CHAPTER 13

Reconciliation and Lifecycles

In this chapter, I explain how React uses a process called *reconciliation* to efficiently deal with content produced by components. The reconciliation process is part of a larger lifecycle that React provides for components, and I describe the different lifecycle stages and show you how stateful components can implement methods to become active lifecycle participants. Table 13-1 puts reconciliation and the component lifecycle in context.

Table 13-1. *Putting Reconciliation and Lifecycle Text in Context*

Question	Answer
What is it?	Reconciliation is the process of efficiently handling the content produced by components to minimize changes to the Document Object Model (DOM). Reconciliation is part of a larger lifecycle that is applied to stateful components.
Why is it useful?	The reconciliation process helps application performance, while the broader component lifecycle provides a consistent model for application development and provides useful features for advanced projects.
How is it used?	The reconciliation process is performed automatically, and no explicit action is required. All stateful components go through the same lifecycle and can participate actively by implementing specific methods (for class-based components) or the effect hook (for functional components).
Are there any pitfalls or limitations?	Care must be taken to write components so they fit into the overall lifecycle, which includes being able to render content even though it may not be used to update the DOM.
Are there any alternatives?	No, the lifecycle and the reconciliation process are fundamental React features.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_13

Table 13-2 summarizes the chapter.

Table 13-2. *Chapter Summary*

Problem	Solution	Listing
Trigger reconciliation	Call the `forceUpdate` method	15, 16
Respond to lifecycle stages	Implement the method that corresponds to the lifecycle stage	17–20
Receive notifications in a functional component	Use the effect hook	21–23
Prevent updates	Implement the `shouldComponentUpdate` method	24, 25
Set state data from props	Implement the `getDerivedStateFromProps` method	26, 27

Preparing for This Chapter

To create the example project for this chapter, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 13-1.

■ **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 13-1. Creating the Example Project

```
npx create-react-app lifecycle
```

Run the commands shown in Listing 13-2 to navigate to the `lifecycle` folder and add the Bootstrap package to the project.

Listing 13-2. Adding the Bootstrap CSS Framework

```
cd lifecycle
npm install bootstrap@4.1.2
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 13-3 to the `index.js` file, which can be found in the `src` folder.

Listing 13-3. Including Bootstrap in the index.js File in the src Folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import 'bootstrap/dist/css/bootstrap.css';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Creating the Example Components

Some basic components are needed for the examples in this chapter. Add a file called `ActionButton.js` to the `src` folder and add the content shown in Listing 13-4.

Listing 13-4. The Contents of the ActionButton.js File in the src Folder

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

export class ActionButton extends Component {

    render() {
        console.log(`Render ActionButton (${this.props.text}) Component `);
        return <button className="btn btn-primary m-2"
                        onClick={ this.props.callback }>
                            { this.props.text }
                </button>
    }
}
```

This component renders a button that invokes a function prop in response to the `click` event. Next, add a file called `Message.js` to the `src` folder and add the content shown in Listing 13-5.

Listing 13-5. The Contents of the Message.js File in the src Folder

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

export class Message extends Component {

    render() {
        console.log(`Render Message Component `);
        return (
            <div>
                <ActionButton theme="primary"  {...this.props} />
                <div className="h5 text-center p-2">
                    { this.props.message }
                </div>
            </div>
        )
    }
}
```

This component displays a message received as a prop and passes on a function prop as the callback for an `ActionButton`, as defined in Listing 13-4. Next, add a file called `List.js` to the `src` folder and add the content shown in Listing 13-6.

Listing 13-6. The Contents of the List.js File in the src Folder

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

export class List extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Bob", "Alice", "Dora"]
        }
    }

    reverseList = () => {
        this.setState({ names: this.state.names.reverse()});
    }

    render() {
        console.log("Render List Component");
        return (
            <div>
                <ActionButton callback={ this.reverseList }
                    text="Reverse Names" />
                { this.state.names.map((name, index) => {
                    return <h5 key={ name }>{ name }</h5>
                })}

            </div>
        )
    }
}
```

This component has its own state data, which it uses to render a list. An `ActionButton` component is provided with the `reverseList` method as its function prop, which reverses the order of the items in the list.

The final change is to replace the contents of the `App.js` file with the code shown in Listing 13-7, which renders content that uses the other components and defines the state data that the `Message` component requires.

Listing 13-7. The Contents of the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
import { List } from "./List";

export default class App extends Component {

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

    incrementCounter = () => {
        this.setState({ counter: this.state.counter + 1 });
    }

    render() {
        console.log("Render App Component");
        return  <div className="container text-center">
                    <div className="row p-2">
                        <div className="col-6">
                            <Message message={ `Counter: ${this.state.counter}`}
                                callback={ this.incrementCounter }
                                text="Increment Counter" />
                        </div>
                        <div className="col-6">
                            <List />
                        </div>
                    </div>
                </div>

    }
}
```

The content rendered by the `App` component displays the `Message` and `List` components side by side using the Bootstrap CSS grid features. The `counter` property is incremented by the `incrementCounter` method, which is used as the function prop for the `Message` component. Using the command prompt, run the command shown in Listing 13-8 in the `lifecycle` folder to start the development tools.

Listing 13-8. Starting the Development Tools

```
npm start
```

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000`, which will display the content shown in Figure 13-1.

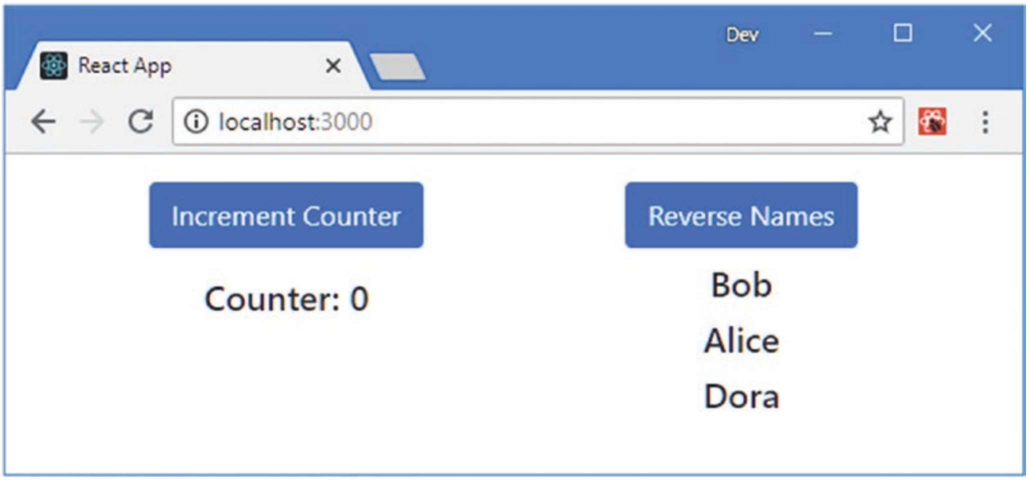

Figure 13-1. *Running the example application*

Understanding How Content Is Rendered

The starting point for the rendering process is the statement in the `index.js` file that invokes the `ReactDOM.render` method.

