Comprehensive JavaScript Notes

Table of Contents

1. Introduction to JavaScript

What is JavaScript?

JavaScript is a high-level, interpreted programming language that conforms to the ECMAScript specification. It is a versatile language primarily known for adding interactivity to web pages, but has evolved to become one of the most widely used programming languages in the world. JavaScript enables dynamic content, controls multimedia, animates images, and pretty much everything else on the web.

Unlike HTML and CSS, which are markup and styling languages respectively, JavaScript is a full-fledged programming language that allows developers to implement complex features on web pages. When a web page does more than just sit there and display static information—displaying timely content updates, interactive maps, animated graphics, scrolling video jukeboxes, etc.—JavaScript is likely involved.

JavaScript runs on the client side of the web, which means it can be executed directly in the user's browser without needing to communicate with the server. This makes web applications faster and more responsive to user interactions.

History and Evolution of JavaScript

JavaScript was created by Brendan Eich in just 10 days in May 1995 while he was working at Netscape Communications Corporation. It was originally named "Mocha," then renamed to "LiveScript," and finally to "JavaScript" when Netscape and Sun Microsystems (the creators of Java) formed a license agreement.

Despite its name, JavaScript has very little to do with the Java programming language. The similar name was primarily a marketing decision, as Java was very popular at the time.

Here's a brief timeline of JavaScript's evolution:

- **1995**: JavaScript was created by Brendan Eich at Netscape
- **1996**: JavaScript was submitted to ECMA International for standardization
- **1997**: The first ECMAScript standard (ECMAScript 1) was released
- **1998**: ECMAScript 2 was released with minor changes
- **1999**: ECMAScript 3 introduced regular expressions, try/catch exception handling, and more
- **2000-2008**: Work on ECMAScript 4 was abandoned due to disagreements
- **2009**: ECMAScript 5 (ES5) was released with strict mode, JSON support, and more
- **2015**: ECMAScript 2015 (ES6) was released with major enhancements including classes, modules, arrow functions, promises, and more
- **2016-Present**: Annual releases (ES2016, ES2017, etc.) with incremental improvements

Each new version has added features and capabilities, making JavaScript more powerful and developer-friendly.

JavaScript vs ECMAScript

Many people use the terms "JavaScript" and "ECMAScript" interchangeably, but they're not exactly the same thing:

ECMAScript is the official name of the language specification. It's maintained by ECMA International (European Computer Manufacturers Association) and defines how the language should work. When people refer to "ES6" or "ES2015," they're talking about specific versions of the ECMAScript specification.

JavaScript is the most popular implementation of the ECMAScript specification. It's what actually runs in web browsers. While JavaScript follows the ECMAScript standard, it also

includes additional features that aren't part of the specification, particularly browser-specific APIs like the Document Object Model (DOM) for manipulating web pages.

Think of ECMAScript as the blueprint and JavaScript as the actual building constructed from that blueprint, with some additional features added by the builders (browser vendors).

Setting up a JavaScript Development Environment

Setting up a JavaScript development environment is relatively straightforward since you don't need to install any special software to start writing JavaScript code. Here's what you need:

Basic Setup:

1. **Text Editor or IDE**: You need a good code editor. Popular options include:
2. Visual Studio Code (free, highly recommended)
3. Sublime Text
4. Atom
5. WebStorm (paid, but powerful)
6. **Web Browser**: Modern browsers have built-in developer tools that are essential for JavaScript development:
7. Google Chrome
8. Mozilla Firefox
9. Microsoft Edge
10. Safari
11. **Browser Developer Tools**: Access them by right-clicking on a webpage and selecting "Inspect" or pressing F12. These tools allow you to:
12. Debug JavaScript code
13. View console output
14. Inspect HTML and CSS
15. Monitor network requests
16. Analyze performance

Advanced Setup:

For more complex projects, you might want to add:

1. **Node.js and npm**: Node.js allows you to run JavaScript outside the browser, and npm (Node Package Manager) helps you manage dependencies.
2. **Version Control**: Git is the standard for tracking code changes.
3. **Build Tools**: Tools like Webpack, Babel, or Parcel help bundle, transpile, and optimize your code.
4. **Linters and Formatters**: ESLint and Prettier help maintain code quality and consistency.
5. **Testing Frameworks**: Jest, Mocha, or Jasmine for writing and running tests.

How to Include JavaScript in HTML

There are several ways to include JavaScript in an HTML document:

1. Internal JavaScript (using the `<script>` tag):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Web Page</title>
    <script>
        // JavaScript code goes here
        function greet() {
            alert('Hello, World!');
        }
    </script>
</head>
<body>
    <button onclick="greet()">Click Me</button>
</body>
</html>
```

2. External JavaScript (linking to a separate .js file):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Web Page</title>
    <!-- Link to an external JavaScript file -->
    <script src="script.js"></script>
</head>
<body>
    <button onclick="greet()">Click Me</button>
</body>
</html>
```

And in script.js:

```
function greet() {
    alert('Hello, World!');
}
```

3. Inline JavaScript (directly in HTML elements):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Web Page</title>
</head>
<body>
    <!-- Inline JavaScript in an HTML attribute -->
    <button onclick="alert('Hello, World!')">Click Me</button>
</body>
</html>
```

Best Practices for Including JavaScript:

1. **Place scripts at the bottom** of the body tag when possible to improve page loading performance.
2. **Use external JavaScript files** for better organization, caching, and separation of concerns.

3. **Add the `defer` attribute** to load scripts after HTML parsing is complete but before the DOMContentLoaded event: ```html

```

1. **Use the `async` attribute** for scripts that don't depend on other scripts or DOM elements: ```html

```

1. **Avoid inline JavaScript** for better maintainability and security (helps with Content Security Policy).

By understanding these fundamentals, you're ready to start your journey into JavaScript programming. The language's flexibility, ubiquity, and continuous evolution make it an essential skill for web developers and increasingly for other types of software development as well.

2. JavaScript Fundamentals

Syntax and Basic Constructs

JavaScript syntax is the set of rules that define how JavaScript programs are constructed. Understanding the basic syntax is essential for writing valid JavaScript code.

Statements and Semicolons

JavaScript statements are commands to the browser/interpreter. Each statement should be terminated with a semicolon (;), although JavaScript has automatic semicolon insertion (ASI).

```
let greeting = "Hello";  // This is a statement
console.log(greeting);   // This is another statement
```

While semicolons are technically optional in many cases due to ASI, it's considered good practice to include them to avoid potential issues.

Case Sensitivity

JavaScript is case-sensitive. Variables, function names, and other identifiers must be typed with consistent capitalization.

```
let name = "John";
// NAME and Name are different variables than name
console.log(NAME);  // ReferenceError: NAME is not defined
```

Whitespace and Line Breaks

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. You can use whitespace to format your code for better readability.

```
// These are equivalent:
let sum = a + b;
let sum=a+b;
```

Comments

JavaScript supports single-line and multi-line comments:

```
// This is a single-line comment

/* This is a
   multi-line comment */
```

Variables and Data Types

Variables are containers for storing data values. In JavaScript, there are three ways to declare variables:

Variable Declaration

```
var oldWay = "I'm the old way";      // Function-scoped, can be redeclar
let modernWay = "I'm recommended";   // Block-scoped, cannot be redeclar
const constant = "I cannot change";  // Block-scoped, cannot be reassigr
```

var

- Function-scoped (or globally-scoped if declared outside a function)
- Can be redeclared and updated
- Hoisted to the top of its scope and initialized with undefined

let

- Block-scoped (available only within the block it's defined in)
- Can be updated but not redeclared in the same scope
- Hoisted to the top of its block but not initialized

const

- Block-scoped
- Cannot be updated or redeclared
- Must be initialized at declaration
- For objects and arrays, the content can still be modified

Primitive Data Types

JavaScript has seven primitive data types:

1. **String**: Represents textual data `javascript let name = "John"; let greeting = 'Hello'; let template = `Hello, ${name}`; // Template literal (ES6)`
2. **Number**: Represents both integer and floating-point numbers `javascript let integer = 42; let float = 3.14; let scientific = 2.998e8; // Scientific notation let infinity = Infinity; let notANumber = NaN; // Result of invalid calculations`
3. **Boolean**: Represents logical entities with two values: true and false `javascript let isActive = true; let isComplete = false;`
4. **Undefined**: Represents a variable that has been declared but not assigned a value `javascript let undefinedVar; console.log(undefinedVar); // undefined`
5. **Null**: Represents the intentional absence of any object value `javascript let emptyValue = null;`
6. **Symbol** (ES6): Represents a unique and immutable primitive value `javascript let sym1 = Symbol(); let sym2 = Symbol('description');`

7. **BigInt** (ES2020): Represents integers of arbitrary length `javascript let bigNumber = 9007199254740991n; let anotherBigNumber = BigInt("9007199254740991");`

Object Type

In addition to primitive types, JavaScript has an Object type that represents a collection of properties:

```
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    isEmployed: true
};

// Accessing object properties
console.log(person.firstName);  // Dot notation
console.log(person["lastName"]); // Bracket notation
```

Arrays, functions, and dates are all specialized types of objects in JavaScript.

```
// Array
let colors = ["red", "green", "blue"];

// Function
function greet() {
    return "Hello!";
}

// Date
let today = new Date();
```

Operators

JavaScript includes various operators for performing operations on values.

Arithmetic Operators

Used to perform mathematical operations:

```
let a = 10;
let b = 3;

console.log(a + b);  // Addition: 13
console.log(a - b);  // Subtraction: 7
console.log(a * b);  // Multiplication: 30
console.log(a / b);  // Division: 3.3333...
console.log(a % b);  // Modulus (remainder): 1
console.log(a ** b); // Exponentiation (ES2016): 1000
```

Increment and decrement:

```
let counter = 5;
counter++;  // Increment: counter is now 6
counter--;  // Decrement: counter is now 5
```

Assignment Operators

Used to assign values to variables:

```
let x = 10;  // Basic assignment

// Compound assignment
x += 5;   // x = x + 5 (15)
x -= 3;   // x = x - 3 (12)
x *= 2;   // x = x * 2 (24)
x /= 4;   // x = x / 4 (6)
x %= 4;   // x = x % 4 (2)
x **= 3;  // x = x ** 3 (8)
```

Comparison Operators

Used to compare values:

```
let a = 5;
let b = "5";

console.log(a == b);   // Equal (value): true
console.log(a === b);  // Strict equal (value and type): false
console.log(a != b);   // Not equal (value): false
```

```
console.log(a !== b);  // Strict not equal (value and type): true
console.log(a > 3);    // Greater than: true
console.log(a < 10);   // Less than: true
console.log(a >= 5);   // Greater than or equal: true
console.log(a <= 4);   // Less than or equal: false
```

Logical Operators

Used to determine the logic between variables or values:

```
let x = 5;
let y = 10;

console.log(x > 3 && y < 15);  // Logical AND: true
console.log(x > 7 || y < 15);  // Logical OR: true
console.log(!(x > 7));         // Logical NOT: true
```

Type Operators

Used to determine the type of a variable or expression:

```
console.log(typeof "John");           // "string"
console.log(typeof 3.14);             // "number"
console.log(typeof true);             // "boolean"
console.log(typeof undefined);        // "undefined"
console.log(typeof null);             // "object" (this is a known JavaS
console.log(typeof {name: "John"});   // "object"
console.log(typeof [1, 2, 3]);        // "object"
console.log(typeof function(){});     // "function"

// instanceof operator
let arr = [1, 2, 3];
console.log(arr instanceof Array);    // true
```

Type Conversion and Coercion

JavaScript is a loosely typed language, which means variables can change types.

Explicit Type Conversion

You can manually convert between types using built-in functions:

```
// To string
let num = 123;
let str1 = String(num);       // "123"
let str2 = num.toString();    // "123"

// To number
let str = "456";
let num1 = Number(str);       // 456
let num2 = parseInt(str);     // 456 (integer)
let num3 = parseFloat("3.14"); // 3.14 (float)

// To boolean
let bool1 = Boolean(1);       // true
let bool2 = Boolean(0);       // false
let bool3 = Boolean("");      // false
let bool4 = Boolean("hello"); // true
```

Implicit Type Coercion

JavaScript automatically converts types in certain contexts:

```
// String conversion
let result = "3" + 4;  // "34" (number is converted to string)

// Numeric conversion
let sum = "3" - 2;     // 1 (string is converted to number)
let product = "3" * 2; // 6 (string is converted to number)

// Boolean conversion
if ("hello") {
    // Non-empty strings are truthy
    console.log("This will execute");
}

if (0) {
    // 0 is falsy
    console.log("This won't execute");
}
```

Truthy and Falsy Values

In JavaScript, values are inherently truthy or falsy when evaluated in a boolean context:

Falsy values: - `false` - `0` - `""` (empty string) - `null` - `undefined` - `NaN`

Everything else is truthy.

Control Structures

Control structures direct the flow of a program's execution.

Conditional Statements

if, else if, else

```
let hour = 14;

if (hour < 12) {
    console.log("Good morning");
} else if (hour < 18) {
    console.log("Good afternoon");
} else {
    console.log("Good evening");
}
```

Ternary Operator

A shorthand for simple if-else statements:

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

switch

Used for multiple conditions against a single value:

```
let day = 2;
let dayName;
```

```
switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    // ... other cases
    default:
        dayName = "Unknown";
}

console.log(dayName);  // "Tuesday"
```

Loops

for Loop

Used when you know how many times you want to execute a block of code:

```
for (let i = 0; i < 5; i++) {
    console.log(i);  // 0, 1, 2, 3, 4
}
```

while Loop

Executes a block of code as long as a condition is true:

```
let i = 0;
while (i < 5) {
    console.log(i);  // 0, 1, 2, 3, 4
    i++;
}
```

do-while Loop

Similar to while, but always executes the block at least once:

```
let i = 0;
do {
    console.log(i);  // 0, 1, 2, 3, 4
    i++;
} while (i < 5);
```

for...in Loop

Iterates over the enumerable properties of an object:

```
let person = {
    name: "John",
    age: 30,
    job: "developer"
};

for (let key in person) {
    console.log(key + ": " + person[key]);
}
// name: John
// age: 30
// job: developer
```

for...of Loop

Iterates over iterable objects like arrays, strings, etc.:

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

for (let color of colors) {
    console.log(color);
}
// red
// green
// blue
```

Break and Continue

- `break` exits a loop entirely
- `continue` skips the current iteration and continues with the next

```
// break example
for (let i = 0; i < 10; i++) {
    if (i === 5) {
        break;  // Exit the loop when i is 5
    }
    console.log(i);  // 0, 1, 2, 3, 4
}

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

Comments and Best Practices

Comments are crucial for code documentation and maintainability.

Single-line Comments

```
// This is a single-line comment
let x = 5;  // This is an inline comment
```

Multi-line Comments

```
/* This is a multi-line comment
   that spans multiple lines
   and can be used for longer explanations */
```

JSDoc Comments

Used for generating documentation:

```
/**
 * Calculates the sum of two numbers
 * @param {number} a - The first number
```

```
 * @param {number} b - The second number
 * @returns {number} The sum of a and b
 */
function sum(a, b) {
    return a + b;
}
```

Best Practices for Comments

1. **Be concise**: Write clear, brief comments that explain "why" rather than "what"
2. **Keep comments updated**: Outdated comments are worse than no comments
3. **Comment complex logic**: Focus on explaining difficult or non-obvious code
4. **Use JSDoc for functions**: Document parameters, return values, and purpose
5. **Avoid commenting obvious code**: Don't state the obvious
6. **Use TODO comments**: Mark areas that need future attention with `// TODO: description`

By mastering these JavaScript fundamentals, you'll have a solid foundation for building more complex applications and understanding advanced JavaScript concepts.

3. Functions in JavaScript

Functions are one of the fundamental building blocks in JavaScript. A function is a reusable block of code designed to perform a particular task. Functions allow you to structure your code, make it more readable, reusable, and maintainable.

Function Declaration vs Expression

There are several ways to define functions in JavaScript:

Function Declaration

```
function greet(name) {
    return "Hello, " + name + "!";
}

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

Function declarations are hoisted, meaning they can be called before they are defined in the code.

