React には日本語の公式サイトがあり、チュートリアルやドキュメントが用意されています。

React 公式サイト: 日本語

関連ページ：

- React の環境構築（セットアップ）
- React 要素、React コンポーネント、インスタンス
- React Hooks の基本的な使い方

React を使うための準備

React を使うための環境を構築するにはいくつかの方法があります。

- HTML に script タグを追加して CDN を参照
- 環境構築ツール（Create React App）を利用
- 独自に環境を構築

関連ページ：React の環境構築（セットアップ）

以下は Create React App を使った環境での例になります。

Create React App

Facebook が提供している環境構築ツール Create React App を使うと簡単に React でアプリケーションを作成するための環境を構築できます。

Create React App はコマンド1つで以下のような機能を含む開発環境を構築してくれます。

- 開発サーバ
- Webpack を使用して React、JSX、ES6 を自動的にコンパイル
- CSS ファイルに自動プレフィックスを付与
- ESLint を使用してコードの誤りをテスト（検証）

関連ページ：React の環境構築（Create React App）

作業する任意のディレクトリで以下の npx コマンドを実行すると指定した名前のディレクトリが作成され、必要なファイルがインストールされます。以下は Mac のターミナルでの実行例です。

```
//プロジェクトの（作業する）ディレクトリで npx コマンドを実行
$ npx create-react-app react-sample   return

npx: 98個のパッケージを3.818秒でインストールしました。

Creating a new React app in
/Applications/MAMP/htdocs/webdesignleaves/pr/jquery/react/samples/react-sample.

Installing packages. This might take a couple of minutes.
// React 本体と ReactDOM ライブラリ、及び react-scripts などがインストールされる
Installing react, react-dom, and react-scripts with cra-template...

・・・中略・・・

//インストール成功と表示され、インストールされたパスが表示される
Success! Created react-sample at
/Applications/MAMP/htdocs/webdesignleaves/pr/jquery/react/samples/react-sample
// インストールされたプロジェクトのディレクトリで実行できるコマンドが表示される
Inside that directory, you can run several commands:

  npm start   //開発サーバを起動するコマンド
    Starts the development server.

  npm run build   //ファイルをビルドするコマンド
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:   //以下をタイプして始めましょう

  cd react-sample   //作成したプロジェクトのディレクトリに移動
  npm start  //開発サーバを起動

Happy hacking!
```

上記を実行すると、実行したディレクトリの中に react-sample という React のプロジェクト（開発環境）が作成されます。

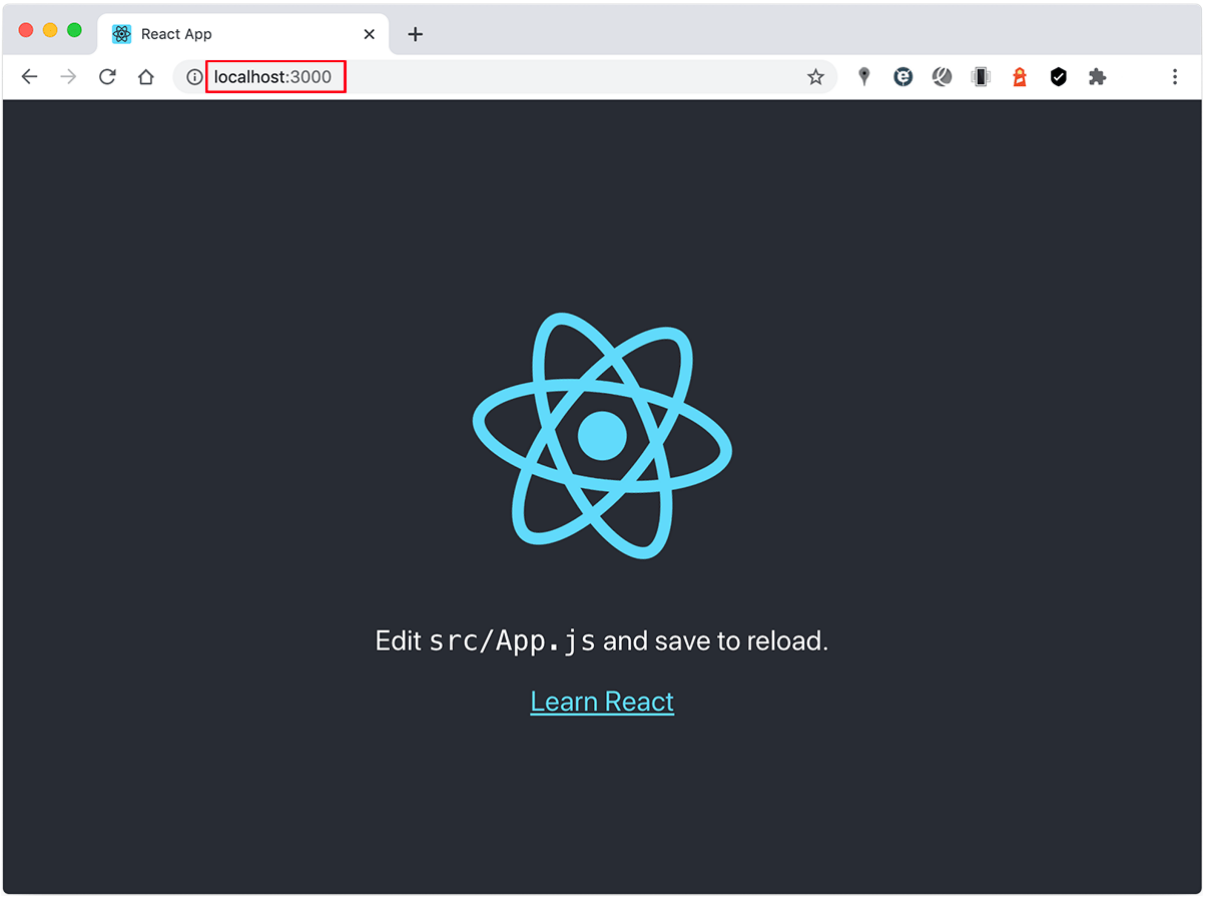
作成されたディレクトリに移動して npm start を実行して開発用サーバを起動します。

```
//プロジェクトのディレクトリ（react-sample）に移動
$ cd react-sample  return

//開発サーバ（development server）を起動
$ npm start  return
```

以下のようなメッセージの表示後、開発用サーバが起動してサンプルのアプリケーションが表示されます。

```
Compiled successfully!

You can now view react-sample in the browser.
//以下のURLでアクセスできます
  Local:            http://localhost:3000
  On Your Network:  http://192.168.11.6:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

ブラウザが自動的に起動しなければ http://localhost:3000 または 上記に表示されている IP アドレスを使ったローカルアドレス（URL）で表示することができます。:3000 はポート番号です。



開発サーバを終了するには `control + c` を押します。

生成されたプロジェクト（react-sample）は以下のような構成になっています。使用するのは開発用のファイルを配置する src フォルダと表示用のファイルが配置されている public フォルダになります。

```
react-sample   //作成されたプロジェクトのディレクトリ
├── README.md
├── node_modules  //インストールされたモジュールが入っているディレクトリ
├── package-lock.json
├── package.json   //インストールされたモジュールに関する設定ファイル
├── public  //表示用テンプレート（index.html）のディレクトリ
│   ├── favicon.ico  //React のファビコン
│   ├── index.html  //メインの HTML ファイル（テンプレート）
│   ├── logo192.png  //React のロゴ画像
│   ├── logo512.png  //React のロゴ画像
│   ├── manifest.json
│   └── robots.txt
└── src //開発用のファイルを配置するディレクトリ（サンプルが入っている）
    ├── App.css  //App コンポーネント用の CSS
    ├── App.js  //App コンポーネントのファイル
    ├── App.test.js
    ├── index.css  //HTML 全体に適用する CSS
    ├── index.js  //エントリーポイント
    ├── logo.svg
    ├── serviceWorker.js
    └── setupTests.js
```

Create React App を使って構築した環境では webpack が使われていて、 src フォルダで編集したファイルは自動的にコンパイルされ、public フォルダの index.html に出力されアプリケーションが表示されるような仕組みになっています。

public フォルダ

public フォルダには表示用ファイルのテンプレート（index.html）とテンプレートで読み込む（参照する）画像などの関連ファイルなどが入っています。

React は表示用ファイル index.html の id="root" の div 要素にコードを挿入して、ブラウザーで実行できるようにします。

初期状態では先程表示したサンプルに使われるファビコンやロゴ画像などのファイルが入っています。

以下は public フォルダにあるテンプレートの index.html です（コメント部分は省略してあります）。

%PUBLIC_URL% はビルドの際に public フォルダのパス（URL）に置き換えられ、public フォルダに配置したファイルのみが HTML から参照できます。

必要に応じて Webフォントや meta タグを追加したり、<title> や <meta name="description" /> などを変更します。ファビコン画像や Web Clip アイコン も変更することができます。

public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta name="description" content="Web site created using create-react-app"/>
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div><!-- アプリケーションが表示される div 要素  -->
  </body>
</html>
```

src フォルダ

src フォルダには開発で利用する JavaScript や CSS などのファイルや画像等を配置し、基本的にこの中で開発を行っていきます。

以下はエントリポイントのファイル index.js です。webpack はこのファイルにインポートされているモジュール（ファイル）を解析して必要なコンパイルやバンドルを実行します。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <App />  // App コンポーネント
  </React.StrictMode>,
  document.getElementById('root')
);

serviceWorker.unregister();
```

先頭で React 本体や ReactDOM、スタイルシート（index.css）、App コンポーネントを読み込んでいます。そして App コンポーネントをテンプレートの id が root の要素にレンダリングしています。

以下はサンプルの表示に使われている App コンポーネントのファイル App.js です。

src/App.js

```
import React from 'react'; //React 本体の読み込み
import logo from './logo.svg'; //ロゴ画像の読み込み
import './App.css'; //スタイルシートの読み込み

//App というコンポーネントを定義
function App() {
  return (  // JSX という構文で記述された表示（レンダリング）する内容
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

//定義した App をエクスポート
export default App;
```

先頭で React 本体の読み込みの他に、表示するロゴ画像（logo.svg）やアニメーションなどを記述したスタイルシート App.css を読み込んでいます。

JavaScript の関数を使った構文で App というコンポーネントを定義しています。コンポーネントの定義では JSX という JavaScript の拡張構文で出力内容を記述しています。

そして定義した App コンポーネントを他のファイルで使えるようにエクスポートしています。

このエクスポートされた App コンポーネントを前述の index.js でインポートして表示しています。

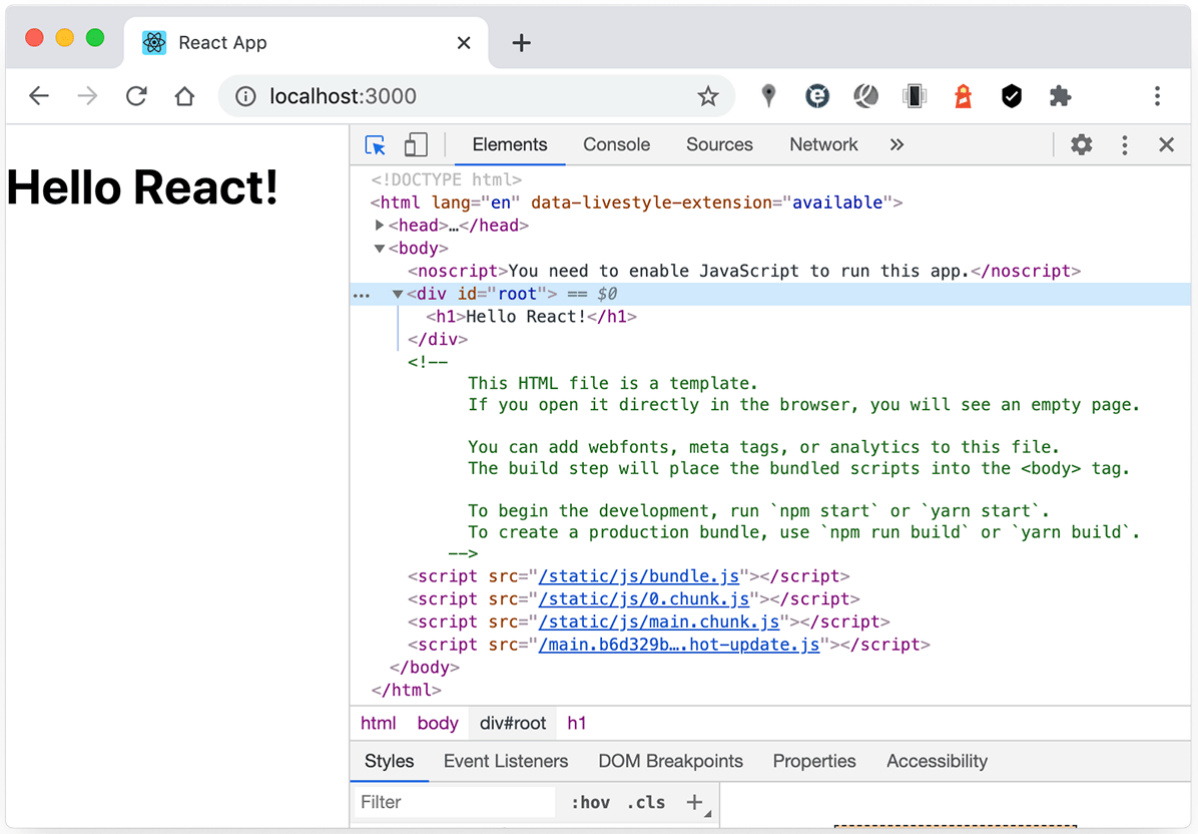
試しに上記の App.js を以下のように変更してみます。

src/App.js

```
import React from 'react';

//Hello React! と表示する App というコンポーネントを定義
function App() {
  return <h1>Hello React!</h1>
}

export default App;
```

5行目の HTML のタグのように見えるのは JSX という JavaScript 構文の拡張を使った記述です。

保存すると、開発サーバに変更が検知されて以下のように Hello React! と表示されます。



ブラウザのインスペクタで確認すると以下のようになっています。

h1 要素が id="root" の div 要素に出力され、webpack によりバンドルされた JavaScript ファイルが </body> タグの直前に挿入されています。

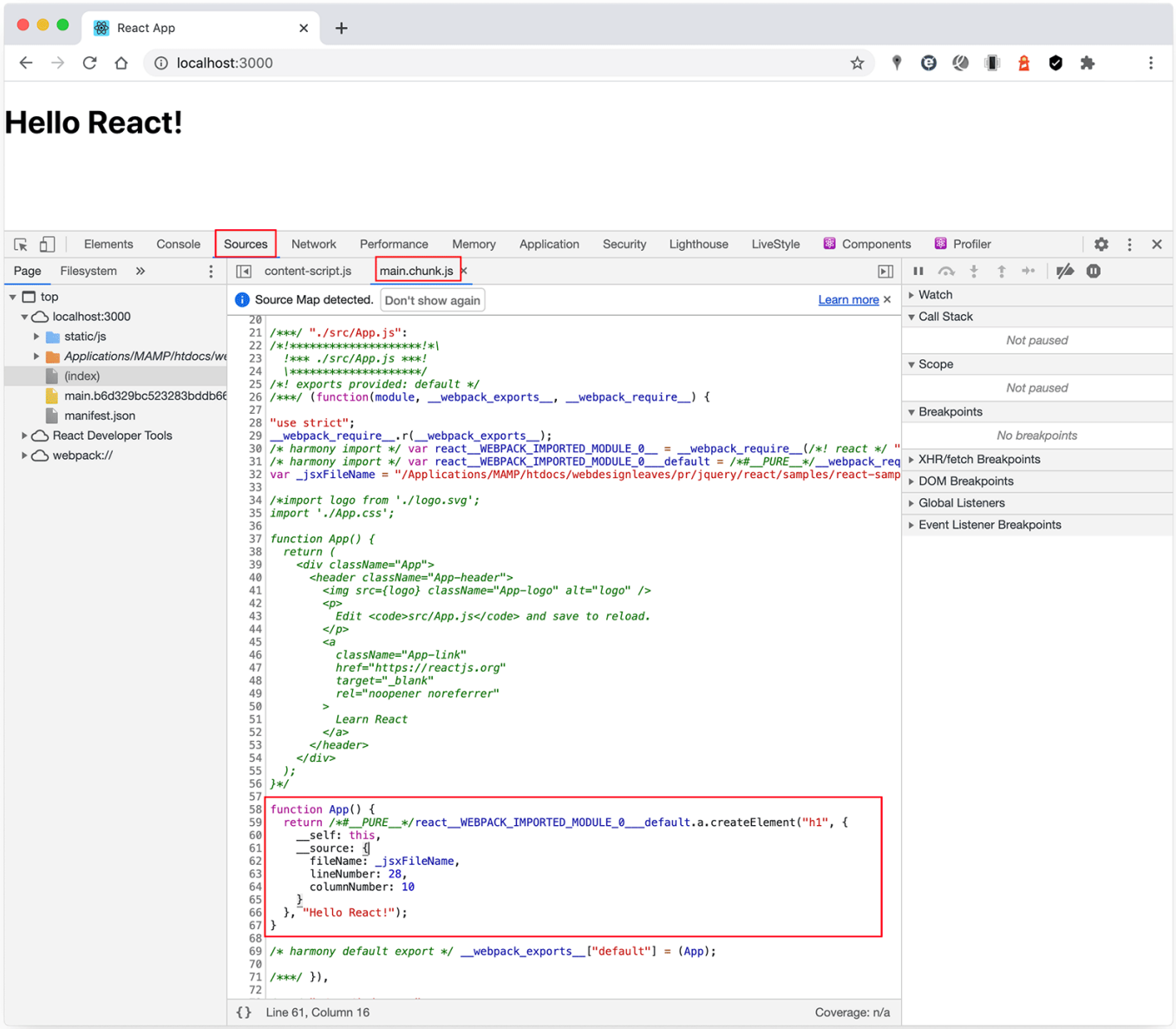
body 部分抜粋

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"><h1>Hello React!</h1></div>

  <script src="/static/js/bundle.js"></script>
  <script src="/static/js/0.chunk.js"></script>
  <script src="/static/js/main.chunk.js"></script>
  <script src="/main.b6d329bc523283bddb66.hot-update.js"></script>
</body>
```

また、webpack によりバンドルされた JavaScript の1つ main.chunk.js には App コンポーネントがビルドの際に Babel によって以下のように変換されてます。

ブラウザは JSX を理解できないので、ビルドの際に Babel によって createElement メソッドの呼び出しに変換されます（参考まで）。

```
function App() {
  return
/*#__PURE__*/react__WEBPACK_IMPORTED_MODULE_0___default.a.createElement("h1", {
    __self: this,
    __source: {
      fileName: _jsxFileName,
      lineNumber: 28,
      columnNumber: 10
    }
  }, "Hello React!");
}
```



関連項目: アプリケーションの公開（ビルド）

インポート

React で使う import について。

Create React App を含む JavaScript のバンドルツールを使った環境で React や ReactDOM のメソッドなどを使う場合、React 本体や ReactDOM を import 文を使って読み込む必要があります。

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

```
// React 本体及び Component の読み込み
import React, { Component } from 'react'
// ReactDOM の読み込み
import ReactDOM from 'react-dom'

// Component を継承して App という名前の React コンポーネントを作成
class App extends Component {
  render() {
    return <h1>Hello world!</h1>
  }
}

// ReactDOM のメソッド render() を使って App コンポーネントをレンダリング
ReactDOM.render(<App />, document.getElementById('root'))
```

例えば、上記2行目の import はデフォルトインポートと名前付きインポート（named import）です。但し、ES6 の import と少し異なる部分があります。

以下は react 本体を変数 React に読み込むデフォルトインポートです。

```
import React from 'react'
```

デフォルトインポートは、モジュールで設定されているデフォルトエクスポートに名前をつけてインポートします。デフォルトインポートの読み込みでは名前に { } は付けません。

※ デフォルトエクスポートはモジュールごとに1つしか作れません。

また、通常のデフォルトインポートは以下のようにモジュールのパスを指定します。

```
import 名前（変数名） from 'モジュールのパス';
```

但し、webpack を使用している場合、npm でインストールしたモジュールは webpack のモジュール解決の仕組みがあるので通常のローカルファイルとは異なりパスや拡張子を省略することができます。そのため import React from 'react' のようにパスや拡張子を省略して記述できます

以下は react で定義されている Component の名前付きインポート（named import）です。

```
import { Component } from 'react'
```

名前付きインポートはエクスポートされた機能の名前を指定してモジュールから選択的にインポートすることができます。import 文に続けてインポートしたい機能のリストをカンマ区切りで { } 内に指定します。

前述のデフォルトインポート同様、webpack のモジュール解決により、読み込むモジュールのパスや拡張子を省略することができます。

つまり以下は react からデフォルトインポートで react 本体を React に、名前付きインポートで react 内で定義されている Component を Component に読み込んでいます。

```
import React, { Component } from 'react';
```

react 本体は node_modules フォルダ内にあります。以下は node_modules/react/index.js です。module.exports（CommonJS）を使ってモードにより開発用または本番用のモジュールをエクスポートしているようです。

node_modules/react/index.js

```
'use strict';

if (process.env.NODE_ENV === 'production') {
  // production モードの場合にエクスポートするモジュール（ファイル）
  module.exports = require('./cjs/react.production.min.js');
} else {
  //development モードの場合にエクスポートするモジュール（ファイル）
  module.exports = require('./cjs/react.development.js');
}
```

以下は開発用の /cjs/react.development.js の最後の方の記述で、CommonJS の exports を使った Component などのエクスポートの記述があります。

node_modules/react/cjs/react.development.js 抜粋

```
exports.Children = Children;
exports.Component = Component;  //Component のエクスポート
exports.Fragment = REACT_FRAGMENT_TYPE;
exports.Profiler = REACT_PROFILER_TYPE;
exports.PureComponent = PureComponent;
exports.StrictMode = REACT_STRICT_MODE_TYPE;
exports.Suspense = REACT_SUSPENSE_TYPE;
exports.__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED =
ReactSharedInternals;
exports.cloneElement = cloneElement$1;
exports.createContext = createContext;
exports.createElement = createElement$1;
・・・
exports.useState = useState;
exports.version = ReactVersion;
```

以下は Create React App をインストールした際にサンプルで src フォルダに入っているエントリポイントのファイル index.js の例です。

3行目はモジュールのパスが ./ で始まっているので同じフォルダにあるローカルファイルですが、変数名（名前）も from もありません。このインポート構文は、webpack から提供されるもので、この CSS ファイルは webpack によりローダーで処理されてからバンドルされます。

4行目もモジュールのパスが ./ で始まっているので同じフォルダのローカルファイルです。 App.js の App コンポーネントをデフォルトインポートしています。拡張子 .js は webpack のモジュール解決（resolve.extentions）があるので省略できます。

5行目は全てのエクスポートをまとめてモジュールオブジェクトとしてインポートしています。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

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

serviceWorker.unregister();
```

App.js では App（コンポーネント）を export default App でデフォルトエクスポートしています。

src/App.js

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
//デフォルトエクスポート
export default App;
```

Hello world

典型的な React アプリではいくつかの小さなコンポーネントのファイルがあり、メインの App コンポーネントでそれらを読み込む構成になっています。

また、Create React App を使った環境ではエントリポイントの index.js で App コンポーネントなどを読み込んでレンダリングすることでアプリケーションを表示するようになっています。

そのため、最低限の構成としては src フォルダの index.js があれば、React を使ったアプリケーションを表示することができます。

以下では src フォルダの index.js のみを使って React で Hello world! と表示する例です。

index.js を以下のように書き換えると、Hello world! という見出し（h1 要素）が表示されます。

src/index.js

```
1.  // React 本体の読み込み
2.  import React from 'react';
3.  // ReactDOM ライブラリの読み込み
4.  import ReactDOM from 'react-dom';
5.
6.  //ReactDOM.render() メソッドを使ってレンダリング（表示）
7.  ReactDOM.render(
8.    <h1>Hello world!</h1>,
9.    document.getElementById('root')
10. );
```

先頭で import を使って React 本体（react）と ReactDOM（react-dom）を読み込んでいます。

2行目の import 文は React ライブラリー（react）を読み込んでいます。React ライブラリーは8行目の JSX で内部的に使われる React.createElement() で必要になります。

