Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

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

Ans.

React Hooks are special functions introduced in React 16.8 that allow you to use state, lifecycle features, and other React capabilities in functional components, without writing class components.

Hooks let you:

- Manage **state**
- Perform **side effects** (API calls, subscriptions, DOM updates)
- Reuse logic across components

Common hooks include:

- useState
- useEffect
- useContext
- useRef
- useMemo

1. useState() Hook

What does useState do?

useState allows you to add **state** to a functional component.

Syntax

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

How it works

- state → current state value
- setState → function to update the state
- initialValue → initial state value

Example :

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

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

Key Points

- State updates cause **re-render**

- State is **local** to the component
- Can manage multiple states using multiple useState calls

2. useEffect() Hook

What does useEffect do?

useEffect lets you perform **side effects** in functional components.

Side effects include:

- Fetching data
- Updating the DOM
- Timers
- Subscriptions

Syntax

```
useEffect(() => {
  // side effect code
  return () => {
    // cleanup (optional)
  };
}, [dependencies]);
```

How useEffect Works (3 Common Cases)

1. Runs after every render (no dependency array)

```
useEffect(() => {
  console.log("Component rendered");
});
```

2. Runs only once (componentDidMount)

```
useEffect(() => {
  console.log("Component mounted");
}, []);
```

3. Runs when specific state/props change (componentDidUpdate)

```
useEffect(() => {
  console.log("Count changed");
}, [count]);
```

Cleanup Example (componentWillUnmount) :

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log("Running...");
  }, 1000);

  return () => {
    clearInterval(timer);
  };
}, []);
```

Comparison with Class Lifecycle Methods

Class Component	Hook Equivalent
componentDidMount	useEffect(() => {}, [])
componentDidUpdate	useEffect(() => {}, [deps])
componentWillUnmount	Cleanup function in useEffect

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

Ans.

Before Hooks, React developers mainly used class components for state and lifecycle logic. This created several problems that Hooks were designed to fix.

1. Complex and Hard-to-Reuse Logic

Problem (Before Hooks)

- Logic related to one feature was spread across lifecycle methods
    - componentDidMount
    - componentDidUpdate
    - componentWillUnmount
- Reusing stateful logic required:
    - Higher-Order Components (HOCs) or
    - Render props

- These patterns caused wrapper hell and made code harder to follow.

Solution with Hooks

- Hooks allow you to group related logic together
- Logic can be extracted into custom hooks

```
function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);

    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return width;
}
```

2. Confusing this Keyword in Class Components

Problem

- Developers often struggled with:
    - this binding
    - Arrow functions vs .bind(this)

- Easy source of bugs and confusion, especially for beginners

Solution with Hooks

- Functional components do not use this
- Code is simpler and more predictable

```
// No this, no binding
const [count, setCount] = useState(0);
```

3. Large, Unmaintainable Class Components

Problem

- Class components grew very large over time
- Lifecycle methods mixed unrelated logic
- Difficult to read, test, and maintain

Solution with Hooks

- Hooks encourage smaller, cleaner components
- Logic is split by concern, not lifecycle method

4. Poor Code Reusability Patterns

Problem

- HOCs and render props:
    - Added extra nesting
    - Made component trees hard to debug
    - Reduced readability

Solution with Hooks

- Custom Hooks reuse logic without changing component structure
- No additional components in the tree

5. Inconsistent Functional vs Class Components

Problem

- Functional components were “stateless”
- Developers had to switch between:
    - Functions (UI)
    - Classes (logic)

Solution with Hooks

- One consistent way to write components
- Functional components can now:
    - Handle state
    - Use lifecycle behavior
    - Access context

6. Harder Testing and Refactoring

Problem

- Lifecycle-heavy class components were difficult to test
- Refactoring often broke behavior

Solution with Hooks

- Hooks are plain JavaScript functions
- Easier to test, refactor, and reason about

Why are Hooks considered an important addition to React ?

1. Simpler and Cleaner Code

- Less boilerplate
- No classes
- No this

2. Better Logic Reuse

- Custom hooks enable reusable, composable logic

3. Improved Readability

- Related logic stays together
- Easier for teams to understand and maintain

4. Future-Friendly React

- Hooks align with React's direction
- Most new features and examples are hook-based

5. Performance and Optimization

- Hooks like useMemo, useCallback, and useRef help optimize rendering

---

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

Ans.

useReducer is a React Hook used for state management in functional components. It is an alternative to useState, best suited for complex state logic or when multiple state values depend on each other. It works on the same concept as Redux reducers.

When should you use useReducer?

Use useReducer when:

- State logic is complex
- Multiple state updates depend on previous state
- You want predictable state transitions
- State is updated in many places

For simple state → useState
For complex state → useReducer

Basic Syntax :

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

Key Parts

- state → current state
- dispatch → function to trigger updates
- reducer → function that describes how state changes
- initialState → starting state

How useReducer Works :

1. 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;
  }
}
```

2. Using useReducer in a Component

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}

export default function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
```

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

        <button onClick={() => dispatch({ type: 'INCREMENT' })}>
          Increase
        </button>

        <button onClick={() => dispatch({ type: 'DECREMENT' })}>
          Decrease
        </button>
      </div>
    );
  }
```

