# React Core Hooks: useMemo, useEffect, useCallback, and More

## useEffect

`useEffect` is used to handle side effects in functional components. It replaces lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

Example:  
useEffect(() => {  
 console.log("Component mounted or updated");  
 return () => {  
 console.log("Cleanup before re-run or unmount");  
 };  
}, [dependency]);

• Runs after render.  
• Re-runs only if values in the dependency array change.  
• Cleanup function is useful for removing event listeners or canceling API calls.

## useMemo

`useMemo` is used to memoize the result of a computation so that it’s only recalculated when dependencies change.

Example:  
const expensiveValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

• Prevents expensive calculations on every render.  
• Only recalculates if dependencies change.  
• Useful for performance optimization.

## useCallback

`useCallback` returns a memoized version of a callback function that only changes if dependencies change.

Example:  
const handleClick = useCallback(() => {  
 console.log("Button clicked");  
}, [dependency]);

• Useful when passing callbacks to child components to prevent unnecessary re-renders.  
• Commonly used with `React.memo`.

## useRef

`useRef` is used to create a mutable reference that persists across renders. It does not trigger re-renders when updated.

Example:  
const inputRef = useRef(null);  
<input ref={inputRef} />

• Great for accessing and interacting with DOM elements directly.  
• Can also be used to persist values between renders without triggering a re-render.

## useState

`useState` is the most basic hook used to manage state in a functional component.

Example:  
const [count, setCount] = useState(0);

• The first value is the current state.  
• The second value is a function to update the state.  
• Updating state causes the component to re-render.

## useReducer

`useReducer` is used for managing more complex state logic. It’s a good alternative to `useState` when dealing with multiple sub-values or actions.

Example:  
function reducer(state, action) {  
 switch (action.type) {  
 case 'increment':  
 return { count: state.count + 1 };  
 default:  
 return state;  
 }  
}  
const [state, dispatch] = useReducer(reducer, { count: 0 });

• Returns a state and a dispatch function.  
• Useful for more predictable and testable state transitions.

## useContext

`useContext` allows you to access context values in any functional component.

Example:  
const value = useContext(MyContext);

• Reads context and subscribes to its changes.  
• Helps avoid prop-drilling.

## Summary and Best Practices

• Use `useMemo` and `useCallback` for optimization, not default behavior.  
  
• Use `useEffect` for side effects such as data fetching, subscriptions, and timers.  
  
• Use `useRef` for mutable references and DOM access.  
  
• Prefer `useReducer` for state with complex transitions.  
  
• All hooks must be called at the top level of a component or custom hook.