ScopeTreasure Courses

REACT COURSES · LESSON 5.13 (SHORT) · CODE SCOPE SERIES

Hook Tricks in 60 Seconds

Five fast hook tricks from the Short: functional updates, lazy initial state, object state in one call, a previous-value ref, and one effect with a cleanup.

Introduction

The 5.13 Short raced through **five hook tricks** in about a minute. These notes slow the same five down so you can read, copy and actually use them: each trick is a small pattern that removes a whole class of small bugs from daily React code. *(Roman Urdu: Short mein paanch tricks tezi se guzri theen — yahan har trick aram se, code ke saath.)*

Key points

- **Trick 1 — Functional updates:** when the new state depends on the old, pass a function: `setCount(c => c + 1)`. React feeds it the freshest value, so rapid or batched updates never read stale state.
- **Trick 2 — Lazy initial state:** if the starting value is expensive to compute (reading localStorage, building a big list), pass a function: `useState(() => computeStart())`. It runs **once**, on the first render, not on every render.
- **Trick 3 — Object state, one call:** related values live in one state object, and one setter updates a single field with spread: `setForm(f => ({ ...f, email: v }))` — the functional form keeps it stale-proof.
  (Roman Urdu: Mutaliq values aik hi object mein — aur setter hamesha purani value se nayee copy banaye.)
- **Trick 4 — Previous value with a ref:** store the last render's value in a `useRef`, update it in an effect, and the current render can show "before vs now" (you built the full version in lesson 5.8).
- **Trick 5 — One effect, with cleanup:** an effect that starts something (interval, listener, subscription) returns the function that stops it. One tidy effect beats three scattered ones — and the cleanup is part of the trick, not an extra.
- **The thread through all five:** hooks reward thinking in *updates and lifecycles*, not in assignments — every trick is that idea wearing a different costume.

Code examples

Example 1 — Tricks 1 + 2: safe updates, lazy start

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

function buildStartList() {
  console.log("Building the start list…");
  return ["Chai", "Coffee"];
}

function DrinkCounter() {
  // Lazy: buildStartList runs ONCE, however often we re-render
  const [drinks, setDrinks] = useState(() => buildStartList());

  function addDrink() {
    // Functional: always extends the freshest list
    setDrinks((list) => [...list, "Juice"]);
  }

  return <button onClick={addDrink}>Drinks: {drinks.join(", ")}</button>;
}
```

What it does: The lazy initialiser runs a single time — the console line proves it — while the functional update guarantees every click appends to the list as it is *now*, even if clicks land in the same instant.

What you will see: "Building the start list…" logs exactly once; the button label grows "Chai, Coffee" → "Chai, Coffee, Juice" → "…Juice, Juice" with no lost entries.

Example 2 — Trick 3: object state in one call

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

function QuickForm() {
  const [form, setForm] = useState({ name: "", city: "" });

  return (
    <div>
      <input
        value={form.name}
        onChange={(e) => setForm((f) => ({ ...f, name: e.target.value }))}
        placeholder="Name"
      />
      <input
        value={form.city}
        onChange={(e) => setForm((f) => ({ ...f, city: e.target.value }))}
        placeholder="City"
      />
      <p>{form.name} — {form.city}</p>
    </div>
  );
}
```

What it does: One state, one pattern per field: spread the freshest form, overwrite one key. It is the lesson 5.6 form style with the updater form added, which keeps fast typing from ever overwriting a field with a stale copy.

What you will see: both fields update independently and the summary line keeps up with every keystroke.

Example 3 — Tricks 4 + 5: previous value, and one clean effect

```
import { useEffect, useRef, useState } from "react";

function ScoreBoard() {
  const [score, setScore] = useState(0);
  const prevScore = useRef(0);

  useEffect(() => {
    prevScore.current = score;
  }, [score]);

  useEffect(() => {
    const id = setInterval(() => console.log("Score now:", score), 5000);
    return () => clearInterval(id);
  }, [score]);

  return (
    <div>
      <p>Score: {score} (was {prevScore.current})</p>
      <button onClick={() => setScore((s) => s + 10)}>+10</button>
    </div>
  );
}
```

What it does: The first effect keeps the ref one step behind, so the screen can show the previous score. The second effect owns one interval and its cleanup — when `score` changes, React cleans the old interval up before starting the new one, so exactly one logger ever runs.

What you will see: "Score: 0 (was 0)" → click → "Score: 10 (was 0)" → click → "Score: 20 (was 10)", with a single console log every five seconds, never a pile of them.

Quick recap

- **Functional updates** (`setX(prev => …)`) are never stale; **lazy initialisers** (`useState(() => …)`) pay the setup cost once.
- **Object state** is updated by spreading the freshest value; a **ref keeps the previous render's value** one step behind.
- Every effect that starts something **returns its cleanup** — that pairing is the trick, not a chore.

Common mistakes

1. **Eager initial state** — `useState(buildStartList())` rebuilds the list on EVERY render and throws it away. The fix: pass the function itself — `useState(() => buildStartList())` — and React calls it once.

2. **Updating objects from stale state** — `setForm({ ...form, name: v })` inside a fast sequence can start from an old `form`. The fix: the functional spread — `setForm(f => ({ ...f, name: v }))`.
3. **An effect without its cleanup** — intervals and listeners multiply silently across renders and mounts. The fix: if the effect starts it, the returned function stops it — always the pair.

Practice

1. Rewrite the counter from lesson 5.4 three ways: direct update, functional update, and a version whose start value comes from a lazy initialiser reading `localStorage` (guard the read so a missing key starts at 0). Note which lines actually changed.
2. Build a `usePrevious`-style demo of your own: a colour box whose background is state, with the previous colour shown as a small swatch beside it, labelled "Before". Use only a ref and one effect — no extra state.

Practice quiz

1. Why use the functional update setCount(c => c + 1)?

- It is shorter to type
- React feeds it the freshest value, so rapid or batched updates never read stale state
- It runs faster in all cases
- It is required by the linter

Answer: React feeds it the freshest value, so rapid or batched updates never read stale state

2. What does lazy initial state useState(() => computeStart()) buy you?

- Nothing, it runs every render anyway
- The expensive starting value is computed once, on the first render, not on every render
- It makes the state lazy in a bad way
- It delays the first render forever

Answer: The expensive starting value is computed once, on the first render, not on every render

3. What must every effect that starts something do?

- Nothing special
- Return the function that stops it, its cleanup: if the effect starts it, the returned function stops it, always the pair
- Start two more effects
- Run only once

Answer: Return the function that stops it, its cleanup: if the effect starts it, the returned function stops it, always the pair

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-hook-tricks · More free tools and notes at https://scopetreasure.com