CHAPTER 11

Stateful Components

In this chapter, I introduce the stateful component, which builds on the features described in Chapter 10 and adds state data that is unique to each component and that can be used to alter the rendered output. Table 11-1 puts stateful components in context.

Table 11-1. *Putting Stateful Components in Context*

Question	Answer
What are they?	Components are the key building blocks in React applications. Stateful components have their own data that can be used to alter the content the component renders.
Why are they useful?	Stateful components make it easier to keep track of the application state provided by each component and provide the means to alter the data values and reflect the change in the content presented to the user.
How are they used?	Stateful components are defined using a class or by adding hooks to a functional component.
Are there any pitfalls or limitations?	Care must be taken to ensure that state data is modified correctly, as described in the "Modifying State Data" section of this chapter.
Are there any alternatives?	Components are the key building block in React applications, and there is no way to avoid their use. There are alternative to props that can be useful in larger and more complex projects, as described in later chapters.

Table 11-2 summarizes the chapter.

Table 11-2. *Chapter Summary*

Problem	Solution	Listing
Add state data to a component	Define a class whose constructor sets the state property or call the `useState` function to create a property and function for a single state property	4–5, 12, 13
Modify state data	Call the `setState` function or call the function returned by `useState`	6–11
Share data between components	Lift the state data to an ancestor component and distribute it using props	14–18
Define prop types and default values in a class-based component	Apply the properties to the class or define static properties within the class	19–20

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_11

Preparing for This Chapter

In this chapter, I continue using the components project created in Chapter 10. To prepare for this chapter, I changed the content rendered by the Summary component so that it uses the SimpleButton component directly as shown in Listing 11-1, rather than the CallbackButton that I used to describe how props are distributed.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/Apress/pro-react-16.

Listing 11-1. Changing the Content in the Summary.js File in the src Folder

```
import React from "react";
//import { CallbackButton } from "./CallbackButton";
import { SimpleButton } from "./SimpleButton";

export function Summary(props) {
    return (
        <React.Fragment>
            <td>{ props.index + 1} </td>
            <td>{ props.name } </td>
            <td>{ props.name.length } </td>
            <td>
                <SimpleButton
                    className="btn btn-warning btn-sm m-1"
                    callback={ props.reverseCallback }
                    text={ `Reverse (${ props.name })`}
                />
                <SimpleButton
                    className="btn btn-info btn-sm m-1"
                    callback={ () => props.promoteCallback(props.name)}
                    text={ `Promote (${ props.name})`}
                />
            </td>
        </React.Fragment>
    )
}
```

In Listing 11-2, I have removed the types and default values for the SimpleButton component's props, which I will restore at the end of the chapter.

Listing 11-2. Removing Properties in the SimpleButton.js File in the src Folder

```
import React from "react";

export function SimpleButton(props) {
    return (
```

```
        <button onClick={ props.callback } className={props.className}
                disabled={ props.disabled === "true" || props.disabled === true }>
            { props.text}
        </button>
    )
}
```

Open a command prompt, navigate to the `components` folder, and run the command shown in Listing 11-3 to start the React development tools.

Listing 11-3. Starting the Development Tools

```
npm start
```

After the initial build process, a new browser window will open and display the contents shown in Figure 11-1.

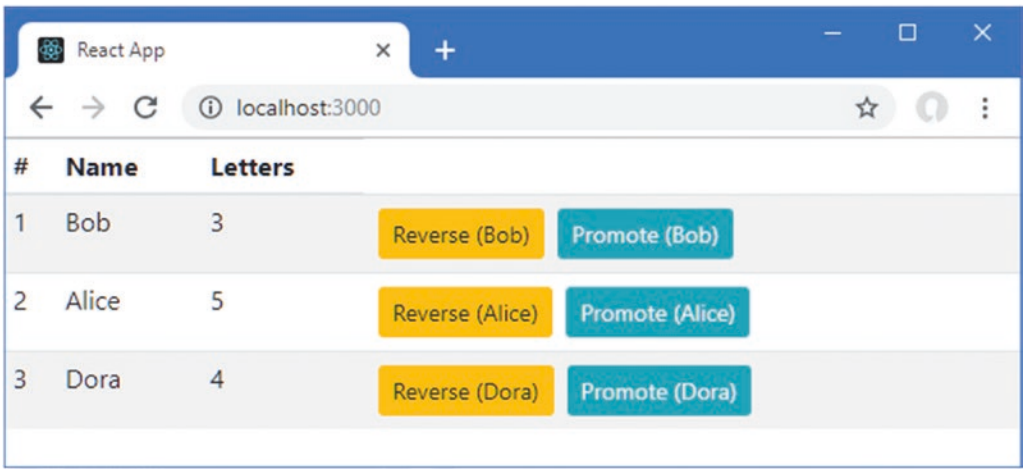

Figure 11-1. *Running the example application*

Understanding the Different Component Types

In the sections that follow, I explain the differences between the types of component that React supports. Understanding how stateful components work will be easier when you see the key difference from the stateless components described in Chapter 10.

Understanding Stateless Components

As you saw in Chapter 10, stateless components consist of a function that React invokes in response to custom HTML elements, passing the prop values as an argument. The same set of prop values on the custom HTML element will result in the same prop argument and produce the same result, as shown in Figure 11-2.

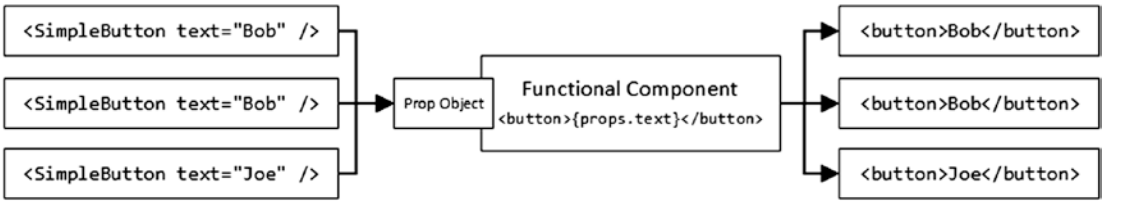

Figure 11-2. *Predictable results from a stateless component*

A stateless component will always render the same HTML elements given the same set of prop values, regardless of how often the function is invoked. It depends entirely on the prop values provided by the parent component to render its content. This means that React can keep invoking the same function regardless of how many `SimpleButton` elements there are in the application and just has to keep track of which props are associated with each `SimpleButton` element.