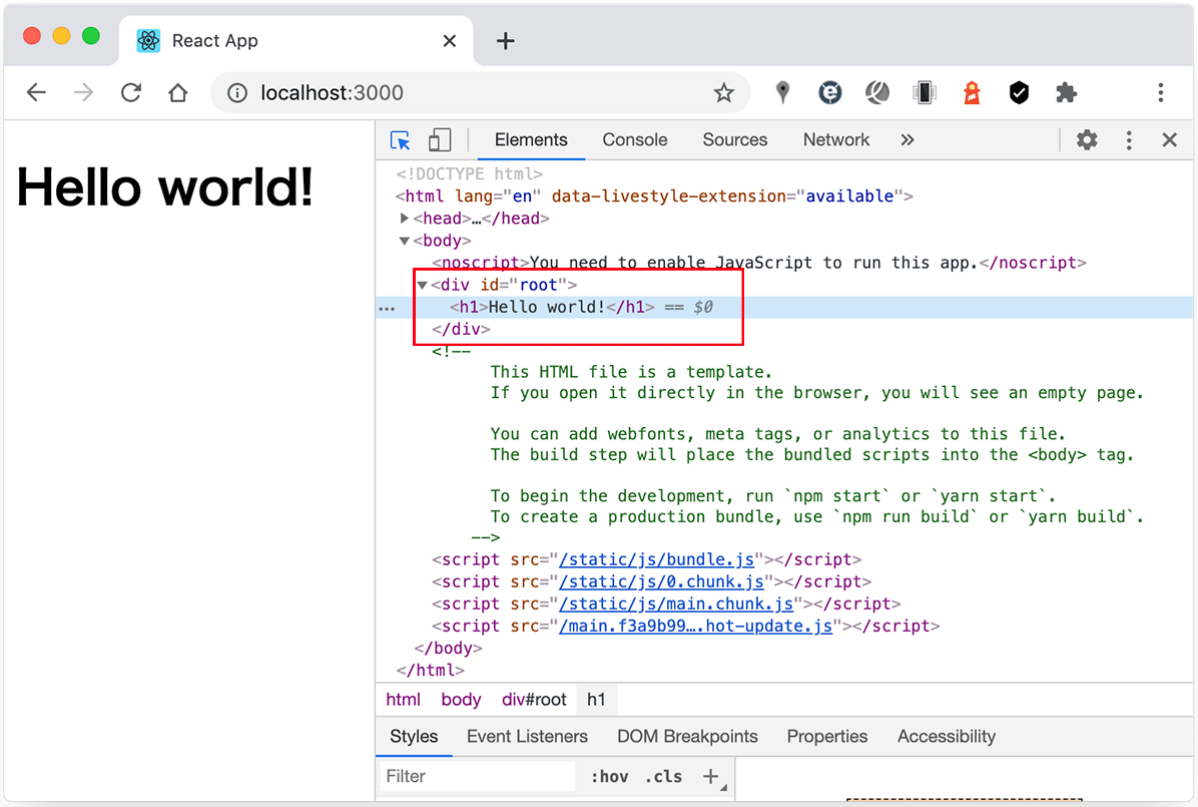
4行目の import 文は React で DOM を扱うための ReactDOM ライブラリー（react-dom）を読み込んでいます。ReactDOM.render() メソッドを使うのに必要になります。

7～10行目は ReactDOM の render() メソッドです。ReactDOM.render() メソッドには表示する React 要素（8行目）と表示するコンテナ（9行目）を渡します。

8行目は HTML 要素のように見えますが、JSX の構文で、React 要素と呼ばれるオブジェクトです。

9行目は表示先のコンテナを document.getElementById() で指定しています。

保存すると以下のように Hello world! と表示されます。



JSX

JSX は Facebook 社が考案した XML に似た JavaScript 構文の拡張です。

JSX は HTML や XML のタグのように要素を入れ子にしたり、属性を指定することができますが、実際は JavaScript なので JavaScript（JSX）特有の制限や記述方法（文法）があります。

以下で仕様（Draft）が公開されています。

Draft: JSX Specification / XML-LIKE SYNTAX EXTENSION TO ECMASCRIPT

以下は React 公式サイトの JSX に関するリンクです。

- React：JSX の導入
- React：JSX を深く理解する

JSX では HTML や XML のタグのように見える JSX 独自の構文を使って画面に描画したい内容を記述します（最終的に DOM にレンダリングするものを定義します）。

JSX により生成される要素（React 要素）は HTML のような見た目ですが、実際には DOM ノードではなく React が管理するオブジェクトで、React によりレンダリングされます。

以下は JSX を使わず、React の関数 createElement() を使って「Hello world!」と表示する例です。

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

//createElement() で生成した React 要素を変数に代入
const element = React.createElement('h1', null, 'Hello world!');

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

以下は JSX を使って「Hello world!」と表示する例です。こちらの方が直感的でわかりやすく記述できます。特に要素を入れ子にして記述する場合では断然こちらのほうがわかりやすくなります。

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

//JSX で生成した React 要素を変数に代入
const element = <h1>Hello world!</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

以下の右辺は HTML のように見えますが、JSX を使った記述で JavaScript の式です。JavaScript の式なので変数に代入することができます。

以下は「Hello world!」と h1 タグで表示する React 要素を JSX で生成して変数に代入しています。

文字列でも HTML でもないので引用符で囲みません（引用符で囲むと単なる文字列として扱われます）。

```
//JSX で生成した React 要素を変数 element に代入
const element = <h1>Hello world!</h1>;
```

JSX を複数行に分けて記述する場合は、自動的にセミコロンが挿入されないように括弧 ( ) で囲みます。

```
//複数行に分けて記述 → 括弧で囲む
const element = (
  <h1>
    Hello world!
  </h1>
)
```

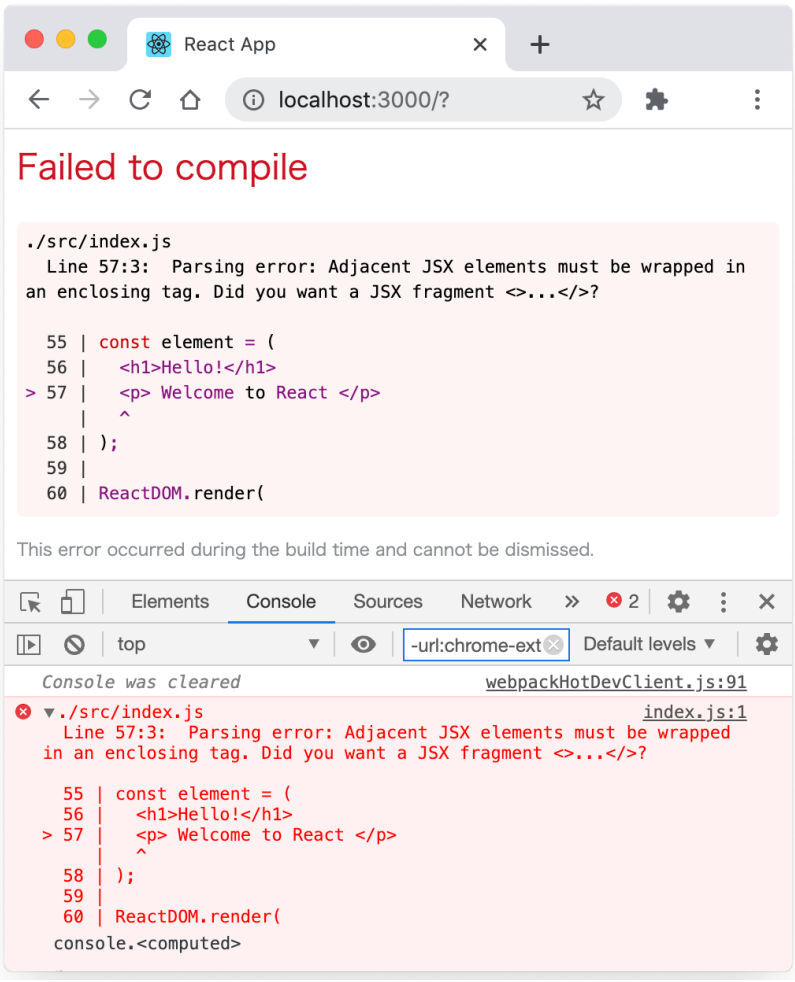
JSX も HTML のように子要素を持つことができます。

```
//子要素を持てる
const element = (
  <header>
    <h1>Hello world!</h1>
  </header>
)
```

但し、以下のように React 要素やコンポーネントを並べて記述するとエラーになります。

```
1.  import React from 'react';
2.  import ReactDOM from 'react-dom';
3.
4.  // エラーになる例
5.  const element = (
6.    <h1>Hello!</h1>
7.    <p> Welcome to React </p>
8.  );
9.
10. ReactDOM.render(
11.   element,
12.   document.getElementById('root')
13. );
```

Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag.（隣接するJSX要素は、囲みタグで囲む必要があります）



隣接する JSX 要素は div 要素などの囲みタグ（親要素）で囲んで1まとまりとする必要があります。

このような場合は、以下のように div 要素などで囲みます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//複数の要素がある場合は、親要素（以下の場合は div 要素）でラップします
const element = (
  <div>
    <h1>Hello!</h1>
    <p> Welcome to React. </p>
  </div>
);

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

フラグメント を使うこともできます。

React.createElement()

JSX は HTML タグのように見えますが React 要素（オブジェクト）を生成する JavaScript の式です。

内部ではオブジェクトを受け取り、React のメソッド createElement() を実行しています。

以下のコードは全て同じ React 要素を生成して変数 element に代入しています。

```
//JSX を使った記述
const element = <h1>Hello world!</h1>;

//JSX を使わず React.createElement() を使った記述
const element = React.createElement('h1', null, 'Hello world!')

//上記は 以下のように複数行に分けて記述することもできます
const element = React.createElement(
  'h1',
  null,
  'Hello world!'
)
```

React：React.createElement()

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、 以下のオンライン Babel コンパイラで試すことができます。

Babel：オンライン コンパイラ

以下は React.createElement() の書式です。

```
React.createElement(
  type,  //タグ名（文字列）や React component 型、React fragment 型
  [props],  //プロパティ
  [...children]  //子要素
)
```

React：JSX なしで React を使う

React.createElement() は以下のようなオブジェクトを生成します。

```
//React.createElement() により生成されるオブジェクト
const element = {
  type: 'h1',
  props: {
    children: 'Hello world!'  //子要素（この場合は文字列）
  }
};
//但し、上記は実際のオブジェクトと異なり構造は単純化されています
```

ビルドの際に Babel（コンパイラ）は JSX を React.createElement() メソッドを使った普通の Javascript に変換します。つまり、JSX は React.createElement() のシンタックスシュガーにすぎません。

JSX を使うと React.createElement() を直接呼ばすに HTML タグに似たような形式で記述できるので、見た目がわかりやすくなります。

React で JSX は必須ではありませんが、JSX を使った方が簡潔に記述することができます。

JSX はインジェクション攻撃を防ぐ（？）

「JSX の導入」には以下のように記述されていますが、全ての XSS を防げるわけではありません。

> JSX にユーザの入力を埋め込むことは安全です。デフォルトでは、React DOM は JSX に埋め込まれた値をレンダリングされる前にエスケープします。このため、自分のアプリケーションで明示的に書かれたものではないあらゆるコードは、注入できないことが保証されます。レンダーの前に全てが文字列に変換されます。これは XSS (cross-site-scripting) 攻撃の防止に役立ちます。

例えば以下のような a.href 属性を介した XSS は可能です。表示されるリンクをクリックすると alert() が実行されてしまいます。

```
const userURL = "javascript:alert('ハックされました！');";

class UserDangerousPage extends React.Component {
  render() {
    return (
      <a href={userURL}>My Website</a>
    )
  }
}

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

また、あまり使うことはないと思いますが、dangerouslySetInnerHTML には XSS の危険があります（そのためこのような名前が付いています）。以下はページを開くと alert() が実行されてしまいます。

```
const badText = "<img onerror='alert(\"ハックされました！\");' src='invalid-image' />";

class MyDangerousComponent extends React.Component {
  render() {
    return (
      <div dangerouslySetInnerHTML={{"__html": badText}} />
    );
  }
}

ReactDOM.render(
  <MyDangerousComponent />,
  document.getElementById("root")
)
```

このため入力された値などを使う場合は、入力された値を検証（チェック）する必要があります。

参考：なぜReact Elementは$$typeofプロパティを持っているの？

JavaScript の式

任意の JavaScript の式を JSX 内で中括弧 { } で囲んで使用することができます。

JSX の中括弧 { } の中で使用できるのは式（Expression：評価した結果を変数に代入できるもの）で、if や const foo = 7 や while などの文（Statement）は使えません。

以下は name という変数を宣言して、それを中括弧で囲んで JSX 内で使用する例です。

```
import React from 'react'
import ReactDOM from 'react-dom'

//name という変数を宣言（JSX の外）
const name = 'Foo';

//name を中括弧で囲んで JSX 内で使用
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
//Hello, Foo と表示される
```

以下は、addExclamation() という JavaScript 関数を中括弧で囲んでその結果を JSX 内に挿入する例です。

```
import React from 'react'
import ReactDOM from 'react-dom'

const name = 'Foo';

//受け取った文字の最後に ！ を追加する関数（JSX の外）
function addExclamation(string) {
  return string + '!'
}

//関数 addExclamation を中括弧で囲んで JSX 内で結果を展開
const element = <h1>Hello, {addExclamation(name)}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
//Hello, Foo！ と表示される
```

以下は map() を使って li 要素を返す式を中括弧で囲んで JSX 内に挿入する例です。（関連：リストと key ）

```
function MyList({ dataList }) {
  return (
    <ul>
      {dataList.map((item,index) =>
        <li key={index}>{item}</li>
      )}
    </ul>
  );
}
```

コンパイル後、JSX の式は普通の JavaScript の関数呼び出しに変換され、JavaScript オブジェクトとして評価されます。

このため、JSX を if 文や for ループの中で使用したり、変数に代入したり、引数として受け取ったり、関数から返したりすることができます。

以下は getGreeting() という JSX を返す関数を使う例です。

```
import React from 'react'
import ReactDOM from 'react-dom'

const name = 'Foo';

function addExclamation(string) {
  return string + '!'
}

//JSX を返す関数
function getGreeting(name) {
  if (name) {
    return <h1>Hello, {addExclamation(name)}</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}

ReactDOM.render(
  getGreeting(name),
  document.getElementById('root')
);
//Hello, Foo! と表示される
```

JSX で属性を指定

属性として文字列リテラルを指定するには引用符を使用できます。

```
const element = <div tabIndex="0"></div>;
```

属性として JavaScript 式を埋め込むには中括弧を使用できます。

```
const user = {
  imgUrl : '/images/user01.jpg' // public/images/ にある画像のパスの例
};

const element = <img src={user.imgUrl}></img>
```

JSX はキャメルケース

React DOM は HTML の属性ではなくキャメルケース (camelCase) のプロパティ命名規則を使用します。

tabindex は tabIndex（I が大文字）、onclick は onClick になります。

また、class は JavaScript では予約されているため、className を使用します。

```
const myClass = 'bar'

const element = <h1 className={myClass}>Hello world!</h1>
```

以下はこのファイルと同じ src フォルダ内の画像を読み込んで、クラス属性（className）や alt 属性（alt）を指定して表示する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';
import logo from './logo.svg';  //src フォルダにある画像をインポート

const element = <img src = {logo} className = "App-logo" alt = "logo" />;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

for も JavaScript では予約されているため、for 属性を指定する場合は htmlFor を使用します。

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

※ アクセシビリティのために用いられる aria-* 属性は、特別で、キャメルケースには変換せず、そのまま HTML と同じようにハイフンケースである必要があります。

```
<button onClick={this.onClickHandler}
    aria-haspopup="true"
    aria-expanded={this.state.isOpen}>
  Select an option
</button>
```

空要素

空要素や子要素を持たない場合には、開始タグにスラッシュを入れることで終了タグを省略できます。

```
const element = <img src={user.imgUrl} />;

//以下と同じこと
const element = <img src={user.imgUrl}></img>
```

子要素

JSX のタグは子要素を持つことができます。

```
const header = (
  <header>
    <h1>Hello world!</h1>
  </header>
);
```

上記を createElement() を使って記述すると以下のように入れ子になっています。

```
const header = React.createElement(
  "header",
  null,
  React.createElement(
    "h1",
    null,
    "Hello world!"
  )
);
```

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（子要素）は props.children という特別なプロパティとして渡されます。

開始タグと終了タグの間に文字列のみを含んでいる場合、その文字列が props.children となります。

以下の場合 props.children は単なる文字列 "Hello world!" となります。

```
const element = <h1>Hello world!</h1>;
```

インラインスタイル

JSX で style 属性を使ってインラインスタイルを設定することができます。style 属性の設定は JavaScript のオブジェクト形式 { } で記述します。

JSX で JavaScript 式を埋め込むには中括弧 { } で囲むので、 style 属性でスタイルを指定する場合はさらに波括弧で囲みます。

以下は p 要素の文字色とフォントサイズ、背景色を指定する例です。

オブジェクト形式で指定するので fontSize のようにキャメルケースで指定する必要があります。区切りはセミコロンではなく、カンマになります。また、サイズを数値で指定した場合は px が適用されます。

```
<p style={ {color:'#0A1F95', fontSize:20, backgroundColor:'red' } }>my style</p>
```

スタイルはオブジェクトなので変数に入れて使うこともできます。

```
//スタイルを変数に代入
const myStyle = {color:'#fff', fontSize:20, backgroundColor:'red' };

const element = <p style={myStyle}>my style</p>;;
```

クラスでスタイルを設定

スタイルシートを読み込んでクラスやセレクタでスタイルを設定することもできます。

例えば、以下のような CSS ファイルを src フォルダに配置します。

MyStyle.css

```
.head_title {
  text-align: center;
  color: #0A7E10;
}

h1 {
  font-size: 60px;
}
```

スタイルシートの読み込みは import を使って読み込むことができます。

Create React App を使っている場合は、以下のように import で CSS ファイルを指定すれば、webpack により適切な位置に CSS が読み込まれます。

また、JSX でクラスを指定する場合は、JavaScript では class は予約語なので使えないので className を使います。

```
import React from 'react';
import ReactDOM from 'react-dom';
//スタイルシートの読み込み
import './MyStyle.css';

//class 属性は className で指定
const element = <h1 className="head_title">Hello React.</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

React 要素を DOM にレンダリング

以下は再び、「Hello world!」と表示するだけの index.js です。

index.js

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
3.
4. const element = <h1>Hello world!</h1>;
5.
6. //React 要素を DOM にレンダリング
7. ReactDOM.render(
8.   element,
9.   document.getElementById('root')
10. );
```

4行目では JSX で記述した React 要素を変数 element に代入しています。

React 要素は画面に表示したい内容を記述したもので、実際の DOM ノード（要素）ではなく React が管理するオブジェクト（Virtual DOM）です。

7～10行目では ReactDOM.render() メソッドを使って React 要素を DOM にレンダリング（描画）しています。

ReactDOM.render()

React 要素やコンポーネントを DOM にレンダリングするには ReactDOM.render() メソッドを使います。

以下は ReactDOM.render() メソッドの書式です。

```
1. ReactDOM.render(element, container[, callback])
```

- element：React 要素やコンポーネント
- container：描画先となる実際の DOM ノード（コンテナ）
- callback：オプションのコールバック関数

Create React App で構築した環境を使用している場合、第2引数の container は、public フォルダの index.html の id 属性が root の div 要素になります。

この id="root" の div 要素の中にあるものは全て React DOM によって管理されます。そのため、この要素を「ルート DOM ノード」と呼ぶこともあります。

public/index.html（一部省略）

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    ・・・中略・・・
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div><!-- 描画先となる実際の DOM ノード -->
  </body>
</html>
```

以下は React 要素 element を（※ public/index.html の）id="root" の div 要素（コンテナ）にレンダリングしています。

表示先のコンテナ（id="root" の div 要素）は document.getElementById() で指定しています。

index.js

```
//React 要素
const element = <h1>Hello world!</h1>;

//React 要素を id="root" の div 要素にレンダリング
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

※ Create React App で構築した環境を使用している場合、webpack の設定で public/index.html の </body> タグの直前に JavaScript が挿入されるようになっています。

React：ReactDOM.render

レンダリングされた要素の更新

React 要素は一度要素を作成するとその子要素もしくは属性を変更することはできません。

そのため UI を更新する1つの方法としては、新しい要素を作成して ReactDOM.render() に渡します。

以下は setInterval() を使って ReactDOM.render() を毎秒呼び出して UI を更新する例です。

※ 通常はこのように何度も ReactDOM.render() を呼び出す使い方はしません。実際には大抵の React アプリケーションは ReactDOM.render() を一度だけ呼び出し、状態（state ）を更新することで再レンダリングします。

src/index.js

```
1.  import React from 'react';
2.  import ReactDOM from 'react-dom';
3.
4.  //React 要素を作成して ReactDOM.render() を呼び出す関数
5.  function tick() {
6.    //現在時刻を表示する React 要素を変数 element に代入
7.    const element = (
8.      <div>
9.        <h1>現在時刻： {new Date().toLocaleTimeString()} </h1>
10.     </div>
11.   );
12.   //React 要素を（element）をレンダリング
13.   ReactDOM.render(element, document.getElementById('root'));
14. }
15.
16. //1000ミリ秒ごと（毎秒）関数 tick を呼び出して実行
17. setInterval(tick, 1000);
```

ブラウザのインスペクタで確認すると以下のように表示されます。

1秒毎に UI ツリー全体を表す要素（div 要素、h1 要素、テキストノード）を作成してますが、内容が変更される時刻部分のテキストノードのみが React DOM により更新されます。

関連項目：state とライフサイクルメソッドを使った例

React は必要な箇所のみを更新

React 要素は実際の DOM ノードではなく React が管理するオブジェクト（Virtual DOM）です。

実際の DOM ノードへのレンダリングは React DOM が React 要素とその子要素を以前のものと比較し必要なだけの DOM の更新を行います。

Virtual DOM（仮想 DOM）

Virtual DOM では、実際の DOM と対になるオブジェクト（仮想 DOM）を用意し、そのオブジェクトに対して処理を行い、その結果生まれた差分だけを実際の DOM に反映します。

React では直接 DOM を操作しない

基本的には React でコードを記述する際は、直接 DOM を操作しません（React 要素は実際の DOM ノードではありません）。JSX で記述している操作は React 要素を操作しているのであって、DOM を操作しているわけではありません。

React が管理する props や state などのデータを更新することで、React がデータの更新を検知して DOM がどのように変わるかの差分を計算し、React が DOM の操作（更新）を行いレンダリングします。

React：要素のレンダー

React：仮想 DOM と内部処理

コンポーネント

React でのコンポーネントとは UI を再利用できる部品に分割して定義したもので、部品に分割して定義することで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。

コンポーネントは簡単に言うと Web ページなどの UI を構成する部品（コードをまとめる単位）で、これらを組み合わせて UI を構築します。

それぞれのコンポーネントは React に何を描画したいかを伝えます（React 要素を返します）。React 要素はコンポーネントを構成する最小単位の構成ブロックです。

React では Javascript の関数または ES6 のクラスとしてコンポーネントを定義することができます。

以下は関数を使ったコンポーネントの書式です。

```
//関数コンポーネントの定義
function コンポーネント名(props) {
  // 必要な処理（もしあれば）
  return レンダリングする内容（通常 JSX で記述された React 要素）;
}
```

関数コンポーネントはレンダリングする内容を記述した React 要素を返す（return する）関数です。また、props という引数（オブジェクト）を使ってデータを受け渡すことができます。

以下は関数を使って Welcome という名前のコンポーネントを定義する例です。この関数は h1 要素で Hello, xxxx と表示するコンポーネントを定義しています。

```
//関数コンポーネントの定義
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

```
//関数コンポーネントの定義（アロー関数を使った場合。上記と同じこと）
const  Welcome = (props) => {
  return <h1>Hello, {props.name} !</h1>;
}
```

コンポーネント名は常に大文字で始める

React は小文字で始まるコンポーネントを のような組み込みのコンポーネント（DOM タグ）として扱います。ユーザ定義のコンポーネントは <Welcome /> のように最初の文字を大文字で指定して、組み込みコンポーネントの DOM タグと区別する必要があります。

今までの例では DOM のタグを表す React 要素のみを扱っていましたが、React 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。

以下は DOM タグ（組み込みのコンポーネント）を表す React 要素の例です。

```
//組み込みのコンポーネントを表す React 要素を JSX で生成して変数に代入
const element = <h1>Hello world!</h1>;
```

以下はユーザ定義のコンポーネントを表す React 要素の例です。

```
//ユーザ定義のコンポーネントを表す React 要素を JSX で生成して変数に代入
const element = <Welcome name="Foo" />;
```

React：JSX を深く理解する

コンポーネントのレンダー

以下は独自のコンポーネント Welcome を定義して表示する例です。

定義したコンポーネントを JSX で記述して React 要素を生成し、ReactDOM.render() でレンダリングしています。

以下のコードはページに Hello, Foo! と表示します。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

//コンポーネント Welcome の定義
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}

//ユーザ定義のコンポーネントを表す React 要素を生成して変数に代入
const element = <Welcome name="Foo" />;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

以下のように、生成した React 要素を変数には入れず、直接 ReactDOM.render() に指定しても同じです。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

//コンポーネント Welcome の定義
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}

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

React はユーザ定義のコンポーネントを見つけると、JSX に書かれている属性と子要素を props としてそのコンポーネントに渡します（props の詳細は次項）。

上記の場合以下のような処理が行われます。

ReactDOM.render() が呼び出される際に引数に <Welcome name="Foo" /> が渡されます。

React は渡されたコンポーネントが大文字から始まるので独自のコンポーネントと判定して <Welcome name="Foo" /> に設定されている属性（name="Foo"）を引数 props として Welcome コンポーネント（の定義の関数）を呼び出します。

