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

* React **Hooks** are special functions that let you “hook into” React features (like state and lifecycle methods) inside **functional components**.
* Before hooks, only **class components** could use state and lifecycle methods.
* Introduced in **React 16.8**.
* Most common hooks: useState(), useEffect(), useContext(), etc.

**How useState() works:-**

* useState allows you to **add and manage state** inside a functional component.
* It returns an **array with two values**:
  1. The current state value
  2. A function to update the state

**Example:**

import React, { useState } from "react";

function Counter() {

const [count, setCount] = useState(0); // count = state, setCount = update function

return (

<div>

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

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

<button onClick={() => setCount(count - 1)}>Decrease</button>

</div>

);

}

export default Counter;

**How useEffect() works:-**

* useEffect lets you perform **side effects** in a component.
* Side effects = tasks that are not directly related to rendering UI, like:
  + Fetching data (API calls)
  + Setting up timers (setInterval, setTimeout)
  + Updating the DOM manually
  + Logging values

**Syntax:**

useEffect(() => {

// effect code (runs after render)

return () => {

// cleanup code (runs when component unmounts)

};

}, [dependencies]);

**Example:**

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

function Timer() {

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

useEffect(() => {

console.log("Count updated:", count);

}, [count]); // runs only when 'count' changes

return (

<div>

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

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

</div>

);

}

export default Timer;

**Q-2) What problems did hooks solve in React development? Why are hooksconsidered an important addition to React?**

**Problems Before Hooks:-**

1. Class components were **complex** (needed this, constructors).
2. **Reusing stateful logic** was hard (HOCs, render props).
3. Lifecycle methods split related code → **hard to maintain**.
4. **Testing** class components was harder.

**How Hooks Solved This:-**

1. Made components **simpler** with functions.
2. Allowed **code reuse** via custom hooks.
3. useEffect keeps side-effect logic **organized**.
4. Components became **shorter, cleaner, and easier to test**.

**Why Hooks are an Important Addition:-**

* They **modernized React** → making functional components powerful enough to replace most class components.
* They **improved developer productivity** → easier to write, maintain, and reuse code.
* They **simplified the learning curve** for new developers (no need to deeply learn classes and this binding).
* They **enabled advanced features** like custom hooks and cleaner state management.

**Q-3) What is useReducer ? How we use in react app?**

* useReducer is a React **hook for state management**.
* It is an alternative to useState, used when state logic is **complex** or involves **multiple sub-values**.
* Works similar to **Redux** (action → reducer → new state).

Syntax:

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

* state → current state
* dispatch → function to send actions
* reducer → pure function (state, action) => newState

**How to use in a React App?**

**Example: Counter with useReducer**

import React, { useReducer } from "react";

1. Reducer function

function reducer(state, action) {

switch (action.type) {

case "INCREMENT":

return { count: state.count + 1 };

case "DECREMENT":

return { count: state.count - 1 };

case "RESET":

return { count: 0 };

default:

return state;

}

}

function Counter() {

2. useReducer hook

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>

<button onClick={() => dispatch({ type: "RESET" })}>Reset</button>

</div>

);

}

export default Counter;

**When to use useReducer?**

* When state is **complex** (multiple values, nested objects).
* When next state depends on **previous state**.
* When you want **Redux-like structure** without installing Redux.

**Q-4) What is the purpose of useCallback & useMemo Hooks?**

**1. useCallback**

* Purpose → **Memoizes a function** so it’s not recreated on every render.
* Useful when passing a function as a **prop** to child components (to prevent unnecessary re-renders).

**Syntax:**

const memoizedCallback = useCallback(() => {

// function logic

}, [dependencies]);

**Example:**

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

function Button({ onClick, children }) {

console.log("Button rendered:", children);

return <button onClick={onClick}>{children}</button>;

}

function App() {

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

// function will not be recreated unless count changes

const increment = useCallback(() => {

setCount(c => c + 1);

}, []);

return (

<div>

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

<Button onClick={increment}>Increase</Button>

</div>

);

}

**2. useMemo**

* Purpose → **Memoizes a computed value** so it doesn’t recalculate on every render.
* Useful for **expensive calculations** (e.g., filtering, sorting, big loops).

**Syntax:**

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

**Example:**

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

function App() {

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

const [text, setText] = useState("");

// Expensive calculation only runs when 'count' changes

const expensiveValue = useMemo(() => {

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

return count \* 1000;

}, [count]);

return (

<div>

<h2>Expensive Value: {expensiveValue}</h2>

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

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

</div>

);

}

**Q-5) What’s the Difference between the useCallback & useMemo Hooks?**

| **Feature** | **useCallback 🖇️** | **useMemo 💡** |
| --- | --- | --- |
| **Purpose** | Memoizes a **function** | Memoizes a **computed value** |
| **Returns** | A **memoized callback function** | A **memoized result/value** |
| **Use Case** | When passing functions to child components (to avoid re-renders) | When doing expensive calculations (to avoid recalculating) |
| **Syntax** | useCallback(() => { ... }, [deps]) | useMemo(() => computeExpensive(), [deps]) |
| **Example** | const memoFn = useCallback(fn, [a]) | const memoVal = useMemo(() => calc(a), [a]) |
| **Optimization** | Prevents **function recreation** | Prevents **recalculation of value** |

**Q-6) What is useRef ? How to work in react app?**

* useRef is a React **hook** that gives you a way to create a **mutable reference** (an object with a .current property).
* It does **not re-render** the component when its value changes (unlike state).
* Commonly used for:
  1. **Accessing DOM elements** directly.
  2. **Storing mutable values** that you don’t want to trigger re-renders.
  3. **Keeping previous values**.

**Syntax:**

const refContainer = useRef(initialValue);

**How useRef works in a React App**

**1. Accessing DOM elements**

import React, { useRef } from "react";

function InputFocus() {

const inputRef = useRef(null);

const focusInput = () => {

inputRef.current.focus(); // directly accesses the DOM element

};

return (

<div>

<input ref={inputRef} type="text" placeholder="Type here..." />

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

</div>

);

}

export default InputFocus;

**2. Storing values without re-render**

import React, { useRef, useState } from "react";

function Timer() {

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

const intervalRef = useRef(null);

const startTimer = () => {

intervalRef.current = setInterval(() => {

setCount(c => c + 1);

}, 1000);

};

const stopTimer = () => {

clearInterval(intervalRef.current);

};

return (

<div>

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

<button onClick={startTimer}>Start</button>

<button onClick={stopTimer}>Stop</button>

</div>

);

}

export default Timer;