Advanced JavaScript Array Methods — Master Guide

This document explains the MOST important and MOST used array methods in real projects and interviews.

Each method includes:

- Simple definition
- Syntax
- Input → Output
- Step-by-step explanation
- When to use
- Advantages & disadvantages
- Real-time use cases
- Interview-focused examples
- Practice problems

Methods covered:

1. **slice()**
2. **splice()**
3. **forEach()**
4. **map()**
5. **filter()**
6. **reduce()**
7. **some()**
8. **every()**
9. **find()**
10. **findIndex()**
11. **flat()**()**

---

1. slice()

Definition

Returns a portion of the array without modifying the original.

Syntax

```
array.slice(start, end);
```

Input → Output

Input:

```
[10, 20, 30, 40].slice(1, 3)
```

Output:

```
[20, 30]
```

Step-by-step

1. `start` → starting index (inclusive)
2. `end` → ending index (exclusive)
3. Creates a **new array**, does NOT change original

When to use

- When you need a safe copy of part of an array
- When you want to extract items for UI or calculation

Advantages

- Non-destructive
- Very predictable

Disadvantages

- Cannot insert/delete items

Real example

Extract first 5 products:

```
const firstFive = products.slice(0, 5);
```

---

2. splice()

Definition

Changes the original array by adding, removing, or replacing elements.

Syntax

```
array.splice(start, deleteCount, ...itemsToAdd)
```

Input → Output

Input:

```
let arr = [1,2,3,4];
arr.splice(1, 2, 9, 9);
```

Output:

```
Original modified → [1, 9, 9, 4]
```

Step-by-step

1. Start at index `1`
2. Remove `2` items → removes 2 & 3
3. Add `9` and `9`

When to use

- Editing original array
- Inserting elements
- Removing elements

Real-time example

Remove a deleted user from list:

```
users.splice(indexOfDeletedUser, 1);
```

---

3. forEach()

Definition

Runs a function on each element but **does not return** a new array.

Syntax

```
array.forEach((element, index, array) => {})
```

Input → Output

Input:

```
[10,20,30].forEach(x => console.log(x));
```

Output:

```
10
20
30
```

Step-by-step

1. Takes each element one by one
2. Executes callback
3. Does NOT return anything

When to use

- Logging
- Updating UI
- Running side effects

Disadvantages

- Cannot break
- Cannot return new array

---

4. map()

Definition

Creates a **new array** by transforming every element.

Syntax

```
array.map((element, index, array) => newValue)
```

Input → Output

Input:

```
[1,2,3].map(x => x * 2)
```

Output:

```
[2,4,6]
```

Step-by-step

1. Reads every element
2. Runs callback
3. Adds returned value to new array

When to use

- Transforming data from API
- Creating UI lists
- Data formatting

Interview point

forEach vs **map**:

- forEach → no return value
- map → returns new array

---

5. filter()

Definition

Returns a new array **only with elements that pass the condition**.

Syntax

```
array.filter((element, index, array) => condition)
```

Input → Output

Input:

```
[10, 25, 30, 5].filter(x => x > 20)
```

Output:

```
[25, 30]
```

When to use

- Filter users by age
- Filter products by price range
- Remove unwanted values

Interview point

filter returns:

- new array
- may return empty array

---

6. reduce()

Definition

Combines all array values into a single result.

Syntax

```
array.reduce((accumulator, element, index, array) => newAccumulatorValue, initialValue)
```

Input → Output

Input:

```
[1,2,3,4].reduce((sum, x) => sum + x, 0)
```

Output:

```
10
```

Step-by-step

- accumulator starts as `initialValue`
- callback returns new accumulator
- final accumulator returned

Real-time use cases

- Sum of prices
- Grouping values
- Converting arrays to objects

---

7. some()

Definition

Checks if **at least one element** passes the condition.

Input → Output

Input:

```
[5,10,15].some(x => x > 12)
```

Output:

```
true
```

---

8. every()

Definition

Checks if **all elements** pass the condition.

Input → Output

Input:

```
[5,10,15].every(x => x > 3)
```

Output:

```
true
```

---

9. find()

Definition

Returns the **first element** that satisfies a condition.

Syntax

```
array.find((element, index, array) => condition)
```

Input → Output

Input:

```
[5,12,8,130].find(x => x > 10)
```

Output:

```
12
```

Step-by-step Explanation

1. Checks each element in order
2. Stops immediately when condition becomes true
3. Returns that element
4. If no match found → returns `undefined`

Real-time Use Cases

- Find first user with balance < 0
- Find first student who failed
- Find first product out of stock

Example 1 — Find first failing score

```
const scores = [45, 78, 32, 90];
const fail = scores.find(s => s < 35);
// Output: 32
```

Example 2 — Find user by ID

```
const users = [
  { id: 1, name: "A" },
  { id: 2, name: "B" }
];

const user = users.find(u => u.id === 2);
// Output: { id: 2, name: "B" }
```

---

10. findIndex()

Definition

Returns the **index** of the first matching element.