```
...
ReactDOM.render(<App />, document.getElementById('root'));
...
```

This method starts the initial rendering process. React creates a new instance of the `App` component, which is specified by the first argument to the `ReactDOM.render` method, and invokes its `render` method. The content rendered by the `App` component includes `Message` and `List` elements, and React creates instances of these components and calls their `render` methods. The process continues to the `ActionButton` elements in the content rendered by the `Message` and `List` elements, creating two instances of the `ActionButton` component and calling the `render` method for each of them. The result of calling the `render` method on each component is a hierarchy of HTML elements that are inserted into the element selected by the second argument to the `ReactDOM.render` method, creating the content shown in Figure 13-1. The result of the initial rendering process is a hierarchy of component objects and HTML elements, as shown in Figure 13-2.

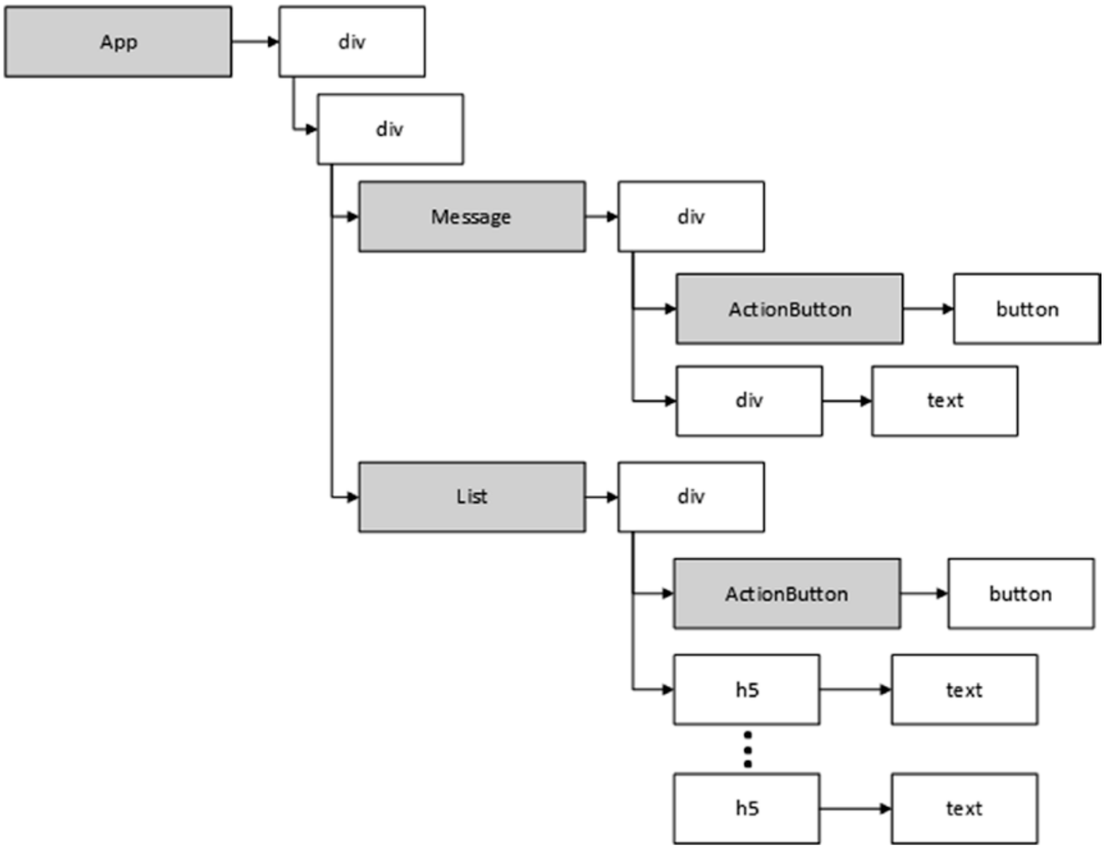

Figure 13-2. *Components and their content*

React uses the browser's API to add HTML elements to the Document Object Model (DOM) so they can be presented to user, as shown in Figure 13-3, and creates a mapping between the components and the content they render.

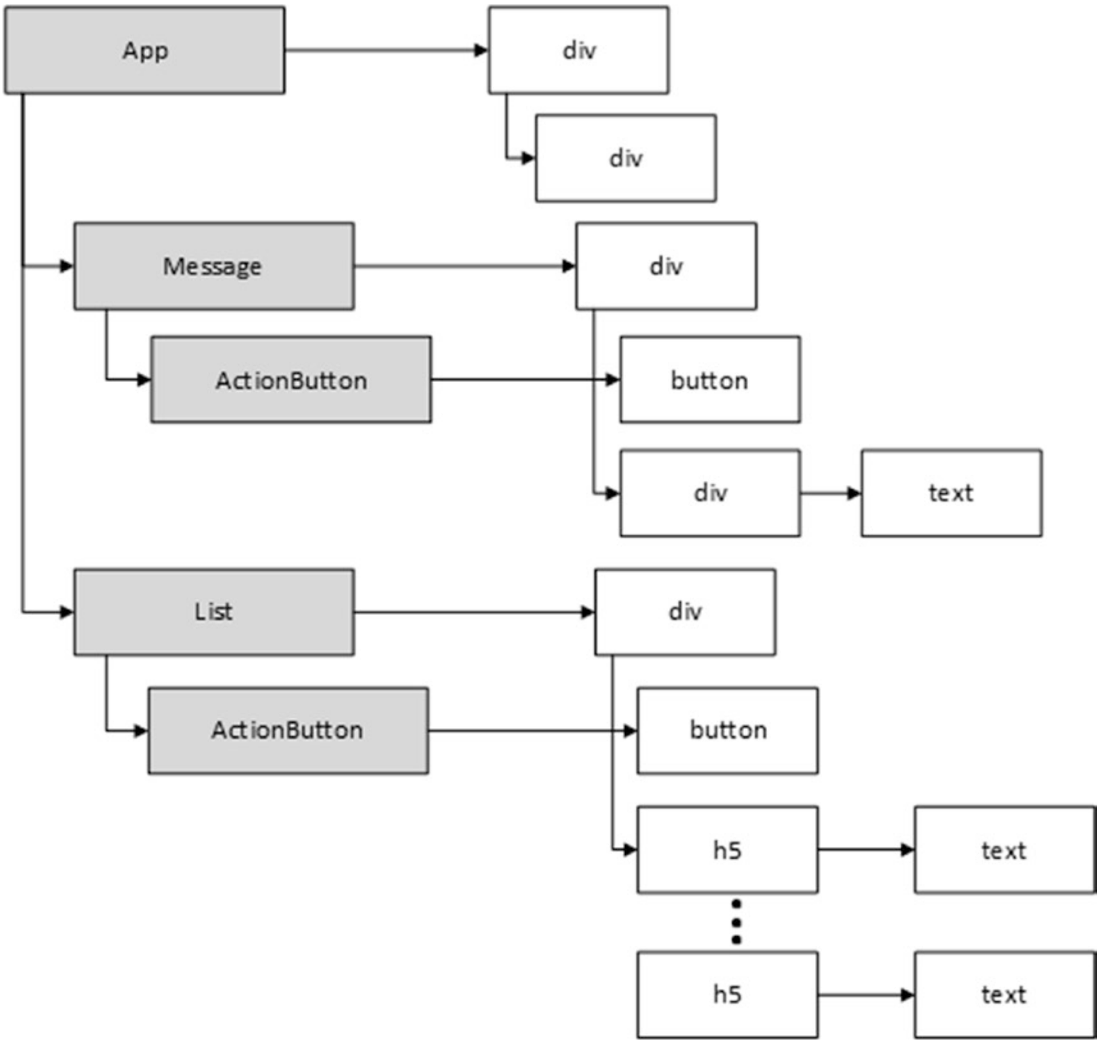

Figure 13-3. *Mapping components to the content they render*

The browser doesn't know—or care—about the components, and its only job is to present the HTML elements in the DOM. React is responsible for managing the components and dealing with the content that is rendered.

Each of the components in the example application has a `console.log` statement in its `render` method, and the messages displayed in the browser's JavaScript console show that each of the five component objects is asked to render its content.

