

JavaScript Interview Coding Problems with Solutions & Explanations

Table of Contents

- [Part 1: Problems 1-10](#)
- [Part 2: Problems 11-20](#)
- [Part 3: Problems 21-30](#)
- [Part 4: Problems 31-40](#)
- [Part 5: Problems 41-50](#)
- [Part 6: Problems 51-60](#)
- [Part 7: Problems 61-70](#)
- [Part 8: Problems 71-80](#)
- [Part 9: Problems 81-90](#)
- [Part 10: Problems 91-100](#)

Part 1: Problems 1-10

Problem 1: Variable Hoisting

Interview Frequency:  High (Common in junior interviews)

Question: What is the output of the following code and why?

```
var a = 1;
function print() {
  console.log(a);
  var a = 2;
}
print();
```

Solution:

```
// Due to hoisting, the variable `a` inside print is undefined at the time of console.log
function print() {
  var a; // hoisted declaration (but not initialization)
  console.log(a); // undefined
  a = 2;
}

print(); // undefined
```

Explanation:

- `var a = 2;` is hoisted as `var a;` to the top of `print()`.
- So `console.log(a)` references the local `a`, which is undefined at that point.
- Demonstrates hoisting and variable shadowing.

Problem 2: Closure Counter

Interview Frequency: 🔴 High (Common in all levels)

Question: Create a function that returns a counter using closures.

Solution:

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

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

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

Explanation:

- `count` is preserved via closure, so it's not reset after each call.
- Demonstrates use of closures for encapsulation and state persistence.

Problem 3: Typeof Null

Interview Frequency: 🟡 Medium (Common trivia question)

Question: What will this log and why?

```
console.log(typeof null);
```

Solution:

```
// typeof null returns 'object'  
console.log(typeof null); // 'object'
```

Explanation:

- This is a historical bug in JavaScript.
- `null` is a primitive, but `typeof null === 'object'` due to legacy reasons.
- Important trivia for type checking questions.

Problem 4: Currying Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make this `add(2)(3)(4)` return the sum (9)

Solution:

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

```
console.log(add(2)(3)(4)); // 9
```

Explanation:

- This is currying – breaking a function into a chain of unary functions.
- Helps with function reuse and partial application.

Problem 5: Closure in Loop

Interview Frequency:  High (Common in all levels)

Question: What's the output and why?

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

Solution:

```
// All logs will print 3  
for (var i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 1000); // prints 3 three times  
}
```

Explanation:

- `var` is function-scoped. The loop completes and `i === 3` when `setTimeout` runs.
- Fix using `let`, which is block-scoped:

```
for (let i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 1000); // 0, 1, 2  
}
```

Problem 6: String Reversal

Interview Frequency:  Very High (Common in all levels)

Question: Implement a function that reverses a string.

Solution:

```
function reverse(str) {
  let result = '';
  for (let i = str.length - 1; i >= 0; i--) {
    result += str[i];
  }
  return result;
}
```

```
console.log(reverse("hello")); // "olleh"
```

Explanation:

- Simple loop logic demonstrates string manipulation.
- Avoids using `.reverse()` to show understanding of character access.

Problem 7: Palindrome Check

Interview Frequency:  Very High (Common in all levels)

Question: Implement a function to check if a string is a palindrome.

Solution:


```
function isPalindrome(str) {
  const reversed = str.split('').reverse().join('');
  return str === reversed;
}
```

```
console.log(isPalindrome("madam")); // true
console.log(isPalindrome("hello")); // false
```

Explanation:

- Uses string reversal to check equality.
- You can also use pointers for a more optimized version.

Problem 8: Array Flattening

Interview Frequency:  Medium (Common in mid-level interviews)

Question: Write a function to flatten a nested array.

Solution:

```
function flatten(arr) {  
  return arr.reduce((acc, val) => {  
    return acc.concat(Array.isArray(val) ? flatten(val) : val);  
  }, []);  
}  
  
console.log(flatten([1, [2, [3, 4], 5], 6])); // [1, 2, 3, 4, 5, 6]
```

Explanation:

- Demonstrates recursion, array flattening, and reduce.

Problem 9: Type Coercion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What will this return and why?

```
console.log([] + []);  
console.log([] + {});  
console.log({} + []);
```

Solution:

```
console.log([] + []);    // '' (empty string)  
console.log([] + {});    // '[object Object]'  
console.log({} + []);    // 0 or '[object Object]' depending on context
```

Explanation:

- `[] + []` → `''` (coerced to string)
- `[] + {}` → `'' + '[object Object]'` → `'[object Object]'`
- `{}` on first line is interpreted as block, not object

Problem 10: Map Polyfill

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a custom implementation of `Array.prototype.map`.

Solution:

```
Array.prototype.myMap = function (callback) {  
  let result = [];  
  
  for (let i = 0; i < this.length; i++) {  
    result.push(callback(this[i], i, this));  
  }  
  
  return result;  
};  
  
const arr = [1, 2, 3];  
console.log(arr.myMap(x => x * 2)); // [2, 4, 6]
```

Explanation:

- Mimics built-in `map()` using loop and callback.
- Must support arguments: value, index, original array.

Part 2: Problems 11-20

Problem 11: String Number Coercion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What does the following return and why?

```
console.log(1 + '2' + '2');  
console.log(1 + +'2' + '2');  
console.log(1 + -'1' + '2');  
console.log(+'1' + '1' + '2');  
console.log('A' - 'B' + '2');  
console.log('A' - 'B' + 2);
```

Solution:

```
console.log(1 + '2' + '2');    // '122'
console.log(1 + +'2' + '2');   // '32'
console.log(1 + -'1' + '2');   // '02'
console.log(+'1' + '1' + '2'); // '112'
console.log('A' - 'B' + '2');  // 'NaN2'
console.log('A' - 'B' + 2);    // NaN
```

Explanation:

- `+` operator can mean string concatenation or numeric addition.
- Unary `+` converts strings to numbers.
- `'A' - 'B'` is `NaN`, and concatenation with string results in `'NaN2'`.

Problem 12: Array Equality

Interview Frequency:  High (Common in junior interviews)

Question: What is the output and why?

```
let a = [1, 2, 3];
let b = [1, 2, 3];
console.log(a == b);
console.log(a === b);
```


Solution:

```
console.log(a == b); // false
console.log(a === b); // false
```

Explanation:

- Arrays are reference types. `==` and `===` both compare references, not contents.
- `a` and `b` point to different arrays.

Problem 13: Array Reference

Interview Frequency:  High (Common in junior interviews)

Question: What's the output and explain why:


```
let a = [1, 2, 3];
let b = a;
a.push(4);
console.log(b);
```

Solution:

```
console.log(b); // [1, 2, 3, 4]
```

Explanation:

- `b` references the same array as `a`, so mutations to `a` reflect in `b`.

Problem 14: Deep Clone

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to deeply clone an object.

Solution:

```
function deepClone(obj) {
  if (obj === null || typeof obj !== 'object') return obj;

  if (Array.isArray(obj)) {
    return obj.map(deepClone);
  }

  const result = {};
  for (let key in obj) {
    result[key] = deepClone(obj[key]);
  }
  return result;
}

// Test
const original = { a: 1, b: { c: 2 } };
const cloned = deepClone(original);
cloned.b.c = 100;
console.log(original.b.c); // Still 2
```

Explanation:

- Demonstrates recursion, object handling, and immutability.

Problem 15: Function Expression

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What will this print and why?

```
let foo = function bar() {  
  return 42;  
};
```

```
console.log(typeof bar);
```

Solution:

```
console.log(typeof bar); // 'undefined'
```

Explanation:

- `bar` is a named function expression.
- The name `bar` is only visible inside the function.

Problem 16: Object Type Check

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function that returns true if a value is an object (not null or array).

Solution:

```
function isObject(val) {  
  return typeof val === 'object' && val !== null && !Array.isArray(val);  
}
```

```
console.log(isObject({}));      // true  
console.log(isObject([]));      // false  
console.log(isObject(null));    // false  
console.log(isObject('string')); // false
```

Explanation:

- Clarifies common JavaScript edge cases: `typeof null === 'object'` , arrays are also objects.

Problem 17: Arguments to Array

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How do you convert arguments into an array?

Solution:

```
function argsToArray() {  
    return Array.prototype.slice.call(arguments);  
}
```

```
// ES6 way  
function argsToArrayES6(...args) {  
    return args;  
}
```

Explanation:

- Shows both traditional and modern ways to handle function arguments.
- Demonstrates use of rest parameters in ES6.

Problem 18: Remove Duplicates

Interview Frequency: 🟢 Very High (Common in all levels)

Question: Write a function to remove duplicates from an array.

Solution:

```
function removeDuplicates(arr) {  
  return [...new Set(arr)];  
}  
  
// Alternative without Set  
function removeDuplicatesAlt(arr) {  
  return arr.filter((item, index) => arr.indexOf(item) === index);  
}  
  
console.log(removeDuplicates([1, 2, 2, 3, 3, 4])); // [1, 2, 3, 4]
```

Explanation:

- Shows modern use of `Set` for deduplication.
- Alternative solution demonstrates array methods.

Problem 19: Factorial Recursion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a recursive function to calculate factorial.

Solution:

```
function factorial(n) {  
  if (n <= 1) return 1;  
  return n * factorial(n - 1);  
}  
  
console.log(factorial(5)); // 120
```

Explanation:

- Demonstrates recursive function implementation.
- Shows base case handling.

Problem 20: Floating Point Precision

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the output of `0.1 + 0.2` and why?

Solution:

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

Explanation:

- JavaScript uses IEEE 754 floating-point arithmetic.
- Some decimal numbers cannot be represented exactly in binary.
- Use `toFixed()` or multiply by 100 and divide by 100 for precise calculations.

Part 3: Problems 21-30

Problem 21: Factorial Recursion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function that returns the factorial of a number using recursion.

Solution:

```
function factorial(n) {  
  if (n === 0) return 1;      // Base case  
  return n * factorial(n - 1); // Recursive call  
}
```

```
console.log(factorial(5)); // 120
```

Explanation:

- Demonstrates recursion and base case termination.
- Common interview topic to test logical and recursive thinking.

Problem 22: Floating Point Precision

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the output and why?

```
console.log(0.1 + 0.2 === 0.3);
```

Solution:

```
console.log(0.1 + 0.2 === 0.3); // false
```

Explanation:

- Due to floating-point precision issues, $0.1 + 0.2 = 0.30000000000000004$.
- Very common gotcha in JavaScript.

Problem 23: Sparse Arrays

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What's the output?

```
let a = [1, 2, 3];  
a[10] = 100;  
console.log(a.length);
```

Solution:

```
console.log(a.length); // 11
```

Explanation:

- Arrays in JS are sparse — setting index 10 sets `.length = 11`.
- Indices [3] to [9] are empty (undefined).

Problem 24: Custom Filter Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a custom implementation of `Array.prototype.filter`.

Solution:

```
Array.prototype.myFilter = function (callback) {  
  let result = [];  
  
  for (let i = 0; i < this.length; i++) {  
    if (callback(this[i], i, this)) {  
      result.push(this[i]);  
    }  
  }  
  
  return result;  
};  
  
// Test  
const nums = [1, 2, 3, 4];  
console.log(nums.myFilter(n => n % 2 === 0)); // [2, 4]
```

Explanation:

- Tests ability to mimic built-in higher-order methods using loops and callbacks.

Problem 25: Array Map

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What will this return?

```
let x = [1, 2, 3];  
let y = x.map((_, i) => x[i] + 1);  
console.log(y);
```

Solution:

```
console.log(y); // [2, 3, 4]
```

Explanation:

- `map()` applies the callback to each element.
- Adds 1 to each number and returns a new array.

Problem 26: Promise Delay

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function that delays execution using a Promise.

Solution:

```
function wait(ms) {  
  return new Promise(resolve => setTimeout(resolve, ms));  
}  
  
wait(1000).then(() => console.log('Waited 1 second'));
```

Explanation:

- Shows understanding of Promises and setTimeout.
- Often used to test async behavior and wrapping legacy APIs.

Problem 27: String Number Coercion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What will this return and why?

```
console.log('5' - 1);  
console.log('5' + 1);
```

Solution:

```
console.log('5' - 1); // 4 → coercion to number  
console.log('5' + 1); // '51' → string concatenation
```

Explanation:

- - triggers numeric coercion.
- + with a string performs concatenation instead of arithmetic.

Problem 28: Variable Hoisting

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What will this print and why?

```
let x = 10;

(function () {
  console.log(x);
  var x = 20;
})();
```

Solution:

```
console.log(x); // undefined
```

Explanation:

- var x is hoisted, so function scope has its own x, initialized as undefined at the top.
- The local x is undefined before initialization.

Problem 29: Function Context Methods

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What's the difference between call, apply, and bind?

