Complete JavaScript Interview Guide

1. Grammar and Types

Basics

Theory: JavaScript is a dynamically typed, interpreted language. Variables are declared with `var`, `let`, or `const`.

Interview Tip: Explain the differences between var (function-scoped), let (block-scoped), and const (block-scoped, immutable binding).

```javascript
// Variable declarations
var a = 1;       // Function scoped, can be redeclared
let b = 2;       // Block scoped, cannot be redeclared
const c = 3;     // Block scoped, immutable binding

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

Comments

Theory: JavaScript supports single-line (//) and multi-line (/* */) comments.

Interview Tip: Mention JSDoc comments for documentation.

```javascript
```

```javascript
// Single line comment

/*
 * Multi-line comment
 * Used for longer explanations
 */

/**
 * JSDoc comment for functions
 * @param {number} a - First parameter
 * @returns {number} Sum
 */
function add(a, b) { return a + b; }
```

Data Structures and Types

Theory: JavaScript has 8 data types: 7 primitive (undefined, null, boolean, number, string, symbol, bigint) and 1 non-primitive (object).

Interview Tip: Explain typeof operator quirks and type coercion.

javascript

```javascript
// Primitive types
let num = 42;              // number
let str = "hello";         // string
let bool = true;           // boolean
let undef;                 // undefined
let nothing = null;        // object (quirk!)
let sym = Symbol('id');    // symbol
let big = 123n;            // bigint

// Non-primitive
let obj = { name: "John" };    // object
```

Literals

Theory: Literals are fixed values written directly in code.

Interview Tip: Show different ways to create objects and arrays.

javascript

```javascript
// Different literal types
const num = 42;              // Number literal
const str = 'Hello';         // String literal
const arr = [1, 2, 3];       // Array literal
const obj = { x: 1, y: 2 };  // Object literal
const regex = /ab+c/;        // Regex literal
const bool = true;           // Boolean literal
```

2. Control Flow and Error Handling

Block Statements

Theory: Block statements group zero or more statements using curly braces.

Interview Tip: Explain block scope with let/const vs function scope with var.

javascript

```javascript
{
  let blockScoped = "only available in this block";
  var functionScoped = "available in entire function";
}

console.log(functionScoped);   // Works
// console.log(blockScoped);   // ReferenceError
```

Conditional Statements

Theory: if/else, switch statements control program flow based on conditions.

Interview Tip: Discuss truthy/falsy values and strict equality.

javascript

```javascript
// if/else with truthy/falsy
if (0) console.log("Never runs");      // 0 is falsy
if ("") console.log("Never runs");     // Empty string is falsy
if ([]) console.log("Always runs");    // Empty array is truthy

// Switch with fall-through
switch (day) {
  case 'Mon':
  case 'Tue': console.log("Weekday"); break;
  default: console.log("Other");
}
```

Exception Handling

Theory: try/catch/finally blocks handle runtime errors gracefully.

Interview Tip: Explain error propagation and custom error types.

javascript

```javascript
try {
  throw new Error("Custom error message");
} catch (error) {
  console.log("Caught:", error.message);
} finally {
  console.log("Always executes");
}

// Custom error
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomError";
  }
}
```

3. Loops and Iteration

For Statement

Theory: Classic for loop with initialization, condition, and increment.

Interview Tip: Show different variations and common pitfalls.

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

// Multiple variables
for (let i = 0, j = 10; i < 5; i++, j--) {
  console.log(i, j);           // 0 10, 1 9, 2 8, etc.
}
```

While and Do-While

Theory: While checks condition before execution, do-while checks after.

Interview Tip: Explain when to use each type.

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

// do-while (executes at least once)
let j = 10;
do {
  console.log(j);              // 10 (runs once even though condition is false)
} while (j < 5);
```

For...in and For...of

Theory: for...in iterates over enumerable properties, for...of iterates over iterable values.

Interview Tip: Explain the key differences and when to use each.