Understanding Stateful Components

A *stateful component* has its own data that influences the content the component renders. This data, which is known as *state data*, is separate from the parent component and the props it provides.

Imagine that the `SimpleButton` component has to keep track of how many times the user has clicks the `button` element it renders and displays the current count as the element's content. To provide this feature, the component needs a counter that is incremented each time the button is clicked and must include the current value of the counter when it renders its content.

Each `SimpleButton` element defined by the parent component will produce a `button` element for which a separate counter is required since each button can be clicked independently of the others. Stateful components are JavaScript objects, and there is a one-to-one relationship between the `SimpleButton` HTML element that applies the component and the component object, each of which has its own state and may render different output, as shown in Figure 11-3.

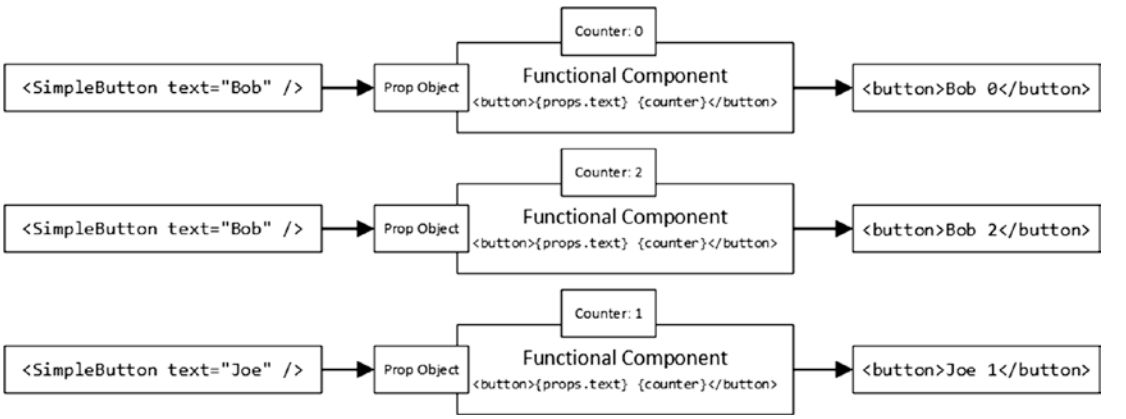

Figure 11-3. *Stateful components with a counter*

There is no longer any certainty that providing the same prop to a stateful component will render the same result because each component object can have different values for its state data and use it to generate different results.

As you will learn, stateful components have many features that are not available in stateless components, and you will find that these features are easier to understand if you remember that each stateful component is a JavaScript object with its own state data and is associated with a single custom HTML element.

Creating a Stateful Component

To get started, I am going to convert one the existing `SimpleButton` component in the example application from a stateless to stateful component, which will let me explain the basics before moving on to more complicated features.

Defining a stateful component is done using a class, which is a template that describes the functionality that each component object will have, as described in Chapter 4. In Listing 11-4, I have replaced the `SimpleButton` component's function with a class.

■ **Note** This is a stateful component that doesn't have any state data. I explain how to define the component and then show you how to add state data in the "Adding State Data" section.

Listing 11-4. Introducing a Class in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {

    render() {
        return (
            <button onClick={ this.props.callback }
                    className={ this.props.className }
                    disabled={ this.props.disabled === "true"
                        || this.props.disabled === true  }>
                { this.props.text}
            </button>
        )
    }
}
```

In the sections that follow, I describe each of the changes made in Listing 11-4 and explain how they are used to create a stateful component.

Understanding the Component Class

When you define a stateful component, you use the `class` and `extends` keywords to denote a class that inherits the functionality provided by the `Component` class defined in the `react` package, like this:

```
...
export class SimpleButton extends Component {
...
```

This combination of keywords defines a class called SimpleButton that extends the Component class provided by React. The export keyword makes the SimpleButton class available for use outside of the JavaScript file in which it is defined, just as it did when the component was defined as a function.

Understanding the Import Statement

To extend from the Component class, an import is used, as follows:

```
...
import React, { Component } from "react";
...
```

As I explained in Chapter 4, there are two types of import in this statement. The default export from the react package is imported and assigned the name React, which allows JSX to work. The react package also has an export named Component that is imported using curly braces (the { and } characters). It is important that you use the import statement exactly as shown when you create a stateful component.

Understanding the render Method

The main purpose of a stateful component is to render content for React to display. The difference is that this is done in a method called render, which is invoked when React wants the component to render. The render method must return a React element, which can be created using the React.createElement method or, more typically, as a fragment of HTML.

```
...
render() {
    return (
        <button onClick={ this.props.callback }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                    || this.props.disabled === true  }>
            { this.props.text}
        </button>
    )
}
...
```

Understanding Stateful Component Props

One of the most noticeable differences when you start working with stateful components is that you must use the this keyword to access prop values, as follows:

```
...
return (
    <button onClick={ this.props.callback }
            className={ this.props.className }
            disabled={ this.props.disabled === "true"
                || this.props.disabled === true  }>
```

```
        { this.props.text}
    </button>
)
...
```

The `this` keyword refers to the component's JavaScript object. When using a stateful component, you must use the `this` keyword to access the `props` property, and you will see an error like this one displayed on the command line, the browser's JavaScript console, and the browser window if you forget:

```
./src/SimpleButton.js  Line 7:  'props' is not defined  no-undef
```

Although I have redefined the component, I haven't changed the content that it renders or changed the way it behaves, and the result is just the same as when the component was defined as a function, as shown in Figure 11-4.

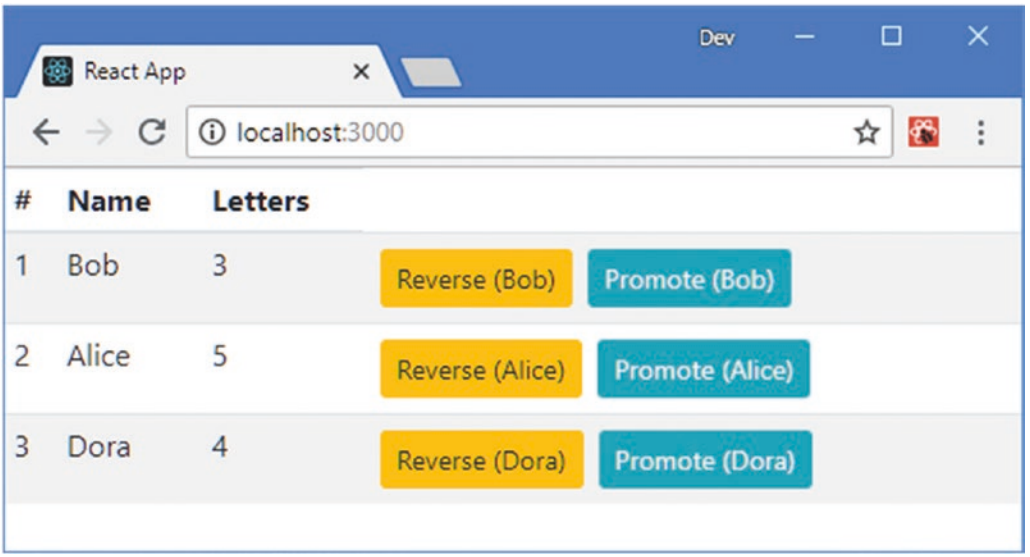

Figure 11-4. *Introducing a stateful component*

Adding State Data

The most import feature of stateful components is that each instance of the component can have its own data, known as *state data*. In Listing 11-5, I have added state data to the `SimpleButton` component.

Listing 11-5. Adding State Data in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {

    constructor(props) {
        super(props);
```

```
        this.state = {
            counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.props.callback }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                    { this.props.text} { this.state.counter }
                    { this.state.hasButtonBeenClicked &&
                        <div>Button Clicked!</div>
                    }
            </button>
        )
    }
}
```

State data is defined using a *constructor*, which is a special method that is invoked when a new object is created using the class and that must follow the form shown in the listing: the constructor should define a `props` parameter, and the first statement should be a call to the special `super` method using the `props` object as an argument, which invokes the constructor of the `Component` class and sets up the features available in a stateful component.

Once you have called `super`, you can define the state data, which is done by assigning an object to `this.state`.

```
...
constructor(props) {
    super(props);
    this.state = {
        counter: 0,
        hasButtonBeenClicked: false
    }
}
...
```

The state data is defined as properties on the object. There is one property in this example, and it creates state data properties called `counter`, whose value is `0`, and `hasButtonBeenClicked`, whose value is `false`.

Reading State Data

Accessing state data is done by reading the properties you have defined through `this.state`, similar to the way that props are accessed.

```
...
render() {
    return (
        <button onClick={ this.props.callback }
```

```
            className={ this.props.className }
            disabled={ this.props.disabled === "true"
                       || this.props.disabled === true  }>
                { this.props.text} { this.state.counter }
                { this.state.hasButtonBeenClicked &&
                    <div>Button Clicked!</div>
                }
        </button>
    )
}
...
```

The `render` method in Listing 11-5 sets the contents of the `button` element so that it contains a prop value and the value of the `counter` state data property, producing the effect shown in Figure 11-5. The additional `div` element I defined in Listing 11-5 won't be shown until the value of the `hasButtonBeenClicked` property is `true`, which I demonstrate in the next section.

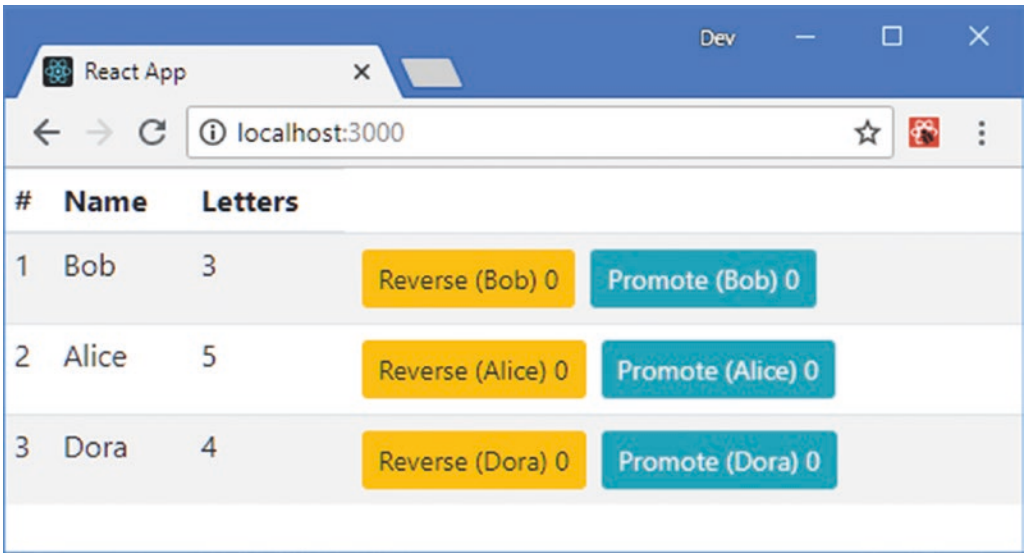

Figure 11-5. *Defining and reading state data*

Modifying State Data

The use of state data makes sense only when it can be modified because that's what allows component objects to render different content. React requires a specific technique for modifying state data, as shown in Listing 11-6.

