リストと key

まず、JavaScript でリストを変換する方法についておさらいしましょう。
以下のコードでは、`map()` 関数を用い、`numbers` という配列を受け取って中身の値を 2 倍しています。`map()` 関数が返す新しい配列を変数 `doubled` に格納し、ログに出力します：

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

このコードはコンソールに `[2, 4, 6, 8, 10]` と出力します。
React では配列を要素のリストに変換することが、ほぼこれと同様のものです。

複数のコンポーネントをレンダリングする
要素の集合を作成し中括弧 `{}` で囲むことで JSX に含めることができます。
以下では、JavaScript の `map()` 関数を利用して、`numbers` という配列に対して反復処理を行っています。それぞれの整数に対して `<li>` 要素を返しています。最後に、結果として得られる要素の配列を `listItems` に格納しています：

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

`listItems` という配列全体を `<ul>` 要素の内側に含め、それを DOM へレンダーします：

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

Try it on CodePen
このコードは、1 から 5 までの数字の箇条書きのリストを表示します。

基本的なリストコンポーネント
通常、リストは何らかのコンポーネントの内部でレンダリングしたいと思うでしょう。
前の例をリファクタリングして、`numbers` という配列を受け取って要素のリストを出力するコンポーネントを作ることができます。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

このコードを実行すると、「リスト項目には key を与えるべきだ」という警告を受け取るでしょう。“key” とは特別な文字列の属性であり、要素のリストを作成する際に含めておく必要があるものです。
なぜ key が重要なのか、次の節で説明します。
`numbers.map()` 内のリスト項目に `key` を割り当てて、key が見つからないという問題を修正しましょう。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
```

```
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Try it on CodePen

Key

Key は、どの要素が変更、追加もしくは削除されたのかを React が識別するのに役立ちます。配列内の項目に安定した識別性を与えるため、それぞれの項目に key を与えるべきです。

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

兄弟間でその項目を一意に特定できるような文字列を key として選ぶのが最良の方法です。多くの場合、あなたのデータ内にある ID を key として使うことになるでしょう：

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

レンダリングされる要素に安定した ID がない場合、最終手段として項目のインデックスを使うことができます：

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

要素の並び順が変更される可能性がある場合、インデックスを key として使用することはお勧めしません。パフォーマンスに悪い影響を与え、コンポーネントの状態に問題を起こす可能性があります。Robin Pokorny による、key としてインデックスを用いる際の悪影響についての詳しい解説をご覧ください。より詳しく学びたい場合は、key が何故必要なのかについての詳しい解説を参照してください。

key のあるコンポーネントの抽出

key が意味を持つのは、それをとり囲んでいる配列の側の文脈です。

例えば、`ListItem` コンポーネントを抽出する際には、key は `ListItem` 自体の `<li>` 要素に書くのではなく、配列内の `<ListItem />` 要素に残しておくべきです。

例： 不適切な key の使用法

```
function ListItem(props) {
  const value = props.value;
  return (
    // Wrong! There is no need to specify the key here:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Wrong! The key should have been specified here:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

例： 正しい key の使用法

```
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()}
              value={number} />

  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Try it on CodePen

基本ルールとしては、 `map()` 呼び出しの中に現れる要素に key が必要です。

key は兄弟要素の中で一意であればよい

配列内で使われる key はその兄弟要素の中で一意である必要があります。しかし全体でユニークである必要はありません。2 つの異なる配列を作る場合は、同一の key が使われても構いません：

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
  {id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
  {id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
ReactDOM.render(
  <Blog posts={posts} />,
  document.getElementById('root')
);
```

Try it on CodePen

key は React へのヒントとして使われますが、あなたが書くコンポーネントには渡されません。同じ値をコンポーネントの中でも必要としている場合は、別の名前の prop として明示的に渡してください：

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

上記の例では、 `Post` コンポーネントは `props.id` を読み取ることができますが、 `props.key` は読み取れません。

map() を JSX に埋め込む

上記の例では `listItems` 変数を別途宣言して、それを JSX に含めました：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
              value={number} />

  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

JSX では任意の式を埋め込むことができますので、 `map()` の結果をインライン化することもできます。

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />

      )}
    </ul>
  );
}
```

Try it on CodePen

時としてこの結果はよりすっきりしたコードとなりますが、この記法は乱用されることもあります。普通の JavaScript でそうであるように、読みやすさのために変数を抽出する価値があるかどうか決めるのはあなたです。 `map()` の中身がネストされすぎている場合は、コンポーネントに抽出する良いタイミングかもしれない、ということにも留意してください。

このページを編集する