Class-Based Components

Course: Java

S1

Learning Outcomes

By the end of this lesson, you will be able to:

01 Rewrite an existing functional component as a class-based component.

02 Set initial state in the constructor or `componentDidMount`.

03 Update state with `setState`.

04 Handle data and function props in a class component.

05 Use the `this` qualifier effectively.

Class-Based Components

Class-Based Quick Start

- A component is a class that extends `React.Component`.
- Props are passed via a constructor.
- Set initial state directly in the constructor. From then on, use `this.setState`.
- The `render` method returns our JSX.

```
// code\try-react\src\class-based\Clicker.js
import React from 'react';

class Clicker extends React.Component {

    constructor(props) {
      super(props);
       this.state = { clicks: 0 };
    }

    render() {
       const clicks = this.state.clicks;
       return (
           <>
             <h1>{this.props.label}</h1>
             <button onClick={() => this.setState({ clicks: clicks + 1 })}>
                    Click
             </button>
             <div>Clicks: {clicks}</div>
            </>
        );
    }
}

export default Clicker;
```

As JSX

- Functional and class-based components are expressed identically with JSX.
- Class-based components use the class name, versus the function name in functional components.
- Props can include any value.

```
return <Clicker label="Clicker #1" />;
```

this

An instance of a class-based component is an object, so all members must be referenced with `this`.

```
constructor(props) {
    super(props);
    this.state = { clicks: 0 };
}

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

render() {
    return (
        <>
            <h1>{this.props.label}</h1>
            <button onClick={() => this.handleClick()}>
                Click
            </button>
            <div>Clicks: {this.state.clicks}</div>
        </>
    );
  }
```

State

- Initial state is set in the constructor. It’s a JavaScript object.
- After that, state is modified only via the `setState` method.
- The `setState` method replaces only the properties specified. It doesn’t replace the whole state object.

```
constructor() {
    super();
    this.state = {
        name: "",
        clicks: 0
    };
}

render() {
    const clicks = this.state.clicks;
    return (
        <>
            <button onClick={() => this.setState({ clicks: clicks + 1 })}
                disabled={this.state.clicks >= 10}>
                Clicks: {clicks}
            </button>
            <input value={this.state.name}
                onChange={(evt) => this.setState({ name: evt.target.value
})}} />
            <div>{this.state.name}</div>
        </>
    );
  }
```

constructor()

- If `this` is required in the constructor, call the `super` constructor before using it.
- If `this` isn't required, the constructor can be omitted.

```
// #1 no props, but use `this`
constructor() {
    super();
    this.state = { n: 1 };
}

// #2 props and `this`
constructor(props) {
    super(props);
    this.state = { n: 1 };
}

// #3 if `this` isn't required,
// the constructor isn't required.
```

Props

- Class-based props work identically to functional components except that they are object members.
- We can pass callback functions.
- Destructure props in `render` to clean up JSX.

```
constructor(props) {
    super(props);
    this.state = { ...props.initialToDo };
}

handleSubmit(evt) {
    evt.preventDefault();
    this.props.onSubmit(this.state);
}

render() {
    const { className, header } = this.props;
    return <h1 className={className}>{header}</h1>
  }
```

bind(this)

- To clean up our JSX a bit, we can bind methods to `this`.
- Binding allows us to use `this.method` directly in JSX event attributes/props.

```
constructor(props) {
    super(props);
    this.state = { ...props.initialToDo };
// required for onSubmit={this.handleSubmit}
      this.handleSubmit = this.handleSubmit.bind(this);
  }

handleSubmit(evt) {
    evt.preventDefault();
    this.props.onSubmit(this.state);
}

render() {
    return (
        <>
            <h1>Add a ToDo</h1>
            <form onSubmit={this.handleSubmit}>
                {/* More JSX */}
            </form>
        </>
    );
}
```

Break

Set State with Props

- State and props don't update at the same time. Updates are asynchronous.
- Never `setState` as an object with a `this.props` value.
- Instead, use the `setState` overload that accepts current state and props.

```
// bad, could fail because of async updates
this.setState(
   { clicks: this.state.clicks + this.props.value }
);

// good, update is guaranteed
this.setState((state, props) => ({
   clicks: state.clicks + props.value
}));
```

componentDidMount

- To set initial state, use the `componentDidMount` method.
- It’s equivalent to `useEffect(() => {}, [])` in a functional component.
- The `componentDidMount` method runs when the component is added (mounted) to the DOM.

```
class ToDoList extends React.Component {

    constructor() {
        // snip
    }

    componentDidMount() {
        fetch("http://localhost:8080/api/todo")
            .then(response => response.json())
            .then(result => {
                this.setState({ todos: result })
            })
            .catch(console.log);
    }

    render() {
        // snip
    }
  }
```

Time to Code

Rewrite a Functional Component as a Class

Suggested Time:

20 Minutes

Activity: Rewrite a Functional Component as a Class

Suggested Time:

60 minutes

RECAP

Questions?