**Lists And Hooks**

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

You can render a list using the .map() function in React.

**🧾 Example:**

jsx

CopyEdit

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

return (

<ul>

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

<li key={index}>{fruit}</li>

))}

</ul>

);

* fruits.map(...) loops through the array.
* Each item is rendered inside a <li>.
* The key prop is passed for each list item (more on that below).

**🔑 2. Why is it Important to Use Keys When Rendering Lists?**

**✅ Reason:**

Keys help **React identify which items have changed**, been added, or removed.

React uses keys to:

* **Optimize rendering performance**
* Avoid unnecessary re-renders
* Correctly track the identity of each element

**🔴 What Happens Without Keys:**

* React may **re-render all list items** unnecessarily
* Or **mix up elements**, especially when adding/removing items dynamically

**✅ Best Practice:**

* Use a **unique and stable key** (like an ID from database or API).
* Avoid using index as a key **unless** you have no unique value (e.g., static list).

• Question 2: What are keysin React, and what happens if you do not provide a unique key?

**🔑 What Are Keys in React?**

**Keys** are special attributes you pass to elements inside a list (like when using .map()), so React can **identify** each item uniquely.

jsx

CopyEdit

const users = [{ id: 1, name: "Amit" }, { id: 2, name: "Neha" }];

return (

<ul>

{users.map(user => (

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

))}

</ul>

);

**❓ Why Are Keys Important?**

React uses keys to:

* Track **which items changed, added, or removed**
* **Minimize re-renders** for better performance
* Maintain **correct element associations** in the DOM

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

**❌ React throws a warning:**

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

**❌ Possible issues:**

* React may re-render elements **incorrectly**
* Input fields may **lose focus**
* Animations or transitions might **misbehave**
* The UI can **break in dynamic lists** (especially when adding/removing items)

**❗ Bad Example (using index as key):**

jsx

CopyEdit

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

<li key={index}>{item}</li> // ❌ Not ideal when list changes

))}

🔸 Index-based keys work only if:

* List is **static**
* Items **don’t move or change order**

**✅ Good Example (using unique ID):**

jsx

CopyEdit

{items.map(item => (

<li key={item.id}>{item.name}</li> // ✅ Recommended

))}

LAB EXERCISE

1)Create a React component that renders a list of items (e.g., a list of fruit names). Use the map() function to render each item in the list.

import React from "react";

export default function FruitList() {

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

return (

<div>

<h2>Fruit List</h2>

<ul>

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

<li key={index}>{fruit}</li> // Using index as key (acceptable here as it's a static list)

))}

</ul>

</div>

);

}

2)Create a list of users where each user has a unique id. Render the user list using React and assign a unique key to each user.

import React from "react";

export default function UserList() {

const users = [

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

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

{ id: 3, name: "Raj" },

{ id: 4, name: "Priya" }

];

return (

<div>

<h2>User List</h2>

<ul>

{users.map(user => (

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

))}

</ul>

</div>

);

}

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

**React Hooks** are special functions that let you **use React features like state and lifecycle methods** inside **functional components** (without using class components).

Hooks were introduced in **React 16.8**.

**🛠️ Why Hooks?**

Before hooks, only **class components** could have:

* State
* Lifecycle methods (like componentDidMount, componentDidUpdate)

With Hooks, **functional components** can now:  
✅ Store and update state  
✅ Run side-effects (API calls, timers, DOM updates)  
✅ Handle lifecycle logic

**🎣 Common Hooks:**

| **Hook** | **Purpose** |
| --- | --- |
| useState() | Add **state** in functional components |
| useEffect() | Handle **side-effects** like API calls, timers |
| useRef() | Get reference to DOM elements or values |
| useContext() | Access React Context data |

**🔹 How useState() Works**

jsx

CopyEdit

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

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

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

</div>

);

}

**🧠 What’s Happening:**

* useState(0) → initializes count with 0
* setCount → used to update the value of count
* Component re-renders whenever count changes

**🔹 How useEffect() Works**

jsx

CopyEdit

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

function Example() {

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

useEffect(() => {

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

return () => {

console.log("Cleanup if needed");

};

}, [name]); // 🔁 only run when `name` changes

return <input value={name} onChange={(e) => setName(e.target.value)} />;

}

**🧠 What’s Happening:**

* useEffect() runs **after render**.
* Useful for: API calls, event listeners, timers, etc.
* Dependency array [name] ensures it runs only when name changes.
* Cleanup function runs when component unmounts or before next effect.

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

**🔧 Problems Hooks Solved**

**🔹 1. Code Reuse Between Components Was Difficult**

* Earlier, developers had to use:
  + **HOCs (Higher-Order Components)** or
  + **Render Props**
* These patterns often made code **hard to read**, **deeply nested**, and **difficult to manage**.

✅ **Hooks like useState, useEffect, and custom hooks** made logic reusable in a clean and modular way.

**🔹 2. Too Much Complexity in Class Components**

* Class components required:
  + constructor, this.state, this.setState
  + Binding methods with this
* Beginners often struggled with **this keyword** and lifecycle methods.

