockHandler).toHaveBeenCalled();
});
```

# Integration Testing

### **Testing Component Interactions**

```
// Testing multiple components working together
class TodoApp {
 constructor() {
   this.todos = [];
   this.storage = new LocalStorage();
   this.ui = new TodoUI();
 }
 addTodo(text) {
   const todo = { id: Date.now(), text, completed: false };
   this.todos.push(todo);
   this.storage.save("todos", this.todos);
   this.ui.render(this.todos);
   return todo;
 }
}
describe("TodoApp Integration", () => {
 let app;
 beforeEach(() => {
   // Set up DOM
   document.body.innerHTML = '<div id="todo-container"></div>';
   localStorage.clear();
   app = new TodoApp();
 });
 test("should add todo and persist to storage", () => {
    const todo = app.addTodo("Learn testing");
   // Check todo was added
    expect(app.todos).toContain(todo);
   // Check storage was updated
    const stored = JSON.parse(localStorage.getItem("todos"));
    expect(stored).toContain(todo);
   // Check UI was updated
   expect(document.querySelector(".todo-item")).toBeTruthy();
 });
});
```

### **API Integration Testing**

```
// Testing with real API calls (use sparingly)
describe("User API Integration", () => {
  test("should create and retrieve user", async () => {
    // Create user
    const userData = {
        name: "Test User",
```

```
email: `test${Date.now()}@example.com`,
    };
    const createdUser = await userAPI.create(userData);
    expect(createdUser.id).toBeDefined();
    expect(createdUser.name).toBe(userData.name);
    // Retrieve user
    const retrievedUser = await userAPI.get(createdUser.id);
    expect(retrievedUser).toEqual(createdUser);
   // Cleanup
   await userAPI.delete(createdUser.id);
 });
});
// Better: Mock the API layer
class UserService {
 constructor(apiClient) {
   this.apiClient = apiClient;
 }
 async createUser(userData) {
   const response = await this.apiClient.post("/users", userData);
    return response.data;
 }
}
test("UserService integration with API client", async () => {
  const mockApiClient = {
    post: jest.fn().mockResolvedValue({
      data: { id: 1, name: "John", email: "john@example.com" },
   }),
 };
 const userService = new UserService(mockApiClient);
 const result = await userService.createUser({ name: "John" });
 expect(mockApiClient.post).toHaveBeenCalledWith("/users", { name: "John" });
 expect(result.id).toBe(1);
});
```

## End-to-End Testing

# Cypress Example

```
// cypress/integration/todo-app.spec.js
describe("Todo App E2E", () => {
  beforeEach(() => {
    cy.visit("/todo-app");
}
```

```
});
 it("should add and complete a todo", () => {
   // Add todo
   cy.get('[data-testid="todo-input"]').type("Learn Cypress{enter}");
   // Verify todo appears
   cy.get('[data-testid="todo-item"]')
      .should("contain", "Learn Cypress")
      .should("not.have.class", "completed");
   // Complete todo
   cy.get('[data-testid="todo-checkbox"]').click();
   // Verify completion
   cy.get('[data-testid="todo-item"]').should("have.class", "completed");
 });
 it("should persist todos after page reload", () => {
   cy.get('[data-testid="todo-input"]').type("Persistent todo{enter}");
   cy.reload();
   cy.get('[data-testid="todo-item"]').should("contain", "Persistent todo");
 });
});
```

### Playwright Example

```
// tests/todo-app.spec.js
const { test, expect } = require("@playwright/test");
test.describe("Todo App", () => {
 test("should handle user workflow", async ({ page }) => {
    await page.goto("/todo-app");
   // Add multiple todos
    await page.fill('[data-testid="todo-input"]', "First todo");
    await page.press('[data-testid="todo-input"]', "Enter");
    await page.fill('[data-testid="todo-input"]', "Second todo");
    await page.press('[data-testid="todo-input"]', "Enter");
    // Check todos are visible
    await expect(page.locator('[data-testid="todo-item"]')).toHaveCount(2);
    // Complete first todo
    await page.click('[data-testid="todo-checkbox"]:first-child');
    // Filter completed todos
    await page.click('[data-testid="filter-completed"]');
```

```
await expect(page.locator('[data-testid="todo-item"]')).toHaveCount(1);
});
});
```

## Test-Driven Development (TDD)

TDD Cycle: Red-Green-Refactor

```
// 1. RED: Write a failing test
test("should calculate area of rectangle", () => {
  expect(calculateArea(5, 3)).toBe(15);
});
// 2. GREEN: Write minimal code to pass
function calculateArea(width, height) {
  return width * height;
}
// 3. REFACTOR: Improve the code
function calculateArea(width, height) {
  if (width <= 0 || height <= 0) {
    throw new Error("Dimensions must be positive");
  }
 return width * height;
}
// Add more tests
test("should throw error for negative dimensions", () => {
  expect(() => calculateArea(-5, 3)).toThrow("Dimensions must be positive");
  expect(() => calculateArea(5, -3)).toThrow("Dimensions must be positive");
});
```

#### TDD Example: Building a Shopping Cart

```
// Step 1: Write tests first
describe("ShoppingCart", () => {
  let cart;

beforeEach(() => {
    cart = new ShoppingCart();
  });

test("should start empty", () => {
    expect(cart.getItems()).toEqual([]);
    expect(cart.getTotal()).toBe(0);
  });

test("should add items", () => {
```

```
cart.addItem({ id: 1, name: "Apple", price: 1.5 });
    expect(cart.getItems()).toHaveLength(1);
    expect(cart.getTotal()).toBe(1.5);
 });
 test("should handle quantity", () => {
    cart.addItem({ id: 1, name: "Apple", price: 1.5 }, 3);
    expect(cart.getTotal()).toBe(4.5);
 });
});
// Step 2: Implement to pass tests
class ShoppingCart {
 constructor() {
   this.items = [];
 getItems() {
   return this.items;
 addItem(item, quantity = 1) {
   this.items.push({ ...item, quantity });
 }
 getTotal() {
    return this.items.reduce((total, item) => {
      return total + item.price * item.quantity;
    }, 0);
 }
}
```

# **Debugging Techniques**

## Console Debugging

```
// Basic logging
console.log("Variable value:", myVariable);
console.error("Error occurred:", error);
console.warn("Warning:", warning);

// Advanced console methods
console.table(arrayOfObjects); // Display as table
console.group("User Operations");
console.log("Creating user...");
console.log("Validating data...");
console.groupEnd();

// Conditional logging
console.assert(user.age >= 18, "User must be 18 or older");
```

```
// Performance timing
console.time("API Call");
await fetchUserData();
console.timeEnd("API Call");

// Stack trace
console.trace("Execution path");
```

## **Debugger Statement**

```
function processUserData(userData) {
   // Execution will pause here when DevTools is open
   debugger;

const processed = userData.map((user) => {
    debugger; // Pause for each iteration
    return {
        ...user,
        fullName: `${user.firstName} ${user.lastName}`,
     };
   });

return processed;
}
```

## **Error Handling for Debugging**

```
// Custom error with context
class ValidationError extends Error {
  constructor(message, field, value) {
    super(message);
    this.name = "ValidationError";
    this.field = field;
   this.value = value;
    this.timestamp = new Date().toISOString();
 }
}
function validateUser(user) {
  if (!user.email) {
    throw new ValidationError("Email is required", "email", user.email);
  }
}
// Global error handler
window.addEventListener("error", (event) => {
  console.error("Global error:", {
    message: event.message,
```

```
filename: event.filename,
    lineno: event.lineno,
    colno: event.colno,
    error: event.error,
    });
});

// Unhandled promise rejection handler
window.addEventListener("unhandledrejection", (event) => {
    console.error("Unhandled promise rejection:", event.reason);
    event.preventDefault(); // Prevent default browser behavior
});
```

## **Debugging Async Code**

```
// Debug async/await
async function fetchUserData(userId) {
 try {
    console.log("Fetching user:", userId);
    const response = await fetch(`/api/users/${userId}`);
    console.log("Response status:", response.status);
   if (!response.ok) {
     throw new Error(`HTTP ${response.status}`);
    const userData = await response.json();
    console.log("User data received:", userData);
   return userData;
  } catch (error) {
    console.error("Error fetching user:", error);
    throw error;
 }
}
// Debug promises
fetchUserData(123)
  .then((user) => {
    console.log("Success:", user);
   return user;
 })
  .catch((error) => {
   console.error("Failed:", error);
   return null;
 });
```

### Sources Panel Debugging

```
// Set breakpoints programmatically
function complexCalculation(data) {
  let result = 0;
  for (let i = 0; i < data.length; i++) {
    // Set conditional breakpoint: i === 5
    result += data[i] * 2;
    if (result > 100) {
        // Logpoint: console.log('Result exceeded 100:', result)
        break;
    }
}
return result;
}
```

### **Network Panel Debugging**

```
// Monitor network requests
class APIClient {
  async request(url, options = {}) {
    const startTime = performance.now();
    try {
      const response = await fetch(url, {
        ...options,
        headers: {
          "Content-Type": "application/json",
          ...options.headers,
       },
      });
      const endTime = performance.now();
      console.log(`Request to ${url} took ${endTime - startTime}ms`);
      return response;
    } catch (error) {
      console.error(`Request to ${url} failed:`, error);
      throw error;
 }
}
```

### Performance Panel Debugging

```
// Mark performance events
function expensiveOperation() {
 performance.mark("expensive-start");
 // Simulate heavy computation
 let result = 0;
 for (let i = 0; i < 1000000; i++) {
   result += Math.random();
 }
 performance.mark("expensive-end");
 performance.measure(
    "expensive-operation",
    "expensive-start",
    "expensive-end"
 );
 // Get measurements
 const measures = performance.getEntriesByType("measure");
 console.log("Performance measures:", measures);
 return result;
}
```

### Memory Debugging

```
// Detect memory leaks
class MemoryTracker {
 constructor() {
    this.objects = new Set();
  }
 track(obj) {
   this.objects.add(obj);
    return obj;
 untrack(obj) {
   this.objects.delete(obj);
  }
 getTrackedCount() {
    return this.objects.size;
  }
 logMemoryUsage() {
   if (performance.memory) {
      console.log("Memory usage:", {
        used: performance.memory.usedJSHeapSize,
        total: performance.memory.totalJSHeapSize,
```

## Performance Testing

### Benchmarking

```
// Simple benchmark
function benchmark(fn, iterations = 1000) {
  const start = performance.now();
  for (let i = 0; i < iterations; i++) {
   fn();
  }
 const end = performance.now();
 return end - start;
}
// Compare implementations
function compareImplementations() {
  const data = Array.from(\{ length: 1000 \}, (_, i) => i);
 const forLoopTime = benchmark(() => {
    let sum = 0;
    for (let i = 0; i < data.length; i++) {
      sum += data[i];
   return sum;
  });
  const reduceTime = benchmark(() => {
    return data.reduce((sum, num) => sum + num, 0);
  });
 console.log("For loop:", forLoopTime + "ms");
  console.log("Reduce:", reduceTime + "ms");
}
```

### **Load Testing**

```
// Simulate concurrent users
async function loadTest(url, concurrentUsers = 10, requestsPerUser = 100) {
 const results = [];
 const userPromises = Array.from(
   { length: concurrentUsers },
    async (_, userId) => {
      const userResults = [];
      for (let i = 0; i < requestsPerUser; i++) {
        const start = performance.now();
        try {
          const response = await fetch(url);
          const end = performance.now();
          userResults.push({
            userId,
            requestId: i,
            status: response.status,
            duration: end - start,
            success: response.ok,
          });
        } catch (error) {
          userResults.push({
            userId,
            requestId: i,
            error: error.message,
            success: false,
          });
        }
      }
      return userResults;
   }
  );
 const allResults = await Promise.all(userPromises);
 const flatResults = allResults.flat();
 // Analyze results
 const successRate =
   flatResults.filter((r) => r.success).length / flatResults.length;
 const avgDuration =
   flatResults
      .filter((r) => r.duration)
      .reduce((sum, r) => sum + r.duration, 0) / flatResults.length;
 console.log("Load test results:", {
    totalRequests: flatResults.length,
```

```
successRate: (successRate * 100).toFixed(2) + "%",
   averageDuration: avgDuration.toFixed(2) + "ms",
});

return flatResults;
}
```

### 

### 1. Testing Implementation Details

```
// X Testing internal implementation
test("should call internal method", () => {
  const spy = jest.spyOn(calculator, "_internalCalculation");
  calculator.add(2, 3);
  expect(spy).toHaveBeenCalled();
});

// Iest behavior, not implementation
test("should add numbers correctly", () => {
  expect(calculator.add(2, 3)).toBe(5);
});
```

### 2. Flaky Tests

```
// X Time-dependent test
test("should process within 100ms", async () => {
  const start = Date.now();
  await processData();
  const duration = Date.now() - start;
  expect(duration).toBeLessThan(100); // Flaky!
});

// Mock time or use proper async testing
test("should process data correctly", async () => {
  const result = await processData();
  expect(result).toEqual(expectedResult);
});
```

### 3. Not Cleaning Up

```
// X Not cleaning up after tests
test("should handle user login", () => {
  localStorage.setItem("user", JSON.stringify(user));
  // Test logic...
```

```
// localStorage still contains data!
});

// Clean up properly
test("should handle user login", () => {
    localStorage.setItem("user", JSON.stringify(user));
    // Test logic...

// Cleanup
    localStorage.clear();
});

// Better: Use beforeEach/afterEach
afterEach(() => {
    localStorage.clear();
    jest.clearAllMocks();
});
```

### 4. Over-mocking

```
// X Mocking everything
test("should process user data", () => {
  const mockUser = { name: "John" };
  const mockProcessor = jest.fn().mockReturnValue("processed");
  const mockValidator = jest.fn().mockReturnValue(true);

// Test becomes meaningless
});

// W Mock only external dependencies
test("should process user data", () => {
  const mockApiCall = jest.fn().mockResolvedValue(userData);
  // Test actual processing logic
});
```

### 5. Debugging in Production

```
// X Leaving debug code in production
function processPayment(amount) {
   console.log("Processing payment:", amount); // Remove this!
   debugger; // Remove this!

   // Payment logic
}

// Use environment-aware logging
function processPayment(amount) {
   if (process.env.NODE_ENV === "development") {
      console.log("Processing payment:", amount);
}
```

```
}

// Payment logic
}
```

## Mini Practice Problems

### Problem 1: Test a User Registration System

```
// Implement and test a user registration system
class UserRegistration {
  constructor(userService, emailService) {
    this.userService = userService;
    this.emailService = emailService;
  }
  async register(userData) {
   // Your implementation here
   // Should:
    // - Validate user data
   // - Check if user already exists
    // - Create user account
   // - Send welcome email
    // - Return user object
 }
}
// Write comprehensive tests for:
// - Valid registration
// - Invalid data handling
// - Duplicate user handling
// - Email service failures
// - Database errors
```

### Problem 2: Debug a Shopping Cart Bug

```
// This shopping cart has bugs - find and fix them
class ShoppingCart {
  constructor() {
    this.items = [];
  }

addItem(product, quantity) {
  const existingItem = this.items.find((item) => item.id === product.id);

if (existingItem) {
  existingItem.quantity += quantity;
  } else {
```

2025-07-22 javascript.md

```
this.items.push({ ...product, quantity });
    }
  }
  removeItem(productId) {
   this.items = this.items.filter((item) => item.id !== productId);
  }
  getTotal() {
    return this.items.reduce((total, item) => {
      return total + item.price * item.quantity;
    }, 0);
  applyDiscount(percentage) {
    this.items.forEach((item) => {
      item.price = item.price * (1 - percentage / 100);
    });
 }
}
// Issues to find:
// 1. What happens with negative quantities?
// 2. What if product price is not a number?
// 3. What if discount is applied multiple times?
// 4. What about floating point precision?
// 5. What if items array is modified during iteration?
```

### Problem 3: Performance Test a Data Processing Function

```
// Test and optimize this data processing function
function processLargeDataset(data) {
 const result = [];
 for (let i = 0; i < data.length; i++) {
   const item = data[i];
   if (item.active && item.score > 50) {
     const processed = {
        id: item.id,
        name: item.name.toUpperCase(),
        category: item.category,
       normalizedScore: item.score / 100,
       tags: item.tags.filter((tag) => tag.length > 3),
     };
     result.push(processed);
   }
 }
 return result.sort((a, b) => b.normalizedScore - a.normalizedScore);
```

```
// Tasks:
// 1. Write performance benchmarks
// 2. Identify bottlenecks
// 3. Optimize the function
// 4. Compare before/after performance
// 5. Test with different data sizes
```

### Problem 4: E2E Test a Todo Application

```
// Write comprehensive E2E tests for a todo app
// Features to test:
// - Add new todos
// - Mark todos as complete
// - Edit existing todos
// - Delete todos
// - Filter todos (all, active, completed)
// - Clear completed todos
// - Persist data across page reloads
// - Handle empty states
// - Validate input constraints

// Use Cypress or Playwright to implement:
describe("Todo Application E2E", () => {
    // Your test implementations here
});
```

### Problem 5: Debug Memory Leaks

```
// This code has memory leaks - find and fix them
class EventManager {
  constructor() {
    this.listeners = new Map();
    this.timers = [];
  }

  addEventListener(element, event, handler) {
    element.addEventListener(event, handler);

  if (!this.listeners.has(element)) {
      this.listeners.set(element, []);
    }

    this.listeners.get(element).push({ event, handler });
}

startPeriodicTask(callback, interval) {
    const timerId = setInterval(callback, interval);
```

```
this.timers.push(timerId);
    return timerId;
  }
  createDOMElements() {
    const container = document.createElement("div");
    for (let i = 0; i < 1000; i++) {
      const element = document.createElement("div");
      element.innerHTML = `Item ${i}`;
      this.addEventListener(element, "click", () => {
        console.log(`Clicked item ${i}`);
      });
      container.appendChild(element);
    }
    return container;
}
// Issues to identify:
// 1. Event listeners not being removed
// 2. Timers not being cleared
// 3. DOM references being held
// 4. Closure memory retention
// 5. Missing cleanup methods
```

## Interview Notes

#### Common Questions:

#### Q: What's the difference between unit, integration, and E2E tests?

- Unit: Test individual functions/components in isolation
- **Integration**: Test how components work together
- **E2E**: Test complete user workflows from start to finish
- Pyramid: Many unit tests, some integration tests, few E2E tests

### Q: How do you test async code?

- Use async/await in test functions
- Return promises from test functions
- Use done callback for callback-based code
- Mock async dependencies appropriately

#### Q: What are mocks, stubs, and spies?

- Mock: Replace entire object/function with fake implementation
- Stub: Replace specific method with predetermined behavior

• **Spy**: Monitor calls to existing function without replacing it

### Q: How do you debug JavaScript in production?

- Use proper logging (not console.log)
- Implement error tracking (Sentry, Bugsnag)
- Use source maps for meaningful stack traces
- Monitor performance metrics
- Implement feature flags for safe rollbacks

#### Q: What's Test-Driven Development (TDD)?

- Write failing test first (Red)
- Write minimal code to pass (Green)
- Refactor while keeping tests green (Refactor)
- Benefits: Better design, fewer bugs, confidence in changes

### Q: How do you handle flaky tests?

- Identify root cause (timing, dependencies, environment)
- Use proper async testing patterns
- Mock external dependencies
- Avoid time-dependent assertions
- Implement proper test isolation

### Asked at Companies:

- Google: "How would you test a complex React component with multiple async operations?"
- Facebook: "Debug this performance issue in a large dataset processing function"
- Amazon: "Design a testing strategy for a microservices architecture"
- Microsoft: "How do you ensure code quality in a large team?"
- Netflix: "Test a video streaming component with various network conditions"

## **©** Key Takeaways

- 1. Test behavior, not implementation Focus on what the code does, not how
- 2. Follow the testing pyramid Many unit tests, fewer integration/E2E tests
- 3. Write tests first TDD leads to better design and fewer bugs
- 4. Mock external dependencies Keep tests fast and reliable
- 5. Use proper debugging tools Browser DevTools, debugger statements, logging
- 6. Clean up after tests Prevent test pollution and flaky tests
- 7. Test edge cases Empty inputs, error conditions, boundary values
- 8. Monitor performance Use benchmarks and profiling tools

**Previous Chapter**: ← Browser APIs

**Next Chapter**: Performance Optimization →

**Practice**: Write comprehensive tests for a real project and use debugging tools to optimize performance!

# Chapter 17: Performance Optimization 4

### **Table of Contents**

- Performance Fundamentals
- Memory Management
- Code Optimization
- DOM Performance
- Network Optimization
- Async Performance
- Bundle Optimization
- Monitoring & Profiling
- Common Pitfalls
- Practice Problems
- Interview Notes

### **©** Performance Fundamentals

### **Understanding Performance Metrics**

```
// Core Web Vitals
function measureCoreWebVitals() {
 // Largest Contentful Paint (LCP)
 new PerformanceObserver((entryList) => {
   const entries = entryList.getEntries();
   const lastEntry = entries[entries.length - 1];
   console.log("LCP:", lastEntry.startTime);
 }).observe({ entryTypes: ["largest-contentful-paint"] });
 // First Input Delay (FID)
 new PerformanceObserver((entryList) => {
   const entries = entryList.getEntries();
   entries.forEach((entry) => {
     console.log("FID:", entry.processingStart - entry.startTime);
   });
 }).observe({ entryTypes: ["first-input"] });
 // Cumulative Layout Shift (CLS)
 let clsValue = ∅;
 new PerformanceObserver((entryList) => {
   const entries = entryList.getEntries();
   entries.forEach((entry) => {
     if (!entry.hadRecentInput) {
        clsValue += entry.value;
     }
   });
    console.log("CLS:", clsValue);
```

```
}).observe({ entryTypes: ["layout-shift"] });
}
```

### Performance Timing API

```
// Measure custom performance
class PerformanceTracker {
 static mark(name) {
    performance.mark(name);
  }
  static measure(name, startMark, endMark) {
    performance.measure(name, startMark, endMark);
    const measure = performance.getEntriesByName(name)[0];
    console.log(`${name}: ${measure.duration.toFixed(2)}ms`);
   return measure.duration;
 }
 static async timeFunction(fn, label) {
    const start = performance.now();
   const result = await fn();
   const end = performance.now();
   console.log(`${label}: ${(end - start).toFixed(2)}ms`);
   return result;
  }
  static getNavigationTiming() {
    const timing = performance.getEntriesByType("navigation")[0];
    return {
      dns: timing.domainLookupEnd - timing.domainLookupStart,
      tcp: timing.connectEnd - timing.connectStart,
      request: timing.responseStart - timing.requestStart,
      response: timing.responseEnd - timing.responseStart,
      domParsing: timing.domInteractive - timing.responseEnd,
      domReady:
        timing.domContentLoadedEventEnd - timing.domContentLoadedEventStart,
      loadComplete: timing.loadEventEnd - timing.loadEventStart,
    };
 }
}
// Usage
PerformanceTracker.mark("data-fetch-start");
await fetchData();
PerformanceTracker.mark("data-fetch-end");
PerformanceTracker.measure("data-fetch", "data-fetch-start", "data-fetch-end");
```

### **Understanding Memory Leaks**

```
// Common memory leak patterns
// 1. Global variables
// X Memory leak
function createUser() {
 // Accidentally creates global variable
 user = { name: "John", data: new Array(1000000) };
 return user;
}
// Proper scoping
function createUser() {
 const user = { name: "John", data: new Array(1000000) };
 return user;
}
// 2. Event listeners not removed
// ★ Memory leak
class Component {
 constructor() {
   this.handleClick = this.handleClick.bind(this);
    document.addEventListener("click", this.handleClick);
  }
 handleClick() {
    console.log("Clicked");
  }
 // Missing cleanup!
// Proper cleanup
class Component {
 constructor() {
   this.handleClick = this.handleClick.bind(this);
    document.addEventListener("click", this.handleClick);
  }
  handleClick() {
   console.log("Clicked");
  }
 destroy() {
    document.removeEventListener("click", this.handleClick);
}
// 3. Timers not cleared
// X Memory leak
class Timer {
```