Listing 11-6. Modifying State Data in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {
```

```
    constructor(props) {
        super(props);
        this.state = {
            counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                    { this.props.text} { this.state.counter }
                    { this.state.hasButtonBeenClicked &&
                        <div>Button Clicked!</div>
                    }
            </button>
        )
    }

    handleClick = () => {
        this.setState({
            counter: this.state.counter + 1,
            hasButtonBeenClicked: true
        });
        this.props.callback();
    }
}
```

React doesn't allow state data to be modified directly and will report an error if you try to assign a new value directly to a state property. Instead, modifications are made through the `setState` method, which is inherited from the `Component` class. In the listing, I have added a method called `handleClick` that is selected by the `button` element's `onClick` expression and that uses the `setState` method to increment the `counter` state property.

■ **Tip** Methods that are selected by the `onClick` property have to be defined in a specific way. I explain how the `onClick` property is used and how its methods are defined in Chapter 12.

The argument to the `setState` method is an object whose properties specify the state data to be updated, like this:

```
...
this.setState({
    counter: this.state.counter + 1,
    hasButtonBeenClicked: true
});
...
```

This statement tells React that the `counter` property should be modified by incrementing the current value and that the `hasButtonBeenClicked` property should be `true`. Notice that I have not used the increment operator (`++`) for `counter` because that would assign a new value to the property and result in an error.

■ **Tip** You only have to define properties for the values you want to change when using the `setState` method. React will merge the changes you specify with the rest of the component's state data and leave unchanged any property for which a value has not been provided.

Although using the `setState` method can feel awkward, the advantage is that React takes care of re-rendering the application to reflect the impact of the change, which means that I don't have manually invoke the `ReactDOM.render` method as I did in Chapter 11. The effect is that clicking the buttons increments the associated component's counter state data, as shown in Figure 11-6. (Clicking the buttons reorders the rows in the table, which means that the button you have clicked may be moved to a new position.)

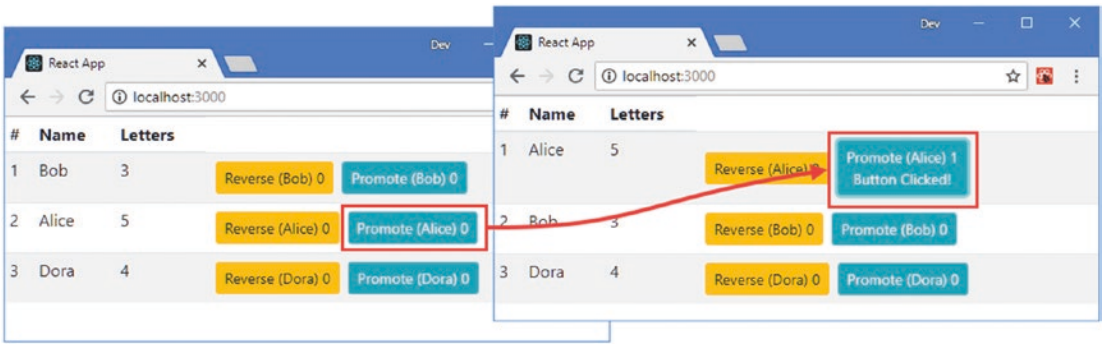

Figure 11-6. *Modifying state data*

Clicking a button changes the state of one of the component objects and leaves the other five component objects unchanged.

Avoiding the State Data Modification Pitfalls

React performs changes to state data asynchronously and may choose to group together several updates to improve performance, which means that the effect of a call to the `setState` may not take effect in the way you expect. There are some common pitfalls when updating state data, which I describe in the sections that follow, along with details of how to avoid them.

■ **Tip** The React Devtools browser extension shows you the state data for a stateful component, which can be a useful way of seeing how the application responds to changes and tracking down problems when you don't get the behavior you expect.

Avoiding the Dependent Value Pitfall

State data values are often related, and a common problem is to assume that the effect of each change is applied individually, as shown in Listing 11-7.

Listing 11-7. Performing Related State Changes in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                    { this.props.text} { this.state.counter }
                    { this.state.hasButtonBeenClicked &&
                        <div>Button Clicked!</div>
                    }
            </button>
        )
    }

    handleClick = () => {
        this.setState({
            counter: this.state.counter + 1,
            hasButtonBeenClicked: this.state.counter > 0
        });
        this.props.callback();
    }
}
```

The update to the `hasButtonBeenClicked` property assumes the `counter` property will have been changed before its expression is evaluated. React doesn't apply changes individually, and the expression for the `hasButtonBeenClicked` property is evaluated using the current `counter` value. This problem also arises when related updates are performed using separate calls to the `setState` method, as shown in Listing 11-8.

Listing 11-8. Making Dependent Updates in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {
```

```
    constructor(props) {
        super(props);
        this.state = {
            counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                    { this.props.text} { this.state.counter }
                    { this.state.hasButtonBeenClicked &&
                        <div>Button Clicked!</div>
                    }
            </button>
        )
    }

    handleClick = () => {
        this.setState({ counter: this.state.counter + 1 });
        this.setState({ hasButtonBeenClicked: this.state.counter > 0 });
        this.props.callback();
    }
}
```

React will batch these updates together for efficiency, which creates the same result as Listing 11-6 and means that the `hasButtonBeenClicked` property won't be `true` until the button has been clicked twice, as shown in Figure 11-7.

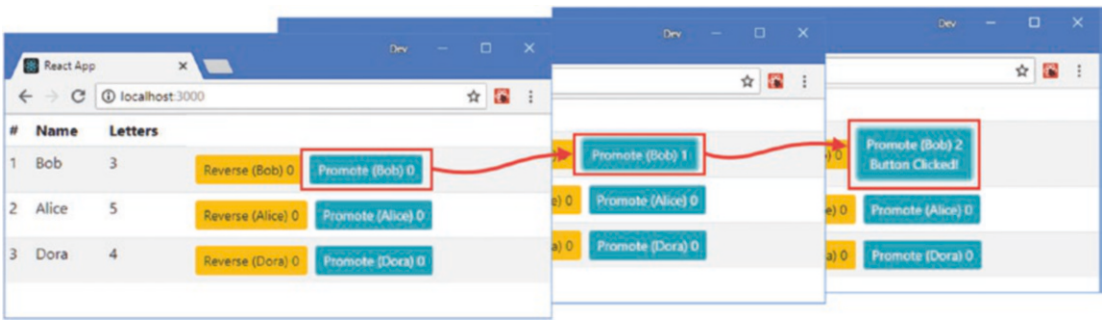

Figure 11-7. *The dependent value pitfall*

When you have a series of dependent changes to make, you can pass a function to the `setState` method that will be invoked when the state data has been updated and that can be used to perform tasks that rely on the changed state values, as shown in Listing 11-9.

Listing 11-9. Using a Callback in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                    { this.props.text} { this.state.counter }
                    { this.state.hasButtonBeenClicked &&
                        <div>Button Clicked!</div>
                    }
            </button>
        )
    }

    handleClick = () => {
        this.setState({ counter: this.state.counter + 1 },
            () => this.setState({ hasButtonBeenClicked: this.state.counter > 0 }));
        this.props.callback();
    }
}
```