✅ **Hooks allow writing simpler, cleaner functional components** — no classes, no this.

**🔹 3. Split Logic Across Lifecycle Methods**

* In class components, related logic often had to be split:

js

CopyEdit

componentDidMount() {

// Fetch data

}

componentDidUpdate() {

// Update logic

}

componentWillUnmount() {

// Cleanup

}

✅ With useEffect(), you can handle **setup and cleanup in one place**.

**🔹 4. State Logic Not Shareable Easily**

* You couldn’t easily share stateful logic across components (e.g., multiple components fetching data).

✅ With **custom hooks**, you can extract stateful logic into reusable functions.

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

**🧠 What is useReducer in React?**

useReducer is a React Hook used for **managing complex state logic** in functional components.

It's an alternative to useState, and is especially useful when:

* You have **multiple related state variables**
* You need **more control over how state changes**
* You want to follow a **Redux-style** state pattern in a component

**🔁 How useReducer Works:**

It takes **two main arguments**:

js

CopyEdit

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

**📌 Parts:**

| **Term** | **Meaning** |
| --- | --- |
| state | Current state value |
| dispatch | Function to send actions |
| reducer | A function that updates the state based on action |
| initialState | Initial value of the state |

**🔧 Example: Counter with useReducer**

jsx

CopyEdit

import React, { useReducer } from "react";

// Step 1: Define 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;

}

}

// Step 2: Create component using useReducer

export default function Counter() {

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>

</div>

);

}

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

**🔁 useCallback — Caches a Function**

**✅ Purpose:**

To **prevent a function from being recreated** every time a component renders — unless its dependencies change.

**🔧 Use Case:**

When you pass a function as a prop to child components, and you want to **avoid unnecessary re-renders** of the child.

**🧠 Example:**

jsx

CopyEdit

const handleClick = useCallback(() => {

console.log("Clicked");

}, []);

This function is **memoized** and will stay the same across renders (unless dependencies change).

**🔢 useMemo — Caches a Computed Value**

**✅ Purpose:**

To **avoid expensive calculations** every time the component renders.

**🔧 Use Case:**

When you have a **heavy calculation** and you want to **recalculate only when needed**.

**🧠 Example:**

jsx

CopyEdit

const expensiveValue = useMemo(() => {

return heavyCalculation(input);

}, [input]);

**🔍 🔁 Summary of Differences:**

| **Hook** | **Caches What?** | **Use When?** |
| --- | --- | --- |
| useCallback | A **function** | Avoid re-creating functions unnecessarily |
| useMemo | A **value (computed)** | Avoid re-running expensive calculations |

**✅ Real World Analogy:**

* **useCallback** = "Yeh function bar-bar mat banao, jab tak zarurat na ho."
* **useMemo** = "Yeh calculation bar-bar mat karo, jab tak input na badle."

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

**🧠 Short Summary:**

| **Hook** | **Caches a...** | **Returns...** | **Used For** |
| --- | --- | --- | --- |
| useCallback | **Function** | The **function** itself | To avoid re-creating functions |
| useMemo | **Computed value** | The **result of a function** | To avoid re-running expensive calculations |

**🔁 useCallback**

* **Purpose**: Memorize a function so it’s not re-created on every render.
* **Syntax**:

js

CopyEdit

const memoizedFn = useCallback(() => {

// some function logic

}, [dependencies]);

* **Use Case**: When you pass a function as a **prop to child components** and want to prevent unnecessary re-renders.

**🧮 useMemo**

* **Purpose**: Memorize the **result of a function** (i.e., a computed value).
* **Syntax**:

js

CopyEdit

const result = useMemo(() => {

return expensiveCalculation(input);

}, [input]);

* **Use Case**: When you want to avoid running a **heavy calculation** again unless needed.

**🧾 Real Example Comparing Both:**

jsx

CopyEdit

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

function Example() {

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

const [dark, setDark] = useState(false);

// useMemo for expensive calculation

const expensiveValue = useMemo(() => {

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

return count \* 2;

}, [count]);

// useCallback to memoize function

const toggleTheme = useCallback(() => {

setDark(prevDark => !prevDark);

}, []);

const themeStyles = {

backgroundColor: dark ? "#333" : "#FFF",

color: dark ? "#FFF" : "#000",

padding: "10px"

};

return (

<div style={themeStyles}>

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

<p>Double: {expensiveValue}</p>

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

<button onClick={toggleTheme}>Toggle Theme</button>

</div>

);

}

**🧠 Final Takeaway:**

| **Situation** | **Use This Hook** |
| --- | --- |
| Avoid re-creating functions | useCallback |
| Avoid re-running expensive calculations | useMemo |

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

**🎣 What is useRef in React?**

useRef is a React hook that allows you to:

1. **Store a mutable value** that doesn't cause re-render when changed
2. **Directly access a DOM element**

**📌 Syntax:**

jsx

CopyEdit

const myRef = useRef(initialValue);

* myRef.current holds the actual value or DOM element.
* Changing myRef.current **does not cause re-render**.