Welcome コンポーネント（の関数）は受け取った props.name が Foo なので、出力として React 要素 <h1>Hello, Foo!</h1> を return します。

React DOM は <h1>Hello, Foo!</h1> に一致するように DOM を更新します（レンダリングします）。

コンポーネントの再レンダリング

コンポーネントは ReactDOM.render() が呼び出される際に最初のレンダリングが行われます。そして、アプリケーション内で以下のような変化が検知されるとそのタイミングで再レンダリングされます。

- 親コンポーネントが再レンダリングされた時
- 親コンポーネントから渡されている props が変化した時
- state が変化した時

props

props はコンポーネントに渡される任意のデータ（JavaScript オブジェクト）です。props を使ってデータをコンポーネントに渡すことができます。

```
1. //コンポーネントの定義
2. function MyComponent(props) {
3.   // 定義では {props.プロパティ名} で設定されたプロパティを参照
4. }
5.
6. // コンポーネントを生成する際に「プロパティ名="値"」で props を設定する場合の例
7. const element = <MyComponent プロパティ名="値" />;
```

属性として渡す（「プロパティ名="値"」で props を設定）

独自に定義したコンポーネントを利用する際に HTML に属性を記述するのと同じ記述の仕方で props を設定してデータを渡すことができます。HTML の場合は「属性名=値」ですが、JSX では「プロパティ名=値」となります。プロパティ名には任意の文字が使えます。

以下の場合、<Welcome name="Foo" /> の name="Foo" の記述によりコンポーネント定義の props.name（2行目） に Foo が渡されます。

そしてレンダリングされる際には Hello, Foo! として表示されます。

```
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
  // {props.name} で name="Foo" を参照
}

const element = <Welcome name="Foo" />;
// プロパティ名="値" で props.プロパティ名 に値を設定

ReactDOM.render(
  element,
  document.getElementById('root')
);
//<h1>Hello, Foo!</h1> とレンダリング
```

以下は myProps1 と myProps2 というプロパティを MyProps コンポーネントに設定する例です。

```
//属性を記述するのと同じ方法で「プロパティ名=値」として props を設定
const element = <MyProps myProps1="Foo" myProps2="Bar"/>;
```

コンポーネントを定義するコード内では、props.プロパティ名 として参照できます。

以下はプロパティ（props）をコンソールに出力して、props.myProps1 及び props.myProps2 でプロパティの値を出力するコンポーネントの例です。

src/index.js

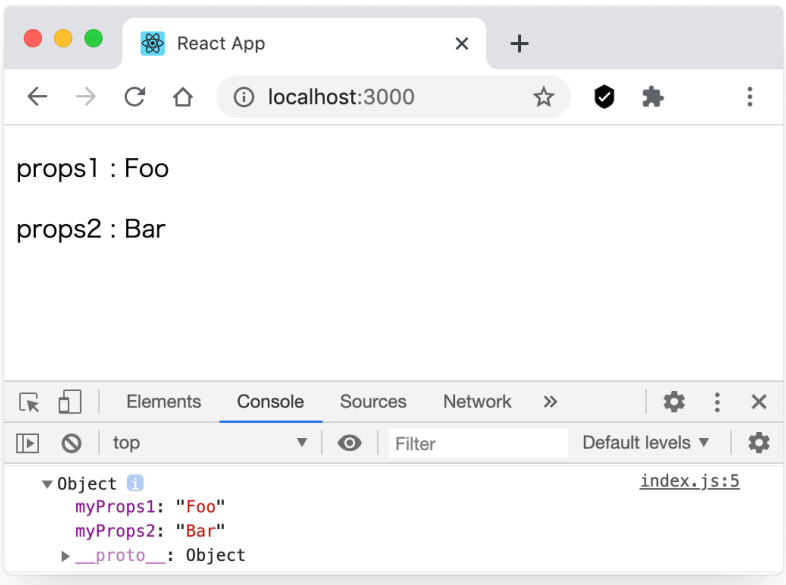
```
import React from 'react';
import ReactDOM from 'react-dom';

//関数コンポーネントの定義
function MyProps(props) {
  //ブラウザのコンソールに props を出力（確認用）
  console.log(props);
  //props.プロパティ名 で参照
  return (
    <div>
      <p>props1 : {props.myProps1} </p>
      <p>props2 : {props.myProps2} </p>
    </div>
  )
};

//ユーザ定義の MyProps コンポーネントを表す React 要素
const element = <MyProps myProps1="Foo" myProps2="Bar"/>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

設定する props をオブジェクトとして別途定義しておいてスプレッド構文（...）を使ってまとめてコンポーネントに渡すこともできます。

以下は上記をスプレッド構文を使って書き換えたもので、結果は同じです。

```
function MyProps(props) {
  console.log(props);
  return (
    <div>
      <p>props1 : {props.myProps1} </p>
      <p>props2 : {props.myProps2} </p>
    </div>
  )
};

//props を別途定義
const myProps = {
  myProps1 : "Foo" ,
  myProps2 : "Bar"
}

//スプレッド構文 {...myProps} でまとめて props を渡す。
//<MyProps myProps1="Foo" myProps2="Bar"/>と同じこと
ReactDOM.render(
  <MyProps {...myProps}/>,
  document.getElementById('root')
);
```

上記は以下のように表示されます。



props として JSX を渡すこともできます。その場合は中括弧 { } で囲む必要があります。

```
function MyProps(props) {

  //props.プロパティ名 で参照
  return (
    <div>
      <p>props1 : {props.myProps1} </p>
      <p>props2 : {props.myProps2} </p>
    </div>
  )
};

//props として JSX を渡す
const element = <MyProps myProps1={<strong>Foo</strong>} myProps2="Bar"/>;
```

以下は h2 要素で表示する文字と追加するクラスを propsを使って渡す例です。

クラスを使ってスタイルを指定するので以下の CSS を読み込みます（クラスでスタイルを設定 ）。

MyStyle.css

```
.head_title {
  color: #0A7E10;
}
.title {
  font-size: 30px;
}
.bg {
  display: inline-block;
  padding: 5px 15px;
  background-color: #EFF59E;
}
```

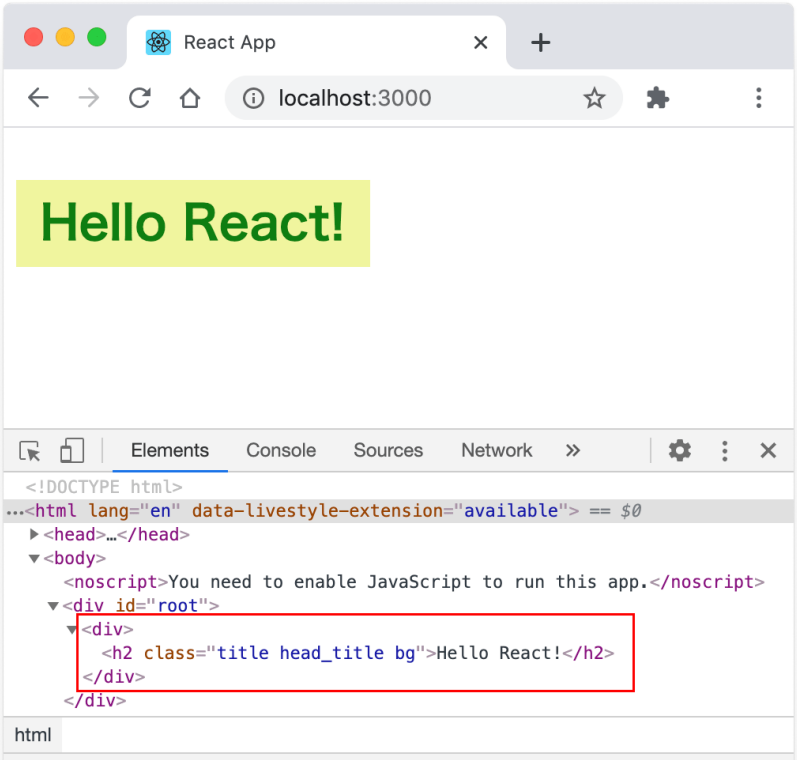
className にはテンプレートリテラルを使ってデフォルトのクラス title に ${props.addClass} で指定したクラスを追加するようにしています。

テンプレートリテラルでは式や変数を ${ } で囲んで展開することができます。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './MyStyle.css';


//関数コンポーネントの定義
function MyPropsStyle(props) {

  //props.プロパティ名 で参照
  return (
    <div>
      <h2 className={`title ${props.addClass}`}>{props.message}</h2>
    </div>
  )
};

//ユーザ定義の MyPropsStyle コンポーネントを表す React 要素
const element = <MyPropsStyle message="Hello React!" addClass="head_title bg" />;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

上記は以下のように表示されます。



props の値により表示を変える

以下は受け取った props の値により3項演算子を使って表示する文字を変える例です。

以下の場合は props.isYes は true が渡されるので Yes と出力されます。false が渡されると No と出力されます。

JSX の中で中括弧 { } で囲んで JavaScript を使うことができます。 if 文は使えませんが、3項演算子は式なので使用できます（最後にセミコロを付けると文となりエラーになります）。

真偽値を props に渡す場合、真偽値は JavaScript なので中括弧 { } で囲む必要があります。文字列を渡していしまうと、この場合 true と判定されます。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyProps(props) {
  //3項演算子を使って props の値により表示を変える
  return (
    <p>{ props.isYes ? "Yes" : "No" }</p>
  )
};

//props に真偽値を渡す（中括弧で囲む）
const element = <MyProps isYes ={true} />;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

分割代入を使う

ES6 の分割代入を使うと簡単に必要な props を変数に代入することができます。

今までの例では渡された props を参照するには、以下のように props.プロパティ名 で参照していました。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyHello(props) {
  //渡された props を props.name と props.message で参照
  return (
    <div>
      <h3>Hello, {props.name}</h3>
      <p>{props.message}</p>
    </div>
  )
};

const element = <MyHello name ="Foo" message="Welcome to my spaceship!"/>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

上記は以下のような HTML がレンダリングされます。

```
<div>
  <h3>Hello, Foo</h3>
  <p>Welcome to my spaceship!</p>
</div>
```

分割代入を使うと以下のように記述することができます。分割代入を使えば、毎回 props.xxxx と記述する必要がなくなります。

```
function MyHello(props) {

  //変数 name には props.name が、message には props.message が代入される
  const {name, message} = props; //分割代入

  //props.name や props.message ではなく name や message で参照できる
  return (
    <div>
      <h3>Hello, {name}</h3>
      <p>{message}</p>
    </div>
  )
};

const element = <MyHello name ="Foo" message="Welcome to my spaceship!"/>;
```

以下のように、仮引数の props の代わりに分割代入 {name, message} を使って記述することができます。

こちらの方がより簡潔に記述することができ、よく使われる書き方です。

```
//仮引数に分割代入を使う
function MyHello({name, message}) {

  //props.name や props.message ではなく name や message で参照できる
  return (
    <div>
      <h3>Hello, {name}</h3>
      <p>{message}</p>
    </div>
  )
};

const element = <MyHello name ="Foo" message="Welcome to my spaceship!"/>;
```

また、分割代入時に初期値を設定することもでき、props が未定義の場合にその設定した値が渡されます。

```
//仮引数に分割代入を使う際に初期値を設定する
function MyHello({name, message="Welcome!"}) {

  return (
    <div>
      <h3>Hello, {name}</h3>
      <p>{message}</p>
    </div>
  )
};

// message が設定されていない（未定義の）場合、初期値 "Welcome!" が渡される
ReactDOM.render(<MyHello name ="Foo"/>, document.getElementById("root"));
```

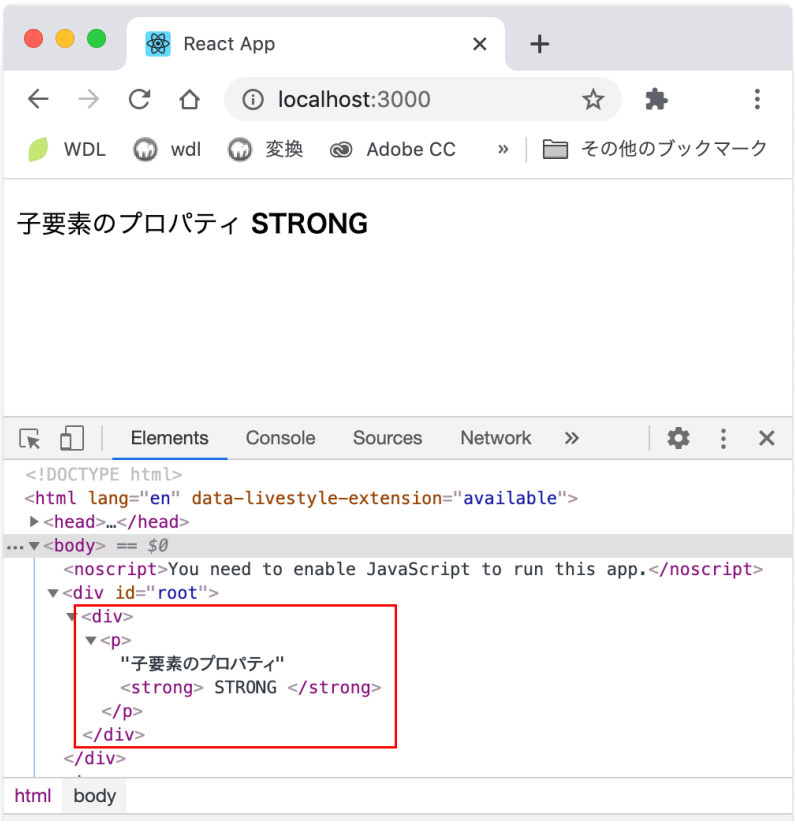
子要素として渡す

props をコンポーネントに渡すには、属性として設定する以外に子要素としてコンポーネントタグに値を挟んで渡すこともできます。

子要素として渡された値は、props.children として参照できます。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

function MyPropsChild(props) {
  //props.children でコンポーネントタグに挟まれた値を参照
  return (
    <div>
      <p>{props.children}</p>
    </div>
  )
};

const element = (
  //コンポーネントタグに値を挟んで渡す
  <MyPropsChild>
    子要素のプロパティ<strong> STRONG </strong>
  </MyPropsChild>
 );

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

上記は以下のように表示されます。



以下の MyButton1 と MyButton2 のコンポーネントは同じ見た目のボタンをレンダリングします。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyButton1(props) {
  return  <button>{ props.children }</button>;
};

function MyButton2(props) {
  return  <button children={ props.children } />;
  // return  <button children={ props.children }></button>; と同じ
};

ReactDOM.render(
  (
    <div>
      <MyButton1>Click</MyButton1>
      <MyButton2>Click</MyButton2>
    </div>
  ),
  document.getElementById('root')
);
```

上記は以下のようにレンダリングされます。

```
<div>
  <button>Click</button>
  <button>Click</button>
</div>
```

それぞれ以下のようにコンパイル（JavaScript へ変換）されます。

```
function MyButton1(props) {
  return  <button>{ props.children }</button>;
};

//上記を JavaScript にコンパイル
React.createElement("button", null, props.children);
```

```
function MyButton2(props) {
  return  <button children={ props.children } />;
};

//上記を JavaScript にコンパイル
React.createElement("button", {children: props.children});
```

```
//以下が React.createElement の書式
React.createElement( type, [props], [...children] )
```

関連項目：コンポジション | JSX における子要素

デフォルト値（defaultProps）

コンポーネントにはデフォルトの props を設定することができます。

以下は Welcome クラスに defaultProps を追加することで、name プロパティをオプションにする例です。name を指定しない場合はデフォルトの値 world が設定されます。

```
import React from 'react';
import ReactDOM from 'react-dom';

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

//デフォルトの props.name を設定
Welcome.defaultProps = {
  name: "world",
};

//name を指定していないので Hello, world! と表示される
const element = <Welcome />;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

React：コンポーネントと props

React：React.Component

React：defaultProps

Props は読み取り専用

コンポーネントは、自分自身の props を変更してはいけません。親コンポーネントから渡された props は、子コンポーネント側で更新することはできません。

同じ入力が与えられた場合、常に同じ出力をレンダリングするようにします。

React：Props は読み取り専用

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することができます。

また、データを Props 経由で渡すことができます。

以下は Welcome と Statement というコンポーネントを作成して、App というコンポーネントから参照して表示する例です。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

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

function Statement(props) {
  return <div>{props.children}</div>
}

function App() {
  return (
    <div>
      <Welcome name="React" />
      <Statement>This is My First React App.</Statement>
    </div>
  );
}

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

データを Props 経由で渡す

この例では App コンポーネントから Welcome コンポーネントに props.name で「React」を、Statement コンポーネントに props.children で子要素である文字列「This is My First React App.」を渡しています。

以下はアロー関数を使って記述した例です（上記と同じことです）。

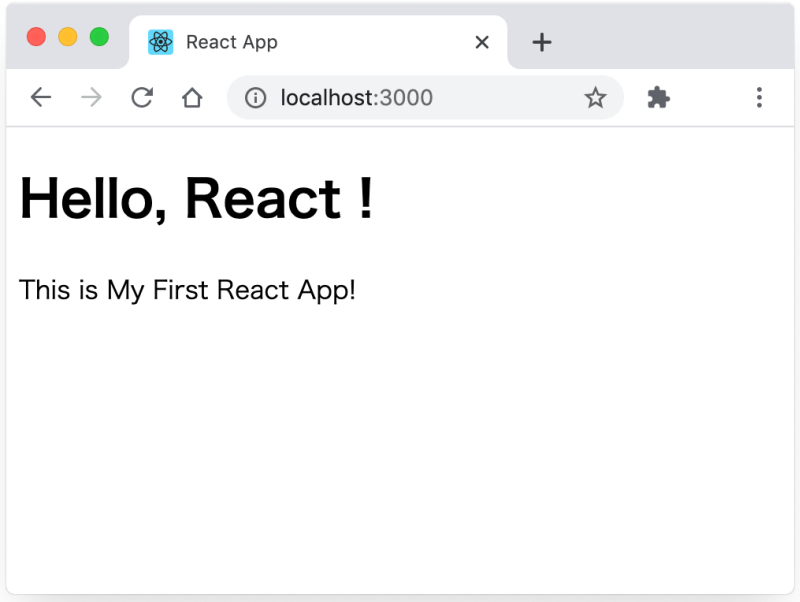
src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

const Welcome = (props) => {
  return <h1>Hello, {props.name} !</h1>;
}

const Statement = (props) => {
  return <div>{props.children}</div>;
}

const App = () => {
  return (
    <div>
      <Welcome name="React" />
      <Statement>This is My First React App.</Statement>
    </div>
  );
}

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

以下のように return を省略して記述することもできます。

```
const Welcome = (props) => (
  <h1>Hello, {props.name} !</h1>
);

const Statement = (props) => (
  <div>{props.children}</div>
);

const App = () => (
  <div>
    <Welcome name="React" />
    <Statement>This is My First React App.</Statement>
  </div>
);

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

上記は以下のように表示されます。



典型的な React アプリでは小さなコンポーネントのファイルがそれぞれあり、メインの App コンポーネントでそれらを読み込み、エントリポイントで App コンポーネントをレンダリングするようになっていることが多いようです。

以下はコンポーネントごとにファイルを分割する例です。

```
.
├── public
│   └── index.html
└── src
    ├── App.js
    ├── Statement.js
    ├── Welcome.js
    └── index.js
```

Welcome コンポーネントのファイル Welcome.js を新たに src フォルダに作成し、コンポーネントを定義してデフォルトエクスポートします。

src/Welcome.js

```
import React from 'react';

const Welcome = (props) => {
  return <h1>Hello, {props.name} !</h1>;
}

export default Welcome;
```

以下は宣言と同時にデフォルトエクスポートする例です。宣言と同時にデフォルトエクスポートする場合、関数やクラスの名前を省略することができます（デフォルトエクスポート）。

src/Welcome.js

```
import React from 'react';

export default (props) => {
  return <h1>Hello, {props.name} !</h1>;
}
```

同様に Statement コンポーネントのファイル Statement.js を新たに src フォルダに作成し、コンポーネントを関数で定義してデフォルトエクスポートします。

src/Statement.js

```
import React from 'react';

const Statement = (props) => {
  return <div>{props.children}</div>
}

export default Statement;
```

App.js を以下のように書き換えます。Welcome と Statement コンポーネントを読み込んで App コンポーネントを関数で定義してデフォルトエクスポートします。

src/App.js

```
import React from 'react';
import Welcome from './Welcome';
import Statement from './Statement';

const App = () => {
  return (
    <div>
      <Welcome name="React" />
      <Statement>This is My First React App!</Statement>
    </div>
  );
}

export default App;
```

src/index.js を以下のように書き換えて、Appコンポーネントを読み込んでレンダリングします。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

上記のように書き換えて保存すると、先程と同じ表示になります。

以下は表（table）を表示するコンポーネントの例です。メインの App コンポーネントで表のコンポーネント Table を読み込んでエントリポイント（index.js）でレンダリングしています。

```
.
└── src
    ├── App.js
    ├── Table.js
    └── index.js
```

src/Table.js

```
import React from 'react'

const Table = (props) => {
  return (
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>E-mail</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Foo</td>
          <td>foo@example.com</td>
        </tr>
        <tr>
          <td>Bar</td>
          <td>bar@example.com</td>
        </tr>
      </tbody>
    </table>
  )
}

export default Table
```

src/App.js （メインのコンポーネント）

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

const App = () =>  {
  return (
    <div className="table_container">
      <Table />
    </div>
  )
}

export default App;
```

src/index.js （エントリポイント）

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

以下のような HTML を出力します。

```
<div id="root">
  <div class="table_container">
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>E-mail</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Foo</td>
          <td>foo@example.com</td>
        </tr>
        <tr>
          <td>Bar</td>
          <td>bar@example.com</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

上記の Table コンポーネントは表の値（データ）や構造が直に記述されているため再利用するのは難しいので、コンポーネントに分割して再利用しやすくします。

まず、Table コンポーネントを書き換えて、以下のように thead 部分の TableHeader と tbody 部分の TableBody の2つのコンポーネントに分割することができます。

そして Table コンポーネントで分割したコンポーネントを読み込めば前と同じ表が表示されます。

src/Table.js

```
import React from 'react'

//thead 部分のコンポーネントに分割
const TableHeader = () => {
  return (
    <thead>
      <tr>
        <th>Name</th>
        <th>E-mail</th>
      </tr>
    </thead>
  )
}

//tbody 部分のコンポーネントに分割
const TableBody = () => {
  return (
    <tbody>
      <tr>
        <td>Foo</td>
        <td>foo@example.com</td>
      </tr>
      <tr>
        <td>Bar</td>
        <td>bar@example.com</td>
      </tr>
    </tbody>
  )
}

//分割した TableHeader と TableBody を読み込む
const Table = (props) => {
  return (
    <table>
      <TableHeader />
      <TableBody />
    </table>
  )
}

export default Table
```

続いて、tbody 部分のデータを props を使って以下のようなオブジェクトの配列から読み込むようにします。

```
const members = [
  {
    name: 'Foo',
    email: 'foo@example.com',
  },
  {
    name: 'Bar',
    email: 'bar@example.com',
  },
]
```

TableBody コンポーネントを以下のように書き換えて、props 経由で上記のデータを props.memberData で受け取り map() を使ってデータを tr 要素と td 要素からなる行の要素に変換します。

tr 要素には map() のコールバックに渡される index を key 属性を指定しています（リストと key ）。

```
const TableBody = (props) => {
  // map() を使って memberData 配列の要素から行（tr と td）に変換して新たな配列 rows に格納
  const rows = props.memberData.map((row, index) => {
    return (
      <tr key={index}>
        <td>{row.name}</td>
        <td>{row.email}</td>
      </tr>
    )
  })
  //上記で生成した行（tr と td）の配列を返す
  return <tbody>{rows}</tbody>
}
```

props を TableBody コンポーネントに渡すように Table コンポーネントを以下のように書き換えます。

props は Table コンポーネント経由で TableBody コンポーネントに渡されます。

```
const Table = (props) => {
  return (
    <table>
      <TableHeader />
      <TableBody memberData={props.memberData}/>
    </table>
  )
}
```

ES6 の分割代入を使って以下のように記述することもできます。

```
const Table = (props) => {
  //分割代入を使って変数 memberData に代入
  const {memberData} = props;
  return (
    <table>
      <TableHeader />
      <TableBody memberData={memberData}/>
    </table>
  )
}
```