Function Expression

```
const greet = function(name) {
    return "Hello, " + name + "!";
};

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

Function expressions are not hoisted, so they cannot be called before they are defined.

Key Differences

1. **Hoisting**: Function declarations are hoisted, function expressions are not.
2. **Usage**: Function expressions can be used as arguments to other functions or as immediately invoked function expressions (IIFE).
3. **Naming**: Function expressions can be anonymous (without a name).

Arrow Functions

Arrow functions were introduced in ES6 (ECMAScript 2015) and provide a more concise syntax for writing functions:

```
// Traditional function expression
const add = function(a, b) {
    return a + b;
};

// Arrow function
const add = (a, b) => a + b;
```

Features of Arrow Functions

1. **Shorter syntax**: Especially for simple, single-expression functions.
2. **Implicit return**: If the function body is a single expression, you can omit the `return` keyword and curly braces.

3. **No binding of `this`**: Arrow functions don't have their own `this` context; they inherit `this` from the surrounding code.

```
// Single parameter (parentheses optional)
const square = x => x * x;

// Multiple parameters (parentheses required)
const multiply = (x, y) => x * y;

// Multiple statements (curly braces and return required)
const calculate = (x, y) => {
    const sum = x + y;
    return sum * 2;
};

// No parameters (empty parentheses required)
const getRandomNumber = () => Math.random();

// Returning an object (parentheses around object literal required)
const createPerson = (name, age) => ({ name, age });
```

When to Use Arrow Functions

Arrow functions are ideal for: - Short callback functions - Functions that don't use `this` - Functional programming patterns (map, filter, reduce)

They should be avoided for: - Methods in objects (where `this` refers to the object) - Constructor functions - Event handlers where `this` should refer to the element

Parameters and Arguments

Parameters are the names listed in the function definition, while arguments are the actual values passed to the function.

```
// name and age are parameters
function introduce(name, age) {
    return `My name is ${name} and I am ${age} years old.`;
}

// "John" and 30 are arguments
introduce("John", 30);
```

Parameter Handling

JavaScript is flexible with parameters:

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

sum(2, 3);      // 5
sum(2);         // NaN (because b is undefined)
sum(2, 3, 4);   // 5 (extra arguments are ignored)
```

Checking for Missing Arguments

You can check for missing arguments and provide default behavior:

```
function sum(a, b) {
    // Old way (pre-ES6)
    a = a || 0;
    b = b || 0;
    return a + b;
}

// Or using the typeof operator
function sum(a, b) {
    a = typeof a !== 'undefined' ? a : 0;
    b = typeof b !== 'undefined' ? b : 0;
    return a + b;
}
```

Default Parameters

ES6 introduced default parameters, which simplify handling missing or undefined arguments:

```
function greet(name = "Guest") {
    return `Hello, ${name}!`;
}
```

```
greet("John");  // "Hello, John!"
greet();        // "Hello, Guest!"
```

Default parameters can also use expressions and previous parameters:

```
function calculateTax(amount, taxRate = 0.1, currency = "USD") {
    const tax = amount * taxRate;
    return `${tax} ${currency}`;
}

// Default parameters can reference previous parameters
function createFullName(firstName, lastName = firstName) {
    return `${firstName} ${lastName}`;
}

// Default parameters can use expressions
function getDate(date = new Date()) {
    return date.toLocaleDateString();
}
```

Rest Parameters and Spread Operator

Rest Parameters

The rest parameter syntax (`...`) allows a function to accept an indefinite number of arguments as an array:

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

sum(1, 2);          // 3
sum(1, 2, 3, 4, 5); // 15
```

Rest parameters must be the last parameter in a function definition:

```
function formatName(prefix, ...names) {
    return prefix + ": " + names.join(", ");
}
```

```
formatName("Students", "John", "Jane", "Jim"); // "Students: John, Jane,
```

Spread Operator

The spread operator also uses the `...` syntax but serves the opposite purpose of the rest parameter. It expands an array into individual elements:

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

// Equivalent to console.log(1, 2, 3)
console.log(...numbers);

// Useful for passing array elements as arguments
function sum(a, b, c) {
    return a + b + c;
}

sum(...numbers); // 6
```

The spread operator can also be used to combine arrays and objects:

```
// Combining arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Copying arrays
const original = [1, 2, 3];
const copy = [...original];

// Spreading objects (ES2018)
const person = { name: "John", age: 30 };
const employee = { ...person, jobTitle: "Developer" };
// { name: "John", age: 30, jobTitle: "Developer" }
```

Function Scope and Closures

Scope

Scope determines the accessibility of variables and functions in your code:

1. **Global Scope**: Variables declared outside any function or block are globally accessible.
2. **Function Scope**: Variables declared within a function are only accessible inside that function.
3. **Block Scope**: Variables declared with `let` or `const` within a block (e.g., if statements, loops) are only accessible within that block.

```
// Global scope
const globalVar = "I'm global";

function exampleFunction() {
    // Function scope
    const functionVar = "I'm function-scoped";

    if (true) {
        // Block scope
        const blockVar = "I'm block-scoped";
        console.log(globalVar);      // Accessible
        console.log(functionVar);    // Accessible
        console.log(blockVar);       // Accessible
    }

    console.log(globalVar);          // Accessible
    console.log(functionVar);        // Accessible
    console.log(blockVar);           // Error: blockVar is not defined
}

console.log(globalVar);              // Accessible
console.log(functionVar);            // Error: functionVar is not define
```

Closures

A closure is a function that has access to variables from its outer (enclosing) function's scope, even after the outer function has returned:

```
function createCounter() {
    let count = 0;  // This variable is "closed over"

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

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

In this example, the inner function maintains access to the `count` variable even after `createCounter` has finished executing.

Practical Uses of Closures

1. **Data Privacy**: Creating private variables that can't be accessed directly.

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

    return {
        deposit: function(amount) {
            balance += amount;
            return balance;
        },
        withdraw: function(amount) {
            if (amount > balance) {
                return "Insufficient funds";
            }
            balance -= amount;
            return balance;
        },
        getBalance: function() {
            return balance;
        }
    };
}
```

```
const account = createBankAccount(100);
account.deposit(50);  // 150
account.withdraw(30); // 120
account.getBalance(); // 120
// balance is not directly accessible
```

1. **Function Factories**: Creating functions with preset parameters.

```
function multiplyBy(factor) {
    return function(number) {
        return number * factor;
    };
}

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

double(5); // 10
triple(5); // 15
```

1. **Maintaining State in Event Handlers**:

```
function setupCounter(buttonId) {
    let count = 0;

    document.getElementById(buttonId).addEventListener('click', functior
        count++;
        console.log(`Button clicked ${count} times`);
    });
}
```

Callbacks

A callback is a function passed as an argument to another function, which is then invoked inside the outer function:

```
function greeting(name) {
    console.log(`Hello, ${name}!`);
}

function processUserInput(callback) {
```

```
    const name = prompt("Please enter your name:");
    callback(name);
}

processUserInput(greeting);
```

Asynchronous Callbacks

Callbacks are commonly used for asynchronous operations:

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

fetchData(function(data) {
    console.log(data); // { id: 1, name: "John" }
});
```

Callback Hell

Nested callbacks can lead to "callback hell" or "pyramid of doom":

```
fetchUserData(function(user) {
    fetchUserPosts(user.id, function(posts) {
        fetchPostComments(posts[0].id, function(comments) {
            // Deeply nested and hard to read
            console.log(comments);
        });
    });
});
```

This issue is often addressed using Promises or async/await (covered in the Asynchronous JavaScript section).

Higher-Order Functions

A higher-order function is a function that either: 1. Takes one or more functions as arguments, or 2. Returns a function as its result

Many array methods in JavaScript are higher-order functions:

Array.prototype.map()

Creates a new array by applying a function to each element of the original array:

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8]
```

Array.prototype.filter()

Creates a new array with elements that pass a test:

```
const numbers = [1, 2, 3, 4, 5, 6];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4, 6]
```

Array.prototype.reduce()

Reduces an array to a single value by applying a function to each element:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((total, num) => total + num, 0);
console.log(sum); // 10
```

Creating Higher-Order Functions

You can create your own higher-order functions:

```
// Function that returns a function
function multiplyBy(factor) {
    return function(number) {
        return number * factor;
```

```
    };
}

// Function that takes a function as an argument
function applyOperation(numbers, operation) {
    return numbers.map(operation);
}

const double = multiplyBy(2);
console.log(double(5)); // 10

const numbers = [1, 2, 3];
console.log(applyOperation(numbers, double)); // [2, 4, 6]
```

Immediately Invoked Function Expressions (IIFE)

An IIFE is a function that is executed immediately after it is created:

```
(function() {
    console.log("This function is executed immediately");
})();

// With parameters
(function(name) {
    console.log(`Hello, ${name}!`);
})("John");
```

Uses of IIFE

1. **Creating Private Scope**: Variables declared inside an IIFE are not accessible from outside.

```
(function() {
    const privateVar = "I am private";
    console.log(privateVar); // "I am private"
})();
```

```
console.log(privateVar); // ReferenceError: privateVar is not defined
```

1. **Avoiding Global Namespace Pollution**: Prevents variables from being added to the global scope.

```
// Without IIFE
let counter = 0;
function increment() {
    counter++;
}

// With IIFE
const counter = (function() {
    let count = 0;
    return {
        increment: function() {
            count++;
            return count;
        },
        reset: function() {
            count = 0;
            return count;
        }
    };
})();

counter.increment(); // 1
counter.increment(); // 2
counter.reset();     // 0
```

1. **Module Pattern**: Creating modules with private and public methods.

```
const calculator = (function() {
    // Private
    function square(x) {
        return x * x;
    }

    // Public API
    return {
        add: function(a, b) {
            return a + b;
```

```
        },
        multiply: function(a, b) {
            return a * b;
        },
        calculateArea: function(radius) {
            return Math.PI * square(radius);
        }
    };
})();

calculator.add(2, 3);         // 5
calculator.calculateArea(2);   // 12.566...
calculator.square(2);         // Error: square is not a public method
```

Functions are a cornerstone of JavaScript programming. Understanding these concepts will help you write more efficient, maintainable, and powerful JavaScript code.

4. Objects and Object-Oriented Programming

Objects are one of the most important data types in JavaScript. Unlike primitive data types, objects can store multiple values as properties and methods, making them powerful tools for organizing and structuring code.

Object Literals

The simplest way to create an object is using object literal notation:

```
const person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    isEmployed: true,
    greet: function() {
        return `Hello, my name is ${this.firstName} ${this.lastName}`;
    }
};
```

Accessing Object Properties

There are two ways to access object properties:

```
// Dot notation
console.log(person.firstName); // "John"

// Bracket notation (useful when property name is stored in a variable c
console.log(person["lastName"]); // "Doe"

const propertyName = "age";
console.log(person[propertyName]); // 30
```

Adding, Modifying, and Deleting Properties

Objects are dynamic, allowing you to add, modify, or delete properties at any time:

```
// Adding a new property
person.email = "john.doe@example.com";

// Modifying an existing property
person.age = 31;

// Deleting a property
delete person.isEmployed;
```

Shorthand Property Names (ES6)

When a variable name is the same as the property name, you can use shorthand notation:

```
const name = "John";
const age = 30;

// Instead of {name: name, age: age}
const person = {name, age};
```

Computed Property Names (ES6)

You can use expressions as property names by wrapping them in square brackets:

```
const propertyName = "birthDate";
const person = {
```

```
    name: "John",
    [propertyName]: "1990-01-01"
};
// Equivalent to {name: "John", birthDate: "1990-01-01"}
```

Method Shorthand (ES6)

Instead of defining methods using function expressions, you can use method shorthand:

```
// Old way
const person = {
    name: "John",
    greet: function() {
        return `Hello, my name is ${this.name}`;
    }
};

// Method shorthand
const person = {
    name: "John",
    greet() {
        return `Hello, my name is ${this.name}`;
    }
};
```

Properties and Methods

Property Descriptors

JavaScript objects have property descriptors that define how properties behave:

```
const person = {
    name: "John"
};

// Get property descriptor
const descriptor = Object.getOwnPropertyDescriptor(person, "name");
console.log(descriptor);
// {value: "John", writable: true, enumerable: true, configurable: true}

// Define a property with custom descriptor
```

```
Object.defineProperty(person, "age", {
    value: 30,
    writable: false,     // Cannot be changed
    enumerable: true,    // Shows up in loops
    configurable: false  // Cannot be deleted or reconfigured
});

person.age = 40; // Will not change the value
console.log(person.age); // 30
```

Property Attributes

- **value**: The property's value
- **writable**: Whether the value can be changed
- **enumerable**: Whether the property appears in for...in loops and Object.keys()
- **configurable**: Whether the property can be deleted or its attributes changed

Getters and Setters

Getters and setters allow you to define special methods that are called when a property is accessed or modified:

```
const person = {
    firstName: "John",
    lastName: "Doe",

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

    // Setter
    set fullName(value) {
        const parts = value.split(" ");
        this.firstName = parts[0];
        this.lastName = parts[1];
    }
};

console.log(person.fullName); // "John Doe"
person.fullName = "Jane Smith";
```

```
console.log(person.firstName); // "Jane"
console.log(person.lastName); // "Smith"
```

this Keyword

The `this` keyword refers to the object that is executing the current function. Its value depends on how the function is called:

In a Method

When a function is called as a method of an object, `this` refers to the object:

```
const person = {
    name: "John",
    greet() {
        return `Hello, my name is ${this.name}`;
    }
};

console.log(person.greet()); // "Hello, my name is John"
```

In a Regular Function

In a regular function (not a method), `this` refers to the global object (in browsers, it's `window`):

```
function showThis() {
    console.log(this);
}

showThis(); // Window object (in browsers) or global object (in Node.js)
```

In strict mode, `this` is `undefined` in regular functions:

```
"use strict";
function showThis() {
    console.log(this);
}
```

```
showThis(); // undefined
```

In Event Handlers

In an event handler, `this` refers to the element that received the event:

```
document.getElementById("myButton").addEventListener("click", function()
    console.log(this); // The button element
});
```

In Arrow Functions

Arrow functions don't have their own `this` context. They inherit `this` from the surrounding code:

```
const person = {
    name: "John",
    regularFunction: function() {
        console.log(this.name); // "John"

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

        setTimeout(() => {
            console.log(this.name); // "John" (arrow function inherits t
        }, 100);
    }
};

person.regularFunction();
```

Explicitly Setting `this`

You can explicitly set the value of `this` using `call()`, `apply()`, or `bind()`:

```
function greet() {
    return `Hello, my name is ${this.name}`;
}
```

```
const person = { name: "John" };

// Using call (arguments passed individually)
console.log(greet.call(person)); // "Hello, my name is John"

// Using apply (arguments passed as an array)
console.log(greet.apply(person)); // "Hello, my name is John"

// Using bind (creates a new function with this bound)
const boundGreet = greet.bind(person);
console.log(boundGreet()); // "Hello, my name is John"
```

Constructor Functions

Before ES6 classes, constructor functions were the primary way to create object templates:

```
function Person(firstName, lastName, age) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;

    this.getFullName = function() {
        return `${this.firstName} ${this.lastName}`;
    };
}

const john = new Person("John", "Doe", 30);
console.log(john.getFullName()); // "John Doe"
```

When a function is called with the `new` keyword: 1. A new empty object is created 2. The function's `this` is set to the new object 3. The function code executes 4. The new object is returned (unless the function explicitly returns something else)

The instanceof Operator

You can check if an object was created from a specific constructor using `instanceof`:

```
console.log(john instanceof Person); // true
```

Prototypes and Inheritance

JavaScript uses prototype-based inheritance, where objects inherit properties and methods from a prototype.

The Prototype Chain

Every JavaScript object has a prototype (except for objects created with `Object.create(null)`). When you try to access a property or method, JavaScript first looks for it on the object itself. If it doesn't find it, it looks at the object's prototype, then that object's prototype, and so on up the prototype chain.

Object.prototype

The top of the prototype chain is `Object.prototype`, which provides methods like `toString()`, `hasOwnProperty()`, etc.

Constructor Prototypes

Each constructor function has a `prototype` property that points to an object. Objects created with the constructor inherit properties from this prototype:

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

// Adding a method to the prototype
Person.prototype.greet = function() {
    return `Hello, my name is ${this.name}`;
};

const john = new Person("John");
const jane = new Person("Jane");

console.log(john.greet()); // "Hello, my name is John"
console.log(jane.greet()); // "Hello, my name is Jane"
```

Adding methods to the prototype is more memory-efficient than adding them in the constructor, as all instances share the same method.