```
start() {
   this.interval = setInterval(() => {
     console.log("Timer tick");
   }, 1000);
 // Missing cleanup!
// ✓ Proper cleanup
class Timer {
 start() {
   this.interval = setInterval(() => {
     console.log("Timer tick");
   }, 1000);
 stop() {
   if (this.interval) {
      clearInterval(this.interval);
     this.interval = null;
   }
 }
}
```

### **Memory Optimization Techniques**

```
// Object pooling
class ObjectPool {
  constructor(createFn, resetFn, initialSize = 10) {
    this.createFn = createFn;
   this.resetFn = resetFn;
   this.pool = [];
   // Pre-populate pool
   for (let i = 0; i < initialSize; i++) {</pre>
     this.pool.push(this.createFn());
  }
  acquire() {
    if (this.pool.length > 0) {
     return this.pool.pop();
    return this.createFn();
  }
  release(obj) {
   this.resetFn(obj);
    this.pool.push(obj);
```

```
size() {
    return this.pool.length;
  }
// Usage for expensive objects
const particlePool = new ObjectPool(
  () => ({ x: 0, y: 0, vx: 0, vy: 0, life: 1 }),
  (particle) => {
    particle.x = 0;
    particle.y = ∅;
    particle.vx = ∅;
    particle.vy = ∅;
    particle.life = 1;
  },
  100
);
// WeakMap for private data
const privateData = new WeakMap();
class User {
  constructor(name) {
   this.name = name;
    privateData.set(this, {
      id: Math.random(),
      secret: "private-data",
    });
  getPrivateData() {
    return privateData.get(this);
  }
}
// When User instance is garbage collected,
// its private data is automatically removed from WeakMap
```

### **Memory Monitoring**

```
// Memory usage monitoring
class MemoryMonitor {
   static getMemoryUsage() {
    if (performance.memory) {
       return {
       used: Math.round(performance.memory.usedJSHeapSize / 1024 / 1024),
       total: Math.round(performance.memory.totalJSHeapSize / 1024 / 1024),
       limit: Math.round(performance.memory.jsHeapSizeLimit / 1024 / 1024),
    };
}
```

2025-07-22 javascript.md

```
return null;
  }
  static startMonitoring(interval = 5000) {
    return setInterval(() => {
      const memory = this.getMemoryUsage();
      if (memory) {
        console.log(
          `Memory: ${memory.used}MB / ${memory.total}MB (limit:
${memory.limit}MB)`
        );
        // Alert if memory usage is high
        if (memory.used / memory.limit > 0.8) {
          console.warn("High memory usage detected!");
    }, interval);
  static measureFunction(fn, label) {
    const beforeMemory = this.getMemoryUsage();
    const start = performance.now();
    const result = fn();
    const end = performance.now();
    const afterMemory = this.getMemoryUsage();
    console.log(`${label}:`, {
      duration: `${(end - start).toFixed(2)}ms`,
      memoryDelta:
        beforeMemory && afterMemory
          ? `${afterMemory.used - beforeMemory.used}MB`
          : "N/A",
    });
    return result;
  }
}
// Usage
const monitorId = MemoryMonitor.startMonitoring();
// Later...
clearInterval(monitorId);
```



## Code Optimization

Algorithm Optimization

```
// Optimize loops
// X Inefficient
function processItems(items) {
  const results = [];
  for (let i = 0; i < items.length; i++) {
    if (items[i].active) {
      for (let j = 0; j < items[i].tags.length; j++) {</pre>
        if (items[i].tags[j].length > 3) {
          results.push({
            id: items[i].id,
            tag: items[i].tags[j].toUpperCase(),
          });
        }
      }
    }
  return results;
// // Optimized
function processItemsOptimized(items) {
  const results = [];
  const itemsLength = items.length;
  for (let i = 0; i < itemsLength; i++) {
    const item = items[i];
    if (!item.active) continue;
    const tags = item.tags;
    const tagsLength = tags.length;
    const itemId = item.id;
    for (let j = 0; j < tagsLength; j++) {
      const tag = tags[j];
      if (tag.length > 3) {
        results.push({
          id: itemId,
          tag: tag.toUpperCase(),
        });
      }
    }
  return results;
}
// Even better: Use functional approach with early returns
function processItemsFunctional(items) {
  return items
    .filter((item) => item.active)
    .flatMap((item) =>
      item.tags
        .filter((tag) => tag.length > 3)
```

```
.map((tag) => ({
    id: item.id,
        tag: tag.toUpperCase(),
    }))
   );
}
```

### **Data Structure Optimization**

```
// Use appropriate data structures
// X Array for lookups (O(n))
class UserManagerSlow {
  constructor() {
    this.users = [];
  }
  addUser(user) {
    this.users.push(user);
  }
  findUser(id) {
    return this.users.find((user) => user.id === id); // O(n)
  }
  removeUser(id) {
    const index = this.users.findIndex((user) => user.id === id); // O(n)
   if (index !== -1) {
      this.users.splice(index, 1); // O(n)
    }
  }
}
// \checkmark Map for lookups (0(1))
class UserManagerFast {
  constructor() {
    this.users = new Map();
  addUser(user) {
    this.users.set(user.id, user); // 0(1)
  }
  findUser(id) {
    return this.users.get(id); // 0(1)
  }
  removeUser(id) {
    return this.users.delete(id); // 0(1)
  }
```

```
getAllUsers() {
    return Array.from(this.users.values());
  }
}
// Set for unique values
class TagManager {
  constructor() {
    this.tags = new Set();
  }
  addTag(tag) {
   this.tags.add(tag.toLowerCase());
  }
  hasTag(tag) {
   return this.tags.has(tag.toLowerCase());
  getUniqueTagCount() {
    return this.tags.size;
  }
}
```

### **Function Optimization**

```
// 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;
 };
// Expensive function
const fibonacci = memoize(function (n) {
  if (n < 2) return n;
 return fibonacci(n - 1) + fibonacci(n - 2);
});
// Debouncing
function debounce(func, wait) {
```

```
let timeout;
  return function executedFunction(...args) {
    const later = () => {
      clearTimeout(timeout);
      func(...args);
    };
    clearTimeout(timeout);
    timeout = setTimeout(later, wait);
  };
}
// Throttling
function throttle(func, limit) {
  let inThrottle;
  return function (...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
  };
}
// Usage
const debouncedSearch = debounce(searchFunction, 300);
const throttledScroll = throttle(scrollHandler, 100);
```

## © DOM Performance

### **Efficient DOM Manipulation**

```
// X Inefficient DOM updates
function updateListSlow(items) {
  const list = document.getElementById("list");
  list.innerHTML = ""; // Triggers reflow

  items.forEach((item) => {
    const li = document.createElement("li");
    li.textContent = item.name;
    list.appendChild(li); // Triggers reflow for each item
  });
}

// M Batch DOM updates
function updateListFast(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); // No reflow
});

list.innerHTML = "";
    list.appendChild(fragment); // Single reflow
}

// Even better: Use template strings
function updateListTemplate(items) {
    const list = document.getElementById("list");
    const html = items.map((item) => `${item.name}).join("");
    list.innerHTML = html; // Single reflow
}
```

### Virtual Scrolling

```
// Virtual scrolling for large lists
class VirtualList {
 constructor(container, items, itemHeight = 50) {
   this.container = container;
   this.items = items;
   this.itemHeight = itemHeight;
    this.visibleCount = Math.ceil(container.clientHeight / itemHeight) + 2;
   this.startIndex = 0;
   this.setupContainer();
   this.render();
   this.bindEvents();
 }
  setupContainer() {
    this.container.style.overflow = "auto";
    this.container.style.position = "relative";
   // Create scrollable area
    this.scrollArea = document.createElement("div");
    this.scrollArea.style.height = `${this.items.length * this.itemHeight}px`;
    this.container.appendChild(this.scrollArea);
    // Create visible items container
    this.itemsContainer = document.createElement("div");
    this.itemsContainer.style.position = "absolute";
   this.itemsContainer.style.top = "0";
   this.itemsContainer.style.width = "100%";
    this.scrollArea.appendChild(this.itemsContainer);
  }
```

```
render() {
    const endIndex = Math.min(
      this.startIndex + this.visibleCount,
      this.items.length
    const visibleItems = this.items.slice(this.startIndex, endIndex);
    this.itemsContainer.innerHTML = "";
    this.itemsContainer.style.transform = `translateY(${
      this.startIndex * this.itemHeight
    }px)`;
    visibleItems.forEach((item, index) => {
      const element = document.createElement("div");
      element.style.height = `${this.itemHeight}px`;
      element.textContent = item.name;
      element.dataset.index = this.startIndex + index;
      this.itemsContainer.appendChild(element);
    });
  }
  bindEvents() {
    this.container.addEventListener("scroll", () => {
      const newStartIndex = Math.floor(
        this.container.scrollTop / this.itemHeight
      );
      if (newStartIndex !== this.startIndex) {
        this.startIndex = newStartIndex;
        this.render();
    });
  }
}
// Usage
const items = Array.from(\{ length: 10000 \}, (_, i) \Rightarrow (\{ name: `Item $\{i\}` \}));
const virtualList = new VirtualList(
  document.getElementById("list-container"),
  items
);
```

### **Event Delegation**

```
// X Multiple event listeners
function attachEventsSlow() {
  const buttons = document.querySelectorAll(".button");
  buttons.forEach((button) => {
    button.addEventListener("click", handleButtonClick);
  });
}
```

```
// Event delegation
function attachEventsFast() {
 document.addEventListener("click", (event) => {
    if (event.target.matches(".button")) {
      handleButtonClick(event);
 });
}
// Advanced event delegation
class EventDelegator {
 constructor(container) {
   this.container = container;
   this.handlers = new Map();
   this.bindEvents();
 }
 on(selector, eventType, handler) {
   const key = `${eventType}:${selector}`;
   if (!this.handlers.has(key)) {
     this.handlers.set(key, []);
    }
   this.handlers.get(key).push(handler);
  }
 off(selector, eventType, handler) {
    const key = `${eventType}:${selector}`;
    const handlers = this.handlers.get(key);
    if (handlers) {
      const index = handlers.indexOf(handler);
     if (index > -1) {
        handlers.splice(index, 1);
      }
   }
  }
 bindEvents() {
   this.container.addEventListener("click", this.handleEvent.bind(this));
   this.container.addEventListener("change", this.handleEvent.bind(this));
   this.container.addEventListener("input", this.handleEvent.bind(this));
  }
 handleEvent(event) {
    const key = `${event.type}:`;
    for (const [handlerKey, handlers] of this.handlers) {
      if (handlerKey.startsWith(key)) {
        const selector = handlerKey.substring(key.length);
        if (event.target.matches(selector)) {
          handlers.forEach((handler) => handler(event));
        }
      }
```

```
}
}

// Usage
const delegator = new EventDelegator(document.body);
delegator.on(".button", "click", handleButtonClick);
delegator.on(".input", "input", handleInputChange);
```

## Metwork Optimization

### Request Optimization

```
// Request batching
class RequestBatcher {
 constructor(batchSize = 10, delay = 100) {
   this.batchSize = batchSize;
   this.delay = delay;
   this.queue = [];
   this.timeoutId = null;
 }
 add(request) {
   return new Promise((resolve, reject) => {
     this.queue.push({ request, resolve, reject });
     if (this.queue.length >= this.batchSize) {
       this.flush();
     } else if (!this.timeoutId) {
        this.timeoutId = setTimeout(() => this.flush(), this.delay);
   });
 }
 async flush() {
   if (this.queue.length === 0) return;
   const batch = this.queue.splice(0, this.batchSize);
   if (this.timeoutId) {
     clearTimeout(this.timeoutId);
     this.timeoutId = null;
   }
   try {
      const requests = batch.map((item) => item.request);
      const results = await this.processBatch(requests);
      batch.forEach((item, index) => {
        item.resolve(results[index]);
      });
```

```
} catch (error) {
      batch.forEach((item) => item.reject(error));
    }
  }
  async processBatch(requests) {
    // Send all requests in a single API call
    const response = await fetch("/api/batch", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ requests }),
    });
   return response.json();
 }
}
// Usage
const batcher = new RequestBatcher();
// These will be batched together
const user1 = await batcher.add({ type: "getUser", id: 1 });
const user2 = await batcher.add({ type: "getUser", id: 2 });
const user3 = await batcher.add({ type: "getUser", id: 3 });
```

### **Caching Strategies**

```
// HTTP cache with TTL
class HTTPCache {
 constructor(defaultTTL = 300000) {
   // 5 minutes
   this.cache = new Map();
   this.defaultTTL = defaultTTL;
 }
 set(key, value, ttl = this.defaultTTL) {
   const expiry = Date.now() + ttl;
   this.cache.set(key, { value, expiry });
 }
 get(key) {
   const item = this.cache.get(key);
   if (!item) return null;
   if (Date.now() > item.expiry) {
     this.cache.delete(key);
     return null;
   }
   return item.value;
```

```
clear() {
   this.cache.clear();
 cleanup() {
    const now = Date.now();
   for (const [key, item] of this.cache) {
     if (now > item.expiry) {
       this.cache.delete(key);
      }
   }
 }
}
// API client with caching
class CachedAPIClient {
 constructor() {
   this.cache = new HTTPCache();
   this.pendingRequests = new Map();
 }
 async get(url, options = {}) {
    const cacheKey = this.getCacheKey(url, options);
   // Check cache first
    const cached = this.cache.get(cacheKey);
    if (cached) {
      return cached;
    // Check if request is already pending
    if (this.pendingRequests.has(cacheKey)) {
      return this.pendingRequests.get(cacheKey);
    }
    // Make request
    const requestPromise = this.makeRequest(url, options)
      .then((response) => {
        this.cache.set(cacheKey, response, options.ttl);
        this.pendingRequests.delete(cacheKey);
        return response;
      })
      .catch((error) => {
        this.pendingRequests.delete(cacheKey);
       throw error;
      });
   this.pendingRequests.set(cacheKey, requestPromise);
    return requestPromise;
  }
  async makeRequest(url, options) {
```

```
const response = await fetch(url, options);
if (!response.ok) {
    throw new Error(`HTTP ${response.status}`);
}
return response.json();
}

getCacheKey(url, options) {
    return `${url}:${JSON.stringify(options)}`;
}
```

### Resource Loading

```
// Lazy loading with Intersection Observer
class LazyLoader {
 constructor(options = {}) {
   this.options = {
      rootMargin: "50px",
     threshold: 0.1,
      ...options,
   };
   this.observer = new IntersectionObserver(
     this.handleIntersection.bind(this),
      this.options
   );
  }
 observe(element) {
   this.observer.observe(element);
  }
 unobserve(element) {
   this.observer.unobserve(element);
  }
 handleIntersection(entries) {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
       this.loadElement(entry.target);
        this.observer.unobserve(entry.target);
      }
   });
 loadElement(element) {
   if (element.dataset.src) {
      element.src = element.dataset.src;
      element.removeAttribute("data-src");
    }
```

```
if (element.dataset.srcset) {
      element.srcset = element.dataset.srcset;
      element.removeAttribute("data-srcset");
   element.classList.add("loaded");
 }
}
// Preloading critical resources
class ResourcePreloader {
  static preloadImage(src) {
    return new Promise((resolve, reject) => {
      const img = new Image();
      img.onload = () => resolve(img);
      img.onerror = reject;
      img.src = src;
   });
  }
 static preloadScript(src) {
    return new Promise((resolve, reject) => {
      const script = document.createElement("script");
      script.onload = resolve;
      script.onerror = reject;
      script.src = src;
      document.head.appendChild(script);
   });
  }
 static preloadCSS(href) {
    return new Promise((resolve, reject) => {
      const link = document.createElement("link");
      link.rel = "stylesheet";
      link.onload = resolve;
      link.onerror = reject;
      link.href = href;
      document.head.appendChild(link);
   });
  }
 static async preloadCriticalResources(resources) {
    const promises = resources.map((resource) => {
      switch (resource.type) {
        case "image":
          return this.preloadImage(resource.src);
        case "script":
          return this.preloadScript(resource.src);
        case "css":
          return this.preloadCSS(resource.href);
        default:
          return Promise.resolve();
```

```
});

return Promise.all(promises);
}

// Usage
const lazyLoader = new LazyLoader();
document.querySelectorAll("img[data-src]").forEach((img) => {
    lazyLoader.observe(img);
});

// Preload critical resources
ResourcePreloader.preloadCriticalResources([
    { type: "image", src: "/hero-image.jpg" },
    { type: "script", src: "/critical.js" },
    { type: "css", href: "/critical.css" },
]);
```

## ← Async Performance

### Parallel vs Sequential Execution

```
// X Sequential execution (slow)
async function fetchUserDataSequential(userIds) {
 const users = [];
 for (const id of userIds) {
   const user = await fetchUser(id); // Waits for each request
   users.push(user);
  }
 return users;
}
// Parallel execution (fast)
async function fetchUserDataParallel(userIds) {
 const promises = userIds.map((id) => fetchUser(id));
 return Promise.all(promises);
}
// Controlled concurrency
async function fetchUserDataConcurrent(userIds, concurrency = 3) {
 const results = [];
 for (let i = 0; i < userIds.length; i += concurrency) {</pre>
    const batch = userIds.slice(i, i + concurrency);
    const batchPromises = batch.map((id) => fetchUser(id));
    const batchResults = await Promise.all(batchPromises);
    results.push(...batchResults);
```

```
return results;
}
// Advanced: Promise pool with retry
class PromisePool {
  constructor(concurrency = 3) {
    this.concurrency = concurrency;
   this.running = 0;
   this.queue = [];
  }
  async add(promiseFactory, retries = 3) {
    return new Promise((resolve, reject) => {
      this.queue.push({
        promiseFactory,
        resolve,
        reject,
        retries,
      });
      this.process();
    });
  }
  async process() {
    if (this.running >= this.concurrency || this.queue.length === 0) {
      return;
    }
    this.running++;
    const { promiseFactory, resolve, reject, retries } = this.queue.shift();
    try {
      const result = await promiseFactory();
      resolve(result);
    } catch (error) {
      if (retries > 0) {
        // Retry with exponential backoff
        setTimeout(() => {
          this.queue.unshift({
            promiseFactory,
            resolve,
            reject,
            retries: retries - 1,
          });
          this.process();
        }, Math.pow(2, 3 - retries) * 1000);
      } else {
        reject(error);
      }
    } finally {
      this.running--;
```

```
this.process();
}
}
// Usage
const pool = new PromisePool(5);
const results = await Promise.all(
  userIds.map((id) => pool.add(() => fetchUser(id)))
);
```

### Web Workers for Heavy Tasks

```
// Main thread
class WorkerPool {
  constructor(workerScript, poolSize = navigator.hardwareConcurrency | 4) {
   this.workers = [];
   this.queue = [];
   this.taskId = 0;
   for (let i = 0; i < poolSize; i++) {
     this.workers.push({
       worker: new Worker(workerScript),
        busy: false,
      });
   }
  }
 execute(data) {
    return new Promise((resolve, reject) => {
      const taskId = ++this.taskId;
     this.queue.push({
       taskId,
        data,
        resolve,
        reject,
      });
      this.processQueue();
   });
 processQueue() {
    if (this.queue.length === 0) return;
    const availableWorker = this.workers.find((w) => !w.busy);
    if (!availableWorker) return;
    const task = this.queue.shift();
    availableWorker.busy = true;
```

```
const handleMessage = (event) => {
      if (event.data.taskId === task.taskId) {
        availableWorker.worker.removeEventListener("message", handleMessage);
        availableWorker.worker.removeEventListener("error", handleError);
        availableWorker.busy = false;
        if (event.data.error) {
         task.reject(new Error(event.data.error));
        } else {
         task.resolve(event.data.result);
        }
       this.processQueue();
      }
    };
    const handleError = (error) => {
      availableWorker.worker.removeEventListener("message", handleMessage);
      availableWorker.worker.removeEventListener("error", handleError);
      availableWorker.busy = false;
     task.reject(error);
      this.processQueue();
   };
    availableWorker.worker.addEventListener("message", handleMessage);
    availableWorker.worker.addEventListener("error", handleError);
    availableWorker.worker.postMessage({
     taskId: task.taskId,
      data: task.data,
   });
 terminate() {
    this.workers.forEach((w) => w.worker.terminate());
 }
}
// worker.js
self.addEventListener("message", (event) => {
  const { taskId, data } = event.data;
 try {
   // Heavy computation
    const result = processLargeDataset(data);
    self.postMessage({
     taskId,
     result,
   });
  } catch (error) {
    self.postMessage({
      taskId,
```

2025-07-22 javascript.md

```
error: error.message,
    });
  }
});
function processLargeDataset(data) {
  // CPU-intensive task
  return data.map((item) => {
    // Complex calculations
    return Math.sqrt(item * item + item);
  });
}
// Usage
const workerPool = new WorkerPool("worker.js", 4);
const result = await workerPool.execute(largeDataArray);
```

### Bundle Optimization

### **Code Splitting**

```
// Dynamic imports for code splitting
class ModuleLoader {
 constructor() {
    this.loadedModules = new Map();
  }
 async loadModule(moduleName) {
    if (this.loadedModules.has(moduleName)) {
      return this.loadedModules.get(moduleName);
    }
    let modulePromise;
    switch (moduleName) {
      case "chart":
        modulePromise = import("./modules/chart.js");
        break;
      case "editor":
        modulePromise = import("./modules/editor.js");
        break;
      case "calendar":
        modulePromise = import("./modules/calendar.js");
        break;
      default:
        throw new Error(`Unknown module: ${moduleName}`);
    }
    this.loadedModules.set(moduleName, modulePromise);
    return modulePromise;
```

```
async loadModuleOnDemand(moduleName, trigger) {
    const loadModule = async () => {
      const module = await this.loadModule(moduleName);
      return module.default || module;
   };
   // Load on user interaction
   if (trigger) {
     trigger.addEventListener("click", loadModule, { once: true });
    }
   return loadModule;
 }
}
// Route-based code splitting
class Router {
 constructor() {
   this.routes = new Map();
   this.moduleLoader = new ModuleLoader();
 }
 addRoute(path, moduleFactory) {
   this.routes.set(path, moduleFactory);
 }
 async navigate(path) {
    const moduleFactory = this.routes.get(path);
   if (!moduleFactory) {
     throw new Error(`Route not found: ${path}`);
    }
   // Show loading indicator
   this.showLoading();
   try {
      const module = await moduleFactory();
      const component = new module.default();
     this.render(component);
    } catch (error) {
     this.showError(error);
    } finally {
     this.hideLoading();
   }
  }
  showLoading() {
    document.getElementById("loading").style.display = "block";
  }
 hideLoading() {
```

```
document.getElementById("loading").style.display = "none";
  }
 render(component) {
    const container = document.getElementById("app");
    container.innerHTML = "";
    container.appendChild(component.render());
  }
  showError(error) {
  console.error("Route loading error:", error);
 }
}
// Usage
const router = new Router();
router.addRoute("/dashboard", () => import("./pages/Dashboard.js"));
router.addRoute("/profile", () => import("./pages/Profile.js"));
router.addRoute("/settings", () => import("./pages/Settings.js"));
```

### Tree Shaking Optimization

```
// X Imports entire library
import * as utils from "./utils.js";
utils.debounce(fn, 300);
// Import only what you need
import { debounce } from "./utils.js";
debounce(fn, 300);
// Create tree-shakable modules
// utils.js
export function debounce(func, wait) {
 // Implementation
}
export function throttle(func, limit) {
 // Implementation
export function memoize(func) {
 // Implementation
}
// Don't export default objects
// X Not tree-shakable
export default {
  debounce,
  throttle,
  memoize,
```

```
};

// Iree-shakable
export { debounce, throttle, memoize };
```

### **Ⅲ** Monitoring & Profiling

### Performance Monitoring

```
// Comprehensive performance monitor
class PerformanceMonitor {
 constructor() {
   this.metrics = new Map();
   this.observers = [];
   this.setupObservers();
 }
  setupObservers() {
   // Long tasks
    if ("PerformanceObserver" in window) {
      const longTaskObserver = new PerformanceObserver((list) => {
        list.getEntries().forEach((entry) => {
          this.recordMetric("longTask", {
            duration: entry.duration,
            startTime: entry.startTime,
          });
       });
      });
      try {
        longTaskObserver.observe({ entryTypes: ["longtask"] });
       this.observers.push(longTaskObserver);
      } catch (e) {
        console.warn("Long task observer not supported");
      }
    }
    // Layout shifts
    const layoutShiftObserver = new PerformanceObserver((list) => {
      list.getEntries().forEach((entry) => {
        if (!entry.hadRecentInput) {
          this.recordMetric("layoutShift", {
            value: entry.value,
            startTime: entry.startTime,
          });
        }
      });
    });
    layoutShiftObserver.observe({ entryTypes: ["layout-shift"] });
```