```javascript
```

```javascript
const obj = { a: 1, b: 2, c: 3 };
const arr = [10, 20, 30];

// for...in (gets keys/indices)
for (let key in obj) console.log(key);   // a, b, c
for (let index in arr) console.log(index); // 0, 1, 2

// for...of (gets values)
for (let value of arr) console.log(value); // 10, 20, 30
// for (let value of obj) // Error: obj is not iterable
```

Break and Continue

Theory: break exits the loop, continue skips current iteration.

Interview Tip: Show usage with labels for nested loops.

javascript

```javascript
// break and continue
for (let i = 0; i < 10; i++) {
  if (i === 3) continue;          // Skip 3
  if (i === 7) break;             // Exit at 7
  console.log(i);                 // 0, 1, 2, 4, 5, 6
}

// Labeled break for nested loops
outer: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) break outer;
  }
}
```

4. Functions

Defining and Calling Functions

Theory: Functions are first-class objects in JavaScript. Multiple ways to define them.

Interview Tip: Explain function hoisting differences between declarations and expressions.

javascript

```javascript
// Function declaration (hoisted)
function add(a, b) {
  return a + b;
}

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

// Function call
console.log(add(5, 3));          // 8
console.log(multiply(4, 2));     // 8
```

Function Scopes and Closures

Theory: Functions create their own scope. Closures allow inner functions to access outer function's variables.

Interview Tip: This is a crucial concept. Explain practical uses like data privacy and callbacks.

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

  function innerFunction(y) {
    return x + y;          // Accesses outer variable
  }

  return innerFunction;
}

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

Arrow Functions

Theory: ES6 arrow functions have lexical this binding and shorter syntax.

Interview Tip: Explain when NOT to use arrow functions (methods, constructors, event handlers needing dynamic this).

```javascript
// Regular function vs arrow function
const regular = function(a, b) { return a + b; };
const arrow = (a, b) => a + b;

// Arrow function variations
const single = x => x * 2;            // Single parameter, no parentheses
const noParams = () => "Hello";       // No parameters
const multiLine = (x, y) => {         // Multiple statements need braces
  const sum = x + y;
  return sum * 2;
};
```

Arguments Object and Parameters

Theory: The arguments object contains all arguments passed to a function. Rest parameters provide a cleaner alternative.

Interview Tip: Show modern alternatives using rest parameters and default parameters.

```javascript
// Arguments object (older approach)
function oldWay() {
  console.log(arguments[0], arguments[1]); // First two arguments
}

// Rest parameters (modern approach)
function modernWay(...args) {
  console.log(args[0], args[1]);
}

// Default parameters
function greet(name = "World") {
  return `Hello, ${name}!`;
}
console.log(greet());              // "Hello, World!"
```

5. Expressions and Operators

Assignment Operators

Theory: Operators that assign values to variables, including compound assignments.

Interview Tip: Mention the difference between = (assignment) and == or === (comparison).

```javascript
let x = 10;                    // Basic 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 %= 3;                        // x = x % 3 (0)
x **= 2;                       // x = x ** 2 (0)

// Destructuring assignment
const [a, b] = [1, 2];
const {name, age} = {name: "John", age: 30};
```

Comparison Operators

Theory: Operators for comparing values. Strict vs loose equality is crucial.

Interview Tip: Always explain === vs == and type coercion pitfalls.

```javascript
// Strict equality (recommended)
console.log(5 === 5);          // true
console.log(5 === "5");        // false

// Loose equality (avoid)
console.log(5 == "5");         // true (type coercion)
console.log(null == undefined);      // true (special case)

// Other comparisons
console.log(10 > 5);           // true
console.log("a" < "b");        // true (lexicographic)
```

Logical Operators

Theory: AND (&&), OR (||), and NOT (!) operators. They short-circuit and can return non-boolean values.

Interview Tip: Explain short-circuiting and practical uses for default values and conditional execution.

