React Event Handling

Responding to User Input the Right Way

JSX Events

- When looking at JSX previously, we saw that typically we don't alter DOM elements made with JSX directly, but instead define behavior during render
- Binding events is similar, we have to bind them initially because a re-render could otherwise break the binding
- Fortunately JSX makes this easy by allowing us to pass properties that are functions, and exposing the event HTML event attributes to us
  - HTML elements all have attributes for binding events in the DOM
  - We did not use this previously, as it's bad practice in a non-react app
- These are camelCased version of the events we've used previously

JSX Events - Comparison

```
<!-- This HTML event binding... -->
<button id="some-button" onclick="someFunction">
    Click me!
</button>

// ...is the same as this vanilla JS binding...
const button = document.getElementById("some-button");
button.addEventListener("click", someFunction);

// ...is the same as this jQuery binding...
$("#some-button").on("click", someFunction);

// ..is the same as this JSX binding!
render() {
    return <button onClick={someFunction}>Click me!</button>;
}
```

JSX Events - Multiple Events

- You can attach as many events to as many elements as you'd like
- I won't list them all here, but here are some examples of the many events

```
render() {
    return (
        <form onSubmit={/* function */}>
            <input
                onChange={/* function */}
                onFocus={/* function */}
                onBlur={/* function */}
            />
            <button
                onClick={/* function */}
                onMouseEnter={/* function */}
                onMouseLeave={/* function */}
            />
        </form>
    )
}
```

JSX Events - Solving `this` Issues

- Most of the time we bind events in React, it's to a class method
  - This is usually to call `this.setState` or reference `this.props`
- Unfortunately, JSX events don't solve the annoying `this` problems we've had in past javascript projects
- To handle this, we'll need to define the class methods in a special way to ensure the `this` keyword points to our component
- There are two ways of doing this, but we'll prefer doing it the second way

JSX Events - Solving `this` Issues (Code)

- This takes the class method we defined, and redefines it where `this` *always* refers to that instance of `MyComponent`

```
class MyComponent extends React.Component {
    constructor(props) {
        super(props);
        this._handleClick = this._handleClick.bind(this);
    }

    _handleClick(ev) {
        console.log("You just clicked on", ev.target);
    }

    render() {
        return <div onClick={this._handleClick}>Click me</div>;
    }
}
```

JSX Events - Solving `this` Issues (Alt. Code)

- This does the exact same thing, we can assign attributes to class instances by simply doing an assignment at the class level

```
class MyComponent extends React.Component {
    _handleClick = (ev) => {
        console.log("You just clicked on", ev.target);
    };

    render() {
        return <div onClick={this._handleClick}>Click me</div>;
    }
}
```

Sidenote: Underscored Functions

- In React Components, it's good practice to name non-overriden internal functions with a leading underscore
- This indicates to other developers that this is a custom function unique to this component that is not called externally
- We'll revisit this more in depth later

JSX Events - Updating State

- One of the main use cases of events is to update some internal state when something happens
- This can be to change a value, or to indicate to the user that we acknowledge their action
- Once we've correctly bound our handler function to refer to `this`, we can call `this.setState` inside of the handler
- This will cause a re-render to trigger after the event, reflecting the change in state

JSX Events - Updating State (Code)

```
// components/CounterButton.js
class CounterButton extends React.Component {
    state = {
        number: 0,
    };

    _incrementCounter = () => {
        this.setState({ number: this.state.number + 1 });
    };

    render() {
        const { number } = this.state;
        return (
            <button onClick={this._incrementCounter}>
                Clicked {number} times
            </button>
        );
    }
}
```

JSX Events - Calling Prop Functions

- Another common use case for events is to have a component report when events happen to it
- To implement this, it can call a function that was passed to it as a prop
  - We're basically inventing our own, custom `on[Event]` props
- This allows them to be as reused all throughout your app
- Think how many uses you've had for the `<input>` or `<button>` elements, we want our components to be that level of flexible

JSX Events - Calling Prop Functions (Code)

- Here the same Button component is used for a save and a cancel button

```
// components/Form.js
render() {
    return (
        <div>
            <Button color="green" onClick={this._save}>Save</Button>
            <Button color="red" onClick={this._cancel}>Cancel</Button>
        </div>
    );
}

// components/Button.js
render() {
    const { color, children } = this.props;

    return (
        <button
            className={`Button is-color-${color}`}
            onClick={this.props.onClick}
        >
            {children}
        </button>
    );
}
```

JSX Events - Prop Function & State (Code)

```
// components/Button.js
render() {
    const { color, children } = this.props;
    const { wasClicked } = this.state;

    return (
        <button
            className={`Button is-color-${color} ${wasClicked && "is-disabled"}`}
            onClick={this._handleClick}
        >
            {children}
        </button>
    );
}

_handleClick = (ev) => {
    // Prevent double-clicks
    if (this.state.wasClicked) {
        return;
    }
    this.setState({ wasClicked: true });
    this.props.onClick(ev);
}
```

JSX Events - Lifting State

- Even though a component often shouldn't be opinionated about what its actions do, sometimes it needs to change after an action
- However, its parent determines a lot about it through props or rendering it
- So in order for a component to correctly behave, we often pass it a prop function that changes state in the parent component, that affects rendering in a child component
- This is a complex topic, so don't be worried if it's a bit confusing

JSX Events - Lifting State (Code pt1)

- The `TodoItem` has a prop funciton for when you click a delete button
- The component should *not* delete itself, because it doesn't know what todo list it corresponds to, the parent component does

```
// components/TodoItem.js
_handleDelete = () => {
    this.onClickDelete(this.props.id);
}

render() {
    return (
        <div className="TodoItem">
            {this.props.name}
            <button className="TodoItem-delete" onClick={this._handleDelete}>
                X
            </button>
        </div>
    );
}
```

JSX Events - Lifting State (Code pt2)

- The parent component renders all the todos in a `this.state.todos` array
- We'll pass a prop function that removes the todo that was clicked from that array

```
// components/TodoList.js
_deleteTodo = (id) => {
    const todos = this.state.todos.filter((todo) => {
        return todo.id !== id;
    });
    this.setState({ todos }); // Same list, minus the clicked todo
}

render() {
    return (
        <div className="TodoList">
            {this.state.todos.map((todo) => {
                return (
                    <Todo
                        id={todo.id}
                        name={todo.name}
                        onClickDelete={this._deleteTodo}
                    />
                );
            })}
        </div>
    )
}
```

Additional Reading

- React Docs - Handling Events
- React Docs - Lifting State Up (Advanced reading)