```
this.observers.push(layoutShiftObserver);
}
recordMetric(name, data) {
  if (!this.metrics.has(name)) {
   this.metrics.set(name, []);
  this.metrics.get(name).push({
    ...data,
   timestamp: Date.now(),
  });
}
getMetrics(name) {
  return this.metrics.get(name) | [];
}
getAverageMetric(name, property) {
 const metrics = this.getMetrics(name);
 if (metrics.length === 0) return 0;
  const sum = metrics.reduce((acc, metric) => acc + metric[property], 0);
 return sum / metrics.length;
}
generateReport() {
 const report = {
    timestamp: new Date().toISOString(),
    navigation: this.getNavigationTiming(),
    longTasks: this.getMetrics("longTask"),
    layoutShifts: this.getMetrics("layoutShift"),
    memory: this.getMemoryInfo(),
    averages: {
      longTaskDuration: this.getAverageMetric("longTask", "duration"),
      layoutShiftValue: this.getAverageMetric("layoutShift", "value"),
    },
  };
  return report;
}
getNavigationTiming() {
  const timing = performance.getEntriesByType("navigation")[0];
 if (!timing) return null;
  return {
    dns: timing.domainLookupEnd - timing.domainLookupStart,
    tcp: timing.connectEnd - timing.connectStart,
    request: timing.responseStart - timing.requestStart,
    response: timing.responseEnd - timing.responseStart,
    domParsing: timing.domInteractive - timing.responseEnd,
    domReady:
      timing.domContentLoadedEventEnd - timing.domContentLoadedEventStart,
```

```
loadComplete: timing.loadEventEnd - timing.loadEventStart,
    };
  }
  getMemoryInfo() {
    if (performance.memory) {
      return {
        used: Math.round(performance.memory.usedJSHeapSize / 1024 / 1024),
        total: Math.round(performance.memory.totalJSHeapSize / 1024 / 1024),
        limit: Math.round(performance.memory.jsHeapSizeLimit / 1024 / 1024),
     };
    }
   return null;
  }
  sendReport(endpoint) {
    const report = this.generateReport();
    // Use sendBeacon for reliability
    if (navigator.sendBeacon) {
      navigator.sendBeacon(endpoint, JSON.stringify(report));
    } else {
      fetch(endpoint, {
        method: "POST",
        body: JSON.stringify(report),
        headers: { "Content-Type": "application/json" },
        keepalive: true,
      }).catch(console.error);
    }
  }
  disconnect() {
    this.observers.forEach((observer) => observer.disconnect());
  }
}
// Usage
const monitor = new PerformanceMonitor();
// Send report every 30 seconds
setInterval(() => {
  monitor.sendReport("/api/performance");
}, 30000);
// Send report before page unload
window.addEventListener("beforeunload", () => {
  monitor.sendReport("/api/performance");
});
```

### 

#### 1. Premature Optimization

```
// X Optimizing before measuring
function processData(data) {
    // Complex optimization that may not be needed
    const cache = new Map();
    const pool = new ObjectPool();
    // ... overly complex code
}

// Measure first, then optimize
function processData(data) {
    // Simple, readable implementation first
    return data.map((item) => transform(item));
}

// Profile and optimize only if needed
```

### 2. Memory Leaks in Event Listeners

```
// X Memory leak
class Component {
 constructor() {
   this.handleClick = this.handleClick.bind(this);
   document.addEventListener("click", this.handleClick);
 }
 handleClick() {
   console.log("Clicked");
 }
 // Missing cleanup!
}
// ✓ Proper cleanup
class Component {
 constructor() {
   this.handleClick = this.handleClick.bind(this);
    document.addEventListener("click", this.handleClick);
 }
 handleClick() {
   console.log("Clicked");
 }
 destroy() {
    document.removeEventListener("click", this.handleClick);
 }
}
```

### 3. Blocking the Main Thread

```
// X Blocking operation
function processLargeArray(data) {
 for (let i = 0; i < data.length; i++) {
    // Heavy computation blocks UI
    heavyCalculation(data[i]);
 }
}
// ✓ Non-blocking with time slicing
function processLargeArrayNonBlocking(data, callback) {
  let index = 0;
  const batchSize = 1000;
  function processBatch() {
    const endIndex = Math.min(index + batchSize, data.length);
    for (let i = index; i < endIndex; i++) {</pre>
      heavyCalculation(data[i]);
    }
    index = endIndex;
    if (index < data.length) {</pre>
      setTimeout(processBatch, ∅); // Yield to browser
    } else {
      callback();
  }
  processBatch();
}
```

### 4. Inefficient DOM Queries

```
elements.forEach((element) => {
    element.textContent = "Updated";
    element.classList.add("active");
    });
}
```

### 5. Not Using RequestAnimationFrame

```
// X Using setTimeout for animations
function animateElement(element) {
 let position = 0;
 const animate = () => {
    position += 1;
    element.style.left = position + "px";
   if (position < 100) {
      setTimeout(animate, 16); // Not synced with display
    }
 };
  animate();
}
// ✓ Using requestAnimationFrame
function animateElementOptimized(element) {
 let position = 0;
  const animate = () => {
    position += 1;
    element.style.left = position + "px";
   if (position < 100) {
      requestAnimationFrame(animate); // Synced with display
    }
 };
 requestAnimationFrame(animate);
```

### Mini Practice Problems

### Problem 1: Optimize a Data Processing Pipeline

```
// Optimize this slow data processing function
function processUserData(users) {
  const results = [];

for (let i = 0; i < users.length; i++) {
   const user = users[i];

  if (user.active) {
    const profile = getUserProfile(user.id); // Sync API call</pre>
```

```
const permissions = getUserPermissions(user.id); // Sync API call
      const preferences = getUserPreferences(user.id); // Sync API call
      const processed = {
        id: user.id,
        name: user.name.toUpperCase(),
        email: user.email.toLowerCase(),
        profile: profile,
        permissions: permissions,
        preferences: preferences,
        score: calculateUserScore(user, profile, permissions),
      };
      results.push(processed);
    }
 return results.sort((a, b) => b.score - a.score);
// Tasks:
// 1. Make API calls parallel
// 2. Implement caching
// 3. Add batch processing
// 4. Use Web Workers for heavy calculations
// 5. Implement progress tracking
```

### Problem 2: Create a High-Performance Virtual List

```
// Implement a virtual list that can handle 100,000+ items smoothly
class HighPerformanceVirtualList {
  constructor(container, items, itemHeight, renderItem) {
   // Your implementation here
   // Requirements:
   // - Smooth scrolling with 100k+ items
   // - Dynamic item heights support
   // - Horizontal scrolling support
   // - Search and filtering
   // - Memory efficient
   // - Keyboard navigation
  }
 // Methods to implement:
 // render()
 // scrollToIndex(index)
 // updateItems(newItems)
 // filter(predicate)
 // search(query)
 // destroy()
}
```

#### Problem 3: Build a Performance Budget Monitor

```
// Create a system that monitors and enforces performance budgets
class PerformanceBudgetMonitor {
 constructor(budgets) {
   // budgets = {
   // bundleSize: 250000, // 250KB
        firstContentfulPaint: 1500, // 1.5s
   //
         largestContentfulPaint: 2500, // 2.5s
   //
   //
         cumulativeLayoutShift: 0.1,
   //
          firstInputDelay: 100 // 100ms
   // }
 // Methods to implement:
 // startMonitoring()
 // checkBudgets()
 // reportViolations()
 // generateReport()
 // setupAlerts()
}
```

### Problem 4: Optimize Image Loading

```
// Create an advanced image loading system
class ImageOptimizer {
 constructor(options) {
   // Features to implement:
   // - Lazy loading with intersection observer
   // - Progressive image loading (blur-up)
   // - WebP/AVIF format detection and serving
   // - Responsive images based on device
   // - Image compression on the fly
   // - Preloading critical images
   // - Error handling and fallbacks
 }
 // Methods to implement:
 // loadImage(src, options)
 // preloadImages(srcs)
 // generateResponsiveSrc(src, sizes)
 // compressImage(file, quality)
 // convertToWebP(file)
}
```

### Problem 5: Memory Leak Detector

```
// Build a tool to detect and report memory leaks
class MemoryLeakDetector {
 constructor() {
   // Features:
   // - Track object creation and destruction
   // - Monitor DOM node leaks
   // - Detect event listener leaks
   // - Monitor closure leaks
   // - Generate leak reports
   // - Suggest fixes
 }
 // Methods to implement:
 // startTracking()
 // trackObject(obj, label)
 // detectLeaks()
 // generateReport()
 // suggestFixes()
 // cleanup()
```

### Interview Notes

### Common Questions:

#### Q: What are the main performance bottlenecks in web applications?

- **Network**: Large bundles, too many requests, no caching
- Rendering: Layout thrashing, unnecessary repaints, large DOM
- JavaScript: Blocking main thread, memory leaks, inefficient algorithms
- Images: Large file sizes, no optimization, blocking loading

### Q: How do you measure web performance?

- Core Web Vitals: LCP, FID, CLS
- Performance API: Navigation timing, resource timing
- Tools: Lighthouse, WebPageTest, Chrome DevTools
- Real User Monitoring: Track actual user experience

#### Q: What's the difference between throttling and debouncing?

- Throttling: Limits function execution to once per time period
- Debouncing: Delays function execution until after a quiet period
- Use cases: Throttle for scroll events, debounce for search input

#### Q: How do you optimize JavaScript bundle size?

- Tree shaking: Remove unused code
- Code splitting: Load code on demand
- Minification: Compress code

• Compression: Gzip/Brotli

• Dynamic imports: Lazy load modules

#### Q: What causes memory leaks in JavaScript?

• Global variables: Unintentional global scope pollution

• Event listeners: Not removed when elements are destroyed

• Timers: setInterval/setTimeout not cleared

• Closures: Retaining references to large objects

• DOM references: Keeping references to removed DOM nodes

#### Q: How do you optimize DOM manipulation?

• Batch updates: Use DocumentFragment

• Minimize reflows: Change styles in batches

• Use CSS transforms: For animations instead of changing layout properties

• Virtual DOM: For complex UIs

• Event delegation: Instead of multiple event listeners

### Asked at Companies:

- Google: "Optimize a React component that renders 10,000 items"
- Facebook: "Debug and fix performance issues in a large JavaScript application"
- Amazon: "Design a caching strategy for a high-traffic e-commerce site"
- Microsoft: "Implement lazy loading for a complex dashboard"
- **Netflix**: "Optimize video streaming performance across different devices"

### **6** Key Takeaways

- 1. **Measure before optimizing** Use profiling tools to identify bottlenecks
- 2. Optimize for the critical path Focus on what affects user experience most
- 3. Use appropriate data structures Map for lookups, Set for unique values
- 4. Minimize DOM manipulation Batch updates, use virtual scrolling
- 5. Implement proper caching HTTP cache, memory cache, service workers
- 6. **Avoid memory leaks** Clean up event listeners, timers, and references
- 7. Use Web Workers For CPU-intensive tasks
- 8. Monitor performance continuously Set up alerts and budgets

Previous Chapter: ← Testing & Debugging Next Chapter: Security Best Practices →

Practice: Profile a real application, identify bottlenecks, and implement optimizations!

# Chapter 17: Performance Optimization 4

### **Table of Contents**

Performance Fundamentals

- Memory Management
- Code Optimization
- DOM Performance
- Network Optimization
- Async Performance
- Bundle Optimization
- Monitoring & Profiling
- Common Pitfalls
- Practice Problems
- Interview Notes

### **©** Performance Fundamentals

### **Understanding Performance Metrics**

```
// Core Web Vitals
function measureCoreWebVitals() {
 // Largest Contentful Paint (LCP)
 new PerformanceObserver((entryList) => {
   const entries = entryList.getEntries();
   const lastEntry = entries[entries.length - 1];
   console.log("LCP:", lastEntry.startTime);
 }).observe({ entryTypes: ["largest-contentful-paint"] });
 // First Input Delay (FID)
 new PerformanceObserver((entryList) => {
   const entries = entryList.getEntries();
   entries.forEach((entry) => {
      console.log("FID:", entry.processingStart - entry.startTime);
   });
 }).observe({ entryTypes: ["first-input"] });
 // Cumulative Layout Shift (CLS)
 let clsValue = 0;
 new PerformanceObserver((entryList) => {
   const entries = entryList.getEntries();
   entries.forEach((entry) => {
     if (!entry.hadRecentInput) {
        clsValue += entry.value;
     }
   });
   console.log("CLS:", clsValue);
 }).observe({ entryTypes: ["layout-shift"] });
}
```

#### Performance Timing API

```
// Measure custom performance
class PerformanceTracker {
 static mark(name) {
    performance.mark(name);
 static measure(name, startMark, endMark) {
    performance.measure(name, startMark, endMark);
    const measure = performance.getEntriesByName(name)[∅];
    console.log(`${name}: ${measure.duration.toFixed(2)}ms`);
    return measure.duration;
  }
 static async timeFunction(fn, label) {
   const start = performance.now();
    const result = await fn();
    const end = performance.now();
    console.log(`${label}: ${(end - start).toFixed(2)}ms`);
    return result;
 }
 static getNavigationTiming() {
    const timing = performance.getEntriesByType("navigation")[0];
    return {
      dns: timing.domainLookupEnd - timing.domainLookupStart,
      tcp: timing.connectEnd - timing.connectStart,
      request: timing.responseStart - timing.requestStart,
      response: timing.responseEnd - timing.responseStart,
      domParsing: timing.domInteractive - timing.responseEnd,
      domReady:
        timing.domContentLoadedEventEnd - timing.domContentLoadedEventStart,
      loadComplete: timing.loadEventEnd - timing.loadEventStart,
   };
 }
}
// Usage
PerformanceTracker.mark("data-fetch-start");
await fetchData();
PerformanceTracker.mark("data-fetch-end");
PerformanceTracker.measure("data-fetch", "data-fetch-start", "data-fetch-end");
```

### Memory Management

### **Understanding Memory Leaks**

```
// Common memory leak patterns
// 1. Global variables
```

```
// X Memory leak
 function createUser() {
   // Accidentally creates global variable
  user = { name: "John", data: new Array(1000000) };
  return user;
 }
// ✓ Proper scoping
function createUser() {
  const user = { name: "John", data: new Array(1000000) };
  return user;
 }
// 2. Event listeners not removed
 // X Memory leak
 class Component {
  constructor() {
    this.handleClick = this.handleClick.bind(this);
    document.addEventListener("click", this.handleClick);
   }
  handleClick() {
    console.log("Clicked");
   }
  // Missing cleanup!
 }
 // // Proper cleanup
 class Component {
   constructor() {
    this.handleClick = this.handleClick.bind(this);
    document.addEventListener("click", this.handleClick);
   }
  handleClick() {
    console.log("Clicked");
   }
  destroy() {
    document.removeEventListener("click", this.handleClick);
 }
 // 3. Timers not cleared
 // X Memory leak
 class Timer {
  start() {
    this.interval = setInterval(() => {
      console.log("Timer tick");
    }, 1000);
   }
  // Missing cleanup!
```

```
// Proper cleanup
class Timer {
    start() {
        this.interval = setInterval(() => {
            console.log("Timer tick");
        }, 1000);
    }

    stop() {
        if (this.interval) {
            clearInterval(this.interval);
            this.interval = null;
        }
    }
}
```

### **Memory Optimization Techniques**

```
// Object pooling
class ObjectPool {
  constructor(createFn, resetFn, initialSize = 10) {
    this.createFn = createFn;
    this.resetFn = resetFn;
    this.pool = [];
    // Pre-populate pool
    for (let i = 0; i < initialSize; i++) {
      this.pool.push(this.createFn());
    }
  }
  acquire() {
    if (this.pool.length > 0) {
      return this.pool.pop();
    return this.createFn();
  }
  release(obj) {
    this.resetFn(obj);
    this.pool.push(obj);
  }
  size() {
    return this.pool.length;
  }
}
// Usage for expensive objects
```

```
const particlePool = new ObjectPool(
  () => ({ x: 0, y: 0, vx: 0, vy: 0, life: 1 }),
  (particle) => {
    particle.x = 0;
    particle.y = ∅;
    particle.vx = ∅;
    particle.vy = ∅;
    particle.life = 1;
 },
 100
);
// WeakMap for private data
const privateData = new WeakMap();
class User {
  constructor(name) {
    this.name = name;
    privateData.set(this, {
      id: Math.random(),
      secret: "private-data",
    });
  }
 getPrivateData() {
   return privateData.get(this);
  }
}
// When User instance is garbage collected,
// its private data is automatically removed from WeakMap
```

#### **Memory Monitoring**

```
console.log(
          `Memory: ${memory.used}MB / ${memory.total}MB (limit:
${memory.limit}MB)`
        );
        // Alert if memory usage is high
        if (memory.used / memory.limit > 0.8) {
          console.warn("High memory usage detected!");
      }
    }, interval);
  static measureFunction(fn, label) {
    const beforeMemory = this.getMemoryUsage();
    const start = performance.now();
    const result = fn();
    const end = performance.now();
    const afterMemory = this.getMemoryUsage();
    console.log(`${label}:`, {
      duration: `${(end - start).toFixed(2)}ms`,
      memoryDelta:
        beforeMemory && afterMemory
          ? `${afterMemory.used - beforeMemory.used}MB`
          : "N/A",
    });
    return result;
  }
}
// Usage
const monitorId = MemoryMonitor.startMonitoring();
// Later...
clearInterval(monitorId);
```

## Code Optimization

### Algorithm Optimization

```
// Optimize loops
// X Inefficient
function processItems(items) {
  const results = [];
  for (let i = 0; i < items.length; i++) {
    if (items[i].active) {</pre>
```

```
for (let j = 0; j < items[i].tags.length; j++) {</pre>
        if (items[i].tags[j].length > 3) {
          results.push({
            id: items[i].id,
            tag: items[i].tags[j].toUpperCase(),
          });
        }
      }
    }
  }
  return results;
}
// V Optimized
function processItemsOptimized(items) {
  const results = [];
  const itemsLength = items.length;
  for (let i = 0; i < itemsLength; i++) {
    const item = items[i];
    if (!item.active) continue;
    const tags = item.tags;
    const tagsLength = tags.length;
    const itemId = item.id;
    for (let j = 0; j < tagsLength; j++) {
      const tag = tags[j];
      if (tag.length > 3) {
        results.push({
          id: itemId,
          tag: tag.toUpperCase(),
        });
      }
    }
  }
  return results;
}
// Even better: Use functional approach with early returns
function processItemsFunctional(items) {
  return items
    .filter((item) => item.active)
    .flatMap((item) =>
      item.tags
        .filter((tag) => tag.length > 3)
        .map((tag) \Rightarrow ({
          id: item.id,
          tag: tag.toUpperCase(),
        }))
    );
}
```

### **Data Structure Optimization**

```
// Use appropriate data structures
// X Array for lookups (O(n))
class UserManagerSlow {
 constructor() {
   this.users = [];
 addUser(user) {
   this.users.push(user);
 findUser(id) {
    return this.users.find((user) => user.id === id); // O(n)
  }
 removeUser(id) {
    const index = this.users.findIndex((user) => user.id === id); // O(n)
   if (index !== -1) {
     this.users.splice(index, 1); // O(n)
}
// Map for lookups (0(1))
class UserManagerFast {
  constructor() {
   this.users = new Map();
  }
  addUser(user) {
   this.users.set(user.id, user); // 0(1)
  }
 findUser(id) {
    return this.users.get(id); // 0(1)
  }
 removeUser(id) {
    return this.users.delete(id); // 0(1)
  }
  getAllUsers() {
    return Array.from(this.users.values());
  }
}
// Set for unique values
class TagManager {
```

```
constructor() {
    this.tags = new Set();
}

addTag(tag) {
    this.tags.add(tag.toLowerCase());
}

hasTag(tag) {
    return this.tags.has(tag.toLowerCase());
}

getUniqueTagCount() {
    return this.tags.size;
}
}
```

### **Function Optimization**

```
// 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;
 };
}
// Expensive function
const fibonacci = memoize(function (n) {
  if (n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
});
// Debouncing
function debounce(func, wait) {
  let timeout;
  return function executedFunction(...args) {
    const later = () => {
      clearTimeout(timeout);
      func(...args);
    };
```

```
clearTimeout(timeout);
    timeout = setTimeout(later, wait);
 };
// Throttling
function throttle(func, limit) {
  let inThrottle;
 return function (...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
 };
}
// Usage
const debouncedSearch = debounce(searchFunction, 300);
const throttledScroll = throttle(scrollHandler, 100);
```

### © DOM Performance

### **Efficient DOM Manipulation**

```
// X Inefficient DOM updates
function updateListSlow(items) {
  const list = document.getElementById("list");
  list.innerHTML = ""; // Triggers reflow
  items.forEach((item) => {
    const li = document.createElement("li");
    li.textContent = item.name;
    list.appendChild(li); // Triggers reflow for each item
 });
}
// Batch DOM updates
function updateListFast(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); // No reflow
  });
```

```
list.innerHTML = "";
list.appendChild(fragment); // Single reflow
}

// Even better: Use template strings
function updateListTemplate(items) {
  const list = document.getElementById("list");
  const html = items.map((item) => `${item.name}`).join("");
  list.innerHTML = html; // Single reflow
}
```

#### Virtual Scrolling

```
// Virtual scrolling for large lists
class VirtualList {
 constructor(container, items, itemHeight = 50) {
   this.container = container;
   this.items = items;
   this.itemHeight = itemHeight;
   this.visibleCount = Math.ceil(container.clientHeight / itemHeight) + 2;
   this.startIndex = 0;
   this.setupContainer();
   this.render();
   this.bindEvents();
 }
 setupContainer() {
   this.container.style.overflow = "auto";
   this.container.style.position = "relative";
   // Create scrollable area
   this.scrollArea = document.createElement("div");
   this.scrollArea.style.height = `${this.items.length * this.itemHeight}px`;
   this.container.appendChild(this.scrollArea);
   // Create visible items container
   this.itemsContainer = document.createElement("div");
   this.itemsContainer.style.position = "absolute";
   this.itemsContainer.style.top = "0";
   this.itemsContainer.style.width = "100%";
   this.scrollArea.appendChild(this.itemsContainer);
 }
 render() {
   const endIndex = Math.min(
     this.startIndex + this.visibleCount,
     this.items.length
   );
    const visibleItems = this.items.slice(this.startIndex, endIndex);
```

```
this.itemsContainer.innerHTML = "";
    this.itemsContainer.style.transform = `translateY(${
      this.startIndex * this.itemHeight
    }px)`;
    visibleItems.forEach((item, index) => {
      const element = document.createElement("div");
      element.style.height = `${this.itemHeight}px`;
      element.textContent = item.name;
      element.dataset.index = this.startIndex + index;
      this.itemsContainer.appendChild(element);
    });
  }
  bindEvents() {
    this.container.addEventListener("scroll", () => {
      const newStartIndex = Math.floor(
       this.container.scrollTop / this.itemHeight
      );
      if (newStartIndex !== this.startIndex) {
        this.startIndex = newStartIndex;
        this.render();
    });
 }
}
// Usage
const items = Array.from({ length: 10000 }, (_, i) => ({ name: `Item ${i}` }));
const virtualList = new VirtualList(
  document.getElementById("list-container"),
  items
);
```

#### **Event Delegation**