Syntax

```
array.findIndex((element, index, array) => condition)
```

Input → Output

Input:

```
[5,12,8,130].findIndex(x => x > 100)
```

Output:

```
3
```

Step-by-step Explanation

1. Loops through array
2. Checks condition
3. Returns index of first match
4. If none found → returns `-1`

Real-time Use Cases

- Find index of product to update
- Find index of user to delete
- Find position of item in sorted list

Example 1 — Find index of first even number

```
const arr = [1,3,7,8,10];
const idx = arr.findIndex(n => n % 2 === 0);
// Output: 3
```

Example 2 — Find index of username

```
const users = ["ram", "sam", "david"];
const index = users.findIndex(u => u === "sam");
// Output: 1
```

---

11. flat()

Definition

Removes nesting by returning a new flattened array.

Syntax

```
array.flat(depth)
```

`depth` = how many levels to flatten (default = 1)

Input → Output

Input:

```
[1, [2, 3], [4, [5]]].flat(2)
```

Output:

```
[1, 2, 3, 4, 5]
```

Step-by-step Explanation

1. Reads each item
2. If item is array → expands items into result
3. Continues until given depth
4. Does NOT change original array

Real-time Use Cases

- Flattening API data
- Converting nested category lists
- Cleaning deeply nested arrays

Example 1 — Flatten 1 level

```
const arr = [1, [2,3], 4];
const flat = arr.flat();
// Output: [1,2,3,4]
```

Example 2 — Flatten many levels

```
const deep = [1, [2, [3, [4]]]];
const flat = deep.flat(3);
// Output: [1,2,3,4]
```

---

12. Real-Time Scenarios

1. Find duplicates

```
function findDuplicates(arr) {
  let result = [];
  let seen = [];
  arr.forEach(item => {
    if (seen.includes(item) && !result.includes(item)) {
      result.push(item);
    }
    seen.push(item);
  });
  return result;
}
```

2. Filter active users

```
const activeUsers = users.filter(u => u.active === true);
```

3. Calculate total cart price

```
const total = cart.reduce((sum, item) => sum + item.price * item.qty, 0);
```

4. Transform API data

```
const names = users.map(u => u.name);
```

---

13. Comparison Table (map vs filter vs reduce vs forEach)

Method	Returns New Array?	Purpose	Common Use Case
forEach	❌No	Looping, side effects	Logging, updating values
map	✅Yes	Transform each item	Formatting API data
filter	✅Yes	Keep only items that match	Filtering products/users
reduce	⚠️One value	Combine items	Totals, grouping, frequency count

---

14. “Which Method Should I Use?” Flowchart

Goal: Transform each value → use map

Goal: Keep only some values → use filter

Goal: Combine all values into one → use reduce

Goal: Just loop without returning → use forEach

Goal: Find first matching value → use find

Goal: Find index of matching value → use findIndex

Goal: Check if at least one value matches → use some

Goal: Check if all values match → use every

Goal: Remove array nesting → use flat

---

15. Deep Real-Time Company Tasks

Task 1 — Group products by category (reduce)

```
function groupByCategory(products) {
  return products.reduce((acc, item) => {
    if (!acc[item.category]) acc[item.category] = [];
    acc[item.category].push(item);
    return acc;
  }, {});
}
```

Task 2 — Pagination logic (slice)

```
function paginate(data, page, size) {
  const start = (page - 1) * size;
  const end = start + size;
  return data.slice(start, end);
}
```

Task 3 — Remove inactive users (filter)

```
const active = users.filter(u => u.active);
```

Task 4 — Convert array to lookup map (reduce)

```
function toMap(arr) {
  return arr.reduce((acc, obj) => {
    acc[obj.id] = obj;
    return acc;
  }, {});
}
```

Task 5 — Find out-of-stock product (find)

```
const outOfStock = products.find(p => p.stock === 0);
```

---

16. Advanced Examples

Example 1 — Frequency Count (reduce)

```
function frequency(arr) {
  return arr.reduce((acc, item) => {
    acc[item] = (acc[item] || 0) + 1;
    return acc;
  }, {});
}
```

Example 2 — Unique Values (filter + indexOf)

```
function unique(arr) {
  return arr.filter((item, index) => arr.indexOf(item) === index);
}
```

Example 3 — Sorting by marks (sort)

```
function sortByMarks(arr) {
  return arr.sort((a, b) => a.marks - b.marks);
}
```

Example 4 — Flatten & filter together

```
const clean = data.flat(3).filter(x => x != null);
```

const names = users.map(u => u.name); ```

---

11. Practice Problems

1. Use map to increase all salaries by 10%.
2. Use filter to return numbers > 50.
3. Use reduce to count occurrences of each category.
4. Use some to detect if array contains negative numbers.
5. Use every to check if all scores are above 40.

---

12. Summary

This document covers: core behavior, differences, real-time use cases, interview pointers, and practice tasks to help students master JavaScript array methods.