```
...
Render App Component
Render Message Component
Render ActionButton (Increment Counter) Component
Render List Component
Render ActionButton (Reverse Names) Component
...
```

There are messages from one `App` component, one `Message` component, one `List` component, and two `ActionButton` components, matching the structure illustrated in Figures 13-2 and 13-3.

Understanding the Update Process

When the application first starts, React asks all the components to render their content so that it can be displayed to the user. Once the content is displayed, the application is in the *reconciled* state, where the content displayed to the user is consistent with the state of the components.

When the application is reconciled, React waits for something to change. In most applications, changes are caused by user interaction, which triggers an event and results in a call to the `setState` method. The `setState` method updates a component's state data, but it also marks the component as "stale," meaning that the HTML content displayed to the user may be out-of-date. A single event may result in multiple state data changes, and once they have all been processed, React invokes the `render` method for each dirty component and its children. To see the effect of a change, click the Increment Counter button in the browser window, as shown in Figure 13-4.

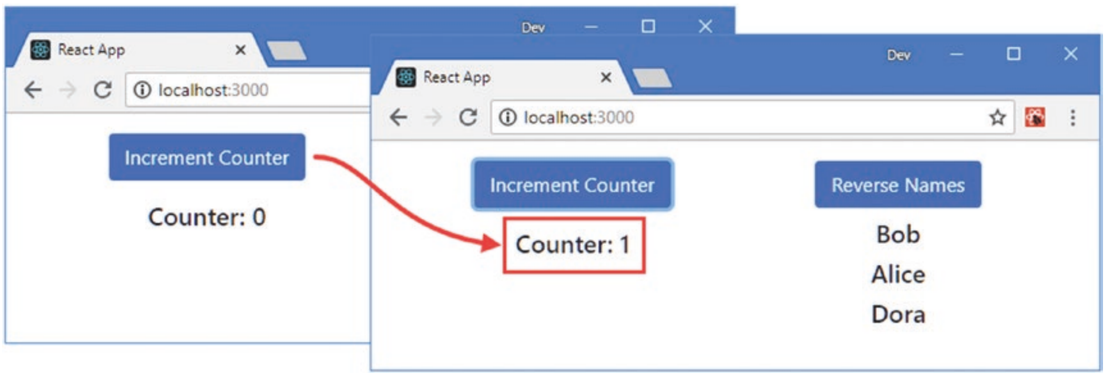

Figure 13-4. *Clicking a button to trigger a change*

The handler that responds to the `click` event updates `App` component's `counter` state data property. Since `App` is the top-level component, that means the `render` method is invoked on all of the application's components, which can be seen in the messages displayed in the browser's JavaScript console.

```
...
Render App Component
Render Message Component
Render ActionButton (Increment Counter) Component
Render List Component
Render ActionButton (Reverse Names) Component
...
```

React only updates the components that are affected by a change, minimizing the amount of work that the application has to do before it is reconciled again. You can see how this works by clicking the Reverse Names button, as shown in Figure 13-5.

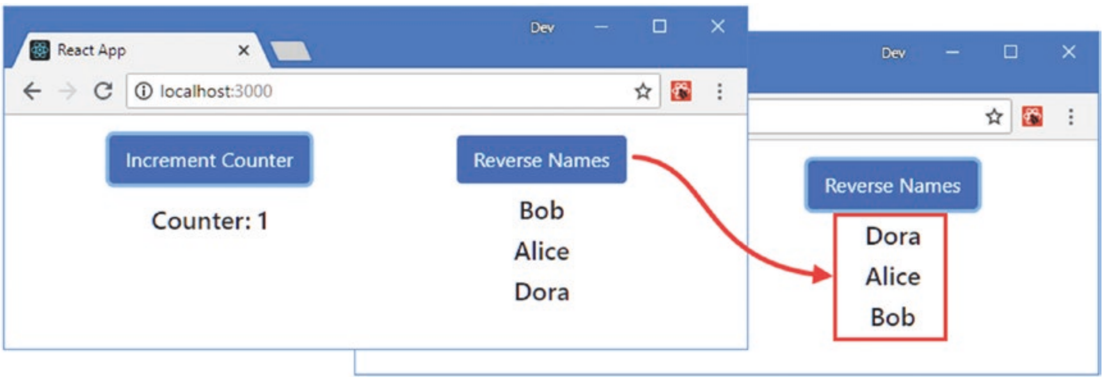

Figure 13-5. *Clicking a button to trigger a limited change*

The `click` event from this button results in a state data change for the `List` component and produces the following messages in the browser's JavaScript console:

```
...
Render List Component
Render ActionButton (Reverse Names) Component
...
```

The `List` component and its child `ActionButton` are marked as stale, but the change hasn't affected the `App` and `Message` components or the other `ActionButton`. React assumes that the content rendered these components is still current and doesn't need to be updated.

Understanding the Reconciliation Process

Although React will invoke the `render` method of any component that has been marked as stale, it doesn't always use the content that is produced. Making changes to the HTML elements in the Domain Object Model is an expensive operation and so React compares the content returned by the components with the previous results so that it can ask the browser to perform the smallest number of operations, a process known as *reconciliation*.

To demonstrate how React minimizes the changes it makes, I have made a change to the content rendered by the `Message` component, as shown in Listing 13-9.

Listing 13-9. Changing Content in the Message.js File in the src Folder

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

```
export class Message extends Component {

    render() {
        console.log(`Render Message Component `);
        return (
            <div>
                <ActionButton theme="primary"  {...this.props} />
                <div id="messageDiv" className="h5 text-center p-2">
                    { this.props.message }
                </div>
            </div>
        )
    }
}
```

The addition of the id attribute makes it easier to manipulate the div element. Using the F12 developer tools, switch to the Console tab, enter the statement shown in Listing 13-10, and press Enter. All browsers allow JavaScript arbitrary statements to be executed, and in Google Chrome this is done by entering code into the prompt at the bottom of the Console tab.

Listing 13-10. Manipulating an HTML Element