```
});
}
// Advanced event delegation
class EventDelegator {
 constructor(container) {
   this.container = container;
   this.handlers = new Map();
   this.bindEvents();
 }
 on(selector, eventType, handler) {
    const key = `${eventType}:${selector}`;
   if (!this.handlers.has(key)) {
     this.handlers.set(key, []);
   this.handlers.get(key).push(handler);
 }
 off(selector, eventType, handler) {
    const key = `${eventType}:${selector}`;
    const handlers = this.handlers.get(key);
    if (handlers) {
      const index = handlers.indexOf(handler);
      if (index > -1) {
       handlers.splice(index, 1);
   }
  }
 bindEvents() {
   this.container.addEventListener("click", this.handleEvent.bind(this));
   this.container.addEventListener("change", this.handleEvent.bind(this));
    this.container.addEventListener("input", this.handleEvent.bind(this));
 }
 handleEvent(event) {
    const key = `${event.type}:`;
    for (const [handlerKey, handlers] of this.handlers) {
      if (handlerKey.startsWith(key)) {
        const selector = handlerKey.substring(key.length);
        if (event.target.matches(selector)) {
          handlers.forEach((handler) => handler(event));
        }
     }
   }
 }
}
// Usage
const delegator = new EventDelegator(document.body);
delegator.on(".button", "click", handleButtonClick);
delegator.on(".input", "input", handleInputChange);
```

### Metwork Optimization

### Request Optimization

```
// Request batching
class RequestBatcher {
 constructor(batchSize = 10, delay = 100) {
   this.batchSize = batchSize;
   this.delay = delay;
   this.queue = [];
   this.timeoutId = null;
 }
 add(request) {
   return new Promise((resolve, reject) => {
     this.queue.push({ request, resolve, reject });
     if (this.queue.length >= this.batchSize) {
       this.flush();
     } else if (!this.timeoutId) {
       this.timeoutId = setTimeout(() => this.flush(), this.delay);
   });
 async flush() {
   if (this.queue.length === 0) return;
   const batch = this.queue.splice(0, this.batchSize);
   if (this.timeoutId) {
     clearTimeout(this.timeoutId);
     this.timeoutId = null;
   }
   try {
     const requests = batch.map((item) => item.request);
     const results = await this.processBatch(requests);
     batch.forEach((item, index) => {
        item.resolve(results[index]);
     });
   } catch (error) {
      batch.forEach((item) => item.reject(error));
   }
 async processBatch(requests) {
   // Send all requests in a single API call
```

```
const response = await fetch("/api/batch", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ requests }),
    });

    return response.json();
}

// Usage
const batcher = new RequestBatcher();

// These will be batched together
const user1 = await batcher.add({ type: "getUser", id: 1 });
const user2 = await batcher.add({ type: "getUser", id: 2 });
const user3 = await batcher.add({ type: "getUser", id: 3 });
```

### **Caching Strategies**

```
// HTTP cache with TTL
class HTTPCache {
 constructor(defaultTTL = 300000) {
  // 5 minutes
   this.cache = new Map();
   this.defaultTTL = defaultTTL;
 }
 set(key, value, ttl = this.defaultTTL) {
   const expiry = Date.now() + ttl;
   this.cache.set(key, { value, expiry });
 }
 get(key) {
   const item = this.cache.get(key);
   if (!item) return null;
   if (Date.now() > item.expiry) {
     this.cache.delete(key);
      return null;
   return item.value;
  }
 clear() {
   this.cache.clear();
 cleanup() {
```

```
const now = Date.now();
    for (const [key, item] of this.cache) {
      if (now > item.expiry) {
       this.cache.delete(key);
   }
 }
}
// API client with caching
class CachedAPIClient {
 constructor() {
   this.cache = new HTTPCache();
   this.pendingRequests = new Map();
 }
  async get(url, options = {}) {
    const cacheKey = this.getCacheKey(url, options);
   // Check cache first
    const cached = this.cache.get(cacheKey);
   if (cached) {
      return cached;
    }
   // Check if request is already pending
    if (this.pendingRequests.has(cacheKey)) {
      return this.pendingRequests.get(cacheKey);
    }
    // Make request
    const requestPromise = this.makeRequest(url, options)
      .then((response) => {
        this.cache.set(cacheKey, response, options.ttl);
        this.pendingRequests.delete(cacheKey);
        return response;
      })
      .catch((error) => {
       this.pendingRequests.delete(cacheKey);
       throw error;
      });
   this.pendingRequests.set(cacheKey, requestPromise);
    return requestPromise;
  }
  async makeRequest(url, options) {
   const response = await fetch(url, options);
    if (!response.ok) {
     throw new Error(`HTTP ${response.status}`);
    return response.json();
  }
```

```
getCacheKey(url, options) {
    return `${url}:${JSON.stringify(options)}`;
}
```

#### Resource Loading

```
// Lazy loading with Intersection Observer
class LazyLoader {
  constructor(options = {}) {
    this.options = {
      rootMargin: "50px",
      threshold: 0.1,
      ...options,
    };
    this.observer = new IntersectionObserver(
      this.handleIntersection.bind(this),
      this.options
    );
  }
  observe(element) {
    this.observer.observe(element);
  }
  unobserve(element) {
    this.observer.unobserve(element);
  }
  handleIntersection(entries) {
    entries.forEach((entry) => {
      if (entry.isIntersecting) {
        this.loadElement(entry.target);
        this.observer.unobserve(entry.target);
      }
    });
  loadElement(element) {
    if (element.dataset.src) {
      element.src = element.dataset.src;
      element.removeAttribute("data-src");
    }
    if (element.dataset.srcset) {
      element.srcset = element.dataset.srcset;
      element.removeAttribute("data-srcset");
    }
    element.classList.add("loaded");
```

```
// Preloading critical resources
class ResourcePreloader {
  static preloadImage(src) {
    return new Promise((resolve, reject) => {
      const img = new Image();
      img.onload = () => resolve(img);
      img.onerror = reject;
      img.src = src;
    });
  }
  static preloadScript(src) {
    return new Promise((resolve, reject) => {
      const script = document.createElement("script");
      script.onload = resolve;
      script.onerror = reject;
      script.src = src;
      document.head.appendChild(script);
    });
  }
  static preloadCSS(href) {
    return new Promise((resolve, reject) => {
      const link = document.createElement("link");
      link.rel = "stylesheet";
      link.onload = resolve;
      link.onerror = reject;
      link.href = href;
      document.head.appendChild(link);
    });
  }
  static async preloadCriticalResources(resources) {
    const promises = resources.map((resource) => {
      switch (resource.type) {
        case "image":
          return this.preloadImage(resource.src);
        case "script":
          return this.preloadScript(resource.src);
        case "css":
          return this.preloadCSS(resource.href);
        default:
          return Promise.resolve();
      }
    });
    return Promise.all(promises);
  }
}
// Usage
```

```
const lazyLoader = new LazyLoader();
document.querySelectorAll("img[data-src]").forEach((img) => {
    lazyLoader.observe(img);
});

// Preload critical resources
ResourcePreloader.preloadCriticalResources([
    { type: "image", src: "/hero-image.jpg" },
    { type: "script", src: "/critical.js" },
    { type: "css", href: "/critical.css" },
]);
```

### 4 Async Performance

### Parallel vs Sequential Execution

```
// X Sequential execution (slow)
async function fetchUserDataSequential(userIds) {
 const users = [];
 for (const id of userIds) {
   const user = await fetchUser(id); // Waits for each request
   users.push(user);
 return users;
}
// Parallel execution (fast)
async function fetchUserDataParallel(userIds) {
 const promises = userIds.map((id) => fetchUser(id));
 return Promise.all(promises);
}
// Controlled concurrency
async function fetchUserDataConcurrent(userIds, concurrency = 3) {
 const results = [];
 for (let i = 0; i < userIds.length; i += concurrency) {</pre>
   const batch = userIds.slice(i, i + concurrency);
   const batchPromises = batch.map((id) => fetchUser(id));
   const batchResults = await Promise.all(batchPromises);
   results.push(...batchResults);
 return results;
// Advanced: Promise pool with retry
class PromisePool {
```

```
constructor(concurrency = 3) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  async add(promiseFactory, retries = 3) {
    return new Promise((resolve, reject) => {
      this.queue.push({
        promiseFactory,
        resolve,
        reject,
        retries,
      });
      this.process();
   });
  }
  async process() {
    if (this.running >= this.concurrency || this.queue.length === 0) {
      return;
    }
    this.running++;
    const { promiseFactory, resolve, reject, retries } = this.queue.shift();
    try {
      const result = await promiseFactory();
      resolve(result);
    } catch (error) {
      if (retries > 0) {
        // Retry with exponential backoff
        setTimeout(() => {
          this.queue.unshift({
            promiseFactory,
            resolve,
            reject,
            retries: retries - 1,
          });
          this.process();
        }, Math.pow(2, 3 - retries) * 1000);
      } else {
        reject(error);
      }
    } finally {
     this.running--;
      this.process();
    }
 }
}
// Usage
const pool = new PromisePool(5);
```

```
const results = await Promise.all(
  userIds.map((id) => pool.add(() => fetchUser(id)))
);
```

#### Web Workers for Heavy Tasks

```
// Main thread
class WorkerPool {
 constructor(workerScript, poolSize = navigator.hardwareConcurrency | | 4) {
    this.workers = [];
    this.queue = [];
    this.taskId = 0;
    for (let i = 0; i < poolSize; i++) {
      this.workers.push({
        worker: new Worker(workerScript),
        busy: false,
     });
    }
 }
 execute(data) {
    return new Promise((resolve, reject) => {
      const taskId = ++this.taskId;
      this.queue.push({
        taskId,
       data,
        resolve,
        reject,
      });
     this.processQueue();
    });
  }
 processQueue() {
    if (this.queue.length === 0) return;
    const availableWorker = this.workers.find((w) => !w.busy);
    if (!availableWorker) return;
    const task = this.queue.shift();
    availableWorker.busy = true;
    const handleMessage = (event) => {
      if (event.data.taskId === task.taskId) {
        availableWorker.worker.removeEventListener("message", handleMessage);
        availableWorker.worker.removeEventListener("error", handleError);
        availableWorker.busy = false;
```

```
if (event.data.error) {
         task.reject(new Error(event.data.error));
        } else {
          task.resolve(event.data.result);
       this.processQueue();
    };
    const handleError = (error) => {
      availableWorker.worker.removeEventListener("message", handleMessage);
      availableWorker.worker.removeEventListener("error", handleError);
      availableWorker.busy = false;
     task.reject(error);
     this.processQueue();
    };
    availableWorker.worker.addEventListener("message", handleMessage);
    availableWorker.worker.addEventListener("error", handleError);
    availableWorker.worker.postMessage({
     taskId: task.taskId,
      data: task.data,
   });
 }
 terminate() {
   this.workers.forEach((w) => w.worker.terminate());
 }
}
// worker.js
self.addEventListener("message", (event) => {
 const { taskId, data } = event.data;
 try {
   // Heavy computation
   const result = processLargeDataset(data);
    self.postMessage({
      taskId,
     result,
   });
  } catch (error) {
    self.postMessage({
     taskId,
      error: error.message,
   });
});
function processLargeDataset(data) {
 // CPU-intensive task
```

2025-07-22 javascript.md

```
return data.map((item) => {
   // Complex calculations
   return Math.sqrt(item * item + item);
 });
// Usage
const workerPool = new WorkerPool("worker.js", 4);
const result = await workerPool.execute(largeDataArray);
```

### Bundle Optimization

### **Code Splitting**

```
// Dynamic imports for code splitting
class ModuleLoader {
 constructor() {
   this.loadedModules = new Map();
 }
 async loadModule(moduleName) {
    if (this.loadedModules.has(moduleName)) {
      return this.loadedModules.get(moduleName);
    }
    let modulePromise;
    switch (moduleName) {
      case "chart":
        modulePromise = import("./modules/chart.js");
        break;
      case "editor":
        modulePromise = import("./modules/editor.js");
        break;
      case "calendar":
        modulePromise = import("./modules/calendar.js");
        break;
      default:
        throw new Error(`Unknown module: ${moduleName}`);
    }
    this.loadedModules.set(moduleName, modulePromise);
    return modulePromise;
  }
  async loadModuleOnDemand(moduleName, trigger) {
    const loadModule = async () => {
      const module = await this.loadModule(moduleName);
      return module.default | module;
    };
```

```
// Load on user interaction
   if (trigger) {
     trigger.addEventListener("click", loadModule, { once: true });
   return loadModule;
 }
}
// Route-based code splitting
class Router {
 constructor() {
   this.routes = new Map();
   this.moduleLoader = new ModuleLoader();
 addRoute(path, moduleFactory) {
   this.routes.set(path, moduleFactory);
  }
 async navigate(path) {
    const moduleFactory = this.routes.get(path);
   if (!moduleFactory) {
     throw new Error(`Route not found: ${path}`);
    }
    // Show loading indicator
   this.showLoading();
   try {
      const module = await moduleFactory();
      const component = new module.default();
     this.render(component);
    } catch (error) {
     this.showError(error);
    } finally {
     this.hideLoading();
   }
  }
  showLoading() {
   document.getElementById("loading").style.display = "block";
  }
 hideLoading() {
   document.getElementById("loading").style.display = "none";
 }
 render(component) {
    const container = document.getElementById("app");
    container.innerHTML = "";
    container.appendChild(component.render());
```

```
showError(error) {
    console.error("Route loading error:", error);
}

// Usage
const router = new Router();

router.addRoute("/dashboard", () => import("./pages/Dashboard.js"));
router.addRoute("/profile", () => import("./pages/Profile.js"));
router.addRoute("/settings", () => import("./pages/Settings.js"));
```

### Tree Shaking Optimization

```
// X Imports entire library
import * as utils from "./utils.js";
utils.debounce(fn, 300);
// ✓ Import only what you need
import { debounce } from "./utils.js";
debounce(fn, 300);
// Create tree-shakable modules
// utils.js
export function debounce(func, wait) {
 // Implementation
export function throttle(func, limit) {
 // Implementation
}
export function memoize(func) {
 // Implementation
}
// Don't export default objects
// ★ Not tree-shakable
export default {
  debounce,
 throttle,
  memoize,
};
// // Tree-shakable
export { debounce, throttle, memoize };
```

### **Ⅲ** Monitoring & Profiling

### Performance Monitoring

```
// Comprehensive performance monitor
class PerformanceMonitor {
 constructor() {
   this.metrics = new Map();
   this.observers = [];
   this.setupObservers();
 }
  setupObservers() {
   // Long tasks
   if ("PerformanceObserver" in window) {
      const longTaskObserver = new PerformanceObserver((list) => {
        list.getEntries().forEach((entry) => {
          this.recordMetric("longTask", {
            duration: entry.duration,
            startTime: entry.startTime,
          });
       });
      });
      try {
        longTaskObserver.observe({ entryTypes: ["longtask"] });
        this.observers.push(longTaskObserver);
      } catch (e) {
        console.warn("Long task observer not supported");
      }
    }
    // Layout shifts
    const layoutShiftObserver = new PerformanceObserver((list) => {
      list.getEntries().forEach((entry) => {
        if (!entry.hadRecentInput) {
          this.recordMetric("layoutShift", {
            value: entry.value,
            startTime: entry.startTime,
          });
        }
      });
    });
    layoutShiftObserver.observe({ entryTypes: ["layout-shift"] });
    this.observers.push(layoutShiftObserver);
  }
  recordMetric(name, data) {
    if (!this.metrics.has(name)) {
      this.metrics.set(name, []);
    }
```

```
this.metrics.get(name).push({
    ...data,
   timestamp: Date.now(),
 });
}
getMetrics(name) {
 return this.metrics.get(name) || [];
}
getAverageMetric(name, property) {
  const metrics = this.getMetrics(name);
  if (metrics.length === 0) return 0;
 const sum = metrics.reduce((acc, metric) => acc + metric[property], 0);
 return sum / metrics.length;
}
generateReport() {
  const report = {
    timestamp: new Date().toISOString(),
    navigation: this.getNavigationTiming(),
    longTasks: this.getMetrics("longTask"),
    layoutShifts: this.getMetrics("layoutShift"),
    memory: this.getMemoryInfo(),
    averages: {
      longTaskDuration: this.getAverageMetric("longTask", "duration"),
      layoutShiftValue: this.getAverageMetric("layoutShift", "value"),
    },
  };
  return report;
}
getNavigationTiming() {
  const timing = performance.getEntriesByType("navigation")[0];
  if (!timing) return null;
  return {
    dns: timing.domainLookupEnd - timing.domainLookupStart,
    tcp: timing.connectEnd - timing.connectStart,
    request: timing.responseStart - timing.requestStart,
    response: timing.responseEnd - timing.responseStart,
    domParsing: timing.domInteractive - timing.responseEnd,
    domReady:
      timing.domContentLoadedEventEnd - timing.domContentLoadedEventStart,
    loadComplete: timing.loadEventEnd - timing.loadEventStart,
  };
}
getMemoryInfo() {
  if (performance.memory) {
    return {
```

```
used: Math.round(performance.memory.usedJSHeapSize / 1024 / 1024),
        total: Math.round(performance.memory.totalJSHeapSize / 1024 / 1024),
        limit: Math.round(performance.memory.jsHeapSizeLimit / 1024 / 1024),
      };
    return null;
  sendReport(endpoint) {
    const report = this.generateReport();
    // Use sendBeacon for reliability
    if (navigator.sendBeacon) {
      navigator.sendBeacon(endpoint, JSON.stringify(report));
    } else {
      fetch(endpoint, {
        method: "POST",
        body: JSON.stringify(report),
        headers: { "Content-Type": "application/json" },
        keepalive: true,
      }).catch(console.error);
    }
  }
 disconnect() {
    this.observers.forEach((observer) => observer.disconnect());
  }
}
// Usage
const monitor = new PerformanceMonitor();
// Send report every 30 seconds
setInterval(() => {
  monitor.sendReport("/api/performance");
}, 30000);
// Send report before page unload
window.addEventListener("beforeunload", () => {
  monitor.sendReport("/api/performance");
});
```

# 

### 1. Premature Optimization

```
// X Optimizing before measuring
function processData(data) {
   // Complex optimization that may not be needed
   const cache = new Map();
```

```
const pool = new ObjectPool();
   // ... overly complex code
}

// Measure first, then optimize
function processData(data) {
   // Simple, readable implementation first
   return data.map((item) => transform(item));
}

// Profile and optimize only if needed
```

# 2. Memory Leaks in Event Listeners

```
// X Memory leak
class Component {
 constructor() {
   this.handleClick = this.handleClick.bind(this);
   document.addEventListener("click", this.handleClick);
 }
 handleClick() {
   console.log("Clicked");
 }
 // Missing cleanup!
// // Proper cleanup
class Component {
 constructor() {
   this.handleClick = this.handleClick.bind(this);
   document.addEventListener("click", this.handleClick);
 }
 handleClick() {
   console.log("Clicked");
 destroy() {
   document.removeEventListener("click", this.handleClick);
  }
}
```

# 3. Blocking the Main Thread

```
// X Blocking operation
function processLargeArray(data) {
  for (let i = 0; i < data.length; i++) {</pre>
```

```
// Heavy computation blocks UI
    heavyCalculation(data[i]);
 }
}
// ✓ Non-blocking with time slicing
function processLargeArrayNonBlocking(data, callback) {
  let index = 0;
  const batchSize = 1000;
  function processBatch() {
    const endIndex = Math.min(index + batchSize, data.length);
    for (let i = index; i < endIndex; i++) {</pre>
      heavyCalculation(data[i]);
    index = endIndex;
    if (index < data.length) {</pre>
      setTimeout(processBatch, ∅); // Yield to browser
    } else {
      callback();
  }
  processBatch();
}
```

# 4. Inefficient DOM Queries

```
// ★ Repeated DOM queries
function updateElements() {
 for (let i = 0; i < 100; i++) {
   document.getElementById("item-" + i).textContent = "Updated";
    document.getElementById("item-" + i).classList.add("active");
 }
}
// Cache DOM references
function updateElementsOptimized() {
 const elements = [];
 for (let i = 0; i < 100; i++) {
    elements[i] = document.getElementById("item-" + i);
 }
 elements.forEach((element) => {
    element.textContent = "Updated";
    element.classList.add("active");
 });
```

# 5. Not Using RequestAnimationFrame

```
// X Using setTimeout for animations
function animateElement(element) {
 let position = 0;
 const animate = () => {
    position += 1;
   element.style.left = position + "px";
    if (position < 100) {
      setTimeout(animate, 16); // Not synced with display
 };
  animate();
}
// ✓ Using requestAnimationFrame
function animateElementOptimized(element) {
 let position = ∅;
 const animate = () => {
    position += 1;
    element.style.left = position + "px";
    if (position < 100) {
      requestAnimationFrame(animate); // Synced with display
    }
 };
  requestAnimationFrame(animate);
```

# Mini Practice Problems

### Problem 1: Optimize a Data Processing Pipeline

```
// Optimize this slow data processing function
function processUserData(users) {
 const results = [];
 for (let i = 0; i < users.length; i++) {
   const user = users[i];
   if (user.active) {
      const profile = getUserProfile(user.id); // Sync API call
      const permissions = getUserPermissions(user.id); // Sync API call
      const preferences = getUserPreferences(user.id); // Sync API call
     const processed = {
        id: user.id,
        name: user.name.toUpperCase(),
        email: user.email.toLowerCase(),
```

```
profile: profile,
        permissions: permissions,
        preferences: preferences,
        score: calculateUserScore(user, profile, permissions),
      };
      results.push(processed);
    }
  }
  return results.sort((a, b) => b.score - a.score);
}
// Tasks:
// 1. Make API calls parallel
// 2. Implement caching
// 3. Add batch processing
// 4. Use Web Workers for heavy calculations
// 5. Implement progress tracking
```

# Problem 2: Create a High-Performance Virtual List

```
// Implement a virtual list that can handle 100,000+ items smoothly
class HighPerformanceVirtualList {
 constructor(container, items, itemHeight, renderItem) {
   // Your implementation here
   // Requirements:
   // - Smooth scrolling with 100k+ items
   // - Dynamic item heights support
   // - Horizontal scrolling support
   // - Search and filtering
   // - Memory efficient
   // - Keyboard navigation
 }
 // Methods to implement:
 // render()
 // scrollToIndex(index)
 // updateItems(newItems)
 // filter(predicate)
 // search(query)
 // destroy()
}
```

## Problem 3: Build a Performance Budget Monitor

```
// Create a system that monitors and enforces performance budgets
class PerformanceBudgetMonitor {
  constructor(budgets) {
```

```
// budgets = {
          bundleSize: 250000, // 250KB
   //
   //
          firstContentfulPaint: 1500, // 1.5s
   //
          largestContentfulPaint: 2500, // 2.5s
          cumulativeLayoutShift: 0.1,
          firstInputDelay: 100 // 100ms
   // }
 // Methods to implement:
 // startMonitoring()
 // checkBudgets()
 // reportViolations()
 // generateReport()
 // setupAlerts()
}
```

# Problem 4: Optimize Image Loading

```
// Create an advanced image loading system
class ImageOptimizer {
 constructor(options) {
   // Features to implement:
   // - Lazy loading with intersection observer
   // - Progressive image loading (blur-up)
   // - WebP/AVIF format detection and serving
   // - Responsive images based on device
   // - Image compression on the fly
   // - Preloading critical images
   // - Error handling and fallbacks
  }
 // Methods to implement:
 // loadImage(src, options)
 // preloadImages(srcs)
 // generateResponsiveSrc(src, sizes)
 // compressImage(file, quality)
  // convertToWebP(file)
}
```

# Problem 5: Memory Leak Detector

```
// Build a tool to detect and report memory leaks
class MemoryLeakDetector {
  constructor() {
    // Features:
    // - Track object creation and destruction
    // - Monitor DOM node leaks
    // - Detect event listener leaks
```

```
// - Monitor closure leaks
// - Generate leak reports
// - Suggest fixes
}

// Methods to implement:
// startTracking()
// trackObject(obj, label)
// detectLeaks()
// generateReport()
// suggestFixes()
// cleanup()
}
```

# Interview Notes

### **Common Questions:**

### Q: What are the main performance bottlenecks in web applications?

- Network: Large bundles, too many requests, no caching
- Rendering: Layout thrashing, unnecessary repaints, large DOM
- JavaScript: Blocking main thread, memory leaks, inefficient algorithms
- Images: Large file sizes, no optimization, blocking loading

### Q: How do you measure web performance?

- Core Web Vitals: LCP, FID, CLS
- Performance API: Navigation timing, resource timing
- Tools: Lighthouse, WebPageTest, Chrome DevTools
- Real User Monitoring: Track actual user experience

### Q: What's the difference between throttling and debouncing?

- **Throttling**: Limits function execution to once per time period
- **Debouncing**: Delays function execution until after a quiet period
- **Use cases**: Throttle for scroll events, debounce for search input

# Q: How do you optimize JavaScript bundle size?

- Tree shaking: Remove unused code
- Code splitting: Load code on demand
- Minification: Compress code
- Compression: Gzip/Brotli
- Dynamic imports: Lazy load modules

### Q: What causes memory leaks in JavaScript?

- **Global variables**: Unintentional global scope pollution
- Event listeners: Not removed when elements are destroyed

- Timers: setInterval/setTimeout not cleared
- **Closures**: Retaining references to large objects
- DOM references: Keeping references to removed DOM nodes

### Q: How do you optimize DOM manipulation?

• Batch updates: Use DocumentFragment

• Minimize reflows: Change styles in batches

• Use CSS transforms: For animations instead of changing layout properties

Virtual DOM: For complex UIs

• Event delegation: Instead of multiple event listeners

# Asked at Companies:

- Google: "Optimize a React component that renders 10,000 items"
- Facebook: "Debug and fix performance issues in a large JavaScript application"
- Amazon: "Design a caching strategy for a high-traffic e-commerce site"
- Microsoft: "Implement lazy loading for a complex dashboard"
- Netflix: "Optimize video streaming performance across different devices"

# **©** Key Takeaways

- 1. **Measure before optimizing** Use profiling tools to identify bottlenecks
- 2. Optimize for the critical path Focus on what affects user experience most
- 3. Use appropriate data structures Map for lookups, Set for unique values
- 4. Minimize DOM manipulation Batch updates, use virtual scrolling
- 5. Implement proper caching HTTP cache, memory cache, service workers
- 6. Avoid memory leaks Clean up event listeners, timers, and references
- 7. Use Web Workers For CPU-intensive tasks
- 8. Monitor performance continuously Set up alerts and budgets

**Previous Chapter**: ← Testing & Debugging **Next Chapter**: Security Best Practices →

Practice: Profile a real application, identify bottlenecks, and implement optimizations!

# Chapter 19: Modern JavaScript Frameworks 🧭



# 

- Framework Overview
- React Fundamentals
- Vue.js Essentials
- Angular Basics
- State Management
- Component Architecture
- Routing & Navigation

- Performance Optimization
- Common Pitfalls
- Practice Problems
- Interview Notes

# **©** Framework Overview

### Why Use Frameworks?

