-:react list and hooks:-

- Explain Life cycle in Class Component and functional component with Hooks.

ans.
:- A React 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.

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" 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.

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."

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.

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.

-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.

-The constructor method
The constructor method is the very first method called during the mounting phase.

It's important to remember that you shouldn't add any side effects within this method (like sending an HTTP request) as it's intended

to be pure.

This method is mostly used for initializing the state of the component and binding event-handler methods within the component. The constructor method is not necessarily required. If you don't intend to make your component stateful (or if that state doesn’t need to be initialized) or bind any method, then it’s not necessary to implement.

The constructor method is called when the component is initiated, but before it’s rendered. It’s also called with props as an argument. It's important you call the super(props) function with the props argument passed onto it within the constructor before any other steps are taken.

This will then initiate the constructor of React.Component (the parent of this class-based component) and it inherits the constructor method and other methods of a typical React component.

For example, below you can see a class-based component implementation of a counter with a constructor in it, where the state is initialized and an event-handler method is bound. :

```jsx
import React from 'react';
class Counter extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0
        };
        this.setCount = this.setCount.bind(this);
    }
    setCount() {
        this.setState({count: this.state.count + 1});
    }

    render() {
        return (
            <div>
                <h1>Counter</h1>
                <button onClick={this.setCount}>Click to add</button>
                <p>Count: {this.state.count}</p>
            </div>
        )
    }
}
```

-In the Counter component, you can see the component's state is initialized within the constructor method to keep track of the count state. The setCount method, which is an event-handler attached to your button in this case, is bound within the constructor.

The static getDerivedStateFromProps method
Props and state are completely different concepts, and part of building your app intelligently is deciding which data goes where.

In many cases though, your component’ s state will be derivative of its props. This is where the static getDerivedStateFromProps method comes in. This method allows you to modify the state value with any props value. It's most useful for changes in props over time, and we’ ll learn later that it’ s also useful in the update phase.

The method static getDerivedStateFromProps accepts two arguments: props and state, and returns an object, or null if no

change is needed. These values are passed directly to the method, so there’ s no need for it to have access to the instance of the class (or any other part of the class) and thus is considered a static method.

In the mounting phase, the getDerivedStateFromProps method is called after the constructor method and right before the component is rendered. This is the most rarely used method in this phase, but it’ s still important to know so you can use it if needed.

-The render method

The render method is the only required method for a class-based React component. It’s called after the getDerivedStateFromProps method and actually renders or inserts the HTML to the DOM.

Typically, the render method returns the JSX which will eventually be rendered, but it can also return other values.

It's important to remember that the render method is meant to be pure. This means you can’t modify the state, have any direct interaction with the browser, or any other kind of side effect like sending an HTTP request. Just think of it as writing HTML, but of course as JSX.

-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.

React has released more default hooks, but useState and useEffect are the ones you should be most familiar with. Let's take a look at how they work and compare them to the component lifecycle we covered above.

- useState

The useState hook is used to store state for a functional component. This hook accepts one parameter: initialState, which will be set as the initial stateful value, and returns two values: the stateful value, and the update function to update the stateful value. The update function accepts one argument, newState, which replaces the existing stateful value.