Immutability and immutable data structures

🧊 Immutability and Immutable Data Structures — Detailed Explanation

🔷 What is Immutability?

Immutability means that once a **data structure is created, it cannot be changed**. Any update to it will result in the creation of a **new copy**, leaving the original intact.

This principle is a cornerstone of **functional programming** and helps build **predictable**, **debuggable**, and **side-effect-free** code.

✅ Why Immutability Matters

Benefit	Description
🧠 Predictable State	Values don't change unexpectedly.
🐛 Easier Debugging	No surprises from hidden mutations.
♻️ Undo/Redo Support	Old versions are preserved.
⚛️ Efficient Reactivity	Helps React and similar libraries optimize re-renders using shallow checks.
🔄 Concurrency Safety	Immutable data is thread-safe (no race conditions).

🔶 Mutable vs Immutable Example

```
// ❌ Mutable
let person = { name: "Abhi" };
person.name = "John"; // Modified in place
```

```
// ✅ Immutable
const person = { name: "Abhi" };
const updatedPerson = { ...person, name: "John" }; // New object
```

🔶 Immutability in Arrays

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

// ❌ Mutates the original array
arr.push(4);

// ✅ Immutable operation
const newArr = [...arr, 4]; // [1, 2, 3, 4]

// ✅ Filtered copy
const filtered = arr.filter(n ⇒ n !== 2); // [1, 3]
```

🔶 Key Tools in JavaScript

1. Spread Operator (`...`)

```
const original = { a: 1 };
const copy = { ...original, b: 2 }; // Does not affect original
```

2. `Array.prototype.map` , `filter` , `reduce`

Immutable alternatives to loops and mutations:

```
const arr = [1, 2, 3];
const doubled = arr.map(x ⇒ x * 2); // [2, 4, 6]
```

3. `Object.assign()`

```
const obj = { a: 1 };
const newObj = Object.assign({}, obj, { b: 2 });
```

🔶 Libraries for Immutability

Library	Description
Immutable.js	Provides persistent immutable collections
Immer	Allows "mutating" syntax while keeping state immutable under the hood
Mori	Functional data structures inspired by ClojureScript

🔷 Example with Immer

```
import produce from "immer";

const baseState = [{ name: "Abhi" }];

const nextState = produce(baseState, draft ⇒ {
  draft.push({ name: "John" }); // Behind the scenes, it's immutable
});
```

🧠 Deep vs Shallow Copy

Type	Description
Shallow	Copies top-level references only
Deep	Recursively copies all nested objects

Example of shallow copy problem:

```
const original = { a: { b: 2 } };
const shallowCopy = { ...original };
shallowCopy.a.b = 10; // Affects original.a.b!
```

Use `structuredClone()` or libraries like `lodash.cloneDeep()` for deep copying.

⚠️ Common Mistakes

- Using `push`, `splice`, or direct assignment on objects/arrays.
- Forgetting that object/array variables hold **references**, not values.

🧾 Summary

Concept	Summary
Immutability	Data doesn't change — new copies are made
Benefits	Predictability, performance, undo/redo, concurrency
Tools	Spread operator, `Object.assign`, map/filter/reduce
Libraries	`Immutable.js`, `Immer`, `Mori`
Caution	Watch for reference copying (use deep cloning when needed)

Would you like a cheat sheet or code snippets that help implement immutability in real projects (like Redux, React, etc.)?