**✅ How useRef Works in React App**

**🔹 Example 1: Focus an Input Field**

jsx

CopyEdit

import React, { useRef } from "react";

function FocusInput() {

const inputRef = useRef(null);

const focusInput = () => {

inputRef.current.focus(); // direct DOM access

};

return (

<div>

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

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

</div>

);

}

export default FocusInput;

🧠 Here:

* useRef(null) creates a reference.
* ref={inputRef} links it to the <input>.
* inputRef.current.focus() calls the native DOM focus method.

**🔹 Example 2: Store Previous State Value (without causing re-render)**

jsx

CopyEdit

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

function PreviousValue() {

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

const prevCount = useRef();

useEffect(() => {

prevCount.current = count; // store previous count

}, [count]);

return (

<div>

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

<p>Previous: {prevCount.current}</p>

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

</div>

);

}

🧠 Here:

* We store the **previous count value** without triggering re-render.

**💡 Common Use Cases of useRef:**

| **Use Case** | **Example** |
| --- | --- |
| Access DOM elements | Focus input, scroll to div, play video |
| Store mutable values without re-render | Previous value, timeout IDs |
| Integrate with third-party libraries | e.g., accessing canvas, charts |

**🧠 Summary:**

| **Feature** | **Description** |
| --- | --- |
| useRef() | Hook to reference DOM or store values |
| ref.current | Access actual DOM or value |
| Re-render? | ❌ Changing ref.current doesn't re-render |

LAB EXERCISE

• 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';

const Counter = () => {

// Step 1: Define the state for the counter, starting with 0

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

// Step 2: Create functions to handle increment and decrement

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

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

return (

<div>

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

{/\* Step 3: Create buttons to increment and decrement \*/}

<button onClick={increment}>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';

const DataFetcher = () => {

// Step 1: Define state variables to hold the data and loading state

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

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

// Step 2: Use useEffect to fetch data when the component mounts

useEffect(() => {

// Step 3: Fetch data from the API

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

.then((response) => response.json())

.then((data) => {

setData(data); // Set the fetched data to the state

setLoading(false); // Set loading to false after data is fetched

})

.catch((error) => {

console.error('Error fetching data:', error);

setLoading(false); // Set loading to false if there's an error

});

}, []); // Empty dependency array to run this effect only once (on mount)

// Step 4: Show loading text while the data is being fetched

if (loading) {

return <h1>Loading...</h1>;

}

// Step 5: Display the fetched data

return (

<div>

<h1>Fetched Data</h1>

<ul>

{data.map((item) => (

<li key={item.id}>{item.title}</li>

))}

</ul>

</div>

);

};

export default DataFetcher;

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

 **Install Redux and React-Redux**:

bash

CopyEdit

npm install @reduxjs/toolkit react-redux

 **Create a Redux slice** (counterSlice.js):

js

CopyEdit

import { createSlice } from '@reduxjs/toolkit';

// Step 1: Create a slice for counter

const counterSlice = createSlice({

name: 'counter',

initialState: { value: 0 },

reducers: {

increment: (state) => {

state.value += 1;

},

decrement: (state) => {

state.value -= 1;

},

},

});

// Export actions

export const { increment, decrement } = counterSlice.actions;

// Export the reducer

export default counterSlice.reducer;

 **Configure Redux store** (store.js):

js

CopyEdit

import { configureStore } from '@reduxjs/toolkit';

import counterReducer from './counterSlice';

// Step 2: Create Redux store and add the counter slice

const store = configureStore({

reducer: {

counter: counterReducer,

},

});

export default store;

 **Set up the Redux Provider** (index.js):

js

CopyEdit

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import App from './App';

import store from './store';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

 **Create the Counter Component** (App.js):

js

CopyEdit

import React from 'react';

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

import { increment, decrement } from './counterSlice';

const App = () => {

// Step 3: Use useSelector to read from Redux store

const count = useSelector((state) => state.counter.value);

// Step 4: Use useDispatch to dispatch actions

const dispatch = useDispatch();

return (

<div>

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

<button onClick={() => dispatch(increment())}>Increment</button>

<button onClick={() => dispatch(decrement())}>Decrement</button>

</div>

);

};

export default App;

• Task 4: Create react app to avoid re-rendersin react application by useRef ?

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

const App = () => {

// Step 1: Regular state for re-rendering (this will cause a re-render)

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

// Step 2: useRef to store a value without triggering re-renders

const countRef = useRef(0);

// Step 3: Increment the count and update both state and ref

const increment = () => {

setCount(count + 1); // This will trigger a re-render

countRef.current = count + 1; // This will not trigger a re-render

};

// Step 4: Display count from state (re-renders) and count from ref (no re-render)

return (

<div>

<h1>State Count: {count}</h1>

<h2>Ref Count (no re-render): {countRef.current}</h2>

<button onClick={increment}>Increment</button>

</div>

);

};

export default App;