State

What state is, how to define it with useState, how to update it, updating based on the previous state, and state immutability -- with simple examples.

Beginner 12 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/state

State

In Events, we learned how to catch a user's actions, but our handlers only ever logged to `console.log` -- nothing changed on screen. This lesson covers how to actually change the screen in React: **state**.

What Is State?

State is data a component "remembers" that can change over time. When state changes, React automatically re-renders that component -- it redraws the screen to match the latest state. A counter, an input's current value, whether a menu is open or closed -- all of these are examples of state.

Defining State with useState

You define state with the `useState` function:

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

function UseStateBasicExample() {
  // useState(0) sets up state with an initial value of 0.
  // count: to read the current value. setCount: to update the value.
  const [count, setCount] = useState(0);

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

`useState(0)` sets up state with an initial value of `0` and returns two things: the current value (`count`) and a function to update it (`setCount`). The `const [count, setCount] = ...` syntax is a JavaScript feature (array destructuring) that splits these two values into separate variables on one line -- similar to the object destructuring we saw in Props, but for arrays.

Updating State

You **can't** update state by writing `count = count + 1` directly -- you have to call the function `useState` gave you (`setCount`):

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

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

  function increment() {
    setCount(count + 1);
  }

  function reset() {
    setCount(0);
  }

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

When `setCount(...)` is called, React does two things: it saves the new value of the state, and it re-renders the component with that new value. Directly assigning to a variable doesn't tell React "something changed," so it doesn't update the screen -- we'll see this in State vs. a Regular Variable.

Updating Based on the Previous State

If calculating the new state needs the previous state, you should pass `setCount` a FUNCTION instead of a direct value:

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

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

  function incrementTwice() {
    // WRONG: count still shows the OLD value here, and both lines use
    // the same "old value + 1" -- so the result only increases by 1,
    // not 2.
    // setCount(count + 1);
    // setCount(count + 1);

    // RIGHT: the function form (prevCount => ...) always uses the most
    // up-to-date value from the previous update.
    setCount((prevCount) => prevCount + 1);
    setCount((prevCount) => prevCount + 1);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementTwice}>Increment by 2</button>
    </div>
  );
}
```

Writing `setCount(count + 1)` isn't reliable if it's called more than once within the same render -- the `count` variable stays fixed for that entire render. Writing `setCount((prevCount) => prevCount + 1)` tells React "whatever happens, calculate based on the MOST RECENT value."

State vs. a Regular Variable

Why not just use a regular variable with `let`? Because changing a regular variable doesn't update the screen:

```
function StateVsVariableExample() {
  let plainCount = 0; // A plain variable -- NOT state.

  function handleClick() {
    plainCount = plainCount + 1;
    console.log("plainCount is now:", plainCount);
    // The value actually changed, but nothing updates on screen --
    // because React has no idea a plain variable changed.
  }

  return (
    <div>
      <p>The value on screen always stays the same: {plainCount}</p>
      <button onClick={handleClick}>Increment (Screen Won't Change)</button>
    </div>
  );
}
```

`plainCount` really does change (you can see it in the console), but since React doesn't know about it, it doesn't re-render the component -- the screen keeps showing the same value. `useState` is special because it tells React "let me know when this value changes."

State Immutability

If state is an object or array, instead of directly changing (mutating) it, you should always create a NEW object/array and pass that to the setter function:

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

function StateImmutabilityExample() {
  const [user, setUser] = useState({ name: "Emma", age: 25 });

  function haveBirthday() {
    // WRONG: mutating the existing object directly.
    // React might not notice the change, and the screen might not update.
    // user.age = user.age + 1;
    // setUser(user);

    // RIGHT: create a NEW object that copies the old values via spread
    // (...) and pass that to the state.
    setUser({ ...user, age: user.age + 1 });
  }

  return (
    <div>
      <p>
        {user.name} is {user.age} years old.
      </p>
      <button onClick={haveBirthday}>Birthday</button>
    </div>
  );
}
```

Writing `user.age = user.age + 1` and passing back the same object doesn't guarantee React notices the change -- React decides whether to re-render partly by checking whether the state is "the same object or a different one." The spread operator (`{ ...user, age: ... }`) copies all the fields of the old object into a new one, changing only the field you specify.

Summary and Glossary

State is a component's data that can change over time and, when it changes, automatically updates the screen. It's defined with `useState` and can only be updated through its setter function -- direct assignment doesn't work. When updating based on the previous state, use the function form. Object/array state is never mutated directly -- a new copy is always created.

Glossary

State — A component's data that can change over time and, when it changes, triggers the component to re-render.

`useState` — The React function that lets you add state to a component; it returns the current value and a function to update it.

Re-render — React re-running a component and updating the screen when its state or props change.

Immutability — The principle of creating a new copy that includes the change, instead of directly modifying an existing object/array.