JavaScript Array Methods — Real-Time Company Tasks (Step-by-Step)

This document is **only for real-time style tasks** using array methods.

Each task includes: - Problem statement (company-style) - What we want to achieve - Methods used (map / filter / reduce / slice / find / etc.) - Step-by-step logic in simple English - Input → Output example - Final function code

You can export this as a separate PDF and share with students.

Task 1 — Group Products by Category (reduce)

1. Problem Statement

A company has a list of products. Each product has a **name**, **price**, and **category**. We want to **group products by category** to show them in different sections (Electronics, Grocery, etc.).

2. What We Want to Achieve

Convert this:

```
[
  { name: "TV", category: "electronics" },
  { name: "Banana", category: "grocery" },
  { name: "Mobile", category: "electronics" }
]
```

into this:

```
{
  electronics: [ {…}, {…} ],
  grocery: [ {…} ]
}
```

3. Methods Used

- **reduce()** → to convert an array into **one object** (grouped by category).

Callback parameters in reduce: - `acc` → accumulator (object we are building) - `item` → current product

4. Step-by-Step Logic

1. Start with an **empty object**: `{}`.
2. For each product:
3. Look at `item.category`.
4. If `acc` does not have that category yet → create empty array.
5. Push current product into that category array.
6. Return `acc` at the end.

5. Input → Output Example

Input:

```
const products = [
  { name: "TV", category: "electronics" },
  { name: "Banana", category: "grocery" },
  { name: "Mobile", category: "electronics" }
];
```

Output:

```
{
  electronics: [
    { name: "TV", category: "electronics" },
    { name: "Mobile", category: "electronics" }
  ],
  grocery: [
    { name: "Banana", category: "grocery" }
  ]
}
```

6. Final Function

```
function groupByCategory(products) {
  return products.reduce(function (acc, item) {
    const key = item.category; // category name

    // if category key not present, create it
    if (!acc[key]) {
      acc[key] = [];
    }

    // push current product into correct category
    acc[key].push(item);

    return acc; // return updated accumulator
```

```
  }, {});
}
```

Task 2 — Pagination Logic (slice)

1. Problem Statement

We have a big array of data (for example, **100 users**). We want to show only **10 users per page**.

2. What We Want to Achieve

Given: - page number (starting from 1) - page size (items per page)

We return only that page's items.

3. Methods Used

- **slice(start, end)** → to cut a part of array without changing original.

4. Step-by-Step Logic

1. `startIndex = (page - 1) * size`
2. `endIndex = startIndex + size`
3. Use `data.slice(startIndex, endIndex)`

5. Input → Output Example

Input:

```
const users = [1,2,3,4,5,6,7,8,9,10,11,12];
paginate(users, 2, 5);
```

Output:

```
[6,7,8,9,10] // page 2, 5 per page
```

6. Final Function

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

Task 3 — Convert Array to Fast Lookup Map (reduce)

1. Problem Statement

Company has user list as an array. But lookup by ID is slow if we always loop. We want an **object map** where key is `id`.

2. What We Want to Achieve

Convert this:

```
[
  { id: 1, name: "A" },
  { id: 2, name: "B" }
]
```

into this:

```
{
  1: { id: 1, name: "A" },
  2: { id: 2, name: "B" }
}
```

3. Methods Used

- **reduce()** → build a single object from many elements.

4. Step-by-Step Logic

1. Start with empty object `{}`.
2. For each user, use `user.id` as key.
3. Store whole user as value.

5. Input → Output Example

Input:

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

Output:

```
{
  1: { id: 1, name: "A" },
```

```
  2: { id: 2, name: "B" }
}
```

6. Final Function

```
function arrayToMap(arr) {
  return arr.reduce(function (acc, obj) {
    acc[obj.id] = obj; // store by id
    return acc;
  }, {});
}
```

Task 4 — Remove Inactive Users (filter)

1. Problem Statement

We have user list with `active: true/false`. We want only active users on dashboard.

2. What We Want to Achieve

From:

```
[
  { name: "A", active: true },
  { name: "B", active: false }
]
```

To:

```
[
  { name: "A", active: true }
]
```

3. Methods Used

- **filter()** → keep only values that pass condition.

4. Step-by-Step Logic

1. For each user, check `user.active`.
2. If `true` → keep it.
3. If `false` → skip it.

5. Input → Output Example

Input:

```
const users = [
  { name: "A", active: true },
  { name: "B", active: false },
  { name: "C", active: true }
];
```

Output:

```
[
  { name: "A", active: true },
  { name: "C", active: true }
]
```

6. Final Function

```
function getActiveUsers(users) {
  return users.filter(function (user) {
    return user.active === true;
  });
}
```

Task 5 — Calculate Total Cart Price (reduce)

1. Problem Statement

In e-commerce, each cart item has `price` and `qty`. We must find total bill amount.

