Mastering JavaScript Array Methods: A Comprehensive Guide

Simple Array Methods (Detailed Explanation)

1. `slice()` **Method**
   - **Purpose**: Used to extract a section of an array without altering the original array. It creates a new array containing the selected elements.
   - **Syntax**: `array.slice(startIndex, endIndex)`
     - `startIndex`: The index at which to start extraction (inclusive).
     - `endIndex`: The index at which to end extraction (exclusive). If omitted, slices to the end of the array.
   - **Features**:
     - Accepts negative indices, which count elements from the end of the array.
     - Returns a shallow copy of the specified portion of the array.
   - **Examples**:

```
let arr = ["a", "b", "c", "d", "e"];

console.log(arr.slice(2)); // ['c', 'd', 'e'] (starts from index 2)
console.log(arr.slice(2, 4)); // ['c', 'd'] (from index 2 to 3)
console.log(arr.slice(-2)); // ['d', 'e'] (last two elements)
console.log(arr.slice(-1)); // ['e'] (last element)
console.log(arr.slice(1, -2)); // ['b', 'c'] (from index 1 to third-
last element)
console.log(arr.slice()); // ['a', 'b', 'c', 'd', 'e'] (full shallow
copy)
```

2. `splice()` **Method**
   - **Purpose**: Used to add, remove, or replace elements in an array. Unlike `slice`, it modifies the original array.
   - **Syntax**: `array.splice(startIndex, deleteCount, ...itemsToAdd)`
     - `startIndex`: The index where changes begin.
     - `deleteCount`: The number of elements to remove. If 0, no elements are removed.
     - `...itemsToAdd`: Additional elements to add at `startIndex`.
   - **Features**:

- Can both remove and insert elements at specific positions.
- Alters the original array, making it a destructive method.

- **Examples**:

```
let arr = ["a", "b", "c", "d", "e"];

// Removing elements
arr.splice(-1); // Removes the last element ('e')
console.log(arr); // ['a', 'b', 'c', 'd']

// Removing and starting at a specific index
arr.splice(1, 2); // Removes 2 elements starting at index 1
console.log(arr); // ['a', 'd']
```

3. `reverse()` **Method**

- **Purpose**: Reverses the order of elements in the array **in place**.
- **Features**:
  - Mutates the original array.
  - The first element becomes the last, and the last becomes the first.
- **Examples**:

```
const arr2 = ["j", "i", "h", "g", "f"];

console.log(arr2.reverse()); // ['f', 'g', 'h', 'i', 'j']
console.log(arr2); // ['f', 'g', 'h', 'i', 'j'] (original array
mutated)
```

4. `concat()` **Method**

- **Purpose**: Combines two or more arrays into a **new array**.
- **Syntax**: `array1.concat(array2, ..., arrayN)`
- **Features**:
  - Does not modify the original arrays.
  - Equivalent to using the spread operator (`...`).
- **Examples**:

```
const arr = ["a", "d"];
const arr2 = ["f", "g", "h", "i", "j"];
```

```
// Using concat
const letters = arr.concat(arr2);
console.log(letters); // ['a', 'd', 'f', 'g', 'h', 'i', 'j']

// Using spread operator
console.log([...arr, ...arr2]); // ['a', 'd', 'f', 'g', 'h', 'i',
'j']
```

5. `join()` **Method**

   - **Purpose**: Joins all elements of an array into a string, separated by a specified delimiter.
   - **Syntax**: `array.join(separator)`
     - `separator`: A string to separate the array elements. Default is a comma (`,`).
   - **Examples**:

```
const letters = ["a", "d", "f", "g", "h", "i", "j"];

console.log(letters.join(" - ")); // 'a - d - f - g - h - i - j'
console.log(letters.join()); // 'a,d,f,g,h,i,j' (default separator
is a comma)
```

The `at()` Method (Modern Access to Array Elements)

- **Purpose**: Provides a clean and readable way to access elements from arrays (or strings) using positive or negative indices.
- **Syntax**: `array.at(index)`
  - `index`: The position of the element to access.
    - Positive values start from the beginning (0 is the first element).
    - Negative values count backward from the end (-1 is the last element).
- **Features**:
  - Works on both arrays and strings.
  - Useful for accessing the last element without calculating `length - 1`.
- **Examples**:

```
const arr = [23, 11, 64];

// Positive indices
console.log(arr.at(0)); // 23 (first element)

// Negative indices
```

```
console.log(arr.at(-1)); // 64 (last element)
console.log(arr.at(-2)); // 11 (second-to-last element)

// On strings
console.log("jonas".at(0)); // 'j' (first character)
console.log("jonas".at(-1)); // 's' (last character)
```

This method is particularly handy for scenarios where readability is a priority, especially when accessing the last element in arrays or strings.

Looping with `forEach()`

The `forEach()` method is used to execute a callback function on each element of an array, map, or set. Unlike a `for` loop, `forEach()` does not create a new array or allow breaking out of the loop. It is primarily for executing side effects, such as logging values or modifying external variables.