App コンポーネントを以下のように書き換えてデータを読み込みます。memberData={members} で props.memberData にデータ（members）が渡され、Table コンポーネント経由で TableBody コンポーネントに渡されます。

src/App.js

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

const App = () =>  {
  //表示に使用するデータ（19行目で指定して props へ渡す）
  const members = [
    {
      name: 'Foo',
      email: 'foo@example.com',
    },
    {
      name: 'Bar',
      email: 'bar@example.com',
    },
  ]

  return (
    <div className="table_container">
      <Table  memberData={members}/>
    </div>
  )
}

export default App;
```

以下が変更後の Table.js です。

src/Table.js

```
import React from 'react'

const TableHeader = () => {
  return (
    <thead>
      <tr>
        <th>Name</th>
        <th>E-mail</th>
      </tr>
    </thead>
  )
}

const TableBody = (props) => {
  // map() を使って memberData 配列の要素から行（tr と td）に変換して新たな配列 rows に格納
  const rows = props.memberData.map((row, index) => {
    return (
      <tr key={index}>
        <td>{row.name}</td>
        <td>{row.email}</td>
      </tr>
    )
  })
  //上記で生成した行（tr と td）の配列を返す
  return <tbody>{rows}</tbody>
}

const Table = (props) => {
  //分割代入を使って変数 memberData に代入
  const {memberData} = props;
  return (
    <table>
      <TableHeader />
      <TableBody memberData={memberData}/>
    </table>
  )
}

export default Table
```

クラスコンポーネント

以下は関数コンポーネントの定義の例です。

```
//関数コンポーネントの定義
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

以下は上記と同等のコンポーネントを ES6 のクラス構文を使って定義する例です。

```
1. //クラスコンポーネントの定義（React.Component を継承）
2. class Welcome extends React.Component {
3.   // render() メソッド を定義
4.   render() {
5.     // レンダリングする内容を return （props にアクセスするには this.props とします）
6.     return <h1>Hello, {this.props.name}</h1>;
7.   }
8. }
```

クラスコンポーネントを定義するには React.Component を継承（extends）します。

また、少なくとも render() メソッドを定義します。render() は React.Component サブクラスで必ず定義しなければならないクラスのメソッドで、ライフサイクルメソッド の1つです。

render() は最初の描画時と props や state が更新する度に呼び出されます。

render() の定義では、レンダリングする内容を記述して返します。render() 内で return されたコードが実際にレンダリングされます。

また、render() の定義では必要に応じて props や state を使って処理を記述することができます。

クラスコンポーネントの定義の中で props にアクセスするには this.props とする必要があります。

```
1. class コンポーネント名 extends React.Component {
2.   //必要に応じてコンストラクタやクラスメソッド（ハンドラ）などを定義
3.
4.   render() { //少なくとも render() メソッド を定義（必須）
5.     /*必要に応じて（state が更新された場合など）何らかの処理を実行
6.     レンダリングする内容（React 要素）を return */
7.   }
8. }
```

state を使う場合やイベント処理のメソッドのバインドが必要な場合は、コンストラクタ を定義します。

```
class コンポーネント名 extends React.Component {

  //コンストラクタの定義
  constructor(props) {
    super(props);
    // state プロパティの初期化など
    this.state = {
       ・・・
    };
    //メソッドのバインド（必要に応じて）
    this.myUpdate = this.myUpdate.bind(this);
  }

  // render() メソッド を定義（必須）
  render() {
    // レンダリングする内容を記述して return
    return JSX で記述した React 要素
  }
}
```

React：React.Component

React：React.Component render()

Hooks

React バージョン16.8 未満では state やライフサイクル などの機能はクラスコンポーネントでしか使えませんでしたが、React バージョン16.8 から導入された Hooks を使用することにより、 state やライフサイクルなどの機能を関数コンポーネントでも使えるようになっています。

現在は Hooks を用いることで、関数コンポーネントでもクラスコンポーネントとほぼ同等の機能を実現することができるようになっています。

関連ページ：React Hooks の基本的な使い方

state

state はコンポーネントの状態を表すプロパティ（JavaScript オブジェクト）で、コンポーネント自体によって作成、更新、保持され、ユーザ操作や時間経過などで動的に変化するデータを扱うための機能です。

state はそのコンポーネント内で使用できるプロパティで、コンポーネント内で setState() を呼び出すことで更新することができます。他のコンポーネントの state を直接参照・更新することはできません（props 経由で渡すことができます）。

setState() が呼び出されると state の値が更新され、コンポーネントの render() メソッドが実行される仕組みになっています。

どのようなデータが state になりうるのかを考えるには以下が目安になります（UI 状態を表現する必要かつ十分な state を決定する）。

- 親から props を通じて与えられたデータであれば、それは state ではありません
- 時間経過で変化しないままでいるデータであれば、それは state ではありません
- コンポーネント内にある他の props や state を使って算出可能なデータであれば、それは state ではありません

コンストラクタ

デフォルトではコンポーネントは state プロパティを持っていないので、state を使うにはコンストラクタで state を初期化します。

コンストラクタの例

```
constructor(props) {
  //他の文の前に super(props) を呼び出す（コンストラクタのオーバライド）
  super(props);
  // state プロパティの初期化（初期値の設定）
  this.state = {
    count: 0,
  };
  //クラスのメソッドはデフォルトではバインドされないのでバインドする
  this.updateCount = this.updateCount.bind(this);
}
```

state を初期化するには、constructor() を利用し、コンストラクタで直接 this.state に初期状態を割り当てます。コンストラクタは、this.state を直接代入する唯一の場所です。

※ constructor() の中で setState() を呼び出してはいけません。また、コンストラクタ以外で this.state に値を代入してはいけません。state を更新するには setState() を使います。

constructor(props) の中では super(props); を呼ぶ必要があります。 そうしないと、this.props はコンストラクタ内で未定義になります（サブクラスで this を参照するには親クラスのコンストラクタ処理を実行してからでないと this を参照できないため）。

コンストラクタ内で props を使わない場合は、以下のように constructor() と super() の引数を省略することができますが、常に super(props) を使うのが良い方法です。（参考：なぜsuper(props) を書くの?）

```
constructor() {
 super();
 this.state = {
    ....
  };
}
```

また、state に props をコピーしてはいけません。

```
constructor(props) {
 super(props);
 /* 以下は NG （state に props をコピーしてはいけません）*/
  this.state = { color: props.color };  //してはいけない
}
```

state の初期化もメソッドのバインドもしない場合はコンストラクタを実装する必要はありません。コンストラクタを省略した場合は親クラス（React.Component）のコンストラクタが適用されます。

React：constructor()

以下はクリックするとその回数を表示するボタンのクラスコンポーネントの例です。

state はコンポーネント内のコンストラクタで初期化されます（10～12行目）。

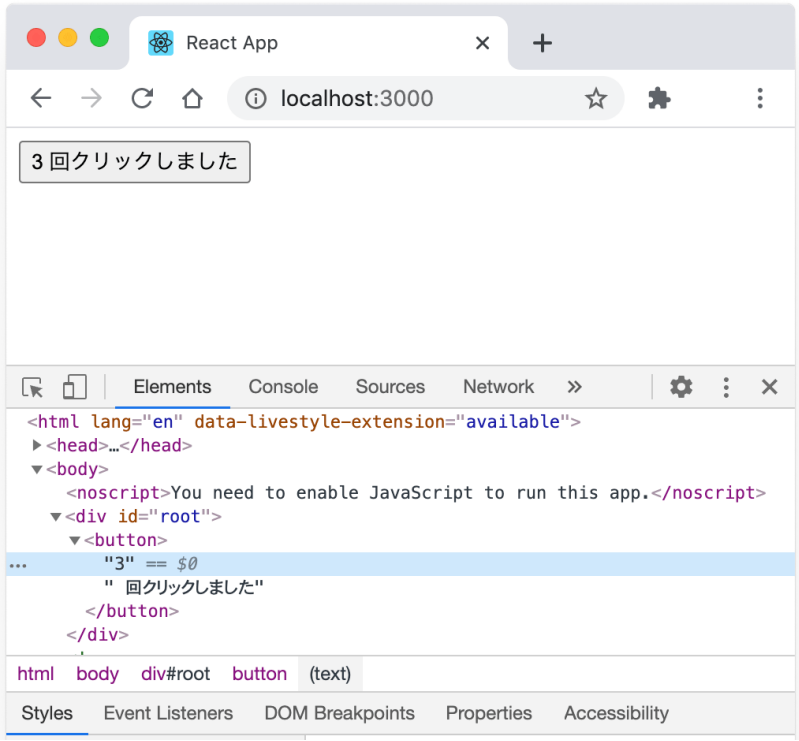
state は setState() を呼び出すことで更新され、state が更新されると自動的にコンポーネントの render() メソッドが実行されます。

以下の場合、onClick を使ったイベント処理で updateCount() が呼び出され、updateCount() の this.setState(updater 関数) で state が更新されます。

state はクラスのプロパティなので、クラス内のどのメソッドからでも参照することができます。以下では render() メソッドの中で state.count（クリックした回数）を参照しています。

src/index.js

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

class CountButton extends React.Component {
  //コンストラクタ
  constructor() {
    //他の文の前に super(props) を呼び出す
    super();
    // state プロパティの初期化（初期値の設定）
    this.state = {
      count: 0,
    };
    //クラスのメソッドはデフォルトではバインドされないのでバインドしておく
    this.updateCount = this.updateCount.bind(this);
  }

  //カウントを更新するクラスのメソッド
  updateCount() {
    this.setState((prevState) => {
      return { count: prevState.count + 1 }
    });
  }

  render() {
    //onClick を使って updateCount() を呼び出しイベント処理
    return (
      <button onClick={this.updateCount}>
        {this.state.count} 回クリックしました
      </button>
     );
  }
}

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



参考サイト：ReactJS: Props vs. State

setState()

state を更新したい場合には、setState() メソッドを使います。

setState() はコンポーネントから呼び出すことができるメソッドで、setState() が呼び出されると値が更新され、コンポーネントの render() メソッドが実行されてコンポーネントが再レンダーされます。

state を直接変更しない

this.state に直接値を代入してもコンポーネントは再レンダーされません。

例えば、以下のコードではコンポーネントは再レンダーされません。this.state に直接代入できる唯一の場所はコンストラクタ内だけです。

```
// 間違い（やってはいけない）※直接代入できるのはコンストラクタ内だけ
this.state.comment = 'Hello';
```

代わりに setState() を使用します。

```
// 正しい方法
this.setState({comment: 'Hello'});
```

state の更新は非同期に行われる可能性がある

React はパフォーマンスのために、複数の setState() 呼び出しを一度の更新にまとめて処理することがあります。そのため、現在の state の値に依存する書き方をする場合は、this.state の値に依存するのではなく setState() の1番目の引数に関数を指定します。

```
// 間違い（期待通りにならない）
this.setState({
  //this.state.counter は最新の値ではない可能性がある
  counter: this.state.counter + 1
});
```

1番目の引数にオブジェクトではなく関数を受け取る書式で、関数が受け取る引数（現在の state への参照）を使って処理をして更新します。

```
// 正しい方法
this.setState((state) => ({
  //関数が受け取る引数 state は最新であることが保証されている
  counter: state.counter + 1
}));
```

setState() は常にコンポーネントを直ちに更新するわけではない

以下は、React.Component/setState()からの抜粋です。

> setState() は常にコンポーネントを直ちに更新するわけではありません。それはバッチ式に更新するか後で更新を延期するかもしれません。これは setState() を呼び出した直後に this.state を読み取ることが潜在的な危険になります。代わりに、componentDidUpdate または setState コールバック（setState(updater, callback)）を使用してください。

setState() で値を更新した直後に this.state を使って処理をする場合はライフサイクルメソッドの componentDidUpdate または setState のコールバック を使います。

以下が setState() の書式です。

```
setState( 関数またはオブジェクト [, callback] )
```

1番目の引数には関数またはオブジェクトを指定することができます。2番目の引数 callback は、setState が完了してコンポーネントが再レンダリングされると実行される省略可能なコールバック関数です。

以下の stateChange は {key:value} のようなオブジェクトで、value には更新した値を指定します。

```
//関数を指定する場合（stateChange は state オブジェクト）
setState((state, props) => (stateChange) [, callback])

//オブジェクトを指定する場合（stateChange は state オブジェクト）
setState( stateChange [, callback] )
```

以下は最初の引数に関数を指定する場合の記述例です。関数はアロー関数や function で記述できます。

引数 state と props はコンポーネントの state と props への参照す。stateChange は更新したオブジェクトです。props は使用しなければ省略可能です。

```
//最初の引数をアロー関数で記述
setState((state, props) => (stateChange));

//アロー関数で return を省略しない場合
setState((state, props) =>
  return stateChange
);

//最初の引数を function 文で記述
setState(function(state, props) {
  return stateChange
});
```

updater 関数が受け取る引数 state と props の両方は最新のものであることが保証されています。

例えば、state.count の値を1増加したい場合は以下のように state への参照を使って新しいオブジェクトを返します。以下の場合 state への参照を表す引数を prevState としているので prevState.count + 1 となります。

```
this.setState((prevState, props) => {
  return { count: prevState.count + 1 }
});

// return を省略する場合
this.setState((prevState, props) => ({
  count: prevState.count + 1
}));

// function 文を使う場合
this.setState(function(prevState, props) {
  return {
    count: prevState.count + 1
  };
});
```

上記のように props を使用しない場合は省略することができます（省略します）。

```
this.setState((prevState) => {
  return { count: prevState.count + 1 }
});
```

以下は最初の引数にオブジェクトを指定して、state.quantity を2に更新する例です。

```
this.setState({quantity: 2})
```

以下はコールバックを使って state.operator を更新した後に handleKeyUp というクラスメソッドを呼び出す例です。

```
this.setState({
  operator:  event.target.value
}, () => {
  //コールバック
  this.handleKeyUp();
});
```

前述 のクリックするとその回数を表示するボタンの例で、カウントを更新するメソッドを以下のように変更すると、1回クリックすると3ずつ増加します。

```
updateCount() {
  this.setState((prevState) => {
    return { count: prevState.count + 1 }
  });
  this.setState((prevState) => {
    return { count: prevState.count + 1 }
  });
  this.setState((prevState) => {
    return { count: prevState.count + 1 }
  });
}
```

但し、以下のようにオブジェクトで指定すると1回クリックするごとに1しか増加しません。

後続の呼び出しは、同じサイクル内の前の呼び出しの値を上書きするため、1 回だけ増分されます。

```
updateCount() {
  this.setState({ count: this.state.count + 1 });
  this.setState({ count: this.state.count + 1 });
  this.setState({ count: this.state.count + 1 });
}
```

また、this.state と this.props は非同期に更新されるため、最新であることは保証されていません。

React：setState()

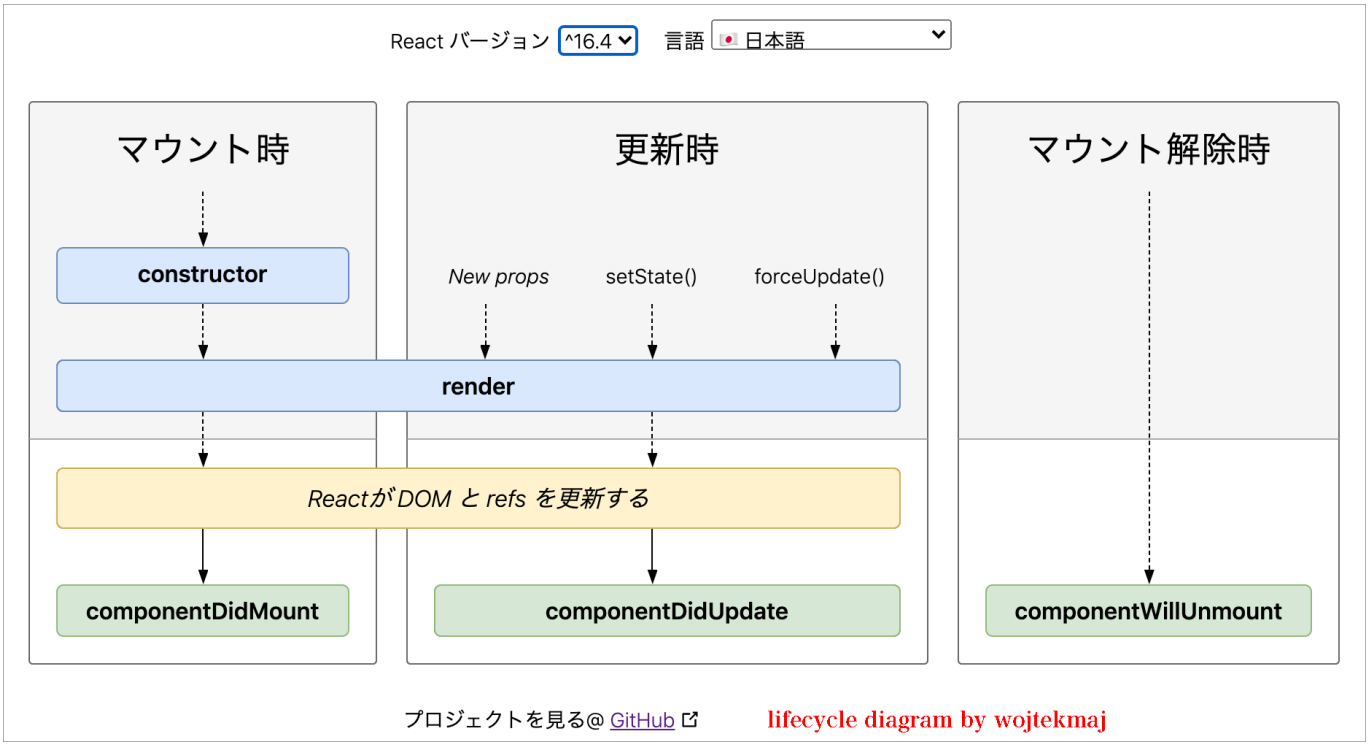
React：state とライフサイクル

React：UI 状態を表現する必要かつ十分な state を決定する

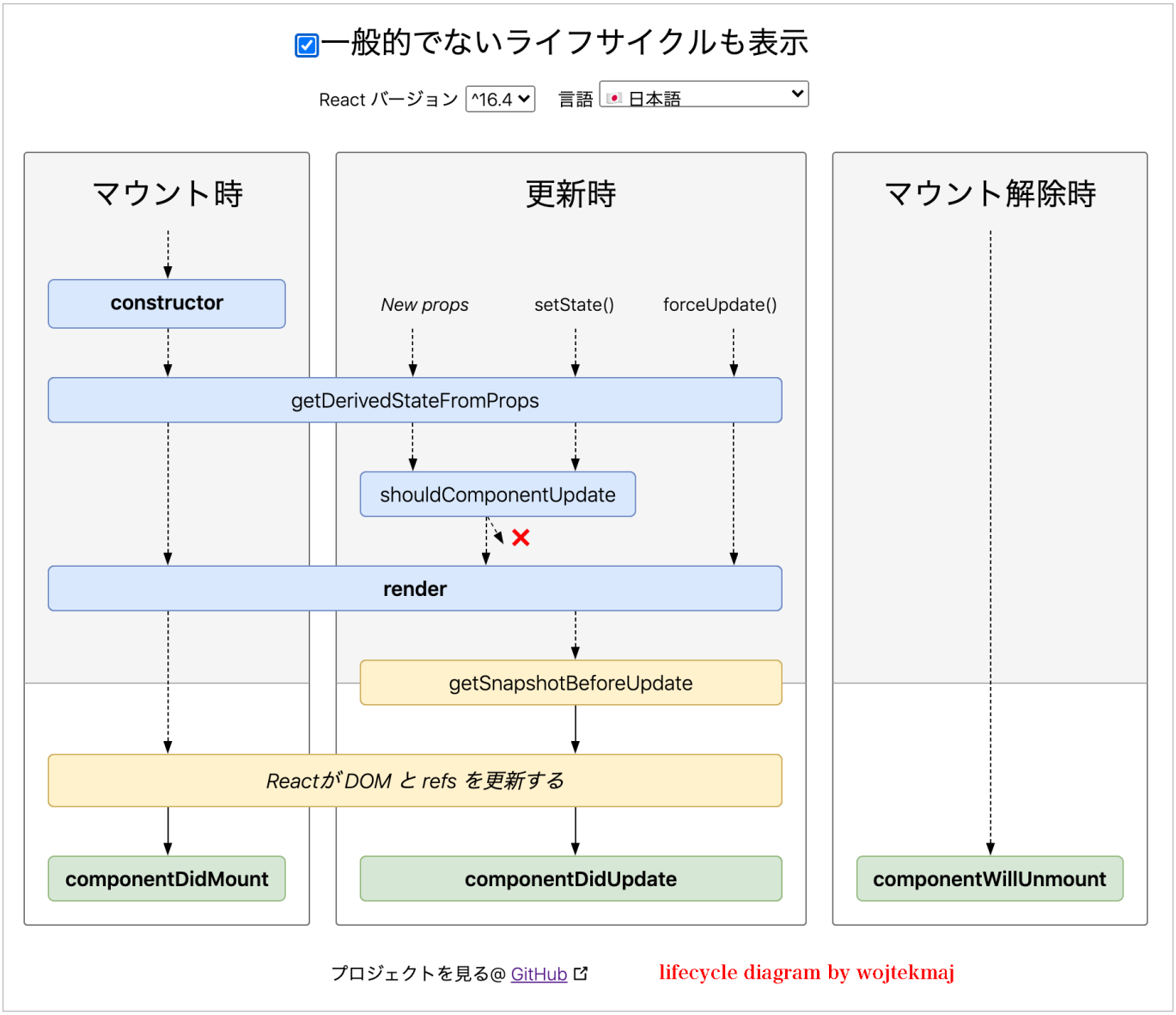
ライフサイクル

各コンポーネントには、処理の過程の特定の時点でコードを実行するためにオーバーライドできるいくつかの「ライフサイクルメソッド」があります。

React Lifecycle Methods diagramに掲載されているダイアグラムを見るとがわかりやすいです。以下はよく使われるライフサイクルメソッドのみを表示している状態のスクリーンショットです。



以下は一般的でないライフサイクルメソッドも表示している状態のスクリーンショットです。



コンポーネントのインスタンスが作成されて DOM に挿入されるときのことを React ではマウント（Mount）と呼びます。

マウント時には以下のメソッドが以下の順序で呼び出されます（ダイアグラム左側）。

- **constructor()**
- static getDerivedStateFromProps()
- **render()**
- **componentDidMount()**

更新（Update）は props や state の変更によって発生する可能性があり、コンポーネントが再レンダーされるときに以下のメソッドが以下の順序で呼び出されます（ダイアグラム中央）。

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- **render()**
- getSnapshotBeforeUpdate()
- **componentDidUpdate()**

コンポーネントが生成した DOM が削除されるときのことを React ではアンマウント（Unmount）と呼びます。アンマウント時（マウント解除）には以下のメソッドが呼び出されます。

- **componentWillUnmount()**

また、以下のメソッドはエラーが発生したときに呼び出されます。

- static getDerivedStateFromError()
- componentDidCatch()

メソッド名	説明
constructor()	マウントされる前に呼び出されます。state の初期化やメソッ
render()	クラスコンポーネントで必ず定義しなければならない唯一のメ
componentDidMount()	コンポーネントのインスタンスが作成されて DOM に挿入され
componentDidUpdate()	更新が行われた直後に呼び出されるメソッドです。コンポーネ
componentWillUnmount()	コンポーネントがアンマウントされて破棄される直前に呼び出

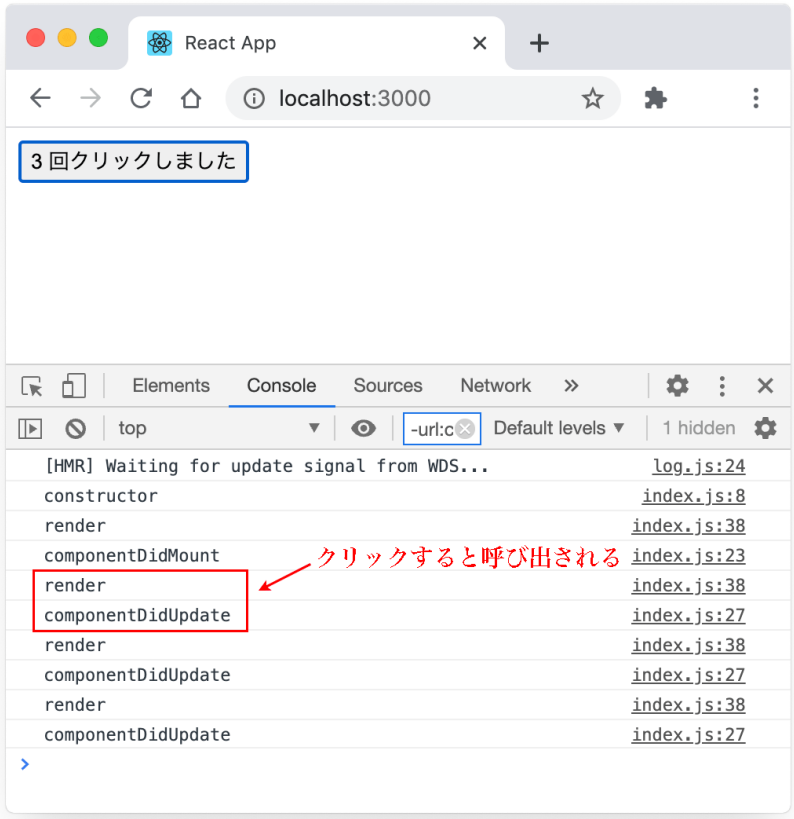
以下は前述のクリックするとその回数を表示するボタンのコンポーネントの例 にいくつかのライフサイクルメソッドを追加して、それらが呼び出されたらコンソールに表示する例です。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

class CountButton extends React.Component {

  constructor(props) {
    //コンストラクタが呼び出されたらコンソールに表示
    console.log('constructor');
    super(props);
    this.state = {
      count: 0,
    };
    this.updateCount = this.updateCount.bind(this);
  }

  updateCount() {
    this.setState((prevState) => {
      return { count: prevState.count + 1 }
    });
  }