```
document.getElementById("messageDiv").classList.add("bg-info")
```

This statement uses the DOM API to select the div element rendered by the Message component and assign it to the bg-info class, which selects a background color defined by the Bootstrap CSS framework. When you click the Increment Counter button, the content of the div element is updated, but the color doesn't change, because React has compared the content returned by the Message component's render method with the previous result and detected that only the content of the div element is different, as illustrated in Figure 13-6.

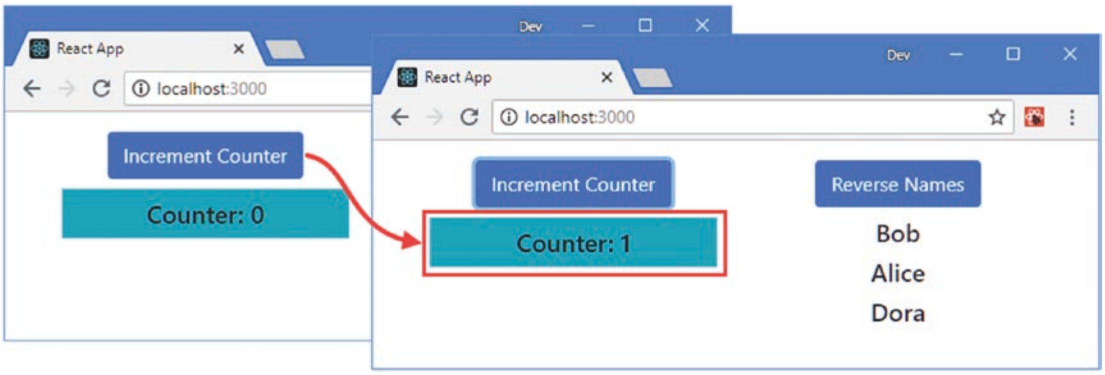

Figure 13-6. *The effect of reconciliation*

React compares the content produced by components with its own cache of previous results, known as the virtual DOM, which is defined in a format that allows for efficient comparisons. The effect is that React doesn't have to query the elements in the DOM to figure out the set of changes.

■ **Tip** Don't confuse the term *virtual DOM*, which is specific to React, with *shadow DOM*, which is a recent browser feature that allows content to be scoped to a specific part of the HTML document.

A second example is required to confirm the reconciliation behavior, demonstrating how React handles a more complex change. In Listing 13-11, I have added state data to the Message component and used it to alternate between two different element types.

Listing 13-11. Alternating Elements in the Message.js File in the src Folder

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

export class Message extends Component {

    constructor(props) {
        super(props);
        this.state = {
            showSpan: false
        }
    }

    handleClick = (event) => {
        this.setState({ showSpan: !this.state.showSpan });
        this.props.callback(event);
    }

    getMessageElement() {
        let div = <div id="messageDiv" className="h5 text-center p-2">
                        { this.props.message }
                    </div>
        return this.state.showSpan ? <span>{ div } </span> : div;
    }

    render() {
        console.log(`Render Message Component `);
        return (
            <div>
                <ActionButton theme="primary" {...this.props}
                    callback={ this.handleClick } />
                { this.getMessageElement() }
            </div>
        )
    }
}
```

The component alternates between displaying a div element directly or wrapping it in a span element. Save the changes and execute the statement shown in Listing 13-12 in the browser's JavaScript console to set the background color of the div element. Notice that I have defined the callback property after passing on props to the ActionButton component using the spread operator. The Message component receives a callback property from its parent, so I have to define my replacement afterward to override it.

■ **Caution** Don't change the top-level element in components in real projects because it causes React to replace elements in the DOM without performing a detailed comparison to detect changes.

Listing 13-12. Manipulating an HTML Element

```
document.getElementById("messageDiv").classList.add("bg-info")
```

When you click the Increment Counter button, the `Message` component's `render` method will return content that includes the `span` element. Click the button a second time, and the `render` method will return to the original content, but the background color will not be shown, as illustrated in Figure 13-7.

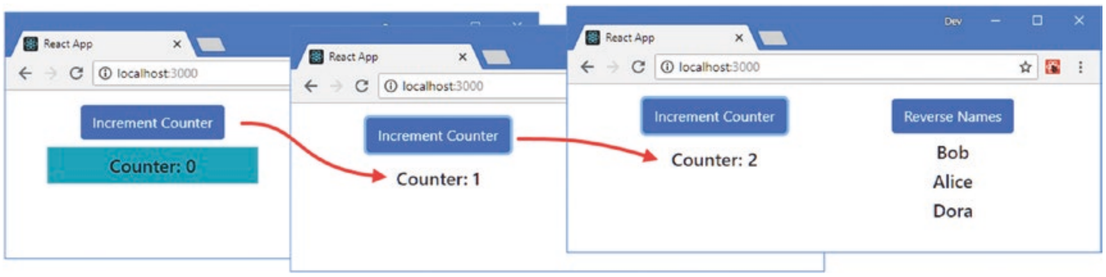

Figure 13-7. *Reconciling different types of element*

React compares the output from the `render` method with the previous results and detects the introduction of the `span` element. React doesn't investigate the content of the new `span` element to perform a more detailed comparison and just uses it to replace the existing `div` element that the browser is displaying.

Understanding List Reconciliation

React has special support for handling elements that display arrays of data. Most operations on lists leave most of the elements in the array, although they can often be in a different location, such as when the objects are sorted. To ensure that React is able to minimize the number of changes it has to make to display a change, elements generated from arrays are required to have a key prop, such as the one defined by the `List` component.

```
...
render() {
    console.log("Render List Component");
    return (
        <div>
            <ActionButton callback={ this.reverseList }
                text="Reverse Names" />
            { this.state.names.map((name, index) => {
                return <h5 key={ name }>{ name }</h5>
            })}
        </div>
    )
}
...
```

The value of the key prop must be unique within the set of elements so that React can identify each one. To demonstrate how React minimizes the changes required to update lists, I added an attribute to the h5 elements rendered by the List component, as shown in Listing 13-13.

■ **Tip** Key values should be stable, such that they should continue to refer to the same object even after operations that make changes to the array. A common mistake is to use the position of an object in the array as its index, which is not stable because many operations on arrays affect the order of objects.

Listing 13-13. Adding an Attribute in the List.js File in the src Folder

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

export class List extends Component {

    constructor(props) {
        super(props);
        this.state = {
            names: ["Bob", "Alice", "Dora"]
        }
    }

    reverseList = () => {
        this.setState({ names: this.state.names.reverse()});
    }

    render() {
        console.log("Render List Component");
        return (
            <div>
                <ActionButton callback={ this.reverseList }
                    text="Reverse Names" />
                { this.state.names.map((name, index) => {
                    return <h5 id={ name.toLowerCase() } key={ name }>{ name }</h5>
                })}
            </div>
        )
    }
}
```

The addition of the id attribute makes it easy to manipulate the element using the browser's JavaScript console, using the same approach as earlier examples. Use the JavaScript console to execute the statements shown in Listing 13-14, which assign the h5 elements to classes that apply Bootstrap background colors.

Listing 13-14. Adding Classes to Elements

