***Module (React Js) - 4***

**1). Explain Life cycle in Class Component and functional component with Hooks**

**Ans.** In React, components are the building blocks of a user interface, and they can be either class components or functional components with hooks. Each type of component has a different way of managing its lifecycle. Let's explore the lifecycle in both class components and functional components with hooks.

**Class Component Lifecycle:**

Class components have a well-defined lifecycle with a set of methods that you can override to perform specific tasks at different stages of a component's life. Here are the main stages of a class component's lifecycle:

1. **Initialization**:
   * **constructor()**: This method is called when the component is created. You can set initial state and bind methods here.
   * **static getDerivedStateFromProps()**: Used to update the component's state based on changes in props (rarely used).
2. **Mounting**:
   * **componentDidMount()**: Called after the component has been inserted into the DOM. It's a good place to fetch data from an API or set up event listeners.
3. **Updating**:
   * **shouldComponentUpdate()**: Allows you to control whether the component should re-render or not. You can optimize performance by returning **false** in some cases.
   * **componentDidUpdate()**: Called after a component's state or props have changed and it has re-rendered.
4. **Unmounting**:
   * **componentWillUnmount()**: Called just before a component is removed from the DOM. You can use it to clean up resources or event listeners.

**Functional Component with Hooks Lifecycle:**

Functional components with hooks have a more streamlined lifecycle compared to class components. Instead of lifecycle methods, you use built-in hooks to manage the component's behavior. Here are the equivalent hooks for the stages mentioned above:

1. **Initialization**:
   * State is declared using the **useState** hook, and there's no constructor. You can set initial state inside the functional component.
2. **Mounting**:
   * **useEffect(() => {}, [])**: The empty dependency array ensures that the effect runs only once after the component has rendered. You can perform side effects like data fetching here.
3. **Updating**:
   * **useEffect(() => {})**: The effect runs after every render. You can conditionally run code based on prop or state changes.
4. **Unmounting**:
   * **useEffect(() => { return () => {} }, [])**: To clean up resources when the component unmounts, you return a cleanup function from the effect. This is equivalent to **componentWillUnmount**.

Functional components with hooks provide a more declarative way to manage component behavior, and they encourage encapsulation and code organization. The code is often easier to read and maintain compared to class components with lifecycle methods. However, the choice between class components and functional components with hooks depends on your specific project and personal preference. In most cases, functional components with hooks are recommended for new projects due to their simplicity and readability.