  //コンポーネントがマウントされた直後に呼び出されるメソッド
  componentDidMount(){
    //呼び出されたらコンソールに表示
    console.log('componentDidMount');
  }

  //更新が行われた直後に呼び出されるメソッド
  componentDidUpdate(){
    //呼び出されたらコンソールに表示
    console.log('componentDidUpdate');
  }

  //コンポーネントがアンマウントされて破棄される直前に呼び出されるメソッド
  componentWillUnmount(){
    //呼び出されたらコンソールに表示
    console.log('componentWillUnmount');
  }

  render() {
    //呼び出されたらコンソールに表示
    console.log('render');
    return (
      <button onClick={this.updateCount}>
        {this.state.count} 回クリックしました
      </button>
     );
  }
}

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

ページを再読込してコンソールを確認すると、constructor、render、componentDidMount と出力され、ボタンをクリックする度に、render と componentDidUpdate が対で出力されるのが確認できます。

constructor と componentDidMount は初回レンダリング時にのみ出力されています。また、クリックしなければその後何も表示されません。

クリックすることにより state が変化し、それを React が検知して再レンダリング、つまり render() メソッドが実行され、その際に componentDidUpdate() が呼び出されています。



以下は「カウント開始」「停止」「リセット」の3つのボタンを持ち、「カウント開始」をクリックするとカウントを1秒ごとに1増加し、「停止」をクリックするとその時点でカウントを停止し、「リセット」をクリックするとカウントを0にリセットするコンポーネントの例です。

また、前述の例と同様、いくつかのライフサイクルメソッドを追加して、それらが呼び出されたらコンソールに表示するようにしています。

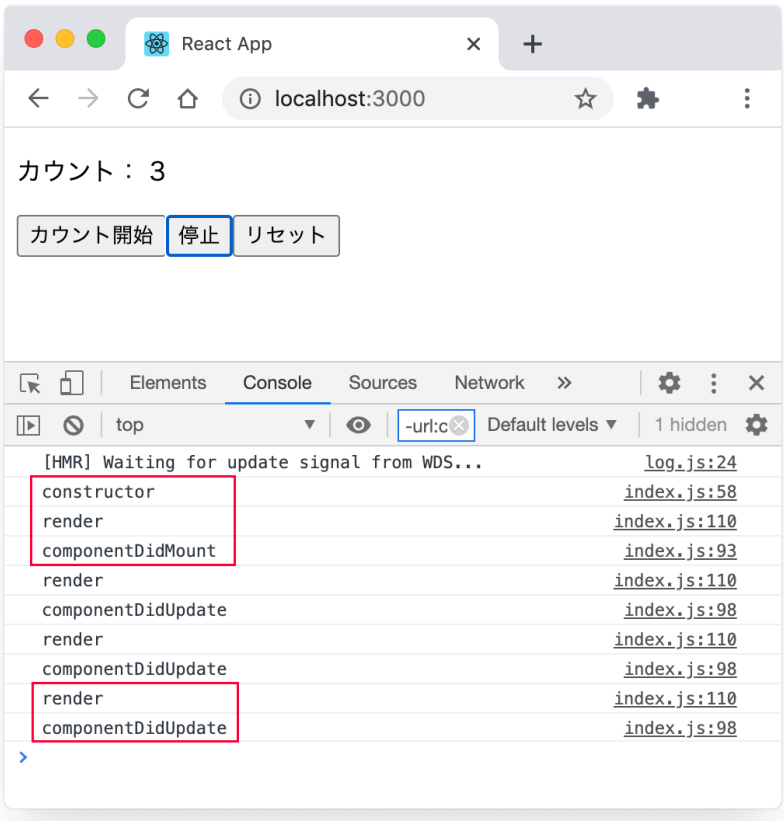
```
import React from 'react';
import ReactDOM from 'react-dom';

class CountButton extends React.Component {

  constructor(props) {
    //コンストラクタが呼び出されたらコンソールに表示
    console.log('constructor');
    super(props);
    this.state = {
      count: 0,
    };
    this.countUp = this.countUp.bind(this);
    this.stopCount = this.stopCount.bind(this);
    this.resetCount = this.resetCount.bind(this);
  }

  countUp() {
    //タイマーを設定
    this.timerID = setInterval(
      //1秒毎に count を1増加
      () => {this.setState((prevState) => {
        return { count: prevState.count + 1 }
      })},
      1000
    );
  }

  stopCount() {
    //タイマーをクリア
    clearInterval(this.timerID);
  }

  resetCount() {
    //タイマーをクリア
    clearInterval(this.timerID);
    //count を 0 にリセット
    this.setState({ count: 0 });
  }

  //コンポーネントがマウントされた直後に呼び出されるメソッド
  componentDidMount(){
    console.log('componentDidMount');
  }

  //更新が行われた直後に呼び出されるメソッド
  componentDidUpdate(){
    console.log('componentDidUpdate');
  }

  //コンポーネントがアンマウントされて破棄される直前に呼び出されるメソッド
  componentWillUnmount(){
    //タイマーをクリア
    clearInterval(this.timerID);
    console.log('componentWillUnmount');
  }

```

```
58.   render() {
59.     console.log('render');
60.
61.     return (
62.       <div>
63.         <p>カウント： {this.state.count} </p>
64.         <button onClick={this.countUp}>カウント開始</button>
65.         <button onClick={this.stopCount}>停止</button>
66.         <button onClick={this.resetCount}>リセット</button>
67.       </div>
68.      );
69.   }
70. }
71.
72. ReactDOM.render(
73.   <CountButton />,
74.   document.getElementById('root')
75. );
```

ページを再読込してコンソールを確認すると、前述の例と同様、初回レンダリング時に constructor、render、componentDidMount と出力されます。

「カウント開始」ボタンをクリックすると、countUp() が呼び出され、毎秒 setState() が実行され state の値が更新されるので、毎秒 render と componentDidUpdate が出力されます。

「停止」をクリックしても setState() は実行されないので、コンソールには何も出力されませんが、「リセット」をクリックすると setState() で値をリセットするので、render と componentDidUpdate が出力されます。



React：コンポーネントライフサイクル

以下は「レンダリングされた要素の更新 」で使った setInterval() を使って ReactDOM.render() を毎秒呼び出して時刻を1秒ごとに表示する例です。

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

//React 要素を作成して ReactDOM.render() を呼び出す関数
function tick() {
  //現在時刻を表示する React 要素を変数 element に代入
  const element = (
    <div>
      <h1>現在時刻： {new Date().toLocaleTimeString()} </h1>
    </div>
  );
  //React 要素を（element）をレンダリング
  ReactDOM.render(element, document.getElementById('root'));
}

//1000ミリ秒ごと（毎秒）関数 tick を呼び出して実行
setInterval(tick, 1000);
```

以下ではコンポーネントが自分でタイマーをセットアップし、自身を毎秒更新するように変更します。

まずは時刻表示の部分をコンポーネントとして分離します。

その際に props を使って、時刻 date を属性 date={new Date()} でコンポーネントに渡します。

```
import React from 'react';
import ReactDOM from 'react-dom';

//時刻表示の部分をコンポーネントとして分離
function Clock(props) {
  return (
    <div>
      <h1>現在時刻： {props.date.toLocaleTimeString()} </h1>
    </div>
  );
}

//「プロパティ名=値」として props を設定
function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Clock コンポーネントに state を追加するため、クラスコンポーネントに書き換えます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//React.Component を extends してクラスコンポーネントを定義
class Clock extends React.Component {
  //render() メソッドを追加して props を this.props に書き換え
  render() {
    return (
      <div>
        <h1>現在時刻： {this.props.date.toLocaleTimeString()} </h1>
      </div>
    );
  }
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

上記のコンポーネントは、setInterval() で1秒毎に tick() を実行することで ReactDOM.render() を呼び出しレンダリングを実行して表示を更新しています。

ReactDOM.render() を毎秒呼び出してレンダリングするのではなく、state を1秒毎に更新することで自動的に毎秒レンダリングされるように変更します。

そのためには state に現在の時刻を値として持たせて、その値を毎秒更新するようにします。

props で渡していた date を state に変更します。state を使うためコンストラクタ を追加して初期状態を設定します。

<Clock date={new Date()} />, の props を設定する属性 date={new Date()} は不要なので削除します。

タイマー（setInterval）で毎秒呼び出してコンポーネントをレンダリングしていた関数 tick() は削除して、コンポーネントは単に ReactDOM.render() で表示します。

タイマーは後でライフサイクルメソッドを使ってコンポーネント自身に設定します。この時点では表示される時刻はページを表示した時点の時刻で変化しません。

```
import React from 'react';
import ReactDOM from 'react-dom';

class Clock extends React.Component {
  //コンストラクタを追加して state の初期状態を設定
  constructor(props) {
    //コンストラクタのオーバライド
    super(props);
    //this.state の初期状態を設定
    this.state = {date: new Date()};
  }

  render() {
    //データの受け渡しに使っていた props を自身の状態を表す state に変更
    return (
      <div>
        <h1>現在時刻： {this.state.date.toLocaleTimeString()} </h1>
      </div>
    );
  }
}

//コンポーネントを表示する関数 tick() を削除
//データの受け渡しに使っていた props のための属性 date={new Date()} を削除
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

ライフサイクルメソッドの追加

コンポーネントクラスでライフサイクルメソッドを宣言することで、コンポーネントがマウントしたりアンマウントしたりした際にコードを実行することができます。

タイマーの設定は、Clock コンポーネントのインスタンスが生成されて DOM に挿入された直後に呼び出される componentDidMount() で行います。

また、タイマーの後片付け（クリア）は Clock コンポーネントが生成した DOM が削除されるときに呼び出される componentWillUnmount() で行います。

```
import React from 'react';
import ReactDOM from 'react-dom';

class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  //Mount 時のライフサイクルメソッドを追加
  componentDidMount() {
    //タイマーを設定（追加のフィールド this.timerID をクラスに追加）
    this.timerID = setInterval(
      //1秒毎にクラスのメソッド tick() を呼び出す
      () => this.tick(),
      1000
    );
  }

  //Unmount 時のライフサイクルメソッドを追加
  componentWillUnmount() {
    //タイマーをクリア
    clearInterval(this.timerID);
  }

  //state を更新するクラスのメソッド
  tick() {
    //setState() で state を更新
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>現在時刻： {this.state.date.toLocaleTimeString()} </h1>
      </div>
    );
  }
}


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

1. <Clock /> が ReactDOM.render() に渡されると、React は Clock コンポーネントのコンストラクタを呼び出します。Clock は state を使って現在時刻を表示するためコンストラクタで this.state を初期化して、あとでこの state を更新していきます。

2. 続いて React は Clock コンポーネントの render() メソッドを呼び出します。
3. Clock コンポーネントの render() メソッドに記述されているコードが DOM に挿入されると、React は componentDidMount() を呼び出し、その中で setInterval() で毎秒 tick() メソッドを呼び出すタイマーを設定します。
4. ブラウザは、毎秒 tick() メソッドを呼び出し、その中で setState() を呼び出すことで UI の更新をスケジュールします。setState() が呼び出されると、React は state が変わったことを検知し、render() メソッドを再度呼び出して DOM を更新します。
5. Clock コンポーネントが DOM から削除されれば、React は componentWillUnmount() を呼び出してタイマーが解除されます。

React：state とライフサイクル

イベント処理

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を追加するのではなく、要素が最初にレンダーされる際にリスナを指定するようにします。

また、React でのイベント処理は DOM 要素のイベントの処理と似ていますが、文法的な違いがあります。

- イベントは JSX の属性で onClick などのイベントハンドラで設定できますが、React のイベントは小文字ではなくキャメルケース（camelCase）を使います。
- JSX ではイベントハンドラとして文字列ではなく中括弧 { } で囲んで関数を渡します。

以下は Javascript での HTML の属性で指定する方法ですが、イベント onclick は小文字で、イベントハンドラとしてはダブルクォートで囲んで文字列を渡します。

Javascript のイベントの例（クリックと書かれたボタンをクリックすると Hello とアラート表示）

```
<button onclick="alert('Hello')">
  クリック
</button>
```

React ではイベントは onClick のようにキャメルケース（on の次が大文字）になり、イベントハンドラは中括弧 { } で囲んで関数を渡します。

React でのイベントの例

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  (
    <button onClick={() => alert('Hello')}>
      クリック
    </button>
  ),
  document.getElementById('root')
);
```

以下は function 文を使った場合です。

```
ReactDOM.render(
  (
    <button onClick={ function(){alert('Hello')} }>
      クリック
    </button>
  ),
  document.getElementById('root')
);
```

onClick プロパティに渡すのは関数です。onClick={alert('Hello')} と書いてしまうと、コンポーネントが再レンダーされるたびにアラートが表示されてしまいます。

イベントハンドラを別途定義することもできます。

この場合も onClick={ hello() } と関数名の後にカッコ ( ) を書いてしまうと、コンポーネントが再レンダーされるたびにアラートが表示されてしまいます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//イベントハンドラを別途定義
const hello = () =>  alert('Hello');

/* function 文を使ったイベントハンドラの定義の例（上記と同じこと）
function hello() {
  alert('Hello');
}
*/

ReactDOM.render(
  (
    <button onClick={ hello }>
      クリック
    </button>
  ),
  document.getElementById('root')
);
```

以下は関数コンポーネント内でイベントハンドラを別途定義する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';

//関数コンポーネント
const MyAlert = () => {

  //イベントハンドラを別途定義
  const hello = () =>  alert('Hello');

  return (
    <button onClick={hello}>
      クリック
    </button>
  )
}

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

以下はボタンをクリックすると、props で渡された props.name を使ってアラート表示する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';

//関数コンポーネント
const MyAlert = (props) => {

  //props で渡された props.name を使ってアラート表示（この場合は Hello Foo）
  const hello = () =>  alert('Hello ' + props.name);

  return (
    <button onClick={hello}>
      クリック
    </button>
  )
}

ReactDOM.render(
  <MyAlert name="Foo" />,
  document.getElementById('root')
);
```

イベントハンドラの引数では通常の JavaScript と同じようにイベントオブジェクトを受け取ることができます。受け取るイベント（以下の例では e） は合成イベント（SyntheticEvent）です。

以下は onChange 属性で値が変更された場合にその値（e.target.value）をコンソールに出力する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';

//関数コンポーネント
const MyEvent = () => {
  return (
    <div>
      <input type="text" onChange={e => console.log(e.target.value)}/>
    </div>
  )
}

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

イベントハンドラを別途定義する場合の例

```
import React from 'react';
import ReactDOM from 'react-dom';

const MyEvent = () => {

  //イベントハンドラを別途定義
  function inputval(e) {
    console.log(e.target.value);
  }

  /* アロー関数の場合
  const inputval = e => console.log(e.target.value);
  */

  return (
    <div>
      <input type="text" onChange={inputval} />
    </div>
  )
}

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

デフォルトの動作を抑止

React では return false; で false を返してもデフォルトの動作を抑止することができないため、明示的に preventDefault を呼び出す必要があります。

以下は preventDefault で a 要素をクリックした際のデフォルトの動作（リンク先への移動）を抑止する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';

//関数コンポーネント
const MyActionLink = () => {
  function handleClick(e) {
    //明示的に preventDefault を呼び出してデフォルトの動作を抑止
    e.preventDefault();
    console.log('クリックされました');
  }

  return (
    <a href="#" onClick={handleClick}>
      クリック
    </a>
  );
}

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

クラスコンポーネントでのイベント処理

クラスコンポーネントでイベントハンドラを定義する際に this を使って値を参照をする場合は、コンストラクタ内でイベントハンドラと this をバインドしておく必要があります。

以下の場合は、イベントハンドラで this を使って参照をしていないので問題ありません。

```
1.  import React from 'react';
2.  import ReactDOM from 'react-dom';
3.
4.  // コンポーネントの定義
5.  class MyAlert extends React.Component {
6.
7.    //単に文字列を出力
8.    hello() {
9.      alert('Hello');
10.   }
11.
12.   render() {
13.     return (
14.       <button onClick={this.hello}>
15.         Hello
16.       </button>
17.     )
18.   }
19. }
20.
21. ReactDOM.render(
22.   <MyAlert />,
23.   document.getElementById('root')
24. );
```

以下のようにイベントハンドラ（クラスのメソッド）で props や state などを参照するには this が必要ですが、その場合、コンストラクタ内でイベントハンドラと this をバインドする必要があります。

9行目のバインドの記述がないと、ボタンをクリックした際にイベントハンドラ（14行目）の props が undefined となりエラー（TypeError: Cannot read property 'props' of undefined）になります。

```
import React from 'react';
import ReactDOM from 'react-dom';

// コンポーネントの定義
class MyAlert extends React.Component {
  constructor(props) {
    super(props);
    //イベントハンドラ（クラスのメソッド）と this をバインド
    this.hello = this.hello.bind(this);
  }

  //props を this を使って参照
  hello() {
    alert('Hello ' + this.props.name);
  }

  render() {
    return (
      <button onClick={this.hello}>
        クリック
      </button>
    )
  }
}

ReactDOM.render(
  <MyAlert name="Foo" />,
  document.getElementById('root')
);
```

以下は初期状態では緑色で ON と表示されているボタンをクリックすると赤色 OFF に切り替わり、再度クリックすると緑色の ON になる Toggle コンポーネントの例です。

MyStyle.css

```
.toggleOn {
  color: green;
}

.toggleOff {
  color: red;
}
```

state の isToggleOn プロパティに真偽値（最初は true）を設定して、クリックされる度にイベントハンドラ handleClick() で state の値（真偽値）を！で反転させています。

レンダリングする際には、state の値により表示する文字（ON と OFF）とクラス（toggleOn と toggleOff）を三項演算子で切り替えます。

```
import React from 'react';
import ReactDOM from 'react-dom';
import './MyStyle.css';


class Toggle extends React.Component {
  constructor(props) {
    super(props);
    // state の初期化（初期値の設定）
    this.state = {isToggleOn: true};
    // イベントハンドラに this をバインド
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    //クリックすると現在の isToggleOn の値（state.isToggleOn）を反転させる
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    //onClick で handleClick を呼び出し、その戻り値により表示する文字とクラスを切り替
える
    return (
      <button onClick={this.handleClick} className={this.state.isToggleOn ?
'toggleOn':'toggleOff'}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

イベントハンドラ handleClick() を function 文で書き換えると以下のように記述できます。

```
handleClick() {
  //クリックすると現在の isToggleOn の値の真偽値（true/false）を反転させて返す
  this.setState(function(state) {
    return {
      isToggleOn: !state.isToggleOn
    };
  });
}
```

バインドを使わない方法

React のイベント処理のページではバインドを使わない方法として以下の2つが紹介されています。

以下は実験的なパブリッククラスフィールド構文を使用する方法です。この方法は Create React App を使った環境では使用できますが、独自に構築した環境などで使用するには babel-plugin-transform-class-properties がインストールされている必要があります。

```
import React from 'react';
import ReactDOM from 'react-dom';

// コンポーネントの定義
class MyAlert extends React.Component {

  //実験的なパブリッククラスフィールド構文を使用
  hello = () => {
    alert('Hello ' + this.props.name);
  }

  render() {
    return (
      <button onClick={this.hello}>
        クリック
      </button>
    )
  }
}

ReactDOM.render(
  <MyAlert name="Foo" />,
  document.getElementById('root')
);
```

以下はコールバック内でアロー関数を使用する方法です。

但し、以下の場合、シンプルな構造では問題ありませんが、MyAlert 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てしまうので効率的ではありません。

```
import React from 'react';
import ReactDOM from 'react-dom';

// コンポーネントの定義
class MyAlert extends React.Component {

  //props を this を使って参照
  hello() {
    alert('Hello ' + this.props.name);
  }

  render() {
    //アロー関数を使用
    return (
      <button onClick={() => this.hello()}>
        クリック
      </button>
    )
  }
}

ReactDOM.render(
  <MyAlert name="Foo" />,
  document.getElementById('root')
);
```

React：イベント処理

React：合成イベント (SyntheticEvent)

条件付きレンダー

JavaScript の if 文や条件演算子などの条件分岐を使ってコンポーネントの表示（レンダリング）を制御することを条件付きレンダーと言います。

以下は if 文を使ってユーザがログインしているかどうかによって、2つコンポーネントの一方だけを表示する例です。props.isLoggedIn の値により、<UserGreeting /> か <GuestGreeting /> を表示します。

ログインしているかどうかの状態の設定については別途 state を使って実装します。

25行目の false を true に変更すると「ようこそ！」と表示されます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//ユーザがログイン後に表示するタイトルのコンポーネント
function UserGreeting(props) {
  return <h1>ようこそ！</h1>;
}

//ユーザがログインしていない場合に表示するタイトルのコンポーネント
function GuestGreeting(props) {
  return <h1>ログインしてください。</h1>;
}

function Greeting(props) {
  //props で受け取る値（props.isLoggedIn）を変数 isLoggedIn に代入
  const isLoggedIn = props.isLoggedIn;

  // 変数 isLoggedIn の値により表示するコンポーネントを切り替える
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // isLoggedIn の値の設定は別途実施。以下の場合は「ログインしてください。」と表示される
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

上記の場合、条件により表示するコンポーネントを切り替えていますが、以下は条件によりコンポーネントで表示する文字を切り替える例で、上記と同じ結果になります。

```
//表示する文字を props で受け取りその値を元に表示を切り替える
function GreetingTitle(props) {
  return <h1>{props.greeting}</h1>;
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  // isLoggedIn の値により GreetingTitle コンポーネントへ渡す props を切り替える
  if (isLoggedIn) {
    //表示する文字を props に設定
    return <GreetingTitle greeting="ようこそ！"/>;
  }
  return <GreetingTitle  greeting="ログインしてください。"/>;
}

ReactDOM.render(
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

上記の例の Greeting コンポーネントの if 文の代わりに3項演算子を使えば以下のように記述することもできます。

```
function GreetingTitle(props) {
  return <h1>{props.greeting}</h1>;
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  // if 文の代わりに3項演算子を使う例
  return <GreetingTitle greeting={isLoggedIn ? "ようこそ！":"ログインしてくださ
い。"}/>;
}

ReactDOM.render(
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

要素変数

React 要素を変数に保持して使うことができます。

以下はログアウトとログインボタンを表す2つの新しいコンポーネントを作成してそれらを条件により変数に格納してレンダーする例です。

以下のような2つのコンポーネントを作成します。

```
//ログインボタンのコンポーネント
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      ログイン
    </button>
  );
}

//ログアウトボタンのコンポーネント
function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      ログアウト
    </button>
  );
}
```

以下は state を使ってログインとログアウトの状態を設定したコンポーネント LoginControl です。

LoginControl は現在の state の値によって、変数に保持した <LoginButton /> もしくは <LogoutButton /> の一方をレンダーし、同時に前の例の <Greeting /> もレンダーします。

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    //イベントハンドラと this をバインド
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    // state の初期化（isLoggedIn の初期値の設定）
    this.state = {isLoggedIn: false};
  }

  //イベントハンドラ
  handleLoginClick() {
    //this.state.isLoggedIn を true に
    this.setState({isLoggedIn: true});
  }

  //イベントハンドラ
  handleLogoutClick() {
    //this.state.isLoggedIn を false に
    this.setState({isLoggedIn: false});
  }

  render() {
    // 現在の state を変数 isLoggedIn に代入
    const isLoggedIn = this.state.isLoggedIn;
    //ログアウトまたはログインボタンを格納する変数
    let button;
    if (isLoggedIn) {
      //ログインしている状態の場合は button にログアウトボタンのコンポーネントを格納
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      //ログインしていない状態の場合は button にログインボタンのコンポーネントを格納
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      //前述の Greeting コンポーネントと変数に格納したボタンのコンポーネントをレンダー
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

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

以下が全文です。

ReactDOM.render() による初回の表示では state.isLoggedIn は false（初期値）なので、Greeting コンポーネントは GuestGreeting を返すので「ログインしてください。」と表示されます。

同様に LoginControl コンポーネントでは変数 button に LoginButton が代入されて返されるので「ログイン」と表示されたログインボタンが表示されます。

ログインボタンがクリックされると handleLoginClick が呼び出され isLoggedIn が true になり state の状態が変化し、自動的に LoginControl コンポーネントの render() が呼び出され表示が変化します。

```
import React from 'react';
import ReactDOM from 'react-dom';

function UserGreeting(props) {
  return <h1>ようこそ！</h1>;
}

function GuestGreeting(props) {
  return <h1>ログインしてください。</h1>;
}

//props の値により UserGreeting または GuestGreeting を返すコンポーネント
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      ログイン
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      ログアウト
    </button>
  );
}