```javascript
// Short-circuiting behavior
const user = { name: "John" };
const name = user.name || "Anonymous";  // Default value
user.isAdmin && console.log("Admin!");  // Conditional execution

// Nullish coalescing (ES2020)
const value = null ?? "default";      // "default"
const zero = 0 ?? "default";          // 0 (only null/undefined)

// Logical assignment (ES2021)
let a = null;
a ??= "default value";              // Assign if nullish
```

Ternary Operator

Theory: Shorthand for if-else statements using condition ? true : false.

Interview Tip: Show when it's helpful vs when it hurts readability.

```javascript
// Basic ternary
const age = 20;
const status = age >= 18 ? "adult" : "minor";

// Nested ternary (use sparingly)
const grade = score >= 90 ? "A" : score >= 80 ? "B" : "C";

// With function calls
const result = isValid() ? processData() : showError();
```

6. Numbers and Strings

Numbers and Math Object

Theory: JavaScript has one number type (IEEE 754 double precision). Math object provides mathematical functions.

Interview Tip: Discuss floating-point precision issues and common Math methods.

```javascript
```

```javascript
// Number quirks
console.log(0.1 + 0.2);              // 0.30000000000000004
console.log(Number.MAX_SAFE_INTEGER);  // 9007199254740991
console.log(parseInt("10px"));        // 10
console.log(parseFloat("3.14"));      // 3.14

// Math object methods
Math.round(4.7);                  // 5
Math.floor(4.7);                  // 4
Math.random();                    // Random number 0-1
Math.max(1, 3, 2);                // 3
```

Strings and Template Literals

Theory: Strings are immutable sequences of characters. Template literals provide string interpolation.

Interview Tip: Explain string methods and the power of template literals for complex strings.

```javascript
// String methods
const str = "JavaScript";
console.log(str.length);             // 10
console.log(str.toUpperCase());       // "JAVASCRIPT"
console.log(str.slice(0, 4));         // "Java"
console.log(str.includes("Script"));   // true

// Template literals
const name = "World";
const greeting = `Hello, ${name}!`;    // "Hello, World!"
const multiline = `Line 1
Line 2`;                            // Preserves line breaks
```

BigInt

Theory: BigInt handles integers larger than Number.MAX_SAFE_INTEGER.

Interview Tip: Explain when to use BigInt and its limitations.

```javascript
```

```javascript
// BigInt creation and usage
const big1 = 123n;              // BigInt literal
const big2 = BigInt(456);         // BigInt constructor
const big3 = BigInt("789");        // From string

console.log(big1 + big2);         // 579n
// console.log(big1 + 123);        // TypeError: can't mix
console.log(big1 + BigInt(123));     // 246n
```

7. Regular Expressions

Creating and Using Regex

Theory: Regular expressions are patterns for matching character combinations in strings.

Interview Tip: Show common patterns and explain when regex is appropriate vs overkill.

```javascript
// Creating regex
const regex1 = /ab+c/;              // Literal notation
const regex2 = new RegExp("ab+c");    // Constructor

// Common patterns
const email = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
const phone = /^\(\d{3}\)\s\d{3}-\d{4}$/;

// Using regex
const text = "The quick brown fox";
console.log(regex1.test("abc"));      // true
console.log(text.match(/\w+/g));      // ["The", "quick", "brown", "fox"]
console.log(text.replace(/fox/, "dog")); // "The quick brown dog"
```

8. Indexed Collections (Arrays)

Array Basics and Methods

Theory: Arrays are ordered collections with numerous built-in methods for manipulation.

Interview Tip: Master array methods like map, filter, reduce. Explain mutating vs non-mutating methods.

