ScopeTreasure Courses

REACT COURSES · LESSON 5.8 · CODE SCOPE SERIES

useRef: Reaching the Real DOM

A quiet box and a remote control: ref.current, focusing an input on click, counting renders without causing them, and ref vs state.

Introduction

Most of the time React should own the DOM — you change state, React redraws. But sometimes you must speak to one real element directly: focus this input, scroll to that section, measure this box. And sometimes you need a value that survives renders **without causing them**. Both jobs belong to `useRef`. *(Roman Urdu: useRef aik khamosh dabba hai — value rakhta hai, lekin screen ko dobara nahi banata.)*

Key points

- `useRef` **returns one object:** `const inputRef = useRef(null);` — an object with a single property, `.current`. That object is the same on every render; React never resets it.
- **Attach it with the** `ref` **prop:** `<input ref={inputRef} />` — after the render, `inputRef.current` *is* the real DOM input element, with all its native powers: `.focus()`, `.select()`, `.scrollIntoView()`.
- **Changing** `.current` **does NOT re-render:** refs are the quiet storage — counters, timers' ids, previous values. The screen does not move when a ref changes; if the screen must move, that value belongs in state.

  (Roman Urdu: Ref badalne se render nahi hota — screen badalni ho to woh kaam state ka hai.)
- **The ref vs state rule:** ask one question — "must the screen show this?" Yes → state. No (it is a handle, an id, a behind-the-scenes number) → ref.
- **Refs start as** `null`**:** before the first render finishes, `.current` is still `null` — reading it during render is a classic crash. Refs are read inside event handlers and effects, where the element exists.
- **A famous use: the previous value.** Keep the last render's value in a ref (updated inside an effect) and the current render can compare "now vs before" — Example 3.
- **Refs are not a back door for everything:** changing DOM styles or text directly through refs fights React's rendering and creates mystery bugs — focus, scroll and measurement are the honest jobs.

Code examples

Example 1 — Focus the input with one click

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

function SearchBox() {
  const inputRef = useRef(null);

  function focusInput() {
    inputRef.current.focus();
  }

  return (
    <div>
      <input ref={inputRef} placeholder="Search courses" />
      <button onClick={focusInput}>Focus the search</button>
    </div>
  );
}
```

What it does: The ref is declared, attached to the input with the `ref` prop, and after render `inputRef.current` is the real input element. The button calls its native `.focus()` — the one job state cannot do, because focusing is an action on the element, not data.

What you will see: click the button and the cursor jumps into the search box, ready to type — no state changed, no re-render happened.

Example 2 — Counting renders without causing them

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

function RenderCounter() {
  const [text, setText] = useState("");
  const renders = useRef(0);
  renders.current = renders.current + 1;

  return (
    <div>
      <input value={text} onChange={(e) => setText(e.target.value)} placeholder="Type
something" />
      <p>This component has rendered {renders.current} times.</p>
    </div>
  );
}
```

What it does: Every render runs the component body again, so the ref climbs by one each time — and because it is a ref, the increment itself triggers **no** new render (a state counter here would loop forever). The paragraph reads the quiet box on the same render the typing caused.

What you will see: the count starts at 1 and rises with every keystroke — proof of how often React re-renders, stored without disturbing the screen.

Example 3 — Remembering the previous value

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

function PriceWatcher() {
  const [price, setPrice] = useState(250);
  const prevPrice = useRef(price);

  useEffect(() => {
    prevPrice.current = price;
  }, [price]);

  return (
    <div>
      <p>Now: Rs {price} · Before: Rs {prevPrice.current}</p>
      <button onClick={() => setPrice(price + 50)}>Raise price</button>
      <button onClick={() => setPrice(price - 50)}>Lower price</button>
    </div>
  );
}
```

What it does: The ref holds the price from the *last* render. The effect updates it only after the current render is done, so during this render the ref still shows the previous value — the standard pattern for comparing now vs before, used by real apps for "price went up" indicators.

What you will see: "Now: Rs 250 · Before: Rs 250"; click Raise and it reads "Now: Rs 300 · Before: Rs 250" — the ref always trails exactly one step behind.

Quick recap

- `useRef` gives one stable object whose `.current` survives renders and never triggers them.
- Attached via the `ref` prop, it reaches real DOM elements for the jobs state cannot do: **focus, scroll, measure**.
- The rule: if the screen must show it, use state; if it works behind the scenes, use a ref.

Common mistakes

1. **Using a ref where state was needed** — updating `.current` and wondering why the screen is frozen: refs never re-render by design. The fix: values the UI displays belong in `useState`, always.
2. **Reading `.current` during render for DOM work** — on the first render the element does not exist yet and `.current` is `null`. The fix: touch DOM elements inside event handlers or effects, never in the render body.
3. **Replacing state with refs to "avoid re-renders"** — direct DOM edits through refs desync the screen from React's data, and the next honest render wipes them. The fix: refs for handles and silent numbers only; let React own everything visible.

Practice

1. Build a `NameForm`: an input with a ref and two buttons — "Focus" (puts the cursor in the input) and "Clear & focus" (empties the input by setting its value through state, then focuses it). Notice which parts needed state and which only the ref.
2. Build a render-safe stopwatch display: a `seconds` state ticking up every second via `setInterval` inside an effect, with the interval's id stored in a ref so a "Stop" button can clear it. Explain in one comment why the id lives in a ref, not state.

Practice quiz

1. What does useRef(null) return, and what happens when you change .current?

- A state value, and changing it re-renders
- One stable object with a .current property; changing .current does NOT re-render, it is quiet storage
- A DOM element directly
- Nothing, refs are empty

Answer: One stable object with a .current property; changing .current does NOT re-render, it is quiet storage

2. How do you attach a ref to a real DOM input, and what can you then do?

- With the value prop
- With the ref prop, <input ref={inputRef} />, after which inputRef.current is the real input, with .focus(), .select() and .scrollIntoView()
- Refs cannot reach real DOM elements
- With onClick

Answer: With the ref prop, <input ref={inputRef} />, after which inputRef.current is the real input, with .focus(), .select() and .scrollIntoView()

3. Where may you safely read a ref's DOM element?

- During render, always

- Inside event handlers and effects, where the element exists; before the first render finishes .current is still null
- Only in the return statement
- Never, refs are write-only

Answer: Inside event handlers and effects, where the element exists; before the first render finishes .current is still null

---

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