**Question 1: What are React hooks? How do useState() and useEffect() hooks work in functional components?**

**What are React Hooks?**

React Hooks are functions that let you “hook into” React state and lifecycle features from functional components.

Before hooks, these features were only available in class components. Hooks allow you to write cleaner and more reusable logic in function components.

**useState(): Managing State**

**Purpose:**  
Allows you to add **state** to a functional component.

**Syntax:**

const [state, setState] = useState(initialValue);

**Example:**

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

**useEffect(): Handling Side Effects**

**Purpose:**  
Runs **side effects** like API calls, subscriptions, or manual DOM manipulation — similar to componentDidMount(), componentDidUpdate(), and componentWillUnmount() in class components.

**Syntax:**

useEffect(() => {

// Side effect code

return () => {

// Cleanup code (optional)

};

}, [dependencies]);

* Runs **after every render** by default.
* Provide a **dependency array** to control when it runs:
  + [] = only once (on mount)
  + [value] = runs when value changes

**Example:**

import React, { useState, useEffect } from 'react';

function Timer() {

const [seconds, setSeconds] = useState(0);

useEffect(() => {

const interval = setInterval(() => {

setSeconds(prev => prev + 1);

}, 1000);

// Cleanup function

return () => clearInterval(interval);

}, []); // Runs once on mount

return <p>Timer: {seconds}s</p>;

}

**🔧 Problems Hooks Solved in React**

**1. ✅ Complex Logic in Class Components Was Hard to Reuse**

* In class components, logic was often split across lifecycle methods (componentDidMount, componentDidUpdate, etc.), making code hard to manage.
* Reusing logic meant creating **Higher-Order Components (HOCs)** or **Render Props**, which could get messy.

**Question 2: What problems did hooks solve in React development? Why are hooks considered an important addition to React?**

**✔ Hooks Solution:**  
Hooks like useEffect, useState, and custom hooks allow logic to be **encapsulated** and **reused cleanly** across components.

**2. ✅ Confusing this Keyword**

* In class components, you had to bind methods manually or use arrow functions to avoid issues with this.

**✔ Hooks Solution:**  
Hooks work inside **functional components**, which don’t use this — making them easier to read and write.

**3. ✅ Too Much Boilerplate**

* Class components needed a lot of extra code: constructors, method bindings, and multiple lifecycle methods.

**✔ Hooks Solution:**  
Hooks reduce boilerplate. You can manage state and side effects with just a few lines of code.

**4. ✅ Poor Separation of Concerns in Class Lifecycle Methods**

* Different concerns (e.g., fetching data and setting up subscriptions) had to be put into the same lifecycle method, which mixed logic and made it hard to organize.

**✔ Hooks Solution:**  
With hooks, you can **split related logic** into smaller functions (custom hooks) — improving **separation of concerns**.

**5. ✅ Sharing Stateful Logic Was Hard**

* Logic that involved state couldn’t be easily reused without complex patterns like HOCs or render props.

**✔ Hooks Solution:**  
You can create **custom hooks** to share stateful logic cleanly across multiple components

**Question 3: What is useReducer ? How we use in react app?**

**What is useReducer?**

useReducer is a hook that is similar to useState, but gives you more control over state transitions by using a reducer function, like in Redux.

**🔧 Syntax:**

const [state, dispatch] = useReducer(reducer, initialState);

* state – the current state
* dispatch – function to send actions
* reducer – function that handles how state should update
* initialState – the starting state

**🔄 Reducer Function**

A pure function that takes the current state and an action, and returns a new state.

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

return state;

}

}

🧩 Example: Counter using useReducer

import React, { useReducer } from 'react';

// Reducer function

function reducer(state, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

return state;

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, { count: 0 });

return (

<div>

<h2>Count: {state.count}</h2>

<button onClick={() => dispatch({ type: 'increment' })}>+</button>

<button onClick={() => dispatch({ type: 'decrement' })}>-</button>

</div>

);

}

export default Counter;

**Question 4: What is the purpose of useCallback & useMemo Hooks?**

**useCallback()**

**📌 Purpose:**

useCallback is used to **memoize a function** — it returns a **cached version of the function** that only changes if one of its dependencies changes.

This helps prevent **unnecessary re-creations of functions** on every render, which is useful when passing callbacks to **child components** (especially memoized ones).

