ScopeTreasure Courses

REACT COURSES · LESSON 5.17 · CODE SCOPE SERIES

useMemo & useCallback

Stop wasting work: memoizing an expensive calculation, keeping a function's identity stable, React.memo, and the honest rule — measure first, memoize second.

Introduction

Every render reruns your component's body — including any expensive calculation inside it, and it rebuilds every function from scratch. `useMemo` lets React *remember a computed value* between renders, and `useCallback` lets it *remember a function* — both recomputed only when their dependencies change. *(Roman Urdu: useMemo mehngi calculation ka jawab yaad rakhta hai, useCallback function ko — taake bekaar kaam dobara na ho.)*

Key points

- `useMemo(fn, deps)` **caches a value:** React runs `fn` on the first render, then returns the stored answer until a dependency changes. The classic job: filtering/sorting a big list that should not be redone when an unrelated state changes.
- `useCallback(fn, deps)` **caches the function itself:** the component's functions are recreated every render; wrapping one keeps its **identity** stable — which matters exactly when the function is passed to a memoized child or used in an effect's dependencies.
- `React.memo` **is the third partner:** wrapping a component (`const Row = memo(function Row(props) {...})`) makes React skip re-rendering it when its props are unchanged — and it only pays off when those props (including callbacks) are themselves stable, which is what `useCallback` provides.

  (Roman Urdu: Teeno saathi hain — memo bacche ko bachata hai, useCallback us ki props ko stable rakhta hai, useMemo mehnga jawab yaad rakhta hai.)
- **Dependencies are the contract:** list everything the computation reads. A stale dep means a stale cached answer — the same honesty rule as useEffect.
- **Memoization is not free:** it costs memory and comparison work. For cheap calculations and small lists, the cache can cost more than the work — which is why the pros' rule is **measure first, memoize second** (lesson 5.23 shows the measuring).
- **What it is NOT for:** never wrap everything "just in case". The honest triggers are a profiler showing real repeated cost, or a child re-rendering despite unchanged props.

Code examples

Example 1 — useMemo: an expensive filter, computed only when it must be

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

const students = Array.from({ length: 5000 }, (_, i) => ({
  id: i,
  name: `Student ${i}`,
  marks: (i * 37) % 100,
}));

function Toppers() {
  const [minMarks, setMinMarks] = useState(80);
  const [theme, setTheme] = useState("light");

  const toppers = useMemo(() => {
    console.log("Filtering 5000 students…");
    return students.filter((s) => s.marks >= minMarks);
  }, [minMarks]);

  return (
    <div className={theme}>
      <p>Toppers (≥ {minMarks}): {toppers.length}</p>
      <button onClick={() => setMinMarks(minMarks + 5)}>Raise the bar</button>
      <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>Toggle
theme</button>
    </div>
  );
}
```

What it does: The filter over 5,000 students depends only on `minMarks`. Toggling the theme re-renders the component, but React returns the cached `toppers` without re-filtering — the console log stays silent. Only the "Raise the bar" button (a real dependency change) runs the filter again.

What you will see: the "Filtering 5000 students…" log appears on mount and on each bar raise — but never on theme toggles, even though the component re-rendered.

Example 2 — useCallback + React.memo: the child stops re-rendering

```
import { memo, useCallback, useState } from "react";

const StudentRow = memo(function StudentRow({ name, onSelect }) {
  console.log("Rendering row:", name);
  return <button onClick={() => onSelect(name)}>{name}</button>;
});

function ClassRoom() {
  const [selected, setSelected] = useState("");
  const [count, setCount] = useState(0);

  const selectStudent = useCallback((name) => {
    setSelected(name);
  }, []);

  return (
    <div>
      <StudentRow name="Ali" onSelect={selectStudent} />
      <StudentRow name="Fatima" onSelect={selectStudent} />
      <p>Selected: {selected}</p>
      <button onClick={() => setCount(count + 1)}>Unrelated counter: {count}</button>
    </div>
  );
}
```

What it does: `StudentRow` is wrapped in `memo`, so it re-renders only when its props change. `selectStudent` is wrapped in `useCallback`, so its identity is the same function on every render — the rows' props are genuinely unchanged when the unrelated counter ticks, and memo can do its job.

What you will see: "Rendering row" logs twice on mount, then never again while the counter climbs — remove either wrapper and the rows log on every click.

Example 3 — When NOT to memoize: the cheap calculation

```
function Bill({ price, qty }) {
  // Cheap: two numbers multiplying. useMemo here costs more
  // (cache storage + dependency comparison) than the work itself.
  const total = price * qty;

  return <p>Total: Rs {total}</p>;
}
```

What it does: Nothing fancy — and that is the lesson. A multiplication is rebuilt in nanoseconds; caching it adds machinery with no profit. The video's honesty rule: memoization is a *medicine for a measured problem*,

not a vitamin for every render.

What you will see: nothing different on screen — which is exactly the point. Reach for useMemo when a log or profiler proves the work is heavy, not before.

Quick recap

- `useMemo` **caches a value**, `useCallback` **caches a function**, `React.memo` **skips a child's re-render** when its props are unchanged — the three work as a team.
- Both hooks live by their **dependency arrays**; a missing dep is a stale answer waiting to happen.
- Memoization has a cost — **measure first, memoize second**, and never wrap cheap work.

Common mistakes

1. **Memoizing everything by default** — caches everywhere make code harder to read and can even slow small apps. The fix: start without, profile, and memoize only the proven hot spots.
2. **useCallback without memo (or memo without useCallback)** — a memoized child receiving a brand-new function prop every render re-renders anyway; the pair must arrive together. The fix: when a function goes to a memo child, it goes wrapped.
3. **Wrong dependencies** — a memoized filter that forgets the query in its deps shows last search's results under the new query. The fix: the deps list names everything the computation reads — no exceptions, the linter's warning is your friend.

Practice

1. Take the searchable course list from lesson 5.16 and grow the data to 2,000 generated courses. Put the filter inside `useMemo` keyed on the query, add an unrelated state (a click counter) to the same component, and use console logs to prove the filter runs only when the query changes.
2. Build a `ProductGrid`: rows are a `memo` component receiving a product and an `onAdd` callback. Add items to a cart count in the parent, and show with logs that (a) without `useCallback` every row re-renders, (b) with it, only the parent does.

Practice quiz

1. What does useMemo(fn, deps) cache?

- A function
- A value: React runs fn on the first render, then returns the stored answer until a dependency changes
- A component
- Nothing, it runs fn every render

Answer: A value: React runs fn on the first render, then returns the stored answer until a dependency changes

2. What does useCallback(fn, deps) cache, and why does it matter?

- A value, for speed
- The function itself, keeping its identity stable, which matters when it is passed to a memoized child or used in an effect's dependencies
- A component's props
- The DOM node

Answer: The function itself, keeping its identity stable, which matters when it is passed to a memoized child or used in an effect's dependencies

3. What is the pros' rule for when to memoize?

- Wrap everything just in case
- Measure first, memoize second; memoization costs memory and comparison work, and for cheap work the cache can cost more than the work
- Memoize only on Mondays
- Never memoize, it is always slower

Answer: Measure first, memoize second; memoization costs memory and comparison work, and for cheap work the cache can cost more than the work

---

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