Using the callback function ensures that the value of the `hasButtonBeenClicked` value won't be changed until the new `counter` property has been applied, ensuring that the values are in sync, as shown in Figure 11-8.

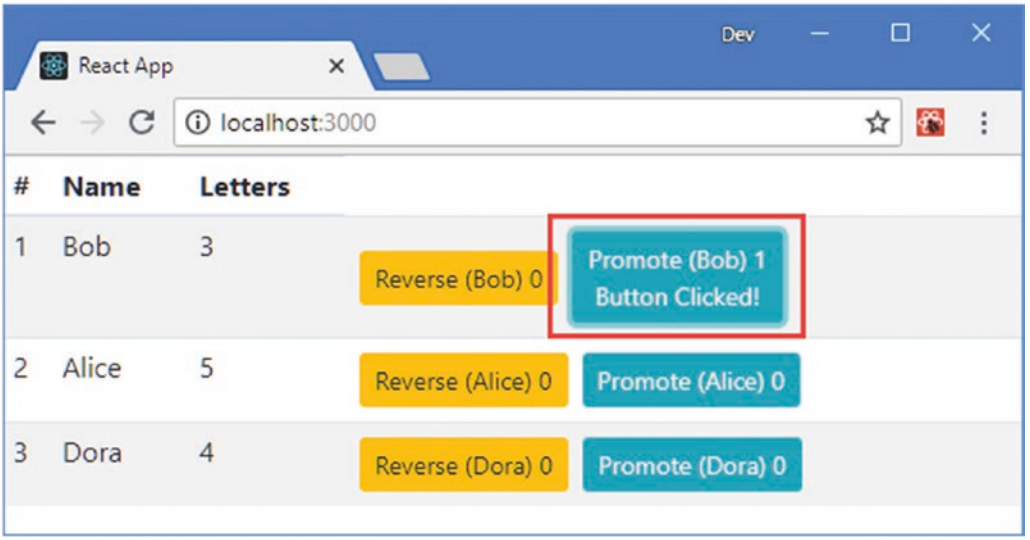

Figure 11-8. *Forcing state changes to be performed in sequence*

Avoiding the Missing Updates Pitfall

The way that React applies updates means that multiple changes to the same state data property are ignored and only the most recent value is applied, as demonstrated in Listing 11-10.

Listing 11-10. Making Multiple Updates in the SimpleButton.js File in the src Folder

```
...
handleClick = () => {
    for (let i = 0; i < 5; i++) {
        this.setState({ counter: this.state.counter + 1});
    }
    this.setState({ hasButtonBeenClicked: true });
    this.props.callback();
}
...
```

In real projects, multiple updates are usually done while processing data, rather than in a `for` loop, so that a state change is performed for each object in an array, for example. This listing shows the effect of repeatedly modifying the same property: rather than incrementing the `counter` value five times, clicking a button increments the value by one, as shown in Figure 11-9.

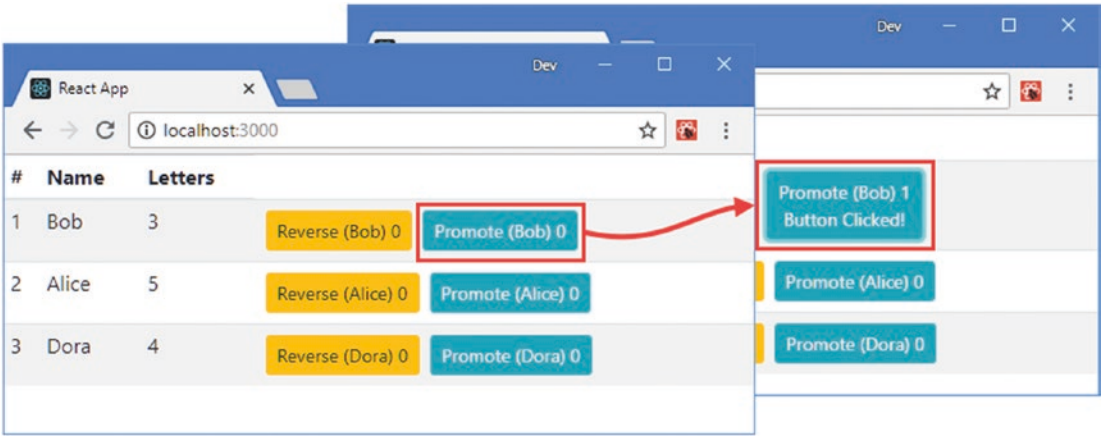

Figure 11-9. *Applying multiple updates to a state property*

If you need to perform multiple updates and have each take effect in sequence, then you can use the version of the `setState` method that accepts a function as its first argument. The function is provided with the current state data and a props object, as shown in Listing 11-11.

■ **Tip** This version of the `setState` method is also useful for updating nested state properties, which you can see demonstrated in Chapter 14.

Listing 11-11. Making Multiple Updates in the SimpleButton.js File in the src Folder

```
...
handleClick = () => {
    for (let i = 0; i < 5; i++) {
        this.setState((state, props) => { return { counter: state.counter + 1 }});
    }
    this.setState({ hasButtonBeenClicked: true });
    this.props.callback();
}
...
```

The function passed to the `setState` method returns an update object using the same format as earlier examples. The difference is that the state data object reflects all of the previous changes that have been grouped together and can be used for repeated updates, producing the effect shown in Figure 11-10.