```
document.getElementById("bob").classList.add("bg-primary")
document.getElementById("alice").classList.add("bg-secondary")
document.getElementById("dora").classList.add("bg-info")
```

Click the Reverse Names button, and you will see that the order of the h5 elements is changed, but no elements are destroyed and re-created, as shown in Figure 13-8.

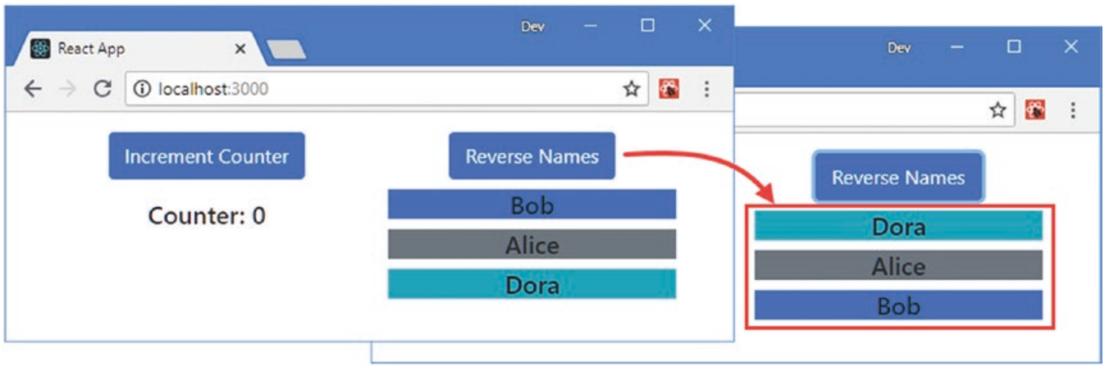

Figure 13-8. *Reordering elements in a list*

Explicitly Triggering Reconciliation

The reconciliation process relies on React being notified of changes through the `setState` method, which allows it to determine which data is stale. It isn't always possible to call the `setState` method if you need to respond to changes that have occurred outside of the application, such as when external data has arrived. For these situations, React provides the `forceUpdate` method, which can be used to explicitly trigger an update and ensures that any changes are reflected in the content presented to the user. To demonstrate explicit reconciliation, I added a file called `ExternalCounter.js` to the `src` folder and used it to define the component shown in Listing 13-15.

■ **Caution** It is worth considering the design of your application if you find yourself using the `forceUpdate` method. The `forceUpdate` method is a blunt instrument and its use can often be avoided by extending the use of state data or by applying one of the composition techniques described in Chapter 14.

Listing 13-15. The Contents of the ExternalCounter.js File in the src Folder

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

let externalCounter = 0;

export class ExternalCounter extends Component {

    incrementCounter = () => {
        externalCounter++;
        this.forceUpdate();
    }
    render() {
        return (
            <div>
                <ActionButton callback={ this.incrementCounter }
                    text="External Counter" />
                <div  className="h5 text-center p-2">
                    External: { externalCounter }
                </div>
            </div>
        )
    }
}
```

This is an obvious candidate for data that could readily be handled as state data, but not all real-world situations are clear-cut. In this case, the component depends on a variable that is outside the control of React, which means that changing the value of the variable won't mark the component as state and start the reconciliation process. Instead, the `incrementCounter` method calls the `forceUpdate` method, which explicitly starts reconciliation and ensures that the new value is incorporated in the content displayed to the user. To incorporate the new component into the applications, I made the changes shown in Listing 13-16 to the `App` component.

Listing 13-16. Adding a New Component in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
import { List } from "./List";
import { ExternalCounter } from './ExternalCounter';

export default class App extends Component {

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

    incrementCounter = () => {
        this.setState({ counter: this.state.counter + 1 });
    }

    render() {
        console.log("Render App Component");
        return  <div className="container text-center">
                    <div className="row p-2">
                        <div className="col-4">
                            <Message message={ `Counter: ${this.state.counter}`}
                                callback={ this.incrementCounter }
                                text="Increment Counter" />
                        </div>
                        <div className="col-4">
                            <List />
                        </div>
                        <div className="col-4">
                            <ExternalCounter />
                        </div>
                    </div>
                </div>
    }
}
```

The new component is displayed on the right side of the application's layout, and clicking the External Counter button explicitly marks that component as stale and triggers the reconciliation process, as shown in Figure 13-9.

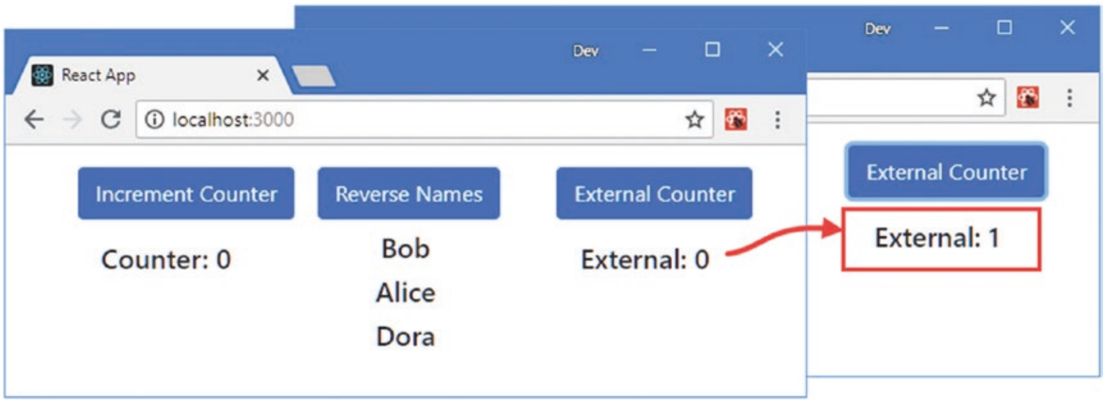

Figure 13-9. *Explicitly starting reconciliation*

Understanding the Component Lifecycle

Most class-based stateful components implement a constructor and the `render` method. The constructor is used to receive props from the parent and to define state data. The `render` method is used to produce content, both when the application starts and when React is responding to an update.

The constructor and `render` method are part of a larger component lifecycle that stateful components can participate in by implementing methods that React invokes to signal changes in the lifecycle. In the sections that follow, I explain the different stages of the component lifecycle and the methods for each of them. For quick reference, Table 13-3 lists the commonly used lifecycle methods. There are also three advanced methods that I describe in the "Using the Advanced Lifecycle Methods" section.

■ **Note** See the "Using the Effect Hook" section for details of how the hooks feature provides access to the lifecycle features for functional components.

***Table 13-3.** The Stateful Component Lifecycle Methods*

Name	Description
`constructor`	This special method is called when a new instance of the component class is created.
`render`	This method is called when React requires content from the component.
`componentDidMount`	This method is called after the initial content rendered by the component has been processed.
`componentDidUpdate`	This method is called after React has completed the reconciliation process following an update.
`componentWillUnmount`	This method is called before a component is destroyed.
`componentDidCatch`	This method is used to handle errors, as described in Chapter 14.

Understanding the Mounting Phase

The process by which React creates a component and renders its content for the first time is called *mounting*, and there are three commonly used methods that components implement to participate in the mounting process, as illustrated in Figure 13-10.

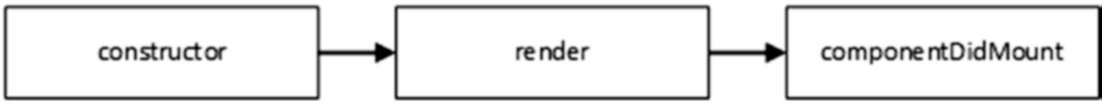

***Figure 13-10.** The mounting phase*

The constructor is called when React needs to create a new instance of a component, which gives the component an opportunity to receive the props from its parents, define its state data, and perform other preparatory work.

Next, the `render` method is called so that the component provides React with the content that will be added to the DOM. Finally, React calls the `componentDidMount` method, which tells the component that its content has been added to the DOM.

The `componentDidMount` method is typically used to perform Ajax requests to get data from web services, which I demonstrate in Part 3. For the purposes of this chapter, I implemented the `componentDidMount` method in the `Message` component and used it to write a message to the browser's JavaScript console, as shown in Listing 13-17.

Listing 13-17. Implementing a Lifecycle Method in the Message.js File in the src Folder

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

export class Message extends Component {
    constructor(props) {
        super(props);
        this.state = {
            showSpan: false
        }
    }

    handleClick = (event) => {
        this.setState({ showSpan: !this.state.showSpan });
        this.props.callback(event);
    }

    getMessageElement() {
        let div = <div id="messageDiv" className="h5 text-center p-2">
                        { this.props.message }
                  </div>
        return this.state.showSpan ? <span>{ div } </span> : div;
    }

    render() {
        console.log(`Render Message Component `);
        return (
            <div>
                <ActionButton theme="primary" {...this.props}
                    callback={ this.handleClick } />
                { this.getMessageElement() }
            </div>
        )
    }

    componentDidMount() {
        console.log("componentDidMount Message Component");
    }
}
```

Save the changes to the Message component and examine the messages displayed in the browser's JavaScript console as the application is updated, and you will see that the componentDidMount method was invoked.

```
...
Render App Component
Render Message Component
Render ActionButton (Increment Counter) Component
Render List Component
Render ActionButton (Reverse Names) Component
Render ActionButton (External Counter) Component
componentDidMount Message Component
...
```

You can see that the componentDidMount method has been called after all the component's render methods have been invoked. The componentDidMount method is invoked when React needs a new instance of the component, which includes application startup. But mounting will also occur when React creates an instance of a component while the application is running, such as when content is conditionally rendered, as shown in Listing 13-18.