Example: Form State with useReducer :-

```
const initialState = {
  name: '',
  email: ''
};

function reducer(state, action) {
  return {
    ...state,
    [action.field]: action.value
  };
}
```

```
function Form() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <form>
      <input
        value={state.name}
        onChange={(e) =>
          dispatch({ field: 'name', value: e.target.value })
        }
      />

      <input
        value={state.email}
        onChange={(e) =>
          dispatch({ field: 'email', value: e.target.value })
        }
      />
    </form>
  );
}
```

useReducer vs useState:

Feature	useState	useReducer
Simple state	Yes	No
Complex logic	No	Yes
Multiple related states	No	Yes
Predictable updates	No	Yes
Redux-like pattern	No	Yes

Advantages of useReducer

- Better state organization
- Easier to debug
- Cleaner logic for complex updates
- More predictable state transitions

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

Ans.

Both useCallback and useMemo are performance optimization hooks in React. They help prevent unnecessary re-renders and expensive recalculations in functional components.

1. useCallback Hook

Purpose

useCallback is used to memoize functions, so the same function reference is reused between renders unless its dependencies change.

In React, functions are re-created on every render.
This can cause unnecessary re-renders of child components.

Syntax

```
const memoizedCallback = useCallback(() => {
  // function logic
}, [dependencies]);
```

Example (Without useCallback) :

```
function Parent() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    console.log("Clicked");
  };

  return <Child onClick={handleClick} />;
}
```

Example (With useCallback) :

```
const handleClick = useCallback(() => {
  console.log("Clicked");
}, []);
```

When to use useCallback

- Passing functions as props to memoized child components
- Preventing re-renders caused by function recreation
- Optimizing large component trees

2. useMemo Hook

Purpose

useMemo is used to memoize computed values, so expensive calculations are only re-run when dependencies change.

Syntax :

```
      const memoizedValue = useMemo(() => {
  return expensiveCalculation();
}, [dependencies]);
```

Example:

```
const expensiveValue = useMemo(() => {
  return items.reduce((total, item) => total + item.price, 0);
}, [items]);
```

When to use useMemo

- Heavy computations (loops, filters, sorting)
- Preventing recalculations on every render
- Optimizing performance-sensitive components

useCallback vs useMemo :

Hook	Memoizes	Returns
useCallback	Function	Function
useMemo	Value	Value

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

Ans.

Both useCallback and useMemo are React performance optimization hooks, but they serve different purposes.

Core Difference (Simple Explanation)

- useCallback → memoizes a function
- useMemo → memoizes a value (result of a calculation)

Syntax Comparison

useCallback :

```
const memoizedFunction = useCallback(() => {
  // function logic
}, [dependencies]);
```

useMemo :

```
const memoizedValue = useMemo(() => {
  return expensiveCalculation();
}, [dependencies]);
```

What They Return

Hook	**What it memoizes**	**What it returns**
useCallback	Function	Same function reference
useMemo	Value	Cached computed value

Example Side-by-Side :-

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

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

  // useCallback → memoized function
  const handleClick = useCallback(() => {
    console.log("Clicked");
  }, []);

  // useMemo → memoized value
  const doubledCount = useMemo(() => {
    return count * 2;
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Doubled: {doubledCount}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

Key Use Cases

When to use useCallback

- Passing callbacks to memoized child components
- Preventing unnecessary child re-renders
- Stabilizing function references

When to use useMemo

- Avoiding expensive calculations
- Optimizing derived data (filtering, sorting)
- Improving render performance

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

Ans.

useRef is a React Hook that lets you persist a mutable value across renders without causing a re-render.
It is commonly used to:

- Access DOM elements directly
- Store values that should not trigger re-renders

Syntax :

```
const ref = useRef(initialValue);
```

ref.current holds the value

Updating ref.current **does not re-render** the component

How useRef Works

- React creates a ref object { current: initialValue }
- The same ref object is preserved between renders
- Changing ref.current does not trigger re-render

1. Accessing DOM Elements (Most Common Use)

Example: Focus Input Field

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

function FocusInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
```

Why useRef here?

- Direct DOM access
- No need to re-render

2. Storing Previous Value

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

function PreviousValue() {
  const [count, setCount] = useState(0);
  const prevCount = useRef(0);

  useEffect(() => {
    prevCount.current = count;
  });

  return (
    <div>
      <p>Current: {count}</p>
      <p>Previous: {prevCount.current}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

3. Storing Mutable Values (Timers, Counters)

```
function Timer() {
 const timerId = useRef(null);

 const startTimer = () => {
  timerId.current = setInterval(() => {
   console.log("Running...");
  }, 1000);
 };

 const stopTimer = () => {
  clearInterval(timerId.current);
 };

 return (
  <div>
   <button onClick={startTimer}>Start</button>
   <button onClick={stopTimer}>Stop</button>
  </div>
 );
}
```

useRef vs useState :-

Feature	**useRef**	**useState**
Triggers re-render	No	Yes
Stores mutable value	Yes	No
Access DOM	Yes	No
Preserves value across renders	Yes	Yes

When to use useRef

- DOM manipulation (focus, scroll, media control)
- Storing previous values
- Holding timers or counters
- Avoiding unnecessary re-renders