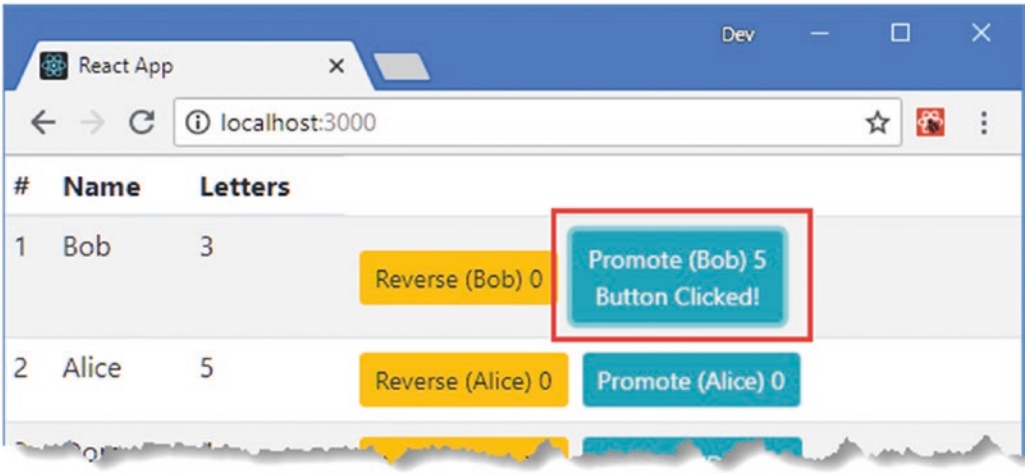

Figure 11-10. *Applying multiple updates to a state property*

Defining Stateful Components Using Hooks

Not all developers like using classes to define stateful components and so React provides an alternative approach, called *hooks*, which allow functional components to define state data. In Listing 11-12, I added a file called `HooksButton.js` to the `src` folder and re-created the stateful component from Listing 11-11 as a function that uses hooks.

Listing 11-12. The Contents of the HooksButton.js File in the src Folder

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

export function HooksButton(props) {
    const [counter, setCounter] = useState(0);
    const [hasButtonBeenClicked, setHasButtonBeenClicked] = useState(false);

    const handleClick = () => {
        setCounter(counter + 5);
        setHasButtonBeenClicked(true);
        props.callback();
    }

    return (
        <button onClick={ handleClick }
            className={ props.className }
            disabled={ props.disabled === "true" || props.disabled === true  }>
                { props.text} { counter }
                { hasButtonBeenClicked && <div>Button Clicked!</div>}
        </button>
    )
}
```

The useState function is used to create state data. Its argument is the initial value for the state data property, and it returns a property that provides the current value and a function that changes the value and triggers an update. The property and the function are returned in an array and are assigned meaningful names using array destructuring, like this:

```
...
const [counter, setCounter] = useState(0);
...
```

This statement creates a state data property named counter whose initial value is zero and whose value can be changed using a function named setCounter. The function used to change the value of a state data property doesn't have all of the features of the setState method, which is why I have incremented the value by five in the handleClick function, rather than performing a series of individual updates, as in Listing 11-11.

```
...
const handleClick = () => {
    setCounter(counter + 5);
    setHasButtonBeenClicked(true);
    props.callback();
}
...
```

In Listing 11-13, I have updated the Summary so it uses the HooksButton component.

Listing 11-13. Using the Hooks Component in the Summary.js File in the src Folder

```
import React from "react";
import { SimpleButton } from "./SimpleButton";
import { HooksButton } from "./HooksButton";

export function Summary(props) {
    return (
        <React.Fragment>
            <td>{ props.index + 1} </td>
            <td>{ props.name } </td>
            <td>{ props.name.length } </td>
            <td>
                <SimpleButton
                    className="btn btn-warning btn-sm m-1"
                    callback={ props.reverseCallback }
                    text={ `Reverse (${ props.name })`} />
                <HooksButton
                    className="btn btn-info btn-sm m-1"
                    callback={ () => props.promoteCallback(props.name)}
                    text={ `Promote (${ props.name})`} />
            </td>
        </React.Fragment>
    )
}
```

The use of hooks is not visible to the Summary component, which provides data and functions via props as normal. This example produces the same result, as shown in Figure 11-10.

SHOULD YOU USE HOOKS OR CLASSES?

Hooks offer an alternative approach to creating stateful components for developers who don't like to use classes. Depending on your personal preference, either this will be an important feature that suits your coding style or you will carry on defining classes and forget about hooks entirely.

The hooks and classes features will both be supported in future versions of React and so you can use whichever suits you best or mix and match freely if you prefer. I like the hooks features, but, aside from describing some related hooks features in Chapter 13, all of the examples in this book use classes. In part that is because the hooks feature is new—but it is also because I have been using class-based programming languages for a long time and using classes to define components suits my way of thinking about code, even for simple stateless components.

If you prefer using hooks but can't work out how to express the book examples without using a class, then e-mail me at adam@adam-freeman.com, and I will try to point you in the right direction.

Lifting Up State Data

At the moment, each SimpleButton and HooksButton component exists in isolation and has its own state data, so clicking a button affects only the state value of a single component and leaves the others unchanged.

A different approach is needed when components need access to the same data. In this situation, the state data is *lifted up*, which means it is moved to the first common ancestor component and distributed back down to the components that require it using props.

■ **Tip** There are alternative approaches available for sharing data between React components. Chapter 13 describes the context feature, and more complex projects can benefit from using a data store (see Chapters 19 and 20) or URL routing (see Chapters 21 and 22).

If I want the SimpleButton and HooksButton components in the same table row to share a counter value, for example, I need to define the state data property in the first common ancestor, which is the Summary component. In Listing 11-14, I have converted Summary to be a class-based stateful component that defines a counter value.

Listing 11-14. Lifting Up State Data in the Summary.js File in the src Folder

```
import React, { Component } from "react";
import { SimpleButton } from "./SimpleButton";
import { HooksButton } from "./HooksButton";
```

