Snippets from Real Javascript Interviews

These JavaScript snippets are inspired by **real interview questions** and **actual tasks from my job**. They're crafted to give you **hands-on experience** with advanced JavaScript concepts that companies truly care about.

I'm a frontend developer who went through **20+ interviews** — and I compiled everything companies were asking. Each snippet is derived from real-world scenarios. If you're familiar with all these topics, you can thrive at any company, on any project — and feel confident.

> ✅ **Pro tip:** Copy any snippet, run it, and step through it with a debugger to understand what's going on under the hood.

🔍 Do you know the output (not guess)?

What will this print?

```
const promise1 = Promise.resolve();
const promise2 = Promise.resolve();

promise1.then(() => console.log(1)).then(() => console.log(2));
promise2.then(() => console.log(3)).then(() => console.log(4));
```

And how about this?

```
console.log("1");

setTimeout(function () {
  console.log("2");

  Promise.resolve().then(function () {
    console.log("3");
  });
}, 0);

Promise.resolve().then(function () {
  console.log("4");

  setTimeout(function () {
    console.log("5");
  }, 0);
});

requestAnimationFrame(function () {
  console.log("7");
});

console.log("6");
```

Understand the Event Loop (page 52)

🔥 Pass your next interview — let's dive in 👇

Advanced JavaScript Code Examples

About Project

💡 Leetcode problems can sometimes feel disconnected from real-world development (with a few valuable exceptions: **graph algorithms** and **system design**).

What truly matters? Understanding how **JavaScript works under the hood**. If you don't understand JavaScript, Leetcode is just a waste of time — you're solving abstract puzzles without mastering the real engine behind your code. Instead, build your skills with **Advanced JavaScript Patterns**, carefully selected code snippets, and concepts that companies *actually* expect in interviews.

Dive in, practice, and develop real confidence.

> 📌 This resource complements foundational learning platforms like MDN Web Docs, JavaScript.info, and react.dev. If you're just starting out, begin there to build strong fundamentals.

Content

🧭 Graph Algorithms

Fundamental graph traversal logic used in system design and coding challenges.

- `Graph Traversal` (5) – **BFS**, **DFS**, adjacency lists, graph algorithms.

📦 JS Native

Deep dive into native JavaScript methods and behavior. Each section includes polyfills (custom implementation), explanations and all possible use cases.

- `Map` (8) – Map, WeakMap, Map.groupBy.
- `Mutating Methods` (12) – concat, fill, pop, push, reverse, shift, unshift, splice.
- `Promises` (18) – Promise.all, race, any, allSettled.
- `Read methods` (21) – at, every, find, findIndex, findLast, includes, indexOf, some.
- `Set and Object` (26) – Set, Object.groupBy, Object.assign, Array.entries.
- `Transform Methods` (30) – filter, flat, flatMap, join, map, reduce, slice.

📖 Learn JS and Mdn

Essential learning topics pulled from MDN, Learn JS, interviews and my web dev job.

- `Advanced Patterns` (37) – Memoize, Mixins, Promise Chaining, Promise Error Handling, Synchronous vs Asynchronous Errors in Promises, Proxy, Range Iterator, Shooters: Closures & Lexical Scope.
- `Delay, Throttling and Debounce` (43) – Delay Decorator, Delaying function execution, Output Every Second, setInterval, setTimeout, throttling and debounce decorator.

- `Core Concepts` (47) – Understanding this, Loop Behavior: Pre- / Post-Increment, Object and Map Conversion, bind, Async Generators, Fibonacci (Iterative Approach, Recursive Approach).
- `Event Loop` (52) – Event Loop edge cases — the **#1 most asked JavaScript interview topic**.
- `This and Object Methods` (59) – Nuances of `this` across different method contexts — the **#2 most asked JavaScript interview topic**.

Lodash

Utility functions from lodash.

- `Chunk, Compact, Partition` (66) – chunk, compact, partition.
- `Difference, Intersection, Union` (68) – difference, differenceBy, intersection, union.
- `Object Utilities` (71) – keyBy, omit, orderBy, pick, curry.

React

React specific topics.

- `React Rendering` (75) – usePrevious, children.
- `State Patterns` (77) – Normalized State, useReducer (custom implementation).

System Design

System design concepts often seen in real interviews.

- `Redux and Twitter` (82) – Redux reducer pattern, most used pattern for state management in react. Twitter feed pattern.
- `Queue and Stack` (85) – Stacks and queues are core to how JavaScript works under the hood — and they show up in nearly every interview.

Theory

High-level principles and mental models.

- `Composition vs Inheritance` (88) - Composition, Inheritance and Function Stack.
- `SOLID in React` (91) - SOLID principles in React.
- `Type Conversions` (97) - Explore the magic behind JavaScript **type coercion** and equality checks.

Various

Advanced utility functions and transformation logic — real-world inspired.

- `Call, bind, apply` (100) – Understanding `call`, `bind`, and `apply` is essential for mastering how JavaScript handles function context (`this`). **Often asked on interviews**.
- `Dictionary of Nested` (105) – Deeply nested array structure (e.g., categories → subcategories → items) into a dictionary format for **constant-time lookup** by `id`.
- `Group List by Quarters` (110) – Groups monthly financial data by calendar quarters.
- `Transform Lists and Trees` (114) – List to tree, tree to list, flat nested items.
- `Promises Closures Event Loop` (119) – Async behavior, state retention, and timing execution in complex JavaScript code.

- `Sorting` (124) – bubble sort, insertion sort, selection sort.
- `Various Utilities` (126) – clsx, contains duplicate, deep clone, filter by related property, filterMap, innerJoin, Reducer pattern with actions, Retry with exponential backoff, Shuffle (Fisher-Yates Algorithm), Topological Sort.

About Me

I'm a frontend developer based in Russia, currently working at Sellego. I specialize in building modern web apps using **React** on the frontend and **Node.js** on the backend.

Github

- this project
- react effector mui
- react redux mui
- react mobx mui

Contacts

- @johnkucharsky – telegram
- johnkucharskyfirst@gmail.com

Graph Traversal Algorithms in JavaScript

Learn how to traverse graphs in JavaScript using Depth-First Search (DFS) and Breadth-First Search (BFS). These examples show how to implement both directed and undirected graph traversal using adjacency lists and recursive or iterative logic. Ideal for interview prep and understanding graph algorithms from the ground up.

Has Path in Directed Graphs (DFS & BFS)

Determines whether a path exists between two nodes in a directed graph. Includes both **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** implementations.

The graph is represented as an **adjacency list**.

```
type Graph = Record<string, string[]>;

// Depth-First Search
const hasPath = (
  graph: Graph,
  src: string,
  dst: string,
): boolean => {
  if (src === dst) return true;

  for (const neighbor of graph[src]) {
    if (hasPath(graph, neighbor, dst)) {
      return true;
    }
  }

  return false;
};

// Breadth-First Search (uncomment to use)
/*
const hasPath = (
  graph: Graph,
  src: string,
  dst: string,
): boolean => {
  const queue: string[] = [src];

  while (queue.length) {
    const current = queue.shift();
    if (current === dst) return true;

    if (current && graph[current]) {
      for (const neighbor of graph[current]) {
        queue.push(neighbor);
      }
    }
  }

  return false;
};
*/

const adjacentList = {
```

```
  f: ["g", "i"],
  g: ["h"],
  h: [],
  i: ["g", "k"],
  j: ["i"],
  k: [],
};

console.log(hasPath(adjacentList, "f", "k")); // true
console.log(hasPath(adjacentList, "f", "j")); // false

console.log(
  hasPath(
    {
      v: ["x", "w"],
      w: [],
      x: [],
      y: ["z"],
      z: [],
    },
    "v",
    "z",
  ),
); // false
```

Path Existence in Undirected Graphs (DFS with Visited Set)

Checks if a path exists between two nodes in an **undirected graph**, using **DFS** with a `visited` set to avoid cycles.

The graph is built from an `edges` list using an **adjacency list** structure.

```
const undirectedPath = (
  edges: [string, string][],
  nodeA: string,
  nodeB: string,
): boolean => {
  const graph = buildGraph(edges);
  return hasPath(graph, nodeA, nodeB, new Set());
};

const buildGraph = (edges: [string, string][]) => {
  const graph: Record<string, string[]> = {};

  for (const [a, b] of edges) {
    if (!(a in graph)) graph[a] = [];
    if (!(b in graph)) graph[b] = [];
    graph[a].push(b);
    graph[b].push(a);
  }

  return graph;
};

const hasPath = (
  graph: Record<string, string[]>,
  src: string,
  dst: string,
  visited: Set<string>,
): boolean => {
```

```
  if (src === dst) return true;
  if (visited.has(src)) return false;
  visited.add(src);

  for (const neighbor of graph[src]) {
    if (hasPath(graph, neighbor, dst, visited)) {
      return true;
    }
  }

  return false;
};

const edges: [string, string][] = [
  ["i", "j"],
  ["k", "i"],
  ["m", "k"],
  ["k", "l"],
  ["o", "n"],
];

console.log(undirectedPath(edges, "j", "m")); // true
console.log(undirectedPath(edges, "k", "o")); // false
```

JavaScript Maps, WeakMaps, and Map.groupBy()

Explore how `Map`, `WeakMap`, and `Map.groupBy()` work in JavaScript. Learn the differences between Maps and plain objects, how `WeakMap` handles memory safely, and how to group values using custom `groupBy()`.

Map

The `Map` object holds **key-value pairs** and remembers the original insertion order of the keys. Keys can be **any value**, including objects, functions, and `NaN`. Unlike objects, maps offer consistent key ordering and better performance for frequent additions and deletions.

Instance methods:

- `Map.prototype.clear()`
- `Map.prototype.delete()`
- `Map.prototype.entries()`
- `Map.prototype.forEach()`
- `Map.prototype.get()`
- `Map.prototype.has()`
- `Map.prototype.keys()`
- `Map.prototype.set()`
- `Map.prototype.values()`

Instance properties:

- `Map.prototype.size`

```
const map = new Map();
map.set("name", "Alice");
map.set("age", 25);
map.set("country", "USA");
console.log(map);
// Map(3) { 'name' => 'Alice', 'age' => 25, 'country' => 'USA' }

// Size
console.log(map.size); // 3

// Accessing values
console.log(map.get("name")); // Alice

// Checking for a key
console.log(map.has("age")); // true

// Removing a key
map.delete("country");
console.log(map);
// Map(2) { 'name' => 'Alice', 'age' => 25 }

map.set("city", "New York");
map.set("hobby", "Painting");

// Using for...of
```

```
console.log("Using for...of:");
for (const [key, value] of map) {
  console.log(key, ":", value);
}

// Getting all keys, values, or entries
console.log("Keys:", [...map.keys()]);
// Keys: [ 'name', 'age', 'city', 'hobby' ]
console.log("Values:", [...map.values()]);
// Values: [ 'Alice', 25, 'New York', 'Painting' ]
console.log("Entries:", [...map.entries()]);
// Entries: [
//     [ 'name', 'Alice' ],
//     [ 'age', 25 ],
//     [ 'city', 'New York' ],
//     [ 'hobby', 'Painting' ]
// ]

// Clearing the map
map.clear();
console.log(map); // Map(0) {}

// Creating a Map with an array of entries
const mapFromArray = new Map([
  ["fruit", "Apple"],
  ["color", "Red"],
  ["quantity", 10],
]);
console.log(mapFromArray);
// Map(3) { 'fruit' => 'Apple', 'color' => 'Red', 'quantity' => 10 }

// Converting a Map back to an object
const obj = Object.fromEntries(mapFromArray);
console.log(obj);
// { fruit: 'Apple', color: 'Red', quantity: 10 }

// Map vs Object comparison
const objExample = { a: 1, b: 2, c: 3 };
const mapExample = new Map(Object.entries(objExample));
console.log(mapExample);
// Map(3) { 'a' => 1, 'b' => 2, 'c' => 3 }

// Keys of different types
const map2 = new Map();
const keyObj = {};
const keyFunc = function () {};

map2.set("hello", "string value");
map2.set(keyObj, "obj value");
map2.set(keyFunc, "func value");
map2.set(NaN, "NaN value");

console.log(map2);
// Map(4) {
//   'hello' => 'string value',
//   {} => 'obj value',
//   [Function: keyFunc] => 'func value',
//   NaN => 'NaN value'
// }

console.log(map2.get("hello")); // string value
console.log(map2.get(keyObj)); // obj value
console.log(map2.get(keyFunc)); // func value
console.log(map2.get(NaN)); // NaN value
```

WeakMap

The `WeakMap` object is a special kind of map whose keys must be **objects**, and those keys are **weakly referenced** — meaning they do **not prevent garbage collection**.

This makes `WeakMap` useful for storing metadata or private data tied to object lifetimes. However, it does not support iteration or methods like `.keys()` or `.clear()`.

- `weakMap.set(key, value)`
- `weakMap.get(key)`
- `weakMap.delete(key)`
- `weakMap.has(key)`

If an object has lost all other references, then it is to be garbage-collected automatically. But technically it's not exactly specified when the cleanup happens

```
let john = { name: "John" };

const map = new Map();
map.set(john, "...");

john = null;
console.log(map.get(john)); // undefined
console.log(map.has(john)); // false
console.log(map.keys());
// [Map Iterator] { { name: 'John' } }

let doe = { name: "Doe" };
const weakMap = new WeakMap();
weakMap.set(doe, "...");

doe = null;
console.log(weakMap.get(doe)); // undefined
// weakMap.keys() not supported
```

Map.groupBy

The `Map.groupBy()` static method groups elements from an iterable based on the return value of a callback function. It returns a new `Map` where each key is the grouping criterion and each value is an array of items in that group.

This example provides a polyfill-style custom implementation of `Map.groupBy()` and demonstrates grouping numbers, objects by property, and even using objects as keys for advanced use cases.

```
// Map.groupBy isn't available yet
function groupBy(array, callback) {
  const map = new Map();

  for (const item of array) {
    const key = callback(item);
    const group = map.get(key) || [];
    group.push(item);
    map.set(key, group);
```

```
  }

  return map;
}

// Using Map.groupBy to group numbers by even and odd
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];

const groupedNumbers = groupBy(numbers, (num) =>
  num % 2 === 0 ? "even" : "odd",
);
console.log(groupedNumbers);
// Map(2) {
//   'odd' => [1, 3, 5, 7, 9],
//   'even' => [2, 4, 6, 8]
// }

// Grouping people by their age group
const people = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 25 },
  { name: "David", age: 35 },
];

const groupedByAge = groupBy(people, (person) => person.age);
console.log(groupedByAge);
// Map(3) {
//   25 => [ { name: 'Alice', age: 25 }, { name: 'Charlie', age: 25 } ],
//   30 => [ { name: 'Bob', age: 30 } ],
//   35 => [ { name: 'David', age: 35 } ]
// }

const inventory = [
  { name: "asparagus", type: "vegetables", quantity: 9 },
  { name: "bananas", type: "fruit", quantity: 5 },
  { name: "goat", type: "meat", quantity: 23 },
  { name: "cherries", type: "fruit", quantity: 12 },
  { name: "fish", type: "meat", quantity: 22 },
];

// Using objects as keys
const restock = { restock: true };
const sufficient = { restock: false };
const result = groupBy(inventory, ({ quantity }) =>
  quantity < 6 ? restock : sufficient,
);
console.log(result);
// Map(2) {
//     { restock: false } => [
//         { name: 'asparagus', type: 'vegetables', quantity: 9 },
//         { name: 'goat', type: 'meat', quantity: 23 },
//         { name: 'cherries', type: 'fruit', quantity: 12 },
//         { name: 'fish', type: 'meat', quantity: 22 }
//     ],
//         { restock: true } => [ { name: 'bananas', type: 'fruit', quantity: 5 } ]
// }
console.log(result.get(restock));
// [{ name: "bananas", type: "fruit", quantity: 5 }]
```

Custom Implementations for Mutating JavaScript Array Methods

These array methods modify the original array in-place. This page includes custom implementations of JavaScript mutating methods like `push()`, `pop()`, `splice()`, `fill()`, `reverse()`, and more — perfect for understanding how they work internally.

Array.concat

The `concat()` method merges two or more arrays and returns a **new array** without modifying the original. It also handles non-array values by appending them as-is. This custom implementation replicates native behavior by flattening one level of array input while preserving nested structures.

```
Array.prototype.myConcat = function (...arrays) {
  const result = [...this];

  for (const array of arrays) {
    if (Array.isArray(array)) {
      result.push(...array);
    } else {
      result.push(array);
    }
  }

  return result;
};

const arr = [1, 2, 3];
const arr2 = [4, 5, 6, [1]];
const arr3 = [7, 8, 9];
const concat = arr.myConcat(arr2, arr3, 10);
console.log(concat);
// [1, 2, 3, 4, 5, 6, [1], 7, 8, 9, 10]
```

Array.fill

The `fill()` method changes all elements in an array between the specified `start` and `end` indices to a **static value**, in place. It returns the **modified array** and is useful for quickly initializing or resetting arrays.

This example includes a custom implementation of `Array.prototype.fill()` with support for negative indices.

```
Array.prototype.customFill = function (value, start = 0, end = this.length) {
  if (start < 0) {
    start = this.length + start;
  }

  if (end < 0) {
    end = this.length + end;
  }
```

```
  for (let i = start; i < Math.min(end, this.length); i++) {
    this[i] = value;
  }

  return this;
};

const array1 = [1, 2, 3, 4];

// Fill with 0 from position 2 until position 4
console.log(array1.customFill(0, 2, 4));
// [1, 2, 0, 0]

// Fill with 5 from position 1
console.log(array1.customFill(5, 1));
// [1, 5, 5, 5]

console.log(array1.customFill(6));
// [6, 6, 6, 6]

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

Array.pop

The `pop()` method removes the **last element** from an array and returns it. It mutates the original array by reducing its length by one.

This example provides a custom implementation of `Array.prototype.pop()` that replicates native behavior.

```
Array.prototype.customPop = function () {
  const length = this.length;

  if (length === 0) {
    return undefined;
  }

  const lastElement = this[length - 1];
  this.length = length - 1;

  return lastElement;
};

const fruits = ["Apple", "Banana", "Cherry"];
const lastFruit = fruits.customPop();
console.log(lastFruit); // Cherry
console.log(fruits); // [ "Apple", "Banana" ]
```

Array.push

The `push()` method adds one or more elements to the **end of an array** and returns the **new length** of the array. It's a common and efficient way to grow an array in-place.

This example demonstrates a custom implementation of `Array.prototype.push()` that handles multiple arguments and mimics native behavior.

```
Array.prototype.customPush = function () {
  for (let i = 0; i < arguments.length; i++) {
    this[this.length] = arguments[i];
  }

  return this.length;
};

const animals = ["pigs", "goats"];

const count = animals.customPush("cows");
console.log(count); // 3

console.log(animals); // [ 'pigs', 'goats', 'cows' ]

animals.customPush("chickens", "cats");
console.log(animals);
// [ 'pigs', 'goats', 'cows', 'chickens', 'cats' ]
```

Array.reverse

The `reverse()` method reverses an array **in place** and returns the **modified array**. The first element becomes the last, and the last becomes the first, effectively reversing the array's order.

This custom implementation of `Array.prototype.reverse()` reverses the array elements by swapping them in place.

```
Array.prototype.customReverse = function () {
  const middle = Math.floor(this.length / 2);

  for (let i = 0; i < middle; i++) {
    const temp = this[i];
    this[i] = this[this.length - 1 - i];
    this[this.length - 1 - i] = temp;
  }

  return this;
};

const array1 = ["one", "two", "three", "four"];

const reversed = array1.customReverse();
console.log(reversed); // [ 'four', 'three', 'two', 'one' ]
// reverse changes the original array
console.log(array1); // [ 'four', 'three', 'two', 'one' ]
```

Array.shift

The `shift()` method removes the **first element** from an array and returns it, shifting the remaining elements down. This method mutates the array by reducing its length.

This custom implementation of `Array.prototype.shift()` manually shifts the array elements and returns the removed first element.

```
Array.prototype.customShift = function () {
  if (!this.length) return;

  const firstElement = this[0];

  for (let i = 0; i < this.length; i++) {
    this[i] = this[i + 1];
  }

  this.length -= 1;

  return firstElement;
};

const array1 = [1, 2, 3];

const firstElement = array1.customShift();

console.log(array1); // [2, 3]
console.log(firstElement); // 1
```

Array.unshift

The `unshift()` method adds one or more elements to the **beginning** of an array and returns the **new length** of the array.

In this custom implementation, the method first shifts all existing elements to the right to make space for the new elements. Then, it inserts the new elements at the beginning of the array and returns the updated length.

```
Array.prototype.unshift = function (...elements) {
  const originalLength = this.length;
  const totalLength = elements.length + originalLength;

  // Shift existing elements to the right
  for (let i = originalLength - 1; i >= 0; i--) {
    this[i + elements.length] = this[i];
  }

  // Add new elements at the beginning
  for (let i = 0; i < elements.length; i++) {
    this[i] = elements[i];
  }

  return totalLength; // Return the new length
};

const arr = [3, 4, 5];
arr.unshift(1, 2);
```

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

Array.splice

The `splice()` method changes the contents of an array by **removing or replacing** existing elements and/or adding new elements in place.

This custom implementation of `Array.prototype.splice()` handles negative indices, normalization of `deleteCount`, and array modification in place, while returning the removed elements.

