**HOOKS:**

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

React hooks are functions that allow you to "hook into" React state and lifecycle features from functional components. Before hooks, state and lifecycle methods were only available in class components. Hooks were introduced in React 16.8 to enable functional components to have the same capabilities as class components, such as managing state and side effects.

Two of the most commonly used hooks are useState() and useEffect(). Here's how each works:

**useState()**

useState() is a hook that allows you to add state to your functional components.

* **Syntax**:

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

**useEffect Hook**

The useEffect hook allows you to perform side effects (e.g., fetching data, updating the DOM, setting up subscriptions) in your functional components. It's similar to lifecycle methods (componentDidMount, componentDidUpdate, and componentWillUnmount) in class components.

**Syntax:**

useEffect(() => {

// Side effect logic here

}, [dependencies]);

* The first argument is a function that contains the side effect code.
* The second argument is an array of dependencies, which tells React when to re-run the effect. If the array is empty, the effect runs only once (on mount and unmount).

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

1. Complex State Logic in Class Components

* Problem: Managing state in class components often required multiple lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount), making the code difficult to read and maintain.
* Solution: Hooks like useState and useReducer allow managing state and side effects in a more organized and reusable way within functional components.

2. Code Reusability and Avoiding Wrapper Hell (Higher-Order Components & Render Props)

* Problem: Before hooks, developers used Higher-Order Components (HOCs) and Render Props to share stateful logic between components, leading to deeply nested structures and difficult-to-maintain code.
* Solution: Hooks like useContext, useEffect, and custom hooks allow for better logic reuse without unnecessary component nesting.

3. Complex Lifecycle Methods in Class Components

* Problem: In class components, componentDidMount, componentDidUpdate, and componentWillUnmount often contained related logic that was split across multiple lifecycle methods, making the code harder to follow.
* Solution: The useEffect hook allows combining all related side effects into a single function.

4. No Need for this (Easier to Understand & Debug)

* Problem: In class components, developers often had to bind this in constructor methods, leading to confusion and errors.
* Solution: Functional components with hooks avoid this altogether, making them easier to reason about.

5. Functional Components Can Now Handle State and Side Effects

* Problem: Before hooks, functional components were stateless and could not handle side effects. Developers had to use class components for these features.
* Solution: Hooks like useState, useEffect, and useReducer bring state management and lifecycle control to functional components.

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

useReducer is a React hook that provides a more structured way to manage complex state logic in functional components. It is an alternative to useState, especially useful when state transitions depend on the previous state or involve multiple actions.

Why Use useReducer Instead of useState?

✅ When state logic is complex (e.g., multiple related states).  
✅ When state transitions involve multiple actions.  
✅ When state changes are triggered by user interactions (e.g., form handling).  
✅ Helps separate state logic from component UI (better maintainability).

Syntax of useReducer

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

* reducerFunction: A function that determines how state should change based on an action.
* initialState: The initial value of the state.
* state: The current state value.
* dispatch: A function that triggers actions to update the state

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

Both useCallback and useMemo are React hooks used for performance optimization by preventing unnecessary computations or re-renders in functional components.

1. useCallback Hook

useCallback memoizes a function, ensuring that it does not get re-created on every render unless its dependencies change.

Why Use useCallback?

✅ Prevents unnecessary re-creation of functions.  
✅ Optimizes performance when passing functions as props to child components.  
✅ Helps avoid unnecessary re-renders of child components.

Syntax

const memoizedFunction = useCallback(() => {

// Function logic

}, [dependencies]);

* useCallback returns a memoized version of the function.
* The function is re-created only if the dependencies change.

What Happens Here?

1. Without useCallback: handleClick would be a new function on every render, causing Child to re-render unnecessarily.
2. With useCallback: handleClick stays the same between renders unless dependencies change, preventing unnecessary re-renders of Child.

2. useMemo Hook

useMemo memoizes the result of a computation, ensuring that it is only re-computed when its dependencies change.

Why Use useMemo?

✅ Prevents expensive recalculations on every render.  
✅ Optimizes performance for computationally heavy functions.  
✅ Helps avoid unnecessary re-renders by memoizing values.

Syntax

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

* useMemo caches the result of the computation.
* Recalculates only if the dependency changes.

Use useCallback when passing functions as props to optimize rendering.  
Use useMemo when an expensive computation is being re-run unnecessarily.

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

Key Difference Between useCallback and useMemo

Both useCallback and useMemo optimize performance in React by preventing unnecessary computations, but they serve different purposes.

| Feature | useCallback | useMemo |
| --- | --- | --- |
| Purpose | Memoizes a function to prevent re-creation | Memoizes a computed value to prevent re-computation |
| Return Type | Returns a memoized function | Returns a memoized value (result of computation) |
| Use Case | When passing functions as props to child components to avoid unnecessary re-renders | When performing expensive calculations that should only re-run when dependencies change |
| Dependencies | Recreates the function only if dependencies change | Recomputes the value only if dependencies change |
| Example Usage | Preventing child components from re-rendering unnecessarily due to function re-creation | Avoiding expensive calculations from running on every render |

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

useRef is a React hook that creates a mutable reference (ref) that persists across renders without causing re-renders when updated. It is mainly used for:  
✅ Accessing & manipulating DOM elements (e.g., focusing an input field).  
✅ Persisting values without re-rendering (like an instance variable in class components).  
✅ Storing previous values of state without triggering re-renders.

Syntax of useRef

const ref = useRef(initialValue);

* ref.current holds the value.
* Unlike useState, updating ref.current does not trigger a re-render.

To interact with the DOM (e.g., focusing input fields, playing videos).  
To persist values across renders (e.g., storing previous values).  
To prevent unnecessary re-renders when state changes.

**LIST AND KEYS:**

**Question 1: How do you render a list of items in React? Why is it important to use keys when rendering lists?**

Rendering lists in React is done using JavaScript’s map() method inside JSX.

1️⃣ Basic Example: Rendering a List

import React from "react";

function ItemList() {

const items = ["Apple", "Banana", "Orange"];

return (

<ul>

{items.map((item, index) => (

<li key={index}>{item}</li> // Key is required

))}

</ul>

);

}

export default ItemList;

🔹 How it Works?

* map() iterates over the items array.
* Each item is rendered as an <li> element.
* key={index} is added to uniquely identify each element.

2️⃣ Why Are Keys Important in Lists?

✅ Keys help React efficiently update the UI

When React renders a list, it uses keys to track which items changed, added, or removed. This helps optimize rendering performance.

🚨 What Happens Without Keys?

* React re-renders the entire list unnecessarily.
* Can cause UI issues when elements are dynamically added/removed.

**Question 2: What are keys in React, and what happens if you do not provide a unique key?**

Keys are unique identifiers assigned to elements when rendering lists in React. They help React efficiently update and re-render components by tracking which items have changed, been added, or removed.

🔹 Why Are Keys Important?

✅ Improve performance by helping React identify changes efficiently.  
✅ Prevent UI bugs when reordering, adding, or deleting elements in a list.  
✅ Avoid unnecessary re-renders of unchanged elements.

How It Works?

* React uses keys to differentiate items in the list.
* When an item is added or removed, React only updates the necessary elements.

What Happens If You Don’t Provide a Unique Key?

Without a unique key, React shows a warning:

"Each child in a list should have a unique 'key' prop."

Problems When Using Index as Key

1. Reordering causes incorrect updates
   * If an item moves, React may update the wrong element, leading to UI bugs.
2. Items retain incorrect state
   * If you have an input field in each list item, input values may jump between elements.
3. Performance issues
   * React re-renders more elements than necessary, slowing down the app.