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

**✅ Rendering a list in React:**

To render a list of items, you typically:

1. Use **JavaScript's map() function**
2. Return a **JSX element** for each item

**Example:**

jsx

CopyEdit

function FruitsList() {

const fruits = ["Apple", "Banana", "Mango"];

return (

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li> // Key is important

))}

</ul>

);

}

**✅ Why is it important to use keys?**

* **Keys** help React identify **which items have changed, are added, or removed**.
* They improve **performance** and ensure correct **re-rendering** of list items.
* Without keys, React may **incorrectly update or reorder** elements.

🔔 **Best Practice:** Use a **unique and stable** value (like an ID) for the key.

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

**✅ What are keys?**

* **Keys** are special attributes used by React when **rendering lists**.
* Each element inside a list should have a unique key prop.

**Example with unique ID:**

jsx

CopyEdit

const users = [

{ id: 1, name: "Himani" },

{ id: 2, name: "Rahul" },

];

return (

<ul>

{users.map(user => (

<li key={user.id}>{user.name}</li>

))}

</ul>

);

**❌ What happens if you don't use unique keys?**

* React will **re-render all items** unnecessarily, reducing performance.
* It may **mismatch components**, causing **bugs** in dynamic UIs (e.g., input losing focus).
* You may see a **warning** in the console:  
  "Warning: Each child in a list should have a unique "key" prop."

**✅ Summary:**

| **Feature** | **Description** |
| --- | --- |
| **map()** | Used to loop over arrays and return JSX elements |
| **keys** | Unique identifiers for each item in a list |
| **Best key** | Use unique IDs from data, avoid using index unless the list never changes |
| **Without key** | Poor performance and potential UI bugs |

**Question 1: How do you handle forms in React? Explain the concept of controlled components.**

**✅ Handling Forms in React:**

React forms are typically handled using **state** to control the values of input elements.

Steps:

1. Define **state** for each form field.
2. Add an **onChange handler** to update the state.
3. Use an **onSubmit handler** to process the form data.

**✅ What are Controlled Components?**

* A **controlled component** is an input element (like <input>, <textarea>, etc.) whose value is **controlled by React state**.
* The form field's **value is tied to state**, and any change is handled by React.

**Example:**

jsx

CopyEdit

import React, { useState } from 'react';

function MyForm() {

const [name, setName] = useState("");

const handleChange = (e) => {

setName(e.target.value); // Update state

};

const handleSubmit = (e) => {

e.preventDefault(); // Prevent page reload

alert(`Submitted name: ${name}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" value={name} onChange={handleChange} />

<button type="submit">Submit</button>

</form>

);

}

🔹 Here, the input value is **controlled** by name state, not directly by the DOM.

**Question 2: What is the difference between controlled and uncontrolled components in React?**

| **Feature** | **Controlled Component** | **Uncontrolled Component** |
| --- | --- | --- |
| **Definition** | React manages the input’s state | DOM manages the input’s value |
| **State** | Uses useState or this.state | Uses ref to access value directly |
| **Data Source** | React state | DOM (value is stored in the actual input element) |
| **Use case** | Recommended for most React forms | Used when React state management isn't needed |
| **Validation and Logic** | Easy to apply logic, validation, conditionals | Harder to manage logic or track changes |

**✅ Example of Uncontrolled Component:**

jsx

CopyEdit

import React, { useRef } from 'react';

function UncontrolledForm() {

const inputRef = useRef();

const handleSubmit = (e) => {

e.preventDefault();

alert(`Entered: ${inputRef.current.value}`);

};

return (

<form onSubmit={handleSubmit}>

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

<button type="submit">Submit</button>

</form>

);

}

**✅ Summary:**

| **Concept** | **Controlled** | **Uncontrolled** |
| --- | --- | --- |
| Manages value? | React | DOM |
| Uses state? | Yes | No |
| Uses ref? | Rarely | Yes |
| Preferred? | ✅ Yes (more consistent) | ❌ Only for special cases |

**Question 1: What are lifecycle methods in React class components? Describe the phases of a component’s lifecycle.**

**✅ What are Lifecycle Methods?**

Lifecycle methods are **special methods** in class components that allow you to run **code at specific points** in a component's lifecycle—like when it's created, updated, or destroyed.

**✅ Phases of a Component’s Lifecycle:**

1. **🟢 Mounting (component is being added to the DOM):**
   * constructor()
   * static getDerivedStateFromProps()
   * render()
   * componentDidMount()
2. **🟡 Updating (component is being re-rendered due to state/props changes):**
   * static getDerivedStateFromProps()
   * shouldComponentUpdate()
   * render()
   * getSnapshotBeforeUpdate()
   * componentDidUpdate()
3. **🔴 Unmounting (component is being removed from the DOM):**
   * componentWillUnmount()
4. **⚪ Error Handling (optional phase for catching errors):**
   * componentDidCatch()
   * getDerivedStateFromError()

⚠️ Note: These lifecycle methods only work in **class components**, not in functional components. Functional components use **Hooks** (e.g., useEffect) instead.

