DevSheets

JavaScript Array Methods

Transform, search and edit.

01 map()

```
[1, 2, 3].map(x => x * 2);      // [2, 4, 6]
const users = [{ name: 'John', age: 30 }];
users.map(u => u.name);         // ['John']
['a', 'b'].map((v, i) => `${i}: ${v}`);
// ['0: a', '1: b']
```

New array of the same length; the original is untouched. The callback gets (value, index, array). An async callback gives promises: await Promise.all(arr.map(fn)).

02 filter()

```
[1, 2, 3, 4].filter(x => x % 2 === 0);
// [2, 4]
[0, 1, '', 'hi', null].filter(Boolean);
// [1, 'hi']
[1, 2, 2, 3].filter((v, i, arr) =>
  arr.indexOf(v) === i);  // [1, 2, 3]
```

New array with the elements that pass the test, [] if none do. filter(Boolean) drops falsy values.

03 reduce()

```
const add = (sum, x) => sum + x;
[1, 2, 3].reduce(add, 0);  // 6
[].reduce(add, 0);         // 0
[].reduce(add);            // TypeError
['a', 'b', 'a'].reduce((acc, x) => {
  acc[x] = (acc[x] || 0) + 1;
  return acc;
}, {});                    // { a: 2, b: 1 }
```

The callback returns the next accumulator. With no initial value the first element is used and an empty array throws, so pass one. For grouping, see Object.groupBy (18).

04 find, findIndex & findLast

```
const nums = [1, 2, 3, 4];
nums.find(x => x > 1);          // 2
nums.findIndex(x => x > 1);     // 1
nums.find(x => x > 9);          // undefined
nums.findIndex(x => x > 9);     // -1
nums.findLast(x => x < 4);      // 3
nums.findLastIndex(x => x < 4); // 2
```

find returns the first match, findIndex its index, and both stop there. findLast and findLastIndex (ES2023) search from the end, with no reversed copy.

05 some() & every()

```
[1, 2, 3].some(x => x % 2 === 0);  // true
[1, 2, 3].every(x => x > 0);       // true
[].some(x => x > 0);               // false
[].every(x => x > 0);              // true
const valid = fields.every(f => f.value);
```

Both return a boolean and stop early. On an empty array some is false and every is true.

06 includes() & indexOf()

```
const arr = [1, 2, 3, 2, NaN];
arr.includes(2);      // true
arr.indexOf(2);       // 1
arr.indexOf(2, 2);    // 3 (from index 2)
arr.lastIndexOf(2);   // 3
arr.includes(NaN);    // true
arr.indexOf(NaN);     // -1
```

includes finds NaN (it compares with SameValueZero); indexOf and lastIndexOf use === and never do. -1 means not found. For many lookups, build a Set and use has().

07 push, pop, shift & unshift

```
const arr = [1, 2, 3];
arr.push(4, 5);     // 5 (new length)
arr.pop();          // 5 (removed element)
arr.unshift(0);     // 5 (new length)
arr.shift();        // 0 (removed element)
// arr is [1, 2, 3, 4]
```

All four mutate. shift and unshift re-index every element, so they are slower. To copy instead: [...arr, x].

08 splice()

```
const arr = [1, 2, 3, 4, 5];
arr.splice(1, 2);            // [2, 3]
// arr is [1, 4, 5]
arr.splice(1, 0, 'a', 'b');  // []
// arr is [1, 'a', 'b', 4, 5]
arr.splice(1, 2, 'x');       // ['a', 'b']
// arr is [1, 'x', 4, 5]
```

splice(start, deleteCount, ...items) mutates and returns the removed elements. A negative start counts from the end; with no deleteCount it removes to the end.

09 slice(), concat() & at()

```
const arr = [1, 2, 3, 4, 5];
arr.slice(1, 3);    // [2, 3] (end excluded)
arr.slice(-2);      // [4, 5]
arr.at(-1);         // 5
arr[-1];            // undefined
arr.concat([6], 7); // [1, 2, 3, 4, 5, 6, 7]
```

None of these change arr. slice and at count negative indexes from the end. slice() and [...arr] copy the array, but only one level deep: nested objects are shared.

DevSheets

JavaScript Array Methods

Sort, copy, create and group.