// state の値により LoginButton か LogoutButton のどちらかと Greeting のコンポーネントを表示するコンポーネント
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;
```

```
    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

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

短絡評価(ショートサーキット)

JSX 内では任意の JavaScript の式を中括弧 { } で囲んで使用することができるので、論理 && 演算子を使った短絡評価を使うと、簡潔な記述で条件に応じて要素を含めることができます。

以下は props で渡される未読のメッセージ（props.unreadMessages）の数（length）を調べて、1以上の場合のみメッセージを含めて表示する例です。

以下の場合 unreadMessages.length > 0 が true であれば && 以降の値（右辺）が評価されて表示されますが、false の場合は無視されるので表示されません。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MessageBox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>こんにちは。</h1>
      {unreadMessages.length > 0 &&
        <h2>
          未読のメッセージ： {unreadMessages.length} 件
        </h2>
      }
    </div>
  );
}

//messages を空の配列にすると、「未読のメッセージ： X 件」の部分は表示されません。
const messages = ['更新情報', 'パスワード変更'];
ReactDOM.render(
  <MessageBox unreadMessages={messages} />,
  document.getElementById('root')
);
```

出力の代わりに null を返す（レンダーを防ぐ）

他のコンポーネントによってあるコンポーネントがレンダーされているようになっている場合に、条件によってそのコンポーネントをレンダリングしたくない場合は、出力の代わりに null を返すことでそのコンポーネントがレンダリングされるのを防ぐことができます。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

//お知らせを表示するコンポーネント
function InfoBanner(props) {
  // プロパティ info の値が false なら null を返してコンポーネントを表示しない
  if (!props.info) {
    return null;
  }

  return (
    <div>
      お知らせ： {props.info_msg}
    </div>
  );
}

//表示・非表示をトグルするボタンと InfoBanner コンポーネントを表示するコンポーネント
class MyPage extends React.Component {
  constructor(props) {
    super(props);
    // state の初期値の設定
    this.state = {showInfo: true};
    // イベントハンドラのバインド
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  //ボタンがクリックされた際に呼び出されるイベントハンドラ
  handleToggleClick() {
    this.setState(state => ({
      //現在の showInfo の値（state.showInfo）を反転（トグル）させる
      showInfo: !state.showInfo
    }));
  }

  render() {
    return (
      <div>
        // props.info に state.showInfo の真偽値を渡し、
        // props.info_msg に表示するお知らせの文字を渡す
        <InfoBanner info={this.state.showInfo} info_msg={this.props.info_msg}/>
        // onClick でイベントハンドラを登録
        <button onClick={this.handleToggleClick}>
          // ボタンに表示する文字は state.showInfo の値により変更
          {this.state.showInfo ? '非表示' : '表示'}
        </button>
      </div>
    );
  }
}

ReactDOM.render(
  <MyPage info_msg="今週末に音楽イベントがあります。"/>,
  document.getElementById('root')
);
```

リストと key

以下は配列を props 経由で受け取り、その配列の要素をリストで表示するコンポーネントの例です。

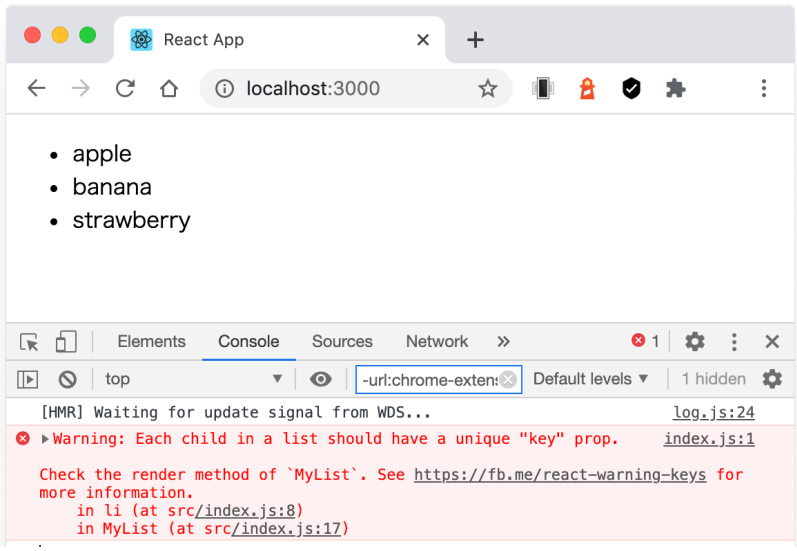
配列の要素をリストで表示するには map() 関数を利用しています。

map() 関数では、コールバック関数で配列（dataList）の各要素の値を item として受け取り、順番に li 要素でラップして新しい配列 listItems に格納しています。

そして li 要素の配列 listItems を ul 要素でラップして返しています。

```
import React from 'react';
import ReactDOM from 'react-dom';


function MyList(props) {
  const dataList = props.dataList;
  // map() を使って配列の要素を li 要素に変換して新しい配列 listItems に格納
  const listItems = dataList.map((item) =>
    <li>{item}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

//リストで表示する配列
const fruits = ['apple', 'banana', 'strawberry'];

//配列を props.dataList で渡す
ReactDOM.render(
  <MyList dataList={fruits} />,
  document.getElementById('root')
);
```

上記のコードを実行すると、以下のように「Warning: Each child in a list should have a unique "key" prop.（リストの子要素にはユニークな key を与えるべきだ）」という警告がコンソールに表示されます。



key

"key" とは要素のリストを作成する際に含めておく必要がある特別な（文字列の）属性です。

JSX の記述で map() などを使ったループ処理で繰り返し生成される要素（動的に生成されるリストの要素など）には key 属性を設定する必要があります。

Key はどの要素が変更、追加もしくは削除されたのかを React が識別するのに利用され、React がリストの変更を効率的に検出するために使用されます。

key は兄弟間でその項目を一意に特定できるような文字列を指定します。兄弟要素間で一意であればよく、全体で一意である必要はありません（別のループで使われている値と重複しても問題ありません）。

また、配列の要素のインデックスを key として渡すことができますが、項目が並び替えられる（ソートされる）ことがなければ問題ありませんが、並び替えられると動作が遅くなります。

React：子要素の再帰的な処理

以下は前述のコンポーネントのリスト項目（li 要素）に key を追加する例です。

この例では map() 関数のコールバック関数で第2引数として受け取れる配列のインデックスを key として渡しています。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyList(props) {
  const dataList = props.dataList;
  //コールバック関数の第2引数のインデックスを key に使用
  const listItems = dataList.map((item,index) =>
    <li key={index}>{item}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const fruits = ['apple', 'banana', 'strawberry'];
ReactDOM.render(
  <MyList dataList={fruits} />,
  document.getElementById('root')
);
```

上記は map() 関数の返す配列を変数 listItems に代入していますが、JSX 内の { } の中では式（評価した結果を変数に代入できるもの）が使えるので、以下のように直接 JSX に埋め込むこともできます。

```
function MyList(props) {
  const dataList = props.dataList;
  return (
    <ul>
      {dataList.map((item,index) =>
        <li key={index}>{item}</li>
      )}
    </ul>
  );
}

const fruits = ['apple', 'banana', 'strawberry'];
ReactDOM.render(
  <MyList dataList={fruits} />,
  document.getElementById('root')
);
```

前述の例では配列のインデックスを key として渡しているため、リストの項目がソートされたり並び替えられる場合は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。

一番良いのはデータ内にある安定した ID（並び替えても変わらない値）を key として使う方法です。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyList(props) {
  const dataList = props.dataList;
  const listItems = dataList.map((user) =>
    <li key={user.id}>{user.name}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

// データがユニークで安定した ID を持っている配列
const users = [
  { id: 'u-01', name: 'Taylor'},
  { id: 'u-02', name: 'Smith'},
  { id: 'u-03', name: 'Davis'}
];

ReactDOM.render(
  <MyList dataList={users} />,
  document.getElementById('root')
);
```

key のあるコンポーネントの抽出

key のあるコンポーネントを抽出する場合、key は map() を呼び出す側で付与します。

以下は前述の例のリストの項目を ListItem コンポーネントとして抽出する例です。key は map() を呼び出す側に残します。

```
import React from 'react';
import ReactDOM from 'react-dom';

//コンポーネントを分離（抽出）
function ListItem(props) {
  // こちらには key は不要
  return <li>{props.name}</li>;
}

function MyList(props) {
  const dataList = props.dataList;
  const listItems = dataList.map((user) =>
    //key は map() 側に残す
    <ListItem key={user.id} name={user.name} />
  );
  return (
    <ul>{listItems}</ul>
  );
}

const users = [
  { id: 'u-01', name: 'Taylor'},
  { id: 'u-02', name: 'Smith'},
  { id: 'u-03', name: 'Davis'}
];

ReactDOM.render(
  <MyList dataList={users} />,
  document.getElementById('root')
);
```

React：リストと key

フォーム

HTML では <input> や <textarea>、<select> などのフォーム要素は、フォーム自身で状態（データ）を保持していて、ユーザの入力に基づいてそれを更新します。

React では状態を state プロパティに保持し、setState() 関数 でのみ更新します。

React の state を「信頼できる唯一の情報源」とすることで上述の 2 つの状態を結合させることができ、そのような方法を「制御されたコンポーネント」と呼びます。

制御されたコンポーネント

フォームを制御されたコンポーネントとするには、フォーム要素の value 属性に state を関連付けます。

それには フォーム要素の value 属性に state プロパティを設定し、onChange イベントで setState() を呼び出して state プロパティを更新することで value 属性を更新します。これにより、データ（state）と UI（入力）は常に同期します。

- state が持つ状態を入力欄に反映（value 属性に state プロパティの値を指定）
- 各入力欄への入力内容を state へ随時保存（onChange イベントで state プロパティを更新）

また、フォームの送信ボタンが押された際の処理では、その時点での state を使って処理を実行しますが、デフォルト動作のフォームの送信によりページがリロードされないように preventDefault() を実行します。

input type="text"

以下は <input type="text"> タグを使ったフォームの例です。

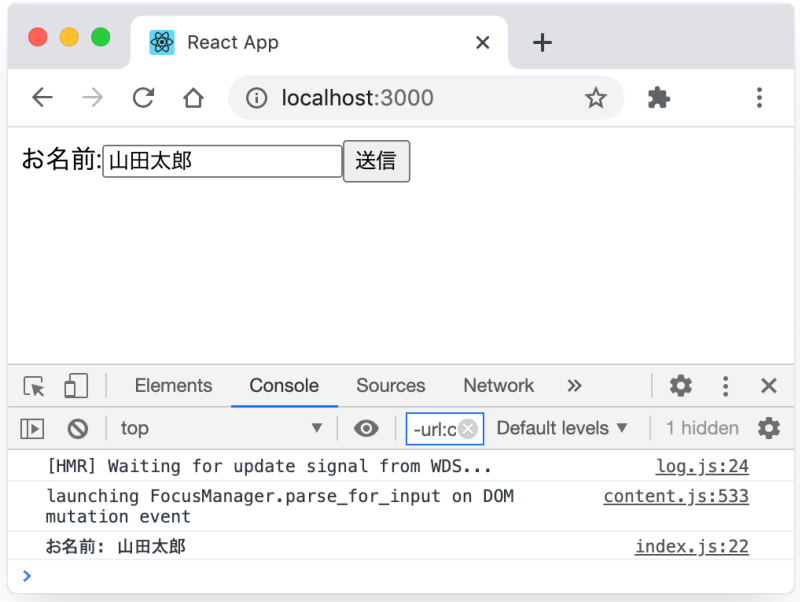
input 要素の value 属性に this.state.value を指定します。そして onChange で設定したイベントハンドラ handleChange() で setState() を呼び出して入力されている値 event.target.value で state.value を更新することで value 属性（input 要素に表示される値）が更新されます。

onChange イベントはキーストロークごとに発生し、handleChange() はその都度実行されるので、表示される値はユーザがタイプするたびに更新されます。

送信ボタンがクリックされると onSubmit で設定したイベントハンドラ handleSubmit() で更新された値 state.value を出力し、event.preventDefault() でデフォルト動作のフォームの送信を止めて、ページがリロードしないようにしています。

このようにすることで、ユーザ入力の値は常に React の state によって制御されるようになります。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    // state プロパティの初期化（state.value の初期値の設定）
    this.state = {value: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  // onChange イベントのハンドラ
  handleChange(event) {
    // ユーザがタイプする度に state を入力される値で更新 → value 属性も更新される
    this.setState({value: event.target.value});
  }

  // onSubmit イベントのハンドラ
  handleSubmit(event) {
    // 送信ボタンがクリックされたらその時点での state をコンソールに出力
    console.log(this.props.name + ': ' + this.state.value);
    //デフォルトの動作（フォームの送信）を抑止
    event.preventDefault();
  }

  render() {
    // value 属性の値に state.value を指定して state を関連付け
    // onChange イベントで value 属性の値を随時更新
    return (
      <form onSubmit={this.handleSubmit}>
      <label>
      {this.props.name}:
        <input type="text" value={this.state.value} onChange={this.handleChange}
/>
      </label>
      <input type="submit" value="送信" />
      </form>
    );
  }
}

ReactDOM.render(
  <MyForm name="お名前"/>,
  document.getElementById('root')
);
```



textarea タグ

HTML では、<textarea> 要素はテキストを子要素として定義しますが、React では代わりに value 属性を使用します。このため、<textarea> を使用するフォームは <input> の入力フォームと同じような書き方ができるようになっています。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    // 初期値を指定しているのでテキストエリアには始めから以下の文字が表示されます
    this.state = {value: '文字を入力して送信ボタンをクリックしてください。'};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    console.log(this.props.name + ': ' + this.state.value);
    event.preventDefault();
  }

  render() {
    // React では textarea 要素の値も value 属性で扱える
    // input 要素の例の場合とほぼ同じ記述
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
        {this.props.name}: <br/>
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <br/>
        <input type="submit" value="送信" />
      </form>
    );
  }
}

ReactDOM.render(
  <MyForm name="入力"/>,
  document.getElementById('root')
);
```

input 要素の例と同様に textarea 要素の value 属性に this.state.value を指定し、onChange のイベントハンドラ handleChange() で setState() を呼び出して state.value を更新しています。

上記の例の場合は、コンストラクタ内で state の初期化で初期値となる文字列を指定しているので、始めから「文字を入力して送信ボタンをクリックしてください。」と表示されます。

select タグ

HTML では、<select> は以下のように選択肢を option 要素で記述し、初期状態で選択されている状態にするには selected 属性を指定します。

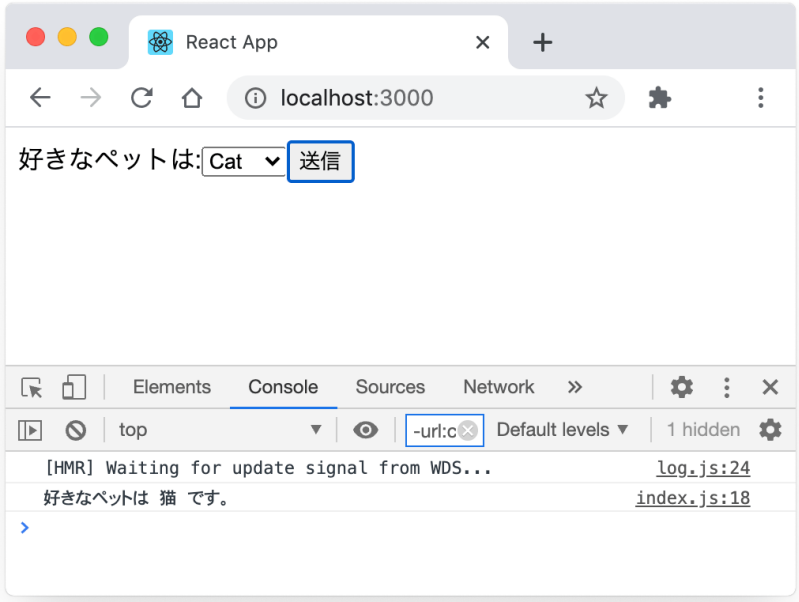
HTML

```html
<select>
  <option value="猫">Cat</option>
  <option value="犬">Dog</option>
  <option selected value="うさぎ">Rabit</option>
  <option value="魚">Fish</option>
</select>
```

React では selected 属性の代わりに select 要素の value 属性を使用し、value 属性に設定した値を持つ option 要素が自動的に選択状態になります。

そのため、select 要素の value 属性に this.state.value を指定します。そして onChange で設定したイベントハンドラ handleChange() で setState() を呼び出して選択されている値 event.target.value で state.value を更新します。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    //初期状態で選択されている値を初期値として設定
    this.state = {value: 'うさぎ'};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    console.log('好きなペットは ' + this.state.value + ' です。');
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          好きなペットは:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="猫">Cat</option>
            <option value="犬">Dog</option>
            <option value="うさぎ">Rabit</option>
            <option value="魚">Fish</option>
          </select>
        </label>
        <input type="submit" value="送信" />
      </form>
    );
  }
}

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



※ <input type="text">、<textarea>、<select> はいずれも value 属性を受け取り、非常に似た動作をするようになっています。

input type="radio"

以下はラジオボタンの例です。項目を選択されている状態にするには checked 属性を true に指定します。

イベントの発生した（ラジオボタンが選択された）要素の値を this.state.value に更新します。

そのボタンが選択されているかどうかは、現在選択されている値（this.state.value）がその value 属性の値と一致するかを === で判定した真偽値を checked 属性に設定します。

初期状態で選択状態にするには初期化の際に値を設定します（空文字を設定すれば何も選択されていない初期状態になります）。

この方法以外にももっと良い方法があるかも知れません。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    //初期状態で選択されている値を初期値として設定
    this.state = {value: '猫'};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    console.log('好きなペットは ' + this.state.value + ' です。');
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          <input
            type="radio"
            value="猫"
            onChange={this.handleChange}
            checked={this.state.value === '猫'}
          />
          Cat
        </label>
        <label>
          <input
            type="radio"
            value="犬"
            onChange={this.handleChange}
            checked={this.state.value === '犬'}
          />
          Dog
        </label>
      <label>
          <input
            type="radio"
            value="うさぎ"
            onChange={this.handleChange}
            checked={this.state.value === 'うさぎ'}
          />
          Rabit
        </label>
        <input type="submit" value="送信" />
      </form>
    );
  }
}

```

```
58. ReactDOM.render(
59.   <MyForm />,
60.   document.getElementById('root')
61. );
```

input type="checkbox"

以下はチェックボックスの例です。項目を選択されている状態にするにはラジオボタン同様 checked 属性を true に指定しますが、チェックボックスの場合は複数選択可能になっています。そのため以下の例では値は配列で保持しています。

項目が選択されているかどうかは、includes() でその項目の値が state に保持されている値（配列）に含まれているかどうかを判定して checked 属性に真偽値を設定します。

チェックボックスが操作されると（イベントが発生すると）イベントが発生した要素の値と state に保持されている値（配列）を比較して、イベントが発生した要素の値が含まれていればチェックを外したと判断してその要素を除いた配列を filter() で新たに生成して setState で更新します。イベントが発生した要素の値が配列に含まれていなければ、チェックを入れたと判断し、イベントが発生した要素を末尾に追加した新たな配列をスプレッド構文で生成して setState で更新します。

前述のラジオボタンの例と同様、この方法以外にももっと良い方法があるかも知れません。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    //初期状態で選択されている値を初期値として設定（複数選択可能なので配列で保持）
    this.state = {value: ['うさぎ']};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    //イベントが発生した要素の値
    const eventValue = event.target.value;
    //state に保持されている値（配列）
    const stateValue = this.state.value;

    if (stateValue.includes(eventValue)) {
      // state の値（配列）にイベントが発生した要素の値が含まれていればチェックを外したと判断し、イベントが発生した要素を除いた配列を生成して返す
      this.setState({value: stateValue.filter(item => item !== eventValue)});
    } else {
      // そうでなければチェックを入れたと判断し、イベントが発生した要素を末尾に加えた配列を生成して返す
      this.setState({value: [...stateValue, eventValue]});
    }
  }

  handleSubmit(event) {
    console.log('好きなペットは ' + this.state.value + ' です。');
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          <input
            type="checkbox"
            value="猫"
            onChange={this.handleChange}
            checked={this.state.value.includes('猫')}
          />
          Cat
        </label>
        <label>
          <input
            type="checkbox"
            value="犬"
            onChange={this.handleChange}
            checked={this.state.value.includes('犬')}
          />
          Dog
        </label>
        <label>
          <input
```

```
            type="checkbox"
            value="うさぎ"
            onChange={this.handleChange}
            checked={this.state.value.includes('うさぎ')}
          />
          Rabit
        </label>
        <input type="submit" value="送信" />
      </form>
    );
  }
}

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

チェックボックスの項目がチェックされているかどうかは event.target.checked でも取得することができます。

入力された値のチェック

HTML5 の Form Validation 機能を利用する方法です。それ以上の検証はしていません。

以下は、入力された値のチェックのために、HTML5 の Form Validation 機能を input 要素に追加して、必須（required）で英数字のみ（pattern="[A-Za-z0-9]+"）とする例です。

入力された文字が英数字であれば、送信ボタンをクリックすると入力された文字に Hello を付けて画面に表示します。入力された文字が英数字以外であったり、未入力の場合はエラーを表示します。

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: '',
      message: 'Hello'
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    this.setState({
      message: 'Hello, ' + this.state.value + '!'
    })
    event.preventDefault();
  }

  render() {
    return (
      <div>
        <h3>{this.state.message}</h3>
        <form onSubmit={this.handleSubmit}>
          <label>
            Name:
            <input type="text" value={this.state.value} onChange=
{this.handleChange} required pattern="[A-Za-z0-9]+"/>
          </label>
          <input type="submit" value="送信" />
        </form>
      </div>
    );
  }
}

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

複数の入力の処理

複数の「制御された」input 要素を処理する場合、それぞれの入力要素に name 属性を指定すれば、event.target.name に基づいて1つのイベントハンドラで処理をすることができます。

以下は2つの <input type="text"> にそれぞれ name 属性（input1 と input2）を設定して1つのイベントハンドラで処理する例です。

両方の input 要素の onChange 属性には同じイベントハンドラ this.handleChange を設定しています。

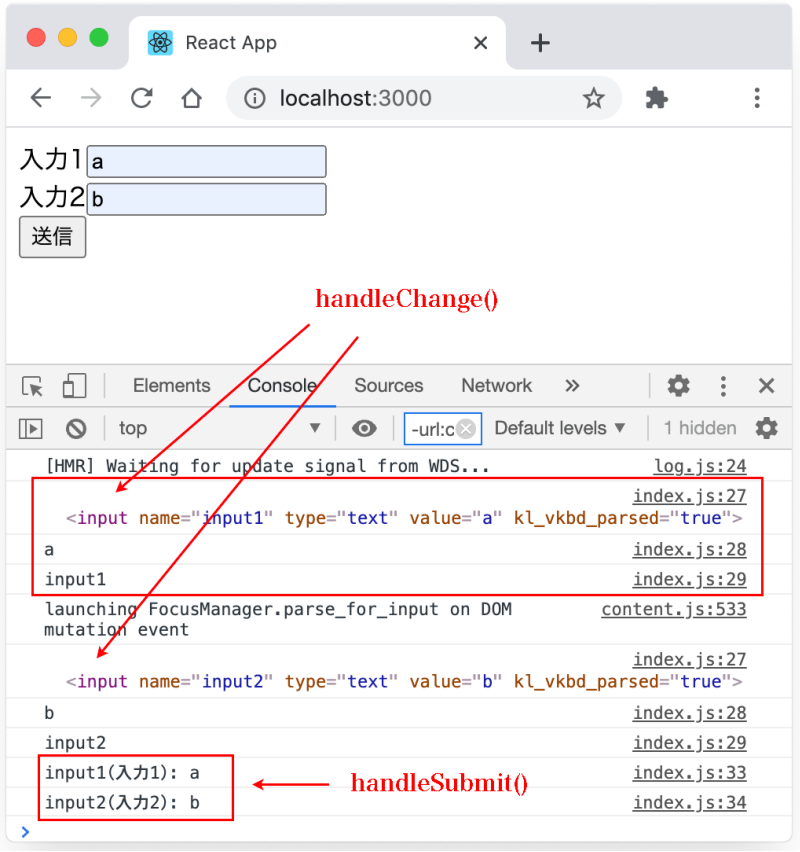
イベントハンドラ handleChange では、発生したイベントの target 属性でイベントが発生した要素の name 属性を特定して、それに基づいて処理をします。

また、setState() で state を更新する際は、[name] のようにオブジェクトのプロパティ名を [ ] で囲むことで変数 name を展開することができます（ES6 算出プロパティ）。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input1: '',
      input2: ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    //イベントの発生した要素
    const target = event.target;
    //イベントの発生した要素の値
    const value = target.value;
    //イベントの発生した要素の name 属性の値
    const name = target.name;
    this.setState({
      // [name] は算出プロパティ（算出されたキー）
      [name]: value
    });
    /* //確認用の出力
    console.log(target);
    console.log(value);
    console.log(name);*/
  }

  handleSubmit(event) {
    console.log('input1: ' + this.state.input1);
    console.log('input2: ' + this.state.input2);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
        入力1
          <input name="input1" type="text" value={this.state.input1} onChange={this.handleChange} />
        </label>
        <br/>
        <label>
        入力2
          <input name="input2" type="text" value={this.state.input2} onChange={this.handleChange} />
        </label>
        <br/>
        <input type="submit" value="送信" />
      </form>
    );
  }
}
```

```
56.
57. ReactDOM.render(
58.   <MyForm />,
59.   document.getElementById('root')
60. );
```