```
## (8) Routing & Navigation
### React Router
```javascript
// App.js with React Router
import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-
dom';
import { useContext } from 'react';
import { UserContext } from './contexts/UserContext';
// Protected Route Component
function ProtectedRoute({ children }) {
    const { user, loading } = useContext(UserContext);
    if (loading) {
        return <div>Loading...</div>;
    }
    return user ? children : <Navigate to="/login" replace />;
}
// Layout with Navigation
function Layout({ children }) {
    const { user, logout } = useContext(UserContext);
    return (
        <div className="app-layout">
            <nav className="navbar">
                <Link to="/" className="nav-brand">MyApp</Link>
                <div className="nav-links">
                    {user ? (
                        <>
                            <Link to="/dashboard">Dashboard</Link>
                            <Link to="/profile">Profile</Link>
                            <button onClick={logout}>Logout</button>
                        </>
                    ): (
                        <>
                            <Link to="/login">Login</Link>
```

```
<Link to="/register">Register</Link>
                         </>>
                     )}
                </div>
            </nav>
            <main className="main-content">
                {children}
            </main>
        </div>
    );
}
function App() {
    return (
        <UserProvider>
            <Router>
                <Layout>
                     <Routes>
                         <Route path="/" element={<Home />} />
                         <Route path="/login" element={<Login />} />
                         <Route path="/register" element={<Register />} />
                         <Route
                             path="/dashboard"
                             element={
                                 <ProtectedRoute>
                                     <Dashboard />
                                 </ProtectedRoute>
                             }
                         />
                         <Route
                             path="/profile"
                             element={
                                 <ProtectedRoute>
                                     <Profile />
                                 </ProtectedRoute>
                             }
                         />
                         <Route path="/users/:id" element={<UserDetail />} />
                         <Route path="*" element={<NotFound />} />
                     </Routes>
                </Layout>
            </Router>
        </UserProvider>
    );
}
// Component with URL parameters and navigation
function UserDetail() {
    const { id } = useParams();
    const navigate = useNavigate();
    const location = useLocation();
    const [user, setUser] = useState(null);
    const [loading, setLoading] = useState(true);
```

```
useEffect(() => {
       fetchUser(id);
   }, [id]);
   const fetchUser = async (userId) => {
       try {
            setLoading(true);
            const response = await fetch(`/api/users/${userId}`);
            if (response.ok) {
                const userData = await response.json();
                setUser(userData);
            } else {
                navigate('/404', { replace: true });
        } catch (error) {
            console.error('Error fetching user:', error);
            navigate('/error', { state: { error: error.message } });
        } finally {
            setLoading(false);
   };
   const handleEdit = () => {
        navigate(`/users/${id}/edit`, {
            state: { from: location.pathname }
       });
   };
   if (loading) return <div>Loading user...</div>;
   if (!user) return <div>User not found</div>;
   return (
        <div className="user-detail">
            <button onClick={() => navigate(-1)}>← Back</button>
            <h1>{user.name}</h1>
            {user.email}
            <button onClick={handleEdit}>Edit User</button>
        </div>
   );
}
```

#### **Vue Router**

```
component: () => import('../views/Home.vue')
    },
    {
        path: '/login',
        name: 'Login',
        component: () => import('../views/Login.vue'),
        meta: { requiresGuest: true }
    },
    {
        path: '/dashboard',
        name: 'Dashboard',
        component: () => import('../views/Dashboard.vue'),
        meta: { requiresAuth: true }
    },
    {
        path: '/users/:id',
        name: 'UserDetail',
        component: () => import('../views/UserDetail.vue'),
        props: true,
        meta: { requiresAuth: true }
    },
    {
        path: '/users/:id/edit',
        name: 'UserEdit',
        component: () => import('../views/UserEdit.vue'),
        props: true,
        meta: { requiresAuth: true }
    },
    {
        path: '/:pathMatch(.*)*',
        name: 'NotFound',
        component: () => import('../views/NotFound.vue')
    }
];
const router = createRouter({
    history: createWebHistory(),
    routes
});
// Navigation guards
router.beforeEach((to, from, next) => {
    const userStore = useUserStore();
    if (to.meta.requiresAuth && !userStore.isAuthenticated) {
        next({ name: 'Login', query: { redirect: to.fullPath } });
    } else if (to.meta.requiresGuest && userStore.isAuthenticated) {
        next({ name: 'Dashboard' });
    } else {
        next();
    }
});
export default router;
```

```
// UserDetail.vue
<template>
   <div class="user-detail">
        <button @click="$router.go(-1)">← Back</button>
        <div v-if="loading">Loading user...</div>
        <div v-else-if="error">Error: {{ error }}</div>
        <div v-else-if="user">
            <h1>{{ user.name }}</h1>
            {{ user.email }}
            <router-link :to="\users/${user.id}/edit\" class="btn">
                Edit User
            </router-link>
        </div>
   </div>
</template>
<script>
import { ref, onMounted, watch } from 'vue';
import { useRoute, useRouter } from 'vue-router';
export default {
   name: 'UserDetail',
   props: {
       id: {
            type: String,
            required: true
       }
   },
    setup(props) {
        const route = useRoute();
        const router = useRouter();
        const user = ref(null);
        const loading = ref(false);
        const error = ref(null);
        const fetchUser = async (userId) => {
           try {
                loading.value = true;
                error.value = null;
                const response = await fetch(`/api/users/${userId}`);
                if (response.ok) {
                    user.value = await response.json();
                } else {
                    throw new Error('User not found');
                }
            } catch (err) {
                error.value = err.message;
            } finally {
                loading.value = false;
            }
       };
```

```
// Watch for route parameter changes
watch(() => props.id, (newId) => {
    if (newId) {
        fetchUser(newId);
      }
    }, { immediate: true });

return {
    user,
    loading,
    error
    };
}
};
</script>
```

# **Angular Routing**

```
// app-routing.module.ts
import { NgModule } from "@angular/core";
import { RouterModule, Routes } from "@angular/router";
import { AuthGuard } from "./guards/auth.guard";
import { GuestGuard } from "./guards/guest.guard";
const routes: Routes = [
  { path: "", component: HomeComponent },
  {
    path: "login",
    component: LoginComponent,
    canActivate: [GuestGuard],
  },
    path: "dashboard",
    component: DashboardComponent,
    canActivate: [AuthGuard],
  },
    path: "users/:id",
    component: UserDetailComponent,
    canActivate: [AuthGuard],
  },
    path: "users/:id/edit",
    component: UserEditComponent,
    canActivate: [AuthGuard],
  },
  { path: "**", component: NotFoundComponent },
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
```

```
exports: [RouterModule],
})
export class AppRoutingModule {}
// auth.guard.ts
import { Injectable } from "@angular/core";
import { CanActivate, Router } from "@angular/router";
import { UserService } from "../services/user.service";
@Injectable({
 providedIn: "root",
})
export class AuthGuard implements CanActivate {
 constructor(private userService: UserService, private router: Router) {}
  canActivate(): boolean {
    if (this.userService.isAuthenticated()) {
      return true;
    } else {
      this.router.navigate(["/login"]);
      return false;
    }
  }
}
// user-detail.component.ts
import { Component, OnInit, OnDestroy } from "@angular/core";
import { ActivatedRoute, Router } from "@angular/router";
import { Subject } from "rxjs";
import { takeUntil } from "rxjs/operators";
@Component({
  selector: "app-user-detail",
  template: `
    <div class="user-detail">
      <button (click)="goBack()">← Back</button>
      <div *ngIf="loading">Loading user...</div>
      <div *ngIf="error">Error: {{ error }}</div>
      <div *ngIf="user">
        <h1>{{ user.name }}</h1>
        {{ user.email }}
        <button [routerLink]="['/users', user.id, 'edit']">Edit User</button>
    </div>
})
export class UserDetailComponent implements OnInit, OnDestroy {
 user: any = null;
  loading = false;
  error: string | null = null;
  private destroy$ = new Subject<void>();
  constructor(private route: ActivatedRoute, private router: Router) {}
```

```
ngOnInit(): void {
   this.route.params.pipe(takeUntil(this.destroy$)).subscribe((params) => {
      const userId = params["id"];
      if (userId) {
       this.fetchUser(userId);
   });
 ngOnDestroy(): void {
   this.destroy$.next();
   this.destroy$.complete();
 }
  private async fetchUser(userId: string): Promise<void> {
   try {
      this.loading = true;
      this.error = null;
      const response = await fetch(`/api/users/${userId}`);
      if (response.ok) {
       this.user = await response.json();
      } else {
        throw new Error("User not found");
      }
    } catch (error) {
     this.error = error.message;
    } finally {
      this.loading = false;
   }
  }
 goBack(): void {
   window.history.back();
 }
}
```

# ♣ Performance Optimization

#### React Performance

```
{p>{user.email}
        <div className="user-actions">
          <button onClick={() => onEdit(user.id)}>Edit</button>
          <button onClick={() => onDelete(user.id)}>Delete/button>
        </div>
      </div>
   );
 },
  (prevProps, nextProps) => {
   // Custom comparison function
   return (
     prevProps.user.id === nextProps.user.id &&
     prevProps.user.name === nextProps.user.name &&
     prevProps.user.email === nextProps.user.email &&
     prevProps.user.avatar === nextProps.user.avatar
   );
 }
);
// 2. useMemo and useCallback
function UserList({ users, searchTerm, sortBy }) {
 // Memoize expensive calculations
 const filteredAndSortedUsers = useMemo(() => {
   console.log("Filtering and sorting users");
   let filtered = users.filter(
      (user) =>
        user.name.toLowerCase().includes(searchTerm.toLowerCase()) ||
        user.email.toLowerCase().includes(searchTerm.toLowerCase())
   );
   return filtered.sort((a, b) => {
     switch (sortBy) {
       case "name":
          return a.name.localeCompare(b.name);
       case "email":
          return a.email.localeCompare(b.email);
        case "date":
          return new Date(b.createdAt) - new Date(a.createdAt);
        default:
          return 0;
 }, [users, searchTerm, sortBy]);
 // Memoize callback functions
 const handleEdit = useCallback((userId) => {
   console.log("Edit user:", userId);
   // Edit logic here
 }, []);
 const handleDelete = useCallback((userId) => {
   console.log("Delete user:", userId);
   // Delete logic here
```

```
}, []);
 return (
    <div className="user-list">
      {filteredAndSortedUsers.map((user) => (
        <UserCard
          key={user.id}
          user={user}
          onEdit={handleEdit}
          onDelete={handleDelete}
       />
      ))}
    </div>
 );
}
// 3. Virtual scrolling for large lists
function VirtualizedUserList({ users }) {
 const [visibleRange, setVisibleRange] = useState({ start: 0, end: 20 });
 const containerRef = useRef(null);
 const itemHeight = 100;
 const containerHeight = 600;
 const visibleCount = Math.ceil(containerHeight / itemHeight);
 useEffect(() => {
   const container = containerRef.current;
   if (!container) return;
    const handleScroll = () => {
      const scrollTop = container.scrollTop;
      const start = Math.floor(scrollTop / itemHeight);
      const end = Math.min(start + visibleCount + 5, users.length);
      setVisibleRange({ start, end });
    };
    container.addEventListener("scroll", handleScroll);
    return () => container.removeEventListener("scroll", handleScroll);
  }, [users.length, itemHeight, visibleCount]);
  const visibleUsers = users.slice(visibleRange.start, visibleRange.end);
  const totalHeight = users.length * itemHeight;
 const offsetY = visibleRange.start * itemHeight;
 return (
    <div
      ref={containerRef}
      className="virtual-list-container"
      style={{ height: containerHeight, overflow: "auto" }}
      <div style={{ height: totalHeight, position: "relative" }}>
        <div
          style={{
            transform: `translateY(${offsetY}px)`,
```

```
position: "absolute",
            top: 0,
            left: 0,
            right: 0,
          }}
          {visibleUsers.map((user, index) => (
              key={user.id}
              style={{ height: itemHeight }}
              className="virtual-list-item"
              <UserCard user={user} />
            </div>
          ))}
        </div>
      </div>
    </div>
 );
}
// 4. Code splitting with React.lazy
const Dashboard = React.lazy(() => import("./components/Dashboard"));
const UserProfile = React.lazy(() => import("./components/UserProfile"));
const Settings = React.lazy(() => import("./components/Settings"));
function App() {
 return (
    <Router>
      <Suspense fallback={<div>Loading...</div>}>
        <Routes>
          <Route path="/dashboard" element={<Dashboard />} />
          <Route path="/profile" element={<UserProfile />} />
          <Route path="/settings" element={<Settings />} />
        </Routes>
      </Suspense>
   </Router>
 );
}
// 5. Error boundaries
class ErrorBoundary extends React.Component {
 constructor(props) {
   super(props);
   this.state = { hasError: false, error: null };
 }
 static getDerivedStateFromError(error) {
   return { hasError: true, error };
  }
  componentDidCatch(error, errorInfo) {
    console.error("Error caught by boundary:", error, errorInfo);
    // Log to error reporting service
```

#### **Vue Performance**

```
// 1. v-memo directive for expensive renders
<template>
    <div class="user-list">
        <div
            v-for="user in users"
            :key="user.id"
            v-memo="[user.id, user.name, user.email]"
            class="user-card"
            <img :src="user.avatar" :alt="user.name" />
            <h3>{{ user.name }}</h3>
            {{ user.email }}
        </div>
    </div>
</template>
// 2. Computed properties for expensive operations
export default {
    setup() {
        const users = ref([]);
        const searchTerm = ref('');
        const sortBy = ref('name');
        // Computed property automatically memoizes
        const filteredUsers = computed(() => {
            console.log('Filtering users');
            return users.value.filter(user =>
                user.name.toLowerCase().includes(searchTerm.value.toLowerCase())
```

```
);
        });
        const sortedUsers = computed(() => {
            console.log('Sorting users');
            return [...filteredUsers.value].sort((a, b) => {
                switch (sortBy.value) {
                    case 'name':
                         return a.name.localeCompare(b.name);
                    case 'email':
                         return a.email.localeCompare(b.email);
                    default:
                         return 0;
                }
            });
        });
        return {
            users,
            searchTerm,
            sortBy,
            sortedUsers
        };
    }
};
// 3. Async components
const AsyncDashboard = defineAsyncComponent({
    loader: () => import('./Dashboard.vue'),
    loadingComponent: LoadingSpinner,
    errorComponent: ErrorComponent,
    delay: 200,
    timeout: 3000
});
// 4. Virtual scrolling with vue-virtual-scroller
<template>
    <RecycleScroller
        class="scroller"
        :items="users"
        :item-size="100"
        key-field="id"
        v-slot="{ item }"
        <UserCard :user="item" />
    </RecycleScroller>
</template>
<script>
import { RecycleScroller } from 'vue-virtual-scroller';
import UserCard from './UserCard.vue';
export default {
    components: {
```

```
RecycleScroller,
    UserCard
},
props: {
    users: {
        type: Array,
        required: true
    }
};
</script>
```

# 

### React Pitfalls

```
// ★ BAD: Mutating state directly
function TodoList() {
 const [todos, setTodos] = useState([]);
 const addTodo = (text) => {
   // DON'T DO THIS - mutates state
   todos.push({ id: Date.now(), text });
   setTodos(todos);
 };
 const toggleTodo = (id) => {
   // DON'T DO THIS - mutates state
   const todo = todos.find((t) => t.id === id);
   todo.completed = !todo.completed;
   setTodos(todos);
 };
// ✓ GOOD: Creating new state objects
function TodoList() {
 const [todos, setTodos] = useState([]);
 const addTodo = (text) => {
   setTodos((prevTodos) => [
      ...prevTodos,
      { id: Date.now(), text, completed: false },
   ]);
 };
 const toggleTodo = (id) => {
    setTodos((prevTodos) =>
      prevTodos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
```

```
);
 };
}
// X BAD: Missing dependencies in useEffect
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetchUser(userId).then(setUser);
  }, []); // Missing userId dependency!
}
// ✓ GOOD: Including all dependencies
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  useEffect(() => {
    fetchUser(userId).then(setUser);
 }, [userId]); // Include userId dependency
}
// X BAD: Creating objects/functions in render
function UserList({ users }) {
  return (
    <div>
      {users.map((user) => (
        <UserCard
          key={user.id}
          user={user}
          onEdit={() => editUser(user.id)} // New function every render
          style={{ margin: 10 }} // New object every render
        />
      ))}
    </div>
 );
}
// ✓ GOOD: Memoizing callbacks and objects
function UserList({ users }) {
  const handleEdit = useCallback((userId) => {
    editUser(userId);
  }, []);
  const cardStyle = useMemo(() => ({ margin: 10 }), []);
  return (
    <div>
      {users.map((user) => (
        <UserCard
          key={user.id}
          user={user}
          onEdit={handleEdit}
          style={cardStyle}
```

```
/>
      ))}
   </div>
 );
}
// ★ BAD: Not cleaning up subscriptions
function ChatComponent() {
  const [messages, setMessages] = useState([]);
 useEffect(() => {
   const socket = io();
   socket.on("message", (message) => {
      setMessages((prev) => [...prev, message]);
   });
   // Missing cleanup!
 }, []);
}
// GOOD: Cleaning up subscriptions
function ChatComponent() {
 const [messages, setMessages] = useState([]);
 useEffect(() => {
    const socket = io();
    socket.on("message", (message) => {
      setMessages((prev) => [...prev, message]);
    });
    return () => {
      socket.disconnect();
   };
 }, []);
```

### **Vue Pitfalls**

```
emits: ['update-user'],
    setup(props, { emit }) {
        const updateUser = () => {
            emit('update-user', { ...props.user, name: 'New Name' });
        };
    }
};
// ★ BAD: Not using reactive properly
export default {
    setup() {
        let user = { name: 'John', age: 30 }; // Not reactive
        const updateAge = () => {
            user.age++; // Won't trigger reactivity
        };
        return { user, updateAge };
    }
};
// G00D: Using reactive/ref properly
export default {
    setup() {
        const user = reactive({ name: 'John', age: 30 });
        // OR
        const user = ref({ name: 'John', age: 30 });
        const updateAge = () => {
            user.age++; // Will trigger reactivity
            // OR for ref: user.value.age++;
        };
        return { user, updateAge };
    }
};
// X BAD: Memory leaks with watchers
export default {
    setup() {
        const count = ref(0);
        watch(count, (newVal) => {
            console.log('Count changed:', newVal);
        });
        // Watcher not cleaned up!
   }
};
// ✓ GOOD: Cleaning up watchers
export default {
    setup() {
        const count = ref(0);
```

```
const stopWatcher = watch(count, (newVal) => {
        console.log('Count changed:', newVal);
});

onUnmounted(() => {
        stopWatcher();
});
}
```

### **Angular Pitfalls**

```
// X BAD: Not unsubscribing from observables
@Component({
  selector: "app-user-list",
  template: `<div>{{ users.length }} users</div>`,
})
export class UserListComponent implements OnInit {
  users: User[] = [];
  constructor(private userService: UserService) {}
  ngOnInit(): void {
    // Memory leak - subscription not cleaned up
    this.userService.getUsers().subscribe((users) => {
      this.users = users;
    });
  }
}
// ✓ GOOD: Unsubscribing properly
@Component({
  selector: "app-user-list",
  template: `<div>{{ users.length }} users</div>`,
})
export class UserListComponent implements OnInit, OnDestroy {
  users: User[] = [];
  private destroy$ = new Subject<void>();
  constructor(private userService: UserService) {}
  ngOnInit(): void {
    this.userService
      .getUsers()
      .pipe(takeUntil(this.destroy$))
      .subscribe((users) => {
       this.users = users;
      });
  }
```

```
ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
 }
}
// X BAD: Mutating input properties
@Component({
  selector: "app-user-card",
  template: `
    <div>
      <h3>{{ user.name }}</h3>
      <button (click)="updateUser()">Update</button>
    </div>
})
export class UserCardComponent {
 @Input() user!: User;
 updateUser(): void {
   // DON'T DO THIS - mutates input
    this.user.name = "Updated Name";
 }
}
// ✓ GOOD: Emitting events
@Component({
  selector: "app-user-card",
  template: `
    <div>
      <h3>{{ user.name }}</h3>
      <button (click)="updateUser()">Update</button>
    </div>
})
export class UserCardComponent {
  @Input() user!: User;
  @Output() userUpdate = new EventEmitter<User>();
  updateUser(): void {
    const updatedUser = { ...this.user, name: "Updated Name" };
    this.userUpdate.emit(updatedUser);
  }
```

# Practice Problems

Mini Practice Problems

### 1. Real-time Chat Application

```
// Build a real-time chat app with:
// - User authentication
// - Multiple chat rooms
// - Message history
// - Online user status
// - Typing indicators
// - File sharing
// Key features to implement:
// - WebSocket connection management
// - Message state management
// - Optimistic updates
// - Error handling and reconnection
// - Message pagination
// - Emoji support
// Example structure:
const ChatApp = {
  components: [
    "ChatRoomList",
    "MessageList",
    "MessageInput",
    "UserList",
    "TypingIndicator",
  ],
  features: [
    "Real-time messaging",
    "User presence",
    "Message persistence",
    "File uploads",
    "Message search",
  ],
};
```

### 2. E-commerce Product Catalog

```
// Create a product catalog with:
// - Product listing with filters
// - Search functionality
// - Shopping cart
// - Wishlist
// - Product comparison
// - Reviews and ratings

// Key features to implement:
// - Advanced filtering (price, category, brand, ratings)
// - Infinite scrolling or pagination
// - Product image gallery
// - Cart state management
// - Local storage persistence
```

```
// - Responsive design

// Example structure:
const EcommerceApp = {
  pages: ["ProductList", "ProductDetail", "Cart", "Checkout", "UserProfile"],
  features: [
    "Product filtering",
    "Search with autocomplete",
    "Cart management",
    "Wishlist functionality",
    "Order history",
  ],
};
```

