

Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.');" return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
```



```
function handleClick(e) {

  e.preventDefault();
  console.log('The link was clicked.');
```



```
}

return (
  <a href="#" onClick={handleClick}>
    Click me
  </a>
);
}
```

Here, `e` is a synthetic event. React defines these synthetic events according to the [W3C spec](#), so you don't need to worry about cross-browser compatibility. React events do not work exactly the same as native events. See the [SyntheticEvent](#) reference guide to learn more.

When using React, you generally don't need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an [ES6 class](#), a common pattern is for an event handler to be a method on the class. For example, this `Toggle` component renders a button that lets the user toggle between "ON" and "OFF" states:

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button type="button" value="Toggle" onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```



```

    <button onClick={this.handleClick}>

        {this.state.isToggleOn ? 'ON' : 'OFF'}
    </button>
  );
}
}

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

```

Try it on CodePen

You have to be careful about the meaning of `this` in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, `this` will be `undefined` when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without `()` after it, such as `onClick={this.handleClick}`, you should bind that method.

If calling `bind` annoys you, there are two ways you can get around this. If you are using the experimental public class fields syntax, you can use class fields to correctly bind callbacks:

```

class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}

```



This syntax is enabled by default in [Create React App](#).

If you aren't using class fields syntax, you can use an [arrow function](#) in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the `LoggingButton` renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

Passing Arguments to Event Handlers



Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if `id` is the row ID, either of the following would work:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use [arrow functions](#) and [Function.prototype.bind](#) respectively.



In both cases, the `e` argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with `bind` any further arguments are automatically forwarded.

Is this page useful?  

[Edit this page](#)

[Previous article](#)

[State and Lifecycle](#)

[Next article](#)

[Conditional Rendering](#)