```javascript
```

```javascript
// Array creation and basic operations
const arr = [1, 2, 3, 4, 5];
arr.push(6);                // Mutates: [1,2,3,4,5,6]
arr.pop();                  // Mutates: [1,2,3,4,5]

// Non-mutating methods
const doubled = arr.map(x => x * 2);   // [2,4,6,8,10]
const evens = arr.filter(x => x % 2 === 0); // [2,4]
const sum = arr.reduce((acc, x) => acc + x, 0); // 15
```

All Array Methods - Complete Reference

Theory: JavaScript arrays have 30+ methods for manipulation, iteration, and transformation.

Interview Tip: Know which methods mutate vs return new arrays. Group them by functionality.

Mutating Methods (Change Original Array)

javascript

```javascript
const arr = [1, 2, 3];

// Adding/Removing elements
arr.push(4);                // [1,2,3,4] - add to end
arr.pop();                  // [1,2,3] - remove from end
arr.unshift(0);             // [0,1,2,3] - add to start
arr.shift();                // [1,2,3] - remove from start
arr.splice(1, 1, 'new');    // [1,'new',3] - remove/add at index

// Sorting/Reversing
arr.reverse();              // [3,'new',1] - reverse order
arr.sort();                 // [1,3,'new'] - sort elements
```

Non-Mutating Methods (Return New Array/Value)

javascript

```javascript
const nums = [1, 2, 3, 4, 5];
const users = [{name: "Alice", age: 25}, {name: "Bob", age: 30}];

// Transformation methods
nums.map(x => x * 2);                 // [2,4,6,8,10] - transform each
nums.filter(x => x > 2);              // [3,4,5] - keep matching
nums.reduce((sum, x) => sum + x, 0);  // 15 - reduce to single value
nums.flatMap(x => [x, x]);            // [1,1,2,2,3,3,4,4,5,5]

// Iteration methods (return undefined, for side effects)
nums.forEach(x => console.log(x));    // Print each element
```

Search and Test Methods

javascript

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

// Finding elements
arr.find(x => x > 3);                 // 4 - first match
arr.findIndex(x => x > 3);            // 3 - index of first match
arr.findLast(x => x > 3);             // 5 - last match (ES2022)
arr.findLastIndex(x => x > 3);        // 4 - index of last match (ES2022)

// Index methods
arr.indexOf(3);                       // 2 - first index of value
arr.lastIndexOf(3);                   // 2 - last index of value
arr.includes(3);                      // true - contains value

// Testing methods
arr.some(x => x > 4);                 // true - at least one matches
arr.every(x => x > 0);                // true - all match
```

Joining and Slicing

javascript

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

// Extracting portions
arr.slice(1, 3);                    // [2,3] - extract portion
arr.slice(-2);                      // [4,5] - from end

// Converting to strings
arr.join(', ');                     // "1, 2, 3, 4, 5"
arr.toString();                     // "1,2,3,4,5"

// Concatenating
arr.concat([6, 7]);                 // [1,2,3,4,5,6,7]
```

Advanced Array Methods

```javascript
const nested = [[1, 2], [3, [4, 5]]];
const sparse = [1, , , 4];

// Flattening
nested.flat();                      // [1,2,3,[4,5]] - flatten 1 level
nested.flat(2);                     // [1,2,3,4,5] - flatten 2 levels
nested.flat(Infinity);              // [1,2,3,4,5] - flatten all

// Array-like to Array
Array.from("hello");                // ['h','e','l','l','o']
Array.from({length: 3}, (_, i) => i);  // [0,1,2]

// Creating arrays
Array.of(1, 2, 3);                  // [1,2,3] - better than new Array()
new Array(3).fill(0);               // [0,0,0] - fill with value
```

ES6+ Array Methods

```javascript
```

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

// Iterator methods
for (let value of arr.values()) console.log(value);    // Values iterator
for (let index of arr.keys()) console.log(index);      // Keys iterator
for (let [i, v] of arr.entries()) console.log(i, v);   // Entries iterator

// Copying and filling
const copy = arr.copyWithin(0, 3, 4);  // [4,2,3,4,5] - copy within array
arr.fill(0, 1, 3);                     // [1,0,0,4,5] - fill range
```