10 Mutate or return a new array?

`push, unshift`	Mutate, return the new length
`pop, shift`	Mutate, return the removed item
`splice`	Mutates, returns removed items
`sort, reverse`	Mutate, return the same array
`fill, copyWithin`	Mutate, return the same array

No other method changes the array. map, filter, slice, concat, flat and flatMap return a new array, as do toSorted, toReversed, toSpliced and with (12). Copies are shallow.

11 sort() & reverse()

```
const nums = [10, 9, 1];
nums.sort();                  // [1, 10, 9]
nums.sort((a, b) => a - b);   // [1, 9, 10]
nums.sort((a, b) => b - a);   // [10, 9, 1]
nums.reverse();               // [1, 9, 10]
['b', 'A'].sort();            // ['A', 'b']
```

The default order compares elements as strings by UTF-16 code unit, so numbers need a compare function. Names: (a, b) => a.name.localeCompare(b.name). Stable since ES2019.

12 Copying methods (ES2023)

```
const nums = [3, 1, 2];
nums.toSorted();          // [1, 2, 3]
nums.toReversed();        // [2, 1, 3]
nums.toSpliced(0, 1);     // [1, 2]
nums.with(0, 99);         // [99, 1, 2]
// nums is [3, 1, 2]
```

Same arguments as sort, reverse and splice; with(i, x) is a copying arr[i] = x. with(-1, x) sets the last element. Node 20+.

13 forEach()

```
['a', 'b'].forEach((val, i) => {
  console.log(`${i}: ${val}`);
});  // logs 0: a, then 1: b
urls.forEach(async url => {
  await fetch(url);
});  // returns before any fetch finishes
for (const url of urls) await fetch(url);
```

Returns undefined and cannot break: use for...of or some. Async callbacks are not awaited: for...of awaits each in turn, Promise.all(urls.map(fn)) runs them together.

14 flat() & flatMap()

```
const deep = [1, [2, [3, [4]]]];
deep.flat();          // [1, 2, [3, [4]]]
deep.flat(2);         // [1, 2, 3, [4]]
deep.flat(Infinity);  // [1, 2, 3, 4]
[1, 2].flatMap(x => [x, x * 2]);
// [1, 2, 2, 4]
```

flat removes one level by default and drops empty slots. flatMap is map then flat(1): return [] to skip an element.

15 join() & split()

```
[1, 2, 3].join();          // '1,2,3'
[1, 2, 3].join(' - ');     // '1 - 2 - 3'
[1, null, 3].join('');     // '13'
'a,b,c'.split(',');        // ['a', 'b', 'c']
'a-b-c'.split('-', 2);     // ['a', 'b']
[...'hey'];                // ['h', 'e', 'y']
```

null and undefined join as empty strings. split('') cuts by UTF-16 code unit and breaks emoji; [...str] and Array.from(str) split by code point.

16 Array.from() & Array.of()

```
Array.from('abc');    // ['a', 'b', 'c']
Array.of(7);          // [7]
Array(3);             // [ <3 empty items> ]
Array(3).map(x => 0); // [ <3 empty items> ]
Array.from({ length: 3 }, (_, i) => i * 2);
// [0, 2, 4]
```

Array.from takes any iterable or array-like (Set, Map, NodeList, arguments) plus an optional map function. Array(3) only sets length: map and forEach skip its empty slots.

17 fill() & copyWithin()

```
[1, 2, 3, 4].fill(0, 1, 3);  // [1, 0, 0, 4]
const rows = Array(2).fill([]);
rows[0].push(1);
// rows is [[1], [1]]
[1, 2, 3, 4, 5].copyWithin(1, 3, 4);
// [1, 4, 3, 4, 5]
```

fill(value, start, end) and copyWithin(target, start, end) both mutate and stop before end. Every slot gets the same object: use Array.from({ length: 2 }, () => []) for separate ones.

18 groupBy & Array.fromAsync

```
Object.groupBy([1, 2, 3], x =>
  x % 2 ? 'odd' : 'even');
// { odd: [1, 3], even: [2] }
Map.groupBy(users, u => u.team); // ES2024
async function* gen() { yield 1; yield 2; }
await Array.fromAsync(gen());    // [1, 2]
```

Object.groupBy and Map.groupBy are ES2024 (Node 21+); Map keys can be objects. Array.fromAsync is ES2026 (Node 22+) and awaits items one at a time, unlike Promise.all.