以下は上記26～29行目のコメントアウトを外した場合のコンソール出力のスクリーンショットです。



上記の例のイベントハンドラ handleChange は以下のように記述することができます。

```
1. handleChange(event) {
2.   this.setState({
3.     [event.target.name]: event.target.value
4.   });
5. }
```

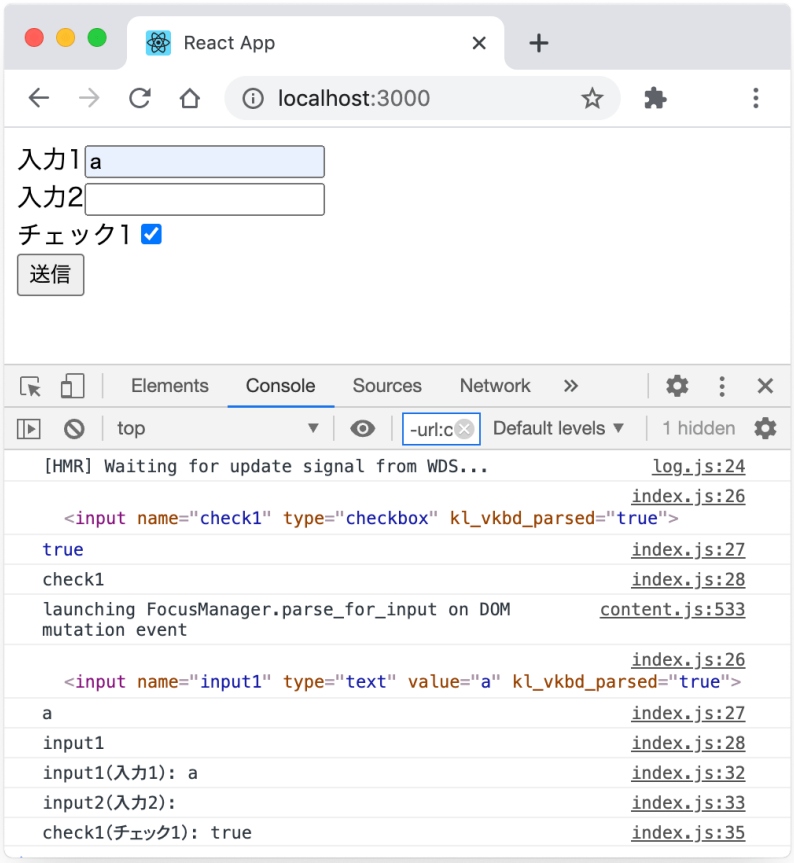
以下はチェックボックスを追加した例です。

チェックボックスの場合、値（target.value）ではなくチェックされているかどうか（target.checked）を使用するので、イベントの発生した要素を name 属性で判定して state に設定する値を処理しています。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input1: '',
      input2: '',
      // チェックボックスの初期値（チェックがついていない状態）
      check1: false
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    const target = event.target;
    //値は nama 属性でチェックボックスかテキストボックスかを判定
    //チェックボックスの場合は checked を、そうでなければ value を代入
    const value = target.name === 'check1' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
    /* //確認用の出力
    console.log(target);
    console.log(value);
    console.log(name);*/
  }

  handleSubmit(event) {
    console.log('input1（入力1）: ' + this.state.input1);
    console.log('input2（入力2）: ' + this.state.input2);
    //チェックボックスの値（target.checked → false または true）
    console.log('check1（チェック1）: ' + this.state.check1);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
        入力1
          <input name="input1" type="text" value={this.state.input1} onChange=
{this.handleChange} />
        </label>
        <br/>
        <label>
        入力2
          <input name="input2" type="text" value={this.state.input2} onChange=
{this.handleChange} />
        </label>
        <br/>
        <label>
          チェック1
          <input
```

```
            name="check1"
            type="checkbox"
            checked={this.state.check1}
            onChange={this.handleChange} />
        </label>
        <br/>
        <input type="submit" value="送信" />
      </form>
    );
  }
}

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

以下は上記26～29行目のコメントアウトを外した場合のコンソール出力のスクリーンショットです。



フォームを使った計算の例

以下はフォームを使った加算と減算の例です。

前述の例と同様、2つの <input type="text"> にそれぞれ name 属性（input1 と input2）を設定して、入力内容の変更に関しては1つのイベントハンドラ（handleChange）で処理しています。

入力された値が変更されると、handleChange で setState() を使って state を更新します。

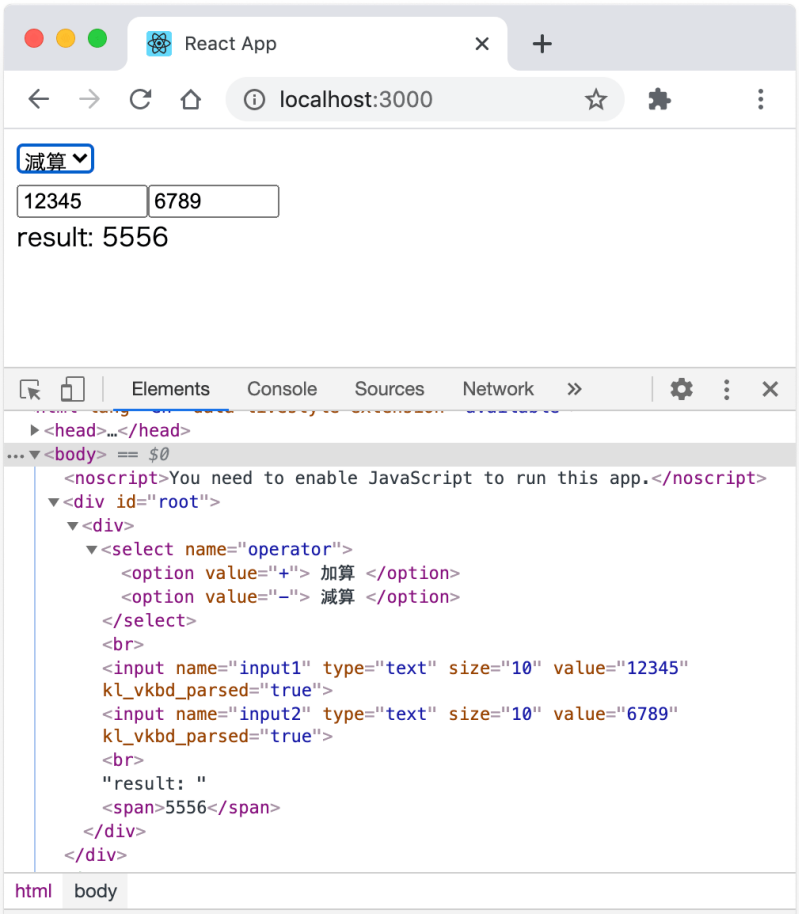
setState() が呼び出されて値が更新されると render() メソッドが実行されるので、その際に計算を実行して計算結果（result）を更新しています（30～40行目）。

加算と減算の切り替えは select 要素で選択します。

切り替えが発生すると handleChangeOperator が呼び出され setState() を使って state を更新し、render() メソッドが実行されて選択された演算子（+ か -）で計算が実行されてレンダリングされます。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyCalculatorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input1: 0,
      input2: 0,
      operator: '+'
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleChangeOperator = this.handleChangeOperator.bind(this);
  }

  handleChange(event) {
    this.setState({
      [event.target.name]: event.target.value
    });
  }

  //加算と減算の演算子を切り替えるハンドラ
  handleChangeOperator(event) {
    this.setState({
      operator:  event.target.value
    });
  }

  render() {
    //演算子（+ か -）
    const operator = this.state.operator;
    //計算結果の初期値
    let result = 0;

    //setState() で更新すると render()が呼び出されるので、その際に計算を実行して計算結果を更新
    if(operator === '+') {
       result = parseFloat(this.state.input1) +  parseFloat(this.state.input2);
    }else{
      result =  parseFloat(this.state.input1) -  parseFloat(this.state.input2);
    }

    return (
      <div>
        <select name="operator" value={this.state.operator} onChange={this.handleChangeOperator}>
          <option value="+"> 加算 </option>
          <option value="-"> 減算 </option>
        </select>
        <br/>
        <input name="input1" type="text" value={this.state.input1} onChange={this.handleChange} size="10" />
        <input name="input2" type="text" value={this.state.input2} onChange={this.handleChange}  size="10" />
        <br/>
        result: <span> {result} </span>
      </div>
```

```
    );
  }
}

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



非制御コンポーネント

制御されたコンポーネントの代替手段として、非制御コンポーネントがあります（React ではフォームの実装には制御されたコンポーネントの使用が推奨されています）。

制御されたコンポーネントではフォームのデータは React コンポーネントが扱いますが、非制御コンポーネントではフォームデータを DOM 自身が扱います。

非制御コンポーネントを記述するには、各 state の更新に対してイベントハンドラを書く代わりに、Ref を使用して DOM からフォームの値を取得します。

React：Ref と DOM

以下は非制御コンポーネントの例です。ボタンをクリックすると input 要素に入力された値をコンソールに出力します。

Ref を作成するには React.createRef()を使用します（9行目）。

作成された Ref は ref 属性を用いて React 要素に紐付けられます（22行目）。

Ref はコンポーネントの構築時にインスタンスプロパティに割り当てられるのでコンポーネントを通して参照が可能です。参照は Ref の current 属性でアクセスできます。

```
1.  import React from 'react';
2.  import ReactDOM from 'react-dom';
3.
4.  class MyForm extends React.Component {
5.     constructor(props) {
6.      super(props);
7.      this.handleSubmit = this.handleSubmit.bind(this);
8.      //this.input に React.createRef で作成した Ref を代入
9.      this.input = React.createRef();
10.   }
11.
12.   handleSubmit(event) {
13.     //参照は Ref（this.input）の current 属性でアクセス
14.     console.log('name: ' + this.input.current.value);
15.     event.preventDefault();
16.   }
17.
18.   render() {
19.     //ref 属性を用いて Ref（コンストラクタで作成した this.input）を React 要素に紐付
    け
20.     return (
21.       <div>
22.         <input type="text" ref={this.input} />
23.         <button onClick={this.handleSubmit}>Sign up</button>
24.       </div>
25.     );
26.   }
27. }
28.
29. ReactDOM.render(
30.   <MyForm />,
31.   document.getElementById('root')
32. );
```

デフォルト値

React のレンダーのライフサイクルでは、フォーム要素の value 属性は DOM の値を上書きします。

非制御コンポーネントで、React に初期値（デフォルト値）を指定させるが後続の更新処理には関与しないようにするには、defaultValue 属性を value の代わりに指定することができます。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
   constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    //this.input に React.createRef で作成した Ref を代入
    this.input = React.createRef();
  }

  handleSubmit(event) {
    //参照は Ref（this.input）の current 属性でアクセス
    console.log('Name : ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    // 初期値（デフォルト値）を defaultValue を使って指定
    return (
      <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
          defaultValue="Foo"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
    );
  }
}

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

以下は select タグで初期状態で選択されている項目を option 要素に Selected を指定する代わりに、defaultValue 属性を使って指定する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';

class MyForm extends React.Component {
   constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    //this.select_value に作成した Ref を代入
    this.select_value = React.createRef();
  }

  handleSubmit(event) {
    //参照は Ref（this.select_value）の current 属性でアクセス
    console.log('Selected : ' + this.select_value.current.value);
    event.preventDefault();
  }

  render() {
    // 初期状態で選択するには defaultValue を指定
    return (
      <div>
        <select ref={this.select_value} defaultValue="うさぎ">
            <option value="猫">Cat</option>
            <option value="犬">Dog</option>
            <option value="うさぎ">Rabit</option>
            <option value="魚">Fish</option>
          </select>
        <button onClick={this.handleSubmit}>決定</button>
      </div>
    );
  }
}

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

React：非制御コンポーネント

コンポジション

コンポジションは「構成」や「組み立て」などの意味がありますが、React でアプリを作成する際にコンポーネントをどのように構成するか（組み立てるか）を「React コンポーネントのコンポジション」などと呼びます。

Containment（子要素を知らないコンポーネント）

例えば、ウェブページのサイドバーなど汎用的な入れ物を表す部品では、どのような子要素が入るのかを知らないコンポーネントがあります。

そのような場合、props の特別なプロパティ children（子要素を渡すための専用の props）を使って受け取った子要素を出力することができます。

以下の MyBorder コンポーネントはクラスを指定した div 要素で枠線を表示しますが、その中にどのようなコンポーネントが入るかは関知しません。MyBorder コンポーネントでは、props.children を子要素として配置します。

```
function MyBorder(props) {
  return (
    <div className={'myBorder myBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

これにより他のコンポーネントから JSX をネストすることで任意の子要素を渡すことができます。

<MyBorder> JSX タグの内側のあらゆる要素は MyBorder に children という props として渡されます。MyBorder は の内側に {props.children} をレンダリングするので、渡された要素が出力されます。

以下の例では h1 要素と p 要素を MyBorder タグで囲むことでそれらの周りに枠線を表示します。

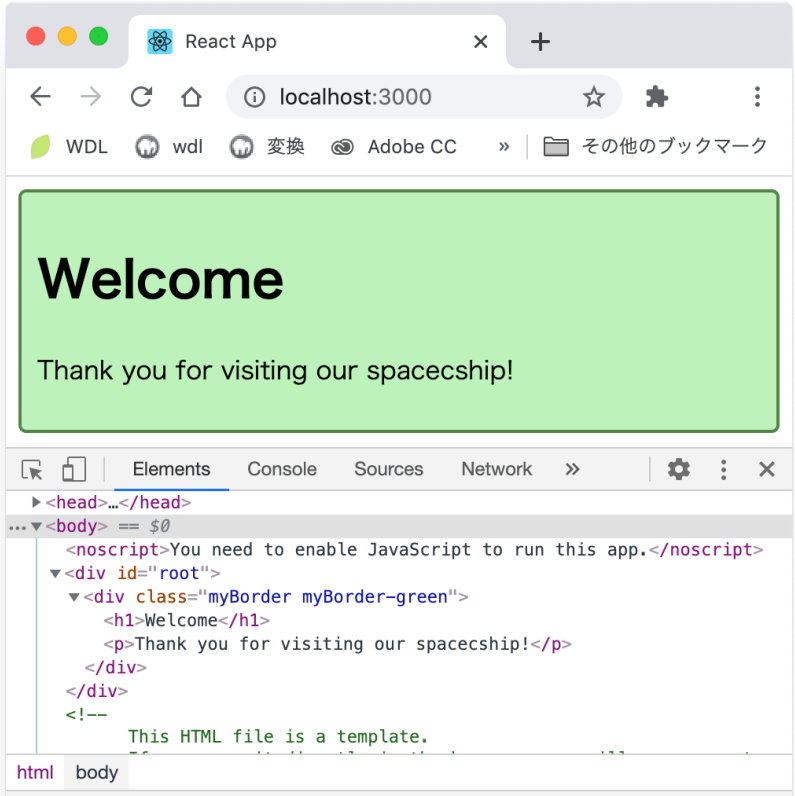
また、以下の場合、props を使って colorプロパティに green を渡しているので、myBorder と myBorder-green というクラスが適用されます。

```
import React from 'react';
import ReactDOM from 'react-dom';
import './MyStyle.css';

function MyBorder(props) {
  return (
    <div className={'myBorder myBorder-' + props.color}>
      {props.children}
    </div>
  );
}

function MyDialog() {
  return (
    <MyBorder color="green">
      <h1>Welcome</h1>
      <p>Thank you for visiting our spacecship!</p>
    </MyBorder>
  );
}

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

例えば、以下のようなクラスを使ったスタイル が設定されていれば、緑色の枠線と背景色が適用されます。

MyStyle.css

```
.myBorder {
  border: 2px solid #999;
  border-radius: 5px;
  padding: 10px;
}

.myBorder-green {
  border-color: #538547;
  background-color: #BDF2BA;
}
```



以下は複数の箇所に受け取った子要素を出力する例です。この場合、 children の props の代わりに独自の props を作成して渡します。

<Main /> や <Side /> のような React の要素はただのオブジェクトなので、他のデータと同様に props として渡すことができます。

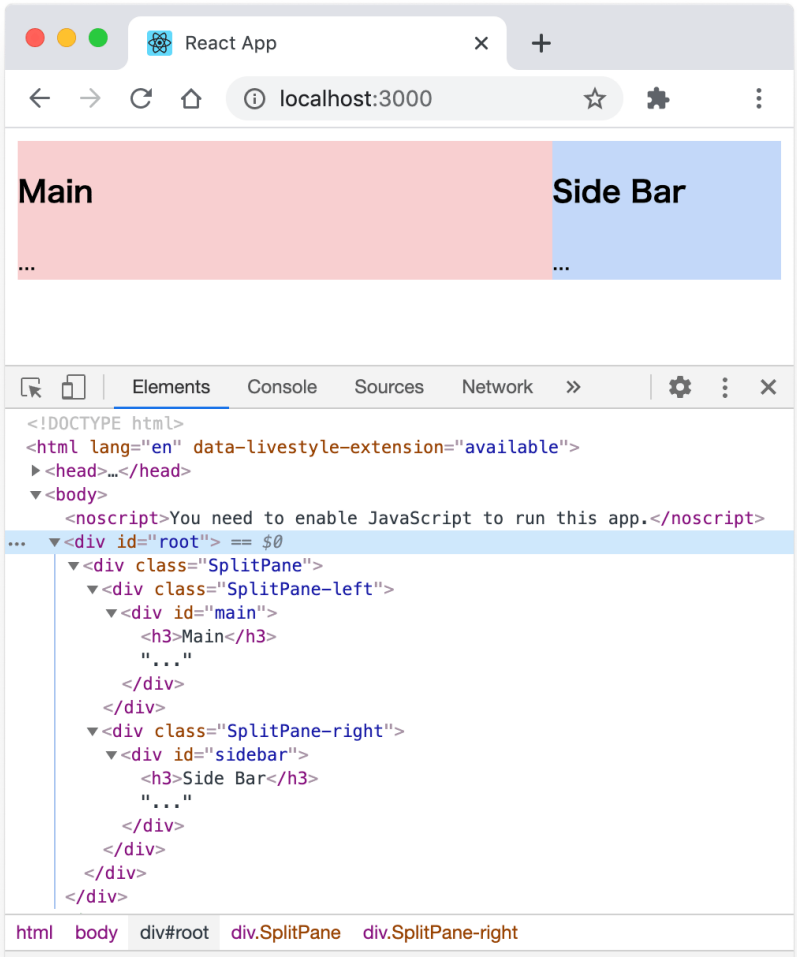
以下の場合、出力する子要素のコンポーネントを SplitPane の props で指定しています。

```
import React from 'react';
import ReactDOM from 'react-dom';
import './MyStyle.css';

function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function Main(props) {
  return (
    <div id="main">
      <h3>Main</h3>
      ...
    </div>
  )
}

function Side(props) {
  return (
    <div id="sidebar">
      <h3>Side Bar</h3>
      ...
    </div>
  )
}

// props（props.left と props.right）にコンポーネントを渡す
function App() {
  return (
    <SplitPane
      left={ <Main /> }
      right={ <Side /> }
    />
  );
}

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

以下のような CSS が設定されていれば、.SplitPane-left（Main）は左側に、.SplitPane-right（Side）は右側にフロートされて表示されます。

MyStyle.css

```css
.SplitPane-left {
  float: left;
  width: 70%;
  background-color: #F8CFD0;
}

.SplitPane-right {
  float: right;
  width: 30%;
  background-color: #C2D8F9;
}
```



Specialization (特化したコンポーネント)

構成が同じで内容が異なるコンポーネントの場合など、コンポーネントを他のコンポーネントの特別なケースとして考えることができます。

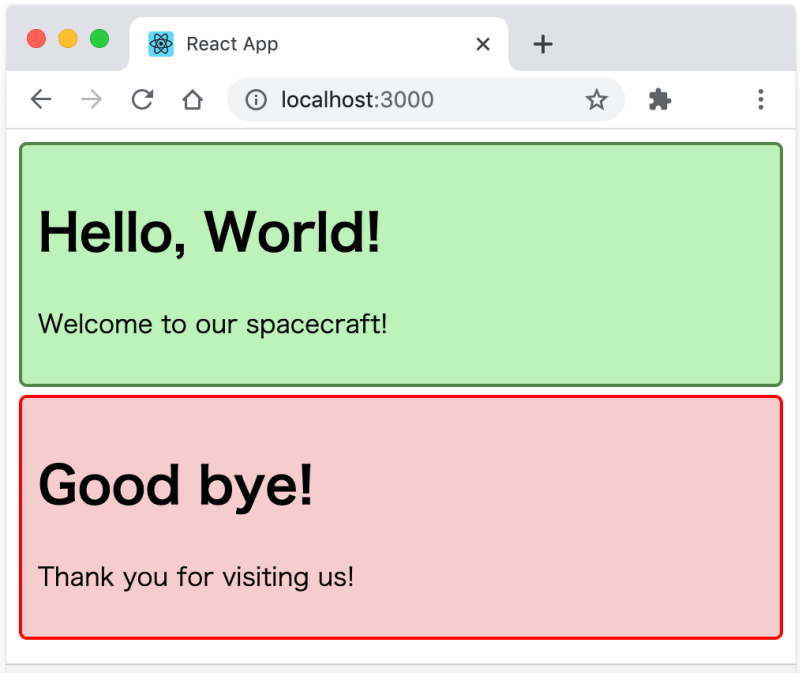
React では汎用的なコンポーネントに props を渡して設定することで、より特化したコンポーネントを作成することができます。

以下の HelloDialog と GoodByeDialog は Dialog の特別なケースとして考えることができます。

```
import React from 'react';
import ReactDOM from 'react-dom';
import './MyStyle.css';

function MyBorder(props) {
  return (
    <div className={'myBorder myBorder-' + props.color}>
      {props.children}
    </div>
  );
}

function Dialog(props) {
  return (
    <MyBorder color={props.dialogColor}>
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </MyBorder>
  );
}

//Dialog の特別なケース
function HelloDialog() {
  return (
    <Dialog
      dialogColor = "green"
      title="Hello, World!"
      message="Welcome to our spacecraft!" />
  );
}

//Dialog の特別なケース
function GoodByeDialog() {
  return (
    <Dialog
      dialogColor = "red"
      title="Good bye!"
      message="Thank you for visiting us!" />
  );
}

//サンプルとして両方のコンポーネントを表示しています
ReactDOM.render(
  (
    <div>
      <HelloDialog />
      <GoodByeDialog />
    </div>
  ),
  document.getElementById('root')
);
```

MyStyle.css

```css
.myBorder {
  border: 2px solid #999;
  border-radius: 5px;
  padding: 10px;
  margin: 5px 0;
}

.myBorder-green {
  border-color: #538547;
  background-color: #BDF2BA;
}

.myBorder-red {
  background-color: #F6CDCE;
  border-color: red;
}
```



React：コンポジション vs 継承

フラグメント

フラグメント (fragment) を使うとコンポーネントが複数の要素を返す際、DOM に余分なノードを追加することなく子要素をまとめることができます。

```
//エラーになる例
class MyComponent extends React.Component {
  render() {
    return (
      <h1>Hello World</h1>
      <p>Welcome to our spacecraft!</p>
    );
  }
}
/* 上記の場合、以下のような Parsing error になります。
Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? （隣接するJSX要素は、囲みタグでラップする必要があります。JSX フラグメント <>...</> が必要ですか？）*/

/* 以下のようにすればエラーにならないが div 要素が不要な場合もある */
class MyComponent extends React.Component {
  render() {
    //div 要素で囲み1つのまとまりにする
    return (
      <div>
        <h1>Hello World</h1>
        <p>Welcome to our spacecraft!</p>
      </div>
    );
  }
}
```

以下のようにフラグメントを使うと余分な div 要素を追加せずに子要素をまとめることができます。

```
class MyComponent extends React.Component {
  render() {
    //フラグメント <React.Fragment> を使ってまとめる
    return (
      <React.Fragment>
        <h1>Hello World</h1>
        <p>Welcome to our spacecraft!</p>
      </React.Fragment>
    );
  }
}
```

例えば、以下のような場合、子要素の Columns の td 要素をまとめるために div 要素を使うと table 要素の中に div 要素が入ってしまうため不正な HTML になってしまいますが、<React.Fragment> を使うことで解決できます。

