

Array Methods

🎓 **CS Perspective:** This section covers **functional programming operations** on collections. `map`, `filter`, and `reduce` are the canonical **higher-order functions** from functional programming—they abstract common iteration patterns. `reduce` (also called `fold`) is particularly powerful: theoretically, any list operation can be expressed as a reduce. These methods favor **immutability** (creating new arrays rather than mutating), which aligns with functional programming principles and helps prevent bugs. Understanding method chaining relates to **fluent interfaces** and the **builder pattern**. The distinction between mutating (`splice`, `reverse`) and non-mutating (`slice`, `map`) methods is crucial for **referential transparency** and predictable code.

📌 Simple Array Methods

`slice` (Does NOT mutate)

Returns a portion of array:

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

arr.slice(2);      // ['c', 'd', 'e']
arr.slice(2, 4);   // ['c', 'd']
arr.slice(-2);     // ['d', 'e']
arr.slice(-1);     // ['e']
arr.slice(1, -2);  // ['b', 'c']
arr.slice();       // ['a', 'b', 'c', 'd', 'e'] (shallow copy)
```

`splice` (MUTATES original)

Removes/replaces elements:

```
const arr = ['a', 'b', 'c', 'd', 'e'];
arr.splice(-1);    // Removes last element
arr.splice(1, 2);  // Removes 2 elements starting at index 1
console.log(arr);  // ['a', 'd']
```

`reverse` (MUTATES)

```
const arr = ['a', 'b', 'c'];
arr.reverse();    // ['c', 'b', 'a']
```

`concat` (Does NOT mutate)

```
const letters = arr1.concat(arr2);  
// Same as: [...arr1, ...arr2]
```

join

```
['a', 'b', 'c'].join(' - '); // 'a - b - c'
```

at Method (ES2022)

```
const arr = [23, 11, 64];  
arr.at(0);    // 23  
arr.at(-1);   // 64 (last element)  
  
// Works on strings too  
'jonas'.at(-1); // 's'
```

forEach

```
const movements = [200, -150, 400, -300];  
  
movements.forEach(function(mov, i, arr) {  
  if (mov > 0) {  
    console.log(`Movement ${i + 1}: Deposited ${mov}`);  
  } else {  
    console.log(`Movement ${i + 1}: Withdrew ${Math.abs(mov)}`);  
  }  
});
```

forEach vs for-of

forEach	for-of
Can't use break/continue	Can use break/continue
Cleaner syntax	More flexible
Access to index built-in	Need .entries() for index

forEach with Maps and Sets

```
// Map  
currencies.forEach(function(value, key, map) {
```

```
    console.log(`${key}: ${value}`);
  });

  // Set (key = value)
  currenciesUnique.forEach(function(value, _, set) {
    console.log(`${value}`);
  });
```

map Method

Creates new array by transforming each element:

```
const movements = [200, -150, 400];
const eurToUsd = 1.1;

const movementsUSD = movements.map(mov => mov * eurToUsd);
// [220, -165, 440]

// With index
const descriptions = movements.map((mov, i) =>
  `Movement ${i + 1}: ${mov > 0 ? 'Deposited' : 'Withdrew'}
  ${Math.abs(mov)}`
);
```

filter Method

Creates new array with elements that pass test:

```
const movements = [200, -150, 400, -300];

const deposits = movements.filter(mov => mov > 0);
// [200, 400]

const withdrawals = movements.filter(mov => mov < 0);
// [-150, -300]
```

reduce Method

Reduces array to single value:

```
const movements = [200, -150, 400, -300];

// Sum
const balance = movements.reduce((acc, cur) => acc + cur, 0);
```

```
// 150

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

reduce Anatomy

```
array.reduce((accumulator, currentValue, index, array) => {
  // return new accumulator value
}, initialValue);
```

Method Chaining

```
const eurToUsd = 1.1;

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

⚠ Don't chain methods that mutate (splice, reverse, sort)

find Method

Returns **first element** that matches:

```
const movements = [200, -150, 400, -300];
const firstWithdrawal = movements.find(mov => mov < 0);
// -150

const accounts = [{ owner: 'Jonas' }, { owner: 'Jessica' }];
const account = accounts.find(acc => acc.owner === 'Jessica');
// { owner: 'Jessica' }
```

find vs filter

find	filter
Returns first match	Returns all matches

find

Returns element

filterReturns array

 **findIndex Method**

Returns index of first match:

```
const index = accounts.findIndex(acc => acc.username === 'js');  
accounts.splice(index, 1); // Remove account at index
```

 **findLast & findLastIndex (ES2023)**

```
const movements = [200, -150, 400, -300, -100];  
  
const lastWithdrawal = movements.findLast(mov => mov < 0);  
// -100  
  
const lastLargeIndex = movements.findLastIndex(mov => Math.abs(mov) >  
300);  
// 3
```

 **some & every**

some - At least one passes

```
const movements = [200, -150, 400, -300];  
  
movements.some(mov => mov > 0); // true  
movements.some(mov => mov > 500); // false
```

every - All must pass

```
movements.every(mov => mov > 0); // false  
  
const allDeposits = [200, 400, 100];  
allDeposits.every(mov => mov > 0); // true
```

 **flat & flatMap**

flat

Flattens nested arrays:

```
const arr = [[1, 2, 3], [4, 5, 6], 7, 8];
arr.flat(); // [1, 2, 3, 4, 5, 6, 7, 8]

const deep = [[[1, 2], 3], [4, [5, 6]]];
deep.flat(2); // [1, 2, 3, 4, 5, 6] (depth = 2)
```

flatMap

Combines map + flat (1 level):

```
const allMovements = accounts.flatMap(acc => acc.movements);
```

sort Method

⚠ **MUTATES** original array!

Strings (default)

```
const owners = ['Jonas', 'Zach', 'Adam'];
owners.sort(); // ['Adam', 'Jonas', 'Zach']
```

Numbers (need callback)

```
const movements = [200, -150, 400, -300];

// Ascending
movements.sort((a, b) => a - b);
// [-300, -150, 200, 400]

// Descending
movements.sort((a, b) => b - a);
// [400, 200, -150, -300]
```

Sort Logic

- Return < 0: a before b (keep order)
- Return > 0: b before a (switch)

📌 Non-Mutating Alternatives (ES2023)

```
const movements = [200, -150, 400];

// toReversed (instead of reverse)
const reversed = movements.toReversed();

// toSorted (instead of sort)
const sorted = movements.toSorted((a, b) => a - b);

// toSpliced (instead of splice)
const spliced = movements.toSpliced(1, 1);

// with (instead of bracket notation)
const newMov = movements.with(1, 1000);
// [200, 1000, 400]
```

📌 Creating Arrays

```
// Empty array + fill
const x = new Array(7);           // [empty × 7]
x.fill(1);                        // [1, 1, 1, 1, 1, 1, 1]
x.fill(1, 3, 5);                  // [empty, empty, empty, 1, 1, empty, empty]

// Array.from
const y = Array.from({ length: 7 }, () => 1);
// [1, 1, 1, 1, 1, 1, 1]

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

// Convert NodeList to Array
const movementsUI = Array.from(
  document.querySelectorAll('.movements__value'),
  el => Number(el.textContent)
);
```

📌 Array Grouping (ES2024)

```
const movements = [200, -150, 400, -300];

const grouped = Object.groupBy(movements, mov =>
  mov > 0 ? 'deposits' : 'withdrawals'
);
// { deposits: [200, 400], withdrawals: [-150, -300] }
```

```
// With objects
const groupedAccounts = Object.groupBy(accounts, ({ type }) => type);
```

includes vs indexOf

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

arr.includes(2); // true (boolean)
arr.indexOf(2); // 1 (index or -1)
```

Which Method to Use?

Goal	Method
Loop without new array	<code>forEach</code>
Transform elements	<code>map</code>
Filter elements	<code>filter</code>
Reduce to single value	<code>reduce</code>
Find first match	<code>find</code>
Find index	<code>findIndex</code>
Check if any match	<code>some</code>
Check if all match	<code>every</code>
Flatten nested	<code>flat</code> / <code>flatMap</code>
Sort	<code>sort</code> / <code>toSorted</code>
Check existence	<code>includes</code>
Group elements	<code>Object.groupBy</code>

Exam Tips

1. **slice** doesn't mutate, **splice** does
2. **map** returns new array, **forEach** returns undefined
3. **find** returns element, **filter** returns array
4. **some** = at least one, **every** = all must pass
5. **sort** mutates! Use **toSorted** for non-mutating
6. **reduce** accumulator is first param, initial value is second
7. **flat(depth)** - default depth is 1
8. **Array.from** can convert iterables AND transform

9. Use **includes** for existence, **indexOf** for position
10. Chain methods left to right, each returns a new array
11. **flatMap** = map + flat(1) combined
12. Return < 0 in sort keeps order, > 0 switches