Listing 13-18. Conditionally Displaying a Component in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
import { List } from "./List";
import { ExternalCounter } from './ExternalCounter';

export default class App extends Component {

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

    incrementCounter = () => {
        this.setState({ counter: this.state.counter + 1 });
    }

    handleChange = () => {
        this.setState({ showMessage: !this.state.showMessage });
    }

    render() {
        console.log("Render App Component");
        return  (
            <div className="container text-center">
                <div className="row p-2">
                    <div className="col-4">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                checked={ this.state.showMessage }
                                onChange={ this.handleChange } />
                            <label className="form-check-label">Show</label>
                        </div>
                        { this.state.showMessage &&
                            <Message message={ `Counter: ${this.state.counter}`}
                                callback={ this.incrementCounter }
                                text="Increment Counter" />
                        }
                    </div>
```

```
                        <div className="col-4">
                            <List />
                        </div>
                        <div className="col-4">
                            <ExternalCounter />
                        </div>
                    </div>
                </div>
            )
        }
    }
```

I added a checkbox and used the onChange property to register the handleChange method to receive change events, which are triggered when the checkbox is toggled. The checkbox is used to control visibility of the Message component, as shown in Figure 13-11.

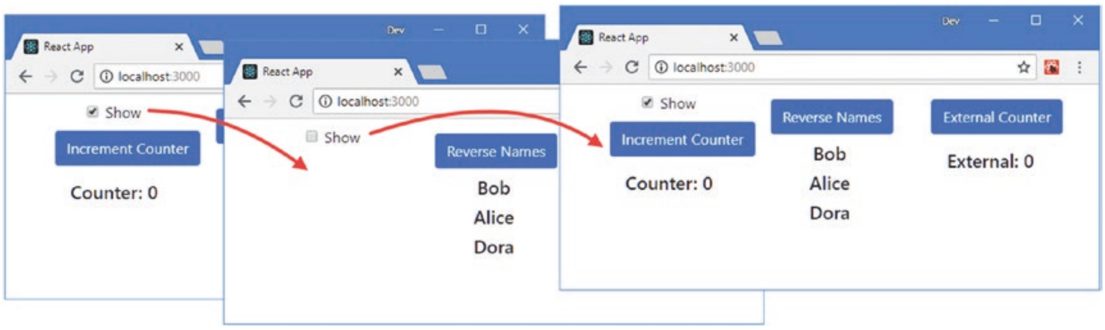

Figure 13-11. *Controlling the visibility of a component*

Each time the checkbox is toggled on, React creates a new Message object and goes through the mounting process, calling each of the methods in turn: constructor, render, and componentDidMount. This can be seen in the messages displayed in the browser's JavaScript console.

Understanding the Update Phase

The process by which React responds to changes and goes through reconciliation is known as the *update phase*, which invokes calling the render method to get content from the component and then calling the componentDidUpdate after the reconciliation process is complete, as shown in Figure 13-12.

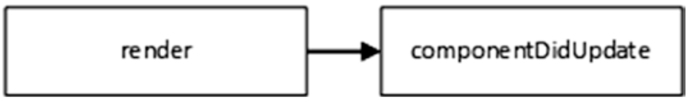

Figure 13-12. *The update phase*

The main use of the componentDidUpdate method is to directly manipulate the HTML elements in the DOM using the React refs feature, which I describe in Chapter 16. For this chapter, I have implemented the method in the Message component and used it to write a message to the browser's JavaScript console, as shown in Listing 13-19.

■ **Tip** The componentDidUpdate method is called even if the reconciliation process determines that the content generated by the component has not changed.

Listing 13-19. Implementing a Lifecycle Method in the Message.js File in the src Folder

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

export class Message extends Component {

    // ...other methods omitted for brevity...

    componentDidMount() {
        console.log("componentDidMount Message Component");
    }

    componentDidUpdate() {
        console.log("componentDidUpdate Message Component");
    }
}
```

After the initial rendering that is performed during the mounting phase, any subsequent calls to the render method will be followed by a call to the componentDidUpdate method once React has completed the reconciliation process and updated the DOM. Clicking the Increment Counter button will start the update phase and produce the following message in the browser's JavaScript console:

```
...
Render App Component
Render Message Component
Render ActionButton (Increment Counter) Component
Render List Component
Render ActionButton (Reverse Names) Component
Render ActionButton (External Counter) Component
componentDidUpdate Message Component
...
```

Understanding the Unmounting Phase

When a component is about to be destroyed, React will call the componentWillUnmount method, which provides components with the opportunity to release resources, close network connections, and stop any asynchronous tasks. In Listing 13-20, I have implemented the componentWillUnmount method in the Message component and used it to write a message to the browser's JavaScript console.

Listing 13-20. Implementing a Lifecycle Method in the Message.js File in the src Folder

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

export class Message extends Component {

    // ...other methods omitted for brevity...

    componentDidMount() {
        console.log("componentDidMount Message Component");
    }

    componentDidUpdate() {
        console.log("componentDidUpdate Message Component");
    }

    componentWillUnmount() {
        console.log("componentWillUnmount Message Component");
    }
}
```

You can trigger the unmounting phase by unchecking the checkbox that I added in Listing 13-20. When React reconciles the new content rendered by the App component, it determines that the Message component is no longer required and calls the componentWillUnmount method before destroying the object, producing the following messages in the browser's JavaScript console:

```
...
Render App Component
Render List Component
Render ActionButton (Reverse Names) Component
Render ActionButton (External Counter) Component
componentWillUnmount Message Component
...
```

React will not reuse components once they have been unmounted. React will create a new object and perform the mounting sequence if another Message component is required, such as when the checkbox is toggled again. This means you can always rely on the constructor and the componentDidMount methods to initialize a component, and a component object will never be asked to recover from an unmounted state.

Using the Effect Hook

Components defined as functions are unable to implement methods and cannot participate in the lifecycle in the same way. For this type of component, the hooks feature provides the effect hook, which is roughly equivalent to the componentDidMount, componentDidUpdate, and componentWillUnmount methods. To show the use of the effect hook, I added a file called HooksMessage.js to the src folder and added the code shown in Listing 13-21.

Listing 13-21. The Contents of the HooksMessage.js File in the src Folder

