Understanding Pure and Impure Functions & React's `useEffect` Hook

---

1. Pure Functions

What is a Pure Function?

A **pure function** is a function that behaves like a **mathematical formula**.

It has **two strict rules**:

1. **Same input → Same output (Always)**
2. **No side effects**

This means:

- It does **not** change anything outside itself.
- It does **not** depend on anything outside its inputs.

Think of a pure function as a **closed box**:

- You put something in.
- You get something out.
- The outside world is untouched.

Characteristics of Pure Functions

1. **Deterministic Output**
   If you call the function multiple times with the same arguments, the result will never change.
2. **No External Dependency**
   It does not read or modify:
   - Global variables
   - Browser APIs
   - Databases
   - Files
   - Network data
   - Time or random values
3. **No Side Effects**
   A side effect is anything the function does besides returning a value.

   Examples of side effects:

- Modifying a variable outside the function
- Updating UI
- Logging to console
- Making API calls
- Writing to files

Why Pure Functions Are Important

1. **Easy to Test**
   You don't need setup or mocks.
   Input → Output is enough.
2. **Easy to Debug**
   If output is wrong, input is wrong.
   No hidden behavior.
3. **Predictable Code**
   The function never surprises you.
4. **Reusable and Composable**
   Pure functions can be safely combined to build complex logic.

Example 1: Pure Function

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

Why this is pure:

- Depends only on `a` and `b`
- Same inputs always give same output
- Does not modify anything outside
- No side effects

Calling:

```
add(2, 3) // always 5
add(2, 3) // always 5
```

---

2. Impure Functions

What is an Impure Function?

An **impure function** is a function that **depends on or affects the outside world**.

Even if it returns a value, it may:

- Modify external state

- Depend on external data
- Produce different outputs for the same input

Characteristics of Impure Functions

1. **Unpredictable Output**
   Output may change even if inputs remain the same.
2. **Side Effects Present**
   It may:
   - Change global variables
   - Call APIs
   - Read system time
   - Modify DOM
   - Log data
3. **Harder to Test**
   Requires environment setup or mocking.

Why Impure Functions Are Sometimes Necessary

Real applications **must interact with the real world**:

- Fetch data
- Save data
- Update UI
- Track user behavior

So impure functions are **necessary**, but they should be **controlled and isolated**.

Example 2: Impure Function

```
let total = 0;

function addToTotal(value) {
  total += value;
  return total;
}
```

Why this is impure:

- Depends on external variable `total`
- Modifies external state
- Same input can produce different output

Calling:

```
addToTotal(5) // 5
addToTotal(5) // 10
addToTotal(5) // 15
```

Same input, different output → impure.

---

3. Why React Cares About Purity

React **expects components to be pure** while rendering.

This means:

- Rendering should only calculate JSX
- No data fetching
- No DOM manipulation
- No side effects

Why?
Because React may:

- Render multiple times
- Pause rendering
- Re-run renders for optimization

Side effects during render break React's guarantees.

---

4. Introducing `useEffect` Hook

Why `useEffect` Exists

React separates:

- **Rendering (pure)**
- **Side effects (impure)**

`useEffect` is React's **official place for impure operations**.

It allows you to:

- Perform side effects **after rendering**
- Control **when** those side effects run
- Clean up when component unmounts

What is a Side Effect in React?

In React, side effects include:

- Fetching data from API

- Subscribing to events
- Updating document title
- Manipulating DOM
- Timers
- Logging

All of these belong inside `useEffect`.

---

5. How `useEffect` Works Conceptually

```
useEffect(() => {
  // side effect logic
}, [dependencies]);
```

React guarantees:

1. Component renders first (pure phase)
2. Effect runs after render (impure phase)

This separation keeps React predictable and safe.

---

6. Lifecycle Management Using `useEffect`

`useEffect` replaces old class lifecycle methods:

Class Component	useEffect Equivalent
componentDidMount	useEffect with empty dependency array
componentDidUpdate	useEffect with dependencies
componentWillUnmount	Cleanup function

---

7. Dependency Array (Very Important)

The dependency array controls **when the effect runs**.

Case 1: No dependency array

```
useEffect(() => {
  // runs after every render
});
```

Runs:

- On first render
- On every update

Case 2: Empty dependency array

```
useEffect(() => {
  // runs only once
```

```
}, []);
```

Runs:

- Only after first render (component mount)

Used for:

- Initial API calls
- One-time setup

Case 3: With dependencies

```
useEffect(() => {
  // runs when dependencies change
}, [count]);
```

Runs:

- On first render
- Whenever `count` changes

This makes effects **conditional and optimized**.

---

8. Cleanup Function in `useEffect`

Why Cleanup Is Needed

Some side effects:

- Keep running
- Consume memory
- Create subscriptions

If not cleaned:

- Memory leaks occur
- Duplicate listeners accumulate
- Performance degrades

Cleanup Mechanism

`useEffect` can return a function:

```
useEffect(() => {
  // setup logic

  return () => {
    // cleanup logic
  };
}, []);
```

Cleanup runs:

- Before component unmounts
- Before re-running the effect

---

9. Core Philosophy Summary

- **Pure functions** = predictable logic
- **Impure functions** = real-world interactions
- **React render** = pure
- **useEffect** = controlled impurity
- **Dependency array** = execution control
- **Cleanup** = safety and performance