CHAPTER 12

Working with Events

In this chapter, I describe the React support for events, which are generated by HTML elements, typically in response to user interaction. The React event features will be familiar if you have used the DOM event API features, but there are some important differences that can confuse the unwary developer. Table 12-1 puts the React event features in context.

Table 12-1. *Putting React Events in Context*

Question	Answer
What are they?	React events are triggered by elements to report important occurrences, most often user interaction.
Why are they useful?	Events allow components to respond to interaction with the content they render, which forms the foundation for interactive applications.
How are they used?	Interest in an event is indicated by adding properties to the elements rendered by a component. When an event in which a component is interested is triggered, the function specified by the property is invoked, allowing the component to update its state, invoke a function prop, or otherwise reflect the effect of the event.
Are there any pitfalls or limitations?	React events are similar to the events provided by the DOM API but with some differences that can present pitfalls for the unwary, especially when it comes to event phases, as described in the "Managing Event Propagation" section. Not all of the events defined by the DOM API are supported (see `https://reactjs.org/docs/events.html` for a list of events that React supports).
Are there any alternatives?	There is no alternative to using events, which provide an essential link between user interaction and the content rendered by a component.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_12

Table 12-2 summarizes the chapter.

Table 12-2. *Chapter Summary*

Problem	Solution	Listing
Handle an event	Add the prop that corresponds to the event name and use the expression to process the event	6–10
Determine the event type	Use the event object's `type` property	11
Prevent an event from being reset before it is used	Use the event object's `persist` method	12, 13
Invoke event handlers with a custom argument	Define an inline function in the prop expression that invokes the handler method with the required data	14, 15
Prevent an event's default behavior	Use the event object's `preventDefault` method	16
Manage the propagation of an event	Determine the event phase	17–23
Stop an event	Use the event object's `stopPropagation` method	24

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 12-1.

■ **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 12-1. Creating the Example Project

```
npx create-react-app reactevents
```

Run the commands shown in Listing 12-2 to navigate to the `reactevents` folder and add the Bootstrap package to the project.

Listing 12-2. Adding the Bootstrap CSS Framework

```
cd reactevents
npm install bootstrap@4.1.2
```

To include the Bootstrap CSS stylesheet in the application, add the statement shown in Listing 12-3 to the `index.js` file, which can be found in the `src` folder.

Listing 12-3. 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();
```

Next, replace the contents of the `App.js` file with the code shown in Listing 12-4, which will provide the starting point for the examples in this chapter. The listing replaces the existing functional component with one that uses a class.

Listing 12-4. The Contents of the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    render() {
        return  (
            <div className="m-2">
                <div className="h4 bg-primary text-white text-center p-2">
                    { this.state.message }
                </div>
                <div className="text-center">
                    <button className="btn btn-primary">Click Me</button>
                </div>
            </div>
        )
    }
}
```

Using the command prompt, run the commands shown in Listing 12-5 in the `reactevents` folder to start the development tools.

Listing 12-5. 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 12-1.

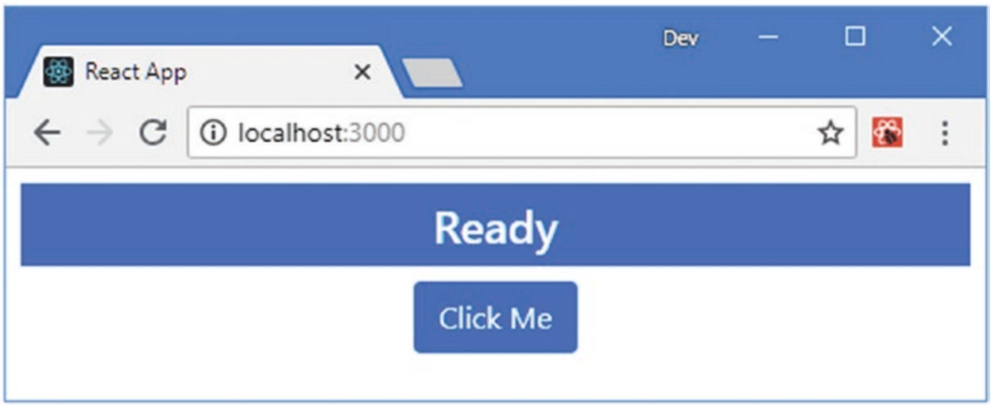

Figure 12-1. *Running the example application*

Understanding Events

Events are triggered by HTML elements to signal important changes, such as when the user clicks a button or types into a text field. Handling events in React is similar to using the Domain Object Model API, although there are important differences. In Listing 12-6, I have added an event handler that is invoked when the `button` element is clicked.

Listing 12-6. Adding an Event Handler in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    render() {
        return (
            <div className="m-2">
                <div className="h4 bg-primary text-white text-center p-2">
                    { this.state.message }
                </div>
                <div className="text-center">
                    <button className="btn btn-primary"
                        onClick={ () => this.setState({ message: "Clicked!"})}>
                            Click Me
                    </button>
                </div>
            </div>
        )
    }
}
```

Events are handled using properties that share the name of the corresponding DOM API property, expressed in camel case. The DOM API `onclick` property is expressed as `onClick` in React applications and specifies how to handle the `click` event, which is triggered when the user clicks an element. The expression for an event handling property is a function that will be invoked when the specified event is triggered, like this:

```
...
<button className="btn btn-primary"
        onClick={ () => this.setState({ message: "Clicked!"})}>
    Click Me
</button>
...
```

This is an example of an inline function, which calls the `setState` method to change the value of the `message` state data property. When the `button` element is clicked, the `click` event is triggered, and React will invoke the inline function, producing the result shown in Figure 12-2.

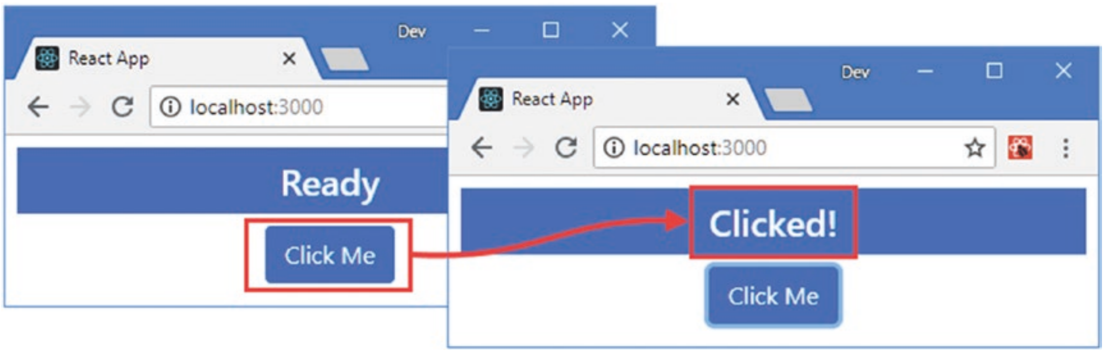

Figure 12-2. *Handling an event*

Invoking a Method to Handle an Event

Stateful components can define methods and use them to respond to events, which helps avoid duplicating code in expressions when several elements handle the same event in the same way. For simple methods that don't change the state of the application or access other component features, the method can be specified as shown in Listing 12-7.

Listing 12-7. Adding an Event Handling Method in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    handleEvent() {
        console.log("handleEvent method invoked");
    }

    render() {
        return  <div className="m-2">
                    <div className="h4 bg-primary text-white text-center p-2">
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ this.handleEvent }>
                                Click Me
                        </button>
                    </div>
                </div>
    }
}
```

Notice that the onClick expression doesn't include parentheses, which would cause React to invoke the function when the render method is invoked, as explained in the sidebar. The handleEvent method doesn't change the state of the application and just writes out a message to the browser's JavaScript console. If you click the button in the browser window, you will see the following output shown in the console:

```
handleEvent method invoked
```

AVOIDING THE EVENT FUNCTION INVOCATION PITFALLS

The value assigned to an event handling property, such as `onClick`, must be an expression that returns a function that React can invoke to handle an event. There are two common mistakes when using an event handling property. The first mistake is to enclose the function you require in quotes rather than braces, like this:

```
...
<button className="btn btn-primary" onClick="this.handleEvent" >
...
```

This provides React with a string value instead of a function and produces an error in the browser's JavaScript console. The other common mistake is to use an expression that invokes the function you require.

```
...
<button className="btn btn-primary" onClick={ this.handleEvent() } >
...
```

This expression results in React invoking the `handleEvent` method when the component object is created and not when an event is triggered. You won't receive an error or warning for this mistake, which makes the problem harder to spot.

Accessing Component Features in an Event Handling Method

Additional work is required if you need to access the component's features in a method that handles an event. The value of the `this` keyword isn't set by default when JavaScript class methods are invoked, which means that there is no way for statements in the `handleEvent` method to access the component's methods and properties. In Listing 12-8, I have added a statement to the `handleEvent` method that invokes the `setState` method, which is accessed using the `this` keyword.

Listing 12-8. Accessing Component Features in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    handleEvent() {
        this.setState({ message: "Clicked!"});
    }
    render() {
        return  <div className="m-2">
                    <div className="h4 bg-primary text-white text-center p-2">
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ this.handleEvent }>
                                Click Me
                        </button>
                    </div>
                </div>
    }
}
```

The handleEvent method will be invoked when the button is clicked, but the following error will be produced because this is undefined:

```
Uncaught TypeError: Cannot read property 'setState' of undefined
```

To ensure that a value is assigned to this, event handling methods can be expressed using the JavaScript public class fields syntax, as shown in Listing 12-9.

Listing 12-9. Redefining an Event Handling Method in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    handleEvent = () => {
        this.setState({ message: "Clicked!"});
    }

    render() {
        return  <div className="m-2">
                    <div className="h4 bg-primary text-white text-center p-2">
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
```

```
                                onClick={ this.handleEvent }>
                                    Click Me
                            </button>
                        </div>
                </div>
        }
    }
```

The name of the method is followed by the equal sign, open and close parentheses, the fat arrow symbol, and then the message body, as shown in the listing. This is an awkward syntax, but I prefer it to the alternatives (described in the sidebar), and this is the approach I use throughout this chapter and the rest of the book. When you click the button element, the `handleEvent` method is provided with a value for `this`, producing the result shown in Figure 12-3.

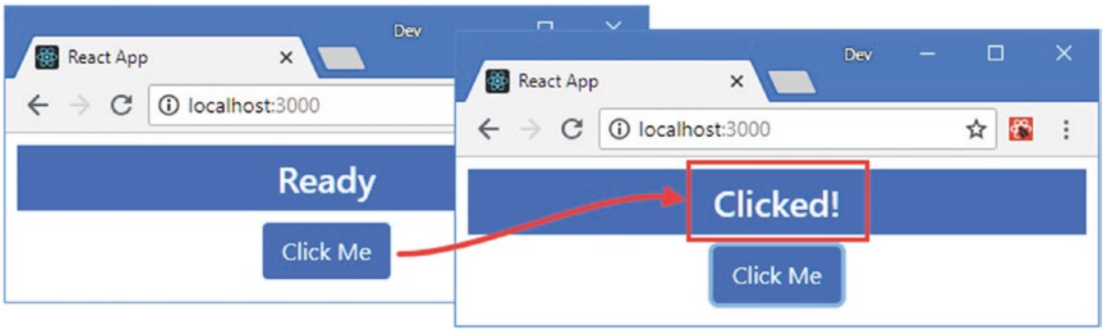

Figure 12-3. *Binding for an event handler*

ALTERNATIVE WAYS TO ACCESS COMPONENT FEATURES

There are two alternative ways to provide an event handling method with a value for `this`. The first is to use an inline function in the expression for the event property.

```
...
<button className="btn btn-primary"
        onClick={ () => this.handleEvent() }>
    Click Me
</button>
...
```

Notice that the event handler method is invoked by the expression, which means that open and close parentheses are required after the method name. The other approach is to add a statement to the constructor for each of the component's event handler methods.

