Class-08: React Hooks

Introduction to React Hooks

React Hooks were introduced in **React 16.8** to allow functional components to use state and lifecycle features without writing a class. Hooks simplify state management and side effects in functional components.

Why Use Hooks?

- Avoids class components complexity
- Enables state and lifecycle methods in functional components
- Promotes code reusability through custom hooks
- Improves readability and maintainability

Basic Hooks

1. useState

The useState hook allows functional components to manage local state.

Syntax:

```
const [state, setState] = useState(initialValue);
```

Example:

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
export default Counter;
```

2. useEffect

The useEffect hook is used to handle side effects such as API calls, subscriptions, and manual DOM manipulations.

Syntax:

```
useEffect(() => {
  // Side-effect logic
  return () => {
    // Cleanup function (optional)
  };
}, [dependencies]);
```

Example:

```
import React, { useState, useEffect } from 'react';

function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setCount(count + 1);
    }, 1000);

    return () => clearInterval(interval); // Cleanup function
  }, [count]);

  return <p>Timer: {count}</p>;
}
export default Timer;
```

Advanced Hooks

3. useContext

The useContext hook allows components to access values from the React Context API without prop drilling.

Example:

```
import React, { createContext, useContext } from 'react';

const ThemeContext = createContext('light');

function ThemedComponent() {
```

```
  const theme = useContext(ThemeContext);
  return <p>Current Theme: {theme}</p>;
}

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedComponent />
    </ThemeContext.Provider>
  );
}
export default App;
```

4. useReducer

The useReducer hook is useful for managing complex state logic in functional components.

Syntax:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Example:

```
import React, { useReducer } from 'react';

const initialState = 0;
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    default:
      return state;
  }
}

function Counter() {
  const [count, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}
export default Counter;
```

5. useRef

The useRef hook provides a way to reference DOM elements or persist values without causing re-renders.

Example:

```
import React, { useRef } from 'react';

function FocusInput() {
  const inputRef = useRef(null);

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Type here..." />
      <button onClick={handleClick}>Focus Input</button>
    </div>
  );
}
export default FocusInput;
```

Custom Hooks

Custom hooks allow the reuse of logic across multiple components. A custom hook is simply a function that uses other hooks.

Example: useFetch Hook:

```
import { useState, useEffect } from 'react';

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        setLoading(false);
      });
  }, [url]);
```

```
  return { data, loading };
}
export default useFetch;
```

Usage Example:

```
import React from 'react';
import useFetch from './useFetch';

function UserList() {
  const { data, loading } =
useFetch('https://jsonplaceholder.typicode.com/users');

  if (loading) return <p>Loading...</p>;

  return (
    <ul>
      {data.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
export default UserList;
```

Summary

- **useState**: Manages state in functional components.
- **useEffect**: Handles side effects and cleanup.
- **useContext**: Provides context without prop drilling.
- **useReducer**: Manages complex state logic.
- **useRef**: References DOM elements or persists values.
- **Custom Hooks**: Encapsulates and reuses logic across components.

React Hooks simplify component logic and make code more reusable and maintainable. Understanding and mastering these hooks will greatly enhance React development efficiency.