LECTURE 28

Hooks and State Management

Giving function components memory and side effects — and sharing that state across a whole app.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. The rules of hooks, and useState for local component state
2. useEffect for side effects: the dependency array and cleanup functions
3. useRef, useMemo, and useCallback for references and performance
4. Lifting state up, the Context API and useContext, and custom hooks
5. A conceptual overview of external state managers: Redux and Zustand

The Rules of Hooks

1. Hooks are special functions that let a function component "hook into" React features like local memory and side effects
2. Every built-in hook starts with use (useState, useEffect...) — custom hooks follow the same convention
3. Rule 1: only call hooks at the top level — never in a loop, condition, or nested function
4. Rule 2: only call hooks from React function components or other custom hooks

Conditional Hook Calls Break the Rules

```
// Wrong: hook called conditionally
function Profile({ isLoggedIn }) {
  if (isLoggedIn) {
    const [name, setName] = useState("");
  }
}

// Correct: always called
function Profile({ isLoggedIn }) {
  const [name, setName] = useState("");
  if (!isLoggedIn) return null;
}
```

Why This Rule Exists

WARNING

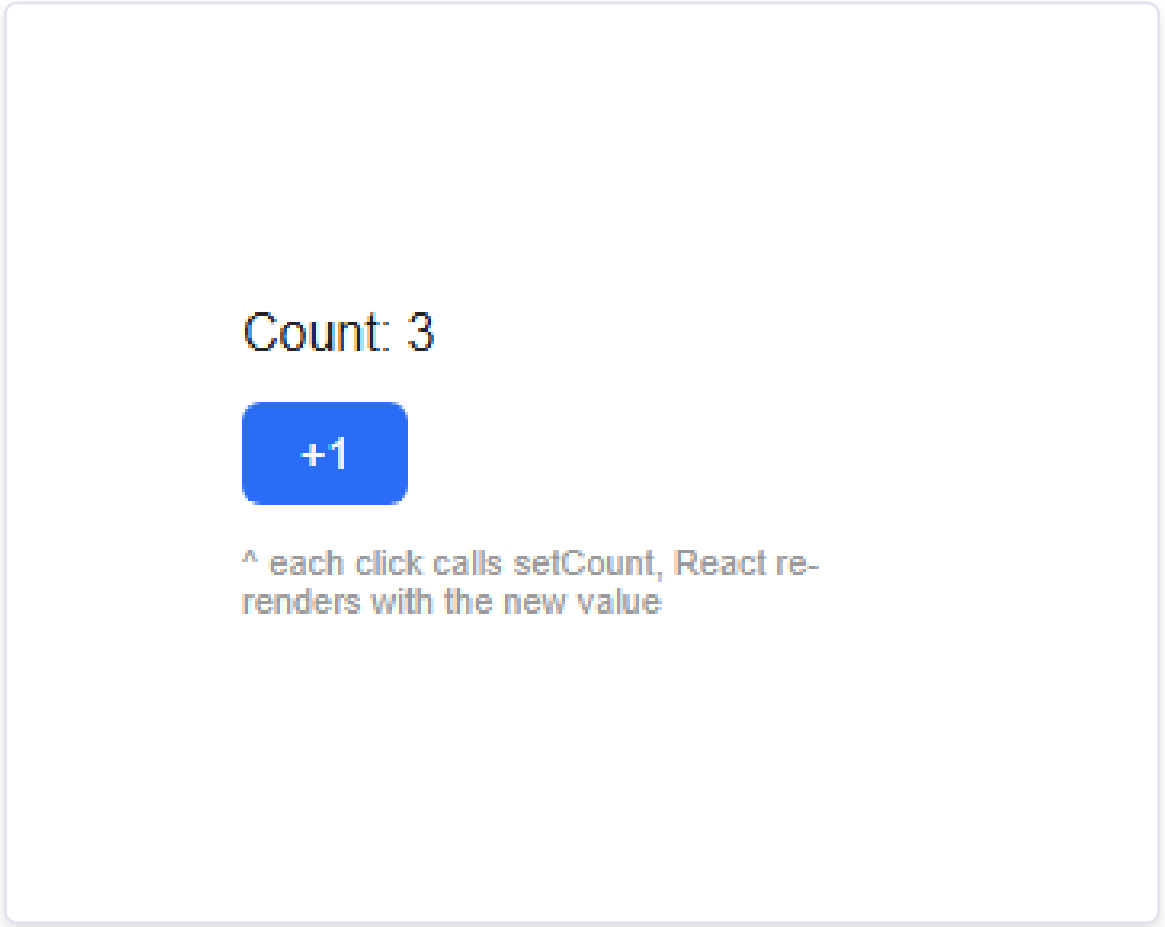
React tracks each hook call by the ORDER it was called in during a render, not by name. If a hook is skipped conditionally, the order shifts, and React can attach the wrong stored value to the wrong hook next render. Calling hooks unconditionally keeps that order stable every time.