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

```
        this.state = {
            message: "Ready"
        }
        this.handleEvent = this.handleEvent.bind(this);
    }
    ...
```

All three approaches take a while to get used to—and all are a little inelegant—and you should follow the approach that you find most comfortable.

Receiving an Event Object

When an event is triggered, React provides a SyntheticEvent object that describes the event to the handler object. The SyntheticEvent is a wrapper around the Event object provided by the DOM API that defines the same features but with additional code to ensure that events are described consistently in different browsers. The SyntheticEvent object has the basic properties and methods described in Table 12-3. (There are further methods and properties that I describe in later sections.)

REACT EVENTS VERSUS DOM EVENTS

React events provide an essential link between a component and the content it renders—but React events are not DOM events, even though they appear the same most of the time. If you go beyond the most commonly used features, you will encounter important differences that can produce unexpected results.

First, React doesn't support all events, which means that there some DOM API events that don't have corresponding React properties that components can use. You can see the set of events that React supports at https://reactjs.org/docs/events.html. The most commonly used events are included in the list, but not every event is available.

Second, React doesn't allow components to create and publish custom events. The React model for interaction between components is through function props, described in Chapter 10, and custom events are not distributed when the Event.dispatchEvent method is used.

Third, React provides a custom object as a wrapper around the DOM event objects, which doesn't always behave in the same way as the DOM event. You can access the DOM event through the wrapper, but this should be done with caution because it can cause unexpected side effects.

Finally, React intercepts DOM events in their bubble phase (described later in this chapter) and feeds them through the hierarchy of components, providing components with the opportunity to respond to events and update the content they render. This means some of the features provided by the event wrapper object don't work as expected, especially when it comes to propagation, as described in the "Managing Event Propagation" section.

Table 12-3. *The Basic Properties and Methods Defined by the SyntheticEvent Object*

Name	Description
`nativeEvent`	This property returns the `Event` object provided by the DOM API.
`target`	This property returns the object that represents the element that is the source of the event.
`timeStamp`	This property returns a timestamp that indicates when the event was triggered.
`type`	This property returns a string that indicates the event type.
`isTrusted`	This property returns `true` when the event has been initiated by the browser and `false` when the event object has been created in code.
`preventDefault()`	This method is called to prevent an events default behavior, as described in the "Preventing Default Behavior" section.
`defaultPrevented`	This property returns `true` if the `preventDefault` method has been called on the event object and `false` otherwise.
`persist()`	This method is called to present React from reusing the event object, which is important for asynchronous operations, as described in the "Avoiding the Event Reuse Pitfall" section.

In Listing 12-10, I have updated the `handleEvent` method so that it uses the event object that React provides to update the component's state.

Listing 12-10. Receiving an Event Object in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    handleEvent = (event) => {
        this.setState({ message:  `Event: ${event.type} `});
    }

    render() {
        return  <div className="m-2">
                    <div className="h4 bg-primary text-white text-center p-2">
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ this.handleEvent }>
                                Click Me
                        </button>
                    </div>
                </div>
    }
}
```

I have added an event parameter to the handleEvent method, which I use to include the value of the type property in the message that is displayed to the user, as shown in Figure 12-4.

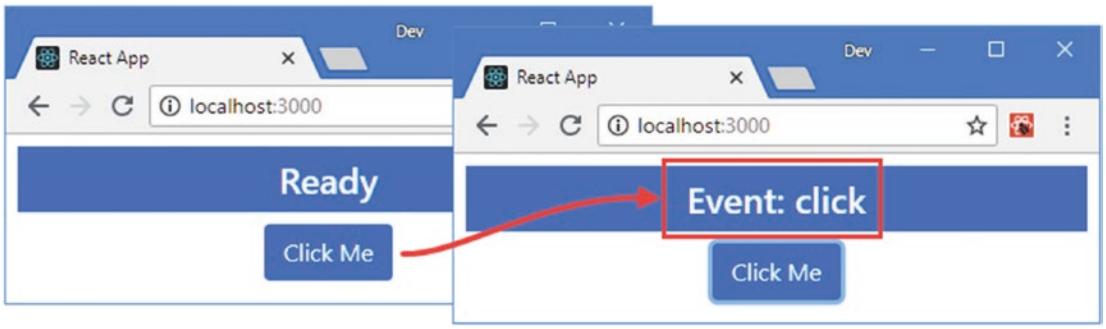

Figure 12-4. *Receiving an event object*

Differentiating Between Event Types

React always provides a SyntheticEvent object when it invokes an event handling function, which can cause confusion if you are accustomed to using the instanceof keyword to differentiate between events created by the DOM API. In Listing 12-11, I have changed the button element so the handleEvent method is used to respond to MouseUp and MouseDown events.

Listing 12-11. Differentiating Events in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready"
        }
    }

    handleEvent = (event) => {
        if (event.type === "mousedown") {
            this.setState({ message: "Down"});
        } else {
            this.setState({ message: "Up"});
        }
    }
    render() {
        return  <div className="m-2">
                    <div className="h4 bg-primary text-white text-center p-2">
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onMouseDown={ this.handleEvent }
                            onMouseUp={ this.handleEvent } >
                                Click Me
                        </button>
                    </div>
            </div>
    }
}
```

The handleEvent method uses the type property to determine which event is being handled and updates the message value accordingly. When you press the mouse button down, a mousedown event is triggered, and when you release, a mouseup event is triggered, as shown in Figure 12-5.

