Perfect. Continuing **systematically**, **NotebookLM-ready**, with **NOTES and CODE strictly separated**.

📘 TOPIC: Array Methods (Functional Core of JS)

(`map`, `filter`, `reduce`, `forEach` • Internals • Edge Cases • Interview Traps)

🟦 NOTE 1: Why array methods matter

- Arrays are **objects with indexed storage**
- Modern JS favors **declarative iteration**
- Array methods:
  - reduce bugs
  - improve readability
  - enable functional programming
- Heavily used in **React, Node, data processing**

🟦 NOTE 2: Imperative vs Declarative style

- **Imperative** → how to do
- **Declarative** → what to do

Array methods are **declarative abstractions over loops**.

🟩 CODE 1: Imperative loop

```js
const nums = [1, 2, 3];
const result = [];

for (let i = 0; i < nums.length; i++) {
  result.push(nums[i] * 2);
}
```

🟩 CODE 2: Declarative `map`

```js
const nums = [1, 2, 3];
const result = nums.map(n => n * 2);
```

🟦 NOTE 3: `map()` — transformation

- Transforms each element
- Returns a **new array**
- Does NOT mutate original array
- Length of output = length of input

🟩 CODE 3: `map()` basics

```js
[1, 2, 3].map(x => x * 2);
// [2, 4, 6]
```

🟦 NOTE 4: `map()` internal behavior

Internally:

- Iterates left → right
- Calls callback for each index
- Skips empty slots (holes)
- Builds new array

🟩 CODE 4: map callback arguments

```js
["a", "b"].map((value, index, array) => {
  return `${index}:${value}`;
});
```

🟦 NOTE 5: `map()` trap — forgetting return

- Arrow functions with `{}` require `return`
- Missing return → `undefined`

🟩 CODE 5: Common mistake

```js
[1, 2, 3].map(x => {
  x * 2;
});
// [undefined, undefined, undefined]
```

🟦 NOTE 6: `forEach()` — side effects

- Executes callback for each element
- Returns **undefined**
- Used for:
  - logging
  - mutation
  - side effects

🟩 CODE 6: forEach example

```js
[1, 2, 3].forEach(x => {
  console.log(x);
});
```

🟦 NOTE 7: `map()` vs `forEach()` (INTERVIEW)

Aspect	map	forEach
Returns	New array	undefined
Purpose	Transform	Side effects
Chainable	Yes	No

🟦 NOTE 8: `filter()` — selection

- Selects elements based on condition
- Returns new array
- Length ≤ original length

🟩 CODE 7: filter basics

```js
[1, 2, 3, 4].filter(x => x % 2 === 0);
// [2, 4]
```

🟦 NOTE 9: filter truthy rule

- Callback must return truthy/falsy
- Any truthy value keeps element

🟩 CODE 8: filter coercion

```js
["", "JS", null, "Code"].filter(Boolean);
// ["JS", "Code"]
```

🟦 NOTE 10: `reduce()` — the most powerful one

- Reduces array to **single value**
- Can return:
  - number
  - object

- array
- anything

🟩 CODE 9: reduce basic example

```js
[1, 2, 3, 4].reduce((acc, curr) => acc + curr, 0);
// 10
```

🟦 NOTE 11: reduce parameters (IMPORTANT)

Callback receives:

1. accumulator
2. currentValue
3. index
4. array

Initial value is **critical**.

🟩 CODE 10: reduce parameters

```js
[10, 20].reduce((acc, curr, index, arr) => {
  return acc + curr;
}, 0);
```

🟦 NOTE 12: reduce without initial value (TRAP)

- First element becomes accumulator
- Loop starts from index 1
- Dangerous for empty arrays

🟩 CODE 11: reduce trap

```js
[].reduce((a, b) => a + b);
// ❌ TypeError
```

🟦 NOTE 13: reduce real-world patterns

🟩 CODE 12: Array → Object

```js
```

```
const users = ["A", "B", "C"];

const obj = users.reduce((acc, name) => {
  acc[name] = true;
  return acc;
}, {});
```

🟦 NOTE 14: Chaining methods (functional pipeline)

- Methods return new arrays
- Enables clean pipelines

🟩 CODE 13: Chaining example

```
js

[1, 2, 3, 4, 5]
  .filter(x => x % 2 === 0)
  .map(x => x * 10)
  .
```