**REACT**

React hooks

### ****Best Answer (in Urdu + English mix for understanding):****

**"Hooks React ke special functions hain jo functional components ko powerful banatay hain. In ki madad se hum state manage kar saktay hain, side effects handle kar saktay hain, context use kar saktay hain aur aur bhi React features ka use kar saktay hain—aur ye sab class components ke bina possible hota hai."**

### 🔍 Short English Version for Interview:

**"Hooks are special functions in React that let you use state and other React features in functional components without writing class components."**

## 🔄 1. useState – ****For Storing Data (State)****

### 🔹 What it does:

Stores a value in your component (like a counter, input, etc.)

### 🔹 Example:

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

* count → current value
* setCount → function to change the value

### 🔹 Real-life example:

Like a **calculator display** that changes when you press buttons.

## ⚙️ 2. useEffect – ****For Side Effects****

### 🔹 What it does:

Runs some code **when the component loads** or when **something changes**.

### 🔹 Example:

useEffect(() => {

console.log("Component loaded");

}, []);

* Empty array [] means run once when loaded.

### 🔹 Real-life example:

Like setting an **alarm** when you enter a room and it rings once.

## 🌐 3. useContext – ****For Sharing Data Globally****

### 🔹 What it does:

Allows components to access **shared data** (like user info, theme) **without props**.

### 🔹 Example:

const user = useContext(UserContext);

### 🔹 Real-life example:

Like a **public announcement** board where everyone sees the same info.

## 📌 4. useRef – ****For Storing Values or Accessing DOM****

### 🔹 What it does:

### Conclusion:

useRef is perfect for **temporary tasks, background values, or DOM access**, where **you don’t want to re-render the component**.

### 🔹 Example:

const inputRef = useRef();

<input ref={inputRef} />

### 🔹 Real-life example:

Like a **bookmark** in a book—you remember where you left off without changing the book.

## 🔁 5. useReducer – ****For Complex State Logic****

### 🔹 What it does:

Used when you have **multiple actions** that change state (like a mini-state machine).

### 🔹 Example:

const reducer = (state, action) => {

if (action.type === "add") return state + 1;

return state;

};

const [count, dispatch] = useReducer(reducer, 0);

### 🔹 Real-life example:

Like using a **remote control** (dispatch) to tell your TV (state) what to do.

## 🚀 6. useMemo – ****For Avoiding Unnecessary Calculations****

### 🔹 What it does:

**Saves the result** of a calculation so React doesn’t redo it every time.

### 🔹 Example:

const result = useMemo(() => heavyCalculation(num), [num]);

### 🔹 Real-life example:

Like **saving your math homework result** so you don’t recalculate every time.

## 🧠 7. useCallback – ****For Avoiding Unnecessary Function Re-Creation****

### 🔹 What it does:

**Saves a function**, so it’s not recreated on every re-render.

### 🔹 Example:

const handleClick = useCallback(() => {

console.log("Clicked");

}, []);

### 🔹 Real-life example:

Like **saving your friend's contact number** so you don’t type it every time.

## 🛠️ 8. ****Custom Hooks**** – ****Your Own Reusable Logic****

### 🔹 What it does:

You can **create your own hook** using existing hooks.

### 🔹 Example:

function useCounter() {

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

return [count, () => setCount(count + 1)];

}

### 🔹 Real-life example:

Like creating your **own cooking recipe** you can reuse anytime.

## ✅ Final Summary Table:

| **Hook** | **What It Does** | **Real-Life Analogy** |
| --- | --- | --- |
| useState | Store and update value | Calculator display |
| useEffect | Run code on load or update | Setting an alarm |
| useContext | Share global data across components | Public notice board |
| useRef | Save value or access DOM | Bookmark in a book |
| useReducer | Manage complex state with actions | TV remote control |
| useMemo | Save result of a heavy calculation | Saved homework result |
| useCallback | Save function to prevent unnecessary re-creation | Saved phone number |
| Custom Hooks | Reusable logic made by combining hooks | Your own recipe |

Would you like small code examples of all these together in a single demo project?

