useEffect hook:-

The useEffect hook is a fundamental React feature that allows functional components to perform side effects. Side effects include operations like data fetching, subscriptions, timers, or manual DOM manipulations—anything that interacts with systems outside of React's rendering process.

Purpose:-

1. Manage Side Effects: Handle tasks that occur after rendering, such as API calls or event listeners.
2. Synchronize with State/Props: React to changes in data and update the component accordingly.
3. Clean Up Resources: Prevent memory leaks by disposing of subscriptions, timers, or other resources when a component unmounts.
4. When to Use useEffect:-
    - Data Fetching (API calls)
    - Event Listeners (keypress, scroll, resize)
    - Timers/Intervals (setInterval, setTimeout)
    - Manual DOM Updates (outside React's control)

Avoid for:-

- State transformations that don't need side effects.
- Calculations that can be done during rendering.

Summary:-

- useEffect runs after rendering and can optionally clean up before re-running or unmounting.
- The dependency array controls when the effect re-executes.
- Always clean up subscriptions, timers, and event listeners to avoid memory leaks.
- Use multiple useEffect calls to separate unrelated logic.

Component lifecycle in React:-

In React, each component goes through a lifecycle—a series of phases from creation to removal. Understanding this lifecycle is essential for managing side effects, optimizing performance, and ensuring clean component behavior.

1. Mounting Phase

When the component is being inserted into the DOM.

Class Component Lifecycle Methods:

- constructor() – Initializes state and binds methods.
- static getDerivedStateFromProps() – Syncs state with props (rarely used).
- render() – Returns JSX to render UI.
- componentDidMount() – Invoked once after the component is mounted. Ideal for API calls, subscriptions

2. Updating Phase

When the component is re-rendered due to changes in props or state.

Class Component Lifecycle Methods:

- static getDerivedStateFromProps() – Called before every render.
- shouldComponentUpdate() – Determines if re-render is necessary.
- render() – Re-renders the component.
- getSnapshotBeforeUpdate() – Captures information before changes (e.g., scroll position).
- componentDidUpdate() – Runs after the update is flushed to the DOM.

3. Unmounting Phase

When the component is removed from the DOM.

Class Component Lifecycle Method:

- componentWillUnmount() – Cleanup (e.g., remove timers, listeners).

Phase	Class Component Methods	Functional Component Hook
Mounting	constructor, render, componentDidMount	useEffect(() => {}, [])
Updating	shouldComponentUpdate, componentDidUpdate	useEffect(() => {}, [deps])
Unmounting	componentWillUnmount	useEffect(() => { return () => {} }, [])

Props vs State in React:-

Props (short for "properties") are the mechanism for **passing data from a parent component to a child**.

- **Read-only**: Cannot be modified by the receiving component.
- Used to configure a component or pass data.
- Passed like HTML attributes.

State is a local data storage that is **mutable** and controls a component's behavior and rendering.

- Managed **within the component**.
- Updated using setState (class components) or useState (functional components).
- Causes re-render when changed.

Feature	Props	State
Definition	Data passed to a component	Data managed within a component
Mutability	Immutable (read-only)	Mutable (can change over time)
Source	Parent component	Declared in the component itself
Lifecycle	Does not change over time	Can change and trigger re-renders
Access	this.props or directly in props	this.state or using useState hook
Purpose	Communication between components	Managing dynamic local data

Props vs State in Use:-

Use Props When:

- You want to pass **data or callbacks** to child components.
- The data **should not change** in the child.

Use State When:

- You want the component to **keep track of user input**, toggle UI elements, fetch dynamic data, etc.
- The data **will change over time**.

Today's Task:-

In your task, you built interactive profile cards using React. Each card displays a equipment image, name, role, and description. The key feature is a dynamic timer that tracks how long each profile has been visible. When users switch between different profiles, the timer automatically resets to zero when refreshed. The implementation uses React's useEffect hook to manage the timer, with proper cleanup to prevent memory leaks. The cards also feature a smooth hover effect that reveals additional details. This project demonstrates core React concepts like component structure, props for data passing, state management for the timer, and lifecycle control with useEffect. The clean, modular design allows easy addition of more profiles while maintaining consistent functionality.