useState: Memory for Components

Why plain variables forget: the counter, useState's two gifts, updating from the old value, state in form inputs, and why the screen re-renders.

Introduction

A counter button that never moves is useless — the screen must **remember** a value and show its new version when it changes. A plain variable cannot do this job in React: the component re-runs, the variable resets, and React never learns the value changed. `useState` is React's memory: it stores a value across re-renders and tells React "this changed — draw the screen again." *(Roman Urdu: useState component ki yaad-dasht hai — value bhi rakhta hai, aur badle to screen ko dobara banata hai.)*

Key points

- `useState` **gives you two things:** `const [count, setCount] = useState(0);` — the current value (`count`) and the **setter function** (`setCount`) that is the only proper way to change it. The `0` is the starting value.
- **The naming pattern is a convention:** value + `set` + Value — `name/setName`, `isOpen/setIsOpen`. The destructuring is array destructuring from the JavaScript series.
- **Changing state re-renders:** when you call the setter, React runs the component again with the new value and updates the screen. That re-run is called a **render**.
- **Never change state directly:** `count = count + 1` does nothing visible — React is not watching plain assignments. Always go through the setter: `setCount(count + 1)`.

  (Roman Urdu: State ko haath se mat badlo — hamesha setter se badlo, warna React ko pata hi nahi chalega.)
- **Updates from the old value use the updater function:** `setCount(prev => prev + 1)`. When the new value depends on the old one, this form is the safe one — React guarantees `prev` is the freshest value, even when several updates queue up in one click.
- **State is private per component copy:** three `<Counter />` components have three independent counts. Each use remembers its own.
- **Typing in an input is state too:** an input whose value lives in state re-renders on every keystroke — the controlled input, the heart of React forms (lesson 5.6).

Code examples

Example 1 — The counter, live

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Add 1</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

What it does: `useState(0)` gives the counter its memory, starting at 0. Clicking "Add 1" calls the setter with the new value; React stores it and re-renders, so `{count}` shows the fresh number. "Reset" sets it straight back to 0.

What you will see: the number climbs 0 → 1 → 2 → 3 with each click, and drops to 0 on Reset — the plain-variable version of this button would stay at 0 forever.

Example 2 — The updater function: why prev matters

```
function TripleAdd() {
  const [count, setCount] = useState(0);

  function addThree() {
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
  }

  return <button onClick={addThree}>Count is {count} — add 3</button>;
}
```

What it does: One click queues three updates. Written as `setCount(count + 1)` three times, all three would read the same stale `count` and the total would rise by only 1. The updater form receives the freshest value each time, so the three updates stack correctly.

What you will see: each click adds exactly 3 — 0 → 3 → 6 → 9.

Example 3 — State in an input: the screen mirrors your typing

```
function NameForm() {
  const [name, setName] = useState("");

  return (
    <div>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Type your name"
      />
      <p>Hello, {name === "" ? "stranger" : name}!</p>
      <p>Characters: {name.length}</p>
    </div>
  );
}
```

What it does: The input's `value` comes from state, and every keystroke fires `onChange`, which saves the new text into state — which re-renders the input and the greeting. State and screen can never disagree, because the screen *is* the state.

What you will see: as you type "Ayesha", the greeting becomes "Hello, Ayesha!" letter by letter, and the character count climbs 1, 2, 3…

Quick recap

- **`useState`** is a component's memory: it returns the value and a setter; only the setter may change it, and changing it **re-renders** the screen.
- When the new value depends on the old, use the updater form: `setCount(prev => prev + 1)`.
- Each component copy owns its state, and inputs keep their text in state — the controlled-input pattern behind all React forms.

Common mistakes

1. **Mutating state directly** — `count++` or `items.push(x)` changes the value behind React's back, and the screen does not update. The fix: always call the setter with a new value — for arrays and objects, a new copy (spread, from the JavaScript series).
2. **Reading state right after setting it** — `setCount(5); console.log(count);` prints the OLD value, because the update applies on the next render, not on the same line. The fix: use the value you just computed, or the updater function — never expect the variable to change mid-function.
3. **Calling useState inside conditions or loops** — hooks must run in the same order on every render, so `if (x) { useState(...) }` corrupts React's memory. The fix: hooks live at the top level of the component, always (the full rules return in lesson 5.9).

Practice

1. Build a `QuantityPicker` component: state starts at 1, a "−" button lowers it (never below 1), a "+" button raises it (never above 10), and the total price `Rs {qty * 250}` shows beside it. Use the updater form for both buttons.
2. Build a `ShowHide` component: a password input and a button that toggles between "Show" and "Hide". Keep the password text and a boolean `visible` in two separate `useState` calls, and switch the input's `type` between "text" and "password".

Practice quiz

1. What does const [count, setCount] = useState(0); give you?

- Only the value 0
- The current value and the setter function, the only proper way to change it; 0 is the starting value
- Two values that change together automatically
- A function and its result

Answer: The current value and the setter function, the only proper way to change it; 0 is the starting value

2. What happens when you call the setter, and what happens if you change state directly?

- The setter does nothing, direct change re-renders
- Calling the setter re-renders the component with the new value; count = count + 1 does nothing visible, React is not watching plain assignments
- Both re-render the screen
- Both crash

Answer: Calling the setter re-renders the component with the new value; count = count + 1 does nothing visible, React is not watching plain assignments

3. Do three <Counter /> components on one page share one count?

- Yes, all counters share state
- No, state is private per component copy, three counters have three independent counts
- Yes, but only the first two
- They share only the starting value, then merge

Answer: No, state is private per component copy, three counters have three independent counts

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-usestate · More free tools and notes at https://scopetreasure.com