Prototype Inheritance

You can create inheritance relationships between constructors:

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

Person.prototype.introduce = function() {
    return `My name is ${this.name} and I am ${this.age} years old`;
};

function Employee(name, age, jobTitle) {
    // Call the parent constructor
    Person.call(this, name, age);
    this.jobTitle = jobTitle;
}

// Inherit from Person's prototype
Employee.prototype = Object.create(Person.prototype);
// Reset the constructor property
Employee.prototype.constructor = Employee;

// Add methods specific to Employee
Employee.prototype.getJobInfo = function() {
    return `I work as a ${this.jobTitle}`;
};

const john = new Employee("John", 30, "Developer");
console.log(john.introduce()); // "My name is John and I am 30 years old
console.log(john.getJobInfo()); // "I work as a Developer"
```

ES6 Classes

ES6 introduced class syntax, which provides a cleaner way to create constructor functions and manage inheritance:

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

```
    }

    introduce() {
        return `My name is ${this.name} and I am ${this.age} years old`;
    }
}

const john = new Person("John", 30);
console.log(john.introduce()); // "My name is John and I am 30 years old
```

Class Inheritance

Inheritance is more straightforward with classes:

```
class Employee extends Person {
    constructor(name, age, jobTitle) {
        super(name, age); // Call parent constructor
        this.jobTitle = jobTitle;
    }

    getJobInfo() {
        return `I work as a ${this.jobTitle}`;
    }

    // Override parent method
    introduce() {
        return `${super.introduce()} and I work as a ${this.jobTitle}`;
    }
}

const jane = new Employee("Jane", 28, "Designer");
console.log(jane.introduce()); // "My name is Jane and I am 28 years old
```

Static Methods

Static methods are called on the class itself, not on instances:

```
class MathUtils {
    static add(a, b) {
        return a + b;
    }
```

```
    static multiply(a, b) {
        return a * b;
    }
}

console.log(MathUtils.add(5, 3)); // 8
```

Getters and Setters in Classes

Classes can also use getters and setters:

```
class Person {
    constructor(firstName, lastName) {
        this._firstName = firstName;
        this._lastName = lastName;
    }

    get firstName() {
        return this._firstName;
    }

    set firstName(value) {
        this._firstName = value;
    }

    get fullName() {
        return `${this._firstName} ${this._lastName}`;
    }

    set fullName(value) {
        const parts = value.split(" ");
        this._firstName = parts[0];
        this._lastName = parts[1];
    }
}

const john = new Person("John", "Doe");
console.log(john.fullName); // "John Doe"
john.fullName = "Jane Smith";
console.log(john.firstName); // "Jane"
```

Private Class Fields (ES2022)

Recent JavaScript versions support private class fields using the `#` prefix:

```
class BankAccount {
    #balance = 0; // Private field

    constructor(owner, initialBalance) {
        this.owner = owner;
        this.#balance = initialBalance;
    }

    deposit(amount) {
        this.#balance += amount;
        return this.#balance;
    }

    withdraw(amount) {
        if (amount > this.#balance) {
            throw new Error("Insufficient funds");
        }
        this.#balance -= amount;
        return this.#balance;
    }

    get balance() {
        return this.#balance;
    }
}

const account = new BankAccount("John", 1000);
console.log(account.balance); // 1000
account.deposit(500);
console.log(account.balance); // 1500
// console.log(account.#balance); // SyntaxError: Private field '#balanc
```

Object Methods

JavaScript provides several built-in methods for working with objects:

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

```
const person = {
    name: "John",
    age: 30,
    job: "Developer"
};

console.log(Object.keys(person)); // ["name", "age", "job"]
console.log(Object.values(person)); // ["John", 30, "Developer"]
console.log(Object.entries(person)); // [["name", "John"], ["age", 30],
```

Object.assign()

Copies properties from one or more source objects to a target object:

```
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };
const result = Object.assign(target, source);

console.log(target); // { a: 1, b: 3, c: 4 }
console.log(result); // { a: 1, b: 3, c: 4 }
```

Object.create()

Creates a new object with the specified prototype:

```
const personProto = {
    greet() {
        return `Hello, my name is ${this.name}`;
    }
};

const john = Object.create(personProto);
john.name = "John";
console.log(john.greet()); // "Hello, my name is John"
```

Object.freeze() and Object.seal()

`Object.freeze()` makes an object immutable (properties can't be added, removed, or changed):

```
const frozen = Object.freeze({ name: "John" });
frozen.name = "Jane"; // Will not change in strict mode
frozen.age = 30; // Will not add in strict mode
console.log(frozen); // { name: "John" }
```

`Object.seal()` prevents adding or removing properties, but allows modifying existing ones:

```
const sealed = Object.seal({ name: "John" });
sealed.name = "Jane"; // Works
sealed.age = 30; // Will not add in strict mode
console.log(sealed); // { name: "Jane" }
```

JSON and JavaScript Objects

JSON (JavaScript Object Notation) is a text-based data format that resembles JavaScript object literals but has some differences:

- JSON keys must be double-quoted strings
- JSON values can only be strings, numbers, objects, arrays, booleans, or null
- JSON doesn't support functions, undefined, or comments

Converting Between Objects and JSON

```
// Object to JSON string
const person = {
    name: "John",
    age: 30,
    isEmployed: true,
    skills: ["JavaScript", "HTML", "CSS"]
};

const jsonString = JSON.stringify(person);
console.log(jsonString);
// {"name":"John","age":30,"isEmployed":true,"skills":["JavaScript","HTM
```

```
// JSON string to object
const jsonData = '{"name":"Jane","age":28,"isEmployed":true}';
const parsedObject = JSON.parse(jsonData);
console.log(parsedObject.name); // "Jane"
```

Customizing JSON Serialization

You can customize how objects are serialized to JSON:

```
const person = {
    name: "John",
    age: 30,
    password: "secret123",
    birthDate: new Date(1990, 0, 1),

    // Custom toJSON method
    toJSON: function() {
        const { password, ...safeData } = this;
        return safeData;
    }
};

console.log(JSON.stringify(person));
// {"name":"John","age":30,"birthDate":"1990-01-01T00:00:00.000Z"}
```

You can also use the replacer parameter:

```
const person = {
    name: "John",
    age: 30,
    password: "secret123"
};

// Exclude password field
const jsonString = JSON.stringify(person, (key, value) => {
    if (key === "password") return undefined;
    return value;
});

console.log(jsonString); // {"name":"John","age":30}
```

Understanding objects and object-oriented programming in JavaScript is crucial for building complex applications. These concepts form the foundation for many JavaScript frameworks and libraries.

5. Arrays and Array Methods

Arrays are ordered collections of values that can store multiple items in a single variable. They are one of the most commonly used data structures in JavaScript and provide powerful methods for data manipulation.

Creating and Manipulating Arrays

Array Creation

There are several ways to create arrays in JavaScript:

```
// Array literal (most common)
const fruits = ["apple", "banana", "orange"];

// Array constructor
const numbers = new Array(1, 2, 3, 4, 5);

// Empty array
const emptyArray = [];

// Array with predefined length but empty slots
const preDefinedLength = new Array(5); // [empty × 5]

// Array.of() method (ES6)
const arrayOf = Array.of(1, 2, 3); // [1, 2, 3]

// Array.from() method (ES6) - creates arrays from array-like or iterabl
const arrayFrom = Array.from("hello"); // ["h", "e", "l", "l", "o"]
```

Accessing Array Elements

Array elements are accessed using zero-based indexing:

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

```
console.log(fruits[0]); // "apple"
console.log(fruits[1]); // "banana"
console.log(fruits[2]); // "orange"
console.log(fruits[3]); // undefined (out of bounds)

// Negative indices are not supported natively
console.log(fruits[-1]); // undefined

// Last element (ES2022 method)
console.log(fruits.at(-1)); // "orange"
console.log(fruits.at(-2)); // "banana"
```

Modifying Arrays

Arrays in JavaScript are mutable, meaning they can be modified after creation:

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

// Changing an element
fruits[1] = "grape";
console.log(fruits); // ["apple", "grape", "orange"]

// Adding elements to the end
fruits.push("mango");
console.log(fruits); // ["apple", "grape", "orange", "mango"]

// Adding elements to the beginning
fruits.unshift("pear");
console.log(fruits); // ["pear", "apple", "grape", "orange", "mango"]

// Removing the last element
const lastFruit = fruits.pop();
console.log(lastFruit); // "mango"
console.log(fruits); // ["pear", "apple", "grape", "orange"]

// Removing the first element
const firstFruit = fruits.shift();
console.log(firstFruit); // "pear"
console.log(fruits); // ["apple", "grape", "orange"]

// Changing multiple elements with splice
// splice(start, deleteCount, item1, item2, ...)
```

```
fruits.splice(1, 1, "kiwi", "lemon");
console.log(fruits); // ["apple", "kiwi", "lemon", "orange"]
```

Array Length

The `length` property returns the number of elements in an array:

```
const fruits = ["apple", "banana", "orange"];
console.log(fruits.length); // 3

// Interesting behavior: setting length property can truncate the array
fruits.length = 2;
console.log(fruits); // ["apple", "banana"]

// Setting length larger than current size creates empty slots
fruits.length = 5;
console.log(fruits); // ["apple", "banana", empty × 3]
```

Array Properties and Methods

JavaScript arrays come with many built-in methods that make data manipulation easier.

Basic Array Methods

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

// concat() - combines arrays
const moreFruits = ["kiwi", "mango"];
const allFruits = fruits.concat(moreFruits);
console.log(allFruits); // ["apple", "banana", "orange", "grape", "kiwi"

// join() - converts array to string with specified separator
const fruitString = fruits.join(", ");
console.log(fruitString); // "apple, banana, orange, grape"

// slice() - returns a portion of the array
const someFruits = fruits.slice(1, 3);
console.log(someFruits); // ["banana", "orange"]

// indexOf() - finds the first occurrence of an element
```

```
const bananaIndex = fruits.indexOf("banana");
console.log(bananaIndex); // 1

// lastIndexOf() - finds the last occurrence of an element
const repeatedFruits = ["apple", "banana", "apple", "orange"];
console.log(repeatedFruits.lastIndexOf("apple")); // 2

// includes() - checks if an array contains an element
console.log(fruits.includes("grape")); // true
console.log(fruits.includes("pear")); // false

// reverse() - reverses the array in place
fruits.reverse();
console.log(fruits); // ["grape", "orange", "banana", "apple"]

// sort() - sorts the array in place
fruits.sort();
console.log(fruits); // ["apple", "banana", "grape", "orange"]
```

Copying Arrays

There are several ways to create a copy of an array:

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

// Using slice()
const copy1 = original.slice();

// Using spread operator (ES6)
const copy2 = [...original];

// Using Array.from()
const copy3 = Array.from(original);

// Using concat()
const copy4 = [].concat(original);

// Using Object.assign()
const copy5 = Object.assign([], original);
```

Note that these methods create shallow copies. For nested arrays or objects, you'll need deep copying techniques.

Iterating Through Arrays

There are multiple ways to iterate through arrays in JavaScript:

for Loop

The traditional way to iterate through arrays:

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

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
// "apple"
// "banana"
// "orange"
```

for...of Loop (ES6)

A cleaner way to iterate through array values:

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

for (const fruit of fruits) {
    console.log(fruit);
}
// "apple"
// "banana"
// "orange"
```

forEach() Method

A functional approach to iteration:

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

fruits.forEach((fruit, index, array) => {
    console.log(`${index}: ${fruit}`);
});
// "0: apple"
```

```
// "1: banana"
// "2: orange"
```

for...in Loop

While this works, it's not recommended for arrays as it iterates over all enumerable properties, not just numeric indices:

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

for (const index in fruits) {
    console.log(`${index}: ${fruits[index]}`);
}
// "0: apple"
// "1: banana"
// "2: orange"
```

Array Transformation Methods

JavaScript provides powerful methods for transforming arrays.

map()

Creates a new array by applying a function to each element:

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

const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6, 8, 10]

const numberStrings = numbers.map(num => `Number ${num}`);
console.log(numberStrings); // ["Number 1", "Number 2", "Number 3", "Num
```

filter()

Creates a new array with elements that pass a test:

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

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

const bigNumbers = numbers.filter(num => num > 5);
console.log(bigNumbers); // [6, 7, 8, 9, 10]
```

reduce()

Reduces an array to a single value by applying a function to each element:

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

// Sum of all numbers
const sum = numbers.reduce((accumulator, currentValue) => accumulator +
console.log(sum); // 15

// Product of all numbers
const product = numbers.reduce((accumulator, currentValue) => accumulato
console.log(product); // 120

// More complex example: grouping objects
const people = [
    { name: "Alice", age: 25 },
    { name: "Bob", age: 30 },
    { name: "Charlie", age: 25 },
    { name: "Dave", age: 30 }
];

const groupedByAge = people.reduce((acc, person) => {
    // If this age doesn't exist in the accumulator yet, create an array
    if (!acc[person.age]) {
        acc[person.age] = [];
    }
    // Add the person to the array for their age
    acc[person.age].push(person.name);
    return acc;
}, {});

console.log(groupedByAge);
// {
//   "25": ["Alice", "Charlie"],
```

```
//   "30": ["Bob", "Dave"]
// }
```

reduceRight()

Like `reduce()`, but works from right to left:

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

const reversedString = letters.reduceRight((acc, letter) => acc + letter
console.log(reversedString); // "dcba"
```

flatMap() (ES2019)

Maps each element using a mapping function, then flattens the result into a new array:

```
const sentences = ["Hello world", "How are you"];

const words = sentences.flatMap(sentence => sentence.split(" "));
console.log(words); // ["Hello", "world", "How", "are", "you"]
```

flat() (ES2019)

Creates a new array with all sub-array elements concatenated recursively up to the specified depth:

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

console.log(nestedArray.flat()); // [1, 2, 3, 4, [5, 6]]
console.log(nestedArray.flat(2)); // [1, 2, 3, 4, 5, 6]

// Use Infinity to flatten all nested arrays regardless of depth
const deeplyNested = [1, [2, [3, [4, [5]]]]];
console.log(deeplyNested.flat(Infinity)); // [1, 2, 3, 4, 5]
```

Sorting and Searching Arrays

sort()

The `sort()` method sorts the elements of an array in place:

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

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

// Numbers need a compare function, otherwise they're sorted as strings
const numbers = [10, 5, 40, 25, 100];

// Wrong way (sorts as strings)
numbers.sort();
console.log(numbers); // [10, 100, 25, 40, 5]

// Correct way (ascending)
numbers.sort((a, b) => a - b);
console.log(numbers); // [5, 10, 25, 40, 100]

// Descending order
numbers.sort((a, b) => b - a);
console.log(numbers); // [100, 40, 25, 10, 5]

// Sorting objects
const people = [
    { name: "Alice", age: 25 },
    { name: "Bob", age: 30 },
    { name: "Charlie", age: 20 }
];

// Sort by age
people.sort((a, b) => a.age - b.age);
console.log(people);
// [
//   { name: "Charlie", age: 20 },
//   { name: "Alice", age: 25 },
//   { name: "Bob", age: 30 }
// ]
```

```
// Sort by name
people.sort((a, b) => a.name.localeCompare(b.name));
console.log(people);
// [
//   { name: "Alice", age: 25 },
//   { name: "Bob", age: 30 },
//   { name: "Charlie", age: 20 }
// ]
```

find() and findIndex()

`find()` returns the first element that satisfies a condition, while `findIndex()` returns its index:

```
const numbers = [5, 12, 8, 130, 44];

const found = numbers.find(num => num > 10);
console.log(found); // 12

const foundIndex = numbers.findIndex(num => num > 10);
console.log(foundIndex); // 1

// With objects
const people = [
    { name: "Alice", age: 25 },
    { name: "Bob", age: 30 },
    { name: "Charlie", age: 20 }
];

const youngPerson = people.find(person => person.age < 25);
console.log(youngPerson); // { name: "Charlie", age: 20 }
```

some() and every()

`some()` checks if at least one element passes a test, while `every()` checks if all elements pass:

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

// Are there any even numbers?
const hasEven = numbers.some(num => num % 2 === 0);
```

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

// Are all numbers even?
const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // false

// Are all numbers less than 10?
const allLessThan10 = numbers.every(num => num < 10);
console.log(allLessThan10); // true
```