```
Array.prototype.customSplice = function (
  startIndex,
  deleteCount,
  ...itemsToAdd
) {
  const length = this.length;

  // Handle negative indices
  startIndex =
    startIndex < 0
      ? Math.max(length + startIndex, 0)
      : Math.min(startIndex, length);

  // If deleteCount is undefined, remove all elements starting from startIndex
  if (deleteCount === undefined) {
    deleteCount = length - startIndex;
  } else {
    // Normalize deleteCount
    deleteCount = Math.max(0, Math.min(deleteCount, length - startIndex));
  }

  // Extract the array to be deleted
  const splicedItems = this.slice(startIndex, startIndex + deleteCount);

  // Create the resulting this by combining parts and items to add
  const remainingItems = [
    ...this.slice(0, startIndex),
    ...itemsToAdd,
    ...this.slice(startIndex + deleteCount),
  ];

  // Update the original array
  this.length = 0; // Clear the this
  for (let i = 0; i < remainingItems.length; i++) {
    this[i] = remainingItems[i];
  }

  // Return the deleted items
  return splicedItems;
};

// Remove 0 (zero) elements before index 2, and insert "drum"
const myFish = ["angel", "clown", "mandarin", "sturgeon"];
const removed = myFish.customSplice(2, 0, "drum");
console.log({ myFish, removed });
// myFish: [ 'angel', 'clown', 'drum', 'mandarin', 'sturgeon' ]
// removed: []
```

```
// Remove 0 (zero) elements before index 2, and insert "drum" and "guitar"
const myFish1 = ["angel", "clown", "mandarin", "sturgeon"];
const removed1 = myFish1.customSplice(2, 0, "drum", "guitar");
console.log({ myFish1, removed1 });
// myFish1: [ 'angel', 'clown', 'drum', 'guitar', 'mandarin', 'sturgeon' ],
// removed1: []

// Remove 0 (zero) elements at index 0, and insert "angel"
// splice(0, 0, ...elements) inserts elements at the start of
// the array like unshift().
const myFish2 = ["clown", "mandarin", "sturgeon"];
const removed2 = myFish2.customSplice(0, 0, "angel");
console.log({ myFish2, removed2 });
// myFish2: [ 'angel', 'clown', 'mandarin', 'sturgeon' ], removed2: []

// Remove 0 (zero) elements at last index, and insert "sturgeon"
// splice(array.length, 0, ...elements) inserts elements at the
// end of the array like push().
const myFish3 = ["angel", "clown", "mandarin"];
const removed3 = myFish3.customSplice(myFish.length, 0, "sturgeon");
console.log({ myFish3, removed3 });
// myFish3: [ 'angel', 'clown', 'mandarin', 'sturgeon' ], removed3: []

// Remove 1 element at index 3
const myFish4 = ["angel", "clown", "drum", "mandarin", "sturgeon"];
const removed4 = myFish4.customSplice(3, 1);
console.log({ myFish4, removed4 });
// myFish4: [ 'angel', 'clown', 'drum', 'sturgeon' ],
// removed4: [ 'mandarin' ]

// Remove 1 element at index 2, and insert "trumpet"
const myFish5 = ["angel", "clown", "drum", "sturgeon"];
const removed5 = myFish5.customSplice(2, 1, "trumpet");
console.log({ myFish5, removed5 });
// myFish5: [ 'angel', 'clown', 'trumpet', 'sturgeon' ],
// removed5: [ 'drum' ]

// Remove 2 elements from index 0, and insert "parrot", "anemone" and "blue"
const myFish6 = ["angel", "clown", "trumpet", "sturgeon"];
const removed6 = myFish6.customSplice(0, 2, "parrot", "anemone", "blue");
console.log({ myFish6, removed6 });
// myFish6: [ 'parrot', 'anemone', 'blue', 'trumpet', 'sturgeon' ],
// removed6: [ 'angel', 'clown' ]

// Remove 2 elements, starting from index 2
const myFish7 = ["parrot", "anemone", "blue", "trumpet", "sturgeon"];
const removed7 = myFish7.customSplice(2, 2);
console.log({ myFish7, removed7 });
// myFish7: [ 'parrot', 'anemone', 'sturgeon' ],
// removed7: [ 'blue', 'trumpet' ]

// Remove 1 element from index -2
const myFish8 = ["angel", "clown", "mandarin", "sturgeon"];
const removed8 = myFish8.customSplice(-2, 1);
console.log({ myFish8, removed8 });
// myFish8: [ 'angel', 'clown', 'sturgeon' ], removed8: [ 'mandarin' ]

// Remove all elements, starting from index 2
const myFish9 = ["angel", "clown", "mandarin", "sturgeon"];
const removed9 = myFish9.customSplice(2);
console.log({ myFish9, removed9 });
// myFish9: [ 'angel', 'clown' ], removed9: [ 'mandarin', 'sturgeon' ]
```

Custom Implementations of Promise.all, Promise.any, Promise.allSettled, and Promise.race

Learn how JavaScript's static Promise methods work by building them from scratch. This page covers custom implementations of `Promise.all`, `Promise.allSettled`, `Promise.any`, and `Promise.race` — useful for understanding async behavior, debugging, and preparing for interviews.

Promise.all

The `Promise.all()` static method takes an iterable of promises and returns a single `Promise` that:

- **fulfills** when all input promises have fulfilled, returning an array of results in the original order
- **rejects** immediately if any promise rejects, with the first rejection reason

This example includes a custom implementation of `Promise.all()` that resolves all promises and preserves their order, even if they complete at different times.

```
function myPromiseAll(promises) {
  return new Promise((resolve, reject) => {
    if (!Array.isArray(promises)) {
      return reject(new TypeError("Argument must be an array"));
    }

    const results = [];
    let completedPromises = 0;

    for (let index = 0; index < promises.length; index++) {
      Promise.resolve(promises[index])
        .then((value) => {
          results[index] = value;
          console.log(value);
          completedPromises += 1;
          if (completedPromises === promises.length) {
            resolve(results);
          }
        })
        .catch(reject);
    }

    if (promises.length === 0) {
      resolve([]);
    }
  });
}

const promise1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, "first");
});
const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, "second");
});
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 5000, "third");
});
```

```
myPromiseAll([promise1, promise2, promise3]).then((values) => {
  console.log(values);
});

// second
// first
// third
// [ 'first', 'second', 'third' ]
```

Promise.allSettled

The `Promise.allSettled()` method returns a promise that **resolves after all promises in the input iterable have settled**, regardless of whether they were fulfilled or rejected. Each result is an object with a `status` of either `"fulfilled"` or `"rejected"` and contains either the `value` or the `reason`.

This example demonstrates a custom implementation of `Promise.allSettled()` using `.then()` and `.catch()` handlers for consistent result formatting.

```
const rejectHandler = (reason) => ({ status: "rejected", reason });
const resolveHandler = (value) => ({ status: "fulfilled", value });

Promise.customAllSettled = function (promises) {
  const convertedPromises = promises.map((p) =>
    Promise.resolve(p).then(resolveHandler, rejectHandler),
  );

  return Promise.all(convertedPromises);
};

Promise.customAllSettled([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) =>
    setTimeout(() => reject(new Error("Whoops!")), 2000),
  ),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000)),
])
  .then(console.info)
  .catch(console.error);
// [
//   { status: 'fulfilled', value: 1 },
//   {
//     status: 'rejected',
//     reason: Error: Whoops!...
// },
// { status: 'fulfilled', value: 3 }
// ]
```

Promise.any

The `Promise.any()` method returns a promise that **fulfills as soon as any of the input promises is fulfilled**. If **all** promises reject, it rejects with an `AggregateError` containing all rejection reasons.

This example demonstrates a custom implementation of `Promise.any()` that resolves with the first successful result or throws an aggregate error if none succeed.

```
Promise.customAny = function (promises) {
  return new Promise((resolve, reject) => {
    const errors = [];
    let remaining = promises.length;

    if (remaining === 0) {
      return reject(new AggregateError([], "All promises were rejected"));
    }

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then(resolve)
        .catch((error) => {
          errors[index] = error;
          remaining -= 1;
          if (remaining === 0) {
            reject(new AggregateError(errors, "All promises were rejected"));
          }
        });
    });
  });
};

const promise1 = Promise.reject(0);
const promise2 = new Promise((resolve) => setTimeout(resolve, 100, "quick"));
const promise3 = new Promise((resolve) => setTimeout(resolve, 500, "slow"));

const promises = [promise1, promise2, promise3];

Promise.customAny(promises).then((value) => console.log(value)); // quick
```

Promise.race

The `Promise.race()` method returns a promise that **settles as soon as the first input promise settles** — whether it fulfills or rejects. This makes it useful when you're interested only in the fastest outcome, regardless of its result.

This example includes a custom implementation of `Promise.race()` that resolves or rejects based on whichever promise finishes first.

```
Promise.customRace = function (promises) {
  return new Promise((resolve, reject) => {
    for (const promise of promises) {
      Promise.resolve(promise).then(resolve, reject);
    }
  });
};

Promise.customRace([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) =>
    setTimeout(() => reject(new Error("Whoops!")), 2000),
  ),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000)),
]).then(console.log); // 1
```

Native JavaScript Array Methods - Read Methods

This collection includes non-mutating JavaScript array methods reimplemented from scratch. Explore how methods like `find()`, `at()`, `every()`, `some()`, `includes()`, and `indexOf()` behave internally — all of which access or test array values without modifying the original array.

Array.find

The `find()` method returns the **first element** in the array that satisfies the provided testing function. If no match is found, it returns `undefined`. Ideal for retrieving an item based on a condition without iterating the entire array manually.

```
Array.prototype.customFind = function (callback) {
  for (let i = 0; i < this.length; i++) {
    if (callback(this[i])) {
      return this[i];
    }
  }
};

const array1 = [5, 12, 8, 130, 44];
const found = array1.customFind((element) => element > 10);
console.log(found); // 12

const inventory = [
  { name: "apples", quantity: 2 },
  { name: "bananas", quantity: 0 },
  { name: "cherries", quantity: 5 },
];

const result = inventory.customFind(({ name }) => name === "cherries");
console.log(result); // { name: 'cherries', quantity: 5 }

const result1 = inventory.customFind(({ name }) => name === "nothing");
console.log(result1); // undefined
```

Array.findIndex

The `findIndex()` method returns the **index** of the first element that passes the provided test function. If no element matches, it returns `-1`. Useful for locating the position of an item instead of the item itself.

```
Array.prototype.customFindIndex = function (callback) {
  for (let i = 0; i < this.length; i++) {
    if (callback(this[i])) {
      return i;
    }
  }

  return -1;
};
```

```
const array1 = [5, 12, 8, 130, 44];
const found = array1.customFindIndex((element) => element > 10);
console.log(found); // 1

const inventory = [
  { name: "apples", quantity: 2 },
  { name: "bananas", quantity: 0 },
  { name: "cherries", quantity: 5 },
];

const result = inventory.customFindIndex(({ name }) => name === "cherries");
console.log(result); // 2

const result1 = inventory.customFindIndex(({ name }) => name === "nothing");
console.log(result1); // -1
```

Array.findLast

The `findLast()` method iterates the array **in reverse** and returns the **first value** that satisfies the given condition. If no element matches, `undefined` is returned. Useful when you want the **last matching item** in an array without reversing it manually.

```
Array.prototype.customFindLast = function (callback) {
  for (let i = this.length; i >= 0; i--) {
    if (callback(this[i])) {
      return this[i];
    }
  }
};

const array1 = [5, 12, 8, 130, 44];
const found = array1.customFindLast((element) => element > 10);
console.log(found); // 44
```

Array.at

The `at()` method takes an integer and returns the item at that index — supporting both **positive and negative** integers. Negative values count from the end of the array, making it cleaner than using `array.length - index`.

This example shows a custom implementation of `Array.prototype.at()` to mimic the native behavior.

```
Array.prototype.customAt = function (index) {
  if (index < 0) {
    index = this.length + index;
  }

  return this[index];
};

const array1 = [5, 12, 8, 130, 44];
console.log(array1.customAt(-1)); // 44
```

Array.every

The `every()` method tests whether **all elements** in an array pass the test implemented by the provided callback. It returns a **boolean** — `true` if all elements satisfy the condition, and `false` otherwise.

This example includes a custom implementation of `Array.prototype.every()` and shows how it can be used for subset checks.

```
Array.prototype.customEvery = function (callback) {
  for (let i = 0; i < this.length; i++) {
    if (!callback(this[i], i)) {
      return false;
    }
  }

  return true;
};

const arr = [1, 30, 39, 29, 10, 13];

console.log(arr.customEvery((currentValue) => currentValue > 40));
// false
console.log(arr.customEvery((currentValue) => currentValue < 40));
// true

const isSubset = (array1, array2) =>
  array2.customEvery((element) => array1.includes(element));

console.log(isSubset([1, 2, 3, 4, 5, 6, 7], [5, 7, 6])); // true
console.log(isSubset([1, 2, 3, 4, 5, 6, 7], [5, 8, 7])); // false
```

Array.includes

The `includes()` method checks if an array contains a specific value, returning `true` or `false`. It uses the `SameValueZero` equality comparison — meaning it treats `NaN` as equal to `NaN`.

This example includes a custom implementation of `Array.prototype.includes()` with support for optional `fromIndex` and negative indexing behavior.

```
function sameValueZero(x, y) {
  return (
    x === y ||
    (typeof x === "number" && typeof y === "number" && x !== x && y !== y)
  );
}

Array.prototype.customIncludes = function (searchElement, fromIndex = 0) {
  const length = this.length;

  if (length === 0) {
    return false;
  }

  if (fromIndex < 0) {
    fromIndex = Math.max(length + fromIndex, 0);
```

```
  }

  for (let i = fromIndex; i < length; i++) {
    if (sameValueZero(this[i], searchElement)) {
      return true;
    }
  }

  return false;
};

console.log([1, 2, 3].customIncludes(2)); // true
console.log([1, 2, 3].customIncludes(4)); // false
console.log([1, 2, 3].customIncludes(3, 3)); // false
console.log([1, 2, 3].customIncludes(3, -1)); // true
console.log([1, 2, NaN].customIncludes(NaN)); // true
console.log(["1", "2", "3"].customIncludes(3)); // false

const arr = ["a", "b", "c"];
// Since -100 is much less than the array length,
// it starts checking from index 0.
console.log(arr.customIncludes("a", -100)); // true
console.log(arr.customIncludes("a", -2)); // false
console.log(arr.customIncludes("a", -3)); // true
```

Array.indexOf

The `indexOf()` method returns the **first index** at which a given element can be found in the array, or `-1` if it is not present. It uses **strict equality (`===`)** for comparison and supports an optional `fromIndex` to control the search starting point.

This custom implementation of `Array.prototype.indexOf()` also handles negative indexing by adjusting the start position.

```
Array.prototype.customIndexOf = function (searchElement, fromIndex = 0) {
  if (fromIndex < 0) {
    fromIndex = this.length + fromIndex;
  }

  for (let i = fromIndex; i < this.length; i++) {
    if (this[i] === searchElement) {
      return i;
    }
  }

  return -1;
};

const beasts = ["ant", "bison", "camel", "duck", "bison"];
console.log(beasts.customIndexOf("bison")); // 1
// Start from index 2
console.log(beasts.customIndexOf("bison", 2)); // 4
console.log(beasts.customIndexOf("giraffe")); // -1

const array = [2, 9, 9];
console.log(array.customIndexOf(9, 2)); // 2
console.log(array.customIndexOf(2, -1)); // -1
```

```
console.log(array.customIndexOf(2, -3)); // 0
console.log(array.customIndexOf(2, -100)); // 0
```

Array.some

The `some()` method tests whether **at least one element** in the array passes the test implemented by the provided function. It returns `true` if any element satisfies the condition; otherwise, it returns `false`. This method does not modify the original array.

This custom implementation of `Array.prototype.some()` iterates through the array and returns `true` if any element passes the provided test.

```
Array.prototype.customSome = function (callback) {
  for (let i = 0; i < this.length; i++) {
    if (callback(this[i], i, this)) {
      return true;
    }
  }

  return false;
};

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

const even = (element) => element % 2 === 0;
console.log(array.customSome(even)); // true

const equal90 = (element) => element === 90;
console.log(array.customSome(equal90)); // false
```

JavaScript Set, Object.assign and groupBy, and Array.entries

JavaScript utilities `Set`, `Object.groupBy()`, `Object.assign()`, and `Array.prototype.entries()`. These examples show how to work with unique collections, group items by category, clone or merge objects, and iterate over arrays with index-value pairs — all with hands-on explanations and custom implementations for deeper insight.

Set

The `Set` object lets you store **unique values of any type**, whether primitive or object references. A `Set` automatically removes duplicate elements and maintains the **insertion order**.

This example demonstrates how to create a set, add/remove elements, check membership, iterate through a set, and perform common set operations like union, intersection, and difference.

```
// Create a Set
const mySet = new Set([1, 2, 3, 4, 5]);
console.log(mySet); // Set { 1, 2, 3, 4, 5 }

// returns a new set iterator object that contains the values
// for each element in this set in insertion order
const setValues = mySet.values();
console.log(setValues);
// [Set Iterator] { 1, 2, 3, 4, 5 }
console.log(setValues.next()); // { value: 1, done: false }

// Add elements to a Set
mySet.add(6); // Adds a new element
mySet.add(2); // Duplicate values are ignored
console.log(mySet); // Set { 1, 2, 3, 4, 5, 6 }

// Check if a Set contains an element
console.log(mySet.has(3)); // true
console.log(mySet.has(10)); // false

// Remove elements from a Set
mySet.delete(4); // Removes the element 4
console.log(mySet); // Set { 1, 2, 3, 5, 6 }

// Get the size of a Set
console.log(mySet.size); // 5

// Iterate over a Set
for (const value of mySet) {
  console.log(value); // Logs each value in the Set
}

// Convert a Set to an Array
const arrayFromSet = Array.from(mySet); // Using Array.from()
console.log(arrayFromSet); // [1, 2, 3, 5, 6]

const anotherArray = [...mySet]; // Using spread operator
console.log(anotherArray); // [1, 2, 3, 5, 6]

// Clear a Set
```

```
mySet.clear(); // Removes all elements
console.log(mySet); // Set {}

// Find the Union of Two Sets
const setA = new Set([1, 2, 3]);
const setB = new Set([3, 4, 5]);
// Combines elements from both sets
const union = new Set([...setA, ...setB]);
console.log(union); // Set { 1, 2, 3, 4, 5 }

// Find the Intersection of Two Sets
// Common elements
const intersection = new Set([...setA].filter((x) => setB.has(x)));
console.log(intersection); // Set { 3 }

// Find the Difference of Two Sets
// Elements in setA not in setB
const difference = new Set([...setA].filter((x) => !setB.has(x)));
console.log(difference); // Set { 1, 2 }

const a = {};
const b = {};
const c = {};
const setOfObjects = new Set([a, a, b]);
console.log(setOfObjects); // Set(2) { {}, {} }
console.log(setOfObjects.has(a)); // true
console.log(setOfObjects.has(c)); // false
```

Object.groupBy

The `Object.groupBy()` static method groups elements of an iterable based on the **string key** returned by a callback function. It returns a plain object with keys representing group names and values as arrays of grouped items.

This example shows a manual implementation using `Array.prototype.reduce()` to group inventory items by their `type` field.

```
const groupBy = (arr, callback) => {
  return arr.reduce((acc = {}, item) => {
    const key = callback(item);
    if (!acc[key]) acc[key] = [];
    acc[key].push(item);

    return acc;
  }, {});
};

const inventory = [
  { name: "asparagus", type: "vegetables", quantity: 5 },
  { name: "bananas", type: "fruit", quantity: 0 },
  { name: "goat", type: "meat", quantity: 23 },
  { name: "cherries", type: "fruit", quantity: 5 },
  { name: "fish", type: "meat", quantity: 22 },
];
const result = groupBy(inventory, ({ type }) => type);
console.log(result);

// {
//   vegetables: [{ name: "asparagus", type: "vegetables", quantity: 5 }],
//   fruit: [
```

```
//     { name: "bananas", type: "fruit", quantity: 0 },
//     { name: "cherries", type: "fruit", quantity: 5 },
//   ],
//   meat: [
//     { name: "goat", type: "meat", quantity: 23 },
//     { name: "fish", type: "meat", quantity: 22 },
//   ],
// };
```

Object.assign

The `Object.assign()` method copies all **enumerable own properties** from one or more source objects to a target object. It returns the **modified target**, making it useful for shallow cloning or merging multiple objects.

This example also shows how `Object.assign()` behaves similarly to the spread operator (`...`) when cloning simple objects.

```
const obj = {
  foo: 1,
  get bar() {
    return 2;
  },
};

let copy = Object.assign({}, obj);
console.log(copy); // { foo: 1, bar: 2 }
// The value of copy.bar is obj.bar's getter's return value.

// Following 2 lines of code are the same.
const objClone = { ...obj }; // { foo: 1, bar: 2 }
const objClone2 = Object.assign({}, obj); // { foo: 1, bar: 2 }
```

Array.entries

The `entries()` method returns a new array iterator object containing `[index, value]` pairs for each element. It's commonly used in `for...of` loops to iterate with both index and value.

This example recreates `Array.prototype.entries()` using a generator to yield key/value pairs.