### 3. Task Management Dashboard

```
// Build a project management tool with:
// - Kanban board
// - Task creation and editing
// - Team collaboration
// - Time tracking
// - Progress analytics
// - File attachments
// Key features to implement:
// - Drag and drop functionality
// - Real-time collaboration
// - Task dependencies
// - Gantt chart view
// - Team member assignment
// - Notification system
// Example structure:
const TaskManager = {
 views: ["KanbanBoard", "ListView", "CalendarView", "GanttChart", "Analytics"],
 features: [
    "Drag and drop",
    "Real-time updates",
    "Task filtering",
    "Time tracking",
    "Team collaboration",
 ],
};
```

### 4. Social Media Feed

```
// Create a social media platform with:
// - User profiles
```

```
// - Post creation (text, images, videos)
// - News feed with infinite scroll
// - Like, comment, share functionality
// - Follow/unfollow users
// - Real-time notifications
// Key features to implement:
// - Image/video upload and preview
// - Feed algorithm (chronological/algorithmic)
// - Comment threading
// - Real-time likes and comments
// - User search and discovery
// - Content moderation
// Example structure:
const SocialMedia = {
  components: [
    "NewsFeed",
    "PostCreator",
    "UserProfile",
    "NotificationCenter",
    "SearchResults",
  ],
  features: [
    "Infinite scrolling",
    "Real-time updates",
    "Media uploads",
    "Social interactions",
    "User discovery",
  ],
};
```

#### 5. Data Visualization Dashboard

```
// Build an analytics dashboard with:
// - Multiple chart types (line, bar, pie, scatter)
// - Real-time data updates
// - Interactive filters
// - Export functionality
// - Responsive design
// - Custom date ranges

// Key features to implement:
// - Chart.js or D3.js integration
// - Data transformation and aggregation
// - Real-time WebSocket updates
// - CSV/PDF export
// - Dashboard customization
// - Performance optimization for large datasets
// Example structure:
```

```
const Dashboard = {
  charts: ["LineChart", "BarChart", "PieChart", "ScatterPlot", "HeatMap"],
 features: [
    "Real-time updates",
    "Interactive filtering",
    "Data export",
    "Custom layouts",
    "Performance optimization",
 ],
};
```

# Interview Notes

**Common Framework Questions** 

#### **React Interview Questions**

```
// Q1: What is the Virtual DOM and how does it work?
// A: The Virtual DOM is a JavaScript representation of the real DOM.
// React uses it to optimize updates by:
// 1. Creating a virtual representation of the UI
// 2. Comparing (diffing) the new virtual DOM with the previous one
// 3. Updating only the changed parts in the real DOM
// Q2: Explain the difference between useState and useReducer
// A: useState is for simple state, useReducer is for complex state logic
const [count, setCount] = useState(∅); // Simple state
const [state, dispatch] = useReducer(reducer, initialState); // Complex state
// Q3: What are React keys and why are they important?
// A: Keys help React identify which items have changed, are added, or removed
// They should be stable, predictable, and unique among siblings
// X BAD: Using array index as key
  items.map((item, index) => <Item key={index} data={item} />);
}
// ✓ GOOD: Using unique identifier
  items.map((item) => <Item key={item.id} data={item} />);
// Q4: Explain React's reconciliation process
// A: Reconciliation is the process React uses to update the DOM efficiently:
// 1. Element type comparison
// 2. Props comparison
// 3. Children comparison
// 4. Key-based reconciliation for lists
```

```
// Q5: What is prop drilling and how can you avoid it?
// A: Prop drilling is passing props through multiple component levels
// Solutions: Context API, State management libraries, Component composition
// Prop drilling example:
function App() {
 const user = { name: "John" };
 return <Parent user={user} />;
}
function Parent({ user }) {
 return <Child user={user} />;
}
function Child({ user }) {
 return <GrandChild user={user} />;
}
function GrandChild({ user }) {
 return <div>{user.name}</div>;
}
// Solution with Context:
const UserContext = createContext();
function App() {
 const user = { name: "John" };
  return (
    <UserContext.Provider value={user}>
      <Parent />
    </UserContext.Provider>
 );
}
function GrandChild() {
 const user = useContext(UserContext);
  return <div>{user.name}</div>;
}
```

#### **Vue Interview Questions**

```
// Q1: What is Vue's reactivity system?
// A: Vue uses a reactive system based on ES6 Proxies (Vue 3) or
Object.defineProperty (Vue 2)
// It automatically tracks dependencies and updates the DOM when data changes

// Q2: Explain the difference between ref and reactive
// A: ref is for primitive values, reactive is for objects
const count = ref(0); // Primitive
const user = reactive({ name: "John", age: 30 }); // Object
```

```
// Q3: What are Vue directives?
// A: Directives are special attributes that apply reactive behavior to the DOM
// Examples: v-if, v-for, v-model, v-show, v-on, v-bind
// Q4: Explain Vue's component lifecycle
// A: Vue 3 Composition API lifecycle hooks:
// - onBeforeMount
// - onMounted
// - onBeforeUpdate
// - onUpdated
// - onBeforeUnmount
// - onUnmounted
// Q5: What is the difference between computed and watch?
// A: Computed is for derived state, watch is for side effects
const fullName = computed(() => `${firstName.value} ${lastName.value}`);
watch(searchTerm, (newTerm) => {
  // Side effect: API call
  fetchSearchResults(newTerm);
});
```

### **Angular Interview Questions**

```
// Q1: What is dependency injection in Angular?
// A: DI is a design pattern where dependencies are provided to a class
// rather than the class creating them itself
@Injectable({
    providedIn: 'root'
})
export class UserService {
    constructor(private http: HttpClient) {}
}
@Component({
    selector: 'app-user'
})
export class UserComponent {
    constructor(private userService: UserService) {} // DI
}
// Q2: Explain Angular's change detection
// A: Angular uses Zone.js to detect changes and update the view
// It runs change detection on:
// - DOM events
// - HTTP requests
// - Timers (setTimeout, setInterval)
// Q3: What are Angular pipes?
```

```
// A: Pipes transform data in templates
{{ user.name | uppercase }}
{{ user.birthDate | date:'short' }}
{{ user.salary | currency: 'USD' }}
// Q4: Explain the difference between template-driven and reactive forms
// A: Template-driven forms use directives in templates,
// reactive forms use FormControl and FormGroup in components
// Template-driven:
<form #userForm="ngForm">
    <input name="name" ngModel required>
</form>
// Reactive:
this.userForm = this.fb.group({
    name: ['', Validators.required]
});
// Q5: What are Angular guards?
// A: Guards control navigation to/from routes
// Types: CanActivate, CanDeactivate, CanLoad, Resolve
```

## Company-Specific Questions

#### Frontend-Heavy Companies (Netflix, Airbnb, Facebook)

```
// Performance optimization questions:
// - How would you optimize a large list rendering?
// - Explain code splitting strategies
// - How do you handle memory leaks in SPAs?
// - What are your strategies for reducing bundle size?

// Architecture questions:
// - How would you structure a large-scale React application?
// - Explain your approach to state management
// - How do you handle error boundaries?
// - What's your testing strategy for components?

// Real-world scenarios:
// - How would you implement infinite scrolling?
// - Design a reusable component library
// - Handle real-time updates in a chat application
// - Implement offline functionality
```

### E-commerce Companies (Amazon, Shopify, eBay)

```
// Business logic questions:
// - How would you implement a shopping cart?
// - Design a product search with filters
// - Handle inventory updates in real-time
// - Implement A/B testing for UI components

// Performance questions:
// - Optimize product image loading
// - Handle large product catalogs
// - Implement caching strategies
// - Design for mobile performance
```

### Fintech Companies (Stripe, PayPal, Square)

```
// Security questions:
// - How do you handle sensitive data in frontend?
// - Implement secure form handling
// - Handle authentication tokens
// - Prevent XSS and CSRF attacks

// Reliability questions:
// - Handle network failures gracefully
// - Implement retry mechanisms
// - Design error handling strategies
// - Ensure data consistency
```

# **Key Takeaways**

- 1. **Choose the Right Tool**: Each framework has its strengths React for flexibility, Vue for ease of use, Angular for enterprise applications
- 2. **Performance Matters**: Always consider performance implications of your architectural decisions
- 3. State Management: Understand when to use local state vs global state management
- 4. Component Design: Focus on reusability, composability, and maintainability
- 5. **Testing Strategy**: Write testable components and implement proper testing strategies
- 6. **Security First**: Always consider security implications, especially for user data
- 7. **Developer Experience**: Choose tools and patterns that improve developer productivity
- 8. Stay Updated: Frontend frameworks evolve rapidly stay current with best practices

Next: Chapter 20 - Node.js & Backend Development @javascript // Without Framework - Vanilla JavaScript class VanillaTodoApp { constructor() { this.todos = []; this.container = document.getElementByld('app'); this.render(); }

```
addTodo(text) {
    this.todos.push({ id: Date.now(), text, completed: false });
    this.render(); // Manual re-render
}
toggleTodo(id) {
    const todo = this.todos.find(t => t.id === id);
    if (todo) {
        todo.completed = !todo.completed;
        this.render(); // Manual re-render
    }
}
render() {
    // Manual DOM manipulation
    this.container.innerHTML = `
        <div>
            <input id="todoInput" type="text" placeholder="Add todo...">
            <button
onclick="app.addTodo(document.getElementById('todoInput').value)">Add</button>
                ${this.todos.map(todo => `
                    <
                        <input type="checkbox" ${todo.completed ? 'checked' : ''}</pre>
                               onchange="app.toggleTodo(${todo.id})">
                        <span style="${todo.completed ? 'text-decoration: line-</pre>
through' : ''}">
                            ${todo.text}
                        </span>
                    `).join('')}
            </div>
    `;
}
```

// Framework Benefits: // 1. Declarative UI // 2. Component reusability // 3. State management // 4. Virtual DOM (React) // 5. Reactive data binding (Vue/Angular) // 6. Developer tools // 7. Community ecosystem

```
### Framework Comparison
```javascript
// Framework characteristics comparison
const frameworkComparison = {
    React: {
        paradigm: 'Component-based library',
```

```
learningCurve: 'Moderate',
        stateManagement: 'useState, useReducer, Redux, Zustand',
        rendering: 'Virtual DOM',
        strengths: ['Large ecosystem', 'Flexibility', 'Job market'],
        weaknesses: ['Boilerplate', 'Decision fatigue'],
        bestFor: 'Large applications, SPAs, mobile (React Native)'
    },
   Vue: {
        paradigm: 'Progressive framework',
        learningCurve: 'Easy',
        stateManagement: 'Vuex, Pinia',
        rendering: 'Virtual DOM with reactive system',
        strengths: ['Easy to learn', 'Great documentation', 'Template syntax'],
        weaknesses: ['Smaller ecosystem', 'Less job opportunities'],
        bestFor: 'Rapid prototyping, small to medium apps'
    },
   Angular: {
        paradigm: 'Full framework',
        learningCurve: 'Steep',
        stateManagement: 'Services, NgRx',
        rendering: 'Real DOM with change detection',
        strengths: ['Complete solution', 'TypeScript', 'Enterprise ready'],
        weaknesses: ['Complex', 'Heavy', 'Steep learning curve'],
        bestFor: 'Enterprise applications, large teams'
   }
};
```

### React Fundamentals

#### Components and JSX

```
// Functional Components
import React, { useState, useEffect } from "react";
// Basic functional component
function Welcome({ name }) {
  return <h1>Hello, {name}!</h1>;
}
// Component with state
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      Count: {count}
      <button onClick={() => setCount(count + 1)}>Increment/button>
      <button onClick={() => setCount(count - 1)}>Decrement/button>
    </div>
  );
```

```
// Component with effects
function UserProfile({ userId }) {
 const [user, setUser] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
 useEffect(() => {
    async function fetchUser() {
      try {
        setLoading(true);
        const response = await fetch(`/api/users/${userId}`);
        if (!response.ok) {
         throw new Error("Failed to fetch user");
        const userData = await response.json();
        setUser(userData);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    }
   fetchUser();
  }, [userId]); // Dependency array
 if (loading) return <div>Loading...</div>;
 if (error) return <div>Error: {error}</div>;
 if (!user) return <div>User not found</div>;
 return (
   <div className="user-profile">
      <img src={user.avatar} alt={user.name} />
      <h2>{user.name}</h2>
      {user.email}
    </div>
 );
}
```

#### **Custom Hooks**

```
// Custom hook for API calls
function useApi(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

useEffect(() => {
   async function fetchData() {
```

```
try {
        setLoading(true);
        setError(null);
        const response = await fetch(url);
        if (!response.ok) {
         throw new Error(`HTTP ${response.status}`);
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    }
   fetchData();
 }, [url]);
 return { data, loading, error };
}
// Custom hook for local storage
function useLocalStorage(key, initialValue) {
  const [storedValue, setStoredValue] = useState(() => {
   try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error("Error reading from localStorage:", error);
      return initialValue;
   }
 });
 const setValue = (value) => {
   try {
      const valueToStore =
        value instanceof Function ? value(storedValue) : value;
      setStoredValue(valueToStore);
     window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error("Error writing to localStorage:", error);
   }
 };
 return [storedValue, setValue];
}
// Custom hook for debounced values
function useDebounce(value, delay) {
 const [debouncedValue, setDebouncedValue] = useState(value);
 useEffect(() => {
    const handler = setTimeout(() => {
```

```
setDebouncedValue(value);
    }, delay);
    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);
 return debouncedValue;
}
// Usage of custom hooks
function SearchComponent() {
  const [searchTerm, setSearchTerm] = useState("");
  const debouncedSearchTerm = useDebounce(searchTerm, 300);
  const {
    data: searchResults,
    loading,
    error,
  } = useApi(
    debouncedSearchTerm ? `/api/search?q=${debouncedSearchTerm}` : null
  const [favorites, setFavorites] = useLocalStorage("favorites", []);
  return (
    <div>
      <input</pre>
       type="text"
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
        placeholder="Search..."
      />
      {loading && <div>Searching...</div>}
      {error && <div>Error: {error}</div>}
      {searchResults && (
        <l
          {searchResults.map((result) => (
            {result.title}
              <button onClick={() => setFavorites([...favorites, result])}>
                Add to Favorites
              </button>
           ))}
        ) }
    </div>
 );
}
```

#### Context API

```
// Theme context
const ThemeContext = React.createContext();
function ThemeProvider({ children }) {
 const [theme, setTheme] = useState("light");
 const toggleTheme = () => {
   setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
 };
 const value = {
   theme,
   toggleTheme,
 };
 return (
   <ThemeContext.Provider value={value}>{children}</ThemeContext.Provider>
 );
}
// Custom hook to use theme
function useTheme() {
 const context = useContext(ThemeContext);
 if (!context) {
   throw new Error("useTheme must be used within a ThemeProvider");
 }
 return context;
}
// User context with authentication
const UserContext = React.createContext();
function UserProvider({ children }) {
 const [user, setUser] = useState(null);
 const [loading, setLoading] = useState(true);
 useEffect(() => {
   // Check for existing session
   checkAuthStatus();
 }, []);
 const checkAuthStatus = async () => {
   try {
     const response = await fetch("/api/auth/me");
      if (response.ok) {
       const userData = await response.json();
        setUser(userData);
      }
    } catch (error) {
      console.error("Auth check failed:", error);
```

```
} finally {
      setLoading(false);
   }
 };
 const login = async (email, password) => {
    try {
      const response = await fetch("/api/auth/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ email, password }),
      });
     if (response.ok) {
       const userData = await response.json();
        setUser(userData);
       return { success: true };
      } else {
        return { success: false, error: "Invalid credentials" };
    } catch (error) {
      return { success: false, error: error.message };
   }
 };
 const logout = async () => {
   try {
      await fetch("/api/auth/logout", { method: "POST" });
    } catch (error) {
     console.error("Logout error:", error);
    } finally {
     setUser(null);
   }
 };
 const value = {
   user,
   loading,
   login,
   logout,
 };
 return <UserContext.Provider value={value}>{children}</UserContext.Provider>;
function useUser() {
 const context = useContext(UserContext);
 if (!context) {
   throw new Error("useUser must be used within a UserProvider");
 return context;
}
// App with providers
```

# Vue.js Essentials

#### **Vue Components**

```
// Vue 3 Composition API
import { ref, computed, onMounted, watch } from 'vue';
// Basic component
export default {
    name: 'Counter',
    setup() {
        const count = ref(0);
        const increment = () => {
            count.value++;
       };
        const decrement = () => {
            count.value--;
        };
        const doubleCount = computed(() => count.value * 2);
        return {
            count,
            increment,
            decrement,
            doubleCount
        };
    },
    template: `
        <div>
            Count: {{ count }}
            Double: {{ doubleCount }}
            <button @click="increment">+</button>
            <button @click="decrement">-</button>
```

```
</div>
};
// Component with lifecycle and watchers
export default {
    name: 'UserProfile',
    props: {
        userId: {
            type: String,
            required: true
        }
    },
    setup(props) {
        const user = ref(null);
        const loading = ref(false);
        const error = ref(null);
        const fetchUser = async (id) => {
            try {
                loading.value = true;
                error.value = null;
                const response = await fetch(`/api/users/${id}`);
                if (!response.ok) {
                    throw new Error('Failed to fetch user');
                }
                user.value = await response.json();
            } catch (err) {
                error.value = err.message;
            } finally {
                loading.value = false;
            }
        };
        // Watch for prop changes
        watch(() => props.userId, (newId) => {
            if (newId) {
                fetchUser(newId);
        }, { immediate: true });
        onMounted(() => {
            console.log('UserProfile mounted');
        });
        return {
            user,
            loading,
            error
        };
    },
    template: `
        <div class="user-profile">
            <div v-if="loading">Loading...</div>
```

#### Vue Composables (Custom Composition Functions)

```
// useApi composable
import { ref, watchEffect } from "vue";
export function useApi(url) {
 const data = ref(null);
 const loading = ref(false);
 const error = ref(null);
 const fetchData = async () => {
   try {
     loading.value = true;
      error.value = null;
      const response = await fetch(url.value || url);
      if (!response.ok) {
        throw new Error(`HTTP ${response.status}`);
      }
      data.value = await response.json();
    } catch (err) {
      error.value = err.message;
    } finally {
      loading.value = false;
   }
 };
 watchEffect(() => {
   if (url.value || url) {
     fetchData();
   }
 });
 return { data, loading, error, refetch: fetchData };
}
// useLocalStorage composable
import { ref, watch } from "vue";
export function useLocalStorage(key, defaultValue) {
  const storedValue = localStorage.getItem(key);
  const initial = storedValue ? JSON.parse(storedValue) : defaultValue;
```

```
const value = ref(initial);
  watch(
    value,
    (newValue) => {
      localStorage.setItem(key, JSON.stringify(newValue));
    },
    { deep: true }
  );
 return value;
}
// useDebounce composable
import { ref, watch } from "vue";
export function useDebounce(source, delay) {
  const debounced = ref(source.value);
 watch(source, (newValue) => {
    setTimeout(() => {
      debounced.value = newValue;
    }, delay);
 });
 return debounced;
}
// Usage in component
export default {
  name: "SearchComponent",
  setup() {
    const searchTerm = ref("");
    const debouncedSearch = useDebounce(searchTerm, 300);
    const searchUrl = computed(() =>
      debouncedSearch.value ? `/api/search?q=${debouncedSearch.value}` : null
    );
    const { data: searchResults, loading, error } = useApi(searchUrl);
    const favorites = useLocalStorage("favorites", []);
    const addToFavorites = (item) => {
      favorites.value.push(item);
    };
    return {
      searchTerm,
      searchResults,
      loading,
      error,
      favorites,
      addToFavorites,
    };
  },
```

#### Vue Store (Pinia)

```
// stores/user.js
import { defineStore } from "pinia";
export const useUserStore = defineStore("user", {
 state: () => ({
   user: null,
   loading: false,
   error: null,
 }),
 getters: {
   isAuthenticated: (state) => !!state.user,
   userName: (state) => state.user?.name || "Guest",
 },
 actions: {
   async login(email, password) {
     try {
       this.loading = true;
       this.error = null;
        const response = await fetch("/api/auth/login", {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify({ email, password }),
       });
        if (response.ok) {
         this.user = await response.json();
          return { success: true };
```

```
} else {
          throw new Error("Invalid credentials");
      } catch (error) {
        this.error = error.message;
        return { success: false, error: error.message };
      } finally {
       this.loading = false;
    },
    async logout() {
      try {
        await fetch("/api/auth/logout", { method: "POST" });
      } catch (error) {
        console.error("Logout error:", error);
      } finally {
       this.user = null;
       this.error = null;
      }
    },
    async checkAuth() {
     try {
        const response = await fetch("/api/auth/me");
        if (response.ok) {
          this.user = await response.json();
        }
      } catch (error) {
        console.error("Auth check failed:", error);
   },
 },
});
// stores/todos.js
import { defineStore } from "pinia";
export const useTodosStore = defineStore("todos", {
 state: () => ({
   todos: [],
   filter: "all", // all, active, completed
 }),
 getters: {
   filteredTodos: (state) => {
      switch (state.filter) {
        case "active":
          return state.todos.filter((todo) => !todo.completed);
        case "completed":
          return state.todos.filter((todo) => todo.completed);
        default:
          return state.todos;
```

```
},
    todoCount: (state) => state.todos.length,
    activeCount: (state) =>
      state.todos.filter((todo) => !todo.completed).length,
    completedCount: (state) =>
      state.todos.filter((todo) => todo.completed).length,
 },
 actions: {
    addTodo(text) {
      this.todos.push({
        id: Date.now(),
       text,
        completed: false,
        createdAt: new Date(),
     });
   },
    toggleTodo(id) {
      const todo = this.todos.find((t) => t.id === id);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
    removeTodo(id) {
      const index = this.todos.findIndex((t) => t.id === id);
      if (index > -1) {
       this.todos.splice(index, 1);
    },
    setFilter(filter) {
     this.filter = filter;
    },
    clearCompleted() {
      this.todos = this.todos.filter((todo) => !todo.completed);
   },
 },
});
```

### A Angular Basics

#### Components and Services

```
// user.service.ts
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
```

```
import { Observable, BehaviorSubject } from "rxjs";
import { map, catchError } from "rxjs/operators";
interface User {
  id: string;
 name: string;
  email: string;
  avatar?: string;
}
@Injectable({
  providedIn: "root",
})
export class UserService {
  private userSubject = new BehaviorSubject<User | null>(null);
  public user$ = this.userSubject.asObservable();
  constructor(private http: HttpClient) {
   this.checkAuthStatus();
  }
  login(
    email: string,
    password: string
  ): Observable<{ success: boolean; error?: string }> {
    return this.http.post<User>("/api/auth/login", { email, password }).pipe(
      map((user) => {
       this.userSubject.next(user);
       return { success: true };
      catchError((error) => {
        return [{ success: false, error: error.message }];
      })
    );
  logout(): Observable<void> {
    return this.http.post<void>("/api/auth/logout", {}).pipe(
      map(() => {
       this.userSubject.next(null);
      })
    );
  }
  private checkAuthStatus(): void {
    this.http.get<User>("/api/auth/me").subscribe({
      next: (user) => this.userSubject.next(user),
      error: () => this.userSubject.next(null),
    });
  }
  getCurrentUser(): User | null {
    return this.userSubject.value;
  }
```

```
isAuthenticated(): boolean {
    return !!this.userSubject.value;
 }
}
// user-profile.component.ts
import { Component, Input, OnInit, OnDestroy } from "@angular/core";
import { Subject } from "rxjs";
import { takeUntil } from "rxjs/operators";
import { UserService } from "./user.service";
@Component({
  selector: "app-user-profile",
  template: `
    <div class="user-profile" *ngIf="user; else loading">
      <img [src]="user.avatar" [alt]="user.name" />
      <h2>{{ user.name }}</h2>
      {{ user.email }}
      <button (click)="logout()" [disabled]="loggingOut">
        {{ loggingOut ? "Logging out..." : "Logout" }}
      </button>
    </div>
    <ng-template #loading>
      <div>Loading user profile...</div>
    </ng-template>
  styles: [
      .user-profile {
       display: flex;
        flex-direction: column;
        align-items: center;
        padding: 20px;
      }
      img {
       width: 100px;
        height: 100px;
        border-radius: 50%;
        margin-bottom: 10px;
      }
  ],
})
export class UserProfileComponent implements OnInit, OnDestroy {
  user: User | null = null;
  loggingOut = false;
  private destroy$ = new Subject<void>();
  constructor(private userService: UserService) {}
  ngOnInit(): void {
```

