**useState Hook in React**

**useState** is a special function introduced in React 16.8 that allows you to manage state within functional components. Prior to Hooks, state was primarily managed in class-based components.

**What is state?**

In React, state refers to data or properties within a component that can change over time. This could be anything from a counter value to user input in a form. When state changes, React re-renders the component with the updated information.

**How to use useState**

useState is called within a functional component. It accepts an initial state value as an argument (which can be a number, string, object, or array) and returns an array containing two values:

1. **Current state:** This represents the current value of the state variable.
2. **Setter function:** This function is used to update the state. When called with a new value, it triggers a re-render of the component with the updated state.

When I use complex computation using useState we have to pass a function inside the useState . Because then the function only rans one time ;

useEffect Hook

### Interview Preparation Notes: useEffect Hook in React.js

#### Overview

* **Purpose**: The **useEffect** hook in React.js allows you to perform side effects in function components. Side effects can include data fetching, subscriptions, or manually changing the DOM.
* **Usage**: **useEffect** replaces lifecycle methods like **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** in class components.

#### Key Concepts

1. **Effect Callback**
   * The function passed to **useEffect** is the effect callback, where you place your side effect logic.
   * This function runs after the first render and after every update by default.
2. **Dependencies Array**
   * The second argument to **useEffect** is an optional array of dependencies.
   * The effect runs only when one of the dependencies has changed.
   * If the array is empty (**[]**), the effect runs only once, similar to **componentDidMount**.
3. **Cleanup Function**
   * The effect callback can optionally return a function to clean up after the effect.
   * This cleanup function runs before the component is removed from the UI, similar to **componentWillUnmount**.
   * It also runs before executing the next effect if the component updates.
4. **Avoid Unnecessary Effects**
   * Ensure dependencies are correctly listed. Omitting dependencies can lead to stale closures.
   * Use multiple **useEffect** hooks if there are different sets of dependencies.
5. **Effect Cleanups**
   * Always clean up subscriptions, timers, or other resources to prevent memory leaks.
6. **Performance Considerations**
   * Avoid heavy computations inside the effect. If needed, consider optimizing with **useMemo** or **useCallback**.
7. **Handling Asynchronous Code**
   * Use async functions within effects correctly. Avoid making the effect callback itself async.

if you have add any event listener in useEffect hook make sure your remove addEventlistner in return statement to clean up the useEffect Hook. You’re return code always run first for clean up the previous code.

Major difference between useMemo and useCallback

useMemo returns the value of the function but useCallback returns thee entire function. We use useMemo and useCallback for referencial equality

In Transition hook we set a code to the hign priority and In useDeffered value we set the code low priority. When our all priority task completed after 100ms the react does the low priority task