```
Array.prototype.customEntries = function () {
  const entries = [];
  for (let i = 0; i < this.length; i++) {
    entries.push([i, this[i]]);
  }

  function* iterator() {
    yield* entries;
  }

  return iterator();
};
```

```
const arr = [{ x: "a" }, "b", ["c"]];
const iterator = arr.customEntries();

console.log(iterator.next());
// { value: [ 0, { x: 'a' } ], done: false }
console.log(iterator.next());
// { value: [ 1, 'b' ], done: false }
console.log(iterator.next());
// { value: [ 2, [ 'c' ] ], done: false }
console.log(iterator.next());
// { value: undefined, done: true }

for (const [index, element] of arr.customEntries()) {
  console.log(index, element);
}
// 0 { x: 'a' }
// 1 b
// 2 [ 'c' ]
```

JavaScript Array Non-Mutating Methods

This section explores essential JavaScript array methods used for transforming, reshaping, and extracting data — including `map()`, `filter()`, `reduce()`, `flat()`, `flatMap()`, `join()`, and `slice()`.

These are all **non-mutating methods**, meaning they return new arrays or values without changing the original input. You'll learn how they work internally with custom polyfills.

Array.filter

The `filter()` method creates a **shallow copy** of an array, including only elements that pass the test implemented by the provided callback function. It's ideal for narrowing down data based on conditions, like filtering by length, type, or custom logic.

This example includes a custom implementation of `Array.prototype.filter()` and practical use cases like filtering long words or prime numbers.

```
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;
};

const words = ["spray", "elite", "exuberant", "destruction", "present"];

const result = words.myFilter((word) => word.length > 6);

console.log(result);
// ["exuberant", "destruction", "present"]

const array = [-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

function isPrime(num) {
  for (let i = 2; num > i; i++) {
    if (num % i === 0) {
      return false;
    }
  }
  return num > 1;
}

console.log(array.myFilter(isPrime)); // [ 2, 3, 5, 7 ]
```

Array.flat

The `flat()` method creates a new array with all sub-array elements **recursively flattened** up to the specified depth. It's useful for simplifying nested arrays into a single-level array — or flattening deeply nested structures with `Infinity`.

This example provides a custom implementation of `Array.prototype.flat()`, supporting variable depth.

```
Array.prototype.customFlat = function (depth = 1) {
  const result = [];

  const flatten = (array, depth) => {
    for (const item of array) {
      if (Array.isArray(item) && depth > 0) {
        flatten(item, depth - 1);
      } else {
        result.push(item);
      }
    }
  };
  flatten(this, depth);

  return result;
};

const arr = [0, 1, [2, [3, [4, 5]]]];
console.log(arr.customFlat());
// [0, 1, 2, [3, [4, 5]]]
console.log(arr.customFlat(2));
// [0, 1, 2, 3, [4, 5]]
console.log(arr.customFlat(Infinity));
// [0, 1, 2, 3, 4, 5]
```

Array.flatMap

The `flatMap()` method first maps each element using a given callback function, then **flattens the result by one level**. It behaves like `array.map(...).flat()`, but is more efficient and expressive for nested transformations.

This example demonstrates both native usage and a custom implementation of `Array.prototype.flatMap()` using a real-world data structure.

```
Array.prototype.customFlatMap = function (callback, thisArg) {
  const result = [];

  for (let i = 0; i < this.length; i++) {
    const mapped = callback.call(thisArg, this[i], i, this);

    if (Array.isArray(mapped)) {
      result.push(...mapped); // Use spread operator for flattening
    } else {
      result.push(mapped);
    }
  }

  return result;
};

const data = [
```

```
    {
      id: 1,
      name: "Category A",
      items: [
        {
          id: 2,
          name: "Subcategory A1",
        },
        {
          id: 5,
          name: "Subcategory A2",
        },
      ],
    },
    {
      id: 8,
      name: "Category B",
      items: [
        {
          id: 9,
          name: "Subcategory B1",
        },
        {
          id: 12,
          name: "Subcategory B2",
        },
      ],
    },
];

const items = data
  .map((category) =>
    category.items.map((item) => ({
      ...item,
      category: category.name,
    })),
  )
  .flat();
console.log(items);
// [
//   { id: 2, name: 'Subcategory A1', category: 'Category A' },
//   { id: 5, name: 'Subcategory A2', category: 'Category A' },
//   { id: 9, name: 'Subcategory B1', category: 'Category B' },
//   { id: 12, name: 'Subcategory B2', category: 'Category B' }
// ]

const items2 = data.customFlatMap((category) =>
  category.items.map((item) => ({
    ...item,
    category: category.name,
  })),
);
console.log(items2);
// ...same result
```

Array.join

The `join()` method creates and returns a **string** by concatenating all elements of an array, separated by a specified string (default is a comma). If the array has only one item, that item is returned as-is. It's useful for formatting output or combining array values into CSV-like strings.

This example includes a custom implementation of `Array.prototype.join()` that replicates the native behavior.

```
Array.prototype.customJoin = function (separator = ",") {
  let result = "";

  for (let i = 0; i < this.length; i++) {
    if (i > 0) {
      result += separator;
    }

    result += this[i];
  }

  return result;
};

const elements = ["Fire", "Air", "Water"];
console.log(elements.customJoin()); // "Fire,Air,Water"
console.log(elements.customJoin("")); // "FireAirWater"
console.log(elements.customJoin("-")); // "Fire-Air-Water"

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

Array.map

The `map()` method creates a **new array** populated with the results of calling a provided function on every element in the original array. It's a key tool for transforming data structures in a clean, functional style — especially for UI rendering, data formatting, or calculations.

This example includes a custom implementation of `Array.prototype.map()` along with common use cases like doubling values, reformatting objects, and calculating cumulative sums.

```
Array.prototype.customMap = function (callbackFn) {
  if (typeof callbackFn !== "function") {
    throw new TypeError("Callback must be a function");
  }

  const arr = [];
  for (let i = 0; i < this.length; i++) {
    arr.push(callbackFn(this[i], i, this));
  }

  return arr;
};

const array1 = [1, 4, 9, 16];
const doubled = array1.customMap((x) => x * 2);
console.log(doubled); // [ 2, 8, 18, 32 ]

const kvArray = [
```

```
  { key: 1, value: 10 },
  { key: 2, value: 20 },
  { key: 3, value: 30 },
];
const formattedArray = kvArray.customMap(({ key, value }) => ({
  [key]: value,
}));
console.log(formattedArray); // [ { 1: 10 }, { 2: 20 }, { 3: 30 } ]

console.log(["1", "2", "3"].customMap(Number)); // [ 1, 2, 3 ]

const numbers2 = [1, 2, 3, 4, 5];
const cumulativeSum = numbers2.customMap((num, idx, arr) => {
  // Calculate cumulative sum by adding the current number to the sum of previous numbers
  const previousSum =
    idx > 0 ? arr.slice(0, idx).reduce((acc, curr) => acc + curr, 0) : 0;
  return previousSum + num;
});
console.log(cumulativeSum); // [ 1, 3, 6, 10, 15 ]
```

Array.reduce

The `reduce()` method executes a user-defined "reducer" callback function on each element of the array, in order. It accumulates a **single value** by applying the callback function and passing the result from one iteration to the next.

The method can accept an **optional initial value**, and if not provided, it defaults to the first element in the array. This custom implementation mimics the native behavior.

This example demonstrates usage of `reduce` for both synchronous and asynchronous operations, including practical examples like summing an array and composing a pipeline of functions.

```
Array.prototype.customReduce = function (callback, initialValue) {
  let accumulator = initialValue !== undefined ? initialValue : this[0];

  const startIndex = initialValue !== undefined ? 0 : 1;

  for (let i = startIndex; i < this.length; i++) {
    accumulator = callback(accumulator, this[i], i, this);
  }

  return accumulator;
};

const array1 = [15, 16, 17, 18, 19];

const sumWithInitial = array1.customReduce((acc, cur) => acc + cur, 0);
console.log(sumWithInitial); // 85

const sumWithoutInitial = array1.customReduce((acc, cur) => acc + cur);
console.log(sumWithoutInitial); // 85

const pipe =
  (...functions) =>
  (initialValue) =>
    functions.customReduce((acc, fn) => fn(acc), initialValue);

const double = (x) => x * 2;
const triple = (x) => x * 3;
```

```
const multiply6 = pipe(double, triple);
const multiply9 = pipe(triple, triple);

console.log(multiply6(6)); // 36
console.log(multiply9(9)); // 81

const asyncPipe =
  (...functions) =>
  (initialValue) =>
    functions.reduce((acc, fn) => acc.then(fn), Promise.resolve(initialValue));

const p1 = async (a) => a * 5;
const p2 = async (a) =>
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(a * 2), 1000);
  });
const f3 = (a) => a * 3;
const p4 = async (a) => a * 4;

asyncPipe(p1, p2, f3, p4)(10).then(console.log); // 1200

const asyncPipeAsync =
  (...functions) =>
  (initialValue) =>
    functions.reduce(async (acc, fn) => fn(await acc), initialValue);

asyncPipeAsync(p1, p2, f3, p4)(10).then(console.log); // 1200
```

Array.slice

The `slice()` method returns a **shallow copy** of a portion of an array into a new array, without modifying the original array. The selection is made from `start` index to `end` index (excluding the `end` element).

This custom implementation of `Array.prototype.slice()` replicates the native behavior, with support for negative indexing.

```
Array.prototype.customSlice = function (start = 0, end) {
  const length = this.length;
  let endIndex = end || length;

  if (start < 0) {
    start = Math.max(length + start, 0);
  }
  if (endIndex < 0) {
    endIndex = Math.max(length + endIndex, 0);
  }

  const result = [];

  for (let i = start; i < endIndex && i < length; i++) {
    result.push(this[i]);
  }

  return result;
};

const animals = ["ant", "bison", "camel", "duck", "elephant"];
```

```
console.log(animals.customSlice(2));
// ["camel", "duck", "elephant"]
console.log(animals.customSlice(2, 4)); // ["camel", "duck"]
console.log(animals.customSlice(-2)); // ["duck", "elephant"]
console.log(animals.customSlice(2, -1)); // ["camel", "duck"]
console.log(animals.customSlice());
// ["ant", "bison", "camel", "duck", "elephant"]
```

JavaScript Memoization, Mixins, Promises, and Proxies

Optimize performance and structure code effectively with advanced JavaScript patterns including `memoize`, `mixin`, `Proxy`, custom iterators, and asynchronous promise handling.

Memoize

Memoization is an optimization technique that caches the result of function calls based on their input arguments. If the same inputs occur again, the cached result is returned instead of recalculating.

This example wraps a basic function (`adder`) with a `memoize` utility using a `Map` and `JSON.stringify` to store and retrieve results by argument key.

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

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

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

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

    return result;
  };
}

const adder = function (a, b) {
  console.log(`Calculating ${a} + ${b}`);
  return a + b;
};

const memoizedAdder = memoize(adder);
const result1 = memoizedAdder(3, 4); // Calculating 3 + 4
console.log(result1); // 7
const result2 = memoizedAdder(3, 4);
console.log(result2); // 7
```

Mixins

Mixins allow objects to share reusable behavior without using classical inheritance. In this example, the `CanSpeak` and `CanWalk` mixins are applied to different classes using `Object.assign`, enabling both `Person` and `Robot` to share functionality like `speak` and `walk`.

mixins

```
// Define a mixin for shared behavior
const CanSpeak = {
  speak() {
    console.log(`${this.name} says: ${this.message}`);
  },
};

// Define another mixin
const CanWalk = {
  walk() {
    console.log(`${this.name} is walking.`);
  },
};

// Create a base class
class Person {
  constructor(name, message) {
    this.name = name;
    this.message = message;
  }
}

// Apply mixins to the class
Object.assign(Person.prototype, CanSpeak, CanWalk);

// Create an instance of the class
const john = new Person('John', 'Hello, world!');

// Use methods from mixins
john.speak(); // Output: John says: Hello, world!
john.walk();  // Output: John is walking.

// Another example with a different class
class Robot {
  constructor(name, message) {
    this.name = name;
    this.message = message;
  }
}

// Apply mixins to the Robot class
Object.assign(Robot.prototype, CanSpeak, CanWalk);

const r2d2 = new Robot('R2D2', 'Beep boop');
r2d2.speak(); // Output: R2D2 says: Beep boop
r2d2.walk();  // Output: R2D2 is walking.
```

Promise Chaining

promise-chaining

This example demonstrates how to chain Promises to perform sequential asynchronous operations. Each `.then()` receives the result of the previous one, allowing for clean, readable logic with predictable timing — in this case, doubling a number step by step with 1-second delays.

```
new Promise(function (resolve, reject) {
  setTimeout(() => resolve(1), 1000);
```

```
})
  .then(function (result) {
    console.log(result); // 1

    return new Promise((resolve, reject) => {
      setTimeout(() => resolve(result * 2), 1000);
    });
  })
  .then(function (result) {
    console.log(result); // 2

    return new Promise((resolve, reject) => {
      setTimeout(() => resolve(result * 2), 1000);
    });
  })
  .then(function (result) {
    console.log(result); // 4
  });
// 1, 2, 4 with 1s between
```

Promise Error Handling

These examples show how to handle errors in promise chains using `.catch()`. They demonstrate how synchronous errors thrown during the executor function are automatically caught, while asynchronous errors (e.g., in `setTimeout`) are **not** — unless they're wrapped in a `try...catch` or returned from a rejected promise.

Understanding where and when errors can be caught is key to building reliable async flows.

```
new Promise((resolve, reject) => {
  throw new Error("Whoops!");
})
  .catch(function (error) {
    if (error instanceof URIError) {
      // handle it
    } else {
      console.log("Can't handle such error");

      throw error;
      // throwing this or another error jumps to the next catch
    }
  })
  .then(function () {
    /* doesn't run here */
  })
  .catch((error) => {
    console.error(`The unknown error has occurred: ${error.message}`);
    // don't return anything => execution goes the normal way
  });

// Can't handle such error
// The unknown error has occurred: Whoops!
```

Synchronous vs Asynchronous Errors in Promises

Promises automatically catch **synchronous errors** thrown inside the executor. However, errors thrown **asynchronously** (e.g., inside `setTimeout`) are not caught unless explicitly wrapped in a rejecting `Promise` or a `try...catch` block inside async functions.

```
new Promise(function (resolve, reject) {
  throw new Error("Whoops!");
}).catch((e) => console.error(e.message)); // Whoops!

// There's an "implicit try..catch" around the function code.
// So all synchronous errors are handled.
// But here the error is generated not while the executor is
// running, but later. So the promise can't handle it.

new Promise(function (resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(console.error); // unhandled error ...
```

Proxy

This example uses a `Proxy` to intercept property access on an object. When a key is found in the `dictionary`, the translation is returned; otherwise, the original key is returned as a fallback. Proxies are powerful for defining custom behavior for fundamental operations like `get`, `set`, and more.

```
let dictionary = {
  Hello: "Hola",
  Bye: "Adiós",
};

dictionary = new Proxy(dictionary, {
  get(target, phrase) {
    // intercept reading a property from dictionary
    if (phrase in target) {
      // if we have it in the dictionary
      return target[phrase]; // return the translation
    } else {
      // otherwise, return the non-translated phrase
      return phrase;
    }
  },
});

// Look up arbitrary phrases in the dictionary!
// At worst, they're not translated.
console.log(dictionary["Hello"]); // Hola
console.log(dictionary["Welcome to Proxy"]); // Welcome to Proxy
```

Range Iterator

This example demonstrates how to create a custom iterator using the iterator protocol. The `makeRangeIterator` function generates numbers from `start` to `end` with a defined `step`, manually implementing the `next()` method.

It’s useful for creating lazy sequences or custom iteration logic without relying on generators.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
  let nextIndex = start;
  let iterationCount = 0;

  return {
    next() {
      let result;

      if (nextIndex < end) {
        result = { value: nextIndex, done: false };
        nextIndex += step;
        iterationCount++;

        return result;
      }

      return { value: iterationCount, done: true };
    },
  };
}

const iter = makeRangeIterator(1, 6, 2);

let result = iter.next();
while (!result.done) {
  console.log(result.value); // 1 3 5
  result = iter.next();
}

console.log("Iterated over sequence of size:", result.value);
// 1, 3, 5
// Iterated over sequence of size: 3
```

Shooters: Closures & Lexical Scope

Every function is meant to output its number. All `shooter` functions are created in the lexical environment of `makeArmy()` function. But when `army[5]()` is called, `makeArmy` has already finished its job, and the final value of `i` is `10` (`while` stops at `i=10`). As the result, all `shooter` functions get the same value from the outer lexical environment and that is, the last value, `i=10`. Solution is to save variable `let j = i`

```
function makeArmy() {
  const shooters = [];

  let i = 0;
  while (i < 10) {
    let j = i; // save local variable
    const shooter = function () {
      // create a shooter function,
      return j; // that should show its number
    };
    shooters.push(shooter); // and add it to the array
    i++;
  }

  // ...and return the array of shooters
```

```
  return shooters;
}

const army = makeArmy();

console.log(army[0]()); // 0
console.log(army[1]()); // 1
console.log(army[5]()); // 5
```

JavaScript Delay, Debounce, Throttle, and Async Execution

Master JavaScript's timing mechanisms like `delay`, `debounce`, `throttle`, `setTimeout`, and `setInterval`. These tools help control execution flow, improve performance, and manage async behavior — most common question on interviews.

Delay Decorator

This example defines a `delay` decorator that wraps a function and delays its execution by a specified number of milliseconds.

It preserves the correct `this` context using `Function.prototype.apply`, allowing methods from different objects to be delayed without losing their binding. Useful for debouncing, throttling, or deferring side effects.

```
const obj1 = {
  info: "obj1 info",
  showInfo(...args) {
    console.log(`${args}: ${this.info}`);
  },
};

const obj2 = {
  info: "obj2 info",
};

function delay(f, ms) {
  return function (...args) {
    setTimeout(() => {
      // Uses apply to handle `this` dynamically
      f.apply(this, args);
    }, ms);
  };
}

obj1.delayedShowInfo = delay(obj1.showInfo, 1000);
obj2.delayedShowInfo = delay(obj1.showInfo, 1000);

obj1.delayedShowInfo("Info", "one"); // Info,one: obj1 info
obj2.delayedShowInfo("Info"); // Info: obj2 info
```

Delaying function execution with and without context (`this`)

This example shows how to create a `delay` utility that defers a function's execution by a specified time using `setTimeout`. It works great for standalone functions but does **not preserve `this` context** when used with object methods.

For object methods, `Function.prototype.bind` or `apply` should be used to retain proper context.

```
function delay(f, ms) {
  return function (...args) {
    setTimeout(() => {
      f(...args);
    }, ms);
  };
}

function showDetails(name, age) {
  console.log(`Name: ${name}, Age: ${age}`);
}

const delayedShowDetails = delay(showDetails, 1000);
delayedShowDetails("Alice", 25);
// Name: Alice, Age: 25

const obj1 = {
  info: "obj1 info",
  showInfo(prefix) {
    console.log(`${prefix}: ${this?.info}`);
  },
};

obj1.delayedShowInfo = delay(obj1.showInfo, 1000);
obj1.delayedShowInfo("Info", "one"); // Info: undefined
```

Output Every Second

setTimeout and setInterval

This example demonstrates two ways to print numbers at 1-second intervals using `setInterval` and recursive `setTimeout`. Both approaches achieve the same result, but `setTimeout` gives finer control over the delay between executions — especially useful for dynamic scheduling or error handling.

setInterval

```
function printNumbers(from, to) {
  let current = from;
  let timerId;

  function go() {
    console.log(current);
    if (current === to) {
      clearInterval(timerId);
    }
    current++;
  }

  go();
  timerId = setInterval(go, 1000);
}

printNumbers(5, 10);
// 5 immediately, and 6 to 10 with 1s between
```

setTimeout

```
function printNumbers(from, to) {
  let current = from;

  function go() {
    console.log(current);
    if (current < to) {
      setTimeout(go, 1000);
    }
    current++;
  }

  go();
}

printNumbers(5, 10);
// 5 immediately, and 6 to 10 with 1s between
```

throttling and debounce decorator

call-apply-decorators

Compared to the debounce decorator, throttling is completely different:

- `debounce` runs the function once after the "cooldown" period.
- `throttle` runs it not more often than given `ms` time

debounce

```
function debounce(func, ms) {
  let timeout;

  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), ms);
  };
}

const timeLoggedConsoleLog = (...args) => {
  console.log(`Logged after ${Date.now() - startTime} ms:`, ...args);
};

const startTime = Date.now();
const f = debounce(timeLoggedConsoleLog, 500);

f("a");
setTimeout(() => f("b"), 200);
setTimeout(() => f("c"), 600);
setTimeout(() => f("d"), 600);
setTimeout(() => f("e"), 600); // Logged after 1118 ms: e
```

throttling

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

  return function (...args) {
    if (inThrottle) return;
    fn.apply(this, args);
    inThrottle = true;
    setTimeout(() => (inThrottle = false), limit);
  };
}

const timeLoggedConsoleLog = (...args) => {
  console.log(`Logged after ${Date.now() - startTime} ms:`, ...args);
};

const startTime = Date.now();
const f = throttle(timeLoggedConsoleLog, 500);

f("a"); // Logged after 0 ms: a
setTimeout(() => f("b"), 200);
setTimeout(() => f("c"), 600); // Logged after 613 ms: c
setTimeout(() => f("d"), 600);
setTimeout(() => f("e"), 600);
```

JavaScript Core Concepts – this, bind, loops, generators, and more

A curated collection of essential JavaScript concepts often asked about in interviews and critical for writing robust code. Each topic is backed by practical examples, real-world context, and explanations that help you **understand how and why** things work — not just memorize syntax.

Understanding this

Below `users.customFilterNoThis(army.canJoin)`, throws, `army.canJoin` was called as a standalone function, with `this=undefined`, thus leading to an instant error. A call to `users.customFilter(army.canJoin, army)` can be replaced with `users.customFilterNoThis(user => army.canJoin(user))`, that does the same. This is an object before dot

```
Array.prototype.customFilter = function (callback, thisArg) {
  let result = [];

  for (let i = 0; i < this.length; i++) {
    if (callback.call(thisArg, this[i], i, this)) {
      result.push(this[i]);
    }
  }

  return result;
};

Array.prototype.customFilterNoThis = function (callback) {
  let result = [];

  for (let i = 0; i < this.length; i++) {
    if (callback(this[i], i, this)) {
      result.push(this[i]);
    }
  }

  return result;
};

const army = {
  minAge: 18,
  maxAge: 27,
  canJoin(user) {
    return user.age >= this.minAge && user.age < this.maxAge;
  },
};

const users = [{ age: 16 }, { age: 20 }, { age: 23 }, { age: 30 }];

const soldiers1 = users.customFilterNoThis(army.canJoin);
// return user.age >= this.minAge && user.age < this.maxAge;
// TypeError: Cannot read properties of undefined (reading 'minAge')
const soldiers2 = users.customFilterNoThis((user) => army.canJoin(user));
const soldiers3 = users.customFilter(army.canJoin, army);
```

```
console.log(soldiers2); // [ { age: 20 }, { age: 23 } ]
console.log(soldiers3); // [ { age: 20 }, { age: 23 } ]
```

Loop Behavior: Pre/Post Increment

This example highlights the subtle differences between `++i` (pre-increment) and `i++` (post-increment) in `while` loops, and how they affect loop behavior and output.

It also compares `for` loops using `i++` vs `++i`, which behave the same in this context, since the increment happens after the loop body executes.

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

let i2 = 0;
while (i2++ < 3) console.log(i2);
// 1, 2, 3

for (let i = 0; i < 3; i++) console.log(i);
// 0, 1, 2

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

Object and Map Conversion

This example shows how to convert between plain JavaScript objects and `Map` instances using `Object.entries()` and `Object.fromEntries()`.

These methods are useful when working with APIs or libraries that prefer one format over the other, or when you need key ordering and additional `Map` features.

