JSX を深く理解する

JSX とは、つまるところ `React.createElement(component, props, ...children)` の糖衣構文にすぎません。例として、次の JSX コードを見てみましょう。

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

これは以下のようにコンパイルされます。

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

子要素を持たない場合には、自己クローズ (self-closing) タグを利用することもできます。次のコードを見てください。

```
<div className="sidebar" />
```

これは以下のようにコンパイルされます。

```
React.createElement(
  'div',
  {className: 'sidebar'},
  null
)
```

具体的に JSX がどのように JavaScript へ変換されるのかをテストしたい場合は、オンライン Babel コンパイラで試すことができます。

React 要素の型を指定する

JSX タグの先頭の部分は、React 要素の型を決定しています。

大文字で始まる型は JSX タグが React コンポーネントを参照していることを示しています。このような JSX タグはコンパイルを経てその大文字で始まる変数を直接参照するようになります。つまり JSX の `<Foo />` 式を使用する場合、`Foo` がスコープになければなりません。

React がスコープ内にあること

JSX は `React.createElement` の呼び出しへとコンパイルされるため、`React` ライブラリは常に JSX コードのスコープ内にある必要があります。

例えば以下のコードでは、`React` も `CustomButton` も JavaScript から直接は参照されていませんが、両方ともインポートされていることが必要です。

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

JavaScript のバンドルツールを使わずに `<script>` タグから React を読み込んでいる場合は、`React` はグローバル変数として既にスコープに入っています。

JSX 型にドット記法を使用する

JSX の中においては、ドット記法を使うことによって React コンポーネントを参照することもできます。これは単一のモジュールがたくさんの React コンポーネントをエクスポートしているような場合に便利です。例えば、`MyComponents.DatePicker` というコンポーネントがあるのであれば、次のように JSX 内から直接利用することができます。

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
```

```
  return <MyComponents.DatePicker color="blue" />;
}
```

ユーザー定義のコンポーネントの名前は大文字で始めること

ある要素の型が小文字から始まっているような場合、それは `<div>` や `<span>` のような組み込みのコンポーネントを参照しており、これらはそれぞれ `'div'` や `'span'` といった文字列に変換されて `React.createElement` に渡されます。一方で `<Foo />` のように大文字で始まる型は `React.createElement(Foo)` にコンパイルされ、JavaScript ファイルにおいて定義あるいはインポートされたコンポーネントを参照します。

コンポーネントを命名するときには、大文字から始めるようにしてください。もしすでに小文字から始まるコンポーネントを作ってしまっていたら、JSX 内で利用する前にいちど大文字から始まる変数に代入しておきましょう。

例えば、以下のコードは期待通りには動きません。

```
import React from 'react';

// 間違った例。これはコンポーネントなので、大文字ではじめなければ行けません。
function hello(props) {
  // 正しい例。div は HTML タグなので、<div> と書くのは正解です。
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 間違った例。大文字ではじまっていないため、React は <hello /> を HTML タグと認識してしまいます。
  return <hello toWhat="World" />;
}
```

`hello` を `Hello` に書き換え、`<Hello />` を使って参照するようにすれば、このコードはきちんと動作するようになります。

```
import React from 'react';

// 正しい例。コンポーネントなので大文字からはじまっています。
function Hello(props) {
  // 正しい例。div は HTML タグなので、<div> と書くのは正解です。
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 正しい例。大文字ではじまっているため、React は <Hello /> がコンポーネントだと認識できます。
  return <Hello toWhat="World" />;
}
```

実行時に型を選択する

一般的な式を React の要素の型として使用することはできません。どうしても一般的な式を使って要素の型を示したいのであれば、まずその式を大文字から始まる変数に代入してから利用しましょう。これはプロパティの値に応じて異なるコンポーネントを表示し分けたいような場合によくあるケースです。

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 間違った例。式は JSX の型に指定できません。
  return <components[props.storyType] story={props.story} />;
}
```

大文字から始まる変数に型を代入することで、上のコードをきちんと動作するようにしてみましょう。

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 正しい例。大文字で始まる変数は JSX の型に指定できます。
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

JSX における props

JSX で props を指定するやり方はいくつかあります。

プロパティとしての JavaScript 式
任意の JavaScript 式は `{}` で囲むことによって props として渡すことができます。例として次の JSX を見てみましょう。

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

この `MyComponent` について、`props.foo` の値は `10` 、つまり `1 + 2 + 3 + 4` という式が評価されます。
`if` 文や `for` 文は JavaScript においては式ではないため、JSX 内で直接利用することはできません。代わりに JSX の近くで間接的に利用してみてください。次がその例です。

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

これについては、条件付きレンダーとループでさらに深く学ぶことができます。

文字列リテラル
文字列リテラルを props として渡すことができます。そのため以下のふたつの JSX の式はまったく等しいものとなります。

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

文字列リテラルを渡す際、その値における HTML エスケープは元の形に復元されます。そのため以下のふたつの JSX の式もまったく等しいものとなります。

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

この振る舞いは多くの場合それほど重要なものではありませんが、包括的な解説の一環としてここでは触れておきます。

プロパティのデフォルト値は true
プロパティに値を与えない場合、デフォルトの値は `true` となります。そのため以下のふたつの JSX の式はまったく等しいものとなります。