**📦 Syntax:**

const memoizedCallback = useCallback(() => {

// function code

}, [dependencies]);

**✅ Example:**

import React, { useState, useCallback } from 'react';

function Button({ handleClick }) {

console.log("Button rendered");

return <button onClick={handleClick}>Click Me</button>;

}

const MemoButton = React.memo(Button);

function App() {

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

const increment = useCallback(() => {

setCount(prev => prev + 1);

}, []);

return (

<div>

<h2>Count: {count}</h2>

<MemoButton handleClick={increment} />

</div>

);

}

🔍 Without useCallback, the function increment would be re-created every render, causing MemoButton to re-render unnecessarily.

**🔹 useMemo()**

**📌 Purpose:**

useMemo is used to **memoize the result of a calculation** — it only recomputes the result when dependencies change.

This is useful for **expensive computations** that shouldn't run on every render.

**📦 Syntax:**

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

**✅ Example:**

import React, { useState, useMemo } from 'react';

function App() {

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

const [multiplier, setMultiplier] = useState(2);

const expensiveCalculation = (num) => {

console.log("Calculating...");

for (let i = 0; i < 1e7; i++) {} // simulate delay

return num \* multiplier;

};

const result = useMemo(() => expensiveCalculation(count), [count, multiplier]);

return (

<div>

<h2>Result: {result}</h2>

<button onClick={() => setCount(count + 1)}>Increase Count</button>

<button onClick={() => setMultiplier(multiplier + 1)}>Increase Multiplier</button>

</div>

);

}

🧠 Without useMemo, the calculation would run on **every re-render**, even when count didn’t change.

**Question 5: What’s the Difference between the useCallback & useMemo Hooks?**

**Key Difference Between useCallback and useMemo**

|  |  |  |
| --- | --- | --- |
| **Feature** | **useCallback** | **useMemo** |
| **Purpose** | Memoizes a function | Memoizes the result of a computation |
| **Returns** | A function | A value (number, object, array, etc.) |
| **Use Case** | When you pass functions to child components | When you do expensive calculations |
| **Syntax** | useCallback(fn, deps) | useMemo(() => compute, deps) |

💡 Think of it like this:

* 🔁 useCallback ⇒ "Keep this function the same unless dependencies change."
* 🧮 useMemo ⇒ "Keep this computed value the same unless dependencies change."

✅ Simple Examples

useCallback Example (Memoizing a Function):

const handleClick = useCallback(() => {

console.log("Clicked");

}, []);

Use when:

* You want to pass handleClick to a child component.
* You want to avoid re-creating the function on every render.

useMemo Example (Memoizing a Value):

jsx

CopyEdit

const result = useMemo(() => {

return expensiveCalculation(input);

}, [input]);

Use when:

* You want to avoid recalculating something costly on every render.
* You need to store derived data efficiently.

🧪 Real-World Example Using Both:

import React, { useState, useMemo, useCallback } from 'react';

function ExpensiveComponent({ onClick }) {

console.log("Child rendered");

return <button onClick={onClick}>Click Me</button>;

}

const MemoizedComponent = React.memo(ExpensiveComponent);

function App() {

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

const [input, setInput] = useState('');

const expensiveValue = useMemo(() => {

console.log("Calculating...");

return input.length \* 1000;

}, [input]);

const handleClick = useCallback(() => {

setCount(count + 1);

}, [count]);

return (

<div>

<input value={input} onChange={e => setInput(e.target.value)} />

<p>Expensive Value: {expensiveValue}</p>

<MemoizedComponent onClick={handleClick} />

</div>

);

}

**Question 6: What is useRef ? How to work in react app?**

**What is useRef in React?**

useRef is a React Hook that provides a way to persist values across renders without causing a re-render when the value changes.

It can be used for:

1. Accessing DOM elements directly.
2. Storing mutable values that don’t trigger a component re-render when updated.

**🧪 Syntax:**

const myRef = useRef(initialValue);

myRef.current holds the mutable value or DOM node.

**Use Case 1: Accessing DOM elements**

import React, { useRef } from 'react';

function InputFocus() {

const inputRef = useRef(null);

const focusInput = () => {

inputRef.current.focus(); // Access the DOM node directly};

return (

<div>

<input ref={inputRef} type="text" />

<button onClick={focusInput}>Focus Input</button>

</div>

);

}

**Use Case 2: Storing mutable values (like instance variables)**

import React, { useRef, useState, useEffect } from 'react';

function Timer() {

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

const intervalRef = useRef();

useEffect(() => {

intervalRef.current = setInterval(() => {

setCount(prev => prev + 1);

}, 1000);

// Cleanup interval on unmount

return () => clearInterval(intervalRef.current);

}, []);

return <h2>Timer: {count}s</h2>;

}