Local Component State

1. State is data a component tracks over time that can change in response to user actions, triggering a re-render
2. useState(initialValue) returns an array of exactly two elements: the current value and a setter function
3. Destructured by convention as [thing, setThing]

A Counter

```
function Counter() {
  const [count, setCount] =
    useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={
        () => setCount(count + 1)
      }>
        +1
      </button>
    </div>
  );
}
```



Never Mutate State Directly

WARNING

Writing count++ or items.push(newItem) will NOT trigger a re-render — React only detects a change when you call the setter with a new value. Always create a new value instead of mutating the old one.

Mutating vs. Replacing an Array

```
// Wrong: mutates the existing array
function addItem(item) {
  items.push(item);
  setItems(items);
}

// Correct: creates a new array
function addItem(item) {
  setItems([...items, item]);
}
```

When new state depends on old state inside a handler, pass a function to the setter instead: setCount((prev) => prev + 1).

useEffect: Side Effects

1. A side effect reaches OUTSIDE of simply computing and returning JSX: fetching data, subscribing to a browser event, a timer, direct DOM manipulation

2. useEffect(effectFn, dependencyArray) runs this code in response to rendering

A Clock, With Cleanup

```
function Clock() {
  const [time, setTime] =
    useState(new Date());
  useEffect(() => {
    const id = setInterval(
      () => setTime(new Date()),
      1000
    );
    return () => clearInterval(id);
  }, []);
  return (
    <p>
      Current time:
      {time.toLocaleTimeString()}
    </p>
  );
}
```

The Dependency Array Controls WHEN

Dependency Array	Effect Runs
Omitted entirely	After every render
[] (empty array)	Only once, right after the first render
[a, b]	After the first render, and again whenever a or b changes

Include Everything the Effect Uses

WARNING

Any prop or state variable the effect reads should generally be listed in the dependency array. Leaving one out is a common source of bugs where the effect keeps using a stale, outdated value — most ESLint React setups warn about this.

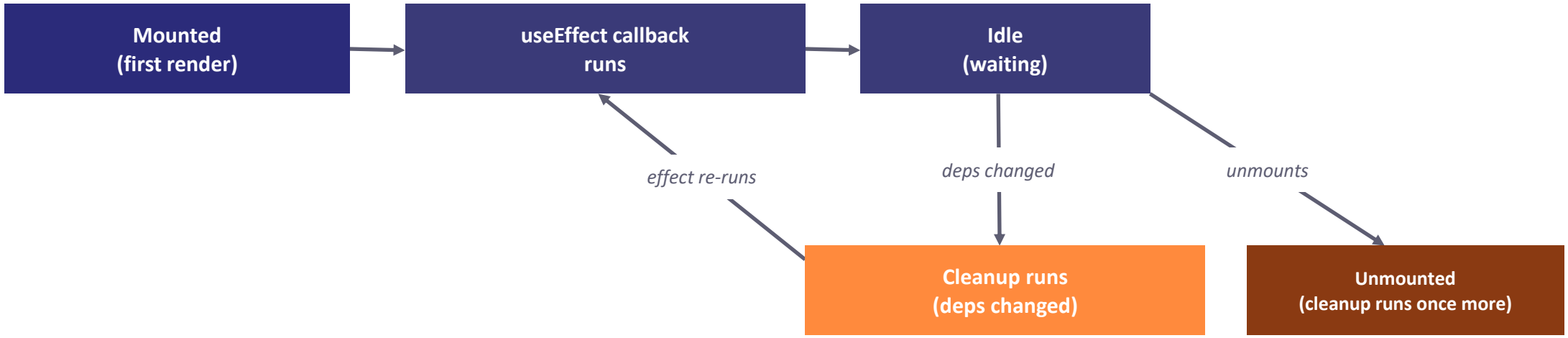
What Cleanup Functions Do

1. If the function passed to useEffect RETURNS another function, React treats it as a cleanup function
2. React calls it right before running the effect again, and when the component unmounts
3. Essential for anything that would otherwise leak: timers, subscriptions, event listeners

Removing an Event Listener

```
useEffect(() => {
  function handleResize() {
    console.log("window resized");
  }
  window.addEventListener(
    "resize", handleResize
  );
  return () => window.removeEventListener(
    "resize", handleResize
  );
}, []);
```

The Mount / Effect / Cleanup Lifecycle



React calls cleanup before every subsequent effect run AND once more when the component unmounts — the same function handles both cases.

useRef: A Value That Survives, Silently

1. Creates a mutable object that persists across renders WITHOUT causing a re-render when it changes

2. Has one property, .current, freely readable and writable

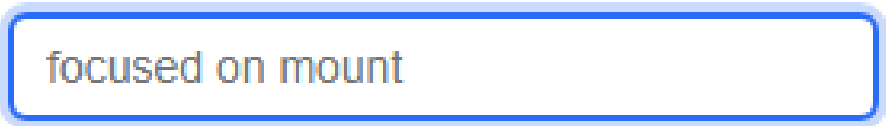
3. Two common uses: (1) accessing a real DOM element directly, (2) storing a mutable value that shouldn't trigger a render, like a timer ID

