Class Components:

**1. Class Component Basics**

**Structure of a Class Component**

* Extends React.Component.
* Must have a render() method that returns .
* Uses this.props and this.state to access props and state.

import React, { Component } from 'react';

class Counter extends Component {



constructor(props) {



super(props); // Required to call `super(props)` before using `this`

this.state = { count: 0 };

}



increment = () => {



this.setState({ count: this.state.count + 1 });



};



render() {

return (



<div>

<h1>Count: {this.state.count}</h1>

<button onClick={this.increment}>Increment</button>

</div>



);



}



}



export default Counter;

**Key Features**

| **Feature** | **Description** |
| --- | --- |
| **constructor(props)** | Initializes state and binds methods. |
| **this.state** | Holds component-specific data. |
| **this.setState()** | Updates state and triggers re-render. |
| **this.props** | Immutable data passed from parent. |
| **render()** | Returns (required). |

**2. State Management in Class Components**

**Rules of State**

1. **Never modify state directly:**

this.state.count = 1; // ❌ Wrong! Use `setState()`.

1. **State updates may be asynchronous:**

this.setState({ count: this.state.count + 1 }); // ✅ Correct

1. **Use a function if new state depends on previous state:**

this.setState((prevState) => ({ count: prevState.count + 1 })); // ✅ Safer for async updates

**Example: Updating State Correctly**

import React, { Component } from "react";

class Counter extends Component {

  state = { count: 0 };

  increment = () => {

    this.setState({ count: this.state.count + 1 });

  };

  decrement = () => {

    this.setState((prevState) => ({ count: prevState.count - 1 }));

  };

  render() {

    return (

      <div>

        <button onClick={this.decrement}>-</button>

        <span>{this.state.count}</span>

        <button onClick={this.increment}>+</button>

      </div>

    );

  }

}

export default Counter;

**3. Props in Class Components**

**Props are Read-Only**

* Passed from parent components.
* Cannot be modified by the child

class Greeting extends Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

// Usage: <Greeting name="Alice" />

**Default Props & PropTypes**

import PropTypes from 'prop-types';

class Greeting extends Component {

static defaultProps = { name: "Guest" }; // Fallback if prop is missing

static propTypes = { name: PropTypes.string }; // Type-checking (optional)

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

**4. Lifecycle Methods (Most Important for Exams)**

**1. Mounting Phase (Component Creation)**

| **Method** | **Purpose** |
| --- | --- |
| **constructor(props)** | Initialize state, bind methods. |
| **render()** | Returns (required). |
| **componentDidMount()** | Runs after component mounts (API calls, subscriptions). |

class DataFetcher extends Component {

  // State holds the fetched data (initially null)

  state = { data: null };

  // componentDidMount runs once when the component is added to the DOM

  componentDidMount() {

    fetch("https://jsonplaceholder.typicode.com/posts/1") // Fetching data from API

      .then((res) => res.json()) // Convert response to JSON

      .then((data) => this.setState({ data })); // Update state with fetched data

  }

  render() {

    return (

      <div>

        {/\* Display data if available, otherwise show "Loading..." \*/}

        {this.state.data ? <p>{this.state.data.title}</p> : <p>Loading...</p>}

      </div>

    );

  }

}

export default DataFetcher;

### ****Step-by-Step Explanation****

1. **State Initialization:**
   * state = { data: null };
   * The data property is initialized as null because no data is available yet.
2. **Lifecycle Method (componentDidMount)**
   * This method runs **once** after the component is mounted.
   * fetch("https://jsonplaceholder.typicode.com/posts/1") makes an API call.
   * .then(res => res.json()) converts the response into JSON format.
   * .then(data => this.setState({ data })); updates the data state.
3. **Rendering Data Conditionally**
   * this.state.data ? <p>{this.state.data.title}</p> : <p>Loading...</p>;
   * If data is available, it displays the title from the API.
   * If data is null, it shows "Loading..."

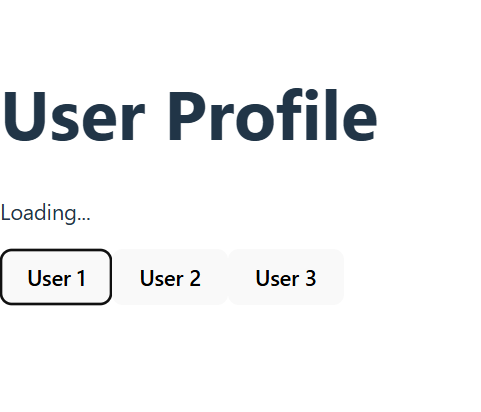
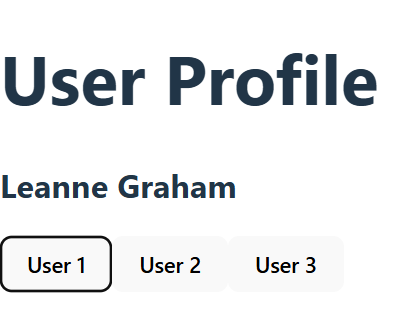
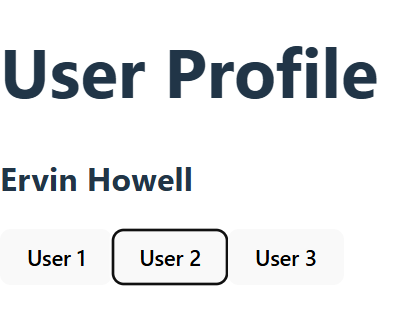
**Why Use** componentDidMount**?**

* It ensures the API call runs **after** the component is added to the DOM.
* If the API request was in render(), it would cause infinite re-renders.

**2. Updating Phase (Props/State Change)**

| **Method** | **Purpose** |
| --- | --- |
| **shouldComponentUpdate(nextProps, nextState)** | Optimize re-renders (return true/false). |
| **render()** | Re-renders UI. |
| **componentDidUpdate(prevProps, prevState)** | Runs after update (side effects). |

|  |  |  |
| --- | --- | --- |
| Parent.jsx | | UserProfile.jsx |
| import React, { useState } from "react";  import UserProfile from "./UserProfile.jsx"; // Import the UserProfile component  const App = () => {    const [userId, setUserId] = useState(1);  // State to store userId    return (      <div>        <h1>User Profile</h1>        {/\* Pass userId as a prop to UserProfile \*/}        <UserProfile userId={userId} />        {/\* Buttons to change userId dynamically \*/}        <button onClick={() => setUserId(1)}>User 1</button>        <button onClick={() => setUserId(2)}>User 2</button>        <button onClick={() => setUserId(3)}>User 3</button>      </div>    );  };  export default App; | import React, { Component } from "react";  class UserProfile extends Component {      // Initialize state with a null user      state = { user: null };      // Runs whenever props change (userId in this case)      componentDidUpdate(prevProps) {          // Check if userId has changed before making a new API call          if (this.props.userId !== prevProps.userId) {              fetch(`https://jsonplaceholder.typicode.com/users/${this.props.userId}`)                  .then((res) => res.json())                  .then((user) => this.setState({ user })); // Update state with new user data          }      }      render() {          return (              <div>                  {/\* Show user's name if available, otherwise show "Loading..." \*/}                  {this.state.user ? <h2>{this.state.user.name}</h2> : <p>Loading...</p>}              </div>          );      }  }  export default UserProfile; | |

**3. Unmounting Phase (Component Removal)**

| **Method** | **Purpose** |
| --- | --- |
| **componentWillUnmount()** | Cleanup (remove event listeners, timers). |

import React, { Component } from "react";

class Timer extends Component {

  state = { seconds: 0 };

  componentDidMount() {

    // Start a timer when the component is mounted

    this.timerID = setInterval(() => {

      this.setState((prevState) => ({ seconds: prevState.seconds + 1 }));

    }, 1000);

  }

  componentWillUnmount() {

    // Clear the timer when the component is removed

    clearInterval(this.timerID);

    console.log("Timer stopped!");

  }

  render() {

    return <h2>Timer: {this.state.seconds} seconds</h2>;

  }

}

export default Timer;

**6. Class Components vs. Functional Components**

| **Feature** | **Class Components** | **Functional Components** |
| --- | --- | --- |
| **State** | this.state & setState() | useState Hook |
| **Lifecycle** | componentDidMount, componentDidUpdate | useEffect Hook |
| **Props** | this.props | Direct props argument |
| **Syntax** | More verbose | Cleaner, modern |
| **Error Boundaries** | Supported (componentDidCatch) | Not supported |

**7. When to Use Class Components Today**

✅ **Legacy codebases** (if not migrated to hooks).  
✅ **Error boundaries** (only possible with classes).  
✅ **Exam scenarios** (if asked explicitly).

Otherwise, **use functional components with hooks** for new projects.

**8. Exam-Style Questions**

1. **What is the purpose of super(props) in a class component?**
   * It calls the parent class (React.Component) constructor and initializes this.props.
2. **How do you prevent unnecessary re-renders in a class component?**
   * Use shouldComponentUpdate(nextProps, nextState) and return false if no update is needed.
3. **What is the difference between componentDidMount and componentDidUpdate?**
   * componentDidMount runs **once after initial render**, while componentDidUpdate runs **after every re-render**.

## ****Lifecycle Methods (Class Components)****

### ****1. Mounting Phase****

* constructor() → render() → componentDidMount()
* Used for initial setup (API calls, subscriptions).

componentDidMount() {

console.log("Component mounted!");

fetch("https://api.example.com/data")

.then(res => res.json())

.then(data => this.setState({ data }));

}

### ****2. Updating Phase****

* Triggered by setState() or new props.
* render() → componentDidUpdate(prevProps, prevState)

componentDidUpdate(prevProps) {

if (this.props.userId !== prevProps.userId) {

fetch(`https://api.example.com/users/${this.props.userId}`)

.then(res => res.json())

.then(user => this.setState({ user }));

}

}

### ****3. Unmounting Phase****

* componentWillUnmount() → Cleanup (remove event listeners, timers).

componentWillUnmount() {

window.removeEventListener("resize", this.handleResize);

}