Multidimensional Arrays

JavaScript doesn't have true multidimensional arrays, but you can create arrays of arrays:

```
// 2D array (matrix)
const matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

// Accessing elements
console.log(matrix[1][2]); // 6 (row 1, column 2)

// Iterating through a 2D array
for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
        console.log(`matrix[${i}][${j}] = ${matrix[i][j]}`);
    }
}

// Using forEach
matrix.forEach((row, i) => {
    row.forEach((value, j) => {
        console.log(`matrix[${i}][${j}] = ${value}`);
    });
});

// Using for...of
for (const row of matrix) {
    for (const value of row) {
        console.log(value);
```

```
    }
}
```

Common Operations on 2D Arrays

```
// Sum all elements in a matrix
const sum = matrix.reduce((acc, row) =>
    acc + row.reduce((rowAcc, value) => rowAcc + value, 0), 0);
console.log(sum); // 45

// Transpose a matrix (convert rows to columns)
const transpose = matrix => matrix[0].map((_, colIndex) =>
    matrix.map(row => row[colIndex]));

const transposed = transpose(matrix);
console.log(transposed);
// [
//   [1, 4, 7],
//   [2, 5, 8],
//   [3, 6, 9]
// ]
```

Array-Like Objects

Some JavaScript objects look like arrays but aren't true arrays:

```
// DOM NodeList
const divs = document.querySelectorAll('div');

// Arguments object in functions
function example() {
    console.log(arguments);
    // arguments has length and indexed elements, but isn't a true array
}

// String (iterable and has indices and length)
const str = "hello";
```

Converting array-like objects to true arrays:

```
// Using Array.from()
const divArray = Array.from(document.querySelectorAll('div'));

// Using spread operator
function example() {
    const args = [...arguments];
    // Now args is a true array
}

// Converting string to array of characters
const chars = Array.from("hello");
// or
const chars = [..."hello"];
```

ES6+ Array Features

Array Destructuring

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

// Basic destructuring
const [first, second, third] = numbers;
console.log(first, second, third); // 1 2 3

// Skipping elements
const [a, , c] = numbers;
console.log(a, c); // 1 3

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

// Default values
const [x = 0, y = 0, z = 0] = [1, 2];
console.log(x, y, z); // 1 2 0

// Swapping variables
let m = 1;
let n = 2;
```

```
[m, n] = [n, m];
console.log(m, n); // 2 1
```

Spread Operator with Arrays

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

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

// Copying arrays
const copy = [...arr1];

// Using spread in function calls
const numbers = [1, 2, 3, 4, 5];
console.log(Math.max(...numbers)); // 5

// Creating arrays with additional elements
const newArray = [0, ...arr1, 4];
console.log(newArray); // [0, 1, 2, 3, 4]
```

Array Methods with Arrow Functions

Arrow functions make array methods more concise:

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

// map
const doubled = numbers.map(n => n * 2);

// filter
const even = numbers.filter(n => n % 2 === 0);

// reduce
const sum = numbers.reduce((acc, n) => acc + n, 0);

// Chaining methods
const result = numbers
    .filter(n => n % 2 === 0)
```

```
    .map(n => n * 2)
    .reduce((acc, n) => acc + n, 0);

console.log(result); // 12 (2*2 + 4*2)
```

Arrays are a fundamental data structure in JavaScript, and mastering array methods is essential for effective JavaScript programming. The array methods covered in this section provide powerful tools for data manipulation, transformation, and analysis.

6. DOM Manipulation

What is the DOM?

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages like JavaScript can interact with the page.

The DOM is not part of JavaScript; it's a Web API used by JavaScript to interact with HTML and XML documents. It provides a structured representation of the document as a tree of objects, where each object corresponds to a part of the document.

DOM Tree Structure

When a browser loads a web page, it creates a DOM tree representing that page. The DOM tree consists of several types of nodes:

1. **Document Node**: The root node, representing the entire document
2. **Element Nodes**: Represent HTML elements (e.g., `<div>`, `<p>`, `<a>`)
3. **Attribute Nodes**: Represent attributes of elements
4. **Text Nodes**: Represent text content within elements
5. **Comment Nodes**: Represent HTML comments

Here's a simple HTML document and its corresponding DOM tree:

```
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
</head>
```

```
<body>
    <h1>Welcome</h1>
    <p>This is a <a href="https://example.com">link</a>.</p>
</body>
</html>
```

DOM Tree: - Document - html - head - title - "My Page" (text node) - body - h1 - "Welcome" (text node) - p - "This is a " (text node) - a - "link" (text node) - href="https://example.com" (attribute node) - "." (text node)

Selecting DOM Elements

Before you can manipulate elements, you need to select them. JavaScript provides several methods to select DOM elements:

getElementById

Selects an element by its ID attribute:

```
const heading = document.getElementById("main-heading");
```

getElementsByClassName

Returns a live HTMLCollection of elements with the specified class name:

```
const items = document.getElementsByClassName("item");
// Access individual elements with index
const firstItem = items[0];
```

getElementsByTagName

Returns a live HTMLCollection of elements with the specified tag name:

```
const paragraphs = document.getElementsByTagName("p");
```

querySelector

Returns the first element that matches a specified CSS selector:

```
const container = document.querySelector(".container");
const heading = document.querySelector("#main-heading");
const firstParagraph = document.querySelector("p");
```

querySelectorAll

Returns a static NodeList of all elements that match a specified CSS selector:

```
const allParagraphs = document.querySelectorAll("p");
const itemsWithClass = document.querySelectorAll(".item");
```

Differences Between Selection Methods

- `getElementById`, `getElementsByClassName`, and `getElementsByTagName` return live collections that update automatically when the DOM changes.
- `querySelector` and `querySelectorAll` return static collections that don't update when the DOM changes.
- `querySelector` is more versatile as it accepts any CSS selector, but it might be slightly slower than direct methods like `getElementById`.

Traversing the DOM

Once you have an element, you can navigate to related elements:

```
const parent = element.parentNode; // or parentElement
const children = element.children;
const firstChild = element.firstChild; // Might be a text node
const firstElementChild = element.firstElementChild; // First child that
const lastChild = element.lastChild;
const nextSibling = element.nextSibling; // Might be a text node
const nextElementSibling = element.nextElementSibling;
const previousSibling = element.previousSibling;
```

Modifying DOM Elements

Changing Text Content

There are several ways to change the text content of an element:

```
// textContent gets/sets the text content of all elements, including <sc
element.textContent = "New text content";

// innerText is aware of styling and won't return the text of hidden ele
element.innerText = "New inner text";

// innerHTML gets/sets the HTML content, which can include tags
element.innerHTML = "Text with <strong>bold</strong> content";
```

Changing Attributes

You can get, set, and remove attributes:

```
// Get attribute
const href = element.getAttribute("href");

// Set attribute
element.setAttribute("href", "https://example.com");

// Check if attribute exists
const hasAttribute = element.hasAttribute("target");

// Remove attribute
element.removeAttribute("target");

// Direct property access for common attributes
element.id = "new-id";
element.className = "new-class";
element.href = "https://example.com";
```

Changing Styles

You can modify an element's style directly or by manipulating its classes:

```
// Direct style manipulation
element.style.color = "red";
element.style.backgroundColor = "black"; // Note the camelCase
element.style.fontSize = "16px";

// Getting computed styles (actual applied styles)
const computedStyle = window.getComputedStyle(element);
```

```
const color = computedStyle.color;

// Class manipulation
element.className = "new-class"; // Replaces all classes
element.className += " another-class"; // Appends a class (note the spac

// Modern class manipulation with classList
element.classList.add("new-class");
element.classList.remove("old-class");
element.classList.toggle("active"); // Adds if absent, removes if preser
element.classList.replace("old-class", "new-class");
const hasClass = element.classList.contains("active");
```

Creating and Removing Elements

Creating Elements

```
// Create a new element
const newParagraph = document.createElement("p");

// Add text content
newParagraph.textContent = "This is a new paragraph.";

// Add attributes
newParagraph.id = "new-paragraph";
newParagraph.className = "content";

// Create a text node
const textNode = document.createTextNode("This is a text node.");

// Create a comment
const comment = document.createComment("This is a comment");
```

Adding Elements to the DOM

```
// Append at the end of parent's children
parent.appendChild(newElement);

// Insert before a specific child
parent.insertBefore(newElement, referenceElement);
```

```
// Modern insertion methods
parent.append(newElement); // Can append multiple nodes and text
parent.prepend(newElement); // Insert at the beginning
referenceElement.before(newElement); // Insert before reference
referenceElement.after(newElement); // Insert after reference
```

Removing Elements

```
// Remove a child element
parent.removeChild(childElement);

// Self-removal (modern)
element.remove();
```

Cloning Elements

```
// Clone without children
const shallowClone = element.cloneNode(false);

// Clone with all descendants
const deepClone = element.cloneNode(true);
```

Document Fragments

Document fragments are lightweight containers that hold multiple nodes to add to the DOM all at once, minimizing reflows and repaints:

```
const fragment = document.createDocumentFragment();

for (let i = 0; i < 100; i++) {
    const listItem = document.createElement("li");
    listItem.textContent = `Item ${i}`;
    fragment.appendChild(listItem);
}

// Only one DOM update
document.getElementById("myList").appendChild(fragment);
```

Event Handling

Events are actions or occurrences that happen in the browser, which can be detected and responded to with JavaScript.

Adding Event Listeners

```
// Basic syntax
element.addEventListener(eventType, handlerFunction, options);

// Example
const button = document.getElementById("myButton");
button.addEventListener("click", function(event) {
    console.log("Button clicked!");
    console.log(event); // The event object contains information about t
});

// Using arrow functions
button.addEventListener("click", (event) => {
    console.log("Button clicked with arrow function!");
});

// Named function reference
function handleClick(event) {
    console.log("Button clicked with named function!");
}
button.addEventListener("click", handleClick);

// With options (third parameter)
button.addEventListener("click", handleClick, {
    once: true, // Only trigger once
    capture: false, // Use bubbling phase (default)
    passive: true // Indicates the handler won't call preventDefault()
});
```

Removing Event Listeners

```
// You must use the same function reference
button.removeEventListener("click", handleClick);

// Anonymous functions can't be removed this way
```

```
button.addEventListener("click", function() { console.log("Can't remove
// This won't work:
button.removeEventListener("click", function() { console.log("Can't remo
```

Common Events

```
// Mouse events
element.addEventListener("click", handler);
element.addEventListener("dblclick", handler);
element.addEventListener("mousedown", handler);
element.addEventListener("mouseup", handler);
element.addEventListener("mousemove", handler);
element.addEventListener("mouseover", handler);
element.addEventListener("mouseout", handler);
element.addEventListener("mouseenter", handler);
element.addEventListener("mouseleave", handler);

// Keyboard events
element.addEventListener("keydown", handler);
element.addEventListener("keyup", handler);
element.addEventListener("keypress", handler);

// Form events
element.addEventListener("submit", handler);
element.addEventListener("change", handler);
element.addEventListener("input", handler);
element.addEventListener("focus", handler);
element.addEventListener("blur", handler);

// Document/Window events
window.addEventListener("load", handler);
document.addEventListener("DOMContentLoaded", handler);
window.addEventListener("resize", handler);
window.addEventListener("scroll", handler);
```

The Event Object

When an event occurs, the browser creates an event object with details about the event:

```
element.addEventListener("click", function(event) {
    // General properties
```

```
    console.log(event.type); // "click"
    console.log(event.target); // The element that triggered the event
    console.log(event.currentTarget); // The element the listener is att
    console.log(event.timeStamp); // When the event occurred

    // Mouse event properties
    console.log(event.clientX, event.clientY); // Coordinates relative t
    console.log(event.pageX, event.pageY); // Coordinates relative to do
    console.log(event.button); // Which mouse button was pressed

    // Keyboard event properties
    console.log(event.key); // The key value
    console.log(event.code); // The physical key code
    console.log(event.altKey, event.ctrlKey, event.shiftKey); // Modifie

    // Prevent default behavior
    event.preventDefault();

    // Stop propagation
    event.stopPropagation();
});
```

Event Propagation

When an event occurs on an element, it first runs the handlers on it, then on its parent, and so on up the tree. This is called "bubbling."

Bubbling

Events bubble up from the target element to the root:

```
// HTML structure:
// <div id="outer">
//   <div id="inner">
//     <button id="button">Click me</button>
//   </div>
// </div>

document.getElementById("button").addEventListener("click", function(eve
    console.log("Button clicked");
});
```

```
document.getElementById("inner").addEventListener("click", function(ever
    console.log("Inner div clicked");
});

document.getElementById("outer").addEventListener("click", function(ever
    console.log("Outer div clicked");
});

// When button is clicked, the output will be:
// "Button clicked"
// "Inner div clicked"
// "Outer div clicked"
```

Capturing

Events can also be captured on their way down from the root to the target:

```
document.getElementById("outer").addEventListener("click", function(ever
    console.log("Outer div captured");
}, true); // true enables capturing phase

document.getElementById("inner").addEventListener("click", function(ever
    console.log("Inner div captured");
}, true);

document.getElementById("button").addEventListener("click", function(eve
    console.log("Button captured");
}, true);

// When button is clicked, the output will be:
// "Outer div captured"
// "Inner div captured"
// "Button captured"
// "Button clicked"
// "Inner div clicked"
// "Outer div clicked"
```

Stopping Propagation

You can stop event propagation to prevent it from bubbling up:

```
document.getElementById("inner").addEventListener("click", function(ever
    console.log("Inner div clicked");
    event.stopPropagation(); // Prevents bubbling to outer div
});
```

Event Delegation

Event delegation is a technique where you attach an event listener to a parent element instead of multiple child elements. It leverages event bubbling to handle events for multiple elements with a single listener:

```
// Instead of:
document.querySelectorAll("li").forEach(item => {
    item.addEventListener("click", handleClick);
});

// Use event delegation:
document.getElementById("list").addEventListener("click", function(event
    if (event.target.tagName === "LI") {
        handleClick(event);
    }
});
```

Benefits of event delegation: 1. Fewer event listeners, better performance 2. Automatically handles dynamically added elements 3. Less memory usage 4. Cleaner code

Practical DOM Manipulation Examples

Toggle Element Visibility

```
const toggleButton = document.getElementById("toggleButton");
const element = document.getElementById("toggleElement");

toggleButton.addEventListener("click", function() {
    if (element.style.display === "none" || element.style.display === "'
        element.style.display = "block";
    } else {
        element.style.display = "none";
    }
```

```
    // Alternative using classList
    // element.classList.toggle("hidden");
});
```

Form Validation

```
const form = document.getElementById("myForm");
const emailInput = document.getElementById("email");
const errorMessage = document.getElementById("errorMessage");

form.addEventListener("submit", function(event) {
    const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    if (!emailPattern.test(emailInput.value)) {
        event.preventDefault(); // Prevent form submission
        errorMessage.textContent = "Please enter a valid email address";
        emailInput.classList.add("error");
    } else {
        errorMessage.textContent = "";
        emailInput.classList.remove("error");
    }
});
```

Creating a Tabbed Interface

```
const tabs = document.querySelectorAll(".tab");
const tabContents = document.querySelectorAll(".tab-content");

tabs.forEach(tab => {
    tab.addEventListener("click", function() {
        // Remove active class from all tabs and contents
        tabs.forEach(t => t.classList.remove("active"));
        tabContents.forEach(content => content.classList.remove("active"

        // Add active class to clicked tab and corresponding content
        this.classList.add("active");
        const contentId = this.getAttribute("data-tab");
        document.getElementById(contentId).classList.add("active");
```

```
    });
});
```

Dynamic List with Add/Remove Functionality

```
const itemInput = document.getElementById("itemInput");
const addButton = document.getElementById("addButton");
const itemList = document.getElementById("itemList");

// Add item
addButton.addEventListener("click", function() {
    if (itemInput.value.trim() !== "") {
        const li = document.createElement("li");
        li.textContent = itemInput.value;

        const deleteButton = document.createElement("button");
        deleteButton.textContent = "Delete";
        deleteButton.className = "delete-button";

        li.appendChild(deleteButton);
        itemList.appendChild(li);
        itemInput.value = "";
    }
});