```
this.userService.user$.pipe(takeUntil(this.destroy$)).subscribe((user) => {
      this.user = user;
   });
  }
 ngOnDestroy(): void {
   this.destroy$.next();
    this.destroy$.complete();
  }
 logout(): void {
   this.loggingOut = true;
    this.userService
      .logout()
      .pipe(takeUntil(this.destroy$))
      .subscribe({
        next: () => {
          this.loggingOut = false;
        error: () => {
         this.loggingOut = false;
        },
      });
 }
}
```

#### **Angular Reactive Forms**

```
// login.component.ts
import { Component, OnInit } from "@angular/core";
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
import { Router } from "@angular/router";
import { UserService } from "./user.service";
@Component({
  selector: "app-login",
  template: `
    <form [formGroup]="loginForm" (ngSubmit)="onSubmit()" class="login-form">
      <h2>Login</h2>
      <div class="form-group">
        <label for="email">Email:</label>
        <input
          id="email"
          type="email"
          formControlName="email"
          [class.error]="email?.invalid && email?.touched"
        />
        <div *ngIf="email?.invalid && email?.touched" class="error-message">
          <span *ngIf="email?.errors?.['required']">Email is required</span>
          <span *ngIf="email?.errors?.['email']"</pre>
```

```
>Please enter a valid email</span
      </div>
    </div>
    <div class="form-group">
      <label for="password">Password:</label>
      <input</pre>
        id="password"
        type="password"
        formControlName="password"
        [class.error]="password?.invalid && password?.touched"
      />
      <div
        *ngIf="password?.invalid && password?.touched"
        class="error-message"
        <span *ngIf="password?.errors?.['required']"</pre>
          >Password is required</span
        <span *ngIf="password?.errors?.['minlength']"</pre>
          >Password must be at least 6 characters</span
      </div>
    </div>
    <button
      type="submit"
      [disabled]="loginForm.invalid || loading"
      class="submit-button"
      {{ loading ? "Logging in..." : "Login" }}
    </button>
    <div *ngIf="error" class="error-message">
      {{ error }}
    </div>
  </form>
styles: [
    .login-form {
     max-width: 400px;
      margin: 0 auto;
      padding: 20px;
    }
    .form-group {
      margin-bottom: 15px;
    }
    label {
      display: block;
      margin-bottom: 5px;
```

```
font-weight: bold;
      }
      input {
       width: 100%;
        padding: 8px;
        border: 1px solid #ddd;
        border-radius: 4px;
      }
      input.error {
        border-color: #ff0000;
      }
      .error-message {
        color: #ff0000;
       font-size: 14px;
        margin-top: 5px;
      .submit-button {
       width: 100%;
        padding: 10px;
        background-color: #007bff;
        color: white;
        border: none;
        border-radius: 4px;
        cursor: pointer;
      }
      .submit-button:disabled {
       background-color: #ccc;
        cursor: not-allowed;
      }
 ],
})
export class LoginComponent implements OnInit {
  loginForm!: FormGroup;
 loading = false;
 error: string | null = null;
 constructor(
    private fb: FormBuilder,
    private userService: UserService,
   private router: Router
  ) {}
 ngOnInit(): void {
   this.loginForm = this.fb.group({
      email: ["", [Validators.required, Validators.email]],
      password: ["", [Validators.required, Validators.minLength(6)]],
    });
  }
```

```
get email() {
  return this.loginForm.get("email");
}
get password() {
  return this.loginForm.get("password");
}
onSubmit(): void {
  if (this.loginForm.valid) {
    this.loading = true;
    this.error = null;
    const { email, password } = this.loginForm.value;
    this.userService.login(email, password).subscribe({
      next: (result) => {
        this.loading = false;
        if (result.success) {
         this.router.navigate(["/dashboard"]);
        } else {
          this.error = result.error || "Login failed";
        }
      },
      error: (error) => {
       this.loading = false;
        this.error = "An error occurred during login";
      },
    });
}
```

## State Management

#### Redux with React

```
// store/userSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

// Async thunks
export const loginUser = createAsyncThunk(
    'user/login',
    async ({ email, password }, { rejectWithValue }) => {
        try {
            const response = await fetch('/api/auth/login', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ email, password })
```

```
});
            if (!response.ok) {
                throw new Error('Invalid credentials');
            return await response.json();
        } catch (error) {
            return rejectWithValue(error.message);
    }
);
export const logoutUser = createAsyncThunk(
    'user/logout',
    async (_, { rejectWithValue }) => {
            await fetch('/api/auth/logout', { method: 'POST' });
        } catch (error) {
            return rejectWithValue(error.message);
    }
);
const userSlice = createSlice({
    name: 'user',
    initialState: {
        user: null,
        loading: false,
        error: null
    },
    reducers: {
        clearError: (state) => {
            state.error = null;
        },
        updateUser: (state, action) => {
            state.user = { ...state.user, ...action.payload };
    },
    extraReducers: (builder) => {
        builder
            // Login
            .addCase(loginUser.pending, (state) => {
                state.loading = true;
                state.error = null;
            })
            .addCase(loginUser.fulfilled, (state, action) => {
                state.loading = false;
                state.user = action.payload;
            })
            .addCase(loginUser.rejected, (state, action) => {
                state.loading = false;
                state.error = action.payload;
            })
```

```
// Logout
            .addCase(logoutUser.fulfilled, (state) => {
                state.user = null;
                state.error = null;
            });
    }
});
export const { clearError, updateUser } = userSlice.actions;
export default userSlice.reducer;
// store/index.js
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './userSlice';
import todosReducer from './todosSlice';
export const store = configureStore({
    reducer: {
        user: userReducer,
        todos: todosReducer
    },
    middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware({
            serializableCheck: {
                ignoredActions: ['persist/PERSIST']
            }
        })
});
export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
// hooks/redux.ts
import { useDispatch, useSelector, TypedUseSelectorHook } from 'react-redux';
import type { RootState, AppDispatch } from '../store';
export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;
// components/LoginForm.jsx
import React, { useState } from 'react';
import { useAppDispatch, useAppSelector } from '../hooks/redux';
import { loginUser, clearError } from '../store/userSlice';
function LoginForm() {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const dispatch = useAppDispatch();
    const { loading, error } = useAppSelector(state => state.user);
    const handleSubmit = async (e) => {
        e.preventDefault();
        dispatch(clearError());
```

```
const result = await dispatch(loginUser({ email, password }));
        if (loginUser.fulfilled.match(result)) {
            // Login successful
            console.log('Login successful');
    };
    return (
        <form onSubmit={handleSubmit}>
            <input</pre>
                type="email"
                value={email}
                onChange={(e) => setEmail(e.target.value)}
                placeholder="Email"
                required
            />
            <input</pre>
                type="password"
                value={password}
                onChange={(e) => setPassword(e.target.value)}
                placeholder="Password"
                required
            />
            <button type="submit" disabled={loading}>
                {loading ? 'Logging in...' : 'Login'}
            </button>
            {error && <div className="error">{error}</div>}
        </form>
    );
}
```

#### Zustand (Lightweight State Management)

```
body: JSON.stringify({ email, password })
                    });
                    if (!response.ok) {
                        throw new Error('Invalid credentials');
                    }
                    const user = await response.json();
                    set({ user, loading: false });
                    return { success: true };
                } catch (error) {
                    set({ error: error.message, loading: false });
                    return { success: false, error: error.message };
                }
            },
            logout: async () => {
                try {
                    await fetch('/api/auth/logout', { method: 'POST' });
                } catch (error) {
                    console.error('Logout error:', error);
                } finally {
                    set({ user: null, error: null });
                }
            },
            updateUser: (updates) => {
                set((state) => ({
                    user: state.user ? { ...state.user, ...updates } : null
                }));
            },
            clearError: () => set({ error: null })
        }),
        {
            name: 'user-storage',
            partialize: (state) => ({ user: state.user }) // Only persist user
data
        }
    )
);
export default useUserStore;
// stores/useTodosStore.js
import { create } from 'zustand';
import { immer } from 'zustand/middleware/immer';
const useTodosStore = create(
    immer((set) => ({
        todos: [],
        filter: 'all',
        addTodo: (text) => set((state) => {
```

```
state.todos.push({
                id: Date.now(),
                text,
                completed: false,
                createdAt: new Date()
            });
        }),
        toggleTodo: (id) => set((state) => {
            const todo = state.todos.find(t => t.id === id);
            if (todo) {
                todo.completed = !todo.completed;
            }
        }),
        removeTodo: (id) => set((state) => {
            const index = state.todos.findIndex(t => t.id === id);
            if (index > -1) {
                state.todos.splice(index, 1);
            }
        }),
        setFilter: (filter) => set({ filter }),
        clearCompleted: () => set((state) => {
            state.todos = state.todos.filter(todo => !todo.completed);
        }),
        // Computed values
        get filteredTodos() {
            const { todos, filter } = useTodosStore.getState();
            switch (filter) {
                case 'active':
                    return todos.filter(todo => !todo.completed);
                case 'completed':
                    return todos.filter(todo => todo.completed);
                default:
                    return todos;
            }
        },
        get stats() {
            const { todos } = useTodosStore.getState();
            return {
                total: todos.length,
                active: todos.filter(t => !t.completed).length,
                completed: todos.filter(t => t.completed).length
            };
        }
   }))
);
export default useTodosStore;
```

```
// Usage in component
function TodoApp() {
    const {
        todos,
        filter,
        addTodo,
        toggleTodo,
        removeTodo,
        setFilter,
        clearCompleted,
        filteredTodos,
        stats
    } = useTodosStore();
    const [newTodo, setNewTodo] = useState('');
    const handleSubmit = (e) => {
        e.preventDefault();
        if (newTodo.trim()) {
            addTodo(newTodo.trim());
            setNewTodo('');
        }
    };
    return (
        <div className="todo-app">
            <form onSubmit={handleSubmit}>
                <input</pre>
                    value={newTodo}
                    onChange={(e) => setNewTodo(e.target.value)}
                    placeholder="Add a new todo..."
                <button type="submit">Add</button>
            </form>
            <div className="filters">
                <button
                    onClick={() => setFilter('all')}
                    className={filter === 'all' ? 'active' : ''}
                    All ({stats.total})
                </button>
                <button
                    onClick={() => setFilter('active')}
                    className={filter === 'active' ? 'active' : ''}
                    Active ({stats.active})
                </button>
                <button
                    onClick={() => setFilter('completed')}
                    className={filter === 'completed' ? 'active' : ''}
                    Completed ({stats.completed})
                </button>
```

```
</div>
         {filteredTodos.map(todo => (
                ''}>
                   <input</pre>
                      type="checkbox"
                      checked={todo.completed}
                      onChange={() => toggleTodo(todo.id)}
                   />
                   <span>{todo.text}</span>
                   <button onClick={() =>
removeTodo(todo.id)}>Delete</button>
                ))}
         {stats.completed > 0 && (
             <button onClick={clearCompleted}>
                Clear Completed ({stats.completed})
             </button>
         )}
      </div>
   );
}
```

## Tomponent Architecture

#### **Design Patterns**

```
// 1. Container/Presentational Pattern
// Container Component (Smart)
function UserListContainer() {
 const [users, setUsers] = useState([]);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
 useEffect(() => {
   fetchUsers();
 }, []);
 const fetchUsers = async () => {
   try {
     setLoading(true);
      const response = await fetch("/api/users");
     const userData = await response.json();
     setUsers(userData);
   } catch (err) {
      setError(err.message);
```

```
} finally {
      setLoading(false);
    }
  };
  const handleUserDelete = async (userId) => {
    try {
      await fetch(`/api/users/${userId}`, { method: "DELETE" });
      setUsers(users.filter((user) => user.id !== userId));
    } catch (err) {
      setError(err.message);
    }
  };
  return (
    <UserListPresentation</pre>
      users={users}
      loading={loading}
      error={error}
      onUserDelete={handleUserDelete}
      onRetry={fetchUsers}
   />
  );
}
// Presentational Component (Dumb)
function UserListPresentation({
  users,
  loading,
  error,
  onUserDelete,
  onRetry,
}) {
  if (loading) return <LoadingSpinner />;
  if (error) return <ErrorMessage message={error} onRetry={onRetry} />;
  return (
    <div className="user-list">
      <h2>Users ({users.length})</h2>
      {users.map((user) => (
        <UserCard
          key={user.id}
          user={user}
          onDelete={() => onUserDelete(user.id)}
        />
      ))}
    </div>
  );
}
// 2. Higher-Order Component (HOC) Pattern
function withLoading(WrappedComponent) {
  return function WithLoadingComponent(props) {
    if (props.loading) {
```

```
return <LoadingSpinner />;
   return <WrappedComponent {...props} />;
 };
function withErrorHandling(WrappedComponent) {
 return function WithErrorHandlingComponent(props) {
   if (props.error) {
      return <ErrorMessage message={props.error} />;
   }
   return <WrappedComponent {...props} />;
 };
}
// Usage
const EnhancedUserList = withLoading(withErrorHandling(UserListPresentation));
// 3. Render Props Pattern
function DataFetcher({ url, children }) {
 const [data, setData] = useState(null);
 const [loading, setLoading] = useState(true);
 const [error, setError] = useState(null);
 useEffect(() => {
   fetchData();
 }, [url]);
 const fetchData = async () => {
   try {
      setLoading(true);
      setError(null);
      const response = await fetch(url);
      const result = await response.json();
      setData(result);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
   }
 };
 return children({ data, loading, error, refetch: fetchData });
}
// Usage
function App() {
 return (
    <DataFetcher url="/api/users">
      {({ data: users, loading, error, refetch }) => (
        <div>
          {loading && <div>Loading...</div>}
          {error && (
            <div>
```

```
Error: {error} <button onClick={refetch}>Retry</button>
            </div>
          )}
          {users && (
            <u1>
              {users.map((user) => (
               {user.name}
              ))}
            )}
        </div>
     ) }
    </DataFetcher>
 );
}
// 4. Compound Component Pattern
function Accordion({ children, ...props }) {
  const [openItems, setOpenItems] = useState(new Set());
 const toggleItem = (id) => {
   const newOpenItems = new Set(openItems);
   if (newOpenItems.has(id)) {
     newOpenItems.delete(id);
    } else {
     newOpenItems.add(id);
   setOpenItems(newOpenItems);
 };
 return (
    <div className="accordion" {...props}>
      {React.Children.map(children, (child, index) => {
        if (React.isValidElement(child)) {
          return React.cloneElement(child, {
            isOpen: openItems.has(index),
           onToggle: () => toggleItem(index),
           index,
         });
       return child;
      })}
    </div>
 );
}
function AccordionItem({ title, children, isOpen, onToggle }) {
 return (
    <div className="accordion-item">
      <button className="accordion-header" onClick={onToggle}>
        {title}
        <span className={`accordion-icon ${isOpen ? "open" : ""}`}>▼</span>
      </button>
      {isOpen && <div className="accordion-content">{children}</div>}
```

```
</div>
 );
}
// Usage
function FAQ() {
 return (
   <Accordion>
      <AccordionItem title="What is React?">
        React is a JavaScript library for building user interfaces.
      </AccordionItem>
      <AccordionItem title="How do I get started?">
        You can start by creating a new React app using Create React App.
      </AccordionItem>
      <AccordionItem title="What are components?">
         Components are reusable pieces of UI that can accept props and manage
         state.
       </AccordionItem>
    </Accordion>
 );
}
```

#### **Component Composition**

```
// Flexible Layout Components
function Layout({ children, sidebar, header, footer }) {
 return (
    <div className="layout">
      {header && <header className="layout-header">{header}</header>}
      <div className="layout-body">
        {sidebar && <aside className="layout-sidebar">{sidebar}</aside>}
        <main className="layout-main">{children}</main>
      </div>
      {footer && <footer className="layout-footer">{footer}</footer>}
    </div>
  );
}
// Flexible Card Component
function Card({ children, title, actions, variant = "default", ...props }) {
  return (
    <div className={`card card--${variant}`} {...props}>
      {title && (
        <div className="card-header">
          <h3 className="card-title">{title}</h3>
          {actions && <div className="card-actions">{actions}</div>}
        </div>
      )}
      <div className="card-content">{children}</div>
```

```
</div>
 );
}
// Modal Component with Portal
function Modal({ isOpen, onClose, title, children, size = "medium" }) {
 useEffect(() => {
    const handleEscape = (e) => {
      if (e.key === "Escape") {
       onClose();
     }
   };
    if (isOpen) {
     document.addEventListener("keydown", handleEscape);
      document.body.style.overflow = "hidden";
    }
    return () => {
      document.removeEventListener("keydown", handleEscape);
      document.body.style.overflow = "unset";
   };
  }, [isOpen, onClose]);
 if (!isOpen) return null;
 return ReactDOM.createPortal(
    <div className="modal-overlay" onClick={onClose}>
        className={`modal modal--${size}`}
        onClick={(e) => e.stopPropagation()}
        <div className="modal-header">
          <h2 className="modal-title">{title}</h2>
          <button className="modal-close" onClick={onClose}>
          </button>
        </div>
        <div className="modal-content">{children}</div>
      </div>
    </div>,
    document.body
 );
}
// Form Components
function Form({ onSubmit, children, ...props }) {
 const handleSubmit = (e) => {
    e.preventDefault();
   const formData = new FormData(e.target);
    const data = Object.fromEntries(formData.entries());
   onSubmit(data);
 };
```

```
return (
    <form onSubmit={handleSubmit} {...props}>
      {children}
   </form>
 );
}
function FormField({ label, error, children, required, ...props }) {
 return (
    <div className="form-field">
      {label && (
        <label className="form-label">
          {label}
          {required && <span className="required">*</span>}
        </label>
      )}
      {children}
      {error && <div className="form-error">{error}</div>}
 );
}
function Input({ type = "text", ...props }) {
 return <input type={type} className="form-input" {...props} />;
}
// Usage Example
function UserForm({ user, onSubmit }) {
 const [errors, setErrors] = useState({});
 const handleSubmit = (data) => {
   // Validation
    const newErrors = {};
   if (!data.name) newErrors.name = "Name is required";
   if (!data.email) newErrors.email = "Email is required";
   if (Object.keys(newErrors).length > 0) {
      setErrors(newErrors);
      return;
    }
    setErrors({});
   onSubmit(data);
 };
 return (
    <Card title="User Information">
      <Form onSubmit={handleSubmit}>
        <FormField label="Name" error={errors.name} required>
          <Input name="name" defaultValue={user?.name} />
        </FormField>
        <FormField label="Email" error={errors.email} required>
          <Input type="email" name="email" defaultValue={user?.email} />
```

2025-07-22 javascript.md

```
</FormField>
        <FormField label="Bio">
          <textarea
            name="bio"
            className="form-input"
            defaultValue={user?.bio}
          />
        </FormField>
        <button type="submit" className="btn btn-primary">
          {user ? "Update" : "Create"} User
        </button>
      </Form>
    </Card>
  );
}
```

# Chapter 20: Node.js & Backend Development 🜮



### Table of Contents

- Introduction to Node.js
- Setting Up Node.js Environment
- Core Modules
- Express.js Framework
- Database Integration
- Authentication & Authorization
- API Development
- File Handling
- Real-time Communication
- Testing Backend Applications
- Deployment & Production
- Common Pitfalls
- Practice Problems
- Interview Notes

### ☆ Introduction to Node.js

#### What is Node.js?

```
// Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine
// It allows you to run JavaScript on the server side
// Key Features:
// 1. Event-driven, non-blocking I/O
// 2. Single-threaded with event loop
```

```
// 3. NPM (Node Package Manager)
// 4. Cross-platform
// 5. Fast execution
// Event Loop Example
console.log("Start");
setTimeout(() => {
  console.log("Timeout callback");
}, 0);
setImmediate(() => {
  console.log("Immediate callback");
});
process.nextTick(() => {
 console.log("Next tick callback");
});
console.log("End");
// Output:
// Start
// End
// Next tick callback
// Immediate callback
// Timeout callback
```

#### Node.js vs Browser JavaScript

```
// Browser JavaScript
// - DOM manipulation
// - Window object
// - Limited file system access
// - Same-origin policy
// Node.js JavaScript
// - File system access
// - Network operations
// - Process management
// - Module system (CommonJS/ES Modules)
// Node.js Global Objects
console.log(__dirname); // Current directory
console.log(__filename); // Current file
console.log(process.env); // Environment variables
console.log(process.argv); // Command line arguments
// Module System
// CommonJS (traditional)
const fs = require("fs");
```

```
const path = require("path");

// ES Modules (modern)
import fs from "fs";
import path from "path";
```

## Setting Up Node.js Environment

### **Project Initialization**

```
# Initialize a new Node.js project
npm init -y

# Install dependencies
npm install express mongoose dotenv
npm install -D nodemon jest supertest

# Create basic project structure
mkdir src
mkdir src/controllers
mkdir src/models
mkdir src/routes
mkdir src/roites
mkdir src/middleware
mkdir src/utils
mkdir tests
```

### Package.json Configuration

```
"name": "my-node-app",
"version": "1.0.0",
"description": "A Node.js backend application",
"main": "src/app.js",
"type": "module",
"scripts": {
  "start": "node src/app.js",
  "dev": "nodemon src/app.js",
  "test": "jest",
  "test:watch": "jest --watch",
  "lint": "eslint src/",
  "build": "babel src -d dist"
},
"dependencies": {
  "express": "^4.18.2",
  "mongoose": "^7.5.0",
  "dotenv": "^16.3.1",
  "bcryptjs": "^2.4.3",
  "jsonwebtoken": "^9.0.2",
```

```
"cors": "^2.8.5",
    "helmet": "^7.0.0",
    "express-rate-limit": "^6.10.0"
},
    "devDependencies": {
        "nodemon": "^3.0.1",
        "jest": "^29.6.4",
        "supertest": "^6.3.3",
        "eslint": "^8.48.0"
}
```

#### **Environment Configuration**

```
// .env file
PORT=3000
MONGODB_URI=mongodb://localhost:27017/myapp
JWT_SECRET=your-super-secret-jwt-key
NODE_ENV=development
API_VERSION=v1
// config/database.js
import mongoose from 'mongoose';
import dotenv from 'dotenv';
dotenv.config();
const connectDB = async () => {
    try {
        const conn = await mongoose.connect(process.env.MONGODB_URI, {
            useNewUrlParser: true,
            useUnifiedTopology: true,
        });
        console.log(`MongoDB Connected: ${conn.connection.host}`);
    } catch (error) {
        console.error('Database connection error:', error.message);
        process.exit(1);
    }
};
export default connectDB;
// config/config.js
import dotenv from 'dotenv';
dotenv.config();
export const config = {
    port: process.env.PORT | 3000,
    mongoUri: process.env.MONGODB_URI,
```

```
jwtSecret: process.env.JWT_SECRET,
nodeEnv: process.env.NODE_ENV || 'development',
apiVersion: process.env.API_VERSION || 'v1'
};
```

# Core Modules

## File System (fs)

```
import fs from "fs";
import { promises as fsPromises } from "fs";
import path from "path";
// Synchronous file operations (blocking)
try {
 const data = fs.readFileSync("file.txt", "utf8");
 console.log(data);
} catch (error) {
 console.error("Error reading file:", error.message);
}
// Asynchronous file operations (non-blocking)
fs.readFile("file.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error reading file:", err.message);
    return;
  }
 console.log(data);
});
// Promise-based file operations
try {
 const data = await fsPromises.readFile("file.txt", "utf8");
 console.log(data);
} catch (error) {
  console.error("Error reading file:", error.message);
}
// File operations utility class
class FileManager {
  static async readJSON(filePath) {
    try {
      const data = await fsPromises.readFile(filePath, "utf8");
      return JSON.parse(data);
    } catch (error) {
      throw new Error(`Failed to read JSON file: ${error.message}`);
    }
  }
  static async writeJSON(filePath, data) {
```