```
import React from 'react';
import ReactDOM from 'react-dom';

class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}

class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}

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

/*出力される HTML
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
*/
```

短縮記法

フラグメントには短縮記法があり <React.Fragment> の代わりに <> を使って以下のように記述できます。

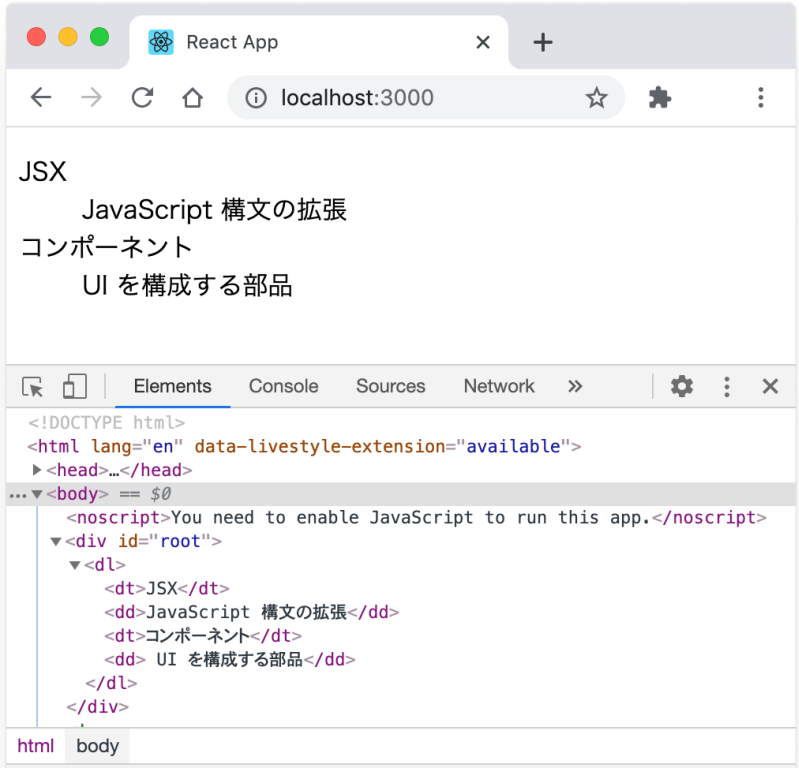
```
class MyComponent extends React.Component {
  render() {
    //フラグメントの短縮記法  <>...</>
    return (
      <>
        <h1>Hello World</h1>
        <p>Welcome to our spacecraft!</p>
      </>
    );
  }
}
```

key 付きフラグメント

<React.Fragment> では key を持つことができます（短縮記法ではサポートされていません）。

以下は定義リストを作成する時にフラグメントに key 属性を追加する例です。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyDataList(props) {
  return (
    <dl>
      {props.items.map(item => (
        //key はフラグメントに渡すことができる唯一の属性
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}

const data_list = [
  { id: '01', term: 'JSX', description: 'JavaScript 構文の拡張'},
  { id: '02', term: 'コンポーネント',description: ' UI を構成する部品'},
]

ReactDOM.render(
  <MyDataList items={data_list} />,
  document.getElementById('root')
);
```



現時点では key はフラグメントに渡すことができる唯一の属性です。

React：フラグメント

strict モード

strict モードはアプリの潜在的な問題を検出するために追加されたツールです。コンポーネントですが、フラグメント同じように UI としては画面に表示されません。

また、strict モードでの検査は開発モードでのみ動き、本番ビルドには影響を与えません。

<React.StrictMode>～</React.StrictMode>で囲まれた子孫要素に対して、付加的な検査を実施して警告を出すので以下のようなことを検出できます。

- 安全でないライフサイクルの特定
- レガシーな文字列 ref API の使用に対する警告
- 非推奨な findDOMNode の使用に対する警告
- 意図しない副作用の検出
- レガシーなコンテクスト API の検出

strict モードはアプリケーションの任意の箇所で有効にできます。

以下の例では、Header と Footer コンポーネントに対しては strict モードの検査はされませんが、ComponentOne、ComponentTwo およびそのすべての子孫要素に対しては検査が働きます。

```
import React from 'react';

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
          <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </div>
  );
}
```

React：strict モード

state のリフトアップ

複数の子要素からデータを集めたり、2つの子コンポーネントで互いにやりとりさせる場合は、親コンポーネント内で共有の state を宣言します。親コンポーネントは props を使うことで子に情報を渡すことができるので、、子コンポーネントが兄弟同士、あるいは親との間で常に同期されるようになります。

以下はクリックするとその回数を表示するボタンのコンポーネント CountButton です。

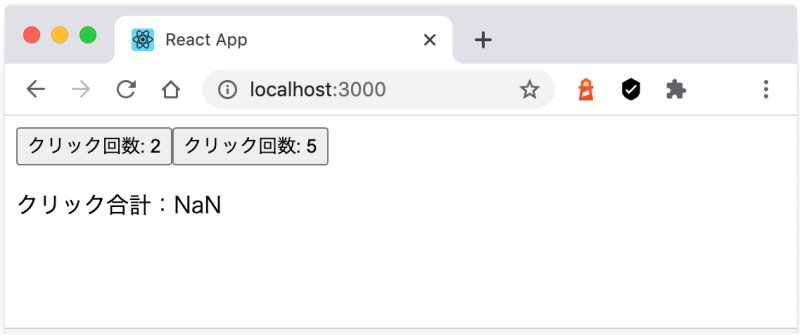
```
import React from 'react';
import ReactDOM from 'react-dom';

class CountButton extends React.Component {
  constructor() {
    super();
    // state プロパティの初期化（初期値の設定）
    this.state = {
      count: 0,
    };
    //クラスのメソッドはデフォルトではバインドされないのでバインドしておく
    this.updateCount = this.updateCount.bind(this);
  }

  //カウントを更新するクラスのメソッド
  updateCount() {
    this.setState((state) => {
      return { count: state.count + 1 }
    });
  }

  render() {
    //onClick を使って updateCount() を呼び出しイベント処理
    return (
      <button onClick={this.updateCount}>
        クリック回数: {this.state.count}
      </button>
     );
  }
}

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

上記のボタンを2つ表示して、クリックした数の合計を表示するコンポーネントを作成します。

2つのボタンに表示されるクリックした数の合計を表示するコンポーネント Sum とボタンと合計を表示するコンポーネント ClickedCounts を追加します。

```
import React from 'react';
import ReactDOM from 'react-dom';

//取り敢えずボタンのコンポーネントはそのまま（後で変更）
class CountButton extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
    this.updateCount = this.updateCount.bind(this);
  }

  updateCount() {
    this.setState((state) => {
      return { count: state.count + 1 }
    });
  }

  render() {
    return (
      <button onClick={this.updateCount}>
        クリック回数: {this.state.count}
      </button>
     );
  }
}

//追加した合計を表示するコンポーネント
function Sum(props) {
  return <p>クリック合計：{props.count1 + props.count2}</p>
}

//追加した全体を表示するコンポーネント
class ClickedCounts extends React.Component {
  render() {
    //ボタンを2つと合計を表示するコンポーネントをレンダリング
    return (
      <div>
        <CountButton />
        <CountButton />
        <Sum />
      </div>
     );
  }
}

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

上記のコードでは、それぞれのボタンをクリックするとクリック回数はカウントアップされますが、まだ合計は表示されません。



CountButton コンポーネントは独立してローカルの state を保持しているため ClickedCounts は、クリックされた回数を知りません。

ClickedCounts でボタンがクリックされた回数を知るためには、CountButton の state を親コンポーネント ClickedCounts に移動します（state のリフトアップ）。CountButton の state は削除します。

2つのボタンのクリック数は独立して変化するので、それぞれを count1、count2 として state で保持し、初期値を設定します（37～40行目）。

また、ボタンがクリックされた際のメソッドをそれぞれ作成して、setState() を使ってクリックされた回数を更新します（47～58行目）。

クリックされた際のメソッド及びクリック数は props 経由で CountButton へ渡します（65～72行目）

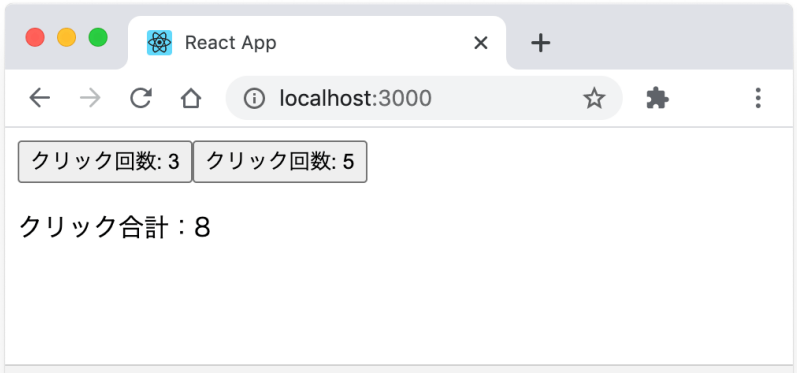
CountButton では props 経由で ClickedCounts からクリック数（props.count）とクリック数を更新するメソッド（props.onCountChange）を受け取り、クリック数を表示し、クリックされたら受け取ったメソッドを呼び出します。

ボタンがクリックされると以下のような流れになります。

- onClick={this.updateCount}により updateCount が呼び出される
- updateCount は props で受け取った onCountChange を実行
- onCountChange は ClickedCounts の handleCountChange1 または handleCountChange2 を呼び出して実行し state を更新しコンポーネントが再レンダリングされる
- 再レンダリングでは render() が実行されますが、その際に更新された値が CountButton と Sum コンポーネントに渡され、クリック回数と合計数が更新されます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//ボタンを表示するコンポーネント
class CountButton extends React.Component {
  constructor() {
    super();
    //ハンドラのバインド
    this.updateCount = this.updateCount.bind(this);
  }

  updateCount() {
    //props で親コンポーネントから onCountChange を受け取って実行
    this.props.onCountChange();
  }

  render() {
    //this.state.count を this.props.count に変更
    return (
      <button onClick={this.updateCount}>
        クリック回数: {this.props.count}
      </button>
     );
  }
}

//合計を表示するコンポーネント
function Sum(props) {
  return <p>クリック合計：{props.count1 + props.count2}</p>
}

//ボタンと合計を表示するコンポーネント
class ClickedCounts extends React.Component {
  constructor(props) {
    super(props);
    //初期値の設定
    this.state = {
      count1: 0,
      count2: 0,
    };
    //ハンドラのバインド
    this.handleCountChange1 = this.handleCountChange1.bind(this);
    this.handleCountChange2 = this.handleCountChange2.bind(this);
  }

  //1つ目のボタンがクリックされた際にクリックの回数 count1 を更新（1増加）
  handleCountChange1(count) {
    this.setState((state) => ({
      count1: state.count1 + 1
    }));
  }

  //2つ目のボタンがクリックされた際にクリックの回数 count2 を更新（1増加）
  handleCountChange2(count) {
    this.setState((state) => ({
      count2: state.count2 + 1
    }));
```

```
  }

  render() {
    const count1 = this.state.count1;
    const count2 = this.state.count2;
    return (
      <div>
        <CountButton
          onCountChange={this.handleCountChange1}
          count={count1}
        />
        <CountButton
          onCountChange={this.handleCountChange2}
          count={count1}
        />
        <Sum
          count1={count1}
          count2={count2}
        />
      </div>
    );
  }
}

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



CountButton は state を削除したので、以下のように関数コンポーネントに書き換えることができます。

```
function CountButton(props) {
  function updateCount() {
    props.onCountChange();
  }

  return (
    <button onClick={updateCount}>
      クリック回数: {props.count}
    </button>
   );
}
```

以下は React 公式サイト MAIN CONCEPTS の「state のリフトアップ」に掲載されているサンプルのコードをほぼそのまま使わせていただいています。

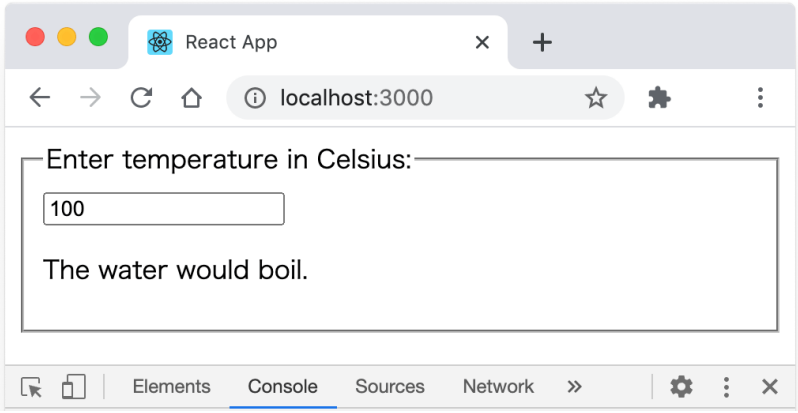
以下は「state のリフトアップ」の最初に掲載されているサンプルで、input 要素に入力された温度を意味する数値が100以上なら「The water would boil」と表示し、100未満なら「The water would not boil」と表示するコンポーネントです。

BoilingVerdict は props 経由で渡される値 celsius が100以上なら「The water would boil」、100未満なら「The water would not boil」と現在の入力値を判定するコンポーネントです。celsius は渡される前に parseFloat で数値に変換されています。

Calculator は input 要素に入力された値（温度）を state プロパティに保持して、入力された値が変更されると onChange イベントでハンドラ handleChange を呼び出し state プロパティを更新します（制御されたコンポーネント ）。

また、Calculator は現在の入力値を判定する BoilingVerdict もレンダーします。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

//現在の入力値を判定するコンポーネント
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

//入力された値によりメッセージ（判定）を表示するコンポーネント
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    //イベントハンドラ（クラスメソッド）をバインド
    this.handleChange = this.handleChange.bind(this);
    //state の初期化（初期値の設定）
    this.state = {temperature: ''};
  }

  //イベントハンドラ
  handleChange(e) {
    //ユーザがタイプする度に state を入力される値で更新
    this.setState({temperature: e.target.value});
  }

  render() {
    //state.temperature を変数に代入
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          //temperature は this.state.temperature
          value={temperature}
          //onChange イベントのハンドラを設定
          onChange={this.handleChange} />
        <BoilingVerdict
          //文字列を数値に変換
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}

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



2 つ目の入力を追加

摂氏の入力に加えて華氏の入力もできるようにして、それらを同期させるように変更します。

具体的には、摂氏（Celsius）の入力フィールドを更新したら、華氏（Fahrenheit）の入力フィールドも華氏に変換された温度で反映し、逆も同様にします。

まずは Calculator から温度を入力するフィールド（TemperatureInput コンポーネント）を抽出します。

props として c（摂氏） または f（華氏） の値をとる props.scale を新しく追加します（33行目）。

13～16行目は上記 scale の値により scaleNames[scale]（算出プロパティ ） で「Celsius」または「Fahrenheit」と表示するためのオブジェクトです。

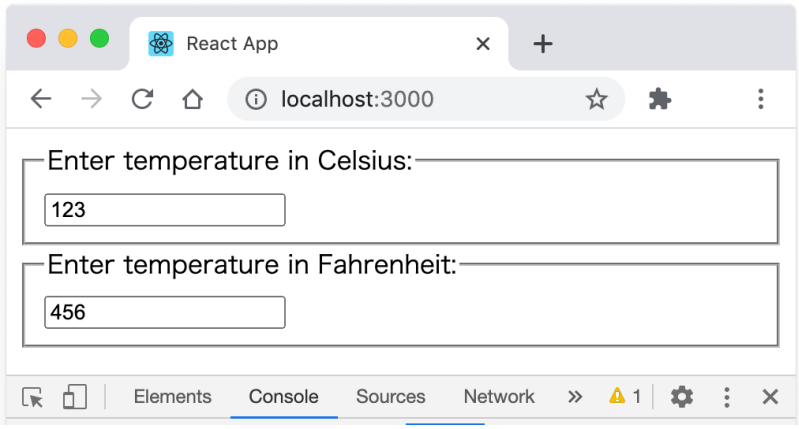
Calculator コンポーネントでは2つの温度を入力するフィールド（TemperatureInput コンポーネント）をレンダリングします。

それぞれのフィールドには props の scale を指定しています。この値により legend 要素の {scaleNames[scale]} 部分に「Celsius」または「Fahrenheit」と表示されます。

```
import React from 'react';
import ReactDOM from 'react-dom';

// この時点では使用していない
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

//追加
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

//温度の入力フィールドのコンポーネントを抽出
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    //摂氏または華氏を表し c または f の値をとる scale を追加
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}

class Calculator extends React.Component {
  //2つの TemperatureInput（入力フィールド）をレンダー
  render() {
    return (
      <div>
        <TemperatureInput
          //props.scale を設定
          scale="c"
        />
        <TemperatureInput
          //props.scale を設定
          scale="f"
        />
      </div>
```

```
    );
  }
}

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

この時点では2つの入力フィールドが表示されますが、温度を入力しても何も起こりません。Calculator は TemperatureInput の中の温度の値を知りませんし、摂氏から華氏（またその逆）に変換する関数もありません。



変換関数の作成

温度を変換する関数を作成します。

以下は華氏から摂氏に変換する関数と、摂氏から華氏に変換する関数です。

```
//華氏から摂氏に変換する関数
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

//摂氏から華氏に変換する関数
function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

以下は第1引数に温度を表す temperature（文字列）を、第2引数に上記で作成した変換関数を受け取り、変換した温度（文字列）を返す関数です。

```
function tryConvert(temperature, convert) {
  //文字列を数値に変換
  const input = parseFloat(temperature);
  //input が無効な値であれば空文字列を返す
  if (Number.isNaN(input)) {
    return '';
  }
  //第2引数に指定した変換関数 convert で摂氏または華氏に変換
  const output = convert(input);
  //小数第 3 位までで四捨五入
  const rounded = Math.round(output * 1000) / 1000;
  //文字列に変換して返す
  return rounded.toString();
}

/* 使用例 */
console.log(tryConvert('98.76', toFahrenheit));
//209.768 （98.76 を華氏に変換）

console.log(tryConvert('65.78', toCelsius));
//18.767 （65.78 を摂氏に変換）

console.log(tryConvert('foo', toCelsius));
// 空文字列 （無効な値）
```

state のリフトアップ

現時点では、両方の TemperatureInput コンポーネントは独立してローカルの state を保持しているため、2 つの入力フィールドはお互いに同期されていません。

React での state の共有は、state を最も近い共通の祖先コンポーネントに移動することによって実現します。これを「state のリフトアップ」と呼びます。

この例では TemperatureInput から state を削除して共通の祖先 Calculator に state を移動します。

Calculator が共有の state を保持すれば、それが両方の入力における現在の温度の「信頼できる情報源」となります。Calculator は props を使ってその情報を TemperatureInput に渡します。

props は親から子へとデータを渡すための手段です。

両方の TemperatureInput の props は同じ親コンポーネント Calculator から与えられるので、2 つの入力は常に同期されているようになります。

まず、TemperatureInput コンポーネントの this.state.temperature を this.props.temperature に置き換えます。this.props.temperature は親コンポーネント Calculator から渡します。

TemperatureInput コンポーネント

```
render() {
  //const temperature = this.state.temperature; を削除して
  //自身の state を Calculator から受け取る props に変更
  const temperature = this.props.temperature;
  ・・・
```

変更前までは TemperatureInput コンポーネントでは setState() を呼び出すことで temperature を更新していましたが、上記の変更で temperature は親コンポーネントから与えられる props.temperature なので TemperatureInput はそれを直接制御することができなくなっています。

temperature（state）の更新は親コンポーネント Calculator で setState() を呼び出して行われます。

そのため、TemperatureInput が自身の温度を更新するには、temperature を更新するハンドラを Calculator から props で受け取り、this.props.onTemperatureChange で呼び出すようにします。

onChange イベントのハンドラを以下のように変更します。

```
handleChange(e) {
  //this.setState({temperature: e.target.value}); を削除して以下に変更
  this.props.onTemperatureChange(e.target.value);
  //更新するには Calculator から props で渡される onTemperatureChange を呼び出す
}
```

そしてコンストラクタでの state の初期化処理を削除します。

上記の変更で TemperatureInput コンポーネントは以下のようになっています。

ローカルの state を削除し、this.state.temperature の代わりに this.props.temperature を読み取るように変更。

temperature を更新するには（入力値が変更されたら） this.setState() を呼び出す代わりに Calculator から与えられる onTemperatureChange を呼び出すように変更します（10行目）。

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    //this.state = {temperature: ''}; 削除（Calculator に移動）
  }

  handleChange(e) {
    //temperature を更新するには Calculator から渡される onTemperatureChange を呼び出す
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    //state から Calculator から受け取る props に変更
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Calculator コンポーネントでは、入力値の temperature と単位を表す scale を state に保存します。これらは入力コンポーネント TemperatureInput から「リフトアップ」したものです。

- temperature：入力された温度
- scale：摂氏または華氏を表す単位（c または f）

scale は2つのフィールドのどちらのフィールドに入力されているものかを識別する値でもあります。

state には最後に変更された値（temperature）とそれが示す単位（scale）を保存します。摂氏と華氏の両方の入力を保存することもできますが、単位がわかれば片方の値を基に変換することができます。

state を更新するハンドラは摂氏用と華氏用があり、TemperatureInput に props 経由で渡されて、値が更新されると呼び出されます（1つに統合することもできます ）。

ハンドラは temperature（入力された温度）を引数に取り、setState() で scale と temperature を更新します。scale は摂氏用か華氏用のどちらのフィールドが更新されたかを表す値です。

{scale: 'c', temperature} はプロパティの短縮構文です。

setState() で temperature が更新されると render() が呼び出され、更新された値を使って tryConvert() で表示する温度が更新されてレンダリングされます。

以下が変更後の Calculator コンポーネントです。

```
//temperature と scale を state に保存（TemperatureInput からリフトアップ）
class Calculator extends React.Component {
  // state を使うのでコンストラクタを追加
  constructor(props) {
    super(props);
    // ハンドラのバインド
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    // TemperatureInput から state をリフトアップ
    this.state = {
      //state の初期値の設定
      temperature: '',
      scale: 'c'
    };
  }

  //摂氏の入力が変更されたら props 経由で呼び出され state を更新するハンドラ
  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
    //this.setState({scale: 'c', temperature:temperature}) の短縮構文
  }

  //華氏の入力が変更されたら props 経由で呼び出さ state を更新するハンドラ
  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    // scale が f なら Celsius に変換
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
    // scale が c なら Fahrenheit に変換
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
temperature;

    return (
      <div>
        <TemperatureInput
          // scale を props に設定（c なので摂氏のフィールド）
          scale="c"
          // temperature（摂氏）を props に設定して渡す
          temperature={celsius}
          // 入力が変更された場合のハンドラを props に設定して渡す
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          // scale を props に設定（f なので華氏のフィールド）
          scale="f"
          // temperature（華氏）を props に設定して渡す
          temperature={fahrenheit}
          // 入力が変更された場合のハンドラを props に設定して渡す
          onTemperatureChange={this.handleFahrenheitChange} />

        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
```

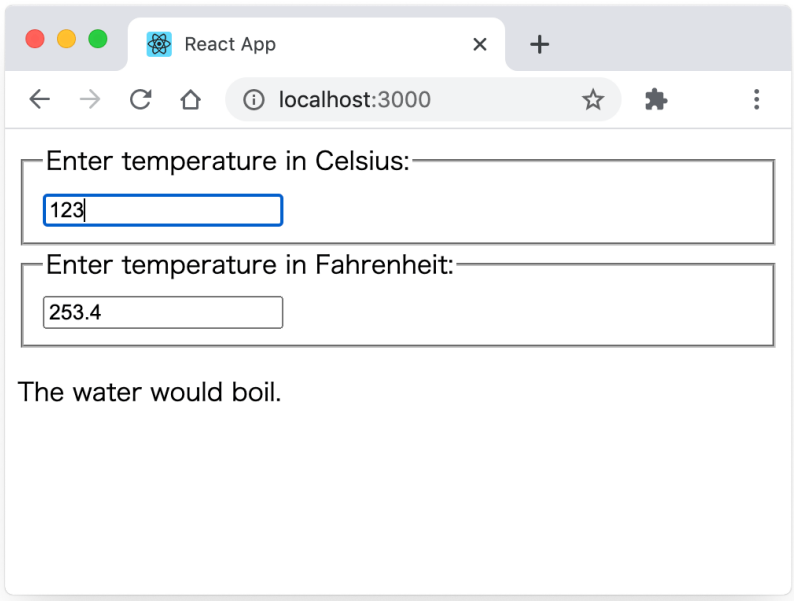
```
    );
  }
}
```

以下が変更後の全体です。

```
import React from 'react';
import ReactDOM from 'react-dom';

// 現在の入力値を判定するコンポーネント
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

//scaleNames[scale] で「Celsius」または「Fahrenheit」と表示
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

//入力フィールドのコンポーネント
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  //onChange イベントのハンドラ
  handleChange(e) {
    //更新するには Calculator から props で渡される onTemperatureChange を呼び出す
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    //temperature を参照するには Calculator から受け取る props を参照
    const temperature = this.props.temperature;
    // c または f の値をとり摂氏または華氏を