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

特別な理由がある場合を除いて、このように値を省略することは推奨していません。ES6 におけるオブジェクトの簡略表記においては、`{foo}` は `{foo: true}` ではなく `{foo: foo}` を意味するため、HTML の動作に似せて作られたこの機能はかえって混乱をきたす可能性があります。

属性の展開
`props` オブジェクトがあらかじめ存在しており、それを JSX に渡したいような場合は `...` を「スプレッド」演算子として使用することで、props オブジェクトそのものを渡すことができます。そのため以下のふたつの JSX の式はまったく等しいものとなります。

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

また、コンポーネントが利用する適当なプロパティを取り出しつつ、残りのすべてのプロパティに対してスプレッド演算子を利用することもできます。

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
```

```
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};
```

上の例では、 `kind` プロパティは無事に取り出され、DOM 中の `<button>` 要素には**渡されていません**。 残りのプロパティは `...other` オブジェクトにより渡され、このコンポーネントを柔軟性の高いものにしています。上記のコードは `onClick` や `children` プロパティを渡していることが見てとれるはずです。
スプレッド演算子は便利ではありますが、コンポーネント内で利用しないプロパティを不用意に渡してしまったり、意味をなさない HTML 属性を DOM に渡してしまうようなことが容易にありえます。そのためこの構文は慎重に利用してください。

JSX における子要素

開始タグと終了タグの両方を含む JSX 式においては、タグに囲まれた部分は、 `props.children` という特別なプロパティとして渡されます。このような子要素を渡す方法はいくつかあります。

文字列リテラル

開始タグと終了タグの間に文字列を挟んでいる場合、その文字列が `props.children` となります。これは HTML 要素を JSX 内で利用するような場合よくあるケースです。次の例を見てください。

```
<MyComponent>Hello world!</MyComponent>
```

この JSX は正しく動作します。この場合 `props.children` は `MyComponent` において、単なる文字列 `"Hello world!"` となります。HTML エスケープは元の文字列に復元されるため、多くの場合は以下のように HTML を書くように JSX を書くことができます。

```
<div>This is valid HTML &amp;amp; JSX at the same time.</div>
```

JSX は行の先頭と末尾の空白文字を削除し、また空白行も削除します。タグに隣接する改行も削除され、文字列リテラル内での改行は 1 つの空白文字に置き換えられます。そのため以下の例はすべて同じものを表示します。

```
<div>Hello World</div>

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

<div>

  Hello World
</div>
```

子要素としての JSX 要素

JSX 要素を子要素として渡すこともできます。これはネストしたコンポーネントを表示したいときに活用することができます。

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

異なる型の子要素を混在させることができるためため、文字列リテラルを JSX 要素と同時に子要素として渡すことができます。この点においても JSX と HTML は似ており、次のような例は JSX としても HTML としても正しく動作します。

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

また React コンポーネントは要素の配列を返すこともできます。

```
render() {
  // リスト化するだけのために要素を用意する必要はありません！
  return [
    // key 属性を書き忘れないようにしてください :)
```

```
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

子要素としての JavaScript 式

任意の JavaScript の式は {} で囲むことによって子要素として渡すことができます。そのため以下の JSX の式はまったく等しいものとなります。

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

これは長さの決まっていない JSX 式のリストを表示したいような場合に特に便利に使うことができます。次の例は HTML の表を表示します。

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

JavaScript の式は異なる型の子要素と併用することができるため、テンプレートリテラルの代わりに次のような書き方をすることもできます。

```
function Hello(props) {
  return <div>Hello {props.addressee}!</div>;
}
```

子要素としての関数

JSX タグに挟まれた JavaScript 式は、多くの場合は文字列や React 要素、あるいはこれらの要素のリストとして評価されます。ただし、 `props.children` はあらゆるプロパティと同様に任意のデータを渡すことができ、そのデータとは必ずしも React がレンダーできるものに限らないということに留意してください。例えば独自コンポーネントに `props.children` を通してコールバックを定義することもできるのです。

```
// numTimes の数だけ子要素のコールバックを呼び出し、コンポーネントを繰り返し作成する
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}
```

独自コンポーネントに渡される子要素は、レンダーが実行されるまでに React が理解できる要素に変換されている限りにおいては、どのようなものでも構いません。このようなやり方は一般的ではありませんが、JSX をさらに拡張したくなった時には活用してみてください。

真偽値、null、undefined は無視される

真偽値つまり `true` と `false`、`null`、そして `undefined` は子要素として渡すことができます。これらは何もレンダーしません。以下の JSX の式はすべて同じ結果となります。

```
<div />

<div></div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

<div>{true}</div>
```

これは条件に応じて React 要素を表示する際に活用できます。次の例は `showHeader` が `true` の時に限って `<Header />` が表示されます。

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

1 つ注意点として、数値 `0` のように偽と評価されるいくつかの値は、React によって表示されます。つまり次のコードは `props.messages` が空の配列の時には `0` が表示されてしまうため、期待通りには動作しません。

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

`&&` の前の式が必ず真偽値となるようにすれば、期待通りに動作します。

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

反対に、`false`、`true`、`null`、または `undefined` といった値を表示したいのであれば、まず文字列に変換する必要があります。

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

このページを編集する