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

**Ans:-**

What Are React Hooks?

**🎣 Commonly Used Hooks:**

* useState() – for managing state
* useEffect() – for side effects (like data fetching, timers)
* useContext() – to use Context API
* useRef() – to persist values between renders without re-rendering
* useMemo(), useCallback() – for performance optimization
* Custom Hooks – user-defined reusable logi

**useState()**

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>

);

}

📝 Explanation:

count is the state variable.

setCount is a function to update count.

useState(0) initializes it with 0.

**useEffect()**

import React, { useEffect } from 'react';

function Example() {

useEffect(() => {

console.log("Component mounted");

return () => {

console.log("Component will unmount");

};

}, []); // Empty array → run once (on mount only)

return <h1>Hello!</h1>;

}

* Runs after the component renders.
* Used for things like:
  + Fetching data
  + Subscribing to events
  + Updating the DOM
  + Starting/clearing timers

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

**ANS:-**

1. Confusing Component Logic

* Related logic was scattered across lifecycle methods like:
  + componentDidMount
  + componentDidUpdate
  + componentWillUnmount

**2. Code Reusability Was Hard**

* **No easy way to reuse stateful logic** between components.
* You had to use:
  + **Higher-Order Components (HOCs)**
  + **Render Props**
* These patterns often caused **"wrapper hell"** or made code harder to read.

**3. Too Much Boilerplate Code**

* Class components required:
  + Binding methods (this.handleClick = this.handleClick.bind(this))
  + Understanding this keyword
* This added **extra complexity** especially for beginners.

**4. Large Components Became Hard to Manage**

* With growing features, class components often became **very large**.
* Managing multiple states and side effects in one component became messy and hard to maintain.

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

**ANS:-**

useReducer is a React hook used for managing complex state logic in a predictable way.

It is an alternative to useState, especially useful when:

* You have multiple related state values
* You need to perform complex updates based on actions
* The state logic needs to be easily testable or reusable

🧠 It’s similar to Redux logic:

You dispatch actions and a reducer function updates the state.

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

**ANS:-**

eact re-renders components when state or props change, and during each render, functions and calculations are recreated. This can lead to performance issues in large apps.

To optimize performance, React provides:

🧠 useCallback() – Memoizes functions

🧮 useMemo() – Memoizes calculated values

**useCallback()**

* **A child component**
* **An optimized component using React.memo()**
* **An event handler that should not change every render**

**useMemo()**

|  |
| --- |
|  |

|  |
| --- |
| **Prevent expensive recalculations** |

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

**ANS:-**

1. useCallback – Memoizes a Function

🧠 Purpose:

Prevents function recreation on every render unless dependencies change.

🔧 Syntax:

jsx

CopyEdit

const memoizedCallback = useCallback(() => {

// function code

}, [dependencies]);

✅ Use When:

* You want to pass a stable function to a child component (to prevent re-render)
* You want to avoid recreating functions on every render

**2. useMemo – Memoizes a Computed Value**

**🧠 Purpose:**

**Caches the result of a computation to avoid recalculating it on every render.**

**🔧 Syntax:**

**jsx**

**CopyEdit**

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

**✅ Use When:**

* **You want to avoid expensive calculations on every render**
* **You need to compute derived data based on state/props**

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

**ANS:-**

What is useRef?

useRef is a React hook that returns a mutable object which persists for the entire lifetime of the component.

js

CopyEdit

const myRef = useRef(initialValue);

It is commonly used for:

1. Accessing DOM elements directly
2. Storing mutable values that don't cause re-renders when changed
3. Keeping track of previous values

🔧 Syntax:

js

CopyEdit

const ref = useRef(initialValue);

* Returns an object: { current: initialValue }
* Updating ref.current does not cause a re-render

**🔑 Key Points:**

* **useRef doesn’t trigger a re-render when updated**
* **Useful for both DOM refs and persistent variables**
* **Great for working with third-party libraries or custom animations**

**Task 1: • Create a functional component with a counter using the useState() hook. Include buttons to increment and decrement the counter.**import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // Initialize state to 0

const increment = () => setCount(count + 1); // Increase count

const decrement = () => setCount(count - 1); // Decrease count

return (

<div style={{ textAlign: 'center', marginTop: '50px' }}>

<h1>Counter: {count}</h1>

<button onClick={increment} style={{ marginRight: '10px' }}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

export default Counter;

**Task 2: • Use the useEffect() hook to fetch and display data from an API when the component mounts.**

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

function DataFetcher() {

const [data, setData] = useState(null);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

useEffect(() => {

// Fetch data when component mounts

fetch('https://jsonplaceholder.typicode.com/posts/1')

.then((response) => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

})

.then((json) => {

setData(json);

setLoading(false);

})

.catch((error) => {

setError(error.message);

setLoading(false);

});

}, []); // Empty dependency array = run only once on mount

if (loading) return <p>Loading...</p>;

if (error) return <p>Error: {error}</p>;

return (

<div>

<h3>Post Title: {data.title}</h3>

<p>{data.body}</p>

</div>

);

}

export default DataFetcher;

**Task 3: • Create react app with use of useSelector & useDispatch.**

**Ans:-**

// src/store.js

import { createStore } from 'redux';

// Initial state

const initialState = {

count: 0,

};

// Reducer function

function counterReducer(state = initialState, action) {

switch (action.type) {

case 'increment':

return { count: state.count + 1 };

case 'decrement':

return { count: state.count - 1 };

default:

return state;

}

}

// Create store

const store = createStore(counterReducer);

export default store;

// src/index.js

import React from 'react';

import ReactDOM from 'react-dom/client';

import { Provider } from 'react-redux';

import App from './App';

import store from './store';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<Provider store={store}>

<App />

</Provider>

);

// src/App.js

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

function App() {

const count = useSelector(state => state.count); // Select count from redux state

const dispatch = useDispatch();

return (

<div style={{ textAlign: 'center', marginTop: '50px' }}>

<h1>Redux Counter: {count}</h1>

<button onClick={() => dispatch({ type: 'increment' })} style={{ marginRight: '10px' }}>

Increment

</button>

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

Decrement

</button>

</div>

);

}

export default App;

**Task 4: • Create react app to avoid re-renders in react application by useRef?**

**Ans:-**

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

function RenderAvoider() {

const [renderCount, setRenderCount] = useState(0); // To show how many times component re-rendered

const countRef = useRef(0); // Mutable value that won't trigger re-render

// Increment ref without causing re-render

const incrementRef = () => {

countRef.current += 1;

console.log('Ref count:', countRef.current);

};

// Increment state to cause re-render

const incrementRender = () => {

setRenderCount(prev => prev + 1);

};

return (

<div style={{ textAlign: 'center', marginTop: '50px' }}>

<h2>Component Render Count: {renderCount}</h2>

<button onClick={incrementRef} style={{ marginRight: '10px' }}>

Increment useRef (No re-render)

</button>

<button onClick={incrementRender}>

Increment State (Triggers re-render)

</button>

</div>

);

}

export default RenderAvoider;