1. Using `forEach()` with Arrays

- **Purpose**: Iterates over each element of an array and applies the callback.
- **Parameters**:
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array on which `forEach()` is called.
- **Examples**:

```
const movements = [200, -100, 300, -50, 400];

movements.forEach(function (mov, i) {
  console.log(
    `Movement ${i + 1}: You ${mov > 0 ? "deposited" : "withdrew"}
${Math.abs(
      mov
    )}`
  );
});

// Output:
// Movement 1: You deposited 200
// Movement 2: You withdrew 100
// Movement 3: You deposited 300
// Movement 4: You withdrew 50
// Movement 5: You deposited 400
```

2. Using `forEach()` with Maps

- **Purpose**: Iterates through the key-value pairs of a `Map` object.
- **Parameters**:
  - `value`: The current value in the map.
  - `key`: The corresponding key.
  - `map` (optional): The `Map` being iterated over.
- **Examples**:

```
const currencies = new Map([
  ["USD", "United States Dollar"],
  ["EUR", "Euro"],
  ["GBP", "Pound Sterling"],
]);

currencies.forEach(function (value, key) {
  console.log(`${key}: ${value}`);
});

// Output:
// USD: United States Dollar
// EUR: Euro
// GBP: Pound Sterling
```

3. Using `forEach()` with Sets

- **Purpose**: Iterates over the unique values of a `Set`.
- **Behavior**: The `key` and `value` are the same since `Set` does not store key-value pairs.
- **Examples**:

```
const currenciesUnique = new Set(["USD", "GBP", "USD", "EUR", "EUR"]);

currenciesUnique.forEach(function (value) {
  console.log(`${value}: ${value}`);
});

// Output:
// USD: USD
// GBP: GBP
// EUR: EUR
```

Transformations with `map()`

The `map()` method is used to create a new array by applying a transformation function to each element of the original array. Unlike `forEach()`, `map()` returns a new array and does not modify the original array.

- **Purpose**: Transform elements of an array.
- **Parameters**:
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array on which `map()` is called.
- **Examples**:

```
const movements = [200, -100, 300, -50, 400];
const eurToUsd = 1.1;

// Convert movements to USD
const movementsUSD = movements.map((mov) => mov * eurToUsd);
console.log(movementsUSD); // [220, -110, 330, -55, 440]

// Create descriptions for movements
const movementsDescriptions = movements.map(
  (mov, i) =>
    `Movement ${i + 1}: You ${mov > 0 ? "deposited" : "withdrew"}
${Math.abs(
      mov
    )}`
);
console.log(movementsDescriptions);
// [
//   "Movement 1: You deposited 200",
//   "Movement 2: You withdrew 100",
//   ...
// ]
```

Filtering with `filter()`

The `filter()` method is used to extract elements from an array that satisfy a given condition. It returns a new array with the filtered elements.

- **Purpose**: Filter elements based on conditions.
- **Parameters**:
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array on which `filter()` is called.
- **Examples**:

```
const movements = [200, -100, 300, -50, 400];
```

```
// Get deposits
const deposits = movements.filter((mov) => mov > 0);
console.log(deposits); // [200, 300, 400]

// Get withdrawals
const withdrawals = movements.filter((mov) => mov < 0);
console.log(withdrawals); // [-100, -50]
```

Aggregation with `reduce()`

The `reduce()` method applies a callback function on the array, reducing it to a single value (e.g., sum, product, or maximum).

- **Purpose**: Aggregate array values into a single result.
- **Parameters**:
  - `accumulator`: The accumulated result of the reduction.
  - `currentValue`: The current element being processed.
  - `index` (optional): The index of the current element.
  - `array` (optional): The array on which `reduce()` is called.
  - **Initial Value**: Optional starting value for the accumulator.
- **Examples**:
  1. **Calculate Total Balance**:

```
const movements = [200, -100, 300, -50, 400];
const balance = movements.reduce((acc, mov) => acc + mov, 0);
console.log(balance); // 750
```

  2. **Find Maximum Value**:

```
const max = movements.reduce(
  (acc, mov) => (acc > mov ? acc : mov),
  movements[0]
);
console.log(max); // 400
```

Chaining Methods

By combining methods like `filter()`, `map()`, and `reduce()`, you can perform complex operations in a concise and readable way. This is known as method chaining.

- **Example**:

```
const movements = [200, -100, 300, -50, 400];
const eurToUsd = 1.1;

const totalDepositsUSD = movements
  .filter((mov) => mov > 0) // Keep deposits only
  .map((mov) => mov * eurToUsd) // Convert to USD
  .reduce((acc, mov) => acc + mov, 0); // Sum up

console.log(totalDepositsUSD); // 990
```

Explanation:

1. `filter()`: Extracts positive values (deposits).
2. `map()`: Converts each deposit to USD.
3. `reduce()`: Sums up the converted deposits into a total.

`find()` Method

The `find()` method is used to retrieve the first element in an array that matches a given condition. Unlike `filter()`, which returns all matching elements in a new array, `find()` stops as soon as it encounters the first match and returns it.