## 🧠 1. useState() – How It Works

### 🔧 What it does again:

It **stores data** in a component and **re-renders** the component when that data changes.

### 🛠️ How it works:

* When you call useState(0), React creates a "box" to store that value (like 0)
* It gives you:
  + the **current value** (e.g., count)
  + a **function** to update it (e.g., setCount)
* When you call setCount(newValue), React:
  + updates the box
  + and **re-renders the component** to show the new value

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

🧠 Think of it like a **whiteboard**: you write a number (count) and use a marker (setCount) to update it, and React **refreshes** the screen to show the new number.

## 🔁 2. useEffect() – How It Works

### 🔧 What it does:

It lets you **run code** at specific times in a component’s life.

### 🛠️ How it works:

React runs the function inside useEffect():

* after the **first render** (component mounted)
* and/or after **state/prop changes** (based on dependency array)

useEffect(() => {

console.log("I run after render");

}, [count]); // runs every time `count` changes

🧠 Imagine setting up **alarms**:

* No dependencies ([]) → run only once after mount
* With [count] → run whenever count changes

## 🌍 3. useContext() – How It Works

### 🔧 What it does:

Gives components access to **global/shared data** without passing props manually.

### 🛠️ How it works:

* You first create a context with React.createContext()
* Then, you **wrap** your components in a **Provider**
* Inside any child, use useContext(ContextName) to **read the value**

const UserContext = React.createContext();

const user = useContext(UserContext);

🧠 Think of it like a **global TV channel**: instead of passing a message to every child manually, they can all **tune in to the same context**.

## 🚀 6. useMemo() – How It Works

### 🔧 What it does:

Remembers the **result** of a calculation so it doesn’t repeat unnecessarily.

### 🛠️ How it works:

* You pass it a **function** and a list of **dependencies**
* If the dependencies **haven’t changed**, it gives you the **cached result**

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

🧠 Like **saving your math exam answer**: no need to solve it again if the numbers haven’t changed.

## 🧩 7. useCallback() – How It Works

### 🔧 What it does:

Remembers a **function** so React doesn’t recreate it every time.

### 🛠️ How it works:

* Like useMemo, but for **functions**
* Prevents child components from re-rendering unnecessarily

const handleClick = useCallback(() => doSomething(), []);

🧠 Think of it like **saving your boss's phone number** — you don’t rewrite it every time you want to call.

## 🛠️ 8. Custom Hooks – How They Work

### 🔧 What they are:

Functions you write using other hooks, to **reuse logic**.

### 🛠️ How it works:

* It’s just a function that **starts with use**
* Inside, you can call useState, useEffect, etc.
* You can use this hook in multiple components

function useCounter() {

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

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

return [count, increment];

}

🧠 Like writing your own **mini-toolbox** that you can use anywhere in your app.

## 🔚 Summary of "How They Work"

| **Hook** | **What Happens Internally** |
| --- | --- |
| useState | React saves a value and re-renders on update |
| useEffect | React runs your code after rendering, based on conditions |
| useContext | React gives you a shared value from a Provider |
| useRef | React gives you a stable object to store data without rerenders |
| useReducer | React uses a function to update state based on actions |
| useMemo | React saves the result of a function to avoid recalculating |
| useCallback | React saves a function to avoid re-creating it on each render |
| Custom Hooks | You combine built-in hooks to make reusable logic |

## 🔴 useReducer — For More Organized State Management

### ✅ ****Definition:****

useReducer is a React Hook used for **managing complex or multiple related state changes** in a clean and organized way.

### 🧠 ****Why/Need:****

Sometimes useState becomes **messy** when:

* You have multiple values (like count, name, age)
* You want to do different actions like increase, decrease, reset, etc.

Instead of writing 3 useStates or 3 setState() calls, you use one useReducer to **handle everything in one place**.

### 🛠️ ****How it works (Easy Terms):****

* You make a reducer() function that handles different actions.
* You send those actions using dispatch().
* React calls the reducer, calculates the new state, and updates it.

