**MODULE: 4 (List and Hooks)**

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

**→** A [React](https://reactjs.org/?ref=retool.com) component undergoes three different phases in its lifecycle, including mounting, updating, and unmounting. Each phase has specific methods responsible for a particular stage in a component's lifecycle. These methods are technically particular to [class-based components and not intended for functional components](https://www.twilio.com/blog/react-choose-functional-components?ref=retool.com).

However, since the concept of Hooks was released in React, you can now use abstracted versions of these lifecycle methods when you’re working with functional component state. Simply put, React Hooks are functions that allow you to [“hook into”](https://javascript.plainenglish.io/building-the-same-application-with-and-without-react-hooks-part-1-4bdcf02c9ab5?ref=retool.com) a React state and the lifecycle features within function components.

* **Phases of a React component's lifecycle:-**

A React component undergoes three phases in its lifecycle: mounting, updating, and unmounting.

* 1. **The mounting phase** is when a new component is created and inserted into the DOM or, in other words, when the life of a component begins. This can only happen once, and is often called “initial render.”
  2. **The updating phase** is when the component updates or re-renders. This reaction is triggered when the props are updated or when the state is updated. This phase can occur multiple times, which is kind of the point of React.
  3. The last phase within a component's lifecycle is **the unmounting phase**, when the component is removed from the DOM.

In a class-based component, you can call different methods for each phase of the lifecycle (more on this below). These lifecycle methods are of course not applicable to functional components because they can only be written/contained within a class. However, React hooks give functional components the ability to use states.

Hooks have gaining popularity because they make working with React cleaner and often less verbose.

* **React lifecycle methods:-**

→ Let’s learn more about the methods that make up each of our three phases.

**React Component Lifecycle**

**Mounting**

**Updating**

**Unmounting**

forceUpdate()

setState()

Props

Change

Constructer

shouldComponentUpdate

static getDerivedStateFromProps

render

getSnapshotBeforeUpdate

componentWillUnmount

componentDidUpdate

componentDidMount

* + **The mounting phase:-**

In the mounting phase, a component is prepared for and actually inserted into the DOM. To get through this phase, four lifecycle methods are called: constructor, static getDerivedStateFromProps, render, and componentDidMount.

For example:-

componentDidMount(){

console.log(“DidMount”)

}

* + **The updating phase:-**

The Updating phase is triggered when component props or state change, and consists of the following methods: static getDerivedFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate, and componentDidUpdate.

For example:-

componentDidUpdate(){

console.log(“DidUpdate”)

}

* + **The unmounting phase:-**

The unmounting phase is the third and final phase of a React component. At this phase, the component is removed from the DOM. Unmounting only has one lifecycle method involved: componentWillUnmount.

For example:-

componentWillUnmount(){

console.log(“WillUnmount”)

}

* **React Hooks and the component lifecycle**

Versions of React before 16.8 consider two kinds of components based on statefulness: the class-based stateful component, and the stateless functional components (often referred to as a “dumb component”). But with the release of React 16.8, Hooks were introduced and empowered developers to access state from functional components, instead of writing an entire class. With this change, building components became easier and less verbose.

Hooks known as default hooks come with React, and you’re also able to create your own custom hook. A custom hook is just a function that starts with use, like useStore, or useWhatever.

The two most common default hooks are useState and useEffect. The useState hook gives state to the functional component, and useEffect allows you to add side effects within it (like after initial render), which aren’t allowed within the function's main body. You can also act upon updates on the state with useEffect.

* + **useEffect**

As with the render() method of a class-based component, the main body of a functional component should be kept pure. With the [useEffect hook](https://reactjs.org/docs/hooks-effect.html?ref=retool.com), you're able to create side effects while maintaining the component's purity. Within this hook, you can send network requests, make subscriptions, and change the state value.

The useEffect hook accepts a function as an argument, where you write all your side effects. This function is invoked after every browser paint and before any new renders (this will depend on the dependency array, which is explained in the next paragraph). This function can return another function called the clean-up function, which can be used to clean up the side effects (i.e. when the component is destroyed) like unsubscribing to a store. It’s kind of a mash up of several of the methods explained in the previous section.

This Hook accepts a second argument called the dependency array, which is an array of dependencies like state or props values, which the useEffect uses as reference to only run when these values change. If the dependency array is empty, then the useEffect will only run once, after the first paint.

The dependency array is optional, so if it's not defined, useEffect will fire first when the component is first mounted, and then on every re-render.

For example:-

import { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

concole.log(“DidMount / DidUpdate”);

return(()=>{console.log(“WillUnmount”);})

}, [what to update]);

return;

}