Array Methods: map, filter, reduce

The factory line for arrays: map transforms, filter selects, reduce folds everything into one value — and a real app shape using all three.

Introduction

Think of an array as items moving on a **factory belt**. A **callback** function is a machine placed on that belt: every item passes through it once, and the machine does one small job to each item. The three machines in this lesson are `map`, `filter` and `reduce` — together they handle most of the array work in real apps, and they are the heart of the React series later. *(Roman Urdu: map badalta hai, filter chhanta hai, reduce sab ko aik value mein samet deta hai.)*

Key points

- **The callback is the machine:** you pass a small function into the method, and the method calls it **once for every item**, in order. You never call it yourself.
- `map` **transforms:** it runs the callback on every item and collects the results into a **new array** of the same length. `map` never edits the original — it builds new.
  (Roman Urdu: map har cheez ko badal kar aik nayi list banata hai, purani list waisi ki waisi rehti hai.)
- `filter` **selects:** it is a net. The callback returns `true` or `false` for each item — `true` items stay in the new array, `false` items fall out. The new array can be shorter than the original.
- `reduce` **folds into one value:** a snowball. It carries an **accumulator** that starts at your start value, and each item folds into it, until one single value is left at the end.
- **Always give** `reduce` **a start value:** `reduce((sum, price) => sum + price, 0)` — the `0` at the end is the start. Forget it and the **first item becomes the start**, which breaks empty arrays and can let strings sneak into a total.
- **Chain them:** the result of `filter` is an array, so `map` can run on it, and `reduce` can finish the job — one clean line of steps.
- **Loops vs methods:** a `for` loop *can* do all of this, but a loop says **how** (step by step), while methods say **what** (transform, select, total) — shorter and clearer to read.
  (Roman Urdu: Loop tareeqa batata hai, methods matlab batate hain — is liye methods zyada saaf lagte hain.)

Code examples

Example 1 — map doubles every price

```
const prices = [250, 200, 300];

const doubled = prices.map((price) => price * 2);

console.log(doubled.join(", "));
console.log(prices.join(", "));
```

What it does: The callback `price => price * 2` is the machine on the belt: `250` goes in, `500` comes out; `200` becomes `400`; `300` becomes `600`. `map` collects those three results into a new array called `doubled` — `prices` itself is never touched.

What you will see: the console prints `500, 400, 600` first, then `250, 200, 300` — proof that the original array is exactly as it was.

Example 2 — filter keeps only what passes

```
const products = [
  { name: "Chai", price: 250, stock: 5 },
  { name: "Coffee", price: 200, stock: 0 },
  { name: "Cake", price: 300, stock: 3 },
];

const inStock = products.filter((product) => product.stock > 0);
console.log(inStock.map((product) => product.name).join(", "));

const above220 = products.filter((product) => product.price > 220);
console.log(above220.map((product) => product.name).join(", "));
```

What it does: The first `filter` keeps only products whose `stock` is more than `0` — Coffee returns `false` and falls out of the net. The second `filter` uses a different test: only prices above `220` stay. The small `map` after each filter just picks out the names so we can print them.

What you will see: the console prints `Chai, Cake` first (Coffee is sold out), then `Chai, Cake` again (Coffee at `200` is also below `220`).

Example 3 — reduce, then all three in a real app shape

```
const prices = [250, 200, 300];
const total = prices.reduce((sum, price) => sum + price, 0);
console.log(total);

const products = [
  { name: "Chai", price: 250, stock: 5 },
  { name: "Coffee", price: 200, stock: 0 },
  { name: "Cake", price: 300, stock: 3 },
];

const cards = products
  .filter((product) => product.stock > 0)
  .map((product) => product.name + " - Rs " + product.price);
console.log(cards.join(" | "));

const bill = products
  .filter((product) => product.stock > 0)
  .reduce((sum, product) => sum + product.price, 0);
console.log("TOTAL: Rs " + bill);
```

What it does: Follow the snowball in the first `reduce`: the accumulator starts at `0`, then folds in each price — `0`, then `250`, then `450`, then `750`. The app shape below it is how real screens are built: `filter` hides the sold-out Coffee, `map` turns each product that is left into card text, and a second chain uses `filter` plus `reduce` to print the bill total.

What you will see: the console prints `750`, then `Chai - Rs 250 | Cake - Rs 300`, then `TOTAL: Rs 550` — Coffee appears nowhere, because the filter removed it before the other machines ran.

Quick recap

- `map` = transform every item into a **new array**; the original never changes.
- `filter` = keep only the items whose callback returns `true`.
- `reduce` = fold everything into **one value** — and always give it a start value.

Common mistakes

1. **Forgetting the start value in reduce** — without the `, 0` at the end, the first item becomes the accumulator, so an empty array throws an error and a list of strings can join into your "total" by accident. The fix: always write the start value, even when it is `0` or `""`.
2. **Expecting map or filter to change the original array** — they never edit; they return a new array. If you call `prices.map(...)` and throw the result away, nothing happened. The fix: save the result in a variable and use that.

3. **Forgetting to return in a block-body callback** — `(price) => { price * 2; }` returns `undefined` for every item, so your new array is full of `undefined`. The fix: either use the one-line form `(price) => price * 2`, or write the word `return` inside the braces.

Practice

1. Start with `const marks = [45, 80, 62, 90, 30]`. Use `filter` to keep only the passing marks (50 and above), use `map` to add 5 bonus marks to each of them, and print both new arrays. Check that `marks` itself did not change.
2. Make an array of four products with `name`, `price` and `stock`. Use one chain to print the names of only the in-stock products, and use `reduce` to print the total price of the in-stock products only.

Practice quiz

1. What does map do to the original array?

- It edits the original in place
- Nothing, it builds a new array of the same length and leaves the original unchanged
- It empties the original
- It sorts the original

Answer: Nothing, it builds a new array of the same length and leaves the original unchanged

2. What does filter keep?

- Every item, always
- Only the items whose callback returns true, in a new array
- Only the first item
- Only items that are numbers

Answer: Only the items whose callback returns true, in a new array

3. Why must reduce always be given a start value, like the 0 in reduce((sum, price) => sum + price, 0)?

- It is only a style habit
- Without it the first item becomes the start, which breaks empty arrays and can let strings sneak into a total
- Without it reduce returns undefined always
- The start value is the answer itself

Answer: Without it the first item becomes the start, which breaks empty arrays and can let strings sneak into a total

---

Watch the video lesson: https://youtu.be/K8nM4H9XyI8

Read online: https://scopetreasure.com/notes/js-array-methods · More free tools and notes at https://scopetreasure.com