**Question 2: Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().**

**✅ 1. componentDidMount()**

* Called **once** right after the component is **mounted** to the DOM.
* Used for:
  + Fetching API data
  + Setting up subscriptions
  + Initial DOM interactions

jsx

CopyEdit

componentDidMount() {

console.log("Component mounted!");

// fetch data here

}

**✅ 2. componentDidUpdate(prevProps, prevState)**

* Called **after every update** (state or props change), except during the initial mount.
* Used for:
  + Responding to prop/state changes
  + Triggering side effects or re-fetching data conditionally

jsx

CopyEdit

componentDidUpdate(prevProps, prevState) {

if (this.state.count !== prevState.count) {

console.log("Count changed!");

}

}

**✅ 3. componentWillUnmount()**

* Called **just before** the component is **removed** from the DOM.
* Used for:
  + Cleanup tasks like removing event listeners or timers
  + Cancelling subscriptions or network requests

jsx

CopyEdit

componentWillUnmount() {

console.log("Component is being unmounted");

clearInterval(this.timer);

}

**✅ Summary Table:**

| **Method** | **Phase** | **Purpose** |
| --- | --- | --- |
| componentDidMount() | Mounting | Run code after first render (e.g., fetch data) |
| componentDidUpdate() | Updating | Respond to changes in props/state |
| componentWillUnmount() | Unmounting | Clean up before the component is removed |

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

**✅ What are Hooks?**

Hooks are special functions in React that allow you to **use state and lifecycle features in functional components** — features that were previously available only in class components.

**✅ useState()**

* Allows you to **create and manage state** in a functional component.
* Returns a state value and a function to update it.

jsx

CopyEdit

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // state variable and updater

return (

<div>

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

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

</div>

);

}

**✅ useEffect()**

* Allows you to **perform side effects** (like fetching data, updating DOM, or setting up subscriptions).
* It replaces lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

jsx

CopyEdit

import { useEffect } from 'react';

function Example() {

useEffect(() => {

console.log("Component mounted or updated");

return () => {

console.log("Cleanup on unmount");

};

}, []); // Empty array means run only once (like componentDidMount)

}

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

**✅ Problems Hooks Solved:**

* Reusing stateful logic was hard — required HOCs or render props.
* Class components were complex to manage (binding this, lifecycle confusion).
* Separation of concerns was poor — related logic split across lifecycle methods.

**✅ Why Hooks Are Important:**

* Allow **functional components to use state and side effects**.
* Make code **simpler, more readable, and reusable**.
* Promote **better separation of concerns** by grouping related logic.

**Question 3: What is useReducer? How is it used in a React app?**

**✅ useReducer is a hook used for complex state management, especially when state updates depend on the previous state.**

It's an alternative to useState, similar to how Redux works.

jsx

CopyEdit

import { 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;

}

}

function Counter() {

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

return (

<>

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

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

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

</>

);

}

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

| **Hook** | **Purpose** |
| --- | --- |
| useCallback | Returns a **memoized version of a function**, preventing re-creation on every render. |
| useMemo | Returns a **memoized computed value**, used to avoid expensive calculations on every render. |

**Question 5: What’s the difference between useCallback and useMemo?**

| **Feature** | **useCallback** | **useMemo** |
| --- | --- | --- |
| Returns | A **function** | A **value/result** |
| Use Case | When you want to memoize a function | When you want to memoize a computed value |
| Syntax | useCallback(fn, deps) | useMemo(() => compute, deps) |

**Example useCallback:**

jsx

CopyEdit

const handleClick = useCallback(() => {

console.log('Clicked');

}, []);

**Example useMemo:**

jsx

CopyEdit

const expensiveResult = useMemo(() => computeExpensiveValue(data), [data]);

**Question 6: What is useRef? How does it work in a React app?**

**✅ useRef is a hook that:**

* Provides a way to **store mutable values** that don’t cause a re-render when changed.
* Can also be used to **reference DOM elements directly**.

**✅ Example 1: Accessing DOM**

jsx

CopyEdit

import { useRef, useEffect } from 'react';

function InputFocus() {

const inputRef = useRef();

useEffect(() => {

inputRef.current.focus(); // Focus the input on mount

}, []);

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

}

**✅ Example 2: Store mutable value**

jsx

CopyEdit

const countRef = useRef(0);

countRef.current += 1; // Does not trigger a re-render