📦 **Think of it like a manager with a walkie-talkie**:

* You say: "increase count"
* The manager (reducer function) decides what to do
* The final result is updated

### 🧾 ****Example: Counter with**** useReducer

import React, { useReducer } from "react";

// Step 1: Reducer function

function reducer(state, action) {

switch (action.type) {

case "increment":

return { count: state.count + 1 };

case "decrement":

return { count: state.count - 1 };

case "reset":

return { count: 0 };

default:

return state;

}

}

// Step 2: Use it in a component

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>

<button onClick={() => dispatch({ type: "reset" })}>Reset</button>

</div>

);

}

## Summary of useReducer:

| **Concept** | **Simple Explanation** |
| --- | --- |
| Reducer | A function that handles logic |
| Dispatch | You send an action (like "add" or "reset") |
| Cleaner code | Great for multiple or grouped state changes |

## 🔵 useRef — To Save Something Without Re-render

### ✅ ****Definition:****

useRef is a hook that lets you **store a value that does not trigger re-render when it changes**.

### 🧠 ****Why/Need:****

We use useRef when:

* You want to get or change a DOM element (like focus on an input)
* You want to save data (like previous value) **without showing it on the screen**
* You want to avoid re-rendering just for storing a value

### 🛠️ ****How it works (Easy Terms):****

* useRef() gives you a box: { current: null }
* You can change .current anytime, and React will **not re-render**
* It stays the same across renders

🧠 **Think of it like a sticky note in your notebook**:

* You can write something on it (like a number)
* It stays there forever
* But it doesn’t change your notebook’s content

### 🧾 ****Example 1: Focus an input****

import React, { useRef } from "react";

function FocusInput() {

const inputRef = useRef();

const handleClick = () => {

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

};

return (

<div>

<input ref={inputRef} placeholder="Type here..." />

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

</div>

);

}

### 🧾 ****Example 2: Track Previous Count Without Re-render****

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

function PreviousCount() {

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

const prevCountRef = useRef();

useEffect(() => {

prevCountRef.current = count; // Save current count after render

});

return (

<div>

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

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

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

</div>

);

}

## Summary of useRef:

| **Purpose** | **Real-world Example** |
| --- | --- |
| Access DOM | Focus input, scroll to div |
| Store values | Previous data or counters |
| No re-render | Won’t slow down UI when value changes |

## 🎯 Final Tips

| **Hook** | **Use When You Want To...** |
| --- | --- |
| useReducer | Handle many actions or complex state logic |
| useRef | Store data or access elements without rerender |

## 🟢 ****Synchronous (Sync) Programming****

### ✅ ****Definition:****

Synchronous programming means that tasks are executed **one after the other**. Each task **waits** for the previous one to finish before it starts.

### 🧠 ****How It Works:****

* Think of it like **a queue**: The tasks are lined up, and each one waits for the previous one to complete before starting.
* If a task takes time (like downloading a file), everything **else stops** until that task is completed.

### 🔧 ****Example:****

function task1() {

console.log("Task 1 started");

// Simulating some time-consuming operation

for (let i = 0; i < 1000000000; i++) {} // Just to slow it down

console.log("Task 1 finished");

}

function task2() {

console.log("Task 2 started");

console.log("Task 2 finished");

}

// Executing tasks synchronously

task1(); // Task 1 runs first

task2(); // Task 2 will only run after Task 1 finishes

### 🔑 Key Points:

* **Blocking**: Later tasks have to wait for earlier ones to finish.
* **Simple**: Easy to understand and follow.
* **Example**: Cooking a meal — you can’t start cooking the second dish until the first dish is finished.

## 🔴 ****Asynchronous (Async) Programming****

### ✅ ****Definition:****

Asynchronous programming allows tasks to run **independently**. This means tasks can be started and completed **without waiting** for others to finish. **Non-blocking**.

### 🧠 ****How It Works:****

* Think of it like **multi-tasking**: You can do one task (like sending an email), and while waiting for a reply, you can start another task (like making coffee).
* Tasks that take time (like loading data from a server) **don’t block the execution** of other tasks.