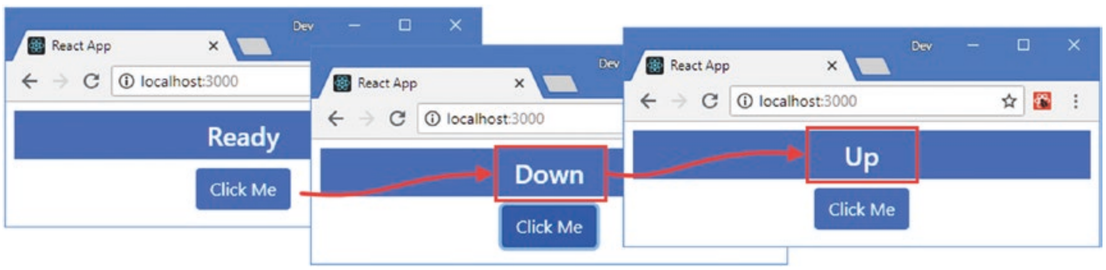

***Figure 12-5.** Differentiating event types*

Avoiding the Event Reuse Pitfall

React reuses SyntheticEvent objects and resets all the properties to null once an event has been handled. This can cause problems if you are relying on asynchronous updates to state data, as described in Chapter 11. Listing 12-12 demonstrates the problem.

Listing 12-12. Using an Event Object Asynchronously in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready",
            counter: 0
        }
    }
    handleEvent = (event) => {
        this.setState({ counter: this.state.counter + 1},
            () => this.setState({ message: `${event.type}: ${this.state.counter}`}));
    }

    render() {
        return  <div className="m-2">
                    <div className="h4 bg-primary text-white text-center p-2">
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ this.handleEvent } >
                                Click Me
                        </button>
                    </div>
                </div>
    }
}
```

The handleEvent method uses the setState method's callback feature to update the message property after an update to the counter property has been applied. The value assigned to the message property includes the event object's type property, which is a problem because that property will be set to null by the time the setState callback function is invoked, which you can see by clicking the button, as shown in Figure 12-6.

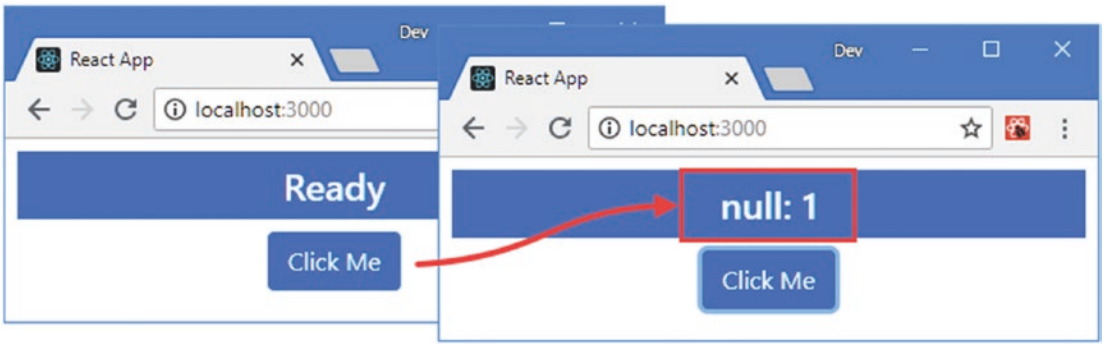

Figure 12-6. *Asynchronously using event objects*

The persist method is used to prevent React from resetting the event object, as shown in Listing 12-13.

Listing 12-13. Persisting an Event Object in the App.js File in the src Folder

```
...
handleEvent = (event) => {
    event.persist();
    this.setState({ counter: this.state.counter + 1},
        () => this.setState({ message: `${event.type}: ${this.state.counter}`}));
}
...
```

The result is that the event's properties can be read from the `setState` method's callback function, producing the result shown in Figure 12-7.

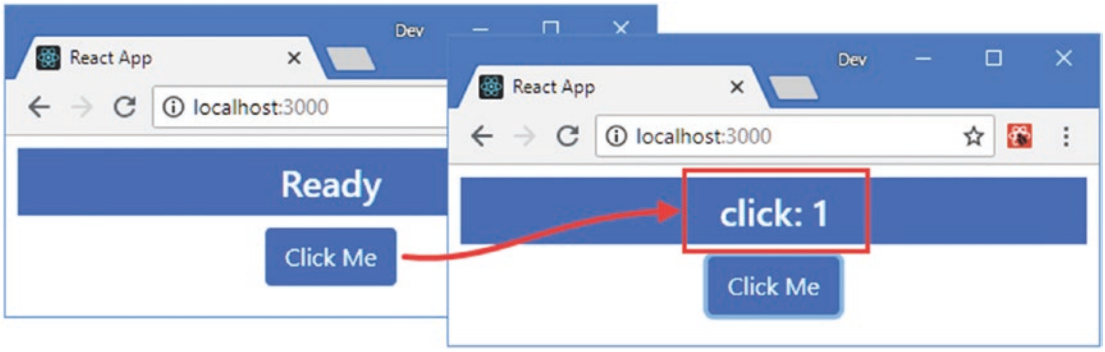

Figure 12-7. *Persisting an event*

Invoking Event Handlers with a Custom Argument

Event handlers are often more useful if they are provided with a custom argument, instead of the `SytheticEvent` object that React provides by default. To demonstrate why the event object isn't always useful, I added another `button` element to the content rendered by the `App` component and set up the event handler so that it uses the event to determine which button has been clicked, as shown in Listing 12-14.

Listing 12-14. Identifying the Source of an Event in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready",
            counter: 0,
            theme: "secondary"
        }
    }

    handleEvent = (event) => {
        event.persist();
        this.setState({
            counter: this.state.counter + 1,
            theme: event.target.innerText === "Normal" ? "primary" : "danger"
        }, () => this.setState({ message: `${event.type}: ${this.state.counter}`}));
    }
    render() {
        return  <div className="m-2">
                    <div className={ `h4 bg-${this.state.theme}
                            text-white text-center p-2`}>
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ this.handleEvent } >
                                Normal
                        </button>
                        <button className="btn btn-danger m-1"
                            onClick={ this.handleEvent } >
                                Danger
                        </button>
                    </div>
            </div>
    }
}
```

The problem with this approach is that the event handler has to understand the significance of the content rendered by the component. In this case, that means knowing that the value of the innerText property can be used to work out the source of the event and determine the value for the theme state data property. This can be difficult to manage if the content rendered by the component changes or if there are multiple interactions that can produce the same result. A more elegant approach is to use an inline expression for the event handler property that invokes the handler method and provides it with the information it needs, as shown in Listing 12-15.