```
const prices = Object.fromEntries([
  ["banana", 1],
  ["orange", 2],
  ["meat", 4],
]);

console.log(prices);
// { banana: 1, orange: 2, meat: 4 }

const map = new Map();
map.set("banana", 1);
map.set("orange", 2);
map.set("meat", 4);

const arrayLikeMapEntries = map.entries();
const arrayMapEntries = Array.from(arrayLikeMapEntries);

const objectFromMap = Object.fromEntries(arrayMapEntries);
console.log(objectFromMap);
```

```
// { banana: 1, orange: 2, meat: 4 }

const mapFromObject = new Map(Object.entries(objectFromMap));
console.log(mapFromObject.get("meat")); // 4
```

bind

This example demonstrates how the `bind` method is used to fix the `this` context of a function.

In JavaScript, methods like `setTimeout` lose the original object context when passing a function reference. Using `bind`, we can permanently associate a method with its object (`user`), ensuring `this.firstName` refers to the correct value.

This is useful when passing object methods as callbacks, especially in asynchronous operations.

```
const user = {
  firstName: "John",
  sayHi() {
    console.log(`Hello, ${this.firstName}!`);
  },
};

user.sayHi(); // Hello, John!
setTimeout(user.sayHi, 0); // Hello, undefined!

// solution 1
setTimeout(function () {
  user.sayHi(); // Hello, John!
}, 0);
// or shorter
setTimeout(() => user.sayHi(), 0); // Hello, John!

// solution 2
const sayHi = user.sayHi.bind(user);
// can run it without an object
sayHi(); // Hello, John!
setTimeout(sayHi, 0); // Hello, John!
```

Async Generators

This example demonstrates how to use an `async function*` (asynchronous generator) to yield values over time with a delay. The `generateSequence` function yields numbers from `start` to `end`, waiting one second between each using `await`.

The `for await...of` loop is used to consume the generator asynchronously, making it perfect for streaming data, timers, or controlled iteration over async events.

```
async function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    await new Promise((resolve) => setTimeout(resolve, 1000));
    yield i;
```

```
  }
}

const timer = async (callback) => {
  const generator = generateSequence(1, 5);
  for await (let value of generator) {
    callback(value);
  }
};

timer(console.log).catch(console.error);
// 1, 2, 3, 4, 5 (with delay between)
```

Fibonacci

The Fibonacci sequence is a series where each number is the sum of the two preceding ones: `1, 1, 2, 3, 5, 8, 13, 21, ...`

This example shows two ways to calculate the `n`-th Fibonacci number:

🔁 Iterative Approach

Efficient and suitable for large values of `n` (like 77). Uses a loop and keeps track of the last two values.

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

console.log(fib(3)); // 2
console.log(fib(7)); // 13
console.log(fib(77)); // 5527939700884757
```

🔁 Recursive Approach

Elegant but inefficient for large `n` due to repeated calculations (exponential time complexity). Best for learning recursion.

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

console.log(fib(3)); // 2
```

```
console.log(fib(7)); // 13
console.log(fib(77)); // -
```

JavaScript Event Loop: Microtasks, Macrotasks, and Execution Order

The JavaScript event loop powers how code executes asynchronously. Understanding how the **call stack**, **microtask queue**, and **macrotask queue** interact helps you write better, non-blocking code—and pass those tricky frontend interviews.

Key Concepts Covered:

- **Promise Resolution Order** Understand how `.then()` callbacks are queued as microtasks and run before `setTimeout`.
- **Microtasks vs. Macrotasks** Compare how `Promise.then`, `queueMicrotask`, `setTimeout`, and `requestAnimationFrame` are scheduled and executed.
- **Async/Await Behavior** Explore how `await` impacts execution order relative to synchronous and timed code.
- **Blocking the Event Loop** See how synchronous loops (like `while`) delay the execution of `setTimeout`.
- **Practical Examples** Trace execution order across complex interleavings of Promises and timeouts to build a mental model of how JavaScript scheduling works.

🧵 Promise.all and the Event Loop

`Promise.all` resolves when all input promises (or values) are fulfilled. Non-promise values are treated as already resolved, but their evaluation still runs asynchronously. This snippet also shows how `setTimeout` helps observe the order of event loop execution and when the microtask queue is cleared.

```
const promise1 = Promise.resolve(3);
const promise2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, "foo");
});
const promise3 = 42;

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log({ values });
});

// Using setTimeout, we can execute code after the queue is empty
setTimeout(() => {
  console.log("the queue is now empty");
});

const p3 = Promise.all([]); // Will be immediately resolved
const p4 = Promise.all([1337, "hi"]);

// Non-promise values are ignored, but the evaluation is done asynchronously
console.log({ p3 });
console.log({ p4 });

setTimeout(() => {
```

```
  console.log({ p4 });
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log({ values2: values });
});

const promise4 = Promise.resolve(3);
const promise5 = 42;

Promise.all([promise4, promise5]).then((values) => {
  console.log({ values3: values });
});

// { p3: Promise { [] } }
// { p4: Promise { <pending> } }
// { values3: [ 3, 42 ] }
// the queue is now empty
// { p4: Promise { [ 1337, 'hi' ] } }
// { values: [ 3, 'foo', 42 ] }
// { values2: [ 3, 'foo', 42 ] }
```

🔄 Promise Chaining and Microtask Queue Order

Even though `promise1` and `promise2` are resolved immediately, their `.then()` callbacks are placed in the **microtask queue**. Microtasks are executed in the order they're queued, and each `.then()` forms its own chain — leading to interleaved output.

```
const promise1 = Promise.resolve();
const promise2 = Promise.resolve();

promise1.then(() => console.log(1)).then(() => console.log(2));
promise2.then(() => console.log(3)).then(() => console.log(4));
// 1 3 2 4
```

🕐 `let` in Loops with `setTimeout`

Using `let` in a loop ensures that each iteration captures its own block-scoped value of `i`. All `setTimeout` callbacks execute after 1 second, printing `0` to `3` as expected — thanks to `let`'s scoping behavior.

```
for (let i = 0; i < 4; i++) {
  setTimeout(() => {
    console.log(i);
  }, 1000);
} // 0, 1, 2, 3. All after 1s
```

🔁 Promise Lifecycle and Event Loop Timing

This example shows the synchronous and asynchronous parts of a Promise's lifecycle. The executor function runs immediately, while `.then()` handlers are queued as microtasks and executed after the current call stack clears — before any `setTimeout` callbacks.

```
const promise = new Promise((resolve, reject) => {
  console.log("Promise callback");
  resolve("resolved");
  console.log("Promise callback end");
}).then((result) => {
  console.log("Promise callback (.then)", result);
});

setTimeout(() => {
  console.log("event-loop cycle: Promise (fulfilled)", promise);
}, 0);

console.log("Promise (pending)", promise);

// Promise callback
// Promise callback end
// Promise (pending) Promise { <pending> }
// Promise callback (.then) resolved
// event-loop cycle: Promise (fulfilled) Promise { undefined }
```

⚙️ Async Function and Timer Execution Order

Even when using `await`, `async` functions execute their synchronous parts immediately. This example highlights how `setTimeout` callbacks are deferred to the **macrotask queue**, while synchronous code inside `async` functions runs first.

```
async function run() {
  console.log("run async");
  setTimeout(() => {
    console.log("run timeout");
  }, 0);
}

setTimeout(() => {
  console.log("timeout");
}, 0);

// await or not, same result
await run();

console.log("script");

// run async
// script
// timeout
// run timeout
```

⏳ Blocking the Event Loop with a While Loop

This snippet demonstrates that synchronous code (a busy while loop) can block the event loop. Even though `setTimeout` is set to execute after 500ms, the callback is delayed until the loop finishes—after roughly 2 seconds.

```
const seconds = new Date().getTime() / 1000;

setTimeout(() => {
  // prints out "2", meaning that the callback is not called immediately after 500 milliseconds.
  console.log(`Ran after ${new Date().getTime() / 1000 - seconds} seconds`);
}, 500);

while (true) {
  if (new Date().getTime() / 1000 - seconds >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}

// Good, looped for 2 seconds
// Ran after 2.01 seconds
```

📜 Script, Microtasks, and Macrotasks in Execution Order

This example demonstrates how JavaScript prioritizes synchronous code first, followed by microtasks (`.then` , `queueMicrotask`), and finally macrotasks (`setTimeout`). It reveals the precise order in which the event loop processes each queue.

```
console.log("Script start");

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

Promise.resolve()
  .then(() => {
    console.log("Promise 1");
  })
  .then(() => {
    console.log("Promise 2");
  });

console.log("Script end");

const promise1 = new Promise((resolve, reject) => {
  console.log("Promise constructor");
  resolve();
}).then(() => {
  console.log("Promise constructor resolve");
});

queueMicrotask(() => {
  console.log("Microtask queue");
});

console.log("After Promise constructor");

// Script start
// Script end
```

```
// Promise constructor
// After Promise constructor
// Promise 1
// Promise constructor resolve
// Microtask queue
// Promise 2
// setTimeout
```

🕑 Blocking Inside Async Callbacks

Even though the task is scheduled with `setTimeout`, once it begins, the **long-running synchronous task blocks** the event loop. This illustrates how JavaScript remains single-threaded—even asynchronous calls can't interrupt blocking operations.

```
function longRunningTask() {
  console.log("Start Long-Running Task");

  const startTime = Date.now();
  while (Date.now() - startTime < 2000) {
    // Simulate a long-running task (3 seconds)
  }

  console.log("Long-Running Task Completed");
}

function simulateNonBlocking() {
  console.log("Start");

  setTimeout(() => {
    console.log("Non-blocking Operation");
    longRunningTask();
  }, 0);

  console.log("End");
}

simulateNonBlocking();

// Start
// End
// Non-blocking Operation
// Start Long-Running Task
// after 2s:
// Long-Running Task Completed
```

🍬 Nested Microtasks in Macrotasks

Microtasks (like `.then()`) always run **before** macrotasks (`setTimeout`). Even when a Promise is placed inside a `setTimeout`, its callback becomes a microtask and runs immediately after the current macrotask finishes.

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

setTimeout(() => {
```

```
  console.log("setTimeout 1");
  Promise.resolve().then(() => {
    console.log("Promise inside setTimeout 1");
  });
}, 0);

setTimeout(() => {
  console.log("setTimeout 2");
}, 0);

Promise.resolve()
  .then(() => {
    console.log("Promise 1");
  })
  .then(() => {
    console.log("Promise 2");
  });

console.log("End");

// Start
// End
// Promise 1
// Promise 2
// setTimeout 1
// Promise inside setTimeout 1
// setTimeout 2
```

🎥 requestAnimationFrame and Task Ordering

This example shows how different task types (`setTimeout` , Promises, `requestAnimationFrame`) are prioritized. Microtasks (`.then`) run before macrotasks (`setTimeout`), and `requestAnimationFrame` is queued to run right before the next paint — **after all other queues** are cleared.

```
console.log("1");

setTimeout(function () {
  console.log("2");

  Promise.resolve().then(function () {
    console.log("3");
  });
}, 0);

Promise.resolve().then(function () {
  console.log("4");

  setTimeout(function () {
    console.log("5");
  }, 0);
});

requestAnimationFrame(function () {
  console.log("7");
});

console.log("6");
```

```
// 1, 6, 4, 2, 3, 5, 7
```

JavaScript Object Methods and `this` Behavior

Understanding how `this` works inside objects, functions, and classes is essential for mastering JavaScript.

This guide walks you through the nuances of `this` across different method contexts — including arrow functions, method chaining, `setTimeout`, class methods, and prototype inheritance — with clear code examples and pitfalls to avoid.

It covers:

- Method binding issues when referencing `this`
- The difference between arrow functions and regular methods
- Preserving `this` in callbacks
- Returning `this` for method chaining
- Context behavior in `class` methods and prototype chains

Understanding these patterns is essential for working with objects, classes, and asynchronous code in JavaScript.

object-methods

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    console.log(user.name); // leads to an error
  },
};

const admin = user;
user = null; // overwrite user object

try {
  admin.sayHi();
} catch (e) {
  console.error(e.message);
}
// Cannot read properties of null (reading 'name')

console.log(admin);
// { name: 'John', age: 30, sayHi: [Function: sayHi] }
```

❌ Accessing `user` directly inside a method

Avoid directly referencing the object (`user`) inside its own method. If the object is reassigned or removed, the method will throw an error. Use `this` instead to safely access object properties.

```
let user = {
  name: "John",
```

```
  age: 30,

  sayHi() {
    console.log(this.name);
  },
};

const admin = user;
user = null; // overwrite user object

try {
  admin.sayHi();
} catch (e) {
  console.error(e.message);
}
// John

console.log(admin);
// { name: 'John', age: 30, sayHi: [Function: sayHi] }
```

🔄 Arrow vs Regular Function and `this`

Arrow functions don't have their own `this` — they inherit it from the surrounding scope. In contrast, regular functions define their own `this`, which can lead to unexpected `undefined` values when used as methods.

```
const user = {
  firstName: "John",
  sayHi() {
    const arrow = () => console.log(this.firstName);
    arrow();
  },
};
user.sayHi(); // John

const user2 = {
  firstName: "John",
  sayHi() {
    function normal() {
      console.log(this.firstName);
    }
    normal();
  },
};
user2.sayHi(); // undefined
```

⚠️ `this` in object literals vs methods

When returning an object from a function, `this` does not refer to the object being created — it refers to the function's execution context. To correctly reference the object itself, define `ref` as a method so that `this` is bound to the object at call time.

```
function makeUser() {
  return {
```

```
    name: "John",
    ref: this,
  };
}
const user = makeUser();
console.log(user.ref.name); // undefined

// The value of this is one for the whole function
// ... same as
function makeUser2() {
  return this; // this time there's no object literal
}
console.log(makeUser2().name); // undefined

function makeUser3() {
  return {
    name: "John",
    ref() {
      return this;
    },
  };
}

const user2 = makeUser3();

console.log(user2.ref().name); // John

// Now it works, because user2.ref() is a method.
// And the value of this is set to the object before dot
```

□ Method Chaining with `this`

By returning `this` from each method, you enable **method chaining** — a clean and expressive way to perform multiple operations on the same object in sequence.

```
const ladder = {
  step: 0,
  up() {
    this.step++;
    return this;
  },
  down() {
    this.step--;
    return this;
  },
  showStep: function () {
    // shows the current step
    console.log(this.step);
    return this;
  },
};

ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
ladder.down();
ladder.showStep(); // 0
```

```
ladder.up().up().down().showStep().down().showStep();
// shows 1 then 0
```

🔍 `this` in Regular vs Arrow Methods

Regular functions bind `this` to the object they belong to, while arrow functions inherit `this` from the outer scope — which may lead to unexpected `undefined` or `NaN` values when used as methods.

```
const obj1 = {
  value: 5,
  regularMethod: function () {
    return this?.value + 10;
  },
  arrowMethod: () => {
    return this?.value + 20;
  },
};

console.log(obj1.regularMethod()); // 15
console.log(obj1.arrowMethod()); // NaN
```

⏱️ Preserving `this` in Async Callbacks

In asynchronous functions like `setTimeout`, regular functions lose their `this` binding. Arrow functions capture `this` from their lexical scope, making them ideal for preserving context in async callbacks.

```
const obj2 = {
  value: 50,
  method: function () {
    setTimeout(function () {
      console.log(this.value);
    }, 100);
  },
  methodArrow: function () {
    setTimeout(() => {
      console.log(this.value);
    }, 100);
  },
};
obj2.method(); // undefined
obj2.methodArrow(); // 50
```

🎯 Arrow Functions Preserve Context

Arrow functions retain the `this` value from where they were defined. Even when the method is detached from the object, it still correctly references `this.value` from its original context.

```
function Example() {
  this.value = 30;
  this.arrowMethod = () => {
    console.log(this.value);
  };
}

const example1 = new Example();
example1.arrowMethod(); // 30

const detachedMethod = example1.arrowMethod;
detachedMethod(); // 30
```

⚠️ Regular Functions Lose `this` When Detached

Regular functions bind `this` dynamically at call time. When a method is called standalone (detached from its object), `this` becomes `undefined` (in strict mode), leading to unexpected behavior.

```
function Example() {
  this.value = 30;
  this.regularMethod = function () {
    console.log(this);
  };
}

const example1 = new Example();
example1.regularMethod();
// Example { value: 30, regularMethod: [Function (anonymous)] }

const detachedMethod = example1.regularMethod;
detachedMethod(); // undefined
```

🧭 `call` with Regular vs Arrow Functions

Regular functions can have their `this` explicitly set using `.call()`. Arrow functions, however, do not bind `this` and always inherit it from the surrounding scope — making `.call()` ineffective.

```
const obj5 = {
  value: 25,
  regularMethod: function () {
    return this.value;
  },
  arrowMethod: () => {
    return this;
  },
};

const anotherObj = {
  value: 50,
};
```

```
console.log(obj5.regularMethod()); // 25
console.log(obj5.regularMethod.call(anotherObj)); // 50
console.log(obj5.arrowMethod()); // undefined
console.log(obj5.arrowMethod.call(anotherObj)); // undefined
```

🔄 Arrow Function Captures Outer `this`

When an arrow function is returned from a method, it keeps the `this` of its surrounding context — in this case, the `obj7` object. This allows the inner function to access `obj7.value` even when returned from another object.

```
const obj = {
  value: 70,
  method: function () {
    return {
      getValue: () => {
        return this.value;
      },
    };
  },
};

const innerObj = obj.method();
console.log(innerObj.getValue()); // 70
```

🧩 `this` in Class Methods: Regular vs Arrow

In class instances, regular methods lose `this` when detached, while arrow functions preserve the class context. Arrow methods defined as properties are especially useful when passing methods as callbacks or event handlers.

```
class MyClass {
  value;

  constructor() {
    this.value = 40;
  }

  regularMethod() {
    console.log(this);
  }

  arrowMethod = () => {
    console.log(this);
  };
}

const instance = new MyClass();

instance.regularMethod();
// MyClass { value: 40, arrowMethod: [Function: arrowMethod] }
instance.arrowMethod();
// MyClass { value: 40, arrowMethod: [Function: arrowMethod] }

const regularFn = instance.regularMethod;
```

```
const arrowFn = instance.arrowMethod;

regularFn(); // undefined
arrowFn();
// MyClass { value: 40, arrowMethod: [Function: arrowMethod] }
```

🚗 Inheriting Methods with `Object.create`

Using `Object.create()`, you can create an object that inherits from another. Here, `myCar` inherits the `getInfo` method from `vehicle`, demonstrating how prototype-based inheritance works in JavaScript.

```
const vehicle = {
  getInfo: function () {
    console.log(this.model + " was made in " + this.year);
  },
};

const myCar = Object.create(vehicle);
myCar.model = "BMW";
myCar.year = 2010;
myCar.getInfo(); // BMW was made in 2010
console.log(myCar);
// { model: 'BMW', year: 2010 }
console.log(myCar.__proto__);
// { getInfo: [Function: getInfo] }
```

JavaScript Array Utilities: `chunk`, `compact`, and `partition`

These utility functions are inspired by Lodash and solve common data transformation problems.

chunk

The `_.chunk()` function in Lodash splits an array into smaller arrays, or "chunks," of a specified size. It returns a new array of chunks, making it easier to manage large datasets by dividing them into smaller, more manageable parts.

This example demonstrates a custom implementation of Lodash's `_.chunk()` function to achieve the same result.

```
const chunk = (inputArray, quantity) => {
  return inputArray.reduce((resultArray, item, index) => {
    const chunkIndex = Math.floor(index / quantity);

    if (!resultArray[chunkIndex]) {
      resultArray[chunkIndex] = []; // start a new chunk
    }

    resultArray[chunkIndex].push(item);

    return resultArray;
  }, []);
};

const perChunk = 2; // items per chunk
const inputArray = ["a", "b", "c", "d", "e"];

console.log(chunk(inputArray, perChunk));
// [['a','b'], ['c','d'], ['e']]
```

compact

The `_.compact()` function in Lodash creates an array by removing all falsey values from the provided array. Falsey values include `false`, `null`, `0`, `""`, `undefined`, and `NaN`. This is useful for cleaning up arrays and removing unwanted, invalid values.

This example shows a custom implementation of Lodash's `_.compact()` function to filter out falsey values from the input array.

