# イベント処理

React でのイベント処理は DOM 要素のイベントの処理と非常に似ています。いくつかの文法的な違いがあります：

- React のイベントは小文字ではなく camelCase で名付けられています。
- JSX ではイベントハンドラとして文字列ではなく関数を渡します。

例えば、以下の HTML：

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

は、React では少し異なります：

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

別の違いとして、React では `false` を返してもデフォルトの動作を抑止することができません。明示的に `preventDefault` を呼び出す必要があります。例えば、プレーンな HTML では、「新しいページを開く」というリンクのデフォルト動作を抑止するために次のように書くことができます。

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

React では、代わりに次のようになります：

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

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

ここで、`e` は合成 (synthetic) イベントです。React はこれらの合成イベントを W3C の仕様に則って定義しているので、ブラウザ間の互換性を心配する必要はありません。詳細については、`SyntheticEvent` のリファレンスガイドを参照してください。

React を使う場合、一般的には DOM 要素の生成後に `addEventListener` を呼び出してリスナーを追加するべきではありません。代わりに、要素が最初にレンダリングされる際にリスナーを指定するようにしてください。

コンポーネントを ES6 のクラスを使用して定義した場合、一般的なパターンではイベントハンドラはクラスのメソッドになります。例えば、以下の `Toggle` コンポーネントはユーザーが “ON” 状態 “OFF” 状態を切り替えられるようなボタンをレンダーします。

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

```
}

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

**Try it on CodePen**

JSX のコールバックにおける `this` の意味に注意しなければなりません。JavaScript では、クラスのメソッドはデフォルトではバインドされません。`this.handleClick` へのバインドを忘れて `onClick` に渡した場合、実際に関数が呼ばれた時に `this` は `undefined` となってしまいます。

これは React に限った動作ではなく、JavaScript における関数の仕組みの一部です。一般的に、`onClick={this.handleClick}` のように `()` を末尾に付けずに何らかのメソッドを参照する場合、そのメソッドはバインドしておく必要があります。

`bind` の呼び出しが苦痛なら、それを回避する方法が 2 つあります。実験的なパブリッククラスフィールド構文を使用しているなら、コールバックを正しくバインドするのにクラスフィールドを利用できます：

```
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>
    );
  }
}
```

この構文は、Create React App ではデフォルトで有効です。

クラスフィールド構文を使用していない場合、コールバック内でアロー関数を使用することもできます：

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

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

この構文での問題は、`LoggingButton` がレンダリングされるたびに異なるコールバック関数が毎回作成されるということです。大抵のケースではこれは問題ありません。しかし、このコールバックが props の一部として下層のコンポーネントに渡される場合、それら下層コンポーネントが余分に再描画されることになります。 一般的にはコンストラクタでバインドするかクラスフィールド構文を使用して、この種のパフォーマンスの問題を避けるようおすすめします。

---

## イベントハンドラに引数を渡す

ループ内では、イベントハンドラに追加のパラメータを渡したくなることがよくあります。例えば、`id` という行の ID がある場合、以下のどちらでも動作します：

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

上記の 2 行は等価であり、上側ではアロー関数が、下側では `Function.prototype.bind` が使われています。

どちらの場合でも、React イベントを表す `e` という引数は ID の次の 2 番目の引数として渡されることになります。アロー関数では `e` を明示的に渡す必要がありますが、`bind` の場合には `id` 以降の追加の引数は自動的に転送されます。

このページを編集する