REACT JS

MODULE:4(List and Hooks)

Explain life cycle in class component and functional component with hooks?

What is the lifecycle of a Component?

As expected, the lifecycle of a component refers to its “lifetime” within our application which starts once the component is first rendered to screen up until the time we’re removing that component from the screen.

A component might be rendered to the screen, which is referred to as the “mounting phase”, it might receive some data that changes during its “lifetime”, which is referred to as the “update phase”, and it might, ultimately, be removed from the screen, which is referred to as “unmounting phase”.

What are the lifecycle methods available to us?

Now that we’ve got a better handle on the concept, let’s dig in and see what are the specific Lifecycle methods that we can use to interact with our components, when are they triggered, and what data are we able to access during that phase.

constructor()

First and foremost, we have the constructor of the actual component.

It is a mounting method that’s used to initialize state and bind methods inside of the class components. It’s also the place in which we’re calling super with props in case we want to initialize props for that component as well (You can read a more detailed explanation here).

```
constructor(props) {
  super(props)
  this.state = {
    portalName: 'Upmostly'
  }
  this.handleLogin = this.handleLogin.bind(this);
}
```

render()

Render is a required method in class-based components. Render should be a pure method that determines the return value of the component, or what the Component will render to the screen under the form of JSX.

```
render() {
  return <h1>Welcome to {this.state.portalName}</h1>
}
```

componentDidMount()

This method is called when the component is first mounted to the DOM.

It is typically used to fetch data from external APIs, as well as to manage subscriptions or set up event listeners.

Update the state from this lifecycle method will trigger a re-render of the Component, allowing the users to see the updated state reflected in the Component.

```
componentDidMount() {
  axios.get(testEndpoint).then(resp => this.setState({ testData: resp }));
}
```

componentWillUnmount()

The method is usually used for something we call “cleanup”.

You can treat it as the opposite of the componentDidMount, as it is where you should cancel any tasks that you might have initialized when the component was mounting (Subscriptions, Event Listeners, Opened Connections, Timers, etc.).

shouldComponentUpdate(nextProps, nextState)

This method is mostly used for improving the performance of the component.

It’s called before the component gets re-rendered through the update of its props or state. By returning false we can skip
the render and componentDidUpdate methods from running.

```
shouldComponentUpdate(nextProps) {
```

```
    if (nextProps.portalName !== this.props.portalName) return true
    return false
  }
```

getSnapshotBeforeUpdate(prevProps, prevState)

This method is called shortly before any of the component output gets added/rendered to the real DOM. It helps us capture some information that can be useful when calling componentDidUpdate.

componentDidUpdate(prevProps)

This method is called right after the component re-renders.

It's typically used for DOM manipulations or for making more efficient requests to other applications. This can be reflected in querying elements from the DOM using vanilla JavaScript methods, adding classes to elements, etc.

```
componentDidMount(prevProps) {
  if (this.props.ID !== prevProps.ID) {
    axios.get(testEndpoint)
      .then(resp => this.setState({ testData: resp }));
  }
}
```

getDerivedStateFromProps(nextProps, prevState)

This method is called every time the component is about to be re-rendered.

You are expected to return an object from it that will be used to update the state.

Lifecycle of a React component:

1. Initial Render or Mount

2. Update (When the states used in the component or props added to the component is changed)
3. Unmount

We will look into only those lifecycle methods which are used in most of the scenarios. Some of the methods are termed as rarely used in React documentation and advised to use them with caution.

1. Initial Render or Mount

```
// Merge of componentDidMount and componentDidUpdate
useEffect(() => {
 console.log("This is mounted or updated.");
});
```

In this variant of useEffect, the message will be printed when the component is mounted and every time the component state or props is updated.

2. Update

```
// Equivalent of componentDidMount
 useEffect(() => {
   console.log("This is mounted only not updated.");
 }, []);
```

In this variant, the message will be printed only once in the component's life cycle and that is after the component is mounted.

```
// Merge of componentDidMount and componentDidUpdate but only
for given dependency useEffect(() => {
   console.log("This is mounted or count state updated.");
 }, [count]);
```

3. Unmount

```
// Equivalent of componentWillUnmount
 useEffect(() => {
   return () => {
     console.log("This is unmounted.");
```

```
    };
 }, []);
```

If we don't want any side effect to happen on component mounting or updating but only when a component is being unmounted, then we can use this variant.

Code Available here

Controlling re-renders: Pure Components

One of the ways to control the re-rendering of a component is using `React.memo` Higher Order Component.

React.memo uses memoization. It **shallowly** compares the previous props with current props to determine if the props are changed. If they are changed, then the component is re-rendered.

We can provide the custom comparator to the React.memo as well to customize the comparison.

PureComponent.tsx

```
import React from "react";

const RegularComponent = (props: any) => {
  console.log("***Pure component is rendered***");
  return <p>Pure Component = {props.name.firstName}</p>;
};

const PureComponent = React.memo(RegularComponent);

export {PureComponent};
```

OR with comparator:

```
import React from "react";

const PureComponent = React.memo(
  (props: any) => {
    console.log("***Pure component is rendered***");
    return <p>Pure Component = {props.name.firstName}</p>;
  },
  (prevProps, nextProps) => {
    /*
    return true if passing nextProps to render would return
    the same result as passing prevProps to render,
    otherwise return false
    */
    if (prevProps.name.firstName === nextProps.name.firstName)
return true;
    return false;
  }
);

export { PureComponent };
```

RegularComponent.tsx

```
import React from "react";

const RegularComponent = (props: any) => {
  console.log("***Regular component is rendered***");

  return <p>Regular Component = {props.name.firstName}</p>;
};

export default RegularComponent;
```