```
export class Summary extends Component {

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

    incrementCounter = (increment) => {
        this.setState((state) => { return { counter: state.counter + increment}});
    }

    render() {
        const props = this.props;
        return (
            <React.Fragment>
                <td>{ props.index + 1} </td>
                <td>{ props.name } </td>
                <td>{ props.name.length } </td>
                <td>
                    <SimpleButton
                        className="btn btn-warning btn-sm m-1"
                        callback={ props.reverseCallback }
                        text={ `Reverse (${ props.name })`}
                        counter={ this.state.counter }
                        incrementCallback={this.incrementCounter }
                    />
                    <HooksButton
                        className="btn btn-info btn-sm m-1"
                        callback={ () => props.promoteCallback(props.name)}
                        text={ `Promote (${ props.name})`}
                        counter={ this.state.counter }
                        incrementCallback={this.incrementCounter }
                    />
                </td>
            </React.Fragment>
        )
    }
}
```

The Summary component defines a counter property and passes it on to its child components as a prop. The component also defines an incrementCounter method that child components will invoke to change the counter property, which is passed on using a prop named incrementCallback. This is required not only because state data is not modified directly but also because props are read-only. The incrementCounter method uses the setState method with a function so that it can be invoked repeatedly by child components.

■ **Tip** I defined a props property in the render method so that I don't have to change all the references to use the this keyword, which is a useful shortcut when converting a function component to use a class.

In Listing 11-15, I removed the counter state data property from the SimpleButton component and used the counter and incrementCounter props instead.

Listing 11-15. Replacing State Data with Props in the SimpleButton.js File in the src Folder

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

export class SimpleButton extends Component {

    constructor(props) {
        super(props);
        this.state = {
            // counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                    { this.props.text} { this.props.counter }
                    { this.state.hasButtonBeenClicked &&
                        <div>Button Clicked!</div>
                    }
            </button>
        )
    }

    handleClick = () => {
        this.props.incrementCallback(5);
        this.setState({ hasButtonBeenClicked: true });
        this.props.callback();
    }
}
```

A corresponding set of changes is required to the HooksButton component, which will share the same set of props, as shown in Listing 11-16.

Listing 11-16. Replacing State Data with Props in the HooksButton.js File in the src Folder

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

export function HooksButton(props) {
    //const [counter, setCounter] = useState(0);
    const[ hasButtonBeenClicked, setHasButtonBeenClicked] = useState(false);
```

```
    const handleClick = () => {
        //setCounter(counter + 5);
        props.incrementCallback(5);
        setHasButtonBeenClicked(true);
        props.callback();
    }

    return (
        <button onClick={ handleClick }
            className={ props.className }
            disabled={ props.disabled === "true" || props.disabled === true  }>
                { props.text} { props.counter }
                { hasButtonBeenClicked && <div>Button Clicked!</div>}
        </button>
    )
}
```

Lifting the `counter` state property to the parent component means that the two buttons presented to the user in each table row share their parent's state data, such that clicking one of the button elements causes both to be updated, as shown in Figure 11-11.

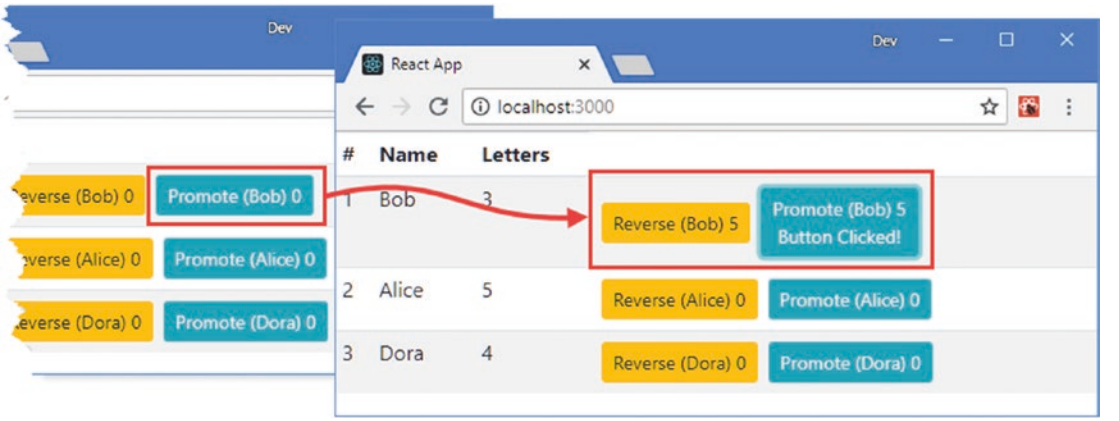

Figure 11-11. *Lifting state data*

Not every item of state data has to be lifted, and the individual components still have their own local state data, such that the `hasButtonBeenClicked` property remains local and independent from the other components.

Lifting Up State Data Further

State data can be lifted up further than the parent component. If I want all the `SimpleButton` and `HooksButton` components to share the same `counter` property, then I can lift it up to the `App` component, as shown in Listing 11-17, in which I have made the stateful using the hooks feature.

Listing 11-17. Lifting State Data in the App.js File in the src Folder

```
import React, { useState } from "react";
import { Summary } from "./Summary";
import ReactDOM from "react-dom";

let names = ["Bob", "Alice", "Dora"]

function reverseNames() {
    names.reverse();
    ReactDOM.render(<App />, document.getElementById('root'));
}

function promoteName(name) {
    names = [name, ...names.filter(val => val !== name)];
    ReactDOM.render(<App />, document.getElementById('root'));
}

export default function App() {
    const [counter, setCounter] = useState(0);

    const incrementCounter = (increment) => setCounter(counter + increment);

    return (
        <table className="table table-sm table-striped">
            <thead>
                <tr><th>#</th><th>Name</th><th>Letters</th></tr>
            </thead>
            <tbody>
                { names.map((name, index) =>
                    <tr key={ name }>
                        <Summary index={index} name={name}
                            reverseCallback={reverseNames}
                            promoteCallback={promoteName}
                            counter={ counter }
                            incrementCallback={ incrementCounter }
                        />
                    </tr>
                )}
            </tbody>
        </table>
    )
}
```

The App component defines the counter state property and the incrementCounter method that modifies it by calling the setCounter function. In Listing 11-18, I have removed the state data from the Summary component and passed on the props that are received from the App component to the children.

Listing 11-18. Removing State Data in the Summary.js File in the src Folder