Listing 12-15. Invoking a Handler with a Custom Argument in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready",
            counter: 0,
            theme: "secondary"
        }
    }

    handleEvent = (event, newTheme) => {
        event.persist();
        this.setState({
            counter: this.state.counter + 1,
            theme: newTheme
        }, () => this.setState({ message: `${event.type}: ${this.state.counter}`}));
    }
    render() {
        return  <div className="m-2">
                    <div className={ `h4 bg-${this.state.theme}
                            text-white text-center p-2`}>
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ (e) => this.handleEvent(e, "primary") } >
                                Normal
                        </button>
                        <button className="btn btn-danger m-1"
                            onClick={ (e) => this.handleEvent(e, "danger") } >
                                Danger
                        </button>
                    </div>
            </div>
    }
}
```

The result is the same, but the handleEvent method doesn't have to inspect the element that triggered the event in order to set the theme property. To see the effect of setting the theme, click either of the button elements, as shown in Figure 12-8.

■ **Tip** If your handler method doesn't need the event object, then you can use the inline expression to call the handler without it: `() => handleEvent("primary")`.

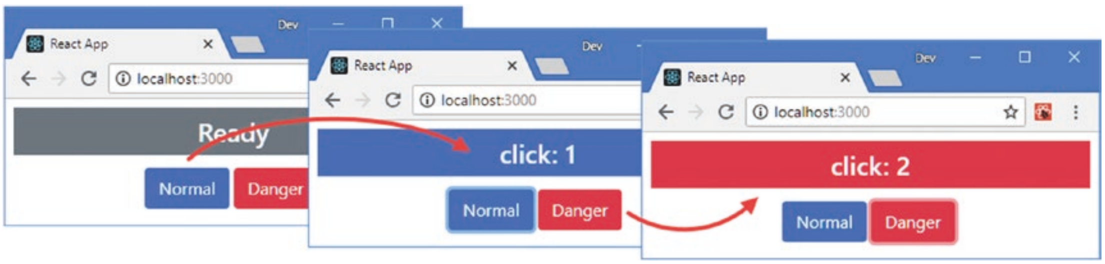

Figure 12-8. *Using a custom argument*

Preventing Default Behavior

Some events have behavior that the browser performs by default. The default behavior for clicking a checkbox, for example, is to toggle the status of that checkbox. The preventDefault method can be called on event objects to prevent the default behavior, and to demonstrate, I added a checkbox element to the content that will be toggled only after one of the button elements has been clicked, as shown in Listing 12-16.

Listing 12-16. Preventing Default Behavior in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready",
            counter: 0,
            theme: "secondary"
        }
    }

    handleEvent = (event, newTheme) => {
        event.persist();
        this.setState({
            counter: this.state.counter + 1,
            theme: newTheme
        },  () => this.setState({ message: `${event.type}: ${this.state.counter}`}));
    }

    toggleCheckBox = (event) => {
        if (this.state.counter === 0) {
            event.preventDefault();
        }
    }

    render() {
        return  <div className="m-2">
                    <div className="form-check">
                        <input className="form-check-input" type="checkbox"
                             onClick={ this.toggleCheckBox }/>
                        <label>This is a checkbox</label>
                    </div>

                    <div className={ `h4 bg-${this.state.theme}
                            text-white text-center p-2`}>
                        { this.state.message }
                    </div>
                    <div className="text-center">
                        <button className="btn btn-primary"
                            onClick={ (e) => this.handleEvent(e, "primary") } >
                                Normal
                        </button>
                        <button className="btn btn-danger m-1"
                            onClick={ (e) => this.handleEvent(e, "danger") } >
                                Danger
                        </button>
                    </div>
            </div>
    }
}
```

The onClick property on the input element tells React to invoke the toggleCheckBox method when the user clicks the checkbox. The preventDefault method is called on the event if the value of the counter state data property is zero, with the result that the checkbox cannot be toggled until after a button has been clicked, as shown in Figure 12-9.

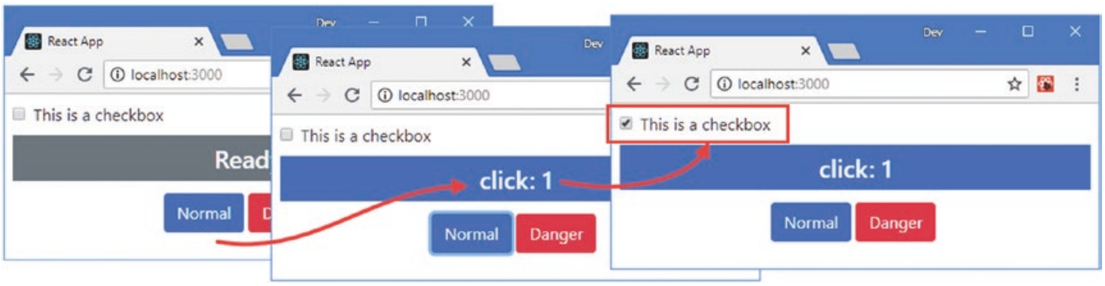

Figure 12-9. *Preventing event default behavior*

Managing Event Propagation

Events have a lifecycle that allows an element's ancestors to receive events triggered by their descendants and also to intercept events before they reach an element. In the sections that follow, I describe how events are propagated through HTML elements and explain the effect this has on React applications, using the properties and methods defined by the SyntheticEvent that are described in Table 12-4.