```
try {
      const jsonData = JSON.stringify(data, null, 2);
      await fsPromises.writeFile(filePath, jsonData, "utf8");
    } catch (error) {
      throw new Error(`Failed to write JSON file: ${error.message}`);
    }
  }
  static async ensureDirectory(dirPath) {
    try {
      await fsPromises.access(dirPath);
    } catch (error) {
      await fsPromises.mkdir(dirPath, { recursive: true });
    }
  }
  static async deleteFile(filePath) {
    try {
      await fsPromises.unlink(filePath);
    } catch (error) {
      if (error.code !== "ENOENT") {
       throw error;
      }
    }
  }
}
// Usage
try {
  await FileManager.ensureDirectory("./uploads");
  const config = await FileManager.readJSON("./config.json");
 await FileManager.writeJSON("./backup.json", config);
} catch (error) {
  console.error("File operation failed:", error.message);
```

#### **HTTP Module**

```
import http from "http";
import url from "url";
import querystring from "querystring";

// Basic HTTP server
const server = http.createServer((req, res) => {
   const parsedUrl = url.parse(req.url, true);
   const { pathname, query } = parsedUrl;
   const method = req.method;

// Set CORS headers
   res.setHeader("Access-Control-Allow-Origin", "*");
   res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
```

2025-07-22 javascript.md

```
res.setHeader("Access-Control-Allow-Headers", "Content-Type, Authorization");
 // Handle preflight requests
 if (method === "OPTIONS") {
    res.writeHead(200);
   res.end();
   return;
 }
 // Route handling
 if (pathname === "/api/users" && method === "GET") {
   res.writeHead(200, { "Content-Type": "application/json" });
   res.end(JSON.stringify({ users: ["John", "Jane"] }));
  } else if (pathname === "/api/users" && method === "POST") {
   let body = "";
    req.on("data", (chunk) => {
      body += chunk.toString();
    });
    req.on("end", () => {
     try {
        const userData = JSON.parse(body);
        res.writeHead(201, { "Content-Type": "application/json" });
        res.end(JSON.stringify({ message: "User created", user: userData }));
      } catch (error) {
        res.writeHead(400, { "Content-Type": "application/json" });
        res.end(JSON.stringify({ error: "Invalid JSON" }));
      }
   });
  } else {
    res.writeHead(404, { "Content-Type": "application/json" });
    res.end(JSON.stringify({ error: "Not Found" }));
  }
});
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
 console.log(`Server running on port ${PORT}`);
});
```

# Testing Backend Applications

Unit Testing with Jest

```
// tests/services/userService.test.js
import UserService from "../../src/services/userService.js";
import User from "../../src/models/User.js";
import { jest } from "@jest/globals";
```

```
// Mock the User model
jest.mock("../../src/models/User.js");
describe("UserService", () => {
 beforeEach(() => {
   jest.clearAllMocks();
 });
 describe("createUser", () => {
   it("should create a new user successfully", async () => {
     const userData = {
       name: "John Doe",
       email: "john@example.com",
        password: "password123",
     };
     const mockUser = {
       _id: "user123",
       ...userData,
       save: jest.fn().mockResolvedValue(true),
     };
     User.mockImplementation(() => mockUser);
     const result = await UserService.createUser(userData);
      expect(User).toHaveBeenCalledWith(userData);
     expect(mockUser.save).toHaveBeenCalled();
      expect(result).toEqual(mockUser);
   });
   it("should throw error when user creation fails", async () => {
     const userData = {
       name: "John Doe",
       email: "john@example.com",
       password: "password123",
     };
      const mockUser = {
        save: jest.fn().mockRejectedValue(new Error("Database error")),
     };
     User.mockImplementation(() => mockUser);
      await expect(UserService.createUser(userData)).rejects.toThrow(
        "Failed to create user: Database error"
     );
   });
 });
 describe("getUserById", () => {
   it("should return user when found", async () => {
      const userId = "user123";
      const mockUser = {
```

```
_id: userId,
       name: "John Doe",
       email: "john@example.com",
     };
     User.findById.mockResolvedValue(mockUser);
      const result = await UserService.getUserById(userId);
      expect(User.findById).toHaveBeenCalledWith(userId);
     expect(result).toEqual(mockUser);
   });
   it("should throw error when user not found", async () => {
      const userId = "nonexistent";
     User.findById.mockResolvedValue(null);
      await expect(UserService.getUserById(userId)).rejects.toThrow(
        "Failed to get user: User not found"
     );
   });
 });
});
```

### **Integration Testing**

```
// tests/integration/auth.test.js
import request from "supertest";
import app from "../../src/app.js";
import User from "../../src/models/User.js";
import connectDB from "../../src/config/database.js";
import mongoose from "mongoose";
describe("Authentication Integration Tests", () => {
 beforeAll(async () => {
   await connectDB();
 });
 afterAll(async () => {
   await mongoose.connection.close();
 });
 beforeEach(async () => {
   await User.deleteMany({});
 });
 describe("POST /api/v1/auth/register", () => {
   it("should register a new user", async () => {
      const userData = {
        name: "John Doe",
```

```
email: "john@example.com",
      password: "password123",
    };
    const response = await request(app)
      .post("/api/v1/auth/register")
      .send(userData)
      .expect(201);
    expect(response.body.success).toBe(true);
    expect(response.body.user.email).toBe(userData.email);
    expect(response.body.accessToken).toBeDefined();
    expect(response.body.refreshToken).toBeDefined();
    // Verify user was created in database
    const user = await User.findOne({ email: userData.email });
    expect(user).toBeTruthy();
    expect(user.name).toBe(userData.name);
  it("should not register user with invalid email", async () => {
    const userData = {
      name: "John Doe",
      email: "invalid-email",
      password: "password123",
    };
    const response = await request(app)
      .post("/api/v1/auth/register")
      .send(userData)
      .expect(400);
    expect(response.body.success).toBe(false);
    expect(response.body.error).toContain("Validation failed");
 });
});
describe("POST /api/v1/auth/login", () => {
 let user;
  beforeEach(async () => {
    user = new User({
      name: "John Doe",
      email: "john@example.com",
      password: "password123",
    });
    await user.save();
 });
  it("should login with valid credentials", async () => {
    const response = await request(app)
      .post("/api/v1/auth/login")
      .send({
        email: "john@example.com",
```

```
password: "password123",
        })
        .expect(200);
      expect(response.body.message).toBe("Login successful");
      expect(response.body.accessToken).toBeDefined();
      expect(response.body.refreshToken).toBeDefined();
    });
    it("should not login with invalid credentials", async () => {
      const response = await request(app)
        .post("/api/v1/auth/login")
        .send({
          email: "john@example.com",
          password: "wrongpassword",
        })
        .expect(401);
      expect(response.body.error).toBe("Invalid credentials");
    });
 });
});
```

#### **API** Testing

```
// tests/api/users.test.js
import request from "supertest";
import app from "../../src/app.js";
import User from "../../src/models/User.js";
import jwt from "jsonwebtoken";
import { config } from "../../src/config/config.js";
describe("Users API", () => {
 let authToken;
  let adminToken;
 let testUser;
  let adminUser;
  beforeEach(async () => {
    await User.deleteMany({});
    // Create test user
    testUser = new User({
      name: "Test User",
      email: "test@example.com",
      password: "password123",
    });
    await testUser.save();
    // Create admin user
    adminUser = new User({
```

```
name: "Admin User",
    email: "admin@example.com",
    password: "password123",
    role: "admin",
  });
  await adminUser.save();
  // Generate tokens
  authToken = testUser.generateAuthToken();
  adminToken = adminUser.generateAuthToken();
});
describe("GET /api/v1/users", () => {
  it("should get all users", async () => {
    const response = await request(app).get("/api/v1/users").expect(200);
    expect(response.body.success).toBe(true);
    expect(response.body.data).toHaveLength(2);
    expect(response.body.pagination).toBeDefined();
  });
  it("should support pagination", async () => {
    const response = await request(app)
      .get("/api/v1/users?page=1&limit=1")
      .expect(200);
    expect(response.body.data).toHaveLength(1);
    expect(response.body.pagination.page).toBe(1);
    expect(response.body.pagination.limit).toBe(1);
    expect(response.body.pagination.total).toBe(2);
 });
});
describe("PUT /api/v1/users/:id", () => {
  it("should update user with valid token", async () => {
    const updateData = { name: "Updated Name" };
    const response = await request(app)
      .put(`/api/v1/users/${testUser._id}`)
      .set("Authorization", `Bearer ${authToken}`)
      .send(updateData)
      .expect(200);
    expect(response.body.success).toBe(true);
    expect(response.body.data.name).toBe("Updated Name");
  });
  it("should not update user without token", async () => {
    const updateData = { name: "Updated Name" };
    const response = await request(app)
      .put(`/api/v1/users/${testUser._id}`)
      .send(updateData)
      .expect(401);
```

```
expect(response.body.error).toBe("Access denied. No token provided.");
   });
 });
 describe("DELETE /api/v1/users/:id", () => {
   it("should delete user as admin", async () => {
      const response = await request(app)
        .delete(`/api/v1/users/${testUser._id}`)
        .set("Authorization", `Bearer ${adminToken}`)
        .expect(200);
     expect(response.body.success).toBe(true);
     expect(response.body.message).toBe("User deleted successfully");
   });
   it("should not delete user as regular user", async () => {
      const response = await request(app)
        .delete(`/api/v1/users/${adminUser._id}`)
        .set("Authorization", `Bearer ${authToken}`)
        .expect(403);
     expect(response.body.error).toBe("Insufficient permissions.");
   });
 });
});
```

# **Ø** Deployment & Production

## **Environment Configuration**

```
// config/production.js
export const productionConfig = {
 // Database
 mongoUri: process.env.MONGODB_URI,
 // Security
 jwtSecret: process.env.JWT_SECRET,
 bcryptRounds: 12,
 // Rate limiting
 rateLimitWindowMs: 15 * 60 * 1000, // 15 minutes
 rateLimitMax: 100,
 // CORS
  corsOrigin: process.env.FRONTEND URL?.split(",") || [],
 // Logging
  logLevel: "info",
```

```
// File uploads
  maxFileSize: 10 * 1024 * 1024, // 10MB
 // Session
  sessionSecret: process.env.SESSION_SECRET,
  sessionMaxAge: 24 * 60 * 60 * 1000, // 24 hours
 // Email
 emailService: process.env.EMAIL_SERVICE,
 emailUser: process.env.EMAIL_USER,
 emailPassword: process.env.EMAIL_PASSWORD,
};
// Dockerfile
FROM node:18-alpine
# Set working directory
WORKDIR /app
# Copy package files
COPY package*.json ./
# Install dependencies
RUN npm ci --only=production
# Copy source code
COPY src/ ./src/
COPY config/ ./config/
# Create non-root user
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nodejs -u 1001
# Change ownership
RUN chown -R nodejs:nodejs /app
USER nodejs
# Expose port
EXPOSE 3000
# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
   CMD node healthcheck.js
# Start application
CMD ["node", "src/app.js"]
*/
// docker-compose.yml
version: '3.8'
services:
```

```
app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - MONGODB_URI=mongodb://mongo:27017/myapp
      - JWT_SECRET=${JWT_SECRET}
    depends_on:
      - mongo
      - redis
    restart: unless-stopped
  mongo:
    image: mongo:6
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db
    restart: unless-stopped
  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    restart: unless-stopped
volumes:
  mongo data:
```

### Process Management with PM2

```
// ecosystem.config.js
module.exports = {
  apps: [
    {
      name: "my-app",
      script: "src/app.js",
      instances: "max", // Use all CPU cores
      exec_mode: "cluster",
      env: {
        NODE_ENV: "development",
        PORT: 3000,
      },
      env_production: {
        NODE_ENV: "production",
        PORT: 3000,
      },
      error_file: "./logs/err.log",
      out_file: "./logs/out.log",
```

```
log_file: "./logs/combined.log",
    time: true,
    max_memory_restart: "1G",
    node_args: "--max-old-space-size=1024",
    },
],
};

// Start with PM2
// pm2 start ecosystem.config.js --env production
// pm2 save
// pm2 startup
```

## 

#### 1. Blocking the Event Loop

```
// X Bad: Blocking operation
app.get("/bad", (req, res) => {
 // This blocks the event loop
 const result = heavyComputationSync();
 res.json({ result });
});
// ✓ Good: Non-blocking operation
app.get("/good", async (req, res) => {
 try {
   // Use worker threads or async operations
   const result = await heavyComputationAsync();
   res.json({ result });
 } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
// ✓ Better: Use worker threads for CPU-intensive tasks
import { Worker, isMainThread, parentPort } from "worker_threads";
if (isMainThread) {
  app.get("/worker", async (req, res) => {
   try {
      const worker = new Worker(__filename);
      worker.postMessage(req.body.data);
      worker.on("message", (result) => {
        res.json({ result });
      });
      worker.on("error", (error) => {
        res.status(500).json({ error: error.message });
```

```
});
} catch (error) {
    res.status(500).json({ error: error.message });
}
});
} else {
    parentPort.on("message", (data) => {
        const result = heavyComputationSync(data);
        parentPort.postMessage(result);
});
}
```

### 2. Memory Leaks

```
// X Bad: Memory leaks
const cache = new Map();
app.get("/leak", (req, res) => {
 // Cache grows indefinitely
 cache.set(req.url, new Date());
 res.json({ cached: cache.size });
});
// Good: Proper cache management
import NodeCache from "node-cache";
const cache = new NodeCache({
 stdTTL: 600, // 10 minutes
 maxKeys: 1000, // Maximum 1000 keys
});
app.get("/cached", (req, res) => {
 const key = req.url;
 let data = cache.get(key);
 if (!data) {
   data = generateData();
   cache.set(key, data);
 }
 res.json(data);
});
// X Bad: Event listener leaks
app.get("/events", (req, res) => {
 const emitter = new EventEmitter();
 // Listener never removed
 emitter.on("data", (data) => {
   console.log(data);
 });
```

```
res.json({ message: "Event listener added" });
});

//  Good: Proper cleanup
app.get("/events-clean", (req, res) => {
  const emitter = new EventEmitter();

  const handler = (data) => {
    console.log(data);
  };

  emitter.on("data", handler);

// Cleanup after use
  setTimeout(() => {
    emitter.removeListener("data", handler);
  }, 5000);

  res.json({ message: "Event listener added with cleanup" });
});
```

### 3. Improper Error Handling

```
// X Bad: Unhandled promise rejections
app.get("/bad-async", (req, res) => {
 // This can crash the application
 someAsyncOperation();
 res.json({ message: "Started operation" });
});
// ✓ Good: Proper error handling
app.get("/good-async", async (req, res) => {
   const result = await someAsyncOperation();
   res.json({ result });
 } catch (error) {
    console.error("Operation failed:", error);
    res.status(500).json({ error: "Operation failed" });
 }
});
// Global error handlers
process.on("uncaughtException", (error) => {
 console.error("Uncaught Exception:", error);
 process.exit(1);
});
process.on("unhandledRejection", (reason, promise) => {
  console.error("Unhandled Rejection at:", promise, "reason:", reason);
```

```
process.exit(1);
});
```

#### 4. Security Issues

```
// X Bad: SQL injection equivalent (NoSQL injection)
app.get("/users/:id", async (req, res) => {
 // Vulnerable to NoSQL injection
 const user = await User.findOne({ _id: req.params.id });
 res.json(user);
});
// Good: Input validation
import mongoose from "mongoose";
app.get("/users/:id", async (req, res) => {
 try {
   // Validate ObjectId
   if (!mongoose.Types.ObjectId.isValid(req.params.id)) {
      return res.status(400).json({ error: "Invalid user ID" });
    }
    const user = await User.findById(req.params.id);
   if (!user) {
      return res.status(404).json({ error: "User not found" });
   res.json(user);
 } catch (error) {
    res.status(500).json({ error: "Server error" });
});
// X Bad: Exposing sensitive information
app.get("/error-info", (req, res) => {
 try {
   throw new Error(
      "Database connection failed: mongodb://admin:password@localhost"
   );
 } catch (error) {
   // Exposes sensitive information
    res.status(500).json({ error: error.message });
 }
});
// ✓ Good: Safe error messages
app.get("/safe-error", (req, res) => {
 try {
   throw new Error(
      "Database connection failed: mongodb://admin:password@localhost"
    );
```

```
} catch (error) {
    // Log full error for debugging
    console.error("Database error:", error.message);

    // Send safe message to client
    res.status(500).json({ error: "Internal server error" });
}
});
```

## Practice Problems

## 1. Real-time Chat Application

```
// Build a real-time chat application with:
// - User authentication
// - Multiple chat rooms
// - Private messaging
// - Message history
// - Online user status
// - Typing indicators
// - File sharing
// - Message reactions

// Key features to implement:
// 1. WebSocket connection management
// 2. Room-based messaging
// 3. User presence tracking
// 4. Message persistence
// 5. Real-time notifications
```

#### 2. E-commerce API

```
// Create a comprehensive e-commerce API with:
// - Product catalog management
// - Shopping cart functionality
// - Order processing
// - Payment integration
// - Inventory management
// - User reviews and ratings
// - Search and filtering
// - Admin dashboard

// Key features to implement:
// 1. Complex database relationships
// 2. Transaction handling
// 3. Payment gateway integration
// 4. Email notifications
// 5. File upload for product images
```

#### 3. Task Management System

```
// Develop a task management system with:
// - Project organization
// - Task assignment
// - Due date tracking
// - Progress monitoring
// - Team collaboration
// - File attachments
// - Time tracking
// - Reporting and analytics
// Key features to implement:
// 1. Role-based access control
// 2. Real-time updates
// 3. Email notifications
// 4. Data visualization
// 5. Export functionality
```

## Interview Notes

## Common Node.js Questions

Q: What is the Event Loop in Node.js? A: The Event Loop is the core mechanism that allows Node.js to perform non-blocking I/O operations. It continuously checks the call stack and processes callbacks from the event queue when the stack is empty.

Q: Explain the difference between process.nextTick() and setImmediate(). A: process.nextTick() has higher priority and executes before any other asynchronous operations, while setImmediate() executes after I/O events in the current event loop phase.

Q: How do you handle errors in Node.js? A: Use try-catch for synchronous code, .catch() for promises, errorfirst callbacks for traditional async operations, and global error handlers for uncaught exceptions.

Q: What are streams in Node.js? A: Streams are objects that allow reading/writing data in chunks rather than loading everything into memory. Types include Readable, Writable, Duplex, and Transform streams.

Q: How do you scale a Node.js application? A: Use clustering to utilize multiple CPU cores, implement load balancing, use caching strategies, optimize database queries, and consider microservices architecture.

Company-Specific Questions

#### **Startup/Scale-up Companies:**

- How would you design a real-time notification system?
- Implement a rate limiting middleware
- Design a file upload system with progress tracking

#### **Enterprise Companies:**

- How do you ensure data consistency in distributed systems?
- Implement authentication with multiple providers
- Design a logging and monitoring system

#### **E-commerce Companies:**

- How would you handle high-traffic product launches?
- Implement a shopping cart with session management
- Design a payment processing system

# **©** Key Takeaways

- 1. Event-Driven Architecture: Master the event loop and non-blocking I/O
- 2. **Security First**: Always validate input and handle errors properly
- 3. **Performance**: Use clustering, caching, and proper database indexing
- 4. Testing: Write comprehensive unit, integration, and API tests
- 5. **Monitoring**: Implement logging, error tracking, and performance monitoring
- 6. Scalability: Design for horizontal scaling from the beginning
- 7. Best Practices: Follow security guidelines and coding standards
- 8. Real-time Features: Understand WebSockets and Socket.io for live updates

Remember: Node.js excels at I/O-intensive applications but requires careful handling of CPU-intensive tasks. Always consider the event loop and write non-blocking code!  $\mathcal{Q}$ 

```
## G Real-time Communication
### WebSocket with Socket.io
```javascript
// socket/socketHandler.js
import { Server } from 'socket.io';
import jwt from 'jsonwebtoken';
import { config } from '../config/config.js';
import User from '../models/User.js';
class SocketHandler {
    constructor(server) {
        this.io = new Server(server, {
            cors: {
                origin: process.env.FRONTEND_URL || 'http://localhost:3000',
                methods: ['GET', 'POST']
            }
        });
        this.connectedUsers = new Map();
        this.setupMiddleware();
```

```
this.setupEventHandlers();
}
setupMiddleware() {
    // Authentication middleware
    this.io.use(async (socket, next) => {
        try {
            const token = socket.handshake.auth.token;
            if (!token) {
                throw new Error('No token provided');
            }
            const decoded = jwt.verify(token, config.jwtSecret);
            const user = await User.findById(decoded.id);
            if (!user) {
                throw new Error('User not found');
            socket.userId = user._id.toString();
            socket.user = user;
            next();
        } catch (error) {
            next(new Error('Authentication failed'));
        }
    });
}
setupEventHandlers() {
    this.io.on('connection', (socket) => {
        console.log(`User ${socket.user.name} connected`);
        // Store connected user
        this.connectedUsers.set(socket.userId, {
            socketId: socket.id,
            user: socket.user,
            connectedAt: new Date()
        });
        // Notify others about user connection
        socket.broadcast.emit('user:online', {
            userId: socket.userId,
            name: socket.user.name
        });
        // Send list of online users
        socket.emit('users:online', this.getOnlineUsers());
        // Join user to their personal room
        socket.join(`user:${socket.userId}`);
        // Chat events
        this.setupChatEvents(socket);
```

```
// Notification events
        this.setupNotificationEvents(socket);
        // Handle disconnection
        socket.on('disconnect', () => {
            console.log(`User ${socket.user.name} disconnected`);
            // Remove from connected users
            this.connectedUsers.delete(socket.userId);
            // Notify others about user disconnection
            socket.broadcast.emit('user:offline', {
                userId: socket.userId,
                name: socket.user.name
            });
        });
    });
}
setupChatEvents(socket) {
    // Join chat room
    socket.on('chat:join', (roomId) => {
        socket.join(`chat:${roomId}`);
        socket.emit('chat:joined', { roomId });
    });
    // Leave chat room
    socket.on('chat:leave', (roomId) => {
        socket.leave(`chat:${roomId}`);
        socket.emit('chat:left', { roomId });
    });
    // Send message
    socket.on('chat:message', (data) => {
        const { roomId, message } = data;
        const messageData = {
            id: Date.now().toString(),
            roomId,
            message,
            sender: {
                id: socket.userId,
                name: socket.user.name
            },
            timestamp: new Date().toISOString()
        };
        // Send to all users in the room
        this.io.to(`chat:${roomId}`).emit('chat:message', messageData);
    });
    // Typing indicators
    socket.on('chat:typing', (data) => {
        const { roomId } = data;
```

```
socket.to(`chat:${roomId}`).emit('chat:typing', {
                userId: socket.userId,
                name: socket.user.name
            });
        });
        socket.on('chat:stop-typing', (data) => {
            const { roomId } = data;
            socket.to(`chat:${roomId}`).emit('chat:stop-typing', {
                userId: socket.userId
            });
        });
    }
    setupNotificationEvents(socket) {
        // Send notification to specific user
        socket.on('notification:send', (data) => {
            const { userId, notification } = data;
            const notificationData = {
                id: Date.now().toString(),
                ...notification,
                from: {
                    id: socket.userId,
                    name: socket.user.name
                },
                timestamp: new Date().toISOString()
            };
            this.io.to(`user:${userId}`).emit('notification:received',
notificationData);
        });
    }
    getOnlineUsers() {
        return Array.from(this.connectedUsers.values()).map(user => ({
            id: user.user._id,
            name: user.user.name,
            connectedAt: user.connectedAt
        }));
    }
    // Send notification to specific user (called from other parts of the app)
    sendNotificationToUser(userId, notification) {
        this.io.to(`user:${userId}`).emit('notification:received', {
            id: Date.now().toString(),
            ...notification,
            timestamp: new Date().toISOString()
        });
    }
    // Broadcast to all connected users
    broadcast(event, data) {
        this.io.emit(event, data);
```

```
}

export default SocketHandler;

// app.js integration
import http from 'http';
import SocketHandler from './socket/socketHandler.js';

// Create HTTP server
const server = http.createServer(app);

// Initialize Socket.io
const socketHandler = new SocketHandler(server);

// Make socket handler available globally
app.set('socketHandler', socketHandler);

server.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});

}
```