```
const compact = (inputArray) => inputArray.filter(Boolean);

const inputArray = ["a", false, null, 0, "", undefined, NaN, "d"];
console.log(compact(inputArray)); // [ 'a', 'd' ]
```

partition

The `_.partition()` method splits an array into two groups based on a predicate function. The first group contains elements for which the predicate returns truthy, and the second group contains elements for which the predicate returns falsey. The predicate function is invoked with one argument: the value of the element.

This custom implementation mimics the `_.partition()` method by using the `reduce()` function to iterate over the array and push elements into two arrays based on the result of the predicate.

```
function partition(array, predicate) {
  return array.reduce(
    (result, element) => {
      result[predicate(element) ? 0 : 1].push(element);
      return result;
    },
    [[], []],
  );
}

const users = [
  { user: "barney", age: 36, active: false },
  { user: "fred", age: 40, active: true },
  { user: "pebbles", age: 1, active: false },
];

const [active, inactive] = partition(users, (user) => user.active);
console.log(active);
// [{ 'user': 'fred', 'age': 40, 'active': true }]
console.log(inactive);
// [
//   { user: 'barney', age: 36, active: false },
//   { user: 'pebbles', age: 1, active: false }
// ]
```

JavaScript Set Utilities: `difference`, `differenceBy`, `intersection`, `union`

Reimplement Lodash's most common set operations using native JavaScript. Learn how to compare arrays, remove duplicates, and find overlaps using `Set` and concise functional patterns.

difference

The `_.difference()` method returns the elements from the first array that are not present in any of the other arrays provided. It computes the difference by using the `Set` data structure for efficient lookups.

This custom implementation compares two arrays and finds the elements unique to each array, returning the differences in two parts: one for elements in `arr1` but not in `arr2`, and one for elements in `arr2` but not in `arr1`.

```
const findDifference = function (arr1, arr2) {
  const set1 = new Set(arr1);
  const set2 = new Set(arr2);

  const diffLeft = [];
  const diffRight = [];

  for (const item of set1) {
    if (!set2.has(item)) diffLeft.push(item);
  }

  for (const item of set2) {
    if (!set1.has(item)) diffRight.push(item);
  }

  return [diffLeft, diffRight];
};

const array1 = [2, 1, 3];
const array2 = [2, 3];
const result = findDifference(array1, array2);
console.log(result); // [ [ 1 ], [] ]
```

differenceBy

The `_.differenceBy()` method works like `difference()`, but allows you to compare elements by a specific property or transformation function. You pass a key or iteratee to compare the objects by their properties.

In this custom implementation, we compare objects in `arr1` and `arr2` by a key (`x`), and return the differences based on that key.

```
const differenceBy = (arr1, arr2, key) => {
  const set2 = new Set(arr2.map((item) => item[key]));
  const set1 = new Set(arr1.map((item) => item[key]));
```

```
  const diffLeft = [];
  const diffRight = [];

  for (const item of arr1) {
    if (!set2.has(item[key])) {
      diffLeft.push(item);
    }
  }

  for (const item of arr2) {
    if (!set1.has(item[key])) {
      diffRight.push(item);
    }
  }

  return [diffLeft, diffRight];
};

const array1 = [{ x: 2 }, { x: 1 }, { x: 1 }];
const array2 = [{ x: 1 }];
const result = differenceBy(array1, array2, "x");
console.log(result); // [ [ { x: 2 } ], [] ]
```

intersection

The `_.intersection()` method computes the intersection of two or more arrays. It returns a new array containing the elements that are present in all provided arrays.

This custom implementation compares two arrays by converting them to `Set` objects for efficient lookup and finding the common elements between them.

```
const intersection = function (nums1, nums2) {
  const set1 = new Set(nums1);
  const set2 = new Set(nums2);
  const result = [];

  for (const nums of set2) {
    if (set1.has(nums)) {
      result.push(nums);
    }
  }

  return result;
};

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

union

The `_.union()` method creates a new array of unique values by combining all given arrays, preserving the order of elements.

This custom implementation utilizes the `Set` object to remove duplicates and `concat()` to merge all arrays before applying `Array.from()` to convert the set back into an array.

```
const union = (...arrays) => {
  return Array.from(new Set([].concat(...arrays)));
};

console.log(union([1, 2], [1, 7, 7], [3, 1]));
// [ 1, 2, 7, 3 ]
```

JavaScript Object Utilities: `keyBy`, `omit`, `orderBy`, `pick`, `curry`

This guide covers powerful utility functions inspired by Lodash — including ways to transform objects, sort data, extract or exclude fields, and write curried functions — all with custom JavaScript implementations.

keyBy

The `_.keyBy()` method creates an object composed of keys generated from the results of running each element in the collection through an iteratee function. The iteratee can be a function or property name.

This custom implementation maps each item in the collection to a key determined by the provided iteratee, and then assigns the item to that key in the result object.

```
function keyBy(collection, iteratee) {
  const result = {};

  for (const item of collection) {
    const key =
      typeof iteratee === "function" ? iteratee(item) : item[iteratee];
    result[key] = item;
  }

  return result;
}

const array = [
  { dir: "left", code: 97 },
  { dir: "right", code: 100 },
];

const res1 = keyBy(array, ({ code }) => String.fromCharCode(code));
console.log(res1);
// { a: { dir: 'left', code: 97 }, d: { dir: 'right', code: 100 } }

const res2 = keyBy(array, "dir");
console.log(res2);
// { left: { dir: 'left', code: 97 }, right: { dir: 'right', code: 100 } }
```

omit

The `_.omit()` method creates a shallow copy of an object without the specified properties. The properties to omit can be provided as a single key or an array of keys.

This custom implementation removes the given properties from the object by first creating a copy and then deleting the specified keys.

```
function omit(obj, keys) {
  const result = { ...obj };
```

```
  if (!Array.isArray(keys)) {
    delete result[keys];
    return result;
  }

  for (const key of keys) {
    delete result[key];
  }

  return result;
}

const user = {
  name: "Alice",
  age: 25,
  email: "alice@example.com",
  city: "Wonderland",
};

const omitted = omit(user, "age");
console.log(omitted);
// { name: 'Alice', email: 'alice@example.com', city: 'Wonderland' }

const omitted2 = omit(user, ["age", "email"]);
console.log(omitted2);
// { name: 'Alice', city: 'Wonderland' }
```

orderBy

The `_.orderBy()` method sorts an array of objects based on one or more properties. You can specify the sort order for each property as either ascending (`"asc"`) or descending (`"desc"`).

This custom implementation replicates the `_.orderBy()` method by accepting an array, a property to sort by, and an optional order parameter (`"asc"` by default). The array is first copied to avoid mutation, then sorted based on the specified property and order.

```
function orderBy(array, property, order = "asc") {
  const multiplier = order === "asc" ? 1 : -1;
  const copy = [...array];

  return copy.sort((a, b) => {
    if (a[property] < b[property]) return -1 * multiplier;
    if (a[property] > b[property]) return 1 * multiplier;
    return 0;
  });
}

const users = [
  { user: "barney", age: 36 },
  { user: "fred", age: 40 },
  { user: "pebbles", age: 1 },
];

const sortedByAgeAsc = orderBy(users, "age", "asc");
console.log(sortedByAgeAsc);
// [
//   { user: 'pebbles', age: 1 },
//   { user: 'barney', age: 36 },
```

```
//   { user: 'fred', age: 40 }
// ]

const sortedByUserDesc = orderBy(users, "user", "desc");
console.log(sortedByUserDesc);
// [
//   { user: 'pebbles', age: 1 },
//   { user: 'fred', age: 40 },
//   { user: 'barney', age: 36 }
// ]
```

pick

The `_.pick()` method creates a new object by picking the specified properties from an existing object. It accepts either a single key or an array of keys to extract.

This custom implementation mimics the `_.pick()` method by using `reduce()` to iterate over the array of keys and select only the properties that exist in the source object, returning a new object with the selected properties.

```
function pick(obj, keys) {
  if (typeof keys === "string") {
    return obj[keys] !== undefined ? { [keys]: obj[keys] } : {};
  }

  return (Array.isArray(keys) ? keys : []).reduce((result, key) => {
    if (key in obj) {
      result[key] = obj[key];
    }
    return result;
  }, {});
}

const user = {
  name: "Alice",
  age: 25,
  email: "alice@example.com",
  city: "Wonderland",
};

const picked = pick(user, ["name", "email"]);
console.log(picked);
// { name: 'Alice', email: 'alice@example.com' }
```

curry

The `_.curry()` function in Lodash transforms a function into a curried version that can be called with a series of partial arguments. Instead of calling the function with all arguments at once, you can call it progressively, one argument at a time.

This example shows a custom implementation of currying for the `sum` function, allowing it to be invoked with one argument at a time or multiple arguments in a sequence.

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn(...args);
    } else {
      return (...nextArgs) => curried(...args, ...nextArgs);
    }
  };
}

const sum = (a, b, c) => a + b + c;
const curriedSum = curry(sum);

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

React Concepts: `usePrevious` and `children`

This guide explores useful React patterns including the `usePrevious` hook for tracking prior state values, and how React handles JSX `children` and component reuse — all with minimal, focused examples using TypeScript.

usePrevious

The `usePrevious` custom hook allows you to track the previous value of a state variable in React. This hook leverages the `useRef` hook to store the previous value of a given state, and `useEffect` to update the ref value after the component has rendered. This approach makes it possible to access the value before the current one, which is useful for cases where you need to compare the previous and current values of a variable.

In this example, the `usePrevious` hook tracks the previous count state value. The value of the `ref` is updated inside the `useEffect`, which runs after the component has rendered. The previous value is stored in the ref and is returned by the hook.

```
import { useState, useEffect, useRef } from "react";

function usePrevious(value: string | number) {
  const ref = useRef<string | number>();

  useEffect(() => {
    ref.current = value;
    console.log("useEffect", ref.current);
    return () => {
      console.log("useEffect return");
    };
  }, [value]);

  console.log({ ref: ref.current });

  return ref.current;
}

function App() {
  const [count, setCount] = useState(0);
  console.log({ count });
  const previousCount = usePrevious(count);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Previous count: {previousCount}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default App;
// initial render logs

// {count: 0}
// {ref: undefined}
// useEffect 0
// useEffect return
// useEffect 0
```

```
// consequent renders log

// {count: 1}
// {ref: 0}
// useEffect return
// useEffect 1
```

children

The `children` concept in React refers to any child components or elements passed into a parent component. In this example, the `TestComponent` is rendered by the `TestWrapper` component, but React does not rerender `TestComponent` after clicking the button because the props have not changed. React reuses the same component instance as long as its props remain unchanged.

This custom implementation demonstrates how React optimizes rendering by not rerendering `TestComponent` unless the props passed to it change, despite the state change in the parent component (`TestWrapper`).

```
import { ReactElement, useState } from "react";

export default function App() {
  return <TestWrapper prop={<TestComponent />} />;
}

function TestWrapper({ prop }: { prop: ReactElement }) {
  const [bool, setBool] = useState(false);
  console.log("TestWrapper");
  return (
    <>
      <button onClick={() => setBool(!bool)}>click</button>
      {prop}
    </>
  );
}

function TestComponent() {
  console.log("TestComponent");
  return <></>;
}

// initial render
// TestWrapper
// TestComponent

// after click
// TestWrapper
```

React State Patterns: Normalized State and `useReducer`

Managing state in complex React applications can be simplified using two powerful patterns: **normalized state** and the `useReducer` **hook**. This guide demonstrates how to structure deeply nested data for easier updates and how to use reducers to manage state transitions cleanly and predictably.

Normalized State

The concept of **normalized state** is commonly used in managing complex data structures, such as nested objects or arrays, in a way that makes it easier to update and manage state in applications. This approach involves flattening the structure into a more accessible format, often using an object with unique IDs as keys, and maintaining relationships between the entities via arrays of IDs (as seen in `childIds`).

In this example, the `TravelPlanComponent` component simulates a travel planning system where each location (node) has a title and a set of child locations (nested nodes). The `normalized state` design keeps all the locations in a flat object, and relationships between locations are managed using the `childIds` array.

The `removeNodeAndChildren` function recursively removes a node and its children from the state, and `handleComplete` ensures that a child node is removed from its parent node's `childIds`. This architecture makes it easier to update and delete nodes or manage complex state in React applications, especially with deep nested structures.

```
import { useState } from "react";

type TravelNode = {
  id: number;
  title: string;
  childIds: number[];
};

type TravelPlan = {
  [key: number]: TravelNode;
};

const initialTravelPlan: TravelPlan = {
  0: { id: 0, title: "(Root)", childIds: [1, 42] },
  1: { id: 1, title: "Earth", childIds: [2, 10] },
  2: { id: 2, title: "Africa", childIds: [3, 4, 5] },
  3: { id: 3, title: "Botswana", childIds: [] },
  4: { id: 4, title: "Egypt", childIds: [] },
  5: { id: 5, title: "Kenya", childIds: [] },
  10: { id: 10, title: "Americas", childIds: [11, 12, 13] },
  11: { id: 11, title: "Argentina", childIds: [] },
  12: { id: 12, title: "Brazil", childIds: [] },
  13: { id: 13, title: "Barbados", childIds: [] },
  42: { id: 42, title: "Moon", childIds: [43, 44, 45] },
  43: { id: 43, title: "Rheita", childIds: [] },
  44: { id: 44, title: "Piccolomini", childIds: [] },
  45: { id: 45, title: "Tycho", childIds: [] },
};

export default function TravelPlanComponent() {
  const [travelPlan, setTravelPlan] = useState<TravelPlan>(initialTravelPlan);

  const removeNodeAndChildren = (id: number, plan: TravelPlan): TravelPlan => {
    const newPlan = { ...plan };
```

```
    const removeRecursively = (nodeId: number) => {
      const node = newPlan[nodeId];
      if (!node) return;

      // Remove all children recursively
      node.childIds.forEach(removeRecursively);

      // Delete the current node
      delete newPlan[nodeId];
    };

    removeRecursively(id);
    return newPlan;
  };

  const handleComplete = (id: number, parentId: number) => {
    setTravelPlan((prevPlan) => {
      const updatedPlan = removeNodeAndChildren(id, prevPlan);

      // Remove the child from the parent's childIds array

      updatedPlan[parentId] = {
        ...updatedPlan[parentId],
        childIds: updatedPlan[parentId].childIds.filter(
          (childId) => childId !== id,
        ),
      };

      return updatedPlan;
    });
  };

  return (
    <div>
      {travelPlan[0].childIds.map((childId) => (
        <RenderPlace
          key={childId}
          id={childId}
          parentId={travelPlan[0].id}
          travelPlan={travelPlan}
          handleComplete={handleComplete}
        />
      ))}
    </div>
  );
}

const RenderPlace = ({
  id,
  parentId,
  handleComplete,
  travelPlan,
}: {
  id: number;
  parentId: number;
  handleComplete: (id: number, parentId: number) => void;
  travelPlan: TravelPlan;
}) => {
  const node = travelPlan[id];

  if (!node) return null;

  return (
    <ol style={{ paddingInlineStart: 0 }}>
      <li
```

```
          style={{
            display: "flex",
            gap: "1rem",
            alignItems: "center",
          }}
        >
          <h5
            style={{
              marginBlockStart: 5,
              marginBlockEnd: 5,
            }}
          >
            {node.title}
          </h5>
          <button onClick={() => handleComplete(node.id, parentId)}>
            Complete
          </button>
        </li>
        {node.childIds.length > 0 && (
          <ol>
            {node.childIds.map((childId) => (
              <li key={childId}>
                <RenderPlace
                  id={childId}
                  parentId={node.id}
                  travelPlan={travelPlan}
                  handleComplete={handleComplete}
                />
              </li>
            ))}
          </ol>
        )}
      </ol>
    );
  };
```

useReducer

The `useReducer` hook is a more advanced alternative to `useState` for managing complex state logic in React. It allows you to handle state transitions using a reducer function, similar to how Redux works. `useReducer` is especially useful when the state depends on previous state values or involves more complex state updates.

In this example, `useReducer` is used to manage a collection of messages. The `messengerReducer` function defines the logic for adding and removing messages. The reducer receives the current state and an action, then returns the updated state based on the action type.

The `useReducer` hook returns the current state and a dispatch function to trigger state updates. The state is updated when a button is clicked, either to add a new message or remove the last message.

```
import { useState } from "react";

type Messages = Record<number, string>;

type Action =
  | { type: "add_message"; message: string }
  | { type: "remove_last_message" };

const initialState: Messages = {
```

```
  0: "Hello, Taylor",
  1: "Hello, Alice",
  2: "Hello, Bob",
};

function useReducer<S, A>(
  reducer: (state: S, action: A) => S,
  initialState: S,
): [S, (action: A) => void] {
  const [state, setState] = useState<S>(initialState);

  function dispatch(action: A) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}

function messengerReducer(state: Messages, action: Action): Messages {
  switch (action.type) {
    case "add_message": {
      const nextId = Math.max(0, ...Object.keys(state).map(Number)) + 1;
      return {
        ...state,
        [nextId]: action.message,
      };
    }
    case "remove_last_message": {
      const ids = Object.keys(state).map(Number);
      if (ids.length === 0) return state;
      const lastId = Math.max(...ids);
      // eslint-disable-next-line @typescript-eslint/no-unused-vars
      const { [lastId]: _, ...rest } = state;
      return rest;
    }
    default:
      throw new Error("Unknown action: " + (action as Action).type);
  }
}

export default function App() {
  const [messages, dispatch] = useReducer<Messages, Action>(
    messengerReducer,
    initialState,
  );

  return (
    <div>
      <ul>
        {Object.entries(messages).map(([id, message]) => (
          <li key={id}>{message}</li>
        ))}
      </ul>
      <button
        onClick={() =>
          dispatch({ type: "add_message", message: "New Message" })
        }
      >
        Add Message
      </button>
      <button onClick={() => dispatch({ type: "remove_last_message" })}>
        Remove Last Message
      </button>
    </div>
```

```
    );
  }
```

Redux and Twitter Architecture Patterns in Vanilla JavaScript

Explore two powerful patterns implemented from scratch in plain JavaScript: a Redux-like state management system and a Twitter-inspired social feed engine. These examples give you a deeper understanding of real-world app architecture—without relying on any libraries or frameworks.

Redux Pattern Implementation

An implementation of the **Redux pattern** using vanilla JavaScript. This custom `Store` class manages state using a reducer and supports dispatching actions and subscribing to state changes.

Redux is commonly used for predictable state management in frontend applications like React.

```
class Store {
  constructor(reducer, initialState) {
    this.reducer = reducer;
    this.state = initialState;
    this.listeners = [];
  }

  getState() {
    return this.state;
  }

  dispatch(action) {
    this.state = this.reducer(this.state, action);
    this.listeners.forEach((listener) => listener());
  }

  subscribe(listener) {
    this.listeners.push(listener);
    return () => {
      console.log("unsubscribed");
      this.listeners = this.listeners.filter((l) => l !== listener);
    };
  }
}

// Example reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { ...state, count: state.count + 1 };
    case "DECREMENT":
      return { ...state, count: state.count - 1 };
    default:
      return state;
  }
};

const initialState = { count: 0 };

const store = new Store(reducer, initialState);

// Subscribe to state changes
```

```
const unsubscribe = store.subscribe(() => {
  console.log(store.getState());
});

// Dispatch actions
console.log(store.getState());
store.dispatch({ type: "INCREMENT" });
store.dispatch({ type: "INCREMENT" });
store.dispatch({ type: "DECREMENT" });

// Unsubscribe from state changes
unsubscribe();

// { count: 0 }
// { count: 1 }
// { count: 2 }
// { count: 1 }
// unsubscribed
```

Twitter Feed Simulation

A simplified simulation of Twitter's core features using JavaScript classes. Implements functionality for posting tweets, following/unfollowing users, and generating personalized news feed.

Key features:

- `postTweet(userId, tweetId)` – Posts a tweet for a user.
- `follow(followerId, followeeId)` – Allows one user to follow another.
- `unfollow(followerId, followeeId)` – Allows a user to unfollow someone (except themselves).
- `getNewsFeed(userId)` – Returns the 10 most recent tweets from the user and users they follow.

