Class-09: Advanced React

Introduction

Advanced React concepts help improve code reusability, maintainability, and performance. This class covers **Higher-Order Components (HOCs), Render Props, Error Boundaries, and Performance Optimization using React Memo.**

Higher-Order Components (HOCs)

What is a Higher-Order Component?

A **Higher-Order Component (HOC)** is a function that takes a component and returns a new component with additional functionality.

Syntax

```
const withLogger = (WrappedComponent) => {
  return (props) => {
    console.log("Props: ", props);
    return <WrappedComponent {...props} />;
  };
};
```

Example

```
import React from 'react';

const withTheme = (WrappedComponent) => {
  return (props) => {
    const theme = 'dark';
    return <WrappedComponent theme={theme} {...props} />;
  };
};

const Button = ({ theme }) => {
  return <button style={{ background: theme === 'dark' ? '#333' : '#fff', color:
'#fff' }}>Click Me</button>;
};
const ThemedButton = withTheme(Button);

export default ThemedButton;
```

When to Use HOCs?

- Code reuse across multiple components
- Abstracting logic like authentication, logging, or styling
- Enhancing component functionality without modifying the original component

Render Props

What is Render Props?

A **render prop** is a function passed as a prop that allows dynamic rendering logic inside a component.

Example

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

const MouseTracker = ({ render }) => {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  const handleMouseMove = (event) => {
    setPosition({ x: event.clientX, y: event.clientY });
  };

  return <div
onMouseMove={handleMouseMove}>{render(position)}</div>;
};

const App = () => {
  return (
    <MouseTracker render={({ x, y }) => (
      <p>Mouse Position: {x}, {y}</p>
    )} />
  );
};

export default App;
```

When to Use Render Props?

- Sharing logic between components without using HOCs
- Handling dynamic UI rendering (e.g., mouse tracking, authentication state, data fetching)

Error Boundaries

What is an Error Boundary?

An **Error Boundary** is a special React component that catches JavaScript errors in child components and displays a fallback UI instead of crashing the entire app.

Syntax

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.error("Error: ", error, info);
  }

  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong.</h2>;
    }
    return this.props.children;
  }
}
```

Example Usage

```
import React from 'react';
import ErrorBoundary from './ErrorBoundary';
import SomeComponent from './SomeComponent';

function App() {
  return (
    <ErrorBoundary>
      <SomeComponent />
    </ErrorBoundary>
  );
}
export default App;
```

When to Use Error Boundaries?

- Wrapping components to catch errors gracefully
- Handling UI crashes and preventing application breakdowns
- Displaying fallback UI for better user experience

React Memo and Performance Optimization

What is React.memo?

React.memo is a higher-order component that optimizes functional components by **preventing unnecessary re-renders** if props haven't changed.

Syntax

```
const MemoizedComponent = React.memo(MyComponent);
```

Example

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

const Counter = ({ count }) => {
  console.log("Counter rendered");
  return <p>Count: {count}</p>;
};

const MemoizedCounter = React.memo(Counter);

const App = () => {
  const [count, setCount] = useState(0);
  const [text, setText] = useState('');

  return (
    <div>
      <MemoizedCounter count={count} />
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <input type="text" value={text} onChange={(e) =>
setText(e.target.value)} />
    </div>
  );
};

export default App;
```

Performance Optimization Tips

- Use React.memo to avoid unnecessary renders.
- Use useCallback and useMemo to memoize functions and values.
- Avoid unnecessary state updates and re-renders.
- Optimize expensive computations using useMemo.

Summary

- **Higher-Order Components (HOCs):** Used for code reuse by wrapping components.
- **Render Props:** Used to share logic between components dynamically.
- **Error Boundaries:** Helps in handling component errors and preventing crashes.
- **React Memo & Optimization:** Improves performance by reducing unnecessary re-renders.

By mastering these advanced concepts, React developers can build more scalable, efficient, and maintainable applications.