### 🔧 ****Example:****

function task1() {

console.log("Task 1 started");

setTimeout(() => { // Simulating delay

console.log("Task 1 finished");

}, 2000); // Delays for 2 seconds

}

function task2() {

console.log("Task 2 started");

console.log("Task 2 finished");

}

// Executing tasks asynchronously

task1(); // Task 1 runs but doesn't block Task 2

task2(); // Task 2 runs immediately, even though Task 1 is still running

In this example, task1 has a delay of 2 seconds, but task2 runs right away, **without waiting** for task1 to finish.

### 🔑 ****Key Points:****

* **Non-blocking**: Tasks don't stop the flow; they can run concurrently.
* **More complex**: Requires more advanced control (like callbacks, promises, async/await).
* **Example**: You can start multiple things at once, like downloading files while browsing the internet.

## 🎯 ****Summary Table:****

| **Aspect** | **Synchronous (Sync)** | **Asynchronous (Async)** |
| --- | --- | --- |
| **Execution Order** | Executes **one after the other** | Executes **independently**; can run in parallel |
| **Blocking** | Later tasks **wait** for earlier ones | Tasks **don’t wait** for each other |
| **Time-consuming Tasks** | Blocks execution until complete | Can run in the background (non-blocking) |
| **Control** | Simple, straightforward | Requires additional tools like callbacks, promises, or async/await |
| **Example** | Cooking one dish at a time | Cooking multiple dishes at the same time |

### 🧑‍🍳 ****Real-Life Analogy:****

* **Synchronous**: Imagine cooking dinner — first, you chop the veggies, then cook them, then prepare the salad, all **one at a time**.
* **Asynchronous**: Imagine cooking dinner while also making dessert and setting the table. You can **do many things at once**, without having to wait for each task to finish.

## 💡 ****When To Use Each?****

* **Synchronous**: Use when tasks depend on each other or need to be done in a specific order (e.g., calculations, simple functions).
* **Asynchronous**: Use when tasks can happen **independently** and you don’t want to **block** other operations (e.g., network requests, file reading).

**Restful Api:**

A **RESTful API** is a way for a **client** to communicate with a **server** using HTTP methods like **GET**, **POST**, **PUT**, and **DELETE** to **manage resources** (data). It follows simple, consistent rules, and is **stateless**, meaning each request contains all the information needed to process it.

**React Topics Name**

### ✅ ****Basic Topics****

1. JSX
2. Components (Functional & Class)
3. Props
4. State
5. Event Handling
6. Conditional Rendering
7. Lists and Keys
8. Forms
9. Lifting State Up

### ✅ ****Hooks****

1. useState
2. useEffect
3. useContext
4. useRef
5. useReducer
6. useCallback
7. useMemo
8. useLayoutEffect

### ✅ ****Advanced Topics****

1. Context API
2. Custom Hooks
3. HOC (Higher-Order Components)
4. Render Props
5. Memoization (React.memo)
6. Reconciliation
7. Virtual DOM

### ✅ ****Routing****

1. React Router
2. Route Parameters
3. Nested Routes
4. Redirects & Navigation
5. useNavigate, useParams, etc.

### ✅ ****Performance****

1. Code Splitting
2. Lazy Loading
3. Optimizing Renders

### ✅ ****Testing****

1. Unit Testing (Jest)
2. Component Testing (React Testing Library)

### ✅ ****State Management****

1. Redux
2. Redux Toolkit
3. Zustand
4. Recoil

### ✅ ****Deployment****

1. Create React App
2. Vite
3. Hosting (Netlify, Vercel, Firebase, etc.)

✅ Basic Topics

JSX JSX stands for JavaScript XML. It lets you write HTML in React code.

const element =

# Hello World

; Components (Functional & Class) Components are reusable blocks of UI. Functional components use functions; class components use classes.

function Welcome() { return

# Hello

; } Props Props are like function arguments. You pass data to components using props.

State State is used to store data in a component. When state changes, the UI updates.

const [count, setCount] = useState(0); Event Handling React handles events like onClick, onChange using camelCase.

Click Me Conditional Rendering Show/hide parts of UI based on condition.

{isLoggedIn ? : } Lists and Keys Used to render a list of items. Keys help React identify items.

{items.map(item =>

 {item.name})} Forms React handles form inputs using useState to manage input values.