// Delete item (using event delegation)
itemList.addEventListener("click", function(event) {
    if (event.target.classList.contains("delete-button")) {
        event.target.parentElement.remove();
    }
});
```

DOM manipulation is a fundamental skill for web development, allowing you to create dynamic, interactive web pages. Understanding how to select, modify, create, and remove elements, as well as how to handle events, gives you the power to build rich user interfaces and enhance the user experience.

7. Asynchronous JavaScript

Asynchronous programming is a crucial aspect of JavaScript, especially for web development. It allows code to run in the background without blocking the main thread,

enabling responsive user interfaces and efficient handling of operations like API calls, file operations, and timers.

Understanding Synchronous vs Asynchronous Code

Synchronous Code

In synchronous programming, operations are executed one after another, in sequence. Each operation must complete before the next one begins:

```
console.log("First");
console.log("Second");
console.log("Third");

// Output:
// First
// Second
// Third
```

Synchronous code is straightforward but can lead to problems when operations take a long time to complete:

```
console.log("Start");
// This would freeze the browser if it were synchronous
const data = fetchDataFromServer(); // Imagine this takes 5 seconds
console.log("Data:", data);
console.log("End");
```

Asynchronous Code

Asynchronous programming allows operations to execute independently from the main program flow:

```
console.log("Start");

setTimeout(() => {
    console.log("This runs after 2 seconds");
}, 2000);

console.log("End");
```

```
// Output:
// Start
// End
// This runs after 2 seconds
```

Asynchronous operations in JavaScript include: - Timers (`setTimeout`, `setInterval`) - AJAX/Fetch requests - File operations in Node.js - Event listeners - Promises and async/await

Callbacks

Callbacks are functions passed as arguments to other functions, to be executed after an operation completes:

```
function fetchData(callback) {
    // Simulate API call with setTimeout
    setTimeout(() => {
        const data = { id: 1, name: "John" };
        callback(data);
    }, 2000);
}

console.log("Start");

fetchData(function(data) {
    console.log("Data received:", data);
});

console.log("End");

// Output:
// Start
// End
// Data received: {id: 1, name: "John"}
```

Callback Hell

When multiple asynchronous operations depend on each other, callbacks can lead to deeply nested code known as "callback hell" or the "pyramid of doom":

```
fetchUserData(function(user) {
    fetchUserPosts(user.id, function(posts) {
        fetchPostComments(posts[0].id, function(comments) {
            fetchCommentAuthor(comments[0].authorId, function(author) {
                console.log("Author:", author);
                // More nested callbacks...
            });
        });
    });
});
```

This code is hard to read, debug, and maintain. Promises and async/await were introduced to solve this problem.

Promises

Promises are objects representing the eventual completion or failure of an asynchronous operation. They allow you to attach callbacks to handle the success or failure of the operation.

Creating Promises

```
const myPromise = new Promise((resolve, reject) => {
    // Asynchronous operation
    setTimeout(() => {
        const success = true;

        if (success) {
            resolve("Operation completed successfully!");
        } else {
            reject("Operation failed!");
        }
    }, 2000);
});
```

Using Promises

```
myPromise
    .then(result => {
```

```
        console.log("Success:", result);
    })
    .catch(error => {
        console.log("Error:", error);
    })
    .finally(() => {
        console.log("Promise settled (fulfilled or rejected)");
    });
```

Promise States

A Promise can be in one of three states: 1. **Pending**: Initial state, neither fulfilled nor rejected 2. **Fulfilled**: The operation completed successfully 3. **Rejected**: The operation failed

Once a promise is fulfilled or rejected, it is settled and cannot change state.

Converting Callback-Based Functions to Promises

```
// Callback-based function
function fetchDataWithCallback(callback) {
    setTimeout(() => {
        callback(null, { id: 1, name: "John" });
    }, 2000);
}

// Promise-based version
function fetchDataWithPromise() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({ id: 1, name: "John" });
            // In case of error: reject(new Error("Failed to fetch data'
        }, 2000);
    });
}

// Usage
fetchDataWithPromise()
    .then(data => console.log("Data:", data))
    .catch(error => console.error("Error:", error));
```

Promise Chaining

Promises can be chained to handle sequential asynchronous operations:

```
fetchUser(userId)
    .then(user => {
        console.log("User:", user);
        return fetchPosts(user.id);
    })
    .then(posts => {
        console.log("Posts:", posts);
        return fetchComments(posts[0].id);
    })
    .then(comments => {
        console.log("Comments:", comments);
    })
    .catch(error => {
        console.error("Error:", error);
    });
```

This is much cleaner than the callback hell example.

Promise.all, Promise.race, Promise.allSettled

Promise.all

Waits for all promises to resolve, or rejects if any promise rejects:

```
const promise1 = fetchUser(1);
const promise2 = fetchUser(2);
const promise3 = fetchUser(3);

Promise.all([promise1, promise2, promise3])
    .then(results => {
        console.log("All users:", results);
        // results is an array of the resolved values
    })
    .catch(error => {
        console.error("At least one promise rejected:", error);
    });
```

Promise.race

Returns the first promise to settle (either fulfill or reject):

```
const promise1 = new Promise(resolve => setTimeout(() => resolve("First'
const promise2 = new Promise(resolve => setTimeout(() => resolve("Second

Promise.race([promise1, promise2])
    .then(result => {
        console.log("Fastest promise:", result); // "Second"
    })
    .catch(error => {
        console.error("Error:", error);
    });
```

Promise.allSettled

Waits for all promises to settle (either fulfill or reject):

```
const promise1 = Promise.resolve("Success");
const promise2 = Promise.reject("Failure");

Promise.allSettled([promise1, promise2])
    .then(results => {
        console.log("All settled:", results);
        // [
        //   { status: "fulfilled", value: "Success" },
        //   { status: "rejected", reason: "Failure" }
        // ]
    });
```

Promise.any (ES2021)

Returns the first promise to fulfill, or rejects if all promises reject:

```
const promise1 = Promise.reject("Error 1");
const promise2 = new Promise(resolve => setTimeout(() => resolve("Succes
const promise3 = Promise.reject("Error 3");

Promise.any([promise1, promise2, promise3])
    .then(result => {
        console.log("First fulfilled promise:", result); // "Success"
```

```
    })
    .catch(error => {
        console.error("All promises rejected:", error);
    });
```

Async/Await

Async/await is syntactic sugar built on top of promises, making asynchronous code look and behave more like synchronous code:

Basic Syntax

```
async function fetchData() {
    try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        console.log("Data:", data);
        return data;
    } catch (error) {
        console.error("Error:", error);
        throw error;
    }
}

// Async functions always return a promise
fetchData()
    .then(data => console.log("Processed data:", data))
    .catch(error => console.error("Caught error:", error));
```

Converting Promise Chains to Async/Await

Before (promise chain):

```
function fetchUserData() {
    return fetchUser(userId)
        .then(user => {
            console.log("User:", user);
            return fetchPosts(user.id);
        })
        .then(posts => {
```

```
            console.log("Posts:", posts);
            return fetchComments(posts[0].id);
        })
        .then(comments => {
            console.log("Comments:", comments);
            return comments;
        })
        .catch(error => {
            console.error("Error:", error);
            throw error;
        });
}
```

After (async/await):

```
async function fetchUserData() {
    try {
        const user = await fetchUser(userId);
        console.log("User:", user);

        const posts = await fetchPosts(user.id);
        console.log("Posts:", posts);

        const comments = await fetchComments(posts[0].id);
        console.log("Comments:", comments);

        return comments;
    } catch (error) {
        console.error("Error:", error);
        throw error;
    }
}
```

Parallel Execution with Async/Await

While async/await makes it easy to write sequential code, sometimes you want to run operations in parallel:

```
async function fetchAllData() {
    try {
        // These will run in parallel
        const [users, posts, comments] = await Promise.all([
```

```
            fetchUsers(),
            fetchPosts(),
            fetchComments()
        ]);

        console.log("Users:", users);
        console.log("Posts:", posts);
        console.log("Comments:", comments);

        return { users, posts, comments };
    } catch (error) {
        console.error("Error:", error);
        throw error;
    }
}
```

Error Handling with Async/Await

Async/await makes error handling more intuitive with try/catch blocks:

```
async function fetchData() {
    try {
        const user = await fetchUser(userId);
        const posts = await fetchPosts(user.id);
        return posts;
    } catch (error) {
        console.error("Error fetching data:", error);
        // Handle error or rethrow
        throw new Error(`Data fetch failed: ${error.message}`);
    } finally {
        console.log("Fetch operation completed");
    }
}
```

Fetch API

The Fetch API provides a modern interface for making HTTP requests, returning promises:

Basic Usage

```
fetch('https://api.example.com/data')
    .then(response => {
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        return response.json(); // Parse JSON response
    })
    .then(data => {
        console.log("Data:", data);
    })
    .catch(error => {
        console.error("Fetch error:", error);
    });
```

With Async/Await

```
async function fetchData() {
    try {
        const response = await fetch('https://api.example.com/data');

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        console.log("Data:", data);
        return data;
    } catch (error) {
        console.error("Fetch error:", error);
        throw error;
    }
}
```

POST Request

```
async function postData(url, data) {
    try {
        const response = await fetch(url, {
```

```
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        return await response.json();
    } catch (error) {
        console.error("Post error:", error);
        throw error;
    }
}

// Usage
postData('https://api.example.com/users', { name: 'John', age: 30 })
    .then(data => console.log("Response:", data))
    .catch(error => console.error("Error:", error));
```

Fetch Options

```
fetch('https://api.example.com/data', {
    method: 'GET', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer your-token'
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-dov
    body: JSON.stringify(data) // body data type must match "Content-Typ
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

Working with APIs

Modern web applications often interact with external APIs to fetch and send data:

RESTful API Example

```
// API service class
class ApiService {
    constructor(baseUrl) {
        this.baseUrl = baseUrl;
    }

    async get(endpoint) {
        const response = await fetch(`${this.baseUrl}${endpoint}`);
        if (!response.ok) {
            throw new Error(`API error! Status: ${response.status}`);
        }
        return response.json();
    }

    async post(endpoint, data) {
        const response = await fetch(`${this.baseUrl}${endpoint}`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });
        if (!response.ok) {
            throw new Error(`API error! Status: ${response.status}`);
        }
        return response.json();
    }

    async put(endpoint, data) {
        const response = await fetch(`${this.baseUrl}${endpoint}`, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });
        if (!response.ok) {
```

```
            throw new Error(`API error! Status: ${response.status}`);
        }
        return response.json();
    }

    async delete(endpoint) {
        const response = await fetch(`${this.baseUrl}${endpoint}`, {
            method: 'DELETE'
        });
        if (!response.ok) {
            throw new Error(`API error! Status: ${response.status}`);
        }
        return response.json();
    }
}

// Usage
const api = new ApiService('https://api.example.com');

async function getUserData() {
    try {
        const users = await api.get('/users');
        console.log("Users:", users);

        const newUser = await api.post('/users', { name: 'John', email:
        console.log("New user:", newUser);

        const updatedUser = await api.put(`/users/${newUser.id}`, { name
        console.log("Updated user:", updatedUser);

        const deleteResult = await api.delete(`/users/${newUser.id}`);
        console.log("Delete result:", deleteResult);
    } catch (error) {
        console.error("API error:", error);
    }
}
```

Handling Authentication

```
class AuthApiService {
    constructor(baseUrl) {
        this.baseUrl = baseUrl;
        this.token = localStorage.getItem('authToken');
```

```
    }

    setToken(token) {
        this.token = token;
        localStorage.setItem('authToken', token);
    }

    clearToken() {
        this.token = null;
        localStorage.removeItem('authToken');
    }

    getHeaders() {
        const headers = {
            'Content-Type': 'application/json'
        };

        if (this.token) {
            headers['Authorization'] = `Bearer ${this.token}`;
        }

        return headers;
    }

    async login(credentials) {
        const response = await fetch(`${this.baseUrl}/login`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(credentials)
        });

        if (!response.ok) {
            throw new Error(`Login failed! Status: ${response.status}`);
        }

        const data = await response.json();
        this.setToken(data.token);
        return data;
    }

    async get(endpoint) {
        const response = await fetch(`${this.baseUrl}${endpoint}`, {
            headers: this.getHeaders()
```

```
        });

        if (response.status === 401) {
            this.clearToken();
            throw new Error('Unauthorized! Please login again.');
        }

        if (!response.ok) {
            throw new Error(`API error! Status: ${response.status}`);
        }

        return response.json();
    }

    // Similar implementations for post, put, delete...
}

// Usage
const authApi = new AuthApiService('https://api.example.com');

async function loginAndFetchData() {
    try {
        await authApi.login({ email: 'user@example.com', password: 'pass
        const userData = await authApi.get('/user/profile');
        console.log("User profile:", userData);
    } catch (error) {
        console.error("Error:", error);
    }
}
```

Practical Asynchronous Patterns

Debouncing

Debouncing limits how often a function can fire by delaying its execution until after a certain amount of time has passed since the last invocation:

```
function debounce(func, delay) {
    let timeoutId;

    return function(...args) {
        const context = this;
```

```
        clearTimeout(timeoutId);

        timeoutId = setTimeout(() => {
            func.apply(context, args);
        }, delay);
    };
}

// Usage
const searchInput = document.getElementById('search');
const debouncedSearch = debounce(function(event) {
    console.log("Searching for:", event.target.value);
    // API call or other expensive operation
}, 500);

searchInput.addEventListener('input', debouncedSearch);
```

Throttling

Throttling ensures a function is called at most once in a specified time period:

```
function throttle(func, limit) {
    let inThrottle;

    return function(...args) {
        const context = this;

        if (!inThrottle) {
            func.apply(context, args);
            inThrottle = true;
            setTimeout(() => {
                inThrottle = false;
            }, limit);
        }
    };
}

// Usage
const scrollHandler = throttle(function() {
    console.log("Scroll event throttled");
    // Expensive calculation or API call
}, 300);
```

```
window.addEventListener('scroll', scrollHandler);
```

Retry Pattern

Automatically retry failed asynchronous operations:

```
async function fetchWithRetry(url, options = {}, retries = 3, backoff =
    try {
        return await fetch(url, options);
    } catch (error) {
        if (retries <= 0) {
            throw error;
        }

        console.log(`Retrying in ${backoff}ms... (${retries} retries le

        // Wait for the backoff period
        await new Promise(resolve => setTimeout(resolve, backoff));

        // Exponential backoff
        return fetchWithRetry(url, options, retries - 1, backoff * 2);
    }
}

// Usage
fetchWithRetry('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log("Data:", data))
    .catch(error => console.error("Failed after retries:", error));
```

Cancellable Fetch with AbortController

Cancel fetch requests that are no longer needed:

```
function fetchWithTimeout(url, options = {}, timeout = 5000) {
    const controller = new AbortController();
    const { signal } = controller;

    // Set up timeout
    const timeoutId = setTimeout(() => controller.abort(), timeout);
```

```
    return fetch(url, { ...options, signal })
        .then(response => {
            clearTimeout(timeoutId);
            return response;
        })
        .catch(error => {
            clearTimeout(timeoutId);
            if (error.name === 'AbortError') {
                throw new Error(`Request timed out after ${timeout}ms`);
            }
            throw error;
        });
}

// Usage
fetchWithTimeout('https://api.example.com/data', {}, 3000)
    .then(response => response.json())
    .then(data => console.log("Data:", data))
    .catch(error => console.error("Error:", error));
```

Asynchronous JavaScript is essential for building responsive, efficient web applications. By understanding callbacks, promises, async/await, and the Fetch API, you can handle complex asynchronous operations while maintaining clean, readable code.

8. Error Handling

Error handling is a critical aspect of JavaScript programming that helps create robust applications by gracefully managing unexpected situations. Proper error handling improves code reliability, debugging, and user experience.

try...catch Statements

The `try...catch` statement is the primary mechanism for handling exceptions in JavaScript:

```
try {
    // Code that might throw an error
    const result = riskyOperation();
    console.log(result);
} catch (error) {
```

```
    // Code to handle the error
    console.error("An error occurred:", error.message);
}
```

try...catch...finally

The `finally` block contains code that will execute regardless of whether an exception was thrown or caught:

```
try {
    console.log("Trying risky operation");
    throw new Error("Something went wrong");
} catch (error) {
    console.error("Error caught:", error.message);
} finally {
    console.log("This always executes");
}

// Output:
// Trying risky operation
// Error caught: Something went wrong
// This always executes
```

The `finally` block is useful for cleanup operations like closing files or database connections.