Solution:

```
function greet(name) {
  console.log(`${this.greeting}, ${name}`);
}

const context = { greeting: 'Hello' };

greet.call(context, 'John'); // Hello, John
greet.apply(context, ['Jane']); // Hello, Jane

const bound = greet.bind(context);
bound('Mike'); // Hello, Mike
```

Explanation:

- call: invokes function immediately with comma-separated args

- apply: same but args are an array
- bind: returns a new function with this fixed — doesn't call it

Problem 30: This Context in setTimeout

Interview Frequency:  High (Common in all levels)

Question: What will this output and why?

```
let obj = {  
  a: 10,  
  fn: function () {  
    setTimeout(function () {  
      console.log(this.a);  
    }, 1000);  
  }  
};  
  
obj.fn();
```

Solution:

```
// undefined because 'this' refers to global object, not obj
```

Explanation:

- Inside setTimeout, this does not refer to obj.
- To fix it:

```
setTimeout(() => console.log(this.a), 1000); // uses arrow function
```

- Arrow functions capture this from their surrounding lexical scope.

Part 4: Problems 31-40

Problem 31: Palindrome Check

Interview Frequency:  Very High (Common in all levels)

Question: How would you check if a string is a palindrome?

Solution:

```
function isPalindrome(str) {  
  const cleaned = str.toLowerCase().replace(/[^a-z0-9]/g, '');  
  return cleaned === cleaned.split('').reverse().join('');  
}  
  
console.log(isPalindrome('madam')); // true  
console.log(isPalindrome('hello')); // false
```

Explanation:

- split, reverse, and join are used to reverse the string.
- Checks for string manipulation and understanding of array methods.

Problem 32: Curry Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a curry function that transforms sum(1)(2)(3) into 6.

Solution:

```
function sum(a) {  
  return function(b) {  
    return function(c) {  
      return a + b + c;  
    }  
  }  
}  
  
console.log(sum(1)(2)(3)); // 6
```

Explanation:

- Uses closures to return nested functions.
- Each function retains access to the previous argument.

Problem 33: Typeof Null

Interview Frequency: 🟡 Medium (Common trivia question)

Question: What does this log and why?

```
console.log(typeof null);
```

Solution:

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

Explanation:

- JavaScript quirk: typeof null incorrectly returns "object".
- Known issue since the early days of JS — often used as a trick question.

Problem 34: Array Flattening

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function that flattens a nested array.

Solution:

```
function flatten(arr) {  
  return arr.reduce((acc, val) =>  
    Array.isArray(val) ? acc.concat(flatten(val)) : acc.concat(val), []);  
}
```

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

Explanation:

- Shows use of recursion, array flattening, and type checking.
- Asked to assess control flow and data structure manipulation.

Problem 35: Equality Operators

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What's the difference between == and ===?

Solution:

```
console.log(5 == '5'); // true (type coercion)
console.log(5 === '5'); // false (strict equality)
```

Explanation:

- == performs type coercion, === checks for type + value.
- Use === in modern JavaScript to avoid unpredictable behavior.

Problem 36: Debounce Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to debounce another function.

Solution:

```
function debounce(fn, delay) {
  let timer;
  return function(...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

// Usage
const log = debounce(() => console.log('Debounced!'), 300);
log(); log(); log(); // Only one log after 300ms
```

Explanation:

- Debouncing delays function calls — useful for input, resize, scroll events.
- Combines closures, setTimeout, and apply.

Problem 37: Sparse Arrays

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What will this return?

```
const a = [];  
a[100] = 42;  
console.log(a.length);
```

Solution:

```
console.log(a.length); // 101
```

Explanation:

- Arrays in JS are not dense; assigning to index 100 makes .length = 101.

Problem 38: Deep Object Equality

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to check deep equality of two objects.

Solution:

```
function deepEqual(a, b) {  
  if (a === b) return true;  
  
  if (typeof a !== 'object' || typeof b !== 'object' || a === null || b === null)  
    return false;  
  
  const keysA = Object.keys(a);  
  const keysB = Object.keys(b);  
  
  if (keysA.length !== keysB.length) return false;  
  
  return keysA.every(key => deepEqual(a[key], b[key]));  
}  
  
// Test  
console.log(deepEqual({ a: 1, b: { c: 2 } }, { a: 1, b: { c: 2 } })); // true
```

Explanation:

- Involves recursion, object traversal, and comparisons.
- Common problem to assess deep knowledge of JS objects.

Problem 39: Map Polyfill

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a polyfill for `Array.prototype.map`.

Solution:

```
Array.prototype.myMap = function(callback) {  
  const result = [];  
  
  for (let i = 0; i < this.length; i++) {  
    result.push(callback(this[i], i, this));  
  }  
  
  return result;  
};  
  
// Test  
const nums = [1, 2, 3];  
console.log(nums.myMap(x => x * 2)); // [2, 4, 6]
```

Explanation:

- Interviewers look for knowledge of higher-order functions and this.

Problem 40: Closure Loop

Interview Frequency: 🔴 High (Common in all levels)

Question: What's the output of this closure loop problem?

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

Solution:

```
// Output: 3, 3, 3
```

Explanation:

- var is function-scoped. All timeouts refer to the same i.
- Fix with let:

```
for (let i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 1000); // 0, 1, 2  
}
```

- Demonstrates closures inside loops and block scoping

Part 5: Problems 41-50

Problem 41: Variable Hoisting

Interview Frequency:  High (Common in junior interviews)

Question: What is hoisting in JavaScript? Predict the output and explain hoisting behavior:

```
console.log(a);  
var a = 5;
```


Solution:

```
// Output: undefined
```

Explanation:

- var a is hoisted to the top of the scope with an initial value of undefined.
- Only declarations are hoisted, not initializations.

Problem 42: Once Function

Interview Frequency:  Medium (Common in mid-level interviews)

Question: Create a once() function that runs a function only once.

Solution:


```
function once(fn) {
  let called = false;
  let result;

  return function (...args) {
    if (!called) {
      result = fn.apply(this, args);
      called = true;
    }
    return result;
  };
}

// Test
const init = once(() => console.log('Initialized'));
init(); // Initialized
init(); // (nothing)
```

Explanation:

- Uses closures to persist state between calls.

Problem 43: Undefined vs Not Defined

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What's the difference between undefined and not defined?

Solution:

```
let x;
console.log(x);           // undefined
console.log(y);           // ReferenceError: y is not defined
```

Explanation:

- undefined: variable is declared but not assigned.
- not defined: variable was never declared.

Problem 44: Temporal Dead Zone

Interview Frequency:  High (Common in junior interviews)

Question: What will this log and why?

```
let a = 1;
function foo() {
  console.log(a);
  let a = 2;
}
foo();
```


Solution:

```
// ReferenceError: Cannot access 'a' before initialization
```

Explanation:

- let variables are hoisted but stay in the Temporal Dead Zone until initialized.

Problem 45: Throttle Function

Interview Frequency:  Medium (Common in mid-level interviews)

Question: Implement a throttle function.

Solution:

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

  return function (...args) {
    if (!inThrottle) {
      fn.apply(this, args);
      inThrottle = true;

      setTimeout(() => {
        inThrottle = false;
      }, limit);
    }
  };
}

// Test
const log = throttle(() => console.log('Called!'), 2000);
window.addEventListener('scroll', log);
```

Explanation:

- Controls rate of execution, e.g. during scroll events.
- Uses closures and timing functions to control execution.
- Shows how to limit function call frequency.

Problem 46: This in Arrow Functions

Interview Frequency:  High (Common in all levels)

Question: What does this refer to in arrow functions?

Solution:

```
const obj = {
  val: 42,
  fn: () => {
    setTimeout(() => {
      console.log(this.val); // 42
    }, 1000);
  }
};

obj.fn();
```

Explanation:

- Arrow functions do not have their own this.
- They use the this value from the surrounding lexical context.

Problem 47: Remove Duplicates

Interview Frequency: 🟢 Very High (Common in all levels)

Question: Write a function that removes duplicates from an array.

Solution:

```
function removeDuplicates(arr) {
  return [...new Set(arr)];
}

console.log(removeDuplicates([1, 2, 2, 3, 1])); // [1, 2, 3]
```

Explanation:

- Set only stores unique values.
- Spread operator creates a new array from the Set.

Problem 48: Shallow vs Deep Copy

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What's the difference between shallow and deep copy?

Solution:

```
let a = { x: 1, y: { z: 2 } };
let b = { ...a };

b.y.z = 100;
console.log(a.y.z); // 100 (shallow copy)
```

Explanation:

- ...a copies top-level properties.
- Nested objects are still referenced, not cloned.

Problem 49: Event Loop

Interview Frequency: 🔴 High (Common in all levels)

Question: Explain Event Loop with this code:

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

setTimeout(() => {
  console.log('Timeout');
}, 0);

Promise.resolve().then(() => console.log('Promise'));

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

Solution:

```
// Output:
// Start
// End
// Promise
// Timeout
```

Explanation:

- JS executes synchronously first.
- Promises (microtask queue) are resolved before setTimeout (macrotask queue).

Problem 50: Closure Example

Interview Frequency:  High (Common in all levels)

Question: What is a closure? Provide a practical use case.

Solution:


```
function makeCounter() {  
  let count = 0;  
  
  return function () {  
    return ++count;  
  };  
}  
  
const counter = makeCounter();  
console.log(counter()); // 1  
console.log(counter()); // 2
```

Explanation:

- A closure preserves access to outer variables.
- Commonly used in data encapsulation or memoization.

Part 6: Problems 51-60

Problem 51: Memoization Function

Interview Frequency:  Medium (Common in mid-level interviews)

Question: Implement a simple memoization function that caches the results of a function to avoid redundant calculations.

Solution:

```
function memoize(fn) {
  const cache = {};

  return function (...args) {
    const key = args.join(',');
    if (key in cache) {
      return cache[key];
    }
    const result = fn.apply(this, args);
    cache[key] = result;
    return result;
  };
}

// Example usage
function add(a, b) {
  console.log('Calculating...');
  return a + b;
}

const memoizedAdd = memoize(add);
console.log(memoizedAdd(1, 2)); // Calculating... 3
console.log(memoizedAdd(1, 2)); // 3 (cached)
```

Explanation:

- Utilizes closures to store cache
- Demonstrates performance optimization and function enhancement
- Shows how to avoid redundant calculations

Problem 52: Context in setTimeout

Interview Frequency:  High (Common in all levels)

Question: What is the output and why?

```
let foo = {
  bar: "baz",
  log: function () {
    console.log(this.bar);
  },
};

setTimeout(foo.log, 1000);
```

Solution:

```
// Output: undefined
```

Explanation:

- `this` is lost when `foo.log` is passed directly to `setTimeout`
- To fix, use `.bind()` :

```
setTimeout(foo.log.bind(foo), 1000); // Output: "baz"
```

- Demonstrates context binding in callbacks

Problem 53: Title Case Conversion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert a string to title case.

Solution:

```
function toTitleCase(str) {
  return str
    .toLowerCase()
    .split(' ')
    .map(word => word[0].toUpperCase() + word.slice(1))
    .join(' ');
}

console.log(toTitleCase("hello world")); // Hello World
```


Explanation:

- Uses split, map, and string indexing
- Tests array manipulation and string transformation
- Shows chaining of array methods

Problem 54: Variable Hoisting

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What's the output of this variable hoisting code?

```
function test() {  
  console.log(a);  
  var a = 10;  
}  
test();
```

Solution:

```
// Output: undefined
```

Explanation:

- `var a` is hoisted to the top of `test()` with `undefined` as initial value
- Demonstrates hoisting behavior with `var` declarations

Problem 55: Array to Tree Structure

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert a flat array into a tree structure.

Solution:

```
function arrayToTree(items, parentId = null) {
  return items
    .filter(item => item.parentId === parentId)
    .map(item => ({
      ...item,
      children: arrayToTree(items, item.id)
    }));
}

// Example
const items = [
  { id: 1, parentId: null },
  { id: 2, parentId: 1 },
  { id: 3, parentId: 1 }
];

console.log(JSON.stringify(arrayToTree(items), null, 2));
```

Explanation:

- Involves object mapping, nested structures, and reference linking
- Shows how to transform flat data into hierarchical structures
- Demonstrates efficient lookup using a hash map

Problem 56: Custom Bind Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a custom bind function.

Solution:

```

Function.prototype.myBind = function (context, ...args) {
  const fn = this;
  return function (...rest) {
    return fn.apply(context, [...args, ...rest]);
  };
};

// Test
function greet(greeting, name) {
  return `${greeting}, ${name}`;
}

const greetHello = greet.myBind(null, 'Hello');
console.log(greetHello('Alice')); // Hello, Alice

```

Explanation:

- Demonstrates binding context using apply
- Mimics the behavior of the native Function.prototype.bind
- Shows how to implement function methods

Problem 57: Deep Cloning

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to clone an object deeply?

Solution:

```

function deepClone(obj) {
  return JSON.parse(JSON.stringify(obj));
}

const original = { a: 1, b: { c: 2 } };
const clone = deepClone(original);
clone.b.c = 100;

console.log(original.b.c); // 2

```

Explanation:

- Simple deep clone via serialization
- Limitations: skips functions, undefined, Date, and circular references
- Shows a quick way to create deep copies of simple objects

Problem 58: Call, Apply, and Bind

Interview Frequency:  High (Common in all levels)

Question: Explain call, apply, and bind with examples.

Solution:

```
function intro(lang1, lang2) {  
  console.log(`${this.name} knows ${lang1} and ${lang2}`);  
}  
  
const dev = { name: 'John' };  
  
intro.call(dev, 'JavaScript', 'Python'); // John knows JavaScript and Python  
intro.apply(dev, ['JavaScript', 'Python']); // Same  
const boundIntro = intro.bind(dev, 'JavaScript');  
boundIntro('Python'); // John knows JavaScript and Python
```

Explanation:

- call: invokes function immediately with comma-separated args
- apply: same as call but args are an array
- bind: returns a new function with bound context
- Shows different ways to control function context

Problem 59: Closure in Loop

Interview Frequency:  High (Common in all levels)

Question: What's the output and why?

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

Solution:

```
// Output: 3, 3, 3
```

Explanation:

- var is function-scoped, so all timeouts refer to the same i = 3
- Fix with let:

```
for (let i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 1000); // 0, 1, 2  
}
```

- Demonstrates closures inside loops and block scoping

Problem 60: Pure Functions

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What's a pure function? Provide an example.

Solution:

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

Explanation:

- Pure functions have:
 - No side effects
 - Same output for same input
- Great for predictability, testing, and immutability
- Demonstrates functional programming principles

Part 7: Problems 61-70

Problem 61: Temporal Dead Zone

Interview Frequency: 🚫 High (Common in junior interviews)

Question: What is the output and why?