Table 12-4. *The SyntheticEvent Properties and Methods for Event Propagation*

Name	Description
eventPhase	This property returns the propagation phase of an event. However, the way that React handles events means this property is not useful, as described in the "Determining the Event Phase" section.
bubbles	This property returns true if the event will enter the bubble phase.
currentTarget	This property returns an object that represents the element whose event handler is processing the event.
stopPropagation()	This method is called to stop event propagation, as described in the "Stopping Event Propagation" section.
isPropagationStopped()	This method returns true if stopPropagation has been called on an event.

Understanding the Target and Bubble Phases

When an event is first triggered, it enters the *target phase*, where event handlers applied to the element that is the source of the event are invoked. Once those event handlers are complete, the event enters the *bubble phase*, where the event works its way up the chain of ancestor elements and is used to invoke any handlers that have been applied for that type of event. To help demonstrate these phases, I added a file called ThemeButton.js to the src folder and used it to define the component shown in Listing 12-17.

Listing 12-17. The Contents of the ThemeButton.js File in the src Folder

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

export class ThemeButton extends Component {

    handleClick = (event) => {
        console.log(`ThemeButton: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
        this.props.callback(this.props.theme);
    }

    render() {
        return  <span className="m-1" onClick={ this.handleClick }>
                    <button className={`btn btn-${this.props.theme}`}
                        onClick={ this.handleClick }>
                            Select {this.props.theme } Theme
                    </button>
                </span>
    }
}
```

This component renders a span element that contains a button and is provided with a theme prop, which specifies a Bootstrap CSS theme name, and a callback prop that is invoked to select the prop. The onClick property has been applied to both the span and button elements. In Listing 12-18, I updated the App component to use the ThemeButton component and to remove some of the code used in earlier examples.

Listing 12-18. Applying a Component in the App.js File in the src Folder

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

export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready",
            counter: 0,
            theme: "secondary"
        }
    }

    selectTheme = (newTheme) => {
        this.setState({
            theme: newTheme,
            message: `Theme: ${newTheme}`
        });
    }
    render() {
        return  (
            <div className="m-2">
                <div className={ `h4 bg-${this.state.theme}
                        text-white text-center p-2`}>
                    { this.state.message }
                </div>
                <div className="text-center">
                    <ThemeButton theme="primary" callback={ this.selectTheme } />
                    <ThemeButton theme="danger" callback={ this.selectTheme } />
                </div>
            </div>
        )
    }
}
```

Click either of the button elements, and you will see the following output in the browser's JavaScript console:

```
...
ThemeButton: Type: click Target: BUTTON CurrentTarget: BUTTON
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
...
```

There are two messages in the console because there are two onClick properties in the content rendered by the ThemeButton component. The first message is generated during the target phase when the event is processed by the handlers of the element that triggered it, which is the button element in this example. The event then enters the bubble phase, where it propagates up through the button element's ancestor and invokes any suitable event handlers. In the example, the span element that is the parent of the button also has an onClick property, which results in two calls to the handleClick method and two messages written to the console.

■ **Tip** Not all types of event have a bubble phase. As a rule of thumb, events that are specific to a single element—such as gaining and losing focus—do not bubble. Events that apply to multiple elements—such as clicking a region of the screen that is occupied by multiple elements—will bubble. You can check to see whether a specific event is going to go through the bubble phase by reading the bubbles property of the event object.

The bubble phase extends beyond the content rendered by the component and propagates throughout the entire hierarchy of HTML elements. To demonstrate, I added onClick handlers to elements rendered by the App component that will receive the click event when it bubbles up from the button element rendered by the ThemeButton component, as shown in Listing 12-19.

Listing 12-19. Adding Event Handlers in the App.js File in the src Folder

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

```
export default class App extends Component {

    constructor(props) {
        super(props);
        this.state = {
            message: "Ready",
            counter: 0,
            theme: "secondary"
        }
    }

    selectTheme = (newTheme) => {
        this.setState({
            theme: newTheme,
            message: `Theme: ${newTheme}`
        });
    }

    handleClick= (event) => {
        console.log(`App: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
    }

    render() {
        return  (
            <div className="m-2" onClick={ this.handleClick }>
                    <div className={ `h4 bg-${this.state.theme}
                            text-white text-center p-2`}>
                        { this.state.message }
                    </div>
                    <div className="text-center" onClick={ this.handleClick }>
                        <ThemeButton theme="primary" callback={ this.selectTheme } />
                        <ThemeButton theme="danger" callback={ this.selectTheme } />
                    </div>
            </div>
        )
    }
}
```

I added the onClick property to two div elements, and when you click one of the buttons, you will see the following series of messages displayed in the browser's JavaScript console (some browsers group the last two messages together since they are the same):

```
...
ThemeButton: Type: click Target: BUTTON CurrentTarget: BUTTON
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
App: Type: click Target: BUTTON CurrentTarget: DIV
App: Type: click Target: BUTTON CurrentTarget: DIV
...
```

The SyntheticEvent object provides the currentTarget property, which returns the element whose event handler is being invoked, as opposed to the target property, which returns the element that triggered the event.

```
...
console.log(`ThemeButton: Type: ${event.type} `
    + `Target: ${event.target.tagName} `
    + `CurrentTarget: ${event.currentTarget.tagName}`);
...
```

These messages show the target and bubble phases of the click event as it is propagated up the hierarchy of HTML elements, as shown in Figure 12-10.

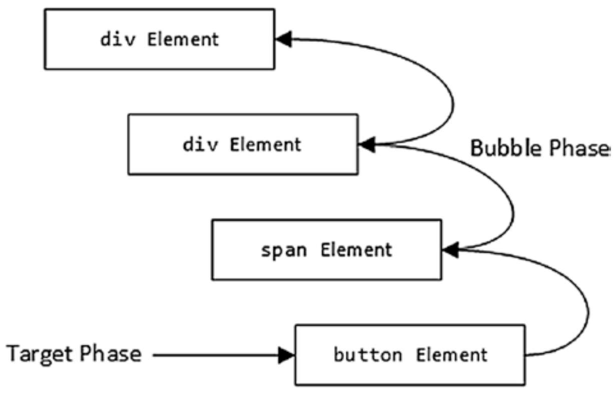

Figure 12-10. *The target and bubble phases of the event*

EVENTS AND ELEMENTS THAT APPLY COMPONENTS

Event handling is performed by the HTML elements that are rendered by components and excludes the custom HTML elements that are used to apply components. Adding event handler properties, such as onClick to the ThemeButton element, for example, has no effect. No error is reported, but the custom element is excluded from the HTML that is displayed by the browser, and the handler will never be invoked.

