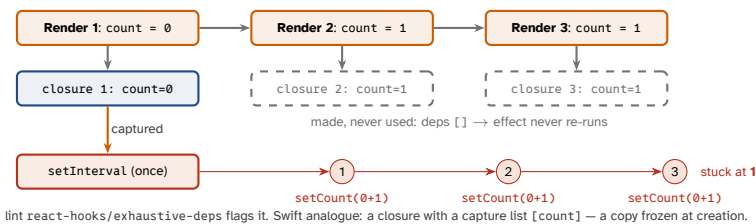


In one line: Hooks are **slots on the fiber**, matched by **call order**. Each render is a **snapshot** its closures capture. Effects sync with the outside world **after** commit; memo hooks keep **identity**. Most bugs: an old snapshot, or a dep with a new identity every render.

Stale closure — `setInterval(() => setCount(count + 1))` in `useEffect(..., [])`



Fixes

1 `setCount(c => c + 1)` — the updater receives the **latest** state; deps stay `[]`

2 `deps [count]` — effect re-runs with the new closure (timer restarts each change)

3 `latest.current = count` each render; the callback reads the ref (a box, not a snapshot)

State

- `useState(init)`: `init` used on mount only — pass a function for costly work. Setter is **stable**; `Object.is(new, old) → no render`. Updates queue: `set(c => c+1)` twice = +2, `set(c+1)` twice = +1.
- `useReducer`: `next = pure(state, action)`; stable dispatch to pass down. Pick it when fields change together. Never mutate: `arr.push(x)`; `set(arr) = same reference = no render`.

Effects — timing, deps, cleanup

- `useEffect`: **after commit, normally after paint**. Deps by `Object.is`; none = every render; `[]` = mount only.
- Cleanup** runs before the next run (old closure) and on unmount. Child effects run before the parent's.
- `useLayoutEffect`: **before paint**, sync — measure and fix without flicker (Fabric: layout readable). Blocks the frame.
- Races**: fast id changes → out-of-order responses. Cleanup cancels (`AbortController`) or ignores (`let stale = true`).
- StrictMode** (dev): mount → cleanup → mount; double subscriptions = missing cleanup. Derived values belong in render, event work in handlers — not effects.

Refs

`useRef` = the same `{current}` box every render; writing it does **not** re-render (timers, latest value). On a host component: the native instance (`focus()`, `measure()`) — an `IBOutlet`. Not read during render.

Example — fetch without races

```
function useUser(id: string) {
  const [user, setUser] = useState<User | null>(null);
  useEffect(() => {
    const ac = new AbortController();
    fetch(`${API}/users/${id}`, { signal: ac.signal })
      .then(r => r.json())
      .then(setUser)
      .catch(e => { if (e.name !== 'AbortError') setErr(e); });
    return () => ac.abort(); // before next id + on unmount
  }, [id]);
  return user;
}
```

React 19 (RN 0.78+)

`use(promise | context)` — suspends on a promise; the one hook allowed in conditions. `useActionState`, `useOptimistic`: async *Actions* with pending/optimistic state. `ref` is a plain prop (no `forwardRef`).

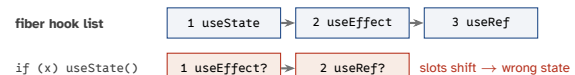
Memoisation — identity, not speed

- `useMemo(fn, deps)` caches a value; `useCallback(fn, deps) = useMemo(() => fn, deps)`; `React.memo(C)` skips a render on shallow-equal props.
- Helps** only when identity is *consumed*: prop of a memo'd child (heavy `FlatList` row), dep of another hook, context value — or a truly expensive computation. Otherwise **waste**; one inline object/arrow/children defeats memo.
- React Compiler** memoises at build time; manual memo gets rare.

Context & external stores

- `useContext`: **every consumer re-renders** when value changes (`Object.is`); memo cannot block it. Memoise value, split contexts (state / dispatch), or use a store with selectors.
- `useSyncExternalStore(subscribe, getSnapshot)`: tear-free reads of a non-React store (Zustand, Redux). `getSnapshot` must return a cached value — a new object per call loops forever.

Rules of Hooks — why



- Top level only — no conditions, loops, or hooks after an early return. No names or keys: *order* is identity.
- Custom hook** (`useX`) shares *logic*, not state.

Interview traps

- ! `useCallback` with no memo'd consumer only adds cost.
- ! Object/array deps built in render → effect every render (or a loop).
- ! `useEffect(async () => ...)` returns a promise, not a cleanup.
- ! Silencing `exhaustive-deps` hides stale closures — restructure.

Remember

Order = identity. Render = snapshot. Effects sync; refs remember.

Likely questions

- Why no conditional hooks? — slots are matched by call order.
- `useEffect` vs `useLayoutEffect`? — after vs before paint.
- When does `useCallback` help? — a memo'd consumer.
- Context over-renders? — split, memoise value, selector store.