Nested try...catch

You can nest try...catch blocks to handle different types of errors at different levels:

```
try {
    try {
        throw new Error("Inner error");
    } catch (innerError) {
        console.error("Inner catch:", innerError.message);
        throw new Error("Rethrown error");
    }
} catch (outerError) {
    console.error("Outer catch:", outerError.message);
}
```

```
// Output:
// Inner catch: Inner error
// Outer catch: Rethrown error
```

throw Statement

The `throw` statement allows you to generate custom errors:

```
function divide(a, b) {
    if (b === 0) {
        throw new Error("Division by zero is not allowed");
    }
    return a / b;
}

try {
    const result = divide(10, 0);
    console.log(result);
} catch (error) {
    console.error("Error:", error.message);
}

// Output:
// Error: Division by zero is not allowed
```

You can throw any value, not just Error objects, but it's best practice to throw Error objects:

```
// Not recommended
throw "Something went wrong";
throw 404;

// Recommended
throw new Error("Something went wrong");
```

Error Objects

JavaScript has several built-in error types that inherit from the base Error constructor:

Error

The generic Error object:

```
const error = new Error("Something went wrong");
console.log(error.message); // "Something went wrong"
console.log(error.name);    // "Error"
console.log(error.stack);   // Stack trace as a string
```

Specific Error Types

JavaScript provides several specific error types for different scenarios:

```
// Syntax error
try {
    eval("alert('Hello world'");  // Missing closing parenthesis
} catch (error) {
    console.log(error instanceof SyntaxError); // true
    console.log(error.message);
}

// Reference error (accessing undefined variables)
try {
    console.log(undefinedVariable);
} catch (error) {
    console.log(error instanceof ReferenceError); // true
    console.log(error.message);
}

// Type error (performing invalid operations)
try {
    const num = 123;
    num.toUpperCase();
} catch (error) {
    console.log(error instanceof TypeError); // true
    console.log(error.message);
}

// Range error (numeric value outside of range)
try {
    const arr = new Array(-1); // Negative array length
} catch (error) {
```

```
    console.log(error instanceof RangeError); // true
    console.log(error.message);
}

// URI error (invalid URI functions)
try {
    decodeURIComponent('%');
} catch (error) {
    console.log(error instanceof URIError); // true
    console.log(error.message);
}
```

Custom Error Types

You can create custom error types by extending the Error class:

```
class ValidationError extends Error {
    constructor(message) {
        super(message);
        this.name = "ValidationError";
        // Maintain proper stack trace in V8 engines (Chrome, Node.js)
        if (Error.captureStackTrace) {
            Error.captureStackTrace(this, ValidationError);
        }
    }
}

class DatabaseError extends Error {
    constructor(message, code) {
        super(message);
        this.name = "DatabaseError";
        this.code = code;
        if (Error.captureStackTrace) {
            Error.captureStackTrace(this, DatabaseError);
        }
    }
}

// Usage
try {
    throw new ValidationError("Invalid email format");
} catch (error) {
    if (error instanceof ValidationError) {
```

```
        console.log("Validation error:", error.message);
    } else if (error instanceof DatabaseError) {
        console.log(`Database error (${error.code}):`, error.message);
    } else {
        console.log("Unknown error:", error);
    }
}
```

Error Handling in Asynchronous Code

Promises

With promises, you use `.catch()` to handle errors:

```
fetchData()
    .then(data => {
        console.log("Data:", data);
    })
    .catch(error => {
        console.error("Error fetching data:", error);
    });
```

Errors in promise chains propagate to the nearest catch handler:

```
fetchData()
    .then(data => {
        // This will throw an error
        return JSON.parse(data);
    })
    .then(parsedData => {
        console.log("Parsed data:", parsedData);
    })
    .catch(error => {
        // This will catch errors from fetchData() and JSON.parse()
        console.error("Error in promise chain:", error);
    });
```

Async/Await

With async/await, you use try/catch blocks:

```
async function fetchDataAndProcess() {
    try {
        const data = await fetchData();
        const parsedData = JSON.parse(data);
        console.log("Parsed data:", parsedData);
    } catch (error) {
        console.error("Error:", error);
    }
}
```

Handling Errors in Callbacks

For callback-based APIs, error handling typically involves checking for an error parameter:

```
fs.readFile('file.txt', (error, data) => {
    if (error) {
        console.error("Error reading file:", error);
        return;
    }

    console.log("File contents:", data);
});
```

Global Error Handling

Window.onerror

In browsers, you can set up a global error handler:

```
window.onerror = function(message, source, lineno, colno, error) {
    console.error("Global error handler:", message);
    console.error("Source:", source);
    console.error("Line:", lineno, "Column:", colno);
    console.error("Error object:", error);

    // Return true to prevent the default browser error handling
    return true;
};
```

unhandledrejection

For unhandled promise rejections:

```
window.addEventListener('unhandledrejection', function(event) {
    console.error("Unhandled promise rejection:", event.reason);

    // Prevent default handling
    event.preventDefault();
});
```

Node.js Process Events

In Node.js, you can handle uncaught exceptions and unhandled rejections:

```
process.on('uncaughtException', (error) => {
    console.error('Uncaught exception:', error);
    // Perform cleanup, log error, etc.
    // It's generally best to exit after an uncaught exception
    process.exit(1);
});

process.on('unhandledRejection', (reason, promise) => {
    console.error('Unhandled promise rejection:', reason);
    // Log error, perform cleanup, etc.
});
```

Error Handling Best Practices

1. Be Specific About What You Catch

Catch specific errors rather than all errors when possible:

```
try {
    // Risky code
} catch (error) {
    if (error instanceof TypeError) {
        // Handle type errors
    } else if (error instanceof NetworkError) {
        // Handle network errors
```

```
    } else {
        // Handle other errors or rethrow
        throw error;
    }
}
```

2. Don't Swallow Errors

Always do something meaningful with caught errors:

```
// Bad - error is caught but ignored
try {
    riskyOperation();
} catch (error) {
    // Empty catch block
}

// Good - error is logged and handled
try {
    riskyOperation();
} catch (error) {
    console.error("Error during risky operation:", error);
    notifyUser("An error occurred. Please try again later.");
}
```

3. Clean Up Resources in Finally Blocks

```
let connection;
try {
    connection = openDatabaseConnection();
    // Use connection
} catch (error) {
    console.error("Database error:", error);
    // Handle error
} finally {
    // This ensures the connection is closed even if an error occurs
    if (connection) {
        connection.close();
    }
}
```

4. Provide Meaningful Error Messages

```
function validateUser(user) {
    if (!user) {
        throw new Error("User object is required");
    }

    if (!user.name) {
        throw new ValidationError("User name is required");
    }

    if (!isValidEmail(user.email)) {
        throw new ValidationError(`Invalid email format: ${user.email}`)
    }
}
```

5. Use Error Boundaries in React

In React applications, use Error Boundaries to catch errors in component trees:

```
class ErrorBoundary extends React.Component {
    constructor(props) {
        super(props);
        this.state = { hasError: false };
    }

    static getDerivedStateFromError(error) {
        return { hasError: true };
    }

    componentDidCatch(error, errorInfo) {
        console.error("Component error:", error);
        console.error("Component stack:", errorInfo.componentStack);
        // Log error to an error reporting service
    }

    render() {
        if (this.state.hasError) {
            return <h1>Something went wrong.</h1>;
        }

        return this.props.children;
```

```
    }
}

// Usage
<ErrorBoundary>
    <MyComponent />
</ErrorBoundary>
```

6. Implement Retry Logic for Transient Errors

```
async function fetchWithRetry(url, options = {}, maxRetries = 3) {
    let retries = 0;

    while (retries < maxRetries) {
        try {
            return await fetch(url, options);
        } catch (error) {
            retries++;

            if (retries >= maxRetries) {
                throw error;
            }

            console.log(`Retry attempt ${retries}/${maxRetries}`);
            // Wait before retrying (exponential backoff)
            await new Promise(resolve => setTimeout(resolve, 1000 * Math
        }
    }
}
```

7. Log Errors with Context

```
function processUserData(userId, data) {
    try {
        // Process data
    } catch (error) {
        console.error("Error processing user data:", {
            userId,
            dataSize: data.length,
            error: {
                name: error.name,
```

```
                message: error.message,
                stack: error.stack
            }
        });
        throw error;
    }
}
```

Effective error handling is a crucial skill for JavaScript developers. By implementing proper error handling strategies, you can create more robust, maintainable, and user-friendly applications that gracefully handle unexpected situations.

9. ES6+ Features

ECMAScript 2015 (ES6) and subsequent versions introduced numerous powerful features that transformed JavaScript development. These modern features have made JavaScript more expressive, concise, and powerful. This section covers the most important ES6+ features that every JavaScript developer should know.

Template Literals

Template literals provide an improved way to work with strings, allowing embedded expressions and multi-line strings:

```
// String concatenation (old way)
const name = "John";
const greeting = "Hello, " + name + "!";

// Template literals (ES6)
const betterGreeting = `Hello, ${name}!`;

// Multi-line strings
const oldMultiline = "This is line 1.\n" +
                     "This is line 2.";

const newMultiline = `This is line 1.
This is line 2.`;

// Expressions in template literals
const a = 5;
const b = 10;
```

```
console.log(`The sum of ${a} and ${b} is ${a + b}.`);
// "The sum of 5 and 10 is 15."

// Tagged templates
function highlight(strings, ...values) {
    return strings.reduce((result, str, i) => {
        return result + str + (values[i] ? `<strong>${values[i]}</strong>` : '');
    }, '');
}

const name = "John";
const age = 30;
const output = highlight`My name is ${name} and I am ${age} years old.`;
// "My name is <strong>John</strong> and I am <strong>30</strong> years old."
```

Destructuring Assignment

Destructuring allows you to extract values from arrays or properties from objects into distinct variables:

Array Destructuring

```
// Old way
const numbers = [1, 2, 3];
const first = numbers[0];
const second = numbers[1];

// Array destructuring
const [a, b, c] = [1, 2, 3];
console.log(a, b, c); // 1 2 3

// Skip elements
const [x, , z] = [1, 2, 3];
console.log(x, z); // 1 3

// Rest pattern
const [head, ...tail] = [1, 2, 3, 4];
console.log(head); // 1
console.log(tail); // [2, 3, 4]

// Default values
const [p = 0, q = 0] = [1];
```

```
console.log(p, q); // 1 0

// Swapping variables
let m = 1;
let n = 2;
[m, n] = [n, m];
console.log(m, n); // 2 1
```

Object Destructuring

```
// Old way
const person = { name: "John", age: 30 };
const personName = person.name;
const personAge = person.age;

// Object destructuring
const { name, age } = person;
console.log(name, age); // "John" 30

// Assigning to different variable names
const { name: fullName, age: years } = person;
console.log(fullName, years); // "John" 30

// Default values
const { name, age, job = "Unknown" } = person;
console.log(job); // "Unknown"

// Nested destructuring
const user = {
    id: 1,
    name: "John",
    address: {
        city: "New York",
        country: "USA"
    }
};

const { name, address: { city, country } } = user;
console.log(name, city, country); // "John" "New York" "USA"

// Rest pattern with objects
const { id, ...rest } = user;
```

```
console.log(id); // 1
console.log(rest); // { name: "John", address: { city: "New York", count
```

Function Parameter Destructuring

```
// Old way
function printPerson(person) {
    console.log(person.name, person.age);
}

// With destructuring
function printPerson({ name, age }) {
    console.log(name, age);
}

printPerson({ name: "John", age: 30 }); // "John" 30

// With default values
function printPerson({ name = "Anonymous", age = 0 } = {}) {
    console.log(name, age);
}

printPerson(); // "Anonymous" 0
printPerson({ name: "John" }); // "John" 0
```

Default Parameters

Default parameters allow you to specify default values for function parameters:

```
// Old way
function greet(name) {
    name = name || "Guest";
    return "Hello, " + name;
}

// With default parameters
function greet(name = "Guest") {
    return `Hello, ${name}`;
}

console.log(greet()); // "Hello, Guest"
```

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

// Default parameters can use expressions
function getDate(date = new Date()) {
    return date.toLocaleDateString();
}

// Default parameters can reference previous parameters
function createFullName(firstName, lastName = firstName) {
    return `${firstName} ${lastName}`;
}

console.log(createFullName("John")); // "John John"
```

Spread and Rest Operators

The spread (`...`) and rest operators provide powerful ways to work with arrays and objects:

Spread Operator

The spread operator expands an iterable (like an array) into individual elements:

```
// Combining arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]

// Copying arrays
const original = [1, 2, 3];
const copy = [...original];

// Spreading into function arguments
function sum(a, b, c) {
    return a + b + c;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6

// Spreading objects (ES2018)
const person = { name: "John", age: 30 };
const employee = { ...person, jobTitle: "Developer" };
```

```
// { name: "John", age: 30, jobTitle: "Developer" }

// Overriding properties
const updated = { ...person, age: 31 };
// { name: "John", age: 31 }
```

Rest Operator

The rest operator collects multiple elements into a single array:

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

console.log(sum(1, 2, 3, 4, 5)); // 15

// Rest with other parameters
function process(first, second, ...others) {
    console.log(first); // 1
    console.log(second); // 2
    console.log(others); // [3, 4, 5]
}

process(1, 2, 3, 4, 5);

// Rest in destructuring
const [first, ...rest] = [1, 2, 3, 4, 5];
console.log(first); // 1
console.log(rest); // [2, 3, 4, 5]

const { name, ...otherProps } = { name: "John", age: 30, job: "Developer
console.log(name); // "John"
console.log(otherProps); // { age: 30, job: "Developer" }
```

Arrow Functions

Arrow functions provide a more concise syntax for writing functions and lexically bind the `this` value:

```
// Traditional function expression
const add = function(a, b) {
    return a + b;
};

// Arrow function
const add = (a, b) => a + b;

// With single parameter (parentheses optional)
const square = x => x * x;

// With no parameters (parentheses required)
const getRandomNumber = () => Math.random();

// With function body (curly braces and return required)
const calculate = (x, y) => {
    const sum = x + y;
    return sum * 2;
};

// Returning an object (parentheses required)
const createPerson = (name, age) => ({ name, age });
```

Lexical `this`

Arrow functions don't have their own `this` context; they inherit `this` from the surrounding code:

```
// Problem with traditional functions
const person = {
    name: "John",
    hobbies: ["reading", "music", "sports"],
    printHobbies: function() {
        this.hobbies.forEach(function(hobby) {
            console.log(`${this.name} likes ${hobby}`); // 'this' is unc
        });
    }
};

// Solutions with traditional functions
// 1. Store 'this' in a variable
const person1 = {
```

```
    name: "John",
    hobbies: ["reading", "music", "sports"],
    printHobbies: function() {
        const self = this;
        this.hobbies.forEach(function(hobby) {
            console.log(`${self.name} likes ${hobby}`);
        });
    }
};

// 2. Use bind()
const person2 = {
    name: "John",
    hobbies: ["reading", "music", "sports"],
    printHobbies: function() {
        this.hobbies.forEach(function(hobby) {
            console.log(`${this.name} likes ${hobby}`);
        }.bind(this));
    }
};

// Solution with arrow functions
const person3 = {
    name: "John",
    hobbies: ["reading", "music", "sports"],
    printHobbies: function() {
        this.hobbies.forEach(hobby => {
            console.log(`${this.name} likes ${hobby}`); // 'this' refers
        });
    }
};
```

Classes

ES6 introduced class syntax, providing a cleaner way to create constructor functions and manage inheritance:

```
// ES5 constructor function
function Person(name, age) {
    this.name = name;
    this.age = age;
}
```

```
Person.prototype.greet = function() {
    return `Hello, my name is ${this.name}`;
};

// ES6 class
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hello, my name is ${this.name}`;
    }

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

const john = new Person("John", 30);
console.log(john.greet()); // "Hello, my name is John"

const anonymous = Person.createAnonymous();
console.log(anonymous.name); // "Anonymous"
```

Class Inheritance

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

    introduce() {
        return `My name is ${this.name} and I am ${this.age} years old`;
    }
}

class Employee extends Person {
    constructor(name, age, jobTitle) {
```

```
        super(name, age); // Call parent constructor
        this.jobTitle = jobTitle;
    }

    // Override parent method
    introduce() {
        return `${super.introduce()} and I work as a ${this.jobTitle}`;
    }