Understanding the Capture Phase

The capture phase provides an opportunity for elements to process events before the target phase. During the capture phase, the browser starts with the body element and works its way down the hierarchy of elements toward the target, following the opposite path to the bubble phase, and gives each element the chance to process the event, as shown in Figure 12-11.

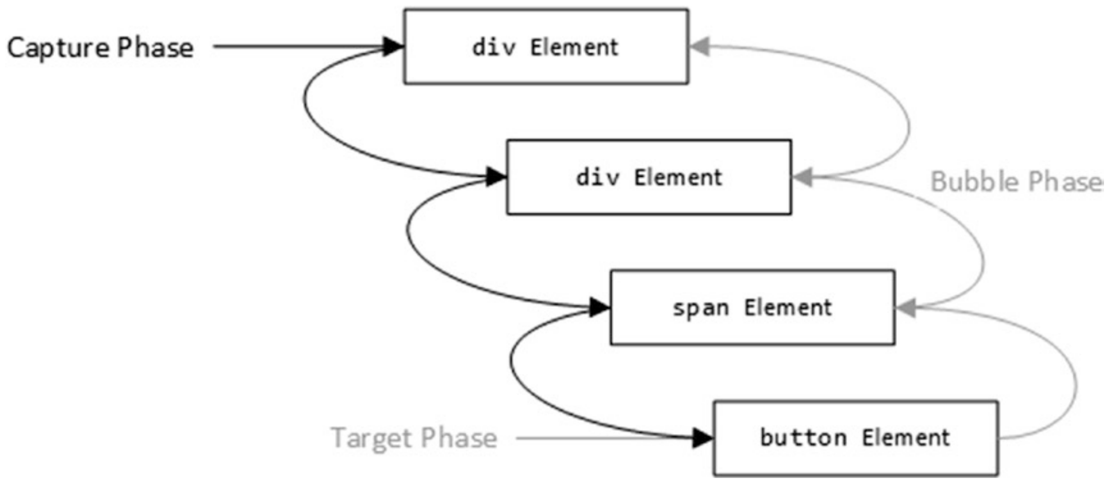

Figure 12-11. *The event capture phase*

A separate property is required to tell React that an event handler should be applied in the capture phase, as shown in Listing 12-20.

Listing 12-20. Capturing an Event in the ThemeButton.js File in the src Folder

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

export class ThemeButton extends Component {

    handleClick = (event) => {
        console.log(`ThemeButton: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
        this.props.callback(this.props.theme);
    }

    render() {
        return  <span className="m-1" onClick={ this.handleClick }
                        onClickCapture={ this.handleClick }>
                    <button className={`btn btn-${this.props.theme}`}
                        onClick={ this.handleClick }>
                            Select {this.props.theme } Theme
                    </button>
                </span>
    }
}
```

For each event handling property, such as onClick, there is a corresponding capture property, onClickCapture, that receives events in the capture phase. In the listing, I applied the onClickCapture property to the span element and specified the handleClick method in the expression. The result is that the span element will receive click events in the capture and bubble phases as the event works its way down the hierarchy of HTML elements and goes back up again. Clicking either of the button elements will produce an additional message in the browser's JavaScript console.

```
...
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
ThemeButton: Type: click Target: BUTTON CurrentTarget: BUTTON
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
App: Type: click Target: BUTTON CurrentTarget: DIV
App: Type: click Target: BUTTON CurrentTarget: DIV
...
```

Determining the Event Phase

The handleClick method defined by the ThemeButton component will handle events several times for each click event, and it moves from the capture to the target and then the bubble phase. Each time the handleClick method is called, it invokes the function prop provided by the parent component, which has the effect of repeatedly changing the value of the App component's theme state property. This is a harmless effect, but in real projects, repeatedly invoking a callback can cause problems, and it is bad practice for a child component to assume that props can be invoked without issue. To highlight the problem, I added a statement to the ThemeButton component's handleEvent method that writes a message to the browser's JavaScript console when the function prop is invoked, as shown in Listing 12-21.

Listing 12-21. Adding a Debugging Message in the ThemeButton.js File in the src Folder

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

export class ThemeButton extends Component {

    handleClick = (event) => {
        console.log(`ThemeButton: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
        console.log("Invoked function prop");
        this.props.callback(this.props.theme);
    }

    render() {
        return  <span className="m-1" onClick={ this.handleClick }
                        onClickCapture={ this.handleClick }>
                    <button className={`btn btn-${this.props.theme}`}
                        onClick={ this.handleClick }>
                            Select {this.props.theme } Theme
                    </button>
                </span>
    }
}
```

Click one of the button's presented by the example application, and you will see that the function prop is invoked for each of the three phases that the `click` event goes through.

```
...
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
Invoked function prop
ThemeButton: Type: click Target: BUTTON CurrentTarget: BUTTON
Invoked function prop
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
Invoked function prop
App: Type: click Target: BUTTON CurrentTarget: DIV
App: Type: click Target: BUTTON CurrentTarget: DIV
...
```

The `SythenticEvent` object that React uses defines an `eventPhase` property, which returns the value of the corresponding property from the native DOM API event object. Unfortunately, the value of that property always indicates that the event is in the bubble phase because React intercepts the native event and uses it to simulate the three propagation phases. As a consequence, a little more work is required to identify event phases.

The first step is to identify events in the capture phase, which can be done by using a different handler method or providing an additional argument to the common handler, which is the approach that I have taken in Listing 12-22.