```
let x = 10;

function test() {
  console.log(x);
  let x = 20;
}

test();
```

Solution:

```
// Output: ReferenceError: Cannot access 'x' before initialization
```

Explanation:

- `let x = 20` is hoisted but in the Temporal Dead Zone
- Accessing `x` before its declaration throws an error
- Demonstrates TDZ behavior with `let` declarations

Problem 62: Array Type Detection

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to detect if a variable is an array?

Solution:

```
function isArray(val) {  
  return Array.isArray(val);  
}  
  
console.log(isArray([1, 2, 3])); // true  
console.log(isArray("123")); // false
```

Explanation:

- `Array.isArray()` is the safest way to check for arrays
- Avoid `typeof`, which returns "object" for arrays
- Shows proper type checking in JavaScript

Problem 63: Custom Map Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a custom map function.

Solution:

```
Array.prototype.myMap = function (callback) {  
  const result = [];  
  for (let i = 0; i < this.length; i++) {  
    result.push(callback(this[i], i, this));  
  }  
  return result;  
};  
  
// Test  
const arr = [1, 2, 3];  
console.log(arr.myMap(x => x * 2)); // [2, 4, 6]
```

Explanation:

- Mimics native `map()`
- Demonstrates prototype extension and callback usage
- Shows how to implement array methods

Problem 64: Equality Operators

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What's the difference between `==` and `===`?

Solution:

```
console.log(5 == '5');    // true (type coercion)
console.log(5 === '5');   // false (strict equality)
```

Explanation:

- `==` performs type coercion, `===` checks for type + value.
- Use `===` in modern JavaScript to avoid unpredictable behavior.

Problem 65: Debounce Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a debounce function.

Solution:

```
function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

// Test
const onResize = debounce(() => {
  console.log('Resize event fired!');
}, 500);

window.addEventListener('resize', onResize);
```

Explanation:

- Delays function execution until a certain cool-down time has passed

- Useful for limiting API calls, resizing, scrolling
- Shows how to control function execution timing

Problem 66: Pass by Value vs Reference

Interview Frequency: 🔴 High (Common in all levels)

Question: Explain pass-by-value vs pass-by-reference.

Solution:

```
let a = 10;
let b = a; // primitive => copy

b = 20;
console.log(a); // 10

let obj1 = { x: 1 };
let obj2 = obj1; // object => reference

obj2.x = 100;
console.log(obj1.x); // 100
```

Explanation:

- Primitives are passed by value
- Objects/arrays are passed by reference
- Shows how JavaScript handles different data types

Problem 67: Custom Iterable

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make an object iterable with Symbol.iterator.

Solution:

```

const myObject = {
  data: [1, 2, 3],
  [Symbol.iterator]() {
    let index = 0;
    const data = this.data;

    return {
      next() {
        return index < data.length
          ? { value: data[index++], done: false }
          : { done: true };
      }
    };
  }
};

for (let value of myObject) {
  console.log(value); // 1, 2, 3
}

```

Explanation:

- Implements a custom iterator using Symbol.iterator
- Enables usage with for...of
- Shows how to make objects iterable

Problem 68: Lexical Scope

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Explain lexical scope with a code example.

Solution:

```
function outer() {
  let a = 10;

  function inner() {
    console.log(a);
  }

  return inner;
}

const func = outer();
func(); // 10
```

Explanation:

- `inner()` has access to `a` because of lexical scope
- Lexical scope is determined at the time of function definition, not execution
- Shows how closures work with scope

Problem 69: Array Flattening

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function that flattens nested arrays.

Solution:

```
function flatten(arr) {
  return arr.reduce((acc, val) =>
    Array.isArray(val) ? acc.concat(flatten(val)) : acc.concat(val), []);
}

console.log(flatten([1, [2, [3, [4]]]])); // [1, 2, 3, 4]
```

Explanation:

- Uses recursion with `reduce()` to flatten nested arrays of any depth
- Shows how to handle nested data structures
- Demonstrates recursive function usage

Problem 70: Promisify

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert callback to Promise (promisify).

Solution:

```
function promisify(fn) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      fn(...args, (err, result) => {
        if (err) return reject(err);
        resolve(result);
      });
    });
  };
}

// Example Node-style callback
function getData(id, callback) {
  setTimeout(() => {
    if (id) callback(null, { id, name: "Alice" });
    else callback("Error");
  }, 1000);
}

const getDataPromise = promisify(getData);

getDataPromise(1).then(console.log).catch(console.error);
```

Explanation:

- Converts callback-style functions into Promise-based
- Essential when integrating with legacy APIs
- Shows how to modernize callback patterns

Part 8: Problems 71-80

Problem 71: Throttling Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a throttle function that limits how often a function can fire.

Solution:

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

  return function (...args) {
    if (!inThrottle) {
      fn.apply(this, args);
      inThrottle = true;

      setTimeout(() => {
        inThrottle = false;
      }, limit);
    }
  };
}

// Example usage
const throttled = throttle(() => console.log('Fired!'), 1000);
window.addEventListener('scroll', throttled);
```

Explanation:

- Throttle restricts function calls to once per limit ms
- Uses closures and timing functions to control execution
- Shows how to limit function call frequency

Problem 72: Closure Counter

Interview Frequency: 🔴 High (Common in all levels)

Question: Create a counter using closures.

Solution:

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

Explanation:

- Demonstrates closures — inner function retains access to count
- Shows how to maintain private state
- Illustrates encapsulation using closures

Problem 73: Array Sum

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Sum an array of numbers using reduce.

Solution:

```
const numbers = [1, 2, 3, 4, 5];  
  
const sum = numbers.reduce((acc, curr) => acc + curr, 0);  
console.log(sum); // 15
```

Explanation:

- `reduce()` is used to accumulate values in an array
- Shows how to use array methods for calculations
- Demonstrates functional programming concepts

Problem 74: Typeof Null

Interview Frequency: 🟢 High (Common in junior interviews)

Question: What's the output of typeof null?

Solution:

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

Explanation:

- This is a well-known JavaScript bug due to legacy reasons
- `null` is not an object, but `typeof null === "object"`
- Shows JavaScript quirks and historical decisions

Problem 75: Event Bubbling

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Explain event bubbling with a demo.

Solution:

```
<div id="parent">
  <button id="child">Click Me</button>
</div>

<script>
  document.getElementById('parent').addEventListener('click', () => {
    console.log('Parent clicked');
  });

  document.getElementById('child').addEventListener('click', () => {
    console.log('Child clicked');
  });
</script>
```

Output on button click:

Child clicked
Parent clicked

Explanation:

- Event bubbling means events propagate from the innermost target outward
- You can stop propagation using `event.stopPropagation()`
- Shows how DOM events work

Problem 76: Remove Duplicates

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Remove duplicates from an array.

Solution:

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

const unique = [...new Set(arr)];
console.log(unique); // [1, 2, 3, 4]
```

Explanation:

- Set only stores unique values
- Spread operator creates a new array from the Set
- Shows modern JavaScript features for data manipulation

Problem 77: Curried Multiplication

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make this work: `multiply(2)(3)(4)` returns 24.

Solution:


```
function multiply(a) {  
  return function (b) {  
    return function (c) {  
      return a * b * c;  
    };  
  };  
}  
  
console.log(multiply(2)(3)(4)); // 24
```

Explanation:

- Uses function currying to collect arguments one at a time
- Shows how to implement partial application
- Demonstrates functional programming techniques

Problem 78: Sleep Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a sleep function using Promise.

Solution:

```
function sleep(ms) {  
  return new Promise(resolve => setTimeout(resolve, ms));  
}  
  
async function test() {  
  console.log('Start');  
  await sleep(1000);  
  console.log('After 1 second');  
}  
  
test();
```

Explanation:

- `sleep` wraps `setTimeout` in a Promise
- Enables delays using `async/await`
- Shows how to work with asynchronous code

Problem 79: Dynamic Property Keys

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create an object with dynamic property keys.

Solution:

```
const key = "name";
const user = {
  [key]: "Alice"
};

console.log(user.name); // Alice
```

Explanation:

- Uses computed property names with square brackets []
- Shows how to create dynamic object properties
- Demonstrates modern JavaScript object syntax

Problem 80: Undefined vs Null

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What's the difference between undefined and null?

Solution:

```
let a;
console.log(a); // undefined

let b = null;
console.log(b); // null
```

Explanation:

- `undefined` : A variable declared but not assigned
- `null` : Intentional absence of value
- Shows how to handle missing values in JavaScript

Part 9: Problems 81-90

Problem 81: Curried Sum Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make this work: `sum(1)(2)(3)` should return 6.

Solution:

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



```
console.log(sum(1)(2)(3)); // 6
```

Explanation:

- Demonstrates currying to chain function calls
- Shows how to implement partial application
- Illustrates closure behavior in JavaScript

Problem 82: Palindrome Check

Interview Frequency: 🟢 Very High (Common in all levels)

Question: Check if a string is a palindrome.

Solution:

```
function isPalindrome(str) {  
  const cleaned = str.toLowerCase().replace(/[^a-z0-9]/g, '');  
  return cleaned === cleaned.split('').reverse().join('');  
}
```



```
console.log(isPalindrome("A man, a plan, a canal: Panama")); // true
```

Explanation:

- Uses regex to clean string of non-alphanumeric characters
- Demonstrates string manipulation with split, reverse, and join
- Shows how to handle case sensitivity and special characters

Problem 83: Object Cloning

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Clone an object without structured cloning.

Solution:

```
const original = { a: 1, b: 2 };
const clone = { ...original };

console.log(clone); // { a: 1, b: 2 }
```

Explanation:

- Spread operator { ...obj } creates a shallow copy
- Shows modern JavaScript object copying techniques
- Demonstrates the difference between shallow and deep copying

Problem 84: Array Equality

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What will this log?

```
console.log([] == false); // ?
console.log([] === false); // ?
```

Solution:

```
// true
// false
```

Explanation:

- == allows type coercion, so [] becomes falsy.
- === compares type and value, so it's false.

Problem 85: Array Intersection

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Get the intersection of two arrays.

Solution:

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

const intersection = arr1.filter(val => arr2.includes(val));
console.log(intersection); // [2, 3]
```

Explanation:

- Uses filter() with includes() to find common elements
- Shows how to use array methods for set operations
- Demonstrates functional programming approach to array manipulation

Problem 86: setTimeout in Loop

Interview Frequency: 🔴 High (Common in all levels)

Question: Explain setTimeout inside a loop with var.