Demonstrates how to work with `Map`, `Set`, arrays, and sorting for feed generation.

```
class Twitter {
  constructor() {
    this.follows = new Map();
    this.tweets = new Map();
    this.timestamp = 0;
  }

  postTweet(userId, tweetId) {
    if (!this.tweets.has(userId)) {
      this.tweets.set(userId, []);
    }
    this.tweets.get(userId).unshift({ tweetId, timestamp: this.timestamp++ });

    if (!this.follows.has(userId)) {
      this.follows.set(userId, new Set());
      // User follows themselves
      this.follows.get(userId).add(userId);
    }
  }

  getNewsFeed(userId) {
    if (!this.follows.has(userId)) return [];

    const followed = this.follows.get(userId);
```

```
    const feed = [];

    for (const followeeId of followed) {
      if (this.tweets.has(followeeId)) {
        feed.push(...this.tweets.get(followeeId));
      }
    }

    feed.sort((a, b) => b.timestamp - a.timestamp);
    return feed.map((tweet) => tweet.tweetId);
  }

  follow(followerId, followeeId) {
    if (!this.follows.has(followerId)) {
      this.follows.set(followerId, new Set());
      // User follows themselves
      this.follows.get(followerId).add(followerId);
    }
    this.follows.get(followerId).add(followeeId);
  }

  unfollow(followerId, followeeId) {
      // Cannot unfollow oneself
    if (followerId === followeeId) return;
    if (this.follows.has(followerId)) {
      this.follows.get(followerId).delete(followeeId);
    }
  }
}

const twitter = new Twitter();

// User 1 posts a tweet with ID 5
twitter.postTweet(1, 5);

// User 1 retrieves their news feed (should contain only tweet 5)
console.log(twitter.getNewsFeed(1)); // [5]

// User 1 follows User 2
twitter.follow(1, 2);

// User 2 posts a tweet with ID 6
twitter.postTweet(2, 6);

// User 1 retrieves their news feed (should contain tweets 6 and 5, most recent first)
console.log(twitter.getNewsFeed(1)); // [6, 5]

// User 1 unfollows User 2
twitter.unfollow(1, 2);

// User 1 retrieves their news feed (should only contain their own tweet, 5)
console.log(twitter.getNewsFeed(1)); // [5]
```

Queue and Stack Patterns in JavaScript

Queue and stack data structures with real JavaScript implementations. This page includes examples of synchronous and asynchronous queues, stack, stack-based palindrome checking.

AsyncQueue: Processing Promises Sequentially

An `AsyncQueue` class that processes asynchronous tasks in sequence. Each task is a function returning a Promise, and tasks are executed one after another in the order they were added.

```
class AsyncQueue {
  constructor() {
    this.queue = [];
  }

  enqueue(task) {
    if (typeof task !== "function") {
      throw new Error("Task must be a function returning a promise");
    }
    this.queue.push(task);
  }

  async process() {
    const results = [];

    for (const task of this.queue) {
      const result = await task();
      results.push(result);
    }

    return results;
  }
}

const queue = new AsyncQueue();
queue.enqueue(
  () =>
    new Promise((resolve) => {
      setTimeout(() => resolve("Task 1"), 1000);
    }),
);
queue.enqueue(
  () =>
    new Promise((resolve) => {
      setTimeout(() => resolve("Task 2"), 500);
    }),
);

queue.process().then((results) => console.log(results));
// ['Task 1', 'Task 2']
```

Queue Implementation: First-In, First-Out (FIFO)

A simple `Queue` class that follows the **FIFO (First In, First Out)** principle. Elements are added to the end and removed from the front.

Commonly used in scheduling, task queues, and breadth-first search algorithms.

```
class Queue {
  constructor() {
    this.collection = [];
  }

  enqueue(element) {
    this.collection.push(element);
    return this;
  }
  dequeue() {
    this.collection.shift();
    return this;
  }
  front() {
    return this.collection[0];
  }
  get size() {
    return this.collection.length;
  }
  isEmpty() {
    return this.collection.length === 0;
  }
}

const queue = new Queue();
queue.enqueue("one").enqueue("two");
// [ 'one', 'two' ]
queue.dequeue(); // [ 'two' ]
queue.enqueue("three").enqueue("four");
// [ 'two', 'three', 'four' ]
queue.front(); // two
queue.size; // 3
queue.isEmpty(); // false
```

Using a Stack to Check for Palindromes

This function uses a **stack** to check if a given word is a palindrome. It pushes each character onto the stack, then pops them in reverse order to compare with the original string.

```
function isPalindrome(word) {
  const letters = []; // this is our stack
  let reversedWord = "";

  // put letters of word into stack
  for (let i = 0; i < word.length; i++) {
    letters.push(word[i]);
  }

  // pop off the stack in reverse order
  for (let i = 0; i < word.length; i++) {
    reversedWord += letters.pop();
  }
```

```
  return reversedWord === word;
}

console.log(isPalindrome("racecar")); // true
console.log(isPalindrome("cheatsheet")); // false
```

Stack Implementation: Last-In, First-Out (LIFO)

A basic `Stack` class implementation that follows **LIFO (Last In, First Out)** behavior. Includes common methods such as `push`, `pop`, `peek`, and `size`.

Useful for:

- Undo functionality
- Recursion
- Reversing data

```
class Stack {
  constructor() {
    this.storage = [];
  }

  push(value) {
    this.storage.push(value);
    return this;
  }

  pop() {
    this.storage.pop();
    return this;
  }

  size() {
    return this.storage.length;
  }

  peek() {
    return this.storage[this.storage.length - 1];
  }
}

const myStack = new Stack();

myStack.push(1).push(2).push(3);
console.log(myStack.peek()); // 3
myStack.pop();
console.log(myStack.peek()); // 2
myStack.push("js-cheatsheet");
console.log(myStack.size()); // 3
console.log(myStack.peek()); // js-cheatsheet
```

Composition vs Inheritance in JavaScript

Understand the trade-offs between **composition** and **inheritance** in JavaScript and React development. This guide walks through practical examples of function composition, class inheritance, and the JavaScript call stack to help you build cleaner, more flexible systems.

composition

Composition vs Inheritance People sometimes say “composition” when contrasting it with inheritance. This has less to do with functions (which we’ve been discussing all along) and more to do with objects and classes — that is, with traditional object-oriented programming.

In particular, if you express your code as classes, it is tempting to reuse behavior from another class by extending it (inheritance). However, this makes it somewhat difficult to adjust the behavior later. For example, you may want to similarly reuse behavior from another class, but you can’t extend more than one base class.

Sometimes, people say that inheritance “locks you into” your first design because the cost of changing the class hierarchy later is too high. When people suggest composition is an alternative to inheritance, they mean that instead of extending a class, you can keep an instance of that class as a field. Then you can “delegate” to that instance when necessary, but you are also free to do something different.

Function composition is a powerful concept, but it raises the level of abstraction and makes your code less direct. If you write your code in a style that composes functions in some way before calling them, and there are other humans on your team, make sure that you’re getting concrete benefits from this approach. It is not “cleaner” or “better”, and there is a price to pay for “beautiful” but indirect code.

composition article

```
const dateFunc = () => new Date();
const textFunc = (date) => date.toDateString();
const labelFunc = (text) => `Today ${text}`;
const showLabelFunc = (label) => console.log(label);

const date = dateFunc();
const text = textFunc(date);
const label = labelFunc(text);
showLabelFunc(label); // Today Sat Sep 28 2024

function pipe(...steps) {
  return function runSteps() {
    let result;
    for (let i = 0; i < steps.length; i++) {
      let step = steps[i];
      result = step(result);
    }
    return result;
  };
}

const showDateLabel = pipe(dateFunc, textFunc, labelFunc, showLabelFunc);
showDateLabel(); // Today Sat Sep 28 2024
```

inheritance

```
// Base class
class Vehicle {
  private readonly _make: string;
  private readonly _model: string;
  private readonly _year: number;

  constructor(make: string, model: string, year: number) {
    this._make = make;
    this._model = model;
    this._year = year;
  }

  displayInfo(): string {
    return `${this._year} ${this._make} ${this._model}`;
  }
}

// Derived class
class Car extends Vehicle {
  private readonly _doors: number;

  constructor(
    make: string,
    model: string,
    year: number,
    doors: number,
  ) {
    super(make, model, year); // Call the constructor of the base class
    this._doors = doors;
  }

  displayInfo(): string {
    return `${super.displayInfo()} - ${this._doors} doors`;
  }
}

const vehicle = new Vehicle("Toyota", "Corolla", 2020);
console.log(vehicle.displayInfo()); // 2020 Toyota Corolla

const car = new Car("Honda", "Civic", 2022, 4);
console.log(car.displayInfo()); // 2022 Honda Civic - 4 doors
```

Function Stack

The **call stack** is a critical concept in JavaScript that keeps track of function calls. When a function is called, it is "pushed" onto the stack. Once the function finishes execution, it is "popped" off the stack. This stack-based approach is essential for understanding recursion and function execution order in JavaScript.

In the **forward phase**, the function calls accumulate as the recursion continues, with each new function invocation pushing itself onto the stack. In the **backward phase**, as functions start to return, they unwind and are popped from the stack, completing their execution.

This behavior is demonstrated in the following example, where `foo` is called recursively until it hits the base case, then the stack is unwound as each function completes.

Forward Phase (Pushing):

1. Call `foo(2)` -> Stack: `[foo(2)]`.
2. Call `foo(1)` -> Stack: `[foo(2), foo(1)]`.
3. Call `foo(0)` -> Stack: `[foo(2), foo(1), foo(0)]`.
4. Call `foo(-1)` -> Stack: `[foo(2), foo(1), foo(0), foo(-1)]`.

Backward Phase (Unwinding):

1. Return from `foo(-1)` -> Stack: `[foo(2), foo(1), foo(0)]`.
2. Complete `foo(0)` -> Stack: `[foo(2), foo(1)]`.
3. Complete `foo(1)` -> Stack: `[foo(2)]`.
4. Complete `foo(2)` -> Stack: `[]`.

```
function foo(i) {
  if (i < 0) {
    return;
  }
  console.log(`begin: ${i}`);
  foo(i - 1);
  console.log(`end: ${i}`);
}
foo(2);

// begin: 2
// begin: 1
// begin: 0
// end: 0
// end: 1
// end: 2
```

SOLID in React: Clean Architecture

Clean architecture isn't just for backend systems — it applies to frontend and React as well. By following the SOLID principles, you'll write components that are easier to test, extend, and maintain — all while avoiding common pitfalls in growing codebases.

- **Single responsibility principle** - Class has one job to do. Each change in requirements can be done by changing just one class
- **Open/closed principle** - Class is happy (open) to be used by others. Class is not happy (closed) to be changed by others
- **Liskov substitution principle** - Class can be replaced by any of its children. Children classes inherit parent's behaviours
- **Interface segregation principle** - When classes promise each other something, they should separate these promises (interfaces) into many small promises, so it's easier to understand
- **Dependency inversion principle** - When classes talk to each other in a very specific way, they both depend on each other to never change. Instead, classes should use promises (interfaces, parents), so classes can change as long as they keep the promise

Single responsibility principle

Don't create too big components that have too much jobs to do. Break into smaller ones, name = description. Every component has one responsibility, first render card, second contains button that calls the dialog component

```
import { useState } from 'react'
// bad
const UserCard = ({ user }: { user: User }) => {
  const [open, setOpen] = useState(false)

  return (
    <>
      <Dialog
        fullWidth
        maxWidth="md"
        open={open}
        onClose={() => setOpen(false)}
      >
        <EditUserDialog id={user.id} handleEditClose={() => setOpen(false)} />
      </Dialog>
      <Button onClick={() => setOpen(true)}>edit user</Button>
      <Box>...here goes user card</Box>
    </>
  )
}

// good
const UserCard = ({ user }: { user: User }) => {

  return (
    <>
      <EditUser id={user.id} />
      <Box>...here goes user card</Box>
    </>
  )
}
```

```
const EditUser = ({ id }: { id: number }) => {
    const [open, setOpen] = useState(false)

  return (
    <>
      <Dialog
        fullWidth
        maxWidth="md"
        open={open}
        onClose={() => setOpen(false)}
      >
        <EditUserDialog id={id} handleEditClose={() => setOpen(false)} />
      </Dialog>
      <Button>edit user</Button>
    </>
  )
}
```

Open/closed principle

Problem with first components is that we can't pass another color without changing `FancyIconButton` code. In the second case, this problem is fixed, now we don't need to change `FancyIconButton2`. So, it's open to be used by others, closed to modification

```
// bad
const FancyIconButton = ({
  red,
  green,
}: {
  red?: boolean
  green?: boolean
}) => {
  const getBackgroundColor = () => {
    switch (true) {
      case red:
        return 'red'
      case green:
        return 'green'
      default:
        return 'black'
    }
  }

  return (
    <>
      <IconButton
        color="secondary"
        sx={{ backgroundColor: getBackgroundColor() }}
      >
        <ArrowBackTwoToneIcon />
      </IconButton>
    </>
  )
}

// good
const FancyIconButton2 = ({
  backgroundColor = 'black',
```

```
}: {
  backgroundColor: string
}) => {
  return (
    <>
      <IconButton color="secondary" sx={{ backgroundColor }}>
        <ArrowBackTwoToneIcon />
      </IconButton>
    </>
  )
}
```

Liskov substitution principle

`FancyIconButton` can be replaced by `IconButton`. Children component inherits parent's behaviours. This way we can have many buttons with different styles, but they all can be used same way as original `IconButton`

```
import { IconButton, IconButtonProps } from '@mui/material'

const fancyStyles = {}
// bad
const FancyIconButton = () => {
  return (
    <>
      <IconButton sx={{ ...fancyStyles }}>
        <ArrowBackTwoToneIcon />
      </IconButton>
    </>
  )
}

// good
const FancyIconButton2 = ({ ...props }: IconButtonProps) => {
  return (
    <>
      <IconButton {...props} sx={{ ...fancyStyles }}>
        <ArrowBackTwoToneIcon />
      </IconButton>
    </>
  )
}

const RandomComponent = () => {
  return (
    <>
      {/*error no such props*/}
      <FancyIconButton size={'small'} />

      {/*here ok*/}
      <FancyIconButton2 size={'small'} />
      <IconButton onClick={() => null} />
    </>
  )
}
```

Interface segregation principle

When classes promise each other something, they should separate these promises (interfaces) into many small promises, so it's easier to understand

```
import { FC } from "react";

interface ButtonProps {
  onClick: () => void;
  onMouseOver: () => void;
}

const SubmitButton: FC<ButtonProps> = ({ onClick, onMouseOver }) => {
  return (
    <button onClick={onClick} onMouseOver={onMouseOver}>
      Submit
    </button>
  );
};
// This forces `SubmitButton` to support all event handlers, even if not all are needed.

// Good
interface Clickable {
  onClick: () => void;
}

interface Hoverable {
  onMouseOver: () => void;
}

const ClickableButton: FC<Clickable> = ({ onClick }) => {
  return <button onClick={onClick}>Submit</button>;
};

const HoverableDiv: FC<Hoverable> = ({ onMouseOver }) => {
  return <div onMouseOver={onMouseOver}>Hover over me!</div>;
};

const SubmitButton2: FC<Clickable & Hoverable> = ({ onClick, onMouseOver }) => {
  return (
    <button onClick={onClick} onMouseOver={onMouseOver}>
      Submit
    </button>
  );
};

const App = () => {
  return (
    <div>
        <h1>Interface segregation in React</h1>
      {/* both are same */}
      <SubmitButton onClick={()=>null} onMouseOver={()=>null} />
      <SubmitButton2 onClick={()=>null} onMouseOver={()=>null} />
      {/* separated */}
      <ClickableButton onClick={()=>null} />
      <HoverableDiv onMouseOver={()=>null} />
    </div>
  );
};

export default App;
```

Dependency inversion principle

When classes talk to each other in a very specific way, they both depend on each other to never change. Instead, classes should use promises (interfaces, parents), so classes can change as long as they keep the promise

```
import { FC } from "react";

// Abstraction
interface AuthProvider {
  login: (username: string, password: string) => Promise<boolean>;
  logout: () => void;
}

// High-level module (React component)
const AuthButton: FC<{ authProvider: AuthProvider }> = ({ authProvider }) => {
  const handleLogin = async () => {
    const success = await authProvider.login("user", "password");
    if (success) {
      console.log("Logged in successfully");
    } else {
      console.error("Login failed");
    }
  };

  const handleLogout = () => {
    authProvider.logout();
    console.log("Logged out");
  };

  return (
    <div>
      <button onClick={handleLogin}>Login</button>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
};

// Implementation of the abstraction
class FirebaseAuthProvider implements AuthProvider {
  async login(username: string, password: string): Promise<boolean> {
    // Simulate Firebase API call
    console.log(`Logging in with Firebase: ${username}`);
    return username === "user" && password === "password"; // Simulate success for demo
  }

  logout(): void {
    console.log("Logging out with Firebase");
  }
}

// Another implementation
class LocalAuthProvider implements AuthProvider {
  async login(username: string, password: string): Promise<boolean> {
    console.log(`Logging in locally: ${username}`);
    return username === "admin" && password === "1234"; // Simulate success for demo
  }

  logout(): void {
    console.log("Logging out locally");
  }
}
```

```
// Using the component
const App = () => {
  // Use any implementation of the abstraction
  const authProvider = new FirebaseAuthProvider();
  const localProvider = new LocalAuthProvider();

  return (
    <div>
      <h1>Dependency Inversion in React</h1>
      <AuthButton authProvider={authProvider} />
      <AuthButton authProvider={localProvider} />
    </div>
  );
};

export default App;
```

JavaScript Type Conversions and Coercion Rules

JavaScript automatically converts values between types in many situations — a behavior known as **type coercion**. This guide helps you master how operators like `+`, `-`, comparisons (`<`, `>`, `==`, `===`), and more behave when mixing strings, numbers, booleans, `null`, and `undefined`.

Operator precedence

plus

Concatenates strings

```
const log = (value) => console.log(value);

log(null + 1); // 1
log(undefined + 1); // NaN
log(false + 1); // 1
log(true + 1); // 2
log("" + 1); // "1"
log("s" + 1); // "s1"
log("1" + 1); // "11"

log(1 + null); // 1
log(1 + undefined); // NaN
log(1 + false); // 1
log(1 + true); // 2
log(1 + ""); // "1"
log(1 + "s"); // "1s"
log(1 + "1"); // "11"
```

minus

Behaves like other operators, converts to numbers

```
const log = (value) => console.log(value);

log(null - 1); // -1
log(undefined - 1); // NaN
log(false - 1); // -1
log(true - 1); // 0
log("" - 1); // -1
log("s" - 1); // NaN
log("1" - 1); // 0

log(1 - null); // 1
log(1 - undefined); // NaN
log(1 - false); // 1
log(1 - true); // 0
log(1 - ""); // 1
```

```
log(1 - "s"); // NaN
log(1 - "1"); // 0
```

comparisons

When comparing values of different types, they convert to numbers

```
const log = (value) => console.log(value);

log(null < 1); // true
log(undefined < 1); // false
log(undefined > 1); // false
log(false < 1); // true
log(true > 0); // true
log("" > 0); // false
log("s" > 1); // false
log("1" > 0); // true
log("12" > "111"); // true
log("12" > 111); // false
log("break");
log(null <= 1); // true
log(null <= 0); // true
log(undefined <= 1); // false
log(undefined >= 1); // false
log(false <= 1); // true
log(true >= 0); // true
log("" >= 0); // true
log("s" >= 1); // false
log("1" >= 0); // true
log("12" >= "111"); // true
log("12" >= 111); // false
log("break");
log("0" >= 0); // true
```

equality

Strict equality === doesn't convert values

```
const log = (value) => console.log(value);

log(null == null); // true
log(undefined == undefined); // true
log(undefined == null); // true
log(null == undefined); // true
log(0 == false); // true
log(1 == true); // true
log(0 == ""); // true
log(1 == "s"); // false
log(1 == "1"); // true
// [] converted to ''
log(0 == []); // true
log(0 == {}); // false
log("break");
log(null === null); // true
```

```
log(undefined === undefined); // true
log(undefined === null); // false
log([] === []); // false
```

random conversions

```
const log = (value) => console.log(value);

log("10" + 2 * "5"); // 1010
log("" + 0 + 1); // "01"
log("" + 0 - 1); // -1
log("" - 1 + 0); // -1
log(true + false); // 1
log(6 / "3"); // 2
log("2" * "3"); // 6
log(4 + 5 + "px"); // "9px"
log("$" + 4 + 5); // "$45"
log("4" - 2); // 2
log("4px" - 2); // NaN
log("  -9  " + 5); // "  -9  5"
log("  -9  " - 5); // -14
log(null + 1); // 1
log(undefined + 1); // NaN
log(" \t \n" - 2); // -2
```

call, bind, and apply in JavaScript

Understanding `call`, `bind`, and `apply` is essential for mastering how JavaScript handles function context (`this`). These tools allow you to control the execution context of functions, borrow behavior across objects, and preserve method bindings in callbacks and asynchronous code.

This guide includes:

- Differences between `call`, `apply`, and `bind`
- Use cases for method borrowing, decorators, and delayed execution
- A custom `bind` implementation
- Behavior of arrow functions with `this`

Losing and Rebinding Context

This example demonstrates what happens when a method loses its context (`this`) and how to recover it using `call()` and `bind()`:

- Calling `counter.add(1)` works as expected — `this` refers to `counter`.
- When passing `counter.add` directly to another function (`operate`), it loses context and throws an error.
- Using `call()` explicitly sets the context.
- Using `bind()` returns a new function with `this` permanently bound to `counter`.

```
const counter = {
  value: 10,
  add(num) {
    this.value += num;
    return this.value;
  },
};

function operate(method, num) {
  return method(num);
}

function operateWithCall(method, num) {
  return method.call(counter, num);
}

console.log(counter.add(1)); // 11
console.log(operateWithCall(counter.add, 1)); // 12
console.log(operate(counter.add.bind(counter), 1)); // 13
try {
  console.log(operate(counter.add, 1));
} catch (e) {
  console.error(e.message);
  // Cannot read properties of undefined (reading 'value')
}
```

Using `call` and `apply` to Change Context

This example shows how `call()` and `apply()` can be used to invoke a method with a different `this` context:

- `objA.print.call(objB)` executes `objA.print` with `objB` as `this`, so it logs `"I am from objB"`.
- `apply()` works similarly, but takes arguments as an array.

```
const objA = {
  data: "I am from objA",
  print() {
    console.log(this.data);
  },
};

const objB = {
  data: "I am from objB",
};

objA.print.call(objB); // I am from objB
objA.print.apply(objB); // I am from objB
```

Preserving Method Context with `setTimeout`

This example shows how to preserve the `this` context when calling a method asynchronously using `setTimeout`.