```
import React, { useState, useEffect} from "react";
import { ActionButton } from "./ActionButton";

export function HooksMessage(props) {
    const [showSpan, setShowSpan] = useState(false);

    useEffect(() => console.log("useEffect function invoked"));

    const handleClick = (event) => {
        setShowSpan(!showSpan);
        props.callback(event);
    }

    const getMessageElement = () => {
        let div = <div id="messageDiv" className="h5 text-center p-2">
                        { props.message }
                  </div>
        return showSpan ? <span>{ div } </span> : div;
    }

    return (
        <div>
            <ActionButton theme="primary" {...props} callback={ handleClick } />
            { getMessageElement() }
        </div>
    )
}
```

This component provides the same functionality as the Message component but is expressed as a function that uses hooks. The useEffect function is used to register a function that will be invoked when the component is mounted, updated, and unmounted. The same function is invoked in all three situations, which reflects the nature of using a function for a component, as opposed to a class. In Listing 13-22, I have added the new component to the content rendered by the App component.

Listing 13-22. Rendering a New Component in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
import { List } from "./List";
import { ExternalCounter } from './ExternalCounter';
import { HooksMessage } from './HooksMessage';

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counter: 0,
            showMessage: true
        }
    }
    incrementCounter = () => {
        this.setState({ counter: this.state.counter + 1 });
    }

    handleChange = () => {
        this.setState({ showMessage: !this.state.showMessage });
    }

    render() {
        console.log("Render App Component");
        return  (
            <div className="container text-center">
                <div className="row p-2">
                    <div className="col-4">
                        <div className="form-check">
                            <input type="checkbox" className="form-check-input"
                                checked={ this.state.showMessage }
                                onChange={ this.handleChange } />
                            <label className="form-check-label">Show</label>
                        </div>
                        { this.state.showMessage &&
                            <div>
                                <Message message={ `Counter: ${this.state.counter}`}
                                    callback={ this.incrementCounter }
                                    text="Increment Counter" />
                                <HooksMessage
                                    message={ `Counter: ${this.state.counter}`}
                                    callback={ this.incrementCounter }
                                    text="Increment Counter" />
                            </div>
                        }
                    </div>
                    <div className="col-4">
                        <List />
                    </div>
                    <div className="col-4">
                        <ExternalCounter />
                    </div>
                </div>
            </div>
        )
    }
}
```

Save the changes to the components and examine the messages shown in the browser's JavaScript console to see the effect hook function being invoked when the component is mounted and updated, as follows:

```
...
Render List Component
ActionButton.js:6 Render ActionButton (Reverse Names) Component
ActionButton.js:6 Render ActionButton (External Counter) Component
```

```
Message.js:37 componentDidMount Message Component
HooksMessage.js:7 useEffect function invoked
...
```

The function passed to useState can return a cleanup function that will be invoked when the component is unmounted, providing a feature similar to the componentWillUnmount method, as shown in Listing 13-23.

Listing 13-23. Using a Cleanup Function in the HooksMessage.js File in the src Folder

```
import React, { useState, useEffect} from "react";
import { ActionButton } from "./ActionButton";

export function HooksMessage(props) {
    const [showSpan, setShowSpan] = useState(false);

    useEffect(() => {
        console.log("useEffect function invoked")
        return () => console.log("useEffect cleanup");
    });

    const handleClick = (event) => {
        setShowSpan(!showSpan);
        props.callback(event);
    }

    const getMessageElement = () => {
        let div = <div id="messageDiv" className="h5 text-center p-2">
                        { props.message }
                  </div>
        return showSpan ? <span>{ div } </span> : div;
    }

    return (
        <div>
            <ActionButton theme="primary" {...props} callback={ handleClick } />
            { getMessageElement() }
        </div>
    )
}
```

Toggle the checkbox to unmount the components, and you will see the following message in the browser's JavaScript console:

```
...
Render ActionButton (Reverse Names) Component
ActionButton.js:6 Render ActionButton (External Counter) Component
Message.js:45 componentWillUnmount Message Component
HooksMessage.js:9 useEffect cleanup
...
```

Using the Advanced Lifecycle Methods

The features described in the previous sections are useful in many projects, especially using the `componentDidMount` method to request remote data, which I demonstrate in Part 3. React provides advanced lifecycle methods for class-based components that are useful in specific situations I describe in the sections that follow, although one of these methods is used in conjunction with the refs feature that I describe in Chapter 16. For quick reference, Table 13-4 describes the advanced lifecycle methods.

Table 13-4. *The Advanced Component Lifecycle Methods*

Name	Description
`shouldComponentUpdate`	This method allows a component to indicate that it does not need to be updated.
`getDerivedStateFromProps`	This method allows a component to set its state data values based on the props it receives.
`getSnapshotBeforeUpdate`	This method allows a component to capture information about its state before the reconciliation process updates the DOM. This method is used in conjunction with the ref feature, described in Chapter 16.

Preventing Unnecessary Component Updates

React's default behavior is to mark a component as stale and render its content whenever its state data changes. And, since a component's state can be passed on to its children as props, the descendant components are rendered as well, as you have seen in earlier examples.

Components can override the default behavior by implementing the `shouldComponentUpdate` method. This feature allows components to improve the application's performance by avoiding calls to the `render` method when they are not required.

The `shouldComponentUpdate` method is called in the update phase, and its result determines whether React will call the `render` method to get fresh content from the component, as illustrated in Figure 13-13. The arguments to the `shouldComponentUpdate` method are new props and state objects that can be inspected and compared to the existing values. React will continue with the update phase if the `shouldComponentUpdate` method returns `true`. If the `shouldComponentUpdate` method returns false, React will abandon the update phase for the component and will not call the `render` and `componentDidUpdate` methods.

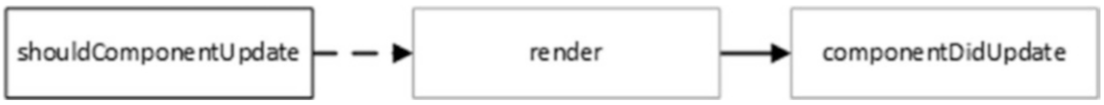

Figure 13-13. *The advanced sequence of update methods*

In Listing 13-24, I have implemented the `showComponentUpdate` method in the `Message` component and used it to prevent updates if the value of the message prop has not changed. (I have also removed the lifecycle methods from earlier examples for the sake of brevity.)

Listing 13-24. Preventing Updates in the Message.js File in the src Folder

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

export class Message extends Component {

    constructor(props) {
        super(props);
        this.state = {
            showSpan: false
        }
    }

    handleClick = (event) => {
        this.setState({ showSpan: !this.state.showSpan });
        this.props.callback(event);
    }

    getMessageElement() {
        let div = <div id="messageDiv" className="h5 text-center p-2">
                        { this.props.message }
                    </div>
        return this.state.showSpan ? <span>{ div } </span> : div;
    }

    render() {
        console.log(`Render Message Component `);
        return (
            <div>
                <ActionButton theme="primary" {...this.props}
                        callback={ this.handleClick } />
                { this.getMessageElement() }
            </div>
        )
    }

    shouldComponentUpdate(newProps, newState) {
        let change = newProps.message !== this.props.message;
        if (change) {
            console.log(`shouldComponentUpdate ${this.props.text}: Update Allowed`)
        } else {
            console.log(`shouldComponentUpdate ${this.props.text}: Update Prevented`)
        }
        return change;
    }
}
```

In Listing 13-25, I have revised the App component so it renders two Message components, each of which receives and modifies a state data value as a prop.

Listing 13-25. Displaying Side-By-Side Components in the App.js File in the src Folder

```
import React, { Component } from 'react';
import { Message } from "./Message";
//import { List } from "./List";
//import { ExternalCounter } from './ExternalCounter';