Lifting State Up Move state to a common parent when multiple components need to share it.

✅ Hooks

useState Adds state to functional components.

useEffect Performs side effects like API calls or setting up timers.

useContext Access global data like user or theme without passing props everywhere.

useRef Access or store a value without causing re-renders (like accessing input DOM).

useReducer Used for complex state logic. Works like Redux.

useCallback Saves a function so it doesn’t get recreated on every render.

useMemo Saves a calculated value to avoid recalculating it unnecessarily.

useLayoutEffect Like useEffect, but runs before the screen updates.

✅ Advanced Topics

Context API Provides and consumes global data (like theme, user, language) without prop drilling.

Custom Hooks Create your own reusable logic with hooks.

HOC (Higher-Order Components) A function that takes a component and returns a new one with extra features.

Render Props Share logic using a function passed as a prop.

Memoization (React.memo) Prevents unnecessary re-renders of components.

Reconciliation React's process of comparing virtual DOM and real DOM to apply changes efficiently.

Virtual DOM A virtual copy of the real DOM used to make updates faster.

✅ Routing

React Router Used for adding navigation and URLs in your app.

Route Parameters Dynamic parts in URL (like /user/:id).

Nested Routes Routes inside other routes for better structure.

Redirects & Navigation Automatically move users to another page.

useNavigate, useParams Hooks to navigate programmatically or get URL params.

✅ Performance

Code Splitting Split code into smaller pieces so only needed code loads.

Lazy Loading Load components only when needed.

Optimizing Renders Avoid unnecessary re-renders using memo, useCallback, etc.

✅ Testing

Unit Testing (Jest) Test individual functions or components.

Component Testing (React Testing Library) Test full React components like how a user would use them.

✅ State Management

**Redux:** A popular and powerful library to manage global state in large applications. It keeps the state in one central store so that any component can access or update it.

**Redux Toolkit:** A modern, easier way to use Redux. It reduces boilerplate code and simplifies writing reducers and actions. It is the recommended way to use Redux now.

**Zustand:** A small and lightweight library to manage state. It is easier to use than Redux and does not require much setup. Good for small to medium apps.

**Recoil:** A state management library created by Facebook. It makes managing shared state simple with a similar approach to React hooks. Good for apps where multiple components share and update the same data.

1. **what is DOM ?**
2. **what is real DOM ?**
3. **what is virtual DOM ?**
4. **what is the diff b/w real and virtual DOM ?**
5. **Make it simple and easy way to explain these concepts?**

### ✅ What is DOM?

**DOM (Document Object Model)** is a tree-like structure that represents the entire webpage.  
Each HTML element (like <div>, <p>, <h1>) becomes a **node** in this tree.

🧠 Think of DOM like a family tree that shows the structure of your webpage.

### ✅ What is Real DOM?

The **Real DOM** is the actual structure of the webpage loaded in the browser.  
When you change something on the page (like a button click), the browser updates the real DOM.

❌ The problem: **Real DOM is slow to update**, especially when the page has many elements.

### ✅ What is Virtual DOM?

**Virtual DOM** is a **lightweight copy** of the real DOM created by React (and similar libraries).  
It lives in memory and **does not directly change the page**.

When something changes:

1. React updates the Virtual DOM first.
2. Then it compares it with the previous version (this is called “diffing”).
3. It finds out exactly **what changed**.
4. Only those small changes are updated in the **Real DOM** (this is called “reconciliation”).

✅ Result: **Much faster updates** and better performance.

### ✅ Difference between Real DOM and Virtual DOM:

| **Feature** | **Real DOM** | **Virtual DOM** |
| --- | --- | --- |
| Definition | Actual webpage structure | Copy of the real DOM (in memory) |
| Speed | Slower updates | Faster updates |
| Update Method | Re-renders the entire tree | Only updates changed parts |
| Created by | Browser | React (or similar libraries) |
| Performance | Less efficient | More efficient and optimized |

### 🎯 Interview Tip:

"Virtual DOM helps React improve performance by avoiding direct updates to the real DOM. It updates only the parts that change, making apps faster and smoother."