    // Add new method
    getJobInfo() {
        return `I work as a ${this.jobTitle}`;
    }
}

const john = new Employee("John", 30, "Developer");
console.log(john.introduce()); // "My name is John and I am 30 years old
console.log(john.getJobInfo()); // "I work as a Developer"
```

Getters and Setters

```
class Person {
    constructor(firstName, lastName) {
        this._firstName = firstName;
        this._lastName = lastName;
    }

    get firstName() {
        return this._firstName;
    }

    set firstName(value) {
        this._firstName = value;
    }

    get lastName() {
        return this._lastName;
    }

    set lastName(value) {
        this._lastName = value;
    }
```

```
    get fullName() {
        return `${this._firstName} ${this._lastName}`;
    }

    set fullName(value) {
        const parts = value.split(" ");
        this._firstName = parts[0];
        this._lastName = parts[1];
    }
}

const john = new Person("John", "Doe");
console.log(john.fullName); // "John Doe"
john.fullName = "Jane Smith";
console.log(john.firstName); // "Jane"
console.log(john.lastName); // "Smith"
```

Private Class Fields (ES2022)

```
class BankAccount {
    // Public field
    owner;

    // Private field
    #balance;

    constructor(owner, initialBalance) {
        this.owner = owner;
        this.#balance = initialBalance;
    }

    // Private method
    #validateAmount(amount) {
        return amount > 0;
    }

    deposit(amount) {
        if (this.#validateAmount(amount)) {
            this.#balance += amount;
            return true;
        }
        return false;
    }
```

```
    withdraw(amount) {
        if (this.#validateAmount(amount) && amount <= this.#balance) {
            this.#balance -= amount;
            return true;
        }
        return false;
    }

    get balance() {
        return this.#balance;
    }
}

const account = new BankAccount("John", 1000);
console.log(account.balance); // 1000
account.deposit(500);
console.log(account.balance); // 1500
// console.log(account.#balance); // SyntaxError: Private field '#balanc
```

Modules (import/export)

ES6 modules provide a standard way to organize and share code between JavaScript files:

Named Exports/Imports

```
// math.js
export const PI = 3.14159;

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

export function subtract(a, b) {
    return a - b;
}

// app.js
import { PI, add, subtract } from './math.js';
```

```
console.log(PI); // 3.14159
console.log(add(5, 3)); // 8
```

Default Exports/Imports

```
// person.js
export default class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        return `Hello, my name is ${this.name}`;
    }
}

// app.js
import Person from './person.js';

const john = new Person("John");
console.log(john.greet()); // "Hello, my name is John"
```

Mixed Exports/Imports

```
// utils.js
export const VERSION = '1.0.0';

export function formatDate(date) {
    return date.toLocaleDateString();
}

export default class Utils {
    static generateId() {
        return Math.random().toString(36).substr(2, 9);
    }
}

// app.js
import Utils, { VERSION, formatDate } from './utils.js';

console.log(VERSION); // "1.0.0"
```

```
console.log(formatDate(new Date())); // "4/7/2025" (depends on locale)
console.log(Utils.generateId()); // Random ID
```

Renaming Imports/Exports

```
// math.js
export const PI = 3.14159;
export const E = 2.71828;

// app.js
import { PI as pi, E as e } from './math.js';

console.log(pi); // 3.14159
console.log(e); // 2.71828
```

Import All

```
// math.js
export const PI = 3.14159;
export function add(a, b) { return a + b; }
export function subtract(a, b) { return a - b; }

// app.js
import * as math from './math.js';

console.log(math.PI); // 3.14159
console.log(math.add(5, 3)); // 8
```

Map and Set

ES6 introduced new data structures: Map, Set, WeakMap, and WeakSet.

Map

Map is a collection of key-value pairs where keys can be of any type:

```
// Creating a Map
const userRoles = new Map();
```

```
// Adding entries
userRoles.set('john', 'admin');
userRoles.set('jane', 'editor');
userRoles.set('bob', 'subscriber');

// Alternative initialization
const userRoles = new Map([
    ['john', 'admin'],
    ['jane', 'editor'],
    ['bob', 'subscriber']
]);

// Getting values
console.log(userRoles.get('john')); // "admin"

// Checking if a key exists
console.log(userRoles.has('jane')); // true
console.log(userRoles.has('alice')); // false

// Size
console.log(userRoles.size); // 3

// Deleting entries
userRoles.delete('bob');
console.log(userRoles.size); // 2

// Clearing the map
userRoles.clear();
console.log(userRoles.size); // 0

// Objects as keys
const userMap = new Map();
const john = { name: 'John' };
const jane = { name: 'Jane' };

userMap.set(john, { role: 'admin', active: true });
userMap.set(jane, { role: 'editor', active: false });

console.log(userMap.get(john)); // { role: 'admin', active: true }
```

Set

Set is a collection of unique values:

```
// Creating a Set
const uniqueNumbers = new Set();

// Adding values
uniqueNumbers.add(1);
uniqueNumbers.add(2);
uniqueNumbers.add(3);
uniqueNumbers.add(1); // Ignored (already exists)

// Alternative initialization
const uniqueNumbers = new Set([1, 2, 3, 1, 2]); // Duplicates are ignore

// Checking if a value exists
console.log(uniqueNumbers.has(2)); // true
console.log(uniqueNumbers.has(4)); // false

// Size
console.log(uniqueNumbers.size); // 3

// Deleting values
uniqueNumbers.delete(2);
console.log(uniqueNumbers.size); // 2

// Clearing the set
uniqueNumbers.clear();
console.log(uniqueNumbers.size); // 0

// Practical use: removing duplicates from an array
const numbers = [1, 2, 3, 4, 2, 3, 5];
const uniqueNumbers = [...new Set(numbers)];
console.log(uniqueNumbers); // [1, 2, 3, 4, 5]
```

WeakMap and WeakSet

WeakMap and WeakSet are similar to Map and Set, but they hold "weak" references to objects:

```
// WeakMap
const weakMap = new WeakMap();
let obj = { name: 'John' };

weakMap.set(obj, 'metadata');
```

```
console.log(weakMap.get(obj)); // "metadata"

obj = null; // The entry in weakMap will be garbage collected

// WeakSet
const weakSet = new WeakSet();
let user = { name: 'John' };

weakSet.add(user);
console.log(weakSet.has(user)); // true

user = null; // The entry in weakSet will be garbage collected
```

Symbol

Symbol is a primitive data type that represents a unique identifier:

```
// Creating symbols
const sym1 = Symbol();
const sym2 = Symbol('description');
const sym3 = Symbol('description'); // Different from sym2

console.log(sym2 === sym3); // false

// Using symbols as object keys
const user = {
    name: 'John',
    [Symbol('id')]: 123,
    [Symbol('secret')]: 'password'
};

console.log(user.name); // "John"
console.log(Object.keys(user)); // ["name"] (symbols are not enumerable)

// Well-known symbols
const iterable = {
    [Symbol.iterator]: function* () {
        yield 1;
        yield 2;
        yield 3;
    }
};
```

```
for (const value of iterable) {
    console.log(value); // 1, 2, 3
}
```

Iterators and Generators

Iterators

An iterator is an object that provides a `next()` method which returns an object with `value` and `done` properties:

```
// Custom iterator
function createIterator(array) {
    let index = 0;

    return {
        next: function() {
            return index < array.length ?
                { value: array[index++], done: false } :
                { done: true };
        }
    };
}

const iterator = createIterator([1, 2, 3]);
console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { done: true }

// Iterable object
const iterable = {
    [Symbol.iterator]: function() {
        let i = 1;
        return {
            next: function() {
                return i <= 3 ?
                    { value: i++, done: false } :
                    { done: true };
            }
        };
    }
```

```
};

for (const value of iterable) {
    console.log(value); // 1, 2, 3
}
```

Generators

Generators are functions that can be paused and resumed, making it easier to create iterators:

```
// Generator function
function* numberGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

const generator = numberGenerator();
console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: false }
console.log(generator.next()); // { done: true }

// Using generators in for...of loops
for (const num of numberGenerator()) {
    console.log(num); // 1, 2, 3
}

// Infinite generator
function* infiniteSequence() {
    let i = 0;
    while (true) {
        yield i++;
    }
}

const infinite = infiniteSequence();
console.log(infinite.next().value); // 0
console.log(infinite.next().value); // 1
console.log(infinite.next().value); // 2

// Generator with parameters
```

```
function* conversation() {
    const name = yield "What's your name?";
    const hobby = yield `Hello, ${name}! What's your hobby?`;
    yield `${name} likes ${hobby}. That's cool!`;
}

const talk = conversation();
console.log(talk.next().value); // "What's your name?"
console.log(talk.next('John').value); // "Hello, John! What's your hobby
console.log(talk.next('coding').value); // "John likes coding. That's co
```

Optional Chaining

Optional chaining (`?.`) allows you to access deeply nested object properties without worrying about whether the property exists:

```
// Without optional chaining
function getUserCity(user) {
    if (user && user.address && user.address.city) {
        return user.address.city;
    }
    return undefined;
}

// With optional chaining
function getUserCity(user) {
    return user?.address?.city;
}

const user1 = {
    name: 'John',
    address: {
        city: 'New York',
        country: 'USA'
    }
};

const user2 = {
    name: 'Jane'
    // No address
};
```

```
console.log(getUserCity(user1)); // "New York"
console.log(getUserCity(user2)); // undefined
console.log(getUserCity(null)); // undefined

// Optional chaining with method calls
const result = obj?.method?.();

// Optional chaining with array elements
const value = arr?.[0];
```

Nullish Coalescing Operator

The nullish coalescing operator (`??`) provides a way to specify a default value when a value is `null` or `undefined`:

```
// Using logical OR (||)
// Problem: treats 0, '', false as falsy values
function getDisplayName(user) {
    return user.displayName || user.firstName || 'Anonymous';
}

// Using nullish coalescing
function getDisplayName(user) {
    return user.displayName ?? user.firstName ?? 'Anonymous';
}

const user1 = { displayName: '', firstName: 'John' };
console.log(getDisplayName(user1)); // "" (with ??) vs "John" (with ||)

const user2 = { displayName: 0, firstName: 'Jane' };
console.log(getDisplayName(user2)); // 0 (with ??) vs "Jane" (with ||)
```

Other ES6+ Features

Object.entries(), Object.values(), Object.fromEntries()

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

// Object.entries() - returns an array of [key, value] pairs
const entries = Object.entries(person);
```

```
console.log(entries);
// [["name", "John"], ["age", 30], ["job", "Developer"]]

// Object.values() - returns an array of values
const values = Object.values(person);
console.log(values);
// ["John", 30, "Developer"]

// Object.fromEntries() - converts entries back to an object
const newObj = Object.fromEntries(entries);
console.log(newObj);
// { name: "John", age: 30, job: "Developer" }
```

Array.flat() and Array.flatMap()

```
// flat() - flattens nested arrays
const nestedArray = [1, 2, [3, 4, [5, 6]]];
console.log(nestedArray.flat()); // [1, 2, 3, 4, [5, 6]]
console.log(nestedArray.flat(2)); // [1, 2, 3, 4, 5, 6]

// flatMap() - maps each element and flattens the result
const sentences = ["Hello world", "How are you"];
const words = sentences.flatMap(sentence => sentence.split(" "));
console.log(words); // ["Hello", "world", "How", "are", "you"]
```

BigInt

```
// BigInt for large integers
const bigNumber = 9007199254740991n; // 'n' suffix creates a BigInt
const anotherBigNumber = BigInt("9007199254740991");

console.log(bigNumber + 1n); // 9007199254740992n
console.log(typeof bigNumber); // "bigint"
```

globalThis

```
// Unified way to access the global object
console.log(globalThis); // Window in browsers, global in Node.js
```

Promise.allSettled()

```
const promises = [
    Promise.resolve("Success"),
    Promise.reject("Error"),
    Promise.resolve("Another success")
];

Promise.allSettled(promises).then(results => {
    console.log(results);
    // [
    //   { status: "fulfilled", value: "Success" },
    //   { status: "rejected", reason: "Error" },
    //   { status: "fulfilled", value: "Another success" }
    // ]
});
```

String.prototype.replaceAll()

```
const text = "apple orange apple banana apple";
const replaced = text.replaceAll("apple", "fruit");
console.log(replaced); // "fruit orange fruit banana fruit"
```

Logical Assignment Operators

```
// ||= (logical OR assignment)
let a = null;
a ||= 10; // a = a || 10
console.log(a); // 10

// &&= (logical AND assignment)
let b = 5;
b &&= 10; // b = b && 10
console.log(b); // 10

// ??= (nullish coalescing assignment)
let c = null;
c ??= 10; // c = c ?? 10
console.log(c); // 10
```

These ES6+ features have significantly improved JavaScript development, making code more concise, readable, and powerful. Understanding and using these features is essential for modern JavaScript development.

10. JavaScript Best Practices

Writing clean, maintainable, and efficient JavaScript code is essential for successful projects. This section covers best practices that will help you write better JavaScript code, avoid common pitfalls, and improve the overall quality of your applications.

Code Organization

Proper code organization is crucial for maintainability, especially as projects grow in size and complexity.

Modular Code Structure

Break your code into small, focused modules that each handle a specific responsibility:

```
// Bad: One large file with multiple responsibilities
const app = {
    data: [],
    init() { /* ... */ },
    fetchData() { /* ... */ },
    renderUI() { /* ... */ },
    handleEvents() { /* ... */ },
    utilities: {
        formatDate() { /* ... */ },
        calculateTotal() { /* ... */ }
    }
};

// Good: Separate modules with clear responsibilities
// data-service.js
export const dataService = {
    fetchData() { /* ... */ },
    processData() { /* ... */ }
};

// ui-controller.js
import { dataService } from './data-service.js';
```

```
export const uiController = {
    renderUI() { /* ... */ },
    updateView() { /* ... */ }
};

// event-handler.js
import { uiController } from './ui-controller.js';
export const eventHandler = {
    setupEventListeners() { /* ... */ },
    handleClick() { /* ... */ }
};

// utilities.js
export const formatDate = () => { /* ... */ };
export const calculateTotal = () => { /* ... */ };

// app.js
import { dataService } from './data-service.js';
import { uiController } from './ui-controller.js';
import { eventHandler } from './event-handler.js';

const app = {
    init() {
        dataService.fetchData();
        uiController.renderUI();
        eventHandler.setupEventListeners();
    }
};
```

Design Patterns

Use established design patterns to solve common problems:

Module Pattern

Encapsulates private variables and functions:

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

    // Public API
    return {
```

```
        increment() {
            count++;
            return count;
        },
        decrement() {
            count--;
            return count;
        },
        getCount() {
            return count;
        }
    };
})();
```

Factory Pattern

Creates objects without specifying the exact class:

```
function createUser(name, role) {
    return {
        name,
        role,
        permissions: getPermissionsForRole(role),
        createdAt: new Date()
    };
}

const admin = createUser('John', 'admin');
const editor = createUser('Jane', 'editor');
```

Observer Pattern

Allows objects to subscribe to events:

```
class EventEmitter {
    constructor() {
        this.events = {};
    }

    on(event, listener) {
        if (!this.events[event]) {
            this.events[event] = [];
```

```
        }
        this.events[event].push(listener);
    }

    emit(event, ...args) {
        if (this.events[event]) {
            this.events[event].forEach(listener => listener(...args));
        }
    }

    off(event, listener) {
        if (this.events[event]) {
            this.events[event] = this.events[event].filter(l => l !== li
        }
    }
}

// Usage
const emitter = new EventEmitter();
emitter.on('userCreated', user => console.log(`New user: ${user.name}`))
emitter.emit('userCreated', { name: 'John' });
```

Project Structure

Organize files in a logical directory structure:

```
project/
├── src/
│   ├── components/
│   │   ├── header/
│   │   │   ├── header.js
│   │   │   └── header.css
│   │   └── footer/
│   │       ├── footer.js
│   │       └── footer.css
│   ├── services/
│   │   ├── api-service.js
│   │   └── auth-service.js
│   ├── utils/
│   │   ├── date-utils.js
│   │   └── string-utils.js
│   └── app.js
├── tests/
```

```
│   ├── components/
│   ├── services/
│   └── utils/
├── dist/
├── package.json
└── README.md
```

Naming Conventions

Consistent naming makes code more readable and easier to understand.

Variables and Functions

- Use camelCase for variables and functions
- Use descriptive names that explain purpose
- Avoid single-letter names except for counters or temporary variables