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counterLeft: 0,
            counterRight: 0
        }
    }

    incrementCounter = (counter) => {
        if (counter === "left") {
            this.setState({ counterLeft: this.state.counterLeft + 1});
        } else {
            this.setState({ counterRight: this.state.counterRight+ 1});
        }
    }

    render() {
        console.log("Render App Component");
        return (
            <div className="container text-center">
                <div className="row p-2">
                    <div className="col-6">
                        <Message
                            message={ `Left: ${this.state.counterLeft}`}
                            callback={ () => this.incrementCounter("left") }
                            text="Increment Left Counter" />
                    </div>
                    <div className="col-6">
                        <Message
                            message={ `Right: ${this.state.counterRight}`}
                            callback={ () => this.incrementCounter("right") }
                            text="Increment Right Counter" />
                    </div>
                </div>
            </div>
        )
    }
}
```

The new content rendered by the App component displays the Message components side by side, as shown in Figure 13-14. Clicking either of the button elements increments the counter for that component.

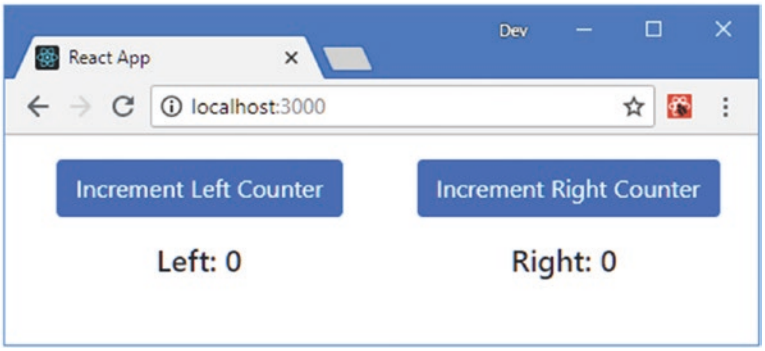

Figure 13-14. *Displaying components side by side*

The default React behavior is to render both `Message` components when either of the `counterLeft` or `counterRight` state data values changes, which results in one of the components rendering content unnecessarily. The implementation of the `shouldComponentUpdate` method in Listing 13-25 overrides this behavior and ensures that only the component affected by the change is updated. If you click either of the buttons presented by the application, you will see a message in the browser's JavaScript console noting that `shouldComponentUpdate` prevented one of the components from being updated.

```
...
Render App Component
shouldComponentUpdate Increment Left Counter: Update Allowed
Render Message Component
Render ActionButton (Increment Left Counter) Component
shouldComponentUpdate Increment Right Counter: Update Prevented
...
```

Setting State Data from Prop Values

The `getDerivedStateFromProps` method is called before the `render` method in the mounting phase and before the `shouldComponentUpdate` method in the update phase, as shown in Figure 13-15. The `getDerivedStateFromProps` method provides components with the opportunity to inspect prop values and use them to update its state data before its content is rendered and is intended for use by components whose behavior is affected by changing prop values over time.

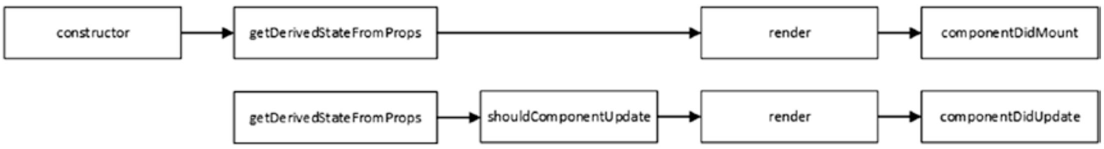

Figure 13-15. *Updating state data from props*

The getDerivedStateFromProps method is static, which means that it is unable to access any of the instance methods or properties via the this keyword. Instead, the method receives a props object, which contains the props values provided by the parent component, and a state object, which represents the current state data. The getDerivedStateFromProps method returns a new state data object that is derived from the prop data.

To demonstrate this method, I added a file called DirectionDisplay.js to the src folder and used it to define the component shown in Listing 13-26.

Listing 13-26. The Contents of the DirectionDisplay.js File in the src Folder

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

export class DirectionDisplay extends Component {

    constructor(props) {
        super(props);
        this.state = {
            direction: "up",
            lastValue: 0
        }
    }

    getClasses() {
        return (this.state.direction === "up" ? "bg-success" : "bg-danger")
            + " text-white text-center p-2 m-2";
    }

    render() {
        return <h5 className={ this.getClasses() }>
                    { this.props.value }
                </h5>
    }

    static getDerivedStateFromProps(props, state) {
        if (props.value !== state.lastValue) {
            return {
                lastValue: props.value,
                direction: state.lastValue > props.value ? "down" : "up"
            }
        }
        return state;
    }
}
```

This component displays a numeric value with a background color that indicates whether the current value is larger or smaller than the last value. The getDerivedStateFromProps method receives the new prop values and the component's current state data and uses them to create a new state data object that includes the direction in which the prop value has changed. In Listing 13-27, I have updated the App component so that it renders the DirectionDisplay component and buttons that change its prop data value.

Listing 13-27. Rendering a New Component in the App.js File in the src Folder

```
import React, { Component } from 'react';
//import { Message } from "./Message";
import { DirectionDisplay } from './DirectionDisplay';

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            counter: 100
        }
    }

    changeCounter = (val) => {
        this.setState({ counter: this.state.counter + val })
    }

    render() {
        console.log("Render App Component");
        return  (
            <div className="container text-center">
                <DirectionDisplay value={ this.state.counter } />
                <div className="text-center">
                    <button className="btn btn-primary m-1"
                        onClick={ () => this.changeCounter(-1)}>Decrease</button>
                    <button className="btn btn-primary m-1"
                        onClick={ () => this.changeCounter(1)}>Increase</button>
                </div>
            </div>
        )
    }
}
```

The result is that the background color selected by the `DirectionDisplay` component changes based on the output from the `getDerivedStateFromProps` method, as shown in Figure 13-16.

■ **Tip** Notice that I create a new state data object only if the value of the prop is different. Remember that React will trigger a component's update phase when an ancestor's state has changed, which means that the `getDerivedStateFromProps` method may be called even though none of the prop values on which the component depends has changed.

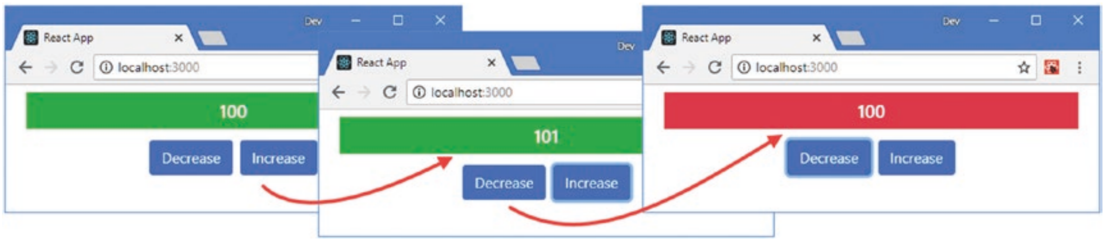

Figure 13-16. *Deriving state data from prop values*

Summary

In this chapter, I explained how React deals with the content rendered by components through the reconciliation process. I also described the broader component lifecycle and showed you how to receive notifications in stateful components by implementing methods. In the next chapter, I describe the different ways that components can be combined to create complex features.