The `delayMethod` function returns a wrapper that defers method execution by 1 second while ensuring that `this` still refers to the original object.

- `user.greet()` uses `this.name`, so it's crucial to call it with the correct context.
- Wrapping it with `obj[methodName](...args)` inside the timeout maintains the correct `this`.

```
function delayMethod(obj, methodName) {
  return function (...args) {
    setTimeout(() => obj[methodName](...args), 1000);
  };
}

const user = {
  name: "Alice",
  greet(greeting) {
    console.log(`${greeting}, ${this.name}!`);
  },
};

const delayedGreet = delayMethod(user, "greet");
delayedGreet("Hello"); // After 1 second: "Hello, Alice!"
```

Custom `bind()` Implementation

This example demonstrates how to implement a simplified version of `Function.prototype.bind()`.

The `customBind` function:

- Stores a reference to the original function.
- Returns a new function that, when called, invokes the original with a fixed `this` context and any partially applied arguments.

This mimics native `bind()` behavior, including partial application of arguments.

```
Function.prototype.customBind = function (context, ...args) {
  const originalFunction = this;
  return function (...newArgs) {
    return originalFunction.apply(context, [...args, ...newArgs]);
  };
};

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

const person = { name: "John" };
const boundGreet = greet.customBind(person, "Hello");
boundGreet("!"); // "Hello, John!"
```

Sharing Methods Across Objects with `call()`

In this example, the `calculateTotal` method from `store1` is reused for `store2` by using `Function.prototype.call()`.

- `call()` sets `this` to `store2`, so the method computes tax using `store2.taxRate`.
- As an alternative, the method is directly assigned to `store2`, allowing normal invocation syntax.

```
const store1 = {
  taxRate: 0.1,
  calculateTotal(price, quantity) {
    const subtotal = price * quantity;
    return subtotal + subtotal * this.taxRate;
  },
};

const store2 = {
  taxRate: 0.2,
};

const price = 50;
const quantity = 2;

// Calculate total for `store2` using the method from `store1`
const total = store1.calculateTotal.call(store2, price, quantity);
console.log(total); // 120 (price * quantity + tax)

// 2 solution
store2.calculateTotal = store1.calculateTotal;
console.log(store2.calculateTotal(price, quantity));
```

`bind()` for Context in Rate-Limited Function

This example demonstrates how to use `bind()` to ensure the correct `this` context when passing an object's method to a wrapper function.

- `logger.log` is passed into `rateLimit()` which returns a throttled version of the method.
- Since `log` relies on `this.message`, `bind(logger)` is required to preserve the context.
- Without `bind()`, `this.message` would be `undefined` when `log()` is called inside the throttled wrapper.

```
function rateLimit(fn, ms) {
  let lastCall = 0;

  return function (...args) {
    const now = Date.now();

    if (now - lastCall < ms) return;
    lastCall = now;
    return fn(...args);
  };
}

const logger = {
  message: "Rate limited log",
  log() {
    console.log(this.message);
  },
};

const rateLimitedLog = rateLimit(logger.log.bind(logger), 1000);
rateLimitedLog(); // Rate limited log
rateLimitedLog();
setTimeout(rateLimitedLog, 1500);
// Logs "Rate limited log" after 1.5s
```

Using `bind()` to Preserve `this`

In this example, `greet()` is a regular function that relies on `this.name`. When it's not called in the context of an object, `this` would be `undefined`.

To ensure the correct context (`person`), we create a bound version of the function using `bind(person)`.

This guarantees that `this.name` inside `greet()` always refers to `"John"`, even if called independently.

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

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

// Create a bound function
```

```
const boundGreet = greet.bind(person);
boundGreet(); // Hello, John
```

Arrow Functions Ignore `call`

In this example, the `arrow` function is defined inside the `greet()` method using an arrow function.

Arrow functions do not have their own `this` — they inherit `this` from their enclosing lexical context. So even when using `call()` with a different `this` value (`{ name: "Bob" }`), the arrow function still uses `this.name` from the outer `greet()` context, which is `user` .

Thus, the output is: `Hello, Alice` .

```
const user = {
  name: "Alice",
  greet() {
    const arrow = () => console.log(`Hello, ${this.name}`);
    arrow.call({ name: "Bob" }); // Arrow functions ignore `call`
  },
};

user.greet(); // Hello, Alice
```

Preserving `this` in Callbacks

When passing a method like `user.logName` to another function (e.g. `executeCallback`), the `this` context is lost unless it is explicitly preserved.

- `bind(user)` ensures that `this` always refers to `user` .
- An arrow function `() => user.logName()` also preserves the context by calling the method directly on `user` .

Both approaches ensure the correct `this` context, so the output is: `Alice` .

```
const user = {
  name: "Alice",
  logName() {
    console.log(this.name);
  },
};

function executeCallback(callback) {
  callback();
}

// Preserve context with `bind`
executeCallback(user.logName.bind(user)); // Alice
executeCallback(() => user.logName()); // Alice
```

Dictionary of Nested Arrays

This example demonstrates how to transform a deeply nested array structure (e.g., categories → subcategories → items) into a dictionary format for **constant-time lookup** by `id`.

Why it matters

When working with large, hierarchical datasets (like tree structures or menu items), repeatedly using `.find()` or `.filter()` on nested arrays becomes expensive and messy. By flattening them into a dictionary, you gain:

- 🔍 **O(1) access time** to any item by `id`
- 💡 Easier updates, deletions, or traversal
- 📦 A solid foundation for normalized state in frontend apps

This guide walks through:

- A simple nested loop approach using JavaScript
- A recursive and reusable `mapToDictionary()` helper
- A strongly typed TypeScript version for enterprise-scale reliability

🔢 Plain JavaScript: Nested Loop Approach

```
const data = [
  {
    id: 1,
    name: "Category A",
    items: [
      {
        id: 2,
        name: "Subcategory A1",
        items: [
          { id: 3, name: "Item A1-1", value: 10 },
          { id: 4, name: "Item A1-2", value: 15 },
        ],
      },
      {
        id: 5,
        name: "Subcategory A2",
        items: [
          { id: 6, name: "Item A2-1", value: 20 },
          { id: 7, name: "Item A2-2", value: 25 },
        ],
      },
    ],
  },
  {
    id: 8,
    name: "Category B",
    items: [
      {
        id: 9,
        name: "Subcategory B1",
        items: [
```

```
            { id: 10, name: "Item B1-1", value: 30 },
            { id: 11, name: "Item B1-2", value: 35 },
          ],
        },
        {
          id: 12,
          name: "Subcategory B2",
          items: [
            { id: 13, name: "Item B2-1", value: 40 },
            { id: 14, name: "Item B2-2", value: 45 },
          ],
        },
      ],
    },
];

function createNestedDictionary(data) {
  const dictionary = {};

  for (const category of data) {
    dictionary[category.id] = { ...category, subcategories: {} };

    for (const subcategory of category.items) {
      dictionary[category.id].subcategories[subcategory.id] = {
        ...subcategory,
        items: {},
      };

      for (const item of subcategory.items) {
        dictionary[category.id].subcategories[subcategory.id].items[item.id] =
          item;
      }
    }
  }

  return dictionary;
}

const nestedDictionary = createNestedDictionary(data);

console.log(nestedDictionary[1].name); // Category A
console.log(nestedDictionary[1].subcategories[5].name); // Subcategory A2
console.log(nestedDictionary[1].subcategories[5].items[7].name); // Item A2-2
console.log(nestedDictionary[8].subcategories[12].items[14]);
// { id: 14, name: 'Item B2-2', value: 45 }
```

♻️ Recursive Helper: `mapToDictionary()`

This reusable utility recursively maps any array of nested structures into a flat dictionary by id.

```
const data = [
  {
    id: 1,
    name: "Category A",
    items: [
      {
        id: 2,
        name: "Subcategory A1",
        items: [
          { id: 3, name: "Item A1-1", value: 10 },
```

```
            { id: 4, name: "Item A1-2", value: 15 },
          ],
        },
        {
          id: 5,
          name: "Subcategory A2",
          items: [
            { id: 6, name: "Item A2-1", value: 20 },
            { id: 7, name: "Item A2-2", value: 25 },
          ],
        },
      ],
    },
    {
      id: 8,
      name: "Category B",
      items: [
        {
          id: 9,
          name: "Subcategory B1",
          items: [
            { id: 10, name: "Item B1-1", value: 30 },
            { id: 11, name: "Item B1-2", value: 35 },
          ],
        },
        {
          id: 12,
          name: "Subcategory B2",
          items: [
            { id: 13, name: "Item B2-1", value: 40 },
            { id: 14, name: "Item B2-2", value: 45 },
          ],
        },
      ],
    },
];

function mapToDictionary(data, keys) {
  const [currentKey, ...remainingKeys] = keys;

  return data.reduce((acc, item) => {
    acc[item.id] = {
      ...item,
      [currentKey || "items"]: item.items
        ? mapToDictionary(item.items, remainingKeys)
        : undefined,
    };
    return acc;
  }, {});
}

const nestedDictionary = mapToDictionary(data, ["subcategories", "items"]);

console.log(nestedDictionary[1].name); // Category A
console.log(nestedDictionary[1].subcategories[5].name); // Subcategory A2
console.log(nestedDictionary[1].subcategories[5].items[7].name); // Item A2-2
console.log(nestedDictionary[8].subcategories[9].items[11]);
// { id: 11, name: 'Item B1-2', value: 35, items: undefined }
```

TypeScript: Fully Typed Nested Dictionary

The following code provides strict types and reusable functions to construct the same dictionary using TypeScript.

```typescript
type Item = {
  id: number;
  name: string;
  value: number;
};

type Subcategory = {
  id: number;
  name: string;
  items: Item[];
};

type Category = {
  id: number;
  name: string;
  items: Subcategory[];
};

type Subcategories = Record<
  number,
  {
    name: string;
    items: Record<number, Item>;
  }
>;

type NestedDictionary = Record<
  number,
  {
    name: string;
    subcategories: Subcategories;
  }
>;

function mapItemsToDictionary(items: Item[]) {
  return items.reduce(
    (acc, item) => {
      acc[item.id] = item;
      return acc;
    },
    {} as Record<number, Item>,
  );
}

function mapSubcategoriesToDictionary(
  subcategories: Subcategory[],
) {
  return subcategories.reduce((acc, subcategory) => {
    acc[subcategory.id] = {
      name: subcategory.name,
      items: mapItemsToDictionary(subcategory.items),
    };
    return acc;
  }, {} as Subcategories);
}

function createNestedDictionary(
  categories: Category[],
): NestedDictionary {
  return categories.reduce((acc, category) => {
    acc[category.id] = {
      name: category.name,
      subcategories: mapSubcategoriesToDictionary(
        category.items,
```

```
      ),
    };
    return acc;
  }, {} as NestedDictionary);
}

const data: Category[] = [
  {
    id: 1,
    name: "Category A",
    items: [
      {
        id: 2,
        name: "Subcategory A1",
        items: [
          { id: 3, name: "Item A1-1", value: 10 },
          { id: 4, name: "Item A1-2", value: 15 },
        ],
      },
      {
        id: 5,
        name: "Subcategory A2",
        items: [
          { id: 6, name: "Item A2-1", value: 20 },
          { id: 7, name: "Item A2-2", value: 25 },
        ],
      },
    ],
  },
  {
    id: 8,
    name: "Category B",
    items: [
      {
        id: 9,
        name: "Subcategory B1",
        items: [
          { id: 10, name: "Item B1-1", value: 30 },
          { id: 11, name: "Item B1-2", value: 35 },
        ],
      },
      {
        id: 12,
        name: "Subcategory B2",
        items: [
          { id: 13, name: "Item B2-1", value: 40 },
          { id: 14, name: "Item B2-2", value: 45 },
        ],
      },
    ],
  },
];

const nestedDictionary = createNestedDictionary(data);

console.log(nestedDictionary[1].name); // Category A
console.log(nestedDictionary[1].subcategories[5].name); // Subcategory A2
console.log(
  nestedDictionary[1].subcategories[5].items[7].name,
); // Item A2-2
console.log(
  nestedDictionary[8].subcategories[12].items[14].value,
); // 45
```

Group List by Quarters

Groups monthly financial data by calendar quarters using the `getQuarter()` method from the `date-fns` library. Each group includes individual monthly entries and an optional cumulative total for the quarter.

Useful for:

- Visualizing quarterly reports
- Aggregating financial or performance data
- Preparing charting datasets or grouped UI views

Steps:

1. `groupDataByQuarter(data)` – Organizes data into a dictionary using keys like `"2023-Q1"` based on the `date` field.
2. `initializeDates(groupedQuarters)` – Prepares a simplified structure listing quarters and associated month strings.
3. `initializeRestData(groupedQuarters)` – Flattens the quarterly structure into tax-specific groups, returning an array of metrics (`profit` , `profit_percent`) with their per-quarter values and totals.

This modular approach is helpful when formatting data for analytics dashboards or quarterly reports.

```
import { getQuarter, parseISO } from "date-fns";

// data
export const TaxEntitiesKeys = {
  profit: "profit",
  profit_percent: "profit_percent",
};

const mockTaxes = {
  cumulative: [
    { date: "2023-01-31", profit: 1000, profit_percent: 100 },
    { date: "2023-02-28", profit: 2000, profit_percent: 200 },
    { date: "2023-04-30", profit: 3000, profit_percent: 250 },
  ],
  report: [
    { date: "2023-01-31", profit: 300, profit_percent: 30 },
    { date: "2023-02-28", profit: 500, profit_percent: 50 },
    { date: "2023-04-30", profit: 700, profit_percent: 70 },
  ],
};
// data

// group by quarters
const groupDataByQuarter = (data) => {
  const quarters = {};

  // Group data into quarters based on their date
  for (const monthData of data.report) {
    const date = parseISO(monthData.date);
    const year = date.getFullYear();
    const quarter = getQuarter(date);
    const key = `${year}-Q${quarter}`;
```

```
    if (!quarters[key]) {
      quarters[key] = { months: [], total: null };
    }
    quarters[key].months.push(monthData);
  }

  // Assign cumulative totals to their respective quarters
  for (const totalData of data.cumulative) {
    const date = parseISO(totalData.date);
    const year = date.getFullYear();
    const quarter = getQuarter(date);
    const key = `${year}-Q${quarter}`;

    if (quarters[key]) {
      quarters[key].total = totalData;
    }
  }

  return quarters;
};

const groupedQuarters = groupDataByQuarter(mockTaxes);
console.log(JSON.stringify({ groupedQuarters }, null, 2));
// group by quarters

// {
//   groupedQuarters: {
//     "2023-Q1": {
//       months: [
//         {
//           date: "2023-01-31",
//           profit: 300,
//           profit_percent: 30,
//         },
//         {
//           date: "2023-02-28",
//           profit: 500,
//           profit_percent: 50,
//         },
//       ],
//       total: {
//         date: "2023-02-28",
//         profit: 2000,
//         profit_percent: 200,
//       },
//     },
//     "2023-Q2": {
//       months: [
//         {
//           date: "2023-04-30",
//           profit: 700,
//           profit_percent: 70,
//         },
//       ],
//       total: {
//         date: "2023-04-30",
//         profit: 3000,
//         profit_percent: 250,
//       },
//     },
//   },
// };

// dates
const initializeDates = (quarters) => {
```

```
  const dates = [];

  Object.entries(quarters).forEach(([dateKey, data]) => {
    const [year, quarter] = dateKey.split("-Q").map(Number);

    let yearEntry = dates.find((entry) => entry.year === year);
    if (!yearEntry) {
      yearEntry = { year, periods: [] };
      dates.push(yearEntry);
    }

    yearEntry.periods.push({
      months: data.months.map((month) => month.date),
      quarter: `${quarter}`,
    });
  });

  return dates;
};

console.log(
  JSON.stringify({ dates: initializeDates(groupedQuarters) }, null, 2),
);
// dates

// {
//   dates: [
//     {
//       year: 2023,
//       periods: [
//         {
//           months: ["2023-01-31", "2023-02-28"],
//           quarter: "1",
//         },
//         {
//           months: ["2023-04-30"],
//           quarter: "2",
//         },
//       ],
//     },
//   ],
// };

// rest data
const initializeRestData = (quarters) => {
  return Object.keys(TaxEntitiesKeys).map((taxKey) => {
    const taxKeyTyped = taxKey;

    return {
      title: taxKeyTyped,
      periods: Object.entries(quarters).map(([_, data]) => ({
        months: data.months.map((month) => month[taxKeyTyped]),
        total: data.total ? data.total[taxKeyTyped] : null,
      })),
    };
  });
};

console.log(
  JSON.stringify({ restData: initializeRestData(groupedQuarters) }, null, 2),
);
// rest data

// {
//   restData: [
//     {
```

```
//       title: "profit",
//       periods: [
//         {
//           months: [300, 500],
//           total: 2000,
//         },
//         {
//           months: [700],
//           total: 3000,
//         },
//       ],
//     },
//     {
//       title: "profit_percent",
//       periods: [
//         {
//           months: [30, 50],
//           total: 200,
//         },
//         {
//           months: [70],
//           total: 250,
//         },
//       ],
//     },
//   ],
// };
```

Transform Lists and Nested Structures

This guide shows how to convert common data structures for better access, mutation, and integration — including lists, trees, dictionaries, and flattened arrays.

🔁 List to Tree

This utility transforms a flat list of items (with `id` and `parentId` fields) into a nested **tree structure**. It uses a `Map` to efficiently link children to their parents while preserving the hierarchy.

It also includes a `treeToList()` function that **flattens** a tree back into a list — useful for data normalization, storing tree nodes in databases, or syncing front-end trees with backend updates.

```
function listToTree(items) {
  const map = new Map();
  const roots = [];

  // Map items by ID
  for (const item of items) {
    map.set(item.id, { ...item, children: [] });
  }

  for (const item of items) {
    const node = map.get(item.id);

    if (item.parentId === null) {
      roots.push(node);
    } else {
      const parent = map.get(item.parentId);

      if (parent) {
        parent.children.push(node);
      }
    }
  }

  return roots;
}

const items = [
  { id: 1, name: "Root 1", parentId: null },
  { id: 2, name: "Child 1.1", parentId: 1 },
  { id: 3, name: "Child 1.2", parentId: 1 },
  { id: 4, name: "Root 2", parentId: null },
  { id: 5, name: "Child 2.1", parentId: 4 },
  { id: 6, name: "SubChild 2.1.1", parentId: 5 },
];

const tree = listToTree(items);
console.log(tree);
// [
//   {
//     "id": 1,
//     "name": "Root 1",
//     "parentId": null,
//     "children": [
//       {
```

```
//         "id": 2,
//         "name": "Child 1.1",
//         "parentId": 1,
//         "children": []
//       },
//       {
//         "id": 3,
//         "name": "Child 1.2",
//         "parentId": 1,
//         "children": []
//       }
//     ]
//   },
//   {
//     "id": 4,
//     "name": "Root 2",
//     "parentId": null,
//     "children": [
//       {
//         "id": 5,
//         "name": "Child 2.1",
//         "parentId": 4,
//         "children": [
//           {
//             "id": 6,
//             "name": "SubChild 2.1.1",
//             "parentId": 5,
//             "children": []
//           }
//         ]
//       }
//     ]
//   }
// ]

function treeToList(tree) {
  const list = [];

  function traverse(node) {
    const { children, ...rest } = node;
    list.push(rest);

    if (children) {
      for (const child of children) {
        traverse(child);
      }
    }
  }

  for (const root of tree) {
    traverse(root);
  }

  return list;
}

const flatList = treeToList(tree);
console.log(flatList);
// [
//   { id: 1, name: 'Root 1', parentId: null },
//   { id: 2, name: 'Child 1.1', parentId: 1 },
//   { id: 3, name: 'Child 1.2', parentId: 1 },
//   { id: 4, name: 'Root 2', parentId: null },
//   { id: 5, name: 'Child 2.1', parentId: 4 },
```

```
//   { id: 6, name: 'SubChild 2.1.1', parentId: 5 }
// ]
```

🧩 Flat Nested Items (With Category + Subcategory)

Transforms deeply nested category-subcategory-item data into a flat array where each item contains metadata about its parent subcategory and category.

You can do it manually with loops or concisely with `flatMap()`.