Auto-Focusing an Input

```
function AutoFocusInput() {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <input ref={inputRef} />
  );
}
```



^ inputRef.current.focus() runs once inside useEffect(..., []), after the DOM node exists

useMemo and useCallback

1. Both avoid unnecessary, expensive recalculation on every render — a technique called memoization
2. useMemo caches the RESULT of a calculation, recomputing only when a listed dependency changes
3. useCallback does the same for FUNCTIONS — it returns the same function reference between renders unless its dependencies change

Caching a Filtered List

```
function ProductList({ products, searchTerm }) {
  const filtered = useMemo(() => {
    return products.filter((p) =>
      p.name.toLowerCase()
        .includes(searchTerm.toLowerCase())
    );
  }, [products, searchTerm]);

  return (
    <ul>
      {filtered.map((p) => (
        <li key={p.id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

A Stable Function Reference

```
function ProductList({ products, onAddToCart }) {
  const handleAdd = useCallback(
    (id) => {
      onAddToCart(id);
    },
    [onAddToCart]
  );
  // same reference between renders
  // unless onAddToCart changes
}
```

Matters mainly when passing a callback to a child optimized with React.memo — a brand-new reference every render would defeat that optimization.

Don't Over-Optimize

TIP

useMemo and useCallback are optimization tools, not something you need on every value or function — they add a small overhead themselves. Reach for them for a genuinely expensive calculation or a measured performance problem, not by default.

Sharing State Between Siblings

1. Moving a piece of state from a child component up to their closest common PARENT
2. Lets multiple sibling components share and stay in sync with the same data
3. Neither sibling owns the state itself — the shared parent does

A Shared celsius Value

```
function TemperatureConverter() {
  const [celsius, setCelsius] =
    useState(0);
  return (
    <div>
      <CelsiusInput
        value={celsius}
        onChange={setCelsius}
      />
      <FahrenheitDisplay
        celsius={celsius}
      />
    </div>
  );
}
```

Solving Prop Drilling

1. Lifting state up works well for a few levels; deeply nested trees make prop drilling (Lecture 27) painful

2. The Context API lets a parent make a value available to ANY descendant, no matter how deeply nested

3. Three steps: createContext, provide a value with <Context.Provider>, consume it anywhere below with useContext

Skipping Straight to the Data

UserBadge reads user directly with useContext -- completely skipping Page and Sidebar, no matter how deep.

```
const UserContext = createContext(null);
function App() {
  const [user] = useState({ name: "Ayesha" });
  return (
    <UserContext.Provider value={user}>
      <Page />
    </UserContext.Provider>
  );
}
function UserBadge() {
  const user = useContext(UserContext);
  return <p>Logged in as {user.name}</p>;
}
```

Extracting Reusable Stateful Logic

1. A custom hook is a JavaScript function whose name starts with use and that calls other hooks inside it

2. Lets you extract and reuse stateful logic between components, the same way a regular function reuses plain logic

useWindowWidth

```
function useWindowWidth() {
  const [width, setWidth] =
    useState(window.innerWidth);
  useEffect(() => {
    function onResize() {
      setWidth(window.innerWidth);
    }
    window.addEventListener("resize", onResize);
    return () =>
      window.removeEventListener("resize", onResize);
  }, []);
  return width;
}
function ResponsiveMessage() {
  const width = useWindowWidth();
  return <p>{width < 600 ? "Mobile" : "Desktop"}</p>;
}
```

Redux vs. Zustand, Conceptually

	Redux	Zustand
Store	One single global store object	A small, hook-based store
State changes via	Actions processed by reducers	Direct function calls
Boilerplate	More — powerful and predictable	Very little
Setup	Wraps the app in a <Provider>	No Provider needed — used as a hook

You Do Not Need These Yet

NOTE

This course does not require Redux or Zustand -- useState, lifting state up, and the Context API are enough for everything you build here. Still worth knowing these names, since you will very likely meet them in real codebases and the advanced course.

KEY TAKEAWAYS

Lecture 28 in Six Points

1. Hooks must be called unconditionally, at the top level of a function component or custom hook — never in loops or conditions.
2. useState gives a component memory across renders; always update state through its setter, never by mutating the previous value.
3. useEffect runs side effects after rendering; its dependency array controls timing, and a returned cleanup function prevents leaks.
4. useRef persists a value or DOM reference without triggering a render; useMemo/useCallback memoize values and functions.
5. Lifting state up shares state via a common parent; the Context API with useContext avoids deep prop drilling.
6. Custom hooks (functions starting with use) let you extract and reuse stateful logic; Redux and Zustand centralize state at scale.