Listing 12-22. Identifying Capture Phase Events in the ThemeButton.js File in the src Folder

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

export class ThemeButton extends Component {

    handleClick = (event, capturePhase = false) => {
        console.log(`ThemeButton: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
        if (capturePhase) {
            console.log("Skipped function prop: capture phase");
        } else {
            console.log("Invoked function prop");
            this.props.callback(this.props.theme);
        }
    }

    render() {
        return  <span className="m-1" onClick={ this.handleClick }
                        onClickCapture={ (e) => this.handleClick(e, true) }>
                    <button className={`btn btn-${this.props.theme}`}
                        onClick={ this.handleClick }>
                            Select {this.props.theme } Theme
                    </button>
                </span>
    }
}
```

I used an inline expression for the `onClickCapture` property that receives the `SythenticEvent` object and uses it to invoke the `handleClick` method, along with an additional argument that indicates the event is in the capture phase. Within the `handleClick` method, I check the value of the `capturePhase` parameter to identify events in their capture phase.

Separating the target and bubble phases is more difficult because events in both phases are handled by the `onClick` property. The most reliable way to determine the phase is to see whether the values for the `target` and `currentTarget` properties are different and to see whether the `bubbles` property is `true`. If the object returned by the `currentTarget` is different from the `target` value and the event has a bubble phase, then it is reasonable to assume that the event is bubbling, as shown in Listing 12-23.

Listing 12-23. Identifying Bubble Phase Events in the ThemeButton.js File in the src Folder

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

export class ThemeButton extends Component {

    handleClick = (event, capturePhase = false) => {
        console.log(`ThemeButton: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
        if (capturePhase) {
            console.log("Skipped function prop: capture phase");
        } else if (event.bubbles && event.currentTarget !== event.target) {
            console.log("Skipped function prop: bubble phase");
        } else {
            console.log("Invoked function prop");
            this.props.callback(this.props.theme);
        }
    }

    render() {
        return  <span className="m-1" onClick={ this.handleClick }
                    onClickCapture={ (e) => this.handleClick(e, true) }>
                <button className={`btn btn-${this.props.theme}`}
                    onClick={ this.handleClick }>
                        Select {this.props.theme } Theme
                </button>
            </span>
    }
}
```

When you click a button, you will see the following sequence of messages in the browser's JavaScript console, indicating that each phase has been identified and that the function prop has been called only in the target phase.

```
...
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
Skipped function prop: capture phase
ThemeButton: Type: click Target: BUTTON CurrentTarget: BUTTON
Invoked function prop
ThemeButton: Type: click Target: BUTTON CurrentTarget: SPAN
Skipped function prop: bubble phase
```

```
App: Type: click Target: BUTTON CurrentTarget: DIV
App: Type: click Target: BUTTON CurrentTarget: DIV
...
```

These messages also confirm the order of the event's phases: capture, target, and then bubble.

Stopping Event Propagation

Understanding event phases can also be important if you want to disrupt the normal propagation sequence and prevent elements from receiving events. In Listing 12-24, I have changed the ThemeButton component so that it intercepts click events in the capture phase and stops them from reaching the target element.

Listing 12-24. Stopping Event Propagation in the ThemeButton.js File in the src Folder

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

export class ThemeButton extends Component {

    handleClick = (event, capturePhase = false) => {
        console.log(`ThemeButton: Type: ${event.type} `
            + `Target: ${event.target.tagName} `
            + `CurrentTarget: ${event.currentTarget.tagName}`);
        if (capturePhase) {
            if (this.props.theme === "danger") {
                event.stopPropagation();
                console.log("Stopped event");
            } else {
                console.log("Skipped function prop: capture phase");
            }
        } else if (event.bubbles && event.currentTarget !== event.target) {
            console.log("Skipped function prop: bubble phase");
        } else {
            console.log("Invoked function prop");
            this.props.callback(this.props.theme);
        }
    }

    render() {
        return  <span className="m-1" onClick={ this.handleClick }
                    onClickCapture={ (e) => this.handleClick(e, true) }>
                    <button className={`btn btn-${this.props.theme}`}
                        onClick={ this.handleClick }>
                            Select {this.props.theme } Theme
                    </button>
                </span>
    }
}
```

The `onClickCapture` property on the `span` element will invoke the `handleClick` method when it receives a `click` event in the capture phase. The `stopPropagation` method is called when the value of the `theme` prop is `danger`, which prevents the event from reaching the `button` element and has the effect of preventing the user from selecting the `danger` theme, as illustrated in Figure 12-12.

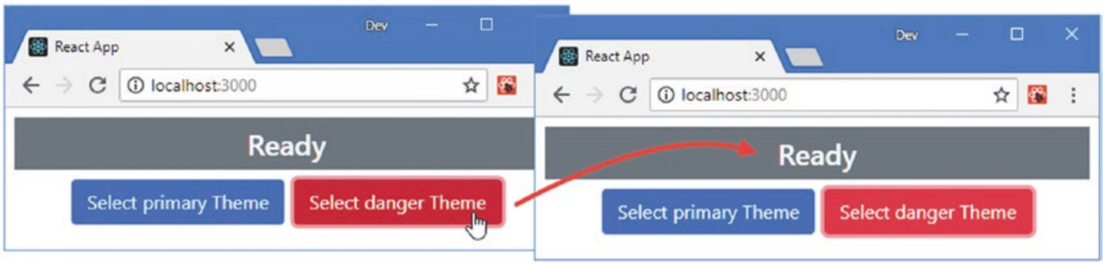

Figure 12-12. *Stopping an event*

Summary

In this chapter, I described the features that React provides for working with events. I demonstrated the different ways that handler functions can be defined, showed you how to work with event objects, and showed how to use custom arguments instead. I also explained how React events are not the same as DOM API events, even though they are similar and closely related. I finished the chapter by introducing the event lifecycle and showing you how events are propagated. In the next chapter, I describe the component lifecycle and explain how state data changes are reconciled.