```
const data = [
  {
    id: 1,
    name: "Category A",
    items: [
      {
        id: 2,
        name: "Subcategory A1",
        items: [
          { id: 3, name: "Item A1-1", value: 10 },
          { id: 4, name: "Item A1-2", value: 15 },
        ],
      },
      {
        id: 5,
        name: "Subcategory A2",
        items: [
          { id: 6, name: "Item A2-1", value: 20 },
          { id: 7, name: "Item A2-2", value: 25 },
        ],
      },
    ],
  },
  {
    id: 8,
    name: "Category B",
    items: [
      {
        id: 9,
        name: "Subcategory B1",
        items: [
          { id: 10, name: "Item B1-1", value: 30 },
          { id: 11, name: "Item B1-2", value: 35 },
        ],
      },
      {
        id: 12,
        name: "Subcategory B2",
        items: [
          { id: 13, name: "Item B2-1", value: 40 },
          { id: 14, name: "Item B2-2", value: 45 },
        ],
      },
    ],
  },
];

function transformItems(data) {
  const transformedItems = [];
```

```
  for (const category of data) {
    for (const subcategory of category.items) {
      for (const item of subcategory.items) {
        transformedItems.push({
          id: item.id,
          name: item.name,
          value: item.value,
          subcategory: subcategory.name,
          category: category.name,
        });
      }
    }
  }

  return transformedItems;
}

const itemsWithCategories = transformItems(data);

console.log(itemsWithCategories);

// [
//   {
//     id: 3,
//     name: "Item A1-1",
//     value: 10,
//     subcategory: "Subcategory A1",
//     category: "Category A",
//   },
//   {
//     id: 4,
//     name: "Item A1-2",
//     value: 15,
//     subcategory: "Subcategory A1",
//     category: "Category A",
//   },
//   {
//     id: 6,
//     name: "Item A2-1",
//     value: 20,
//     subcategory: "Subcategory A2",
//     category: "Category A",
//   },
//   {
//     id: 7,
//     name: "Item A2-2",
//     value: 25,
//     subcategory: "Subcategory A2",
//     category: "Category A",
//   },
//   {
//     id: 10,
//     name: "Item B1-1",
//     value: 30,
//     subcategory: "Subcategory B1",
//     category: "Category B",
//   },
//   {
//     id: 11,
//     name: "Item B1-2",
//     value: 35,
//     subcategory: "Subcategory B1",
//     category: "Category B",
//   },
//   {
```

```
//     id: 13,
//     name: "Item B2-1",
//     value: 40,
//     subcategory: "Subcategory B2",
//     category: "Category B",
//   },
//   {
//     id: 14,
//     name: "Item B2-2",
//     value: 45,
//     subcategory: "Subcategory B2",
//     category: "Category B",
//   },
// ];

function transformItems2(data) {
  return data.flatMap((category) =>
    category.items.flatMap((subcategory) =>
      subcategory.items.map((item) => ({
        id: item.id,
        name: item.name,
        value: item.value,
        subcategory: subcategory.name,
        category: category.name,
      })),
    ),
  );
}

const itemsWithCategories2 = transformItems2(data);
console.log(itemsWithCategories2);

// ...same result
```

🔄 Promises, Closures and Event Loop

This section demonstrates how **JavaScript promises**, **closures**, and the **event loop** interact in real-world scenarios. These examples cover:

- How closures preserve state across async calls
- What happens when you `await` vs. run multiple async calls in parallel
- The importance of capturing variables in closures before `setTimeout`
- How the event loop schedules tasks with `setTimeout`, `Promise`, and `console.log`
- Sequential chaining of promises with closures

Perfect for understanding async behavior, state retention, and timing execution in complex JavaScript code.

Async function with closure and delayed execution

This example shows how **closures** capture state (`count`) and how `await` affects the execution order. Each call to `asyncCounter()` increments `count` and logs it immediately. However, `await` delays further code inside the async function, allowing subsequent calls to update `count` before the first finishes.

- Multiple calls to the same async function share the same closure state.
- `console.log("Script complete")` runs immediately because `await` yields control to the event loop.

```
function createAsyncCounter() {
  let count = 0;
  return async function incrementAsyncCounter() {
    count++;
    console.log({ count });
    await new Promise((resolve) => setTimeout(resolve, 1000));
    console.log("Async Counter:", count);
  };
}

const asyncCounter = createAsyncCounter();

asyncCounter();
asyncCounter().then(() => {
  asyncCounter();
});

console.log("Script complete");

// { count: 1 }
// { count: 2 }
// Script complete
// 1 s delay
// Async Counter: 2
// Async Counter: 2
// { count: 3 }
// 1 s delay
// Async Counter: 3
```

Sequential async calls with `await` and closure

This example demonstrates how sequential `await` calls work with closures. Each async call completes before the next one begins, ensuring the counter increments correctly. The `count` variable is preserved across invocations due to the closure, and each `setTimeout` resolves after 1 second, leading to predictable, step-by-step output.

```
function createAsyncCounter() {
  let count = 0;
  return async function incrementAsyncCounter() {
    count++;
    await new Promise((resolve) => setTimeout(resolve, 1000));
    console.log("Async Counter:", count);
  };
}

const asyncCounter = createAsyncCounter();

await asyncCounter();
await asyncCounter();
await asyncCounter();

console.log("Script complete");

// with 1s between:
// Async Counter: 1
// Async Counter: 2
// Async Counter: 3
// immediately after counter 3:
// Script complete
```

Task scheduling with closures and async timing

This example highlights how closure captures the shared `taskCount` variable, which is incremented before each scheduled task. All tasks reference the same `taskCount` value by the time their `setTimeout` callbacks run, resulting in each log displaying `Task Count: 3`. It also demonstrates how microtasks (like `Promise.resolve().then`) run before timers and the event loop order between synchronous code, promises, and `setTimeout`.

```
function createTaskScheduler() {
  let taskCount = 0;
  return function scheduleTask() {
    taskCount++;
    setTimeout(() => {
      console.log("Task Count:", taskCount);
    }, taskCount * 1000);
  };
}

const scheduleTask = createTaskScheduler();

scheduleTask();
scheduleTask();

Promise.resolve().then(() => {
  console.log("promise");
  scheduleTask();
```

```
});

console.log("Tasks scheduled");

// Tasks scheduled
// promise
// after 1s with 1s between
// Task Count: 3
// Task Count: 3
// Task Count: 3
```

Scheduling tasks with preserved state using closures

In this version, the closure captures the `taskCount` value in a separate variable `savedCount` before scheduling the `setTimeout`. This ensures that each task logs the correct value at the time it was created, rather than referencing the final shared `taskCount`. The result is that each `setTimeout` logs a different value in increasing order, showcasing how to avoid timing issues by saving state early.

```
function createTaskScheduler() {
  let taskCount = 0;
  return function scheduleTask() {
    taskCount++;
    const savedCount = taskCount;
    setTimeout(() => {
      console.log("Task Count:", savedCount);
    }, taskCount * 1000);
  };
}

const scheduleTask = createTaskScheduler();

scheduleTask();
scheduleTask();

Promise.resolve().then(() => {
  scheduleTask();
});

console.log("Tasks scheduled");

// Tasks scheduled
// after 1s with 1s between:
// Task Count: 1
// Task Count: 2
// Task Count: 3
```

Preserving async state with closure in chained promises

This example demonstrates how closures can maintain and update state (`count`) across multiple asynchronous calls. Because the promise is chained (`then -> then`), each `counter()` call executes sequentially, allowing `count` to increment between calls. The use of `setTimeout` inside the promise simulates asynchronous work, and the closure ensures the `count` value is correctly preserved across executions.

```
function createCounter() {
  let count = 0;
  return function incrementCounter() {
    count++;
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(count);
      }, 1000);
    });
  };
}

const counter = createCounter();

// if it was called at the same time, count would
// have not been saved in closure

counter()
  .then((result) => {
    console.log("Counter 1:", result);
    return counter();
  })
  .then((result) => {
    console.log("Counter 2:", result);
  });

console.log("Script in progress");

// Script in progress
// after 1s with 1s between:
// Counter 1: 1
// Counter 2: 2
```

Asynchronous multiplier using closure and chained promises

This example showcases how closures can maintain internal state (`factor = 2`) in an asynchronous function. The `multiplyByTwo` function, returned by `createAsyncMultiplier`, remembers the `factor` across multiple chained `.then()` calls. Each multiplication is delayed with `setTimeout`, and the result of one multiplication is passed to the next, demonstrating sequential asynchronous logic with shared state.

```
function createAsyncMultiplier() {
  let factor = 2;

  return function multiply(value) {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(value * factor);
      }, 1000);
    });
  };
}

const multiplyByTwo = createAsyncMultiplier();

multiplyByTwo(5)
  .then((result) => {
    console.log("Multiply 1:", result);
```

```
    return multiplyByTwo(result);
  })
  .then((result) => {
    console.log("Multiply 2:", result);
    return multiplyByTwo(result);
  })
  .then((result) => {
    console.log("Multiply 3:", result);
  });

console.log("Multiplication started");

// Multiplication started
// after 1s with 1s between:
// Multiply 1: 10
// Multiply 2: 20
// Multiply 3: 40
```

JavaScript Sorting Algorithms

Sorting algorithms are foundational to understanding performance and logic in computer science. These are 3 fundamental ways to sort an array.

bubble sort

The `bubbleSort()` function sorts an array by repeatedly stepping through the list, comparing adjacent items, and swapping them if they are in the wrong order. This process is repeated until the array is sorted.

This custom implementation demonstrates the bubble sort algorithm, where the outer loop ensures that each pass moves the largest element to its correct position. The process is optimized by checking whether any elements were swapped in the inner loop, allowing the algorithm to break early if the array is already sorted.

```
function bubbleSort(array) {
  const n = array.length;

  for (let i = 0; i < n - 1; i++) {
    let swapped = false;

    for (let j = 0; j < n - i - 1; j++) {
      if (array[j] > array[j + 1]) {
        [array[j], array[j + 1]] = [array[j + 1], array[j]];
        swapped = true;
      }
    }

    if (!swapped) {
      break;
    }
  }
  return array;
}

const myArray = [7, 3, 9, 12, 11];
const sortedArray = bubbleSort(myArray);
console.log(sortedArray);
// [ 3, 7, 9, 11, 12 ]
```

insertion sort

The `insertionSort()` function sorts an array by repeatedly picking the next element and inserting it into the correct position within the already sorted portion of the array. It builds the sorted array one item at a time by comparing the current item to the previous ones.

This custom implementation demonstrates the insertion sort algorithm. For each element, it is compared to the previous ones and inserted into the correct position, shifting elements as necessary.

```
function insertionSort(array) {
  const n = array.length;

  for (let i = 1; i < n; i++) {
    let insertIndex = i;
    const currentValue = array[i];

    for (let j = i - 1; j >= 0; j--) {
      if (array[j] > currentValue) {
        array[j + 1] = array[j];
        insertIndex = j;
      } else {
        break;
      }
    }

    array[insertIndex] = currentValue;
  }
  return array;
}

const myArray = [64, 34, 25, 12, 22, 11, 90, 5];
const sortedArray = insertionSort(myArray);
console.log(sortedArray);
// [ 5, 11, 12, 22, 25, 34, 64, 90 ]
```

selection sort

The `selectionSort()` algorithm sorts an array by repeatedly finding the smallest element from the unsorted part and swapping it with the element at the current position. It iterates through the array, and in each iteration, selects the smallest element from the remaining unsorted portion and moves it to its correct position.

This custom implementation demonstrates the selection sort algorithm by scanning the array for the smallest element and swapping it to the front, progressively moving through the array.

```
const selectionSort = (arr) => {
  const n = arr.length;
  for (let i = 0; i < n - 1; i++) {
    let minIndex = i;

    for (let j = i + 1; j < n; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }

    const minValue = arr.splice(minIndex, 1)[0];
    arr.splice(i, 0, minValue);
  }
};

const myArray = [64, 34, 25, 5, 22, 11, 90, 12];
selectionSort(myArray);
console.log(myArray);
// [ 5, 11, 12, 22, 25, 34, 64, 90 ]
```

JavaScript Most Commonly Used Utils

This catalog includes some of the **most commonly used utility functions and logic patterns** in JavaScript. They're designed to simplify development, reduce bugs, and promote cleaner, more maintainable code.

You'll find utilities like `clsx`, `deepClone`, retry logic, filtering by related properties, topological sort, and reducer patterns — all of which are frequently used in real-world frontend projects and technical interviews.

clsx

The `clsx` utility is commonly used to conditionally join class names in JavaScript and React projects.

This custom implementation mimics the behavior of the `clsx` library by:

- Skipping falsy values (`false`, `null`, `undefined`, `0`, `""`, `NaN`)
- Joining strings as-is
- Recursively flattening arrays
- Including keys from objects whose values are truthy

This allows for clean, readable class name composition in UI code.

```
function clsx(...args) {
  const classes = [];

  for (const arg of args) {
    // Skip the current iteration if the argument is falsy
    if (!arg) continue;

    if (typeof arg === "string") {
      classes.push(arg);
    } else if (Array.isArray(arg)) {
      classes.push(clsx(...arg)); // Recursively process arrays
    } else if (typeof arg === "object") {
      for (const key in arg) {
        if (arg[key]) {
          classes.push(key); // Push key if value is truthy
        }
      }
    }
  }

  return classes.join(" "); // Join classes with a space
}

console.log(
  clsx("base-class", { active: true, disabled: false }, [
    "additional-class",
    "another-class",
  ]),
); // base-class active additional-class another-class
console.log(
  clsx(null, false, "bar", undefined, { baz: null }, "", [[[{ one: 1 }]]]),
); // bar one
```

contains duplicate

This function checks if a given array contains any duplicate values.

It uses a `Set` to track elements that have already been seen while iterating through the array. If a value is found in the `Set`, the function returns `true`, indicating a duplicate. If no duplicates are found by the end of iteration, it returns `false`.

```
function containsDuplicate(nums) {
  const seen = new Set();

  for (const num of nums) {
    if (seen.has(num)) {
      return true;
    }

    seen.add(num);
  }

  return false;
}

console.log(containsDuplicate([1, 1, 3, 4])); // true
console.log(containsDuplicate([1, 3, 4])); // false
```

deep clone

The `deepClone` function recursively copies all levels of an object or array, creating a fully independent clone. This avoids shared references and ensures that changes to nested structures in the cloned object won't affect the original.

It handles:

- Primitive types and `null`
- Arrays using recursive `map`
- Plain objects using recursive property traversal

Perfect for deep copying JSON-compatible data without using `JSON.parse(JSON.stringify(...))`, which fails on special types like `Date`, `Map`, `Set`, or circular references.

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

  if (Array.isArray(obj)) {
    // Recursively clone array elements
    return obj.map(deepClone);
  }

  const clonedObj = {};
  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      // Recursively clone object properties
```

```
      clonedObj[key] = deepClone(obj[key]);
    }
  }

  return clonedObj;
}

const obj = { a: 1, b: [{ c: 2, d: 3 }] };
const clonedObj = deepClone(obj);
console.log(clonedObj); // { a: 1, b: [ { c: 2, d: 3 } ] }
```

filter by related property

This utility filters a list of objects based on a **related property** found in another array — similar to a **join + filter** operation.

In this example, we filter items in the `objects` array based on the `class` value of their related `object_type`, which is matched against the `object_types` array.

Highlights:

- `groupBy()` helps group `object_types` by `class` for fast lookup.
- `filterObjectsByClass()` efficiently filters items by matching related object type's class.
- Uses TypeScript generics for reusability and type safety.

```
const objects = [
  { id: 1, name: "Test 1", object_type: 1 },
  { id: 2, name: "Test 2", object_type: 1 },
  { id: 3, name: "Test 3", object_type: 2 },
  { id: 4, name: "Test 4", object_type: 3 },
];

const object_types = [
  { id: 1, class: "orange" },
  { id: 2, class: "orange" },
  { id: 3, class: "apple" },
  { id: 4, class: "cheese" },
];

const groupBy = <T, K extends string | number | symbol>(
  arr: T[],
  callback: (item: T) => K,
): Record<K, T[]> => {
  return arr.reduce(
    (acc: Record<K, T[]>, item: T) => {
      const key = callback(item);
      if (!acc[key]) acc[key] = [];
      acc[key].push(item);

      return acc;
    },
    {} as Record<K, T[]>,
  );
};

const filterObjectsByClass = <T>(
  cls: string,
```

```
  objects: (T & { object_type: number })[],
  objectTypes: { id: number; class: string }[],
): T[] => {
  const result: T[] = [];

  const objTypesByClass = groupBy(
    objectTypes,
    (item) => item.class,
  );

  for (const item of objects) {
    if (
      objTypesByClass[cls].find(
        (objectType) => objectType.id === item.object_type,
      )
    ) {
      result.push(item);
    }
  }

  return result;
};

const filteredObjects = filterObjectsByClass(
  "orange",
  objects,
  object_types,
);
console.log(filteredObjects);

// [
//   { id: 1, name: 'Test 1', object_type: 1 },
//   { id: 2, name: 'Test 2', object_type: 1 },
//   { id: 3, name: 'Test 3', object_type: 2 }
// ]
```

filterMap

The `filterMap()` utility combines filtering and mapping into a single `reduce()` pass — making it more efficient than chaining `.filter().map()`.

This function takes:

- an `array`,
- a `filterBoolean` function to determine which elements to include,
- and a `mapCallback` to transform each included item.

```
export const filterMap = (array, filterBoolean, mapCallback) => {
  return array.reduce((acc, item, idx) => {
    if (filterBoolean(item)) {
      acc.push(mapCallback(item, idx));
    }
    return acc;
  }, []);
};

const people = [
  { name: "Alice", age: 25, active: true },
```

```
  { name: "Bob", age: 30, active: false },
  { name: "Charlie", age: 35, active: true },
];

const activeNames = filterMap(
  people,
  (person) => person.active,
  (person) => person.name,
);

console.log(activeNames); // ['Alice', 'Charlie']
```

innerJoin

Implements a basic `innerJoin` function for arrays — similar to SQL inner joins. This method takes a `predicate`, a list of `records`, and a list of `ids`. It returns all records where the predicate returns true for at least one ID.

This approach is helpful for:

- Matching entities across two datasets
- Resolving references between relational data
- Filtering records based on foreign key relations

The example joins musician records by matching their `id` with an array of selected IDs.

```
function innerJoin(predicate, records, ids) {
  return records.filter((record) => ids.some((id) => predicate(record, id)));
}

const result = innerJoin(
  (record, id) => record.id === id,
  [
    { id: 824, name: "Richie Furay" },
    { id: 956, name: "Dewey Martin" },
    { id: 313, name: "Bruce Palmer" },
    { id: 456, name: "Stephen Stills" },
    { id: 177, name: "Neil Young" },
  ],
  [177, 456, 999],
);

console.log(result);
// [{id: 456, name: 'Stephen Stills'}, {id: 177, name: 'Neil Young'}]
```

Reducer pattern with actions

This example demonstrates the use of a reducer function similar to the pattern used in React's `useReducer`. The reducer function handles different types of actions (`added`, `changed`, `deleted`) to manage a list of tasks. It accumulates state changes over time as actions are applied via `Array.prototype.reduce`, making it a powerful pattern for predictable state updates in complex logic flows.

```
function tasksReducer(tasks, action) {
  switch (action.type) {
    case "added": {
      return [
        ...tasks,
        {
          id: action.id,
          text: action.text,
          done: false,
        },
      ];
    }
    case "changed": {
      return tasks.map((t) => {
        if (t.id === action.id) {
          const { type, ...actionNoType } = action;
          return actionNoType;
        } else {
          return t;
        }
      });
    }
    case "deleted": {
      return tasks.filter((t) => t.id !== action.id);
    }
    default: {
      throw Error("Unknown action: " + action.type);
    }
  }
}

const initialState = [];
const actions = [
  { type: "added", id: 1, text: "Visit Kafka Museum" },
  { type: "added", id: 2, text: "Watch a puppet show" },
  { type: "deleted", id: 1 },
  { type: "added", id: 3, text: "Lennon Wall pic" },
  { type: "changed", id: 3, text: "Lennon Wall", done: true },
];
const finalState = actions.reduce(tasksReducer, initialState);
console.log(finalState);
// [
//     { id: 2, text: 'Watch a puppet show', done: false },
//     { id: 3, text: 'Lennon Wall', done: true }
// ]
```

Retry with exponential backoff

This snippet demonstrates how to implement a retry mechanism when fetching data from an API. It attempts to fetch the resource multiple times (up to a given `retries` limit) and waits a specified `delay` between each retry. If the request ultimately fails, it throws an error. This is particularly useful for handling flaky network requests or ensuring robustness when dealing with unstable APIs.

```
async function fetchWithRetry(url, retries, delay = 1000) {
  for (let attempt = 1; attempt <= retries; attempt++) {
    try {
      const response = await fetch(url);
```

```
      if (!response.ok) {
        throw new Error(`HTTP Error: ${response.status}`);
      }
      return await response.json();
    } catch (error) {
      console.error(`Attempt ${attempt} failed:`, error);

      if (attempt === retries) {
        throw new Error(`Failed to fetch after ${retries} retries`);
      }

      await new Promise((resolve) => setTimeout(resolve, delay));
    }
  }
}

const res = await fetchWithRetry("https://pokeapi.co/api/v2/pokemon-color", 3);
console.log(res);
```

Shuffle (Fisher-Yates Algorithm)

This example implements the Fisher-Yates shuffle, a reliable way to randomly shuffle elements in an array. The algorithm works by iterating the array from the end to the beginning, swapping the current element with a randomly selected one from earlier in the array (or itself). It ensures a uniform distribution of permutations.

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1));

    [array[i], array[j]] = [array[j], array[i]];
  }
}

const shuffledArray = [1, 2, 3, 4, 5];
shuffle(shuffledArray);
console.log(shuffledArray); // [ 2, 1, 4, 5, 3 ]
```

Topological Sort

This example performs a topological sort on a set of items with dependencies. Each item can only appear in the result after all of its dependencies are resolved. It's a common algorithm used in build systems, task schedulers, and dependency resolution tools.

```
const cards = [
  { id: 1, dependent: [6, 7, 8] },
  { id: 2, dependent: [6] },
  { id: 3, dependent: [] },
  { id: 4, dependent: [6, 7, 8] },
  { id: 5, dependent: [6, 8] },
  { id: 6, dependent: [] },
  { id: 7, dependent: [6] },
  { id: 8, dependent: [7] },
  { id: 9, dependent: [1] },
```

```
  { id: 10, dependent: [9] },
];

const getOrderedCards = (cards) => {
  const result = [];
  const added = new Set();

  while (result.length < cards.length) {
    for (const card of cards) {
      if (
        !added.has(card.id) &&
        card.dependent.every((dep) => added.has(dep))
      ) {
        result.push(card.id);
        added.add(card.id);
      }
    }
  }

  return result;
};

console.log(getOrderedCards(cards));
// [
//   3, 6, 7, 8,  1,
//   2, 4, 5, 9, 10
// ]
```