Pure functions and side effects

Sure, let's explore **Pure Functions** and **Side Effects** in detail — two fundamental concepts in JavaScript (and functional programming in general).

✅ What is a Pure Function?

A **pure function** is a function that meets **two main conditions**:

1. **Given the same input, it always returns the same output.**
2. **It does not cause any observable side effects** (i.e., it doesn't modify anything outside itself).

🔁 Example of a Pure Function:

```
function add(a, b) {
  return a + b;
}
```

- Same input → same output (`add(2, 3)` always returns `5`)
- No modification of external state → ✅ pure

❌ Impure Function Example:

```
let counter = 0;

function increment() {
  counter++;
  return counter;
}
```

- Output depends on external variable `counter` → ❌ not pure

- It **modifies external state** → ❌ side effect

🔍 Side Effects — What Are They?

A **side effect** is **any interaction with the outside world** from within a function, such as:

- Modifying a global variable
- Changing the DOM
- Making an HTTP request
- Logging to the console
- Writing to disk or localStorage

💥 Example of Side Effects:

```
let user = { name: "Abhi" };

function updateName(newName) {
  user.name = newName;
}
```

- The function modifies the `user` object outside its scope → ⚠️ Side effect

🧼 Why Use Pure Functions?

✅ Benefits of Pure Functions:

- **Predictable:** Easier to test and debug.
- **Reusable:** No reliance on external state.
- **Composability:** Easy to combine with other functions.
- **Referential transparency:** Can be replaced with their output.

🧠 Impure vs Pure Function: Comparison

Feature	Pure Function	Impure Function
Output consistency	Always same output for same input	May vary depending on external state
Side effects	❌ No side effects	✅ Can have side effects
Testing	Easy	Hard (due to external dependencies)
Example	`add(2, 3)` → `5`	`fetchData()` → depends on network

✨ How to Avoid Side Effects

- **Avoid global variables**
- **Use function arguments instead of external state**
- **Return new objects/arrays instead of mutating existing ones**

❌ Mutating Array (impure):

```
const arr = [1, 2, 3];
arr.push(4); // modifies the original array
```

✅ Pure Alternative:

```
const arr = [1, 2, 3];
const newArr = [...arr, 4]; // creates a new array
```

🎯 Real-Life Use Case: Redux Reducers

In Redux (a state management library), **reducers** must be **pure functions**.

```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case "INCREMENT":
      return state + 1;
    default:
```

```
    return state;
  }
}
```

- No mutation
- Always returns new state based on input

📝 Summary

Concept	Description
Pure Function	Returns same output for same input, no side effects
Side Effect	Any change outside the function (e.g., DOM, file, variable)
Why It Matters	Pure functions improve testability, reliability, and maintainability

Let me know if you'd like to move on to **Recursion and Tail Call Optimization** next!