Method Chaining Examples

```javascript
const users = [
  {name: "Alice", age: 25, active: true, posts: 5},
  {name: "Bob", age: 30, active: false, posts: 3},
  {name: "Charlie", age: 35, active: true, posts: 8},
  {name: "Diana", age: 28, active: true, posts: 2}
];

// Complex chaining
const result = users
  .filter(user => user.active)           // Get active users
  .sort((a, b) => b.posts - a.posts)      // Sort by posts desc
  .slice(0, 2)                           // Top 2
  .map(user => ({                        // Transform to new shape
    name: user.name,
    score: user.posts * user.age
  }))
  .reduce((acc, user) => {               // Group results
    acc[user.name] = user.score;
    return acc;
  }, {});

console.log(result);                     // {Charlie: 280, Alice: 125}
```

Performance Tips for Arrays

```javascript
```

```
// Use for...of for simple iteration (fastest)
for (const item of arr) { /* process item */ }

// Use forEach for functional style
arr.forEach(item => process(item));

// Use map/filter/reduce for transformations
const processed = arr.map(transform).filter(isValid);

// Avoid creating unnecessary arrays in loops
// Bad: arr.filter().map().reduce()
// Better: single reduce when possible
const result = arr.reduce((acc, item) => {
  if (isValid(item)) {
    acc.push(transform(item));
  }
  return acc;
}, []);
```

Multi-dimensional Arrays

Theory: Arrays can contain other arrays, creating matrices or nested structures.

Interview Tip: Show how to access and manipulate nested data.

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

console.log(matrix[1][2]);          // 6
matrix[0][0] = 10;                  // Modify element

// Flattening arrays
const nested = [[1, 2], [3, 4], [5]];
const flat = nested.flat();         // [1, 2, 3, 4, 5]
const deepFlat = [[[1]], [2, 3]].flat(2); // [1, 2, 3]
```

9. Keyed Collections

Maps

Theory: Maps hold key-value pairs where keys can be any type (unlike objects with string keys).

Interview Tip: Explain when to use Map vs Object and iteration differences.

```javascript
// Creating and using Maps
const map = new Map();
map.set("name", "John");
map.set(42, "number key");
map.set(true, "boolean key");

console.log(map.get("name"));       // "John"
console.log(map.size);              // 3
console.log(map.has(42));           // true

// Iterating Maps
for (let [key, value] of map) {
  console.log(key, value);
}
```

Sets

Theory: Sets store unique values of any type. No duplicates allowed.

Interview Tip: Show practical uses like removing duplicates from arrays.

```javascript
// Creating and using Sets
const set = new Set([1, 2, 2, 3, 3, 4]);
console.log(set);                   // Set {1, 2, 3, 4}

set.add(5);
set.delete(1);
console.log(set.has(2));            // true

// Remove duplicates from array
const arr = [1, 2, 2, 3, 3, 4];
const unique = [...new Set(arr)];     // [1, 2, 3, 4]
```

10. Working with Objects

Object Creation and Properties

Theory: Objects are collections of key-value pairs. Multiple ways to create and manipulate them.

Interview Tip: Explain property descriptors, Object.create(), and modern object features.

```javascript
// Object creation methods
const obj1 = { name: "John", age: 30 };        // Literal
const obj2 = new Object();                     // Constructor
const obj3 = Object.create(null);              // No prototype

// Property manipulation
obj1.city = "New York";                       // Add property
delete obj1.age;                              // Remove property
console.log("name" in obj1);                  // true
console.log(Object.keys(obj1));               // ["name", "city"]
```

Getters and Setters

Theory: Accessor properties that allow computed values and validation.

Interview Tip: Show practical uses for data validation and computed properties.

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

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

  set fullName(value) {
    [this.firstName, this.lastName] = value.split(" ");
  }
};

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

Object Methods and this

Theory: Methods are functions stored as object properties. 'this' refers to the object.

Interview Tip: Explain this binding in different contexts and arrow function behavior.

