Using State in Class Components

(https://github.com/learn-co-curriculum/react-hooks-class-state) (https://github.com/learn-co-curriculum/react-hooks-class-state/issues/new)

Learning Goals

- Create an initial state object in a class component
- Use the `this.setState` method to update state

Introduction

Before React Hooks came along, using **state** in a component was only possible in class components. In this lesson, we'll see how state works inside of class components and a few of the similarities and differences with how it works in function components.

Initializing State

In a function component, any time we needed to add state variables to our components, we use the `useState` hook, like so:

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

function TodoList() {
  const [todos, setTodos] = useState([]);
  const [filter, setFilter] = useState("incomplete");

  const displayedTodos = todos
    .filter((todo) => todo.status === filter)
    .map((todo) => <li key={todo.id}>{todo.text}</li>);
```

```
  return <ul>{displayedTodos}</ul>;
}
```

In a class, we can't use Hooks! So instead of using the `useState` hook, we can create an initial state by adding a special property of `state` to our component instance, like so:

```
import React from "react";

class TodoList extends React.Component {
  // initialize state as an object
  state = {
    todos: [],
    filter: "incomplete",
  };

  render() {
    // access state by calling `this.state`
    const displayedTodos = this.state.todos
      .filter((todo) => todo.status === this.state.filter)
      .map((todo) => <li key={todo.id}>{todo.text}</li>);

    return <ul>{displayedTodos}</ul>;
  }
}
```

In even older React code bases, you might see state initialized like this:

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    // initialize state as an object
    this.state = {
      todos: [],
```

```
      filter: "incomplete",
    };
  }

  render() {
    // access state by calling `this.state`
    const displayedTodos = this.state.todos
      .filter((todo) => todo.status === this.state.filter)
      .map((todo) => <li key={todo.id}>{todo.text}</li>);

    return <ul>{displayedTodos}</ul>;
  }
}
```

Both of these versions of initializing state will work; the first version uses a relatively new feature of Javascript called **public instance fields (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields#Public_instance_fields)**, but thanks to Babel we can safely use this new feature and still support older browsers!

You'll also notice that in the `render` method, we can access our state values by calling `this.state`. Just like with props, we need to use `this` to access data that is saved to our component.

Setting State

In our function components, we use the **setter** function returned by `useState` to update each state variable individually. For example, to update the filter variable from our example, we'd do:

```
setFilter("complete");
```

And to update the todos, we'd do:

```
setTodos([...todos, newTodo]);
```

In the class version of our component, there's no `setFilter` and no `setTodos` function... so how do we update state? For React class components, we get a special method from React to do just that: `this.setState()`.

When we use `this.setState`, we pass in an **object** with the key in state we're trying to update, like this:

```
this.setState({
  filter: "Incomplete",
});
```

Or this:

```
this.setState({
  todos: [...this.state.todos, newTodo],
});
```

Notice that when using `setState`, even though our initial state looks like this:

```
state = {
  todos: [],
  filter: "incomplete",
};
```

When setting state, the object passed to `setState` *only* needs the keys that we're trying to update (no need for a fancy spread operator or anything). This is because **state updates are merged (https://reactjs.org/docs/state-and-lifecycle.html#state-updates-are-merged)**.

Also, just like when we use the setter function from `useState`, calling `setState` will not only update the values for our component's state, it will also cause our component (and its children) to **re-render**.

Setting State is Asynchronous

As noted when we learned about the `useState` hook, setting state is **asynchronous**, which means that state is not immediately updated when we call the setter function:

```
function increment() {
  console.log(`before setState: ${count}`);
  // => 0
  setCount(count + 1);
  console.log(`after setState once: ${count}`);
  // => 0
  setCount(count + 1);
  console.log(`after setState twice: ${count}`);
  // => 0
}
```

The same is true when using `this.setState`:

```
increment() {
  console.log(`before setState: ${this.state.count}`);
  // => 0
  this.setState({
    count: this.state.count + 1
  });
  console.log(`after setState once: ${this.state.count}`);
  // => 0
  this.setState({
    count: this.state.count + 1
  });
  console.log(`after setState twice: ${this.state.count}`);
  // => 0
}
```

However, if we pass a **callback function** to `setState` instead of passing an object, we can access the current value of state between each function call:

```
  increment() {
    this.setState(prevState => {
      console.log(prevState.count) // => 0
      return { count: prevState.count + 1 }
    });
    this.setState(prevState => {
      console.log(prevState.count) // => 1
      return { count: prevState.count + 1 }
    });
  }
```

In this version of `setState`, we provide a callback function that will be called with the previous values of state for this component. The callback should return an object with any values from state that we want to update.

Conclusion

In class components, we can initialize state by creating a `state` object in the class. To update state, we must use the `this.setState` function with either an object, or a callback function that returns an object.

Resources

- **React setState (https://reactjs.org/docs/react-component.html#setstate)**