2. What We Want to Achieve

From:

```
[
  { name: "Pen", price: 10, qty: 2 },
  { name: "Book", price: 50, qty: 1 }
]
```

To total:

```
70
```

3. Methods Used

- **reduce()** → sum all `(price * qty)` values.

4. Step-by-Step Logic

1. Set starting total as `0`.
2. For each item, calculate `item.price * item.qty`.
3. Add it to running total.
4. Return final total.

5. Input → Output Example

Input:

```
const cart = [
  { name: "Pen", price: 10, qty: 2 },
  { name: "Book", price: 50, qty: 1 }
];
```

Output:

```
70
```

6. Final Function

```
function getCartTotal(cart) {
  return cart.reduce(function (sum, item) {
    const itemTotal = item.price * item.qty;
    return sum + itemTotal;
  }, 0);
}
```

Task 6 — Find Duplicate Values (forEach + includes)

1. Problem Statement

We have an array of IDs. Some IDs are repeated. We want a list of IDs that appear more than once.

2. What We Want to Achieve

From:

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

To:

```
[2,1] // order can vary
```

3. Methods Used

- **forEach()** → loop through all IDs.
- **includes()** → check if value exists in array.

4. Step-by-Step Logic

1. Create `seen = []` to store all items.
2. Create `duplicates = []` to store only repeated ones.
3. For each id:
4. If `seen` already has it **and** `duplicates` does not have it yet → push to `duplicates`.
5. Add id to `seen`.

5. Final Function

```
function findDuplicates(arr) {
  let seen = [];
  let duplicates = [];

  arr.forEach(function (item) {
    if (seen.includes(item) && !duplicates.includes(item)) {
      duplicates.push(item);
    }
    seen.push(item);
  });

  return duplicates;
}
```

Task 7 — Sorting Students by Marks (sort)

1. Problem Statement

HR team wants student list sorted by marks from lowest to highest.

2. What We Want to Achieve

From:

```
[
  { name: "A", marks: 50 },
  { name: "B", marks: 80 },
```

```
  { name: "C", marks: 60 }
]
```

To:

```
[
  { name: "A", marks: 50 },
  { name: "C", marks: 60 },
  { name: "B", marks: 80 }
]
```

3. Methods Used

- **sort(compareFn)** → sort based on custom logic.

4. Step-by-Step Logic

1. `Array.sort` takes two items `a` and `b`.
2. If we return:
3. `a.marks - b.marks` → ascending order.
4. sort modifies the original array.

5. Final Function

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

Task 8 — Flatten and Clean Nested Data (flat + filter)

1. Problem Statement

API returns nested array with some `null` or `undefined` values. We want one clean, flat array.

2. What We Want to Achieve

From:

```
[1, [2, null], [3, [4, undefined]]]
```

To:

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

3. Methods Used

- **flat(depth)** → remove nesting.
- **filter()** → remove `null` / `undefined`.

4. Step-by-Step Logic

1. Use `flat(3)` (or higher) to flatten all levels.
2. Use `filter(x => x != null)` to remove `null` and `undefined`.

5. Final Function

```
function cleanNested(data) {
  const flat = data.flat(3); // make flat array
  const clean = flat.filter(function (item) {
    return item != null; // remove null & undefined
  });
  return clean;
}
```

Task 9 — Basic Dashboard Metrics (map + reduce + filter)

1. Problem Statement

We have list of orders. Each order has `amount` and `status` ("success" / "failed"). We want: - Total amount of successful orders - Total count of failed orders

2. Methods Used

- **filter()** → separate success / failed orders
- **reduce()** → sum amounts

3. Final Function

```
function dashboardMetrics(orders) {
  const successOrders = orders.filter(function (o) {
    return o.status === "success";
  });

  const failedOrders = orders.filter(function (o) {
    return o.status === "failed";
  });

  const totalSuccessAmount = successOrders.reduce(function (sum, o) {
    return sum + o.amount;
  }, 0);

  return {
```

```
    totalSuccessAmount: totalSuccessAmount,
    failedCount: failedOrders.length
  };
}
```

Summary

These tasks show **how array methods are used in real company logic**: - Grouping - Pagination - Filtering - Aggregation (totals) - Sorting - Cleaning data

You can extend this document by adding your own company-specific tasks using the same pattern:

Problem → What we want → Methods used → Step-by-step → Input/Output → Final code.