```javascript
const calculator = {
  value: 0,

  add(n) {
    this.value += n;
    return this;                  // Method chaining
  },

  multiply(n) {
    this.value *= n;
    return this;
  }
};

calculator.add(5).multiply(2);           // Chaining: value = 10
```

11. Classes

Class Declaration and Constructor

Theory: ES6 classes provide a cleaner syntax for creating objects and implementing inheritance.

Interview Tip: Explain that classes are syntactic sugar over prototypes, not a new concept.

```javascript
```

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

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

  static species() {
    return "Homo sapiens";
  }
}

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

Inheritance with Extends

Theory: Classes can inherit from other classes using extends and super keywords.

Interview Tip: Show method overriding and super usage.

javascript

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

  greet() {          // Override parent method
    return `${super.greet()}, I'm in grade ${this.grade}`;
  }
}

const alice = new Student("Alice", 16, 10);
console.log(alice.greet());          // "Hello, I'm Alice, I'm in grade 10"
```

Private Fields

Theory: ES2022 private fields use # prefix and are truly private.

Interview Tip: Compare with older naming conventions and explain true privacy.

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

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

  getBalance() {
    return this.#balance;
  }

  #validateAmount(amount) {        // Private method
    return amount > 0;
  }
}

const account = new BankAccount(1000);
console.log(account.getBalance());          // 1000
// console.log(account.#balance);           // SyntaxError
```

12. Asynchronous JavaScript

Promises

Theory: Promises represent eventual completion/failure of asynchronous operations.

Interview Tip: Explain Promise states and chaining vs async/await syntax.

```javascript
```

```javascript
// Creating a Promise
const fetchData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const success = Math.random() > 0.5;
      success ? resolve("Data loaded") : reject("Error occurred");
    }, 1000);
  });
};

// Using Promises
fetchData()
  .then(data => console.log(data))
  .catch(error => console.error(error))
  .finally(() => console.log("Cleanup"));
```

Async/Await

Theory: Async/await provides a synchronous-looking way to work with Promises.

Interview Tip: Show error handling with try/catch and parallel execution.

javascript

```javascript
// Async function
async function loadUserData(id) {
  try {
    const user = await fetchUser(id);
    const posts = await fetchPosts(id);
    return { user, posts };
  } catch (error) {
    console.error("Failed to load data:", error);
    return null;
  }
}

// Parallel execution
async function loadMultipleUsers() {
  const [user1, user2] = await Promise.all([
    fetchUser(1),
    fetchUser(2)
  ]);
  return [user1, user2];
}
```

Promise Utilities

Theory: Promise.all, Promise.race, Promise.allSettled for handling multiple promises.

Interview Tip: Explain when to use each utility and their behavior on rejection.

javascript

```javascript
// Promise utilities
const promises = [
  Promise.resolve("First"),
  Promise.resolve("Second"),
  Promise.reject("Third failed")
];

// All must succeed
Promise.all(promises)
  .then(results => console.log(results))
  .catch(error => console.log("One failed:", error));

// Wait for all to settle
Promise.allSettled(promises)
  .then(results => {
    results.forEach(result => console.log(result.status));
  });
```

13. Advanced Topics

Closures Deep Dive

Theory: Closures are functions that have access to outer function's variables even after outer function returns.

Interview Tip: This is a very common interview question. Show practical examples.

