Q. Explain Life Cycle in Class Component and Functional Component with Hooks

A. In React, components go through various lifecycle stages during their existence. These lifecycle stages are essential for managing component initialization, updates, and cleanup. The way you manage lifecycle in React differs between class components and functional components with Hooks.

**Class Component Lifecycle:**

Class components have access to lifecycle methods, which are special methods automatically called by React at specific points in a component's lifecycle. The main lifecycle stages in class components include:

1. **Mounting Phase:**
   * **constructor()**: Initializes state and binds event handlers.
   * **static getDerivedStateFromProps()**: Rarely used, but allows you to update state based on props.
   * **render()**: Renders the component's UI.
   * **componentDidMount()**: Called after the component is inserted into the DOM. Ideal for data fetching and other side effects.
2. **Updating Phase:**
   * **static getDerivedStateFromProps()**: As in the mounting phase, rarely used to update state based on props.
   * **shouldComponentUpdate()**: Decides whether the component should re-render. Useful for optimizing performance.
   * **render()**: Re-renders the updated UI.
   * **getSnapshotBeforeUpdate()**: Allows you to capture information from the DOM before changes are made.
   * **componentDidUpdate()**: Called after the component is updated, useful for side effects based on the updated state or props.
3. **Unmounting Phase:**
   * **componentWillUnmount()**: Called right before the component is removed from the DOM. Useful for cleanup.

**Functional Component with Hooks Lifecycle Equivalents:**

Functional components with Hooks introduce the **useEffect** Hook to manage lifecycle-like behavior. Hooks allow functional components to have similar lifecycle behavior as class components. Here's how the equivalent lifecycle stages are handled in functional components:

1. **Mounting Phase:**
   * **useState()**: Initializes state.
   * **useEffect(() => {}, [])**: Equivalent to **componentDidMount()**. The function inside **useEffect** runs after the component is inserted into the DOM. You pass an empty dependency array (**[]**) to ensure it runs only once, similar to **componentDidMount**.
2. **Updating Phase:**
   * **useState()**: Continues to manage state.
   * **useEffect(() => {})**: Equivalent to **componentDidUpdate()**. The function inside **useEffect** runs whenever the component updates (after rendering). If you want it to run conditionally based on specific props or state changes, include those dependencies in the dependency array (**[props, state]**).
   * You can use the **useEffect** cleanup function to perform cleanup tasks similar to **componentWillUnmount**.
3. **Unmounting Phase:**
   * **useEffect(() => {}, [])**: Equivalent to **componentWillUnmount()**. If you pass an empty dependency array, the cleanup function runs when the component unmounts.

Hooks provide a more flexible and concise way to manage component lifecycle in functional components. They allow you to organize side effects, state, and lifecycle behavior in a more declarative manner.

Top of Form