```
import React, { Component } from "react";
import { SimpleButton } from "./SimpleButton";
import { HooksButton } from "./HooksButton";

export class Summary extends Component {

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

    // incrementCounter = (increment) => {
    //     this.setState((state) => { return { counter: state.counter + increment}});
    // }

    render() {
        const props = this.props;
        return (
            <React.Fragment>
                <td>{ props.index + 1} </td>
                <td>{ props.name } </td>
                <td>{ props.name.length } </td>
                <td>
                    <SimpleButton
                        className="btn btn-warning btn-sm m-1"
                        callback={ props.reverseCallback }
                        text={ `Reverse (${ props.name })`}
                        { ...this.props }
                    />
                    <HooksButton
                        className="btn btn-info btn-sm m-1"
                        callback={ () => props.promoteCallback(props.name)}
                        text={ `Promote (${ props.name})`}
                        { ...this.props }
                    />
                </td>
            </React.Fragment>
        )
    }
}
```

No constructor is required when a stateful component doesn't have state data, and you will receive a warning if you define a constructor that doesn't do anything except pass on props to the base class using `super`. I used the destructuring operator to pass on the props received from the `App` component to the `SimpleButton` and `HooksButton` components.

Now that the state data has been lifted up to the `App` component, all of the `SimpleButton` components that are descendants of the `App` component share a `counter` value, as shown in Figure 11-12.

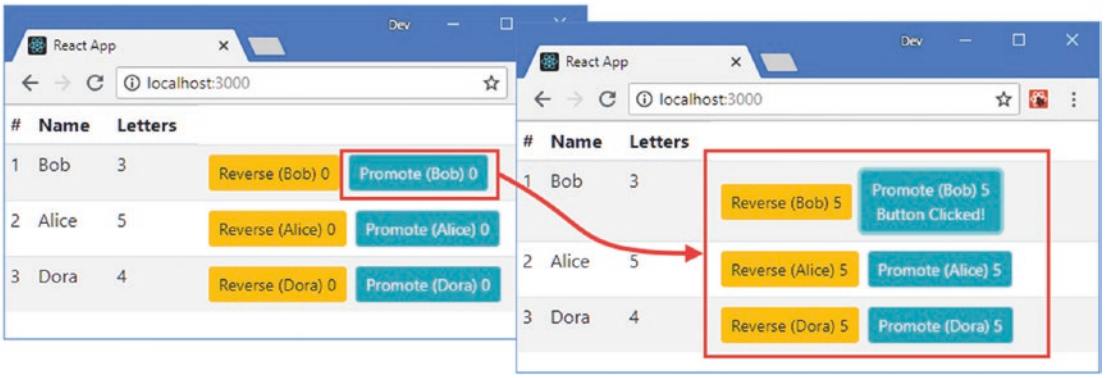

Figure 11-12. *Lifting state data to the top-level component*

No changes are required to the `SimpleButton` and `HooksButton` components, which are unaware of where the state data is defined and receive the data values and the callback functions required to change it as props.

Defining Prop Types and Default Values

At the start of the chapter, I removed the prop default values and types so I could focus on the transition from stateless to stateful components. Class-based components support these features in the same way as functional components, as shown in Listing 11-19.

Listing 11-19. Adding Prop Types and Values in the SimpleButton.js File in the src Folder

```
import React, { Component } from "react";
import PropTypes from "prop-types";

export class SimpleButton extends Component {

    constructor(props) {
        super(props);
        this.state = {
            // counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
                { this.props.text} { this.props.counter }
                { this.state.hasButtonBeenClicked &&
                    <div>Button Clicked!</div>
                }
```

```
            </button>
        )
    }

    handleClick = () => {
        this.props.incrementCallback(5);
        this.setState({ hasButtonBeenClicked: true });
        this.props.callback();
    }

}

SimpleButton.defaultProps = {
    disabled: false
}

SimpleButton.propTypes = {
    text: PropTypes.string,
    theme: PropTypes.string,
    callback: PropTypes.func,
    disabled: PropTypes.oneOfType([PropTypes.bool, PropTypes.string ])
}
```

You can also define types and default prop values using class properties that have been decorated with the `static` keyword, as shown in Listing 11-20. The `static` keyword defines a property that applies to the component's class rather than objects created from that class and is transformed by the build process into the same form used in Listing 11-19.

Listing 11-20. Defining Static Properties in the SimpleButton.js File in the src Folder

```
import React, { Component } from "react";
import PropTypes from "prop-types";

export class SimpleButton extends Component {

    constructor(props) {
        super(props);
        this.state = {
            // counter: 0,
            hasButtonBeenClicked: false
        }
    }

    render() {
        return (
            <button onClick={ this.handleClick }
                className={ this.props.className }
                disabled={ this.props.disabled === "true"
                            || this.props.disabled === true  }>
```

```
                { this.props.text} { this.props.counter }
                { this.state.hasButtonBeenClicked &&
                    <div>Button Clicked!</div>
                }
            </button>
        )
    }

    handleClick = () => {
        this.props.incrementCallback(5);
        this.setState({ hasButtonBeenClicked: true });
        this.props.callback();
    }

    static defaultProps = {
        disabled: false
    }

    static propTypes = {
        text: PropTypes.string,
        theme: PropTypes.string,
        callback: PropTypes.func,
        disabled: PropTypes.oneOfType([PropTypes.bool, PropTypes.string ])
    }
}
```

These changes don't alter the appearance of the example application, but they ensure that the component will receive only the prop types it expects and that a default value for the `disabled` prop is available.

Summary

In this chapter, I introduced the stateful component, which has its own data values that can be used to alter the rendered output. I explained that stateful components are defined using classes and showed you how to define state data in a constructor. I also showed you the different ways that state data can be modified and how to avoid the most common pitfalls. In the next chapter, I explain how React deals with events.