```javascript
// Module pattern using closures
const counterModule = (function() {
  let count = 0;                    // Private variable

  return {
    increment() { return ++count; },
    decrement() { return --count; },
    getCount() { return count; }
  };
})();

console.log(counterModule.getCount());      // 0
console.log(counterModule.increment());     // 1
// count is not accessible directly
```

Prototypes and Inheritance

Theory: JavaScript uses prototypal inheritance. Every object has a prototype.

Interview Tip: Explain prototype chain and how class inheritance works under the hood.

javascript

```javascript
// Prototype-based inheritance
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  return `${this.name} makes a sound`;
};

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

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

Dog.prototype.speak = function() {
  return `${this.name} barks`;
};
```

Event Loop and Asynchronous Execution

Theory: JavaScript's concurrency model is based on an event loop.

Interview Tip: Explain call stack, task queue, and microtask queue.

javascript

```javascript
console.log("1");                          // Synchronous

setTimeout(() => console.log("2"), 0);        // Macrotask

Promise.resolve().then(() => console.log("3")); // Microtask

console.log("4");                          // Synchronous

// Output: 1, 4, 3, 2
// Microtasks run before macrotasks
```

Memory Management

Theory: JavaScript has automatic garbage collection, but memory leaks can still occur.

Interview Tip: Show common leak patterns and how to avoid them.

```javascript
// Common memory leak patterns
let theThing = null;
let replaceThing = function () {
  let leak = theThing;                    // Keeps reference
  theThing = {
    longStr: new Array(1000000).join('*'),
    someMethod() {
      return leak;                    // Closure keeps leak alive
    }
  };
};

// Better: break the reference
let betterReplace = function() {
  let previousThing = theThing;
  theThing = { /* new object */ };
  previousThing = null;                   // Break reference
};
```

Modules (ES6+)

Theory: Modules provide a way to organize code into separate files with explicit imports/exports.

Interview Tip: Explain module scope and different export/import syntaxes.

```javascript
// math.js - Named exports
export function add(a, b) { return a + b; }
export const PI = 3.14159;

// calculator.js - Default export
export default class Calculator {
  add(a, b) { return a + b; }
}

// main.js - Importing
import Calculator from './calculator.js';     // Default import
import { add, PI } from './math.js';          // Named imports
import * as Math from './math.js';            // Namespace import
```

Destructuring and Spread

Theory: Destructuring extracts values from arrays/objects. Spread operator expands iterables.

Interview Tip: Show advanced patterns and practical uses.

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

// Object destructuring with renaming and defaults
const {name: userName = "Anonymous", age} = {name: "John", age: 30};

// Spread operator
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5];              // [1, 2, 3, 4, 5]
const obj1 = {a: 1, b: 2};
const obj2 = {...obj1, c: 3};              // {a: 1, b: 2, c: 3}
```

Common Interview Questions

1. What is hoisting?

Variables and function declarations are moved to the top of their scope during compilation.

2. Explain event bubbling and capturing

Events propagate through the DOM tree. Capturing goes down, bubbling goes up.

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

== performs type coercion, === checks strict equality without coercion.

4. How does 'this' work?

'this' depends on how a function is called: method call, function call, constructor call, or explicit binding.

5. What is a callback function?

A function passed as an argument to another function, executed later.

6. Explain Promise vs async/await

Both handle asynchronous operations, but async/await provides cleaner, more readable syntax.

7. What is event delegation?

Using event bubbling to handle events on parent elements instead of individual child elements.

8. How do you handle errors in async operations?

Use .catch() with Promises or try/catch with async/await.

9. What is the difference between let, const, and var?

Scope (function vs block), hoisting behavior, and reassignment rules differ.

10. Explain the module pattern

Uses closures to create private variables and expose only necessary functionality.

Performance Tips

1. **Use const/let instead of var** for better scoping
2. **Prefer array methods** like map/filter over for loops for readability
3. **Use async/await** for cleaner asynchronous code
4. **Implement proper error handling** with try/catch
5. **Avoid global variables** to prevent naming conflicts
6. **Use strict mode** to catch common mistakes

7. **Minimize DOM manipulation** and batch updates
8. **Use event delegation** for better performance with many elements
9. **Implement debouncing** for frequent events like scroll/resize
10. **Profile your code** to identify actual bottlenecks

This guide covers the essential JavaScript concepts you'll need for interviews. Practice implementing these concepts and explaining them clearly!