Solution:

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

// Output: 3, 3, 3

// Fix:
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1000); // 0, 1, 2
}
```

Explanation:

- var is function-scoped. Use let for block scoping.

Problem 87: Remove Falsy Values

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Remove falsy values from an array.

Solution:

```
const arr = [0, 1, false, '', null, 'hello'];
```

```
const filtered = arr.filter(Boolean);  
console.log(filtered); // [1, "hello"]
```

Explanation:

- Boolean constructor used as a callback to remove all falsy values
- Shows how to use filter with type coercion
- Demonstrates JavaScript's falsy values

Problem 88: Once Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a once function (runs only once).

Solution:

```
function once(fn) {
  let called = false;
  let result;

  return function (...args) {
    if (!called) {
      result = fn.apply(this, args);
      called = true;
    }
    return result;
  };
}

// Test
const init = once(() => console.log("Initialized!"));
init(); // "Initialized!"
init(); // (nothing)
```

Explanation:

- Uses closure to remember if function was called
- Shows function decorator pattern
- Demonstrates state management with closures

Problem 89: Group Array by Property

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Group an array of objects by a property.

Solution:

```
const people = [
  { name: "Alice", role: "admin" },
  { name: "Bob", role: "user" },
  { name: "Charlie", role: "admin" }
];

const grouped = people.reduce((acc, person) => {
  const role = person.role;
  if (!acc[role]) acc[role] = [];
  acc[role].push(person);
  return acc;
}, {});

console.log(grouped);
// Output: { admin: [{ name: "Alice", role: "admin" }, { name: "Charlie", role: "admin" }], user: [{ name: "Bob", role: "user" }]}
```

Explanation:

- Uses `reduce()` to create a grouped object
- Demonstrates object manipulation and array methods
- Shows how to organize data by property

Problem 90: Function and Variable Hoisting

Interview Frequency:  High (Common in junior interviews)

Question: Explain hoisting with functions and variables.

Solution:

```
greet(); // "Hello"
function greet() {
  console.log("Hello");
}

console.log(a); // undefined
var a = 5;
```

Explanation:

- Function declarations are hoisted with their definitions

- var variables are hoisted but initialized to undefined
- Demonstrates JavaScript's hoisting behavior

Part 10: Problems 91-100

Problem 91: Filter Polyfill

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a polyfill for `Array.prototype.filter`.

Solution:

```
Array.prototype.myFilter = function (callback) {  
  const result = [];  
  
  for (let i = 0; i < this.length; i++) {  
    if (callback(this[i], i, this)) {  
      result.push(this[i]);  
    }  
  }  
  
  return result;  
};  
  
console.log([1, 2, 3].myFilter(x => x > 1)); // [2, 3]
```

Explanation:

- Reimplements `filter()` behavior manually
- Useful for polyfills and understanding prototypes
- Shows how to implement array methods

Problem 92: forEach vs Map

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What's the difference between `forEach` and `map`?

Solution:

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

// forEach: executes a function on each element but doesn't return anything
arr.forEach(x => console.log(x * 2)); // 2, 4, 6
console.log(arr); // [1, 2, 3] (unchanged)

// map: transforms and returns a new array
const doubled = arr.map(x => x * 2);
console.log(doubled); // [2, 4, 6]
console.log(arr); // [1, 2, 3] (unchanged)
```

Explanation:

- forEach: executes a function on each element but doesn't return anything
- map: transforms and returns a new array
- Shows the difference between methods that modify and those that return new values

Problem 93: Arrow Function This

Interview Frequency:  High (Common in all levels)

Question: How do arrow functions handle this?

Solution:

```
const obj = {
  name: 'Alice',
  greet: () => {
    console.log(this.name);
  }
};

obj.greet(); // undefined
```

Explanation:

- Arrow functions don't bind their own this, they inherit it from lexical context
- Regular functions create their own this context
- Shows how to maintain context in callbacks

Problem 94: Element Frequency Counter

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Count frequency of elements in an array.

Solution:

```
const items = ['a', 'b', 'a', 'c', 'b'];

const freq = items.reduce((acc, item) => {
  acc[item] = (acc[item] || 0) + 1;
  return acc;
}, {});

console.log(freq); // { a: 2, b: 2, c: 1 }
```

Explanation:

- Pattern to count elements using reduce()
- Shows how to build objects from arrays
- Demonstrates accumulator pattern

Problem 95: Array to Object Conversion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert array to object with key as value.

Solution:

```
const arr = ['a', 'b', 'c'];

const obj = arr.reduce((acc, item) => {
  acc[item] = true;
  return acc;
}, {});

console.log(obj); // { a: true, b: true, c: true }
```

Explanation:

- Maps array elements to object keys
- Shows how to transform data structures
- Demonstrates accumulator pattern with objects

Problem 96: Call, Apply, and Bind

Interview Frequency: 🔴 High (Common in all levels)

Question: What is the difference between call, apply, and bind?

Solution:

```
function greet(greeting, punctuation) {  
  console.log(`${greeting}, ${this.name}${punctuation}`);  
}  
  
const person = { name: "Alice" };  
  
greet.call(person, "Hello", "!"); // Hello, Alice!  
greet.apply(person, ["Hi", "."]); // Hi, Alice.  
const boundGreet = greet.bind(person, "Hey");  
boundGreet("?"); // Hey, Alice?
```

Explanation:

- call: invokes function immediately, passing arguments one by one
- apply: same as call but args are an array
- bind: returns a new function with bound context and optional preset args

Problem 97: Array to Tree Structure

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert a flat array to a nested object tree.

Solution:

```

const items = [
  { id: 1, name: "root", parentId: null },
  { id: 2, name: "child1", parentId: 1 },
  { id: 3, name: "child2", parentId: 1 }
];

function buildTree(items, parentId = null) {
  return items
    .filter(item => item.parentId === parentId)
    .map(item => ({
      ...item,
      children: buildTree(items, item.id)
    }));
}

console.log(JSON.stringify(buildTree(items), null, 2));

```

Explanation:

- Uses recursion to create a tree
- Filters by parentId, builds children recursively
- Shows how to transform flat data into hierarchical structures

Problem 98: Temporal Dead Zone

Interview Frequency:  High (Common in junior interviews)

Question: What is the temporal dead zone (TDZ)?

Solution:

```

// console.log(x); // ReferenceError
let x = 10;

```

Explanation:

- TDZ refers to the time between entering scope and variable declaration
- Accessing let/const before declaration throws ReferenceError
- Shows how let and const differ from var in terms of hoisting

Problem 99: Deep Clone

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a deep clone function.

Solution:

```
function deepClone(obj) {
  if (obj === null || typeof obj !== "object") return obj;

  if (Array.isArray(obj)) {
    return obj.map(deepClone);
  }

  const clone = {};
  for (let key in obj) {
    clone[key] = deepClone(obj[key]);
  }

  return clone;
}

// Test
const original = { a: 1, b: { c: 2 } };
const copy = deepClone(original);
copy.b.c = 99;
console.log(original.b.c); // 2
```

Explanation:

- Recursively clones all nested objects and arrays
- Demonstrates recursion and object traversal
- Shows how to create true copies of complex objects

Problem 100: Variable Hoisting in IIFE

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What will the output be? Why?

```
let a = 10;
(function () {
  console.log(a);
  var a = 5;
})();
```

Solution:

```
// Output: undefined
```

Explanation:

- var a is hoisted, so function scope has its own a, initialized as undefined at the top
- Demonstrates hoisting behavior in Immediately Invoked Function Expressions (IIFE)

Problem 101: Merge Sorted Arrays

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Merge two sorted arrays into one sorted array.

Solution:

```
function mergeSorted(arr1, arr2) {
  let result = [];
  let i = 0, j = 0;

  while (i < arr1.length && j < arr2.length) {
    if (arr1[i] < arr2[j]) result.push(arr1[i++]);
    else result.push(arr2[j++]);
  }

  return result.concat(arr1.slice(i)).concat(arr2.slice(j));
}

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

Explanation:

- Classic two-pointer technique, useful for merge sort logic
- Demonstrates array manipulation and sorting algorithms

- Shows how to merge sorted arrays efficiently

Problem 102: Array Flattening

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a function that flattens a nested array.

Solution:

```
function flatten(arr) {  
  return arr.reduce((acc, val) =>  
    Array.isArray(val) ? acc.concat(flatten(val)) : acc.concat(val), []);  
}  
  
console.log(flatten([1, [2, [3, [4]]]])); // [1, 2, 3, 4]
```

Explanation:

- Uses recursion with `reduce()` and `concat()`
- Shows how to handle nested data structures
- Demonstrates functional programming concepts

Problem 103: Object Freeze

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the use of `Object.freeze()`?

Solution:

```
const obj = { name: "Alice" };  
Object.freeze(obj);  
obj.name = "Bob";  
  
console.log(obj.name); // Alice
```

Explanation:

- `Object.freeze()` makes object immutable — no new props, no value change
- Shows how to create immutable objects

- Demonstrates JavaScript's object immutability features

Problem 104: Empty Object Check

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create an isEmpty function for objects.

Solution:

```
function isEmpty(obj) {  
  return Object.keys(obj).length === 0;  
}  
  
console.log(isEmpty({})); // true  
console.log(isEmpty({ a: 1 })); // false
```

Explanation:

- Object.keys() returns an array of property names
- Shows how to check object properties
- Demonstrates object manipulation

Problem 105: Primitive vs Reference Types

Interview Frequency: 🔴 High (Common in all levels)

Question: What's the difference between primitive and reference types?

Solution:

```
// Primitives: stored by value
let a = 5;
let b = a;
b++;
console.log(a); // 5

// Reference: stored as a pointer
let obj1 = { x: 1 };
let obj2 = obj1;
obj2.x = 9;
console.log(obj1.x); // 9
```

Explanation:

- Primitives: stored by value (string, number, boolean, null, undefined, symbol, bigint)
- Reference: stored as a pointer (objects, arrays, functions)
- Shows how JavaScript handles different data types

Problem 106: Optional Chaining

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is optional chaining?

Solution:

```
const user = { profile: { name: "Alice" } };

console.log(user.profile?.name); // Alice
console.log(user.contact?.email); // undefined
```

Explanation:

- `?.` avoids errors when accessing deeply nested properties that may not exist
- Shows modern JavaScript features for safe property access
- Demonstrates null-safe property access

Problem 107: Map Polyfill

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a custom map function (polyfill).

Solution:

```
Array.prototype.myMap = function (callback) {  
  const result = [];  
  for (let i = 0; i < this.length; i++) {  
    result.push(callback(this[i], i, this));  
  }  
  return result;  
};  
  
console.log([1, 2, 3].myMap(x => x * 2)); // [2, 4, 6]
```

Explanation:

- Custom implementation of map() using for loop and callback
- Shows how to implement array methods
- Demonstrates prototype extension

Problem 108: Case Conversion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert camelCase to kebab-case.

Solution:

```
function camelToKebab(str) {  
  return str.replace(/[A-Z]/g, m => "-" + m.toLowerCase());  
}  
  
console.log(camelToKebab("backgroundColor")); // background-color
```

Explanation:

- Uses replace() with regex and toLowerCase() to insert hyphens
- Shows string manipulation with regular expressions
- Demonstrates case conversion techniques

Problem 109: Debounce Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement debounce function.

Solution:

```
function debounce(fn, delay) {  
  let timeout;  
  
  return function (...args) {  
    clearTimeout(timeout);  
    timeout = setTimeout(() => fn.apply(this, args), delay);  
  };  
}  
  
const log = debounce(() => console.log("Typing..."), 500);  
window.addEventListener("keyup", log);
```

Explanation:

- Debounce delays execution until user stops calling function for a while
- Uses closures and timers
- Shows how to optimize event handlers

Problem 110: Pass by Value vs Reference

Interview Frequency: 🔴 High (Common in all levels)

Question: Explain pass by value vs pass by reference.

Solution:

```
// Primitives are passed by value
let num = 5;
function inc(x) { x++; }
inc(num);
console.log(num); // 5 (primitive)

// Objects/arrays are passed by reference
let obj = { val: 5 };
function mutate(o) { o.val++; }
mutate(obj);
console.log(obj.val); // 6 (reference)
```

Explanation:

- Primitives are passed by value — copies are created
- Objects/arrays are passed by reference — both variables point to same memory
- Shows how JavaScript handles parameter passing

Problem 111: Unique Array Values

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to get unique values from an array.

Solution:

```
function getUnique(arr) {
  return [...new Set(arr)];
}

console.log(getUnique([1, 2, 2, 3, 4, 4, 5])); // [1, 2, 3, 4, 5]
```

Explanation:

- Set stores only unique values
- Spread operator ... converts the set back into an array
- Shows modern JavaScript features for data manipulation

Problem 112: Array Equality Check

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to check if two arrays are equal?

Solution:

```
function arraysEqual(a, b) {  
  if (a.length !== b.length) return false;  
  return a.every((val, index) => val === b[index]);  
}  
  
console.log(arraysEqual([1, 2], [1, 2])); // true  
console.log(arraysEqual([1, 2], [2, 1])); // false
```

Explanation:

- Uses `every()` to compare values at each index
- Shows how to compare arrays element by element
- Demonstrates array methods and strict equality

Problem 113: Event Emitter Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a simple event emitter.

Solution:

```

class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, fn) {
    (this.events[event] || []).push(fn);
  }

  emit(event, data) {
    (this.events[event] || []).forEach(fn => fn(data));
  }

  off(event, fn) {
    this.events[event] = (this.events[event] || []).filter(f => f !== fn);
  }
}

// Usage
const emitter = new EventEmitter();
function greet(name) { console.log("Hello", name); }

emitter.on("hi", greet);
emitter.emit("hi", "Alice"); // Hello Alice
emitter.off("hi", greet);
emitter.emit("hi", "Bob");    // No output

```

Explanation:

- Mimics Node.js EventEmitter
- Shows how to implement pub/sub pattern
- Demonstrates class implementation and event handling

Problem 114: Word Reversal

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to reverse each word in a sentence.

Solution:

```
function reverseWords(str) {
  return str
    .split(" ")
    .map(word => word.split("").reverse().join(""))
    .join(" ");
}

console.log(reverseWords("hello world")); // "olleh dlrow"
```

Explanation:

- Splits sentence into words, reverses each word
- Shows string manipulation with split, map, and join
- Demonstrates chaining array methods

Problem 115: Once Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a once function (runs only once).

Solution:

```
function once(fn) {
  let called = false;
  return function (...args) {
    if (!called) {
      called = true;
      return fn.apply(this, args);
    }
  };
}

// Test
const init = once(() => console.log("Initialized!"));
init(); // "Initialized!"
init(); // nothing
```

Explanation:

- Uses closure to remember if function was called

- Shows function decorator pattern
- Demonstrates state management with closures

Problem 116: Typeof Null

Interview Frequency: 🟢 High (Common in junior interviews)

Question: Explain typeof null and its quirk.

Solution:

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

Explanation:

- This is a legacy bug in JavaScript
- `null` is a primitive, but `typeof null` returns "object"
- Shows JavaScript's historical quirks

Problem 117: Fibonacci Recursive

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to get the nth Fibonacci number (recursive).

Solution:

```
function fib(n) {  
  if (n <= 1) return n;  
  return fib(n - 1) + fib(n - 2);  
}
```

```
console.log(fib(6)); // 8
```

Explanation:

- Recursive approach to Fibonacci
- Poor performance for large n (exponential time)
- Shows basic recursion implementation

Problem 118: Memoized Fibonacci

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Improve Fibonacci with memoization.

Solution:

```
function memoizedFib() {  
  const cache = {};  
  return function fib(n) {  
    if (n in cache) return cache[n];  
    if (n <= 1) return n;  
    return (cache[n] = fib(n - 1) + fib(n - 2));  
  };  
}  
  
const fib = memoizedFib();  
console.log(fib(40)); // Fast result
```

Explanation:

- Uses memoization (caching) to improve performance
- Shows how to optimize recursive functions
- Demonstrates closure for cache storage

Problem 119: Equality Comparisons

Interview Frequency: 🔴 High (Common in all levels)

Question: How does JavaScript handle equality comparisons?

Solution:

```
console.log("5" == 5); // true (abstract equality)  
console.log("5" === 5); // false (strict equality)
```

Explanation:

- == performs type coercion (abstract equality)
- === checks type and value (strict equality)

- Shows JavaScript's type coercion behavior

Problem 120: Character Count

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Count character occurrences in a string.

Solution:

```
function countChars(str) {  
  return str.split('').reduce((acc, char) => {  
    acc[char] = (acc[char] || 0) + 1;  
    return acc;  
  }, {});  
}  
  
console.log(countChars("hello")); // { h: 1, e: 1, l: 2, o: 1 }
```

Explanation:

- Uses `reduce()` to build frequency map
- Shows how to build objects from arrays
- Demonstrates accumulator pattern

Problem 121: Missing Number

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Find missing number in an array 1 to N.

Solution:

```
function missingNumber(arr, n) {  
  const expectedSum = (n * (n + 1)) / 2;  
  const actualSum = arr.reduce((a, b) => a + b, 0);  
  return expectedSum - actualSum;  
}  
  
console.log(missingNumber([1, 2, 4, 5], 5)); // 3
```

Explanation:

- Uses sum formula to find missing number
- Shows mathematical approach to array problems
- Demonstrates array reduction

Problem 122: Variable Declaration

Interview Frequency: 🔴 High (Common in all levels)

Question: Explain the difference between let, const, and var.

Solution:

```
var x = 10; // Function-scoped
let y = 20; // Block-scoped
const z = 30; // Block-scoped and cannot be reassigned

if (true) {
  var x = 40;
  let y = 50;
  const z = 60;
}

console.log(x); // 40
// console.log(y); // ReferenceError
// console.log(z); // ReferenceError
```

Explanation:

- var: function-scoped, hoisted
- let/const: block-scoped, not hoisted in usable way
- Shows modern JavaScript variable declarations

Problem 123: Object Cloning

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to clone an object using spread?

Solution:

```
const original = { a: 1, b: 2 };
const copy = { ...original };
copy.a = 99;

console.log(original.a); // 1
```

Explanation:

- Spread operator { ...obj } creates a shallow copy
- Shows modern object copying techniques
- Demonstrates object immutability

Problem 124: Private Variables

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to create private variables using closures?

Solution:

```
function createCounter() {
  let count = 0;

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

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.count);      // undefined
```

Explanation:

- count is not exposed, thanks to closure
- Only accessible via methods
- Shows encapsulation using closures

Problem 125: IIFE

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is IIFE (Immediately Invoked Function Expression)?

Solution:

```
(function () {  
  console.log("IIFE ran!");  
})();
```

Explanation:

- Runs immediately after definition
- Used to create private scopes
- Shows function expression syntax

Problem 126: Async Operations

Interview Frequency: 🔴 High (Common in all levels)

Question: How does JavaScript handle async operations?

Solution:

```
console.log("Start");  
  
setTimeout(() => {  
  console.log("Timeout");  
}, 0);  
  
Promise.resolve().then(() => {  
  console.log("Promise");  
});  
  
console.log("End");
```

Explanation:

- Via the event loop
- Uses callback queue and microtask queue for promises
- Shows JavaScript's asynchronous behavior

Problem 127: Query String

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert an object to query string.

Solution:

```
function toQueryString(obj) {  
  return Object.entries(obj)  
    .map(([k, v]) => `${encodeURIComponent(k)}=${encodeURIComponent(v)}`)  
    .join("&");  
}  
  
console.log(toQueryString({ name: "John", age: 30 }));  
// name=John&age=30
```

Explanation:

- Object.entries() gets key-value pairs
- encodeURIComponent() handles URL-safe formatting
- Shows URL parameter handling

Problem 128: Array Truthiness

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the output?

```
let x = [];  
if (x) console.log("Truthy");  
else console.log("Falsy");
```

Solution:

```
// Output: Truthy
```

Explanation:

- Empty arrays and objects are truthy in JS
- Shows JavaScript's truthiness rules
- Demonstrates type coercion in conditionals

Problem 129: Promise Delay

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a delay function using Promises.

Solution:

```
function delay(ms) {  
  return new Promise(resolve => setTimeout(resolve, ms));  
}  
  
delay(1000).then(() => console.log("1 second later"));
```

Explanation:

- Promise wrapper around setTimeout()
- Allows for await delay() usage
- Shows Promise-based timing

Problem 130: String Number Coercion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the output?

```
console.log(1 + "2" + "2"); // "122"  
console.log(1 + +"2" + "2"); // "32"
```

Solution:


```
// Output:  
// "122"  
// "32"
```

Explanation:

- "2" → string
- +"2" → number coercion to 2
- JS performs left-to-right evaluation with type coercion
- Shows JavaScript's type coercion rules

Problem 131: Closure in Loop

Interview Frequency:  High (Common in all levels)

Question: What's the output of this closure-based loop example?

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

Solution:

```
// Output:  
// 3  
// 3  
// 3
```

Explanation:

- var is function-scoped
- All 3 callbacks share the same reference to i, which becomes 3 after the loop
- To fix it, use let i or an IIFE

Problem 132: Let in Loop

Interview Frequency:  High (Common in all levels)

Question: Fix the closure to log 0,1,2 correctly using let.

Solution:

```
for (let i = 0; i < 3; i++) {  
  setTimeout(() => console.log(i), 1000);  
}
```

Explanation:

- let creates a new block scope for each iteration
- Each iteration gets its own i value

Problem 133: IIFE in Loop

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Fix the closure using an IIFE.

Solution:

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

Explanation:

- IIFE captures the value of i during each iteration using a parameter
- Creates a new scope for each iteration

Problem 134: Debounce Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a function to debounce another function.

Solution:

```
function debounce(fn, delay) {
  let timer;
  return function (...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

// Usage
const log = debounce(() => console.log("Debounced!"), 500);
log(); log(); log(); // Only one "Debounced!" after 500ms
```

Explanation:

- Debounce delays execution until pause in rapid calls
- Useful for search inputs, window resize events
- Shows how to control function execution timing

Problem 135: Throttle Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a throttle function.

Solution:

```
function throttle(fn, limit) {
  let wait = false;
  return function (...args) {
    if (!wait) {
      fn.apply(this, args);
      wait = true;
      setTimeout(() => (wait = false), limit);
    }
  };
}
```

Explanation:

- Throttling ensures function runs at most once per interval
- Useful for scroll events, API calls

- Shows how to limit function call frequency

Problem 136: Temporal Dead Zone

Interview Frequency: 🚫 High (Common in junior interviews)

Question: Explain temporal dead zone.

Solution:

```
console.log(a); // ReferenceError
let a = 5;
```

Explanation:

- let and const are hoisted but not initialized
- This period is the temporal dead zone
- Accessing variables in TDZ throws ReferenceError

Problem 137: Scope Chain

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Explain scope chain with an example.

Solution:

```
function outer() {
  const a = 10;
  function inner() {
    console.log(a); // 10
  }
  inner();
}
outer();
```

Explanation:

- inner() has access to a because of scope chain
- Shows lexical scoping in JavaScript
- Demonstrates closure behavior

Problem 138: Call Method

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Use call to borrow a method.

Solution:

```
const person = {
  name: "Alice",
  greet() {
    console.log("Hello " + this.name);
  }
};

const another = { name: "Bob" };
person.greet.call(another); // Hello Bob
```

Explanation:

- call() sets the this context manually
- Allows borrowing methods from other objects
- Shows function context manipulation

Problem 139: Apply Method

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Use apply with a function and arguments.

Solution:

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

console.log(sum.apply(null, [3, 4])); // 7
```

Explanation:

- apply() is like call(), but takes arguments as an array
- Useful when arguments are in array form

- Shows array to argument list conversion

Problem 140: Bind Method

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Use bind to permanently bind a context.

Solution:

```
const user = {
  name: "Sam",
  greet() {
    console.log("Hi " + this.name);
  }
};

const greetFn = user.greet.bind(user);
setTimeout(greetFn, 500); // Hi Sam
```

Explanation:

- bind() returns a new function with this permanently set
- Useful for event handlers and callbacks
- Shows how to preserve context

Problem 141: Array Addition

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the result of [] + []?

Solution:

```
console.log([] + []); // ""
```

Explanation:

- Empty arrays are converted to "", so it's string concatenation
- Shows JavaScript's type coercion rules
- Demonstrates array to string conversion

Problem 142: Array Object Addition

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the result of `[] + {}` and `{ } + []`?

Solution:

```
console.log([] + {}); // "[object Object]"
console.log({} + []); // 0 or "[object Object]" depending on context
```

Explanation:

- Depends on parsing
- First is string coercion
- Second might be interpreted as block `{}` + array (leading to 0)
- Shows JavaScript's parsing rules

Problem 143: Array Flattening

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Flatten a nested array.

Solution:

```
function flatten(arr) {
  return arr.reduce((acc, val) =>
    Array.isArray(val) ? acc.concat(flatten(val)) : acc.concat(val), []);
}

console.log(flatten([1, [2, [3, 4]]])); // [1, 2, 3, 4]
```

Explanation:

- Uses recursion with `reduce()`
- Shows how to handle nested arrays
- Demonstrates recursive array manipulation

Problem 144: Deep Clone

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a deep clone function.

Solution:

```
function deepClone(obj) {
  if (obj === null || typeof obj !== "object") return obj;
  if (Array.isArray(obj)) return obj.map(deepClone);

  const clone = {};
  for (let key in obj) {
    clone[key] = deepClone(obj[key]);
  }
  return clone;
}
```

Explanation:

- Recursively clones nested values
- Handles arrays and objects
- Shows deep copying implementation

Problem 145: Copy Types

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Explain shallow vs deep copy.

Solution:

```
// Shallow copy
const obj1 = { a: 1, b: { c: 2 } };
const obj2 = { ...obj1 }; // or Object.assign({}, obj1)

// Deep copy
const obj3 = JSON.parse(JSON.stringify(obj1));
```

Explanation:

- Shallow copy copies top-level references
- Deep copy duplicates all nested objects/arrays
- Shows different copying strategies

Problem 146: Event Bubbling

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is event bubbling in DOM?

Solution:

```
document.getElementById('parent').addEventListener('click', () => {
  console.log('Parent clicked');
});

document.getElementById('child').addEventListener('click', (e) => {
  console.log('Child clicked');
  // e.stopPropagation(); // Stop bubbling
});
```

Explanation:

- When an event occurs, it bubbles up from the target to ancestors
- Can be stopped using `e.stopPropagation()`
- Shows event propagation in DOM

Problem 147: Prevent Default

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to stop default action on event?

Solution:

```
document.querySelector("a").addEventListener("click", (e) => {
  e.preventDefault();
});
```

Explanation:

- `preventDefault()` stops anchor link from navigating
- Shows how to control default browser behavior
- Demonstrates event handling

Problem 148: Promise.all

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How does `Promise.all()` behave?

Solution:

```
Promise.all([
  Promise.resolve(1),
  Promise.resolve(2)
]).then(console.log); // [1, 2]
```

Explanation:

- Waits for all promises to resolve
- If any fail, the whole fails
- Shows parallel promise handling

Problem 149: Promise.race

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How does `Promise.race()` behave?

Solution:

```
Promise.race([
  new Promise(r => setTimeout(() => r("first"), 100)),
  new Promise(r => setTimeout(() => r("second"), 200)),
]).then(console.log); // "first"
```

Explanation:

- Resolves/rejects with the first result
- Useful for timeouts and fallbacks
- Shows competitive promise handling

Problem 150: Typeof NaN

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: What is the output of `typeof NaN`?

Solution:

```
console.log(typeof NaN); // "number"
```

Explanation:

- NaN is of type number despite being "Not-a-Number"
- Shows JavaScript's type system quirks
- Demonstrates special number values

Problem 151: Null vs Undefined Equality

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What is the result of `null == undefined`?

Solution:

```
console.log(null == undefined); // true
```

Explanation:

- In non-strict equality, null and undefined are loosely equal
- But `null === undefined` is false

Problem 152: Optional Chaining

Interview Frequency: 🔴 High (Common in mid-level interviews)

Question: Use optional chaining to avoid `TypeError` when accessing deeply nested properties.

Solution:

```
const user = { profile: { name: "Alex" } };  
console.log(user?.profile?.name); // "Alex"  
console.log(user?.settings?.theme); // undefined, not error
```

Explanation:

- Optional chaining ?. avoids accessing undefined/null chains
- Provides safe access to nested properties

Problem 153: Loose vs Strict Equality

Interview Frequency: 🟡 High (Common in junior interviews)

Question: Explain the difference between == and ===.

Solution:

```
console.log(5 == "5"); // true  
console.log(5 === "5"); // false
```

Explanation:

- == does type coercion before comparing
- === checks both value and type
- Best practice is to use === for predictable comparisons

Problem 154: String to Number Conversion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Show different ways to convert a string to a number.

Solution:

```
let a = "123";  
console.log(+a); // 123  
console.log(Number(a)); // 123  
console.log(parseInt(a)); // 123
```

Explanation:

- Unary + is the fastest for string-number coercion
- Number() and parseInt() provide more control over conversion

Problem 155: Number to String Conversion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Show different ways to convert a number to a string.

Solution:

```
let num = 42;
console.log(num + "");      // "42"
console.log(String(num));   // "42"
console.log(num.toString()); // "42"
```

Explanation:

- All forms convert number to string
- String() is the most explicit method
- toString() can throw on null/undefined

Problem 156: Boolean Conversion

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Show how to convert values to boolean.

Solution:

```
console.log(Boolean("")); // false
console.log(!!"hello");   // true
```

Explanation:

- !! is shorthand for boolean coercion
- Falsy values: "", 0, null, undefined, NaN, false
- Everything else is truthy

Problem 157: Array Methods Comparison

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Explain the difference between `slice()` and `splice()`.

Solution:

```
const arr = [1, 2, 3, 4, 5];
console.log(arr.slice(1, 3)); // [2, 3]
console.log(arr.splice(1, 2)); // [2, 3]
console.log(arr);             // [1, 4, 5]
```

Explanation:

- `slice(start, end)` → returns a shallow copy, non-mutating
- `splice(start, deleteCount)` → modifies original array
- `slice` is for reading, `splice` is for modifying

Problem 158: Promise-based Delay

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a delay function using promises.

Solution:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(1000).then(() => console.log("1 second later"));
```

Explanation:

- Useful in `async/await` and retry strategies
- Creates a promise that resolves after specified time
- Can be used with `async/await` for cleaner code

Problem 159: Async Retry Logic

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a retry logic for async function.

Solution:

```
async function retry(fn, attempts = 3) {  
  for (let i = 0; i < attempts; i++) {  
    try {  
      return await fn();  
    } catch (e) {  
      if (i === attempts - 1) throw e;  
    }  
  }  
}
```

Explanation:

- Retries a promise-returning function up to N times
- Useful for handling transient failures
- Throws the last error if all attempts fail

Problem 160: Variable Hoisting

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What is hoisting in JavaScript?

Solution:

```
console.log(a); // undefined  
var a = 5;
```

Explanation:

- Variable and function declarations are moved to top of their scope
- Only var and function are hoisted
- let and const are hoisted but not initialized (TDZ)

Problem 161: Temporal Dead Zone

Interview Frequency: 🔴 High (Common in junior interviews)

Question: What happens when accessing let/const before declaration?

Solution:

```
console.log(b); // ReferenceError
let b = 10;
```

Explanation:

- let and const are hoisted but not initialized
- They stay in Temporal Dead Zone (TDZ)
- Accessing them in TDZ throws ReferenceError

Problem 162: Variable Declaration Keywords

Interview Frequency: 🔴 High (Common in junior interviews)

Question: Explain the differences between const, let, and var.

Solution:

```
var x = 1;    // function-scoped, hoisted
let y = 2;    // block-scoped, not hoisted
const z = 3;  // block-scoped, immutable
```

Explanation:

- const: Block-scoped, immutable binding
- let: Block-scoped, mutable
- var: Function-scoped, hoisted and initialized as undefined

Problem 163: Closure Scope

Interview Frequency: 🔴 High (Common in mid-level interviews)

Question: Find the output of the following closure.

Solution:

```
function outer() {  
  let a = 10;  
  return function inner() {  
    console.log(a++);  
  };  
}  
  
const fn = outer();  
fn(); // 10  
fn(); // 11
```

Explanation:

- The a variable is preserved by closure
- Each call to fn() increments the preserved value
- Demonstrates lexical scoping

Problem 164: Private Variables

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make a private variable using closure.

Solution:

```
function counter() {  
  let count = 0;  
  return {  
    increment: () => ++count,  
    get: () => count  
  };  
}  
  
const c = counter();  
console.log(c.increment()); // 1  
console.log(c.get()); // 1
```

Explanation:

- count is private and accessible only via returned functions
- Demonstrates encapsulation using closures

- Provides controlled access to internal state

Problem 165: Basic Currying

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a curried function to add numbers.

Solution:

```
function add(a) {  
  return function (b) {  
    return a + b;  
  };  
}  
  
console.log(add(2)(3)); // 5
```

Explanation:

- Currying transforms a function into a chain of unary functions
- Each function takes one argument and returns another function
- Useful for partial application

Problem 166: Multiple Argument Currying

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Chain multiple curry functions.

Solution:

```
function sum(a) {  
  return function (b) {  
    return function (c) {  
      return a + b + c;  
    };  
  };  
}  
  
console.log(sum(1)(2)(3)); // 6
```

Explanation:

- Demonstrates function chaining with multiple arguments
- Each function returns another function until final calculation
- Shows nested closure behavior

Problem 167: Infinite Currying

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement infinite currying.

Solution:

```
function sum(a) {  
  return function (b) {  
    if (b) return sum(a + b);  
    return a;  
  };  
}  
  
console.log(sum(1)(2)(3)(4)()); // 10
```

Explanation:

- Curries until empty call () returns accumulated value
- Allows for flexible number of arguments
- Demonstrates recursive currying

Problem 168: Map Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement .map() from scratch.

Solution:

```
Array.prototype.customMap = function (callback) {  
  let result = [];  
  for (let i = 0; i < this.length; i++) {  
    result.push(callback(this[i], i, this));  
  }  
  return result;  
};  
  
console.log([1, 2, 3].customMap(x => x * 2)); // [2, 4, 6]
```

Explanation:

- Mimics native Array.prototype.map
- Takes a callback function with value, index, and array
- Returns new array with transformed values

Problem 169: Filter Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement .filter() from scratch.

Solution:

```
Array.prototype.customFilter = function (callback) {  
  let result = [];  
  for (let i = 0; i < this.length; i++) {  
    if (callback(this[i], i, this)) {  
      result.push(this[i]);  
    }  
  }  
  return result;  
};  
  
console.log([1, 2, 3].customFilter(x => x > 1)); // [2, 3]
```

Explanation:

- Only includes items where callback returns true
- Creates new array with filtered elements
- Preserves original array

Problem 170: Reduce Implementation

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement `.reduce()` from scratch.

Solution:

```
Array.prototype.customReduce = function (callback, initialValue) {  
  let accumulator = initialValue;  
  for (let i = 0; i < this.length; i++) {  
    accumulator = callback(accumulator, this[i], i, this);  
  }  
  return accumulator;  
};  
  
console.log([1, 2, 3].customReduce((acc, val) => acc + val, 0)); // 6
```

Explanation:

- Simulates behavior of reduce by accumulating values
- Takes callback and optional initial value
- Returns single accumulated result

Problem 171: String Reversal

Interview Frequency: 🟡 Medium (Common in junior interviews)

Question: Create a function that reverses a string without using `reverse()`.

Solution:

```
function reverseString(str) {  
  let reversed = '';  
  for (let i = str.length - 1; i >= 0; i--) {  
    reversed += str[i];  
  }  
  return reversed;  
}  
  
console.log(reverseString("hello")); // "olleh"
```

Explanation:

- Manually iterates from the end of the string
- Builds a new string character by character
- Demonstrates basic string manipulation and loops

Problem 172: Array Flattening

Interview Frequency:  High (Common in mid-level interviews)

Question: Write a function to flatten a nested array.

Solution:


```
function flattenArray(arr) {  
  return arr.reduce((flat, toFlatten) => {  
    return flat.concat(Array.isArray(toFlatten) ? flattenArray(toFlatten) : toFlatten);  
  }, []);  
}
```

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

Explanation:

- Uses recursion with reduce to process each element
- Recursively flattens nested arrays
- Demonstrates array manipulation and recursion

Problem 173: Power of Two Check

Interview Frequency:  Medium (Common in mid-level interviews)

Question: How to check if a number is a power of two?

Solution:

```
function isPowerOfTwo(n) {  
    return n > 0 && (n & (n - 1)) === 0;  
}  
  
console.log(isPowerOfTwo(8)); // true  
console.log(isPowerOfTwo(10)); // false
```

Explanation:

- A power of 2 has only one bit set in binary
- $n \& (n-1)$ removes the lowest set bit
- Result is zero if only one bit was set

Problem 174: Function Memoization

Interview Frequency:  High (Common in mid-level interviews)

Question: Create a memoized version of a function.

Solution:

```

function memoize(fn) {
  const cache = {};
  return function(...args) {
    const key = args.join(',');
    if (cache[key]) {
      return cache[key];
    } else {
      const result = fn(...args);
      cache[key] = result;
      return result;
    }
  }
}

const slowSquare = n => {
  for (let i = 0; i < 1e6; i++); // simulate delay
  return n * n;
};

const fastSquare = memoize(slowSquare);
console.log(fastSquare(4)); // 16
console.log(fastSquare(4)); // 16 (cached)

```

Explanation:

- Caches function results for repeated calls
- Uses closure to maintain cache
- Improves performance for expensive calculations

Problem 175: Debounce Implementation

Interview Frequency:  High (Common in mid-level interviews)

Question: Implement a debounce function.

Solution:



```
function debounce(fn, delay) {  
  let timeout;  
  return function(...args) {  
    clearTimeout(timeout);  
    timeout = setTimeout(() => fn.apply(this, args), delay);  
  }  
}
```

```
const sayHello = () => console.log("Hello");  
const debouncedHello = debounce(sayHello, 1000);  
debouncedHello();  
debouncedHello();  
debouncedHello();
```

Explanation:

- Ensures function only executes after pause in events
- Prevents rapid firing of function calls
- Useful for search inputs and window resize events

Problem 176: Equality Operators

Interview Frequency:  High (Common in junior interviews)

Question: Explain the difference between == and ===.

Solution:

```
console.log(5 == "5"); // true  
console.log(5 === "5"); // false
```

Explanation:

- == performs type coercion before comparison
- === checks both value and type
- Best practice is to use === for predictable comparisons

Problem 177: Arrow Function This

Interview Frequency:  High (Common in mid-level interviews)

Question: How does this behave in arrow functions?

Solution:

```
const obj = {
  name: "Alice",
  greet: function() {
    setTimeout(() => {
      console.log(`Hello, ${this.name}`);
    }, 1000);
  }
};
obj.greet(); // Hello, Alice
```

Explanation:

- Arrow functions don't bind their own this
- They inherit this from enclosing scope
- Demonstrates lexical this binding

Problem 178: Event Loop

Interview Frequency:  High (Common in mid-level interviews)

Question: Explain the event loop with an example.

Solution:

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("End");

// Output:
// Start
// End
// Promise
// Timeout
```

Explanation:

- Synchronous code runs first

- Then microtasks (Promises)
- Finally macrotasks (setTimeout)
- Shows JavaScript's asynchronous execution model

Problem 179: Custom Bind

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Implement a custom bind method.

Solution:

```
Function.prototype.myBind = function(context, ...args) {  
  const fn = this;  
  return function(...newArgs) {  
    return fn.apply(context, [...args, ...newArgs]);  
  }  
}  
  
function greet(greeting, name) {  
  return `${greeting}, ${name}`;  
}  
  
const greetHello = greet.myBind(null, "Hello");  
console.log(greetHello("John")); // Hello, John
```

Explanation:

- Creates a new function with preset context
- Allows partial application of arguments
- Demonstrates function context binding

Problem 180: Null vs Undefined

Interview Frequency: 🟡 Medium (Common in junior interviews)

Question: Explain the difference between null and undefined.

Solution:

```
let a;  
console.log(a); // undefined  
  
let b = null;  
console.log(b); // null
```

Explanation:

- undefined means variable declared but not assigned
- null is an intentional assignment of "no value"
- Shows JavaScript's type system nuances

Problem 181: This Binding

Interview Frequency:  High (Common in mid-level interviews)

Question: Explain implicit vs explicit binding with examples.

Solution:

```
// Implicit Binding  
const obj = {  
  name: "Tom",  
  greet() {  
    console.log("Hi " + this.name);  
  }  
};  
obj.greet(); // "Hi Tom"  
  
// Explicit Binding  
const person = { name: "Jerry" };  
function sayHi() {  
  console.log("Hi " + this.name);  
}  
sayHi.call(person); // "Hi Jerry"
```

Explanation:

- Implicit binding: Called via object (dot notation).
- Explicit binding: Use call, apply, or bind.
- Shows different ways to set this context

Problem 182: Async/Await Execution

Interview Frequency:  High (Common in mid-level interviews)

Question: What's the output of this async/await example?

Solution:

```
async function test() {  
  console.log("Start");  
  await Promise.resolve();  
  console.log("After await");  
}  
test();  
console.log("Outside");
```


Output:

```
Start  
Outside  
After await
```

Explanation:

- await pauses test()'s execution
- Allows outside code to run
- Demonstrates asynchronous execution order

Problem 183: Await Non-Promise

Interview Frequency:  Medium (Common in mid-level interviews)

Question: What happens if you await a non-promise value?

Solution:

```
async function run() {  
  const res = await 42;  
  console.log(res); // 42  
}  
run();
```

Explanation:

- JavaScript automatically wraps non-promise values in Promise.resolve
- Makes await work with any value
- Simplifies async code

Problem 184: Promisify Function

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Turn a callback-based function into a promise.

Solution:

```
function doAsync(cb) {
  setTimeout(() => cb(null, "done"), 500);
}

function promisify(fn) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      fn(...args, (err, data) => {
        if (err) reject(err);
        else resolve(data);
      });
    });
  };
}

const pDoAsync = promisify(doAsync);
pDoAsync().then(console.log); // "done"
```

Explanation:

- Promisification converts callback-style APIs into promise-based
- Handles error-first callbacks
- Makes async code more manageable

Problem 185: Sequential Promise Chain

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Convert an array of promises to a sequential chain.

Solution:

```
const funcs = [
  () => Promise.resolve(1),
  () => Promise.resolve(2),
  () => Promise.resolve(3),
];

funcs.reduce((p, fn) => p.then(fn).then(console.log), Promise.resolve());
```

Explanation:

- Chaining is done with .reduce()
- Waits for each promise in order
- Ensures sequential execution

Problem 186: Once Utility

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make a function run only once.

Solution:

```
function once(fn) {
  let called = false;
  let result;
  return function (...args) {
    if (!called) {
      called = true;
      result = fn.apply(this, args);
    }
    return result;
  };
}
```

Explanation:

- Caches the first call result
- Blocks further execution

- Useful for initialization code

Problem 187: Object Freeze

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Use `Object.freeze()` to make object immutable.

Solution:

```
const obj = Object.freeze({ name: "Alex" });
obj.name = "Bob"; // Does nothing in strict mode
console.log(obj.name); // "Alex"
```

Explanation:

- `Object.freeze()` locks the object
- Prevents modification of properties
- Creates shallow immutability

Problem 188: Deep Freeze

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Make an object immutable deeply.

Solution:

```
function deepFreeze(obj) {
  Object.keys(obj).forEach((key) => {
    if (typeof obj[key] === "object") deepFreeze(obj[key]);
  });
  return Object.freeze(obj);
}
```

Explanation:

- Recursively freezes every nested object
- Creates complete immutability
- Handles nested structures

Problem 189: Arrow Function This

Interview Frequency: 🔴 High (Common in mid-level interviews)

Question: How do arrow functions bind this?

Solution:

```
const obj = {
  name: "Sam",
  arrow: () => console.log(this.name),
  regular() {
    console.log(this.name);
  }
};
obj.arrow();    // undefined
obj.regular();  // "Sam"
```

Explanation:

- Arrow functions don't have their own this
- They inherit from the parent scope
- Regular functions create new this context

Problem 190: Empty Object Check

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Check if an object is empty.

Solution:

```
function isEmpty(obj) {
  return Object.keys(obj).length === 0;
}
```

Explanation:

- Object.keys() returns an array of own properties
- Empty array means no properties
- Simple way to check object emptiness

Problem 191: Rest and Spread

Interview Frequency:  High (Common in mid-level interviews)

Question: Explain rest and spread operators.

Solution:


```
// Rest
function sum(...args) {
  return args.reduce((a, b) => a + b);
}

// Spread
const arr = [1, 2];
const newArr = [...arr, 3]; // [1, 2, 3]
```

Explanation:

- ... can collect into array (rest)
- ... can unpack arrays/objects (spread)
- Versatile operator with multiple uses

Problem 192: Object Spread Clone

Interview Frequency:  Medium (Common in mid-level interviews)

Question: How to clone an object with spread?

Solution:

```
const original = { a: 1, b: 2 };
const copy = { ...original };
```

Explanation:

- Creates a shallow copy
- Copies own enumerable properties
- Simple way to clone objects

Problem 193: Object Merging

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: How to merge multiple objects?

Solution:

```
const a = { x: 1 };
const b = { y: 2 };
const merged = { ...a, ...b };
```

Explanation:

- Later properties override earlier ones
- Creates new object with combined properties
- Order matters for overrides

Problem 194: Memoization

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Create a memoization utility.

Solution:

```
function memoize(fn) {
  const cache = {};
  return function (n) {
    if (n in cache) return cache[n];
    return (cache[n] = fn(n));
  };
}
```

Explanation:

- Speeds up repeated calls
- Caches inputs/outputs
- Useful for expensive calculations

Problem 195: Prototype Chain

Interview Frequency: 🔴 High (Common in mid-level interviews)

Question: What is prototype chain in JS?

Solution:

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

Explanation:

- Objects inherit from **Prototype** chain
- Functions, arrays, etc., link to base prototypes
- Forms basis of inheritance

Problem 196: Prototype Method

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Add a method to all arrays using prototype.

Solution:

```
Array.prototype.last = function () {  
    return this[this.length - 1];  
};  
console.log([1, 2, 3].last()); // 3
```

Explanation:

- Prototype lets you add reusable methods
- Available to all instances
- Modifies built-in objects

Problem 197: Event Loop

Interview Frequency: 🔴 High (Common in mid-level interviews)

Question: What is the event loop in JavaScript?

Solution:

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
// Output: 1, 4, 3, 2
```

Explanation:

- Manages execution of sync and async tasks
- Tasks from call stack, microtask queue, macrotask queue
- Ensures non-blocking execution

Problem 198: Microtask vs Macrotask

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Difference between microtask and macrotask?

Solution:

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

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

Explanation:

- Microtask: Promises, queueMicrotask
- Macrotask: setTimeout, setInterval, UI events
- Microtasks run before next render or macrotasks

Problem 199: Queue Microtask

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Use queueMicrotask() with an example.

Solution:

```
queueMicrotask(() => console.log("microtask"));
console.log("sync");
```

Output:

```
sync
microtask
```

Explanation:

- Ensures callback runs right after current script
- Higher priority than setTimeout
- Useful for precise timing

Problem 200: Event Emitter

Interview Frequency: 🟡 Medium (Common in mid-level interviews)

Question: Write a custom event emitter.

Solution:

```
class Emitter {
  constructor() {
    this.events = {};
  }

  on(event, cb) {
    (this.events[event] || []).push(cb);
  }

  emit(event, ...args) {
    (this.events[event] || []).forEach(cb => cb(...args));
  }
}

// Usage
const emitter = new Emitter();
emitter.on("say", msg => console.log(msg));
emitter.emit("say", "Hello!"); // "Hello!"
```

Explanation:

- Implements pub/sub pattern
- `on()` to listen and `emit()` to trigger events
- Useful for decoupled communication

Problem 201: Function Declaration vs Expression

Interview Frequency:  High

Question: Explain the difference between function declarations and function expressions in JavaScript.

Solution:

```
// Function Declaration - hoisted
function greet() {
  return "Hello";
}

// Function Expression - not hoisted
const sayHi = function () {
  return "Hi";
};
```

Explanation:

- Function declarations are hoisted entirely
- Function expressions are hoisted as variables (undefined)
- Declarations can be used before they appear in the code
- Expressions must be defined before use

Problem 202: Default Parameters

Interview Frequency:  High

Question: Explain how default parameters work in JavaScript functions.

Solution:

```
function multiply(a, b = 2) {  
  return a * b;  
}  
  
console.log(multiply(3)); // 6
```

Explanation:

- Default parameters provide fallback values
- They are only used when the parameter is undefined
- They can be any expression, including function calls
- They are evaluated at call time

Problem 203: Array and Object Destructuring

Interview Frequency:  High

Question: Demonstrate how to destructure arrays and objects in JavaScript.

Solution:

```
const [x, y] = [1, 2];  
const { name, age } = { name: "Alice", age: 30 };
```

Explanation:

- Destructuring provides a clean way to extract values
- Array destructuring uses position
- Object destructuring uses property names
- Can provide default values with =

Problem 204: Value Swapping with Destructuring

Interview Frequency:  Medium

Question: Show how to swap two values using destructuring.

Solution:


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

Explanation:

- Destructuring allows elegant variable swaps
- No temporary variable needed
- Works with any number of variables
- Maintains reference integrity

Problem 205: Array Flattening

Interview Frequency: 🟡 Medium

Question: How to flatten a nested array in JavaScript?

Solution:

```
const nested = [1, [2, [3, 4]]];
console.log(nested.flat(2)); // [1, 2, 3, 4]
```

Explanation:

- flat() method flattens arrays up to specified depth
- Default depth is 1
- Returns new array, doesn't modify original
- Can handle any level of nesting

Problem 206: Arguments to Array

Interview Frequency: 🟡 Medium

Question: Convert function arguments to an array.

Solution:

```
function list() {
  return Array.from(arguments);
}
console.log(list(1, 2, 3)); // [1, 2, 3]
```

Explanation:

- arguments is an array-like object
- Array.from() converts array-like objects to arrays
- More modern than [...arguments]
- Preserves all arguments including undefined

Problem 207: Equality Operators

Interview Frequency:  High

Question: Explain the difference between == and === operators.

Solution:

```
console.log(2 == "2"); // true
console.log(2 === "2"); // false
```

Explanation:

- == performs type coercion
- === enforces strict type match
- === is generally preferred
- == can lead to unexpected results

Problem 208: Type Checking Methods

Interview Frequency:  High

Question: Compare typeof, instanceof, and Array.isArray() for type checking.

Solution:

```
typeof [] // "object"
[] instanceof Array // true
Array.isArray([]) // true
```

Explanation:

- typeof is not reliable for arrays
- instanceof checks prototype chain
- Array.isArray() is most reliable for arrays

- Each has specific use cases

Problem 209: NaN Checking

Interview Frequency: 🟡 Medium

Question: How to properly check for NaN values?

Solution:

```
isNaN("hello")           // true
Number.isNaN("hello")    // false
```

Explanation:

- isNaN() tries to convert to number first
- Number.isNaN() is more precise
- NaN is the only value not equal to itself
- Use Number.isNaN() for reliable checks

Problem 210: Range Function

Interview Frequency: 🟡 Medium

Question: Create a function that generates a range of numbers.

Solution:

```
function range(start, end) {
  return Array.from({ length: end - start + 1 }, (_, i) => i + start);
}
console.log(range(3, 7)); // [3, 4, 5, 6, 7]
```

Explanation:

- Uses Array.from() with mapping function
- Generates sequential numbers
- Inclusive of both start and end
- Efficient for small ranges

Problem 211: Event Delegation

Interview Frequency: 🔴 High

Question: Explain and implement event delegation.

Solution:

```
document.body.addEventListener("click", function (e) {  
  if (e.target.matches(".btn")) {  
    console.log("Button clicked");  
  }  
});
```

Explanation:

- Parent handles events from children
- Uses event bubbling
- More efficient than multiple listeners
- Dynamic elements work automatically

Problem 212: Debounce Function

Interview Frequency: 🔴 High

Question: Implement a debounce function.

Solution:

```
function debounce(fn, delay) {  
  let timer;  
  return (...args) => {  
    clearTimeout(timer);  
    timer = setTimeout(() => fn(...args), delay);  
  };  
}
```

Explanation:

- Delays execution until pause in calls
- Clears previous timer on new call
- Useful for search inputs
- Prevents excessive function calls

Problem 213: Throttle Function

Interview Frequency:  High

Question: Implement a throttle function.

Solution:

```
function throttle(fn, limit) {  
  let ready = true;  
  return (...args) => {  
    if (!ready) return;  
    ready = false;  
    fn(...args);  
    setTimeout(() => ready = true, limit);  
  };  
}
```

Explanation:

- Limits function execution rate
- Ensures minimum time between calls
- Useful for scroll events
- Maintains function context

Problem 214: Call Stack Overflow

Interview Frequency:  Medium

Question: Demonstrate and explain call stack overflow.

Solution:

```
function recurse() {  
  return recurse();  
}  
  
recurse(); // RangeError: Maximum call stack size exceeded
```

Explanation:

- Infinite recursion without base case exhausts stack.

Problem 215: Deep Clone with JSON

Interview Frequency: 🟡 Medium

Question: Clone an object deeply using JSON.

Solution:

```
const original = { a: 1, b: { c: 2 } };
const clone = JSON.parse(JSON.stringify(original));
```

Explanation:

- Simple but limited deep cloning
- Works for plain objects
- Loses functions and Dates
- Circular references fail

Problem 216: Logical Operators

Interview Frequency: 🔴 High

Question: Compare logical OR (||) and nullish coalescing (??) operators.

Solution:

```
const a = null;
console.log(a || "default"); // "default"
console.log(a ?? "default"); // "default"
```

Explanation:

- || returns right-hand if left is falsy
- ?? returns right-hand only if left is null/undefined
- ?? is more precise for null/undefined checks
- || considers all falsy values

Problem 217: Temporal Dead Zone

Interview Frequency: 🔴 High

Question: Explain the temporal dead zone (TDZ).

Solution:

```
console.log(x); // ReferenceError  
let x = 10;
```

Explanation:

- Variables in TDZ from scope start to declaration
- Applies to let and const
- Prevents accessing before declaration
- Helps catch potential bugs

Problem 218: String to Number

Interview Frequency: 🟡 Medium

Question: Convert string to number safely.

Solution:

```
const str = "123";  
const num = Number(str); // 123
```

Explanation:

- Number() is preferred over parseInt()
- Handles decimals correctly
- Returns NaN for invalid numbers
- More predictable than unary +

Problem 219: Remove Duplicates

Interview Frequency: 🟡 Medium

Question: Remove duplicates from an array.

Solution:

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

Explanation:

- Set stores unique values
- Spread operator converts back to array
- Maintains order
- Works with any type of values

Problem 220: Bitwise NOT for Index Check

Interview Frequency: 🟡 Medium

Question: Use bitwise NOT to check array index existence.

Solution:

```
const arr = [1, 2, 3];  
console.log(~arr.indexOf(2)); // -2 → truthy
```

Explanation:

- $\sim x$ converts -1 (not found) to 0 (falsy), anything else → truthy.

Problem 221: Null vs Undefined

Interview Frequency: 🔴 High

Question: Explain the difference between null and undefined.

Solution:

```
let a;  
let b = null;  
  
console.log(a); // undefined  
console.log(b); // null
```

Explanation:

- undefined: variable declared but not assigned
- null: explicitly assigned to indicate "no value"
- undefined is a type
- null is an object

Problem 222: Number to String

Interview Frequency: 🟡 Medium

Question: Convert a number to a string.

Solution:

```
const num = 42;  
const str1 = String(num);  
const str2 = num.toString();
```

Explanation:

- String() works with any value
- toString() requires value to exist
- Both produce same result
- String() is more reliable

Problem 223: String to Number

Interview Frequency: 🟡 Medium

Question: Convert a string to a number.

Solution:

```
const str = "123";  
const num = +str; // 123
```

Explanation:

- Unary + is quick conversion
- Number() is more explicit
- parseInt() for integers
- parseFloat() for decimals

Problem 224: Void Operator

Interview Frequency: 🟡 Medium

Question: Explain the use of void 0.

Solution:

```
console.log(void 0); // undefined
```

Explanation:

- Safely returns undefined
- Useful in old JS environments
- Prevents undefined reassignment
- Modern JS rarely needs it

Problem 225: Map Polyfill

Interview Frequency: 🟡 Medium

Question: Create a polyfill for `Array.prototype.map`.

Solution:

```
Array.prototype.myMap = function (cb) {  
  const res = [];  
  for (let i = 0; i < this.length; i++) {  
    res.push(cb(this[i], i, this));  
  }  
  return res;  
};  
  
console.log([1, 2, 3].myMap(x => x * 2)); // [2, 4, 6]
```

Explanation:

- Implements basic map logic manually.
- Preserves array length
- Passes index and array to callback
- Returns new array

Problem 226: Filter Polyfill

Interview Frequency: 🟡 Medium

Question: Write a polyfill for `Array.prototype.filter`.

Solution:

```
Array.prototype.myFilter = function (cb) {  
  const result = [];  
  for (let i = 0; i < this.length; i++) {  
    if (cb(this[i], i, this)) result.push(this[i]);  
  }  
  return result;  
};
```

Explanation:

- Filters based on callback truthiness
- Preserves original array
- Passes index and array to callback
- Returns new array

Problem 227: Reduce Polyfill

Interview Frequency: 🟡 Medium

Question: Write a polyfill for `Array.prototype.reduce`.

Solution:

```
Array.prototype.myReduce = function (cb, init) {  
  let acc = init ?? this[0];  
  let start = init ? 0 : 1;  
  
  for (let i = start; i < this.length; i++) {  
    acc = cb(acc, this[i], i, this);  
  }  
  return acc;  
};
```

Explanation:

- Handles initial value option
- Accumulates result
- Passes index and array to callback
- Returns single value

Problem 228: Array Freezing

Interview Frequency: 🟡 Medium

Question: How to freeze an array and prevent mutation?

Solution:

```
const arr = Object.freeze([1, 2, 3]);  
arr.push(4); // TypeError in strict mode
```

Explanation:

- Object.freeze() makes immutable
- Prevents adding/removing elements
- Prevents modifying elements
- Shallow freeze only

Problem 229: Symbol Usage

Interview Frequency: 🟡 Medium

Question: Explain the use of Symbol.

Solution:

```
const id = Symbol("id");  
const user = { [id]: 123 };
```

Explanation:

- Creates unique object keys
- Not enumerable by default
- Useful for private properties
- Description is optional

Problem 230: Counter with Closures

Interview Frequency: 🔴 High

Question: Create a counter using closures.

Solution:

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

Explanation:

- Closure maintains private state
- Count persists between calls
- Encapsulates implementation
- Provides clean interface

Problem 231: Toggle with Closures

Interview Frequency: 🟡 Medium

Question: Implement a toggle function using closures.

Solution:

```
function createToggle() {  
  let on = false;  
  return () => (on = !on);  
}  
  
const toggle = createToggle();  
console.log(toggle()); // true  
console.log(toggle()); // false
```

Explanation:

- Closure holds internal state
- Toggles boolean value
- Returns current state
- Maintains privacy

Problem 232: Palindrome Check

Interview Frequency: 🟡 Medium

Question: Check if a string is a palindrome.

Solution:

```
function isPalindrome(str) {  
  const reversed = str.split('').reverse().join('');  
  return str === reversed;  
}
```

Explanation:

- Reverses string and compares
- Case-sensitive
- Ignores spaces
- Simple implementation

Problem 233: Optional Chaining

Interview Frequency: 🔴 High

Question: How to use optional chaining?

Solution:

```
const user = { profile: { name: "Tom" } };  
console.log(user?.profile?.name); // "Tom"
```

Explanation:

- Prevents null/undefined errors
- Short-circuits on first null
- Safe property access
- Modern JavaScript feature

Problem 234: Nullish Coalescing

Interview Frequency: 🔴 High

Question: How to use nullish coalescing operator?

Solution:

```
const val = null;  
const res = val ?? "default";
```

Explanation:

- Returns right-hand if left is null/undefined
- More precise than ||
- Ignores other falsy values
- Modern JavaScript feature

Problem 235: Deep Clone Recursive

Interview Frequency:  High

Question: Deep clone an object with recursion.

Solution:

```
function deepClone(obj) {  
  if (obj === null || typeof obj !== "object") return obj;  
  const cloned = Array.isArray(obj) ? [] : {};  
  for (const key in obj) {  
    cloned[key] = deepClone(obj[key]);  
  }  
  return cloned;  
}
```

Explanation:

- Handles nested objects/arrays
- Preserves structure
- Recursive approach
- Handles primitive values

Problem 236: Function Binding

Interview Frequency:  High

Question: Explain Function.prototype.bind.

Solution:

```
function greet() {  
  console.log(this.name);  
}  
const person = { name: "Alex" };  
const boundGreet = greet.bind(person);  
boundGreet(); // "Alex"
```

Explanation:

- Sets permanent this context
- Returns new function
- Can partially apply arguments
- Useful for event handlers

Problem 237: Promise Timeout

Interview Frequency: 🟡 Medium

Question: Add timeout to a Promise.

Solution:

```
function timeout(promise, ms) {  
  const timer = new Promise((_, reject) =>  
    setTimeout(() => reject("Timeout"), ms)  
  );  
  return Promise.race([promise, timer]);  
}
```

Explanation:

- Uses Promise.race
- Rejects after timeout
- Preserves original promise
- Handles cleanup

Problem 238: Promise Retry

Interview Frequency: 🟡 Medium

Question: Implement retry logic for a Promise.

Solution:

```
async function retry(fn, attempts) {
  for (let i = 0; i < attempts; i++) {
    try {
      return await fn();
    } catch (e) {
      if (i === attempts - 1) throw e;
    }
  }
}
```

Explanation:

- Retries failed promises
- Configurable attempts
- Preserves error handling
- Async/await syntax

Problem 239: Promise Delay

Interview Frequency: 🟡 Medium

Question: Create delay using Promises.

Solution:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

Explanation:

- Returns promise that resolves after delay
- Useful for async operations
- Can be awaited
- Simple implementation

Problem 240: Object.create(null)

Interview Frequency: 🟡 Medium

Question: What is the purpose of Object.create(null)?

Solution:

```
const obj = Object.create(null);
console.log(obj.toString()); // undefined
```

Explanation:

- Creates object without prototype
- No inherited methods
- True hash map
- Better performance

Problem 241: Remove Falsy Values

Interview Frequency: 🟡 Medium

Question: Remove falsy values from an array.

Solution:

```
const values = [0, false, "", null, 1, "hello"];
const truthy = values.filter(Boolean);
```

Explanation:

- Boolean constructor as filter
- Removes all falsy values
- Preserves truthy values
- Simple one-liner

Problem 242: Element Counting

Interview Frequency: 🟡 Medium

Question: Count occurrences of elements in array.

Solution:

```
const items = ['a', 'b', 'a'];
const count = items.reduce((acc, cur) => {
  acc[cur] = (acc[cur] || 0) + 1;
  return acc;
}, {});
```

Explanation:

- Uses reduce to build frequency map
- Handles any type of elements
- Returns object with counts
- Efficient for large arrays

Problem 243: Currying Function

Interview Frequency:  High

Question: Implement a curry function.

Solution:

```
function curry(fn) {
  return function curried(...args) {
    return args.length >= fn.length
      ? fn(...args)
      : (...next) => curried(...args, ...next);
  };
}
```

Explanation:

- Breaks function into smaller calls
- Preserves function context
- Allows partial application
- Functional programming concept

Problem 244: Prototype Inheritance

Interview Frequency:  High

Question: Explain prototype inheritance with example.

Solution:

```
function Animal(name) {  
  this.name = name;  
}  
Animal.prototype.sayHi = function () {  
  return `Hi, I'm ${this.name}`;  
};  
  
const dog = new Animal("Rex");  
console.log(dog.sayHi()); // Hi, I'm Rex
```

Explanation:

- Prototypes enable inheritance
- Methods shared across instances
- Memory efficient
- Core JavaScript concept

Problem 245: Array Equality

Interview Frequency: 🟡 Medium

Question: Compare two arrays for equality.

Solution:

```
function arraysEqual(a, b) {  
  return a.length === b.length && a.every((v, i) => v === b[i]);  
}
```

Explanation:

- Checks length and elements
- Order matters
- Simple comparison
- No deep comparison

Problem 246: Unique Objects

Interview Frequency: 🟡 Medium

Question: Get unique values from array of objects by key.

Solution:

```
const people = [{ name: "Alice" }, { name: "Bob" }, { name: "Alice" }];
const unique = [...new Map(people.map(p => [p.name, p])).values()];
```

Explanation:

- Uses Map for uniqueness
- Preserves object references
- Efficient lookup
- Maintains order

Problem 247: Object Entries

Interview Frequency: 🟡 Medium

Question: Use Object.entries() for looping.

Solution:

```
const user = { name: "Sam", age: 30 };
for (const [key, value] of Object.entries(user)) {
  console.log(`${key}: ${value}`);
}
```

Explanation:

- Destructures key-value pairs
- Iterates over properties
- Modern JavaScript feature
- Clean syntax

Problem 248: Nested Value Access

Interview Frequency: 🔴 High

Question: Get nested value with fallback.

Solution:

```
const obj = { a: { b: { c: 5 } } };  
const val = obj?.a?.b?.c ?? "Not Found";
```

Explanation:

- Combines optional chaining
- Uses nullish coalescing
- Safe property access
- Modern JavaScript features

Problem 249: Deep Merge

Interview Frequency:  High

Question: Merge deeply nested objects.

Solution:

```
function deepMerge(target, source) {  
  for (let key in source) {  
    if (source[key] instanceof Object) {  
      target[key] = deepMerge(target[key] ?? {}, source[key]);  
    } else {  
      target[key] = source[key];  
    }  
  }  
  return target;  
}
```

Explanation:

- Recursively merges objects
- Preserves nested structure
- Handles arrays and objects
- Modifies target object

Problem 250: JavaScript Single Thread

Interview Frequency:  High

Question: Why is JavaScript single-threaded and what's the impact?

Explanation:

- JS runs in a single thread
- Only one line runs at a time
- Uses event loop for concurrency
- Async patterns for responsiveness
- Avoids race conditions
- Requires careful async handling