- **Purpose**: Locate a single element based on a condition.
- **Parameters**:
  - `callback`: A function that tests each element.
  - The callback receives the following arguments:
    - `currentValue`: The current element being processed.
    - `index` (optional): The index of the current element.
    - `array` (optional): The array on which `find()` is called.
- **Example**:

```
const movements = [200, -100, 300, -50, 400];

const firstWithdrawal = movements.find((mov) => mov < 0);
console.log(firstWithdrawal); // -100
```

Explanation:

- The `find()` method checks each element (`mov`) to see if it is less than 0.
- The first element matching this condition is returned (`-100`).

`some()` and `every()` Methods

These methods are used for testing elements in an array against a condition.

`some()`

- **Purpose**: Checks if **any** element in the array matches the condition.
- **Returns**: `true` if at least one element satisfies the condition; otherwise, `false`.
- **Example**:

```
const movements = [200, -100, 300, -50, 400];

const anyDeposits = movements.some((mov) => mov > 0);
console.log(anyDeposits); // true
```

`every()`

- **Purpose**: Checks if **all** elements in the array match the condition.
- **Returns**: `true` if all elements satisfy the condition; otherwise, `false`.
- **Example**:

```
const movements = [200, 100, 300, 50, 400];

const allDeposits = movements.every((mov) => mov > 0);
console.log(allDeposits); // true
```

Flattening Arrays

Flattening arrays involves converting nested arrays into a single-level array.

1. `flat()`

- **Purpose**: Flattens nested arrays into a single array up to a specified depth.
- **Parameters**: `depth` (optional, defaults to 1) specifies how deep the flattening should go.
- **Example**:

```
const arrDeep = [[[1, 2], 3], [4, [5, 6]], 7, 8];

console.log(arrDeep.flat()); // [ [1, 2], 3, 4, [5, 6], 7, 8 ]
console.log(arrDeep.flat(2)); // [1, 2, 3, 4, 5, 6, 7, 8]
```

2. `flatMap()`

- **Purpose**: Combines the functionality of `map()` and `flat()` into a single operation. It applies a mapping function to each element, then flattens the result to a single level.
- **Example**:

```
const accounts = [
  { movements: [200, -100, 300] },
  { movements: [-50, 400, -150] },
];

const overallBalance = accounts
  .flatMap((acc) => acc.movements)
  .reduce((sum, mov) => sum + mov, 0);
console.log(overallBalance); // 600
```

Sorting Arrays

The `sort()` method is used to sort elements of an array in place. It can handle strings and numbers but requires a custom comparator for numerical sorting.

1. Sorting Strings

- **Default Behavior**: Sorts strings alphabetically (in Unicode order).
- **Example**:

```
const owners = ["Zach", "Alice", "John", "Mary"];
console.log(owners.sort()); // ["Alice", "John", "Mary", "Zach"]
```

2. Sorting Numbers

- **Purpose**: Sort numbers in ascending or descending order.
- **Custom Comparator**: Use a comparator function `a - b` for ascending order or `b - a` for descending order.
- **Example**:

```
const movements = [200, -100, 300, -50, 400];

movements.sort((a, b) => a - b); // Ascending
console.log(movements); // [-100, -50, 200, 300, 400]

movements.sort((a, b) => b - a); // Descending
console.log(movements); // [400, 300, 200, -50, -100]
```

Creating and Filling Arrays

1. `Array.from()`

- **Purpose**: Creates an array from an iterable or based on a length and mapping function.
- **Example**:

```
const z = Array.from({ length: 7 }, (_, i) => i + 1);
console.log(z); // [1, 2, 3, 4, 5, 6, 7]
```

2. `fill()`

- **Purpose**: Fills elements of an array with a specified value between a start and an end index.
- **Example**:

```
const arr = new Array(10);
arr.fill(23, 2, 6); // Fill with 23 from index 2 to index 6
console.log(arr); // [empty × 2, 23, 23, 23, 23, empty × 4]
```

Practicing Array Methods

1. Sum of Bank Deposits

- **Purpose**: Calculate the total sum of all deposits across accounts.
- **Example**:

```
const bankDepositSum = accounts
  .flatMap((acc) => acc.movements)
  .filter((mov) => mov > 0)
  .reduce((sum, cur) => sum + cur, 0);
console.log(bankDepositSum); // Total deposits
```

2. Count Deposits ≥ 1000

- **Purpose**: Count how many deposits are greater than or equal to 1000.
- **Example**:

```
const numDeposits1000 = accounts
  .flatMap((acc) => acc.movements)
  .reduce((count, cur) => (cur >= 1000 ? ++count : count), 0);
console.log(numDeposits1000); // Number of deposits >= 1000
```

3. Convert to Title Case

- **Purpose**: Convert strings to title case, excluding specified exceptions.
- **Example**:

```
const convertTitleCase = (title) => {
  const exceptions = [
    "a",
    "an",
    "the",
    "and",
    "but",
    "or",
    "on",
    "in",
    "with",
  ];
  return title
    .toLowerCase()
    .split(" ")
    .map((word) =>
      exceptions.includes(word) ? word : word[0].toUpperCase() +
word.slice(1)
    )
    .join(" ");
};

console.log(convertTitleCase("this is a test title with and"));
// "This Is a Test Title With and"
```