```
// Bad
const x = getUserData();
function calc(a, b) { return a + b; }

// Good
const userData = getUserData();
function calculateTotal(price, tax) { return price + tax; }
```

Constants

Use UPPERCASE_WITH_UNDERSCORES for constants:

```
const MAX_ITEMS_PER_PAGE = 20;
const API_BASE_URL = 'https://api.example.com';
```

Classes and Constructors

Use PascalCase for classes and constructor functions:

```
class UserProfile {
    constructor(user) {
        this.user = user;
```

```
    }
}

function ShoppingCart() {
    this.items = [];
}
```

Private Properties and Methods

Use underscore prefix for private properties and methods:

```
class User {
    constructor(name) {
        this.name = name;
        this._password = null; // Private property
    }

    _hashPassword(password) { // Private method
        // Hash the password
    }

    setPassword(password) {
        this._password = this._hashPassword(password);
    }
}
```

Boolean Variables

Prefix boolean variables with "is", "has", or "should":

```
const isActive = true;
const hasPermission = checkPermission(user);
const shouldRedirect = status === 404;
```

File Naming

- Use kebab-case for file names: `user-profile.js`
- Use consistent extensions: `.js` for JavaScript, `.jsx` for React components
- Group related files: `user-profile.js`, `user-profile.test.js`, `user-profile.css`

Performance Optimization

Optimizing JavaScript performance is crucial for creating responsive applications.

Minimize DOM Manipulation

DOM operations are expensive. Minimize them by:

1. Batching DOM updates
2. Using document fragments
3. Modifying elements before adding them to the DOM

```
// Bad: Multiple DOM operations
for (let i = 0; i < 100; i++) {
    const listItem = document.createElement('li');
    listItem.textContent = `Item ${i}`;
    document.getElementById('myList').appendChild(listItem);
}

// Good: Using document fragment
const fragment = document.createDocumentFragment();
for (let i = 0; i < 100; i++) {
    const listItem = document.createElement('li');
    listItem.textContent = `Item ${i}`;
    fragment.appendChild(listItem);
}
document.getElementById('myList').appendChild(fragment);
```

Optimize Loops

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

// Bad: Recalculating length in each iteration
for (let i = 0; i < arr.length; i++) {
    // Do something
}

// Good: Cache the length
for (let i = 0, len = arr.length; i < len; i++) {
    // Do something
}
```

```
// Better: Use for...of for arrays
for (const item of arr) {
    // Do something
}

// For objects, use for...in
const obj = { a: 1, b: 2, c: 3 };
for (const key in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, key)) {
        // Do something with obj[key]
    }
}
```

Debounce and Throttle

Limit the rate at which functions execute, especially for expensive operations:

```
// Debounce: Execute after a delay with no new calls
function debounce(func, delay) {
    let timeoutId;
    return function(...args) {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => func.apply(this, args), delay);
    };
}

// Throttle: Execute at most once per specified period
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);
```

```
window.addEventListener('input', debouncedSearch);
window.addEventListener('scroll', throttledScroll);
```

Memoization

Cache results of expensive function calls:

```
function memoize(func) {
    const cache = new Map();
    return function(...args) {
        const key = JSON.stringify(args);
        if (cache.has(key)) {
            return cache.get(key);
        }
        const result = func.apply(this, args);
        cache.set(key, result);
        return result;
    };
}

// Usage
const expensiveFunction = (a, b) => {
    console.log('Computing...');
    return a * b;
};

const memoizedFunction = memoize(expensiveFunction);

console.log(memoizedFunction(4, 2)); // Computing... 8
console.log(memoizedFunction(4, 2)); // 8 (from cache)
```

Avoid Memory Leaks

Common causes of memory leaks:

1. **Forgotten event listeners**:

```
// Bad: Event listener not removed when component is destroyed
element.addEventListener('click', onClick);

// Good: Store reference and remove when needed
element.addEventListener('click', onClick);
```

```
// Later:
element.removeEventListener('click', onClick);
```

1. **Closures capturing large objects**:

```
// Bad: Closure captures the entire data array
function processData(data) {
    return function() {
        return data.length; // Keeps the entire data array in memory
    };
}

// Good: Only capture what you need
function processData(data) {
    const length = data.length;
    return function() {
        return length; // Only keeps the length value
    };
}
```

1. **Circular references**:

```
// Bad: Circular reference
function createNode() {
    const node = {};
    const child = { parent: node };
    node.child = child; // Circular reference
    return node;
}

// Good: Avoid circular references or use WeakMap/WeakSet
function createNode() {
    const node = {};
    const child = { data: 'child data' };
    node.child = child;
    return { node, child };
}
```

Memory Management

JavaScript has automatic garbage collection, but understanding memory management is still important.

Garbage Collection Basics

JavaScript uses two main garbage collection strategies: 1. **Reference counting**: Counts references to objects 2. **Mark and sweep**: Identifies and removes unreachable objects

Efficient Object Creation

```
// Bad: Creating many objects
function createPoints(n) {
    const points = [];
    for (let i = 0; i < n; i++) {
        points.push({ x: i, y: i * 2 });
    }
    return points;
}

// Better: Object pooling for frequently created/destroyed objects
class ObjectPool {
    constructor(createFn, initialSize = 10) {
        this.createFn = createFn;
        this.pool = Array(initialSize).fill().map(() => createFn());
    }

    get() {
        return this.pool.length > 0 ? this.pool.pop() : this.createFn();
    }

    release(obj) {
        this.pool.push(obj);
    }
}

// Usage
const pointPool = new ObjectPool(() => ({ x: 0, y: 0 }));
const point = pointPool.get();
point.x = 10;
point.y = 20;
```

```
// Use point...
pointPool.release(point);
```

Avoid Global Variables

Global variables stay in memory for the lifetime of the application:

```
// Bad: Global variables
let data = fetchData();
let cache = {};

// Good: Encapsulate in modules or functions
function processData() {
    const data = fetchData();
    const cache = {};
    // Process data using local variables
}
```

Security Considerations

Security is a critical aspect of JavaScript development.

Content Security Policy (CSP)

CSP helps prevent cross-site scripting (XSS) attacks:

```
<!-- Add to HTML header -->
<meta http-equiv="Content-Security-Policy" content="default-src 'self';
```

Prevent XSS Attacks

Cross-site scripting occurs when untrusted data is included in a web page:

```
// Bad: Directly inserting user input into the DOM
element.innerHTML = userInput;

// Good: Sanitize input
function sanitizeHTML(text) {
    const element = document.createElement('div');
```

```
    element.textContent = text;
    return element.innerHTML;
}

element.innerHTML = sanitizeHTML(userInput);

// Better: Use textContent instead of innerHTML when possible
element.textContent = userInput;
```

Avoid eval()

The `eval()` function executes arbitrary code and is a security risk:

```
// Bad: Using eval
function calculateFromUserInput(input) {
    return eval(input); // Security risk!
}

// Good: Use safer alternatives
function calculateFromUserInput(input) {
    // Use a proper parser or restrict to safe operations
    return Function('"use strict"; return (' + input + ')')();
}

// Best: Avoid dynamic code execution entirely
// Use a proper math library or parser
```

Secure Cookies

When using cookies, set appropriate security flags:

```
// Set secure cookies
document.cookie = "session=123; Secure; HttpOnly; SameSite=Strict";
```

Validate User Input

Always validate user input on both client and server:

```
function validateEmail(email) {
    const regex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
```

```
    return regex.test(email);
}

function validateForm() {
    const email = document.getElementById('email').value;
    if (!validateEmail(email)) {
        showError('Please enter a valid email address');
        return false;
    }
    return true;
}
```

Debugging Techniques

Effective debugging is essential for identifying and fixing issues in JavaScript code.

Console Methods

Beyond `console.log()`, there are many useful console methods:

```
// Grouping related logs
console.group('User Details');
console.log('Name:', user.name);
console.log('Email:', user.email);
console.groupEnd();

// Timing operations
console.time('Operation');
performOperation();
console.timeEnd('Operation');

// Tabular data
console.table([
    { name: 'John', age: 30 },
    { name: 'Jane', age: 25 }
]);

// Conditional logging
console.assert(x > 0, 'x must be positive');

// Stack traces
console.trace('Tracing function calls');
```

```
// Styling console output
console.log('%cHello World', 'color: blue; font-size: 20px');
```

Breakpoints

Use breakpoints in browser developer tools to pause execution:

1. **Line breakpoints**: Set by clicking on the line number in dev tools
2. **Conditional breakpoints**: Break only when a condition is true
3. **XHR breakpoints**: Break when an AJAX request is made
4. **Event listener breakpoints**: Break when specific events occur

debugger Statement

Insert the `debugger` statement to create a breakpoint in code:

```
function calculateTotal(items) {
    let total = 0;
    for (const item of items) {
        debugger; // Execution will pause here when dev tools are open
        total += item.price * item.quantity;
    }
    return total;
}
```

Source Maps

Source maps help debug minified code by mapping it back to the original source:

```
// In webpack config
module.exports = {
    // ...
    devtool: 'source-map',
    // ...
};
```

Testing JavaScript Code

Testing is crucial for ensuring code quality and preventing regressions.

Types of Tests

1. **Unit Tests**: Test individual functions or components
2. **Integration Tests**: Test how components work together
3. **End-to-End Tests**: Test the entire application flow

Testing Frameworks

Popular JavaScript testing frameworks include: - Jest - Mocha - Jasmine - Cypress (for E2E testing)

Writing Testable Code

```
// Hard to test
function fetchAndProcessUserData() {
    const data = fetch('/api/users').then(res => res.json());
    return processData(data);
}

// More testable: Separate concerns
async function fetchUserData() {
    const response = await fetch('/api/users');
    return response.json();
}

function processUserData(data) {
    // Process the data
    return transformedData;
}

// Now you can test processUserData independently
```

Example Unit Test (Jest)

```
// math.js
export function add(a, b) {
```

```
    return a + b;
}

// math.test.js
import { add } from './math';

describe('Math functions', () => {
    test('add correctly adds two numbers', () => {
        expect(add(1, 2)).toBe(3);
        expect(add(-1, 1)).toBe(0);
        expect(add(0, 0)).toBe(0);
    });
});
```

Test-Driven Development (TDD)

Follow the TDD cycle: 1. Write a failing test 2. Write the minimum code to make the test pass 3. Refactor the code 4. Repeat

Code Quality Tools

Use tools to enforce code quality and consistency.

Linters

ESLint helps identify and fix code issues:

```
// .eslintrc.js
module.exports = {
    "env": {
        "browser": true,
        "es2021": true
    },
    "extends": "eslint:recommended",
    "parserOptions": {
        "ecmaVersion": 12,
        "sourceType": "module"
    },
    "rules": {
        "indent": ["error", 4],
        "semi": ["error", "always"],
        "no-unused-vars": "warn"
```

```
    }
};
```

Code Formatters

Prettier automatically formats code for consistency:

```
// .prettierrc
{
    "printWidth": 80,
    "tabWidth": 2,
    "useTabs": false,
    "semi": true,
    "singleQuote": true,
    "trailingComma": "es5",
    "bracketSpacing": true,
    "arrowParens": "avoid"
}
```

Type Checking

TypeScript or JSDoc comments provide type checking:

```
// With TypeScript
function add(a: number, b: number): number {
    return a + b;
}

// With JSDoc
/**
 * Adds two numbers together
 * @param {number} a - The first number
 * @param {number} b - The second number
 * @returns {number} The sum of a and b
 */
function add(a, b) {
    return a + b;
}
```

Documentation

Good documentation makes code more maintainable and easier for others to use.

JSDoc Comments

Use JSDoc to document functions, classes, and modules:

```
/**
 * Represents a user in the system
 * @class
 */
class User {
    /**
     * Create a user
     * @param {string} name - The user's name
     * @param {string} email - The user's email
     * @param {Object} [options] - Optional settings
     * @param {boolean} [options.isAdmin=false] - Whether the user is ar
     */
    constructor(name, email, options = {}) {
        this.name = name;
        this.email = email;
        this.isAdmin = options.isAdmin || false;
    }

    /**
     * Get the user's display name
     * @returns {string} The formatted display name
     */
    getDisplayName() {
        return `${this.name} <${this.email}>`;
    }
}
```

README Files

Create comprehensive README files for projects:

```
# Project Name

Brief description of the project.
```

````
## Installation

```bash
npm install project-name
````

Usage

```
const project = require('project-name');
project.doSomething();
```

API

doSomething()

Description of the function...

License

MIT

````
### Code Comments

Write meaningful comments that explain "why" rather than "what":

```javascript
// Bad: Explains what the code does (obvious from the code itself)
// Increment the counter
counter++;

// Good: Explains why the code is necessary
// Increment the retry counter to prevent infinite loops
retryCounter++;
````

Browser Compatibility

Ensure your code works across different browsers and environments.

Feature Detection

Check if a feature is available before using it:

```
// Bad: Assuming a feature exists
document.querySelector('.element').classList.add('active');

// Good: Feature detection
if (document.querySelector && element.classList) {
    document.querySelector('.element').classList.add('active');
} else {
    // Fallback
    const element = document.getElementsByClassName('element')[0];
    element.className += ' active';
}
```

Polyfills

Add polyfills for missing features:

```
// Polyfill for Array.from
if (!Array.from) {
    Array.from = function(arrayLike) {
        return [].slice.call(arrayLike);
    };
}

// Using a polyfill library like core-js
import 'core-js/stable';
import 'regenerator-runtime/runtime';
```

Transpiling

Use Babel to transpile modern JavaScript to compatible versions:

```
// .babelrc
{
    "presets": [
        ["@babel/preset-env", {
            "targets": "> 0.25%, not dead"
        }]
    ]
}
```

Performance Monitoring

Monitor and optimize performance in production.

Web Vitals

Track Core Web Vitals metrics: - Largest Contentful Paint (LCP) - First Input Delay (FID) - Cumulative Layout Shift (CLS)

```
import {getLCP, getFID, getCLS} from 'web-vitals';

function sendToAnalytics({name, delta, id}) {
    // Send metrics to your analytics service
    console.log(name, delta, id);
}

getCLS(sendToAnalytics);
getFID(sendToAnalytics);
getLCP(sendToAnalytics);
```

Performance API

Use the Performance API to measure code execution:

```
// Mark the start of an operation
performance.mark('operationStart');

// Perform the operation
doSomething();

// Mark the end of the operation
```

```
performance.mark('operationEnd');

// Measure the duration
performance.measure('operation', 'operationStart', 'operationEnd');

// Get all measurements
const measurements = performance.getEntriesByType('measure');
console.log(measurements);
```

Lighthouse

Use Lighthouse to audit your web application: - Performance - Accessibility - Best Practices - SEO - Progressive Web App

Accessibility

Make your JavaScript applications accessible to all users.

Focus Management

Manage focus for keyboard navigation:

```
// Set focus to an element
document.getElementById('myButton').focus();

// Create a focus trap for modals
function trapFocus(element) {
    const focusableElements = element.querySelectorAll(
        'button, [href], input, select, textarea, [tabindex]:not([tabinc
    );

    const firstElement = focusableElements[0];
    const lastElement = focusableElements[focusableElements.length - 1];

    element.addEventListener('keydown', function(e) {
        if (e.key === 'Tab') {
            if (e.shiftKey && document.activeElement === firstElement) {
                e.preventDefault();
                lastElement.focus();
            } else if (!e.shiftKey && document.activeElement === lastEle
                e.preventDefault();
                firstElement.focus();
```

```
            }
        }
    });
}
```

ARIA Attributes

Use ARIA attributes to improve accessibility:

```
// Toggle a dropdown
function toggleDropdown(button, menu) {
    const isExpanded = button.getAttribute('aria-expanded') === 'true';

    button.setAttribute('aria-expanded', !isExpanded);
    menu.hidden = isExpanded;

    if (!isExpanded) {
        // Focus the first item when opening
        menu.querySelector('a, button').focus();
    }
}
```

Screen Reader Announcements

Use ARIA live regions for dynamic content:

```
<div aria-live="polite" id="announcer" class="sr-only"></div>
```

```
function announce(message) {
    const announcer = document.getElementById('announcer');
    announcer.textContent = message;
}

// Usage
announce('New items have been loaded');
```

By following these best practices, you'll write JavaScript code that is more maintainable, performant, secure, and accessible. These practices will help you avoid common pitfalls and create higher-quality applications.