**Assignment 7: React Hooks**

**Task: Introduce React hooks (useState, useEffect).**

**Refactor class components to functional components with hooks.**

React hooks are functions that enable functional components to have state and side effects. Two fundamental hooks are useState for managing state and useEffect for handling side effects in functional components. Here's an introduction to these hooks and how to refactor a class component to a functional component using hooks.

React Hooks:

1.useState:

useState allows functional components to manage state.

It returns an array with the current state value and a function to update it.

| import React, { useState } from 'react';  const ExampleComponent = () => {  const [count, setCount] = useState(0);  return (  <div>  <p>Count: {count}</p>  <button onClick={() => setCount(count + 1)}>Increment</button>  </div>  );  }; |
| --- |

2.useEffect:

useEffect is used for side effects in functional components, such as data fetching, subscriptions, or manually changing the DOM.

It takes a function that contains the code for the side effect.

| import React, { useState, useEffect } from 'react';  const DataFetchingComponent = () => {  const [data, setData] = useState(null);  useEffect(() => {  // Code for data fetching or other side effects  // This will run after the component renders  }, []); // The empty dependency array means it only runs once (componentDidMount)  return <div>{data ? <p>Data: {data}</p> : <p>Loading...</p>}</div>;  }; |
| --- |

Refactoring Class Components to Functional Components with Hooks:

Let's consider a class component and refactor it to a functional component using hooks:

| // ClassComponent.js  import React, { Component } from 'react';  class Counter extends Component {  constructor(props) {  super(props);  this.state = {  count: 0,  };  }  componentDidMount() {  // Code to run after the component is mounted  }  componentDidUpdate(prevProps, prevState) {  // Code to run after the component is updated  }  componentWillUnmount() {  // Code to run before the component is unmounted  }  render() {  return (  <div>  <p>Count: {this.state.count}</p>  <button onClick={() => this.setState({ count: this.state.count + 1 })}>  Increment  </button>  </div>  );  }  }  export default Counter; |
| --- |

Now, let's refactor it to a functional component with hooks:

| // FunctionalComponent.js  import React, { useState, useEffect } from 'react';  const Counter = () => {  const [count, setCount] = useState(0);  useEffect(() => {  // Code to run after the component is mounted  return () => {  // Code to run before the component is unmounted  };  }, []); // The empty dependency array means it only runs once (componentDidMount)  useEffect(() => {  // Code to run after the component is updated  }, [count]); // Run whenever count changes  return (  <div>  <p>Count: {count}</p>  <button onClick={() => setCount(count + 1)}>Increment</button>  </div>  );  };  export default Counter; |
| --- |

This refactoring maintains the functionality of the original class component but now uses functional components and React hooks. The useState hook manages the state, and the useEffect hook handles lifecycle methods.