コンポーネントと props

コンポーネントにより UI を独立した再利用できる部品に分割し、部品それぞれを分離して考えることができるようになります。このページではコンポーネントという概念の導入を行います。詳細な API リファレンスはこちらで参照できます。
概念的には、コンポーネントは JavaScript の関数と似ています。（“props” と呼ばれる）任意の入力を受け取り、画面上に表示すべきものを記述する React 要素を返します。

関数コンポーネントとクラスコンポーネント

コンポーネントを定義する最もシンプルな方法は JavaScript の関数を書くことです：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

この関数は、データの入った “props”（「プロパティ」の意味）というオブジェクトを引数としてひとつ受け取り、React 要素を返すので、有効な React コンポーネントです。これは文字通り JavaScript の関数ですので、このようなコンポーネントのことを “関数コンポーネント (function component)” と呼びます。
コンポーネントを定義するために ES6 クラスも使用できます：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

上記 2 つのコンポーネントは React の視点からは等価です。
クラスには次のセクションで説明するいくつかの追加の機能があります。それまでは、簡単なので関数コンポーネントの方を使いましょう。

コンポーネントのレンダー

前節では、DOM のタグを表す React 要素のみを扱いました：

```
const element = <div />;
```

しかし、要素はユーザ定義のコンポーネントを表すこともできます：

```
const element = <Welcome name="Sara" />;
```

React がユーザ定義のコンポーネントを見つけた場合、JSX の属性を単一のオブジェクトとしてこのコンポーネントに渡します。このオブジェクトのことを “props” と呼びます。
例えば以下のコードではページ上に “Hello, Sara” を表示します：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Try it on CodePen

この例で何が起こるのかおさらいしてみましょう。

1. `<Welcome name="Sara" />` という要素を引数として `ReactDOM.render()` を呼び出します。
2. React は `Welcome` コンポーネントを呼び出し、その時に props として `{name: 'Sara'}` を渡します。
3. `Welcome` コンポーネントは出力として `<h1>Hello, Sara</h1>` 要素を返します。
4. React DOM は `<h1>Hello, Sara</h1>` に一致するよう、DOM を効率的に更新します。

> **補足: コンポーネント名は常に大文字で始めてください。**

> React は小文字で始まるコンポーネントを DOM タグとして扱います。例えば、`<div />` は HTML の div タグを表しますが、`<Welcome />` はコンポーネントを表しており、スコープ内に `Welcome` が存在する必要があります。
> この規約の背後にある理由については JSX を深く理解するを参照してください。

コンポーネントを組み合わせる

コンポーネントは自身の出力の中で他のコンポーネントを参照できます。これにより、どの詳細度のレベルにおいても、コンポーネントという単一の抽象化を利用できます。ボタン、フォーム、ダイアログ、画面：React アプリでは、これらは共通してコンポーネントとして表現されます。
例えば、`Welcome` を何回もレンダリングする `App` コンポーネントを作成できます：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

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

Try it on CodePen

典型的には、新規の React アプリは階層の一番上に単一の `App` コンポーネントを持っています。しかし、既存のアプリに React を統合する場合は、`Button` のような小さなコンポーネントからボトムアップで始め、徐々にビューの階層構造の頂上に向かって進んでいってもよいでしょう。

コンポーネントの抽出

コンポーネントをより小さなコンポーネントに分割することを恐れないでください。
例えば、この `Comment` コンポーネントについて考えましょう：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Try it on CodePen

これは props として `author`（オブジェクト）、`text`（文字列）、および `date`（日付）を受け取り、ソーシャルメディアサイトにおける 1 つのコメントを表します。
これだけのネストがあるため、このコンポーネントの変更には苦労を伴い、また内部の個々の部品を再利用することも困難です。ここからいくつかのコンポーネントを抽出しましょう。
まず、`Avatar` を抽出します：

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />

  );
}
```

`Avatar` は、自身が `Comment` の内側でレンダリングされているということを知っている必要はありません。なので props の名前として、`author` ではなく `user` というもっと一般的な名前を付けました。
コンポーネントが使用されるコンテキストではなく、コンポーネント自身からの観点で props の名前を付けることをお勧めします。
これで `Comment` をほんの少しシンプルにできます：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

次に、ユーザ名の隣の `Avatar` をレンダリングするために使われる、`UserInfo` コンポーネントを抽出しましょう。

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

これにより `Comment` をさらにシンプルにできます：

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Try it on CodePen
コンポーネントの抽出は最初は面倒な仕事のように思えますが、再利用できるコンポーネントをパレットとして持っておくことは、アプリケーションが大きくなれば努力に見合った利益を生みます。役に立つ経験則として、UI の一部（`Button`、`Panel`、`Avatar` など）が複数回使われている場合、またはその UI 自体が複雑（`App`、`FeedStory`、`Comment` など）である場合、それらは再利用可能なコンポーネントにする有力な候補であるといえます。

Props は読み取り専用
コンポーネントを関数で宣言するかクラスで宣言するかに関わらず、自分自身の props は決して変更してはいけません。この `sum` 関数を考えましょう：

```
function sum(a, b) {
  return a + b;
}
```

このような関数は入力されたものを変更しようとせず、同じ入力に対し同じ結果を返すので“純粋”であると言われます。
対照的に、以下の関数は自身への入力を変更するため純関数ではありません：

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

React は柔軟ですが、1 つだけ厳格なルールがあります：
全ての React コンポーネントは、自己の props に対して純関数のように振る舞わねばなりません。

もちろんアプリケーションの UI は動的で、時間に応じて変化するものです。next section では、“state” という新しい概念を紹介します。state により React コンポーネントは上述のルールを壊すことなく、時間と共にユーザのアクション、ネットワークのレスポンスや他の様々な事に反応して、出力を変更することができます。

このページを編集する