CS390 WEB APPLICATION DEVELOPMENT

NISARG KOLHE

Lifecycle of components

Lifecycle of components

In applications with many components, it’s very important to free up resources taken by the components when they are destroyed.

We care about two events in the life of a component:

1. When it’s **rendered** in the DOM.
2. When it’s **removed** from the DOM.

Lifecycle of components

componentDidMount()
runs after component output has been rendered to the DOM.

componentWillUnmount()
runs before the component is removed from the DOM.

Example (React docs)

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({date: new Date()});
  }

  render() {
    return <h2>It’s {this.state.date.toLocaleTimeString()}.</h2>;
  }
}
```

Handling events

HTML events with JS

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

Events in React

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Pitfalls with events

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
 }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

Pitfalls with events

```
handleClick() {
  this.setState(state => ({
    isToggleOn: !state.isToggleOn
  }));
}
```

this in handleClick() refers to the button, not the component class!

Hence, **this.setState** won’t work.

Three ways to fix **this**.

1. Use .bind() to bind context of the class to the function

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
 }

  handleClick() {
    this.setState(state => ({isToggleOn: !state.isToggleOn}));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

2. Use experimental public class fields syntax

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
  }

  handleClick = () => {
    this.setState(state => ({isToggleOn: !state.isToggleOn}));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

3. Use **arrow** functions

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
  }

  handleClick() {
    this.setState(state => ({isToggleOn: !state.isToggleOn}));
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

3. Need to use **arrow** functions for passing arguments

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
  }

  handleClick = (toggle) => {
    this.setState(state => ({isToggleOn: !state.isToggleOn}));
  }

  render() {
    return (
      <button onClick={() => this.handleClick(true)}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

Conditional rendering.

Just use if conditions!

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
}
```

Just use **if** conditions!

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        { isLoggedIn ? (
           <LogoutButton onClick={this.handleLogoutClick} />
          ) : (
           <LoginButton onClick={this.handleLoginClick} />
        )}
      </div>
    );
}
```

Add CSS classes using **className**

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    return (
      <div className="btn">
        <Greeting isLoggedIn={isLoggedIn} />
        { isLoggedIn ? (
           <LogoutButton onClick={this.handleLogoutClick} />
          ) : (
           <LoginButton onClick={this.handleLoginClick} />
        )}
      </div>
    );
}
```

Add CSS classes using className

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;
    let btnStyle = isLoggedIn ? 'logoutBtn' : 'loginBtn';

    return (
      <div className={btnStyle}>
        <Greeting isLoggedIn={isLoggedIn} />
        { isLoggedIn ? (
           <LogoutButton onClick={this.handleLogoutClick} />
          ) : (
           <LoginButton onClick={this.handleLoginClick} />
        )}
      </div>
    );
}
```

Loops.

Use **.map** to render multiple components

```
/*
  props.todos = [“Do laundry”, “Finish Homework”, “Clean up”]
*/

function TodoList() {
    let todos = props.todos.map((msg, i) => <Todo key={i} msg={msg}/>

    return (
      <div className={btnStyle}>
        {todos}
      </div>
    );
}
```

Note: Multiple instances of components need unique **key** attribute

Resources

Entire lab 2 implemented in React (as shown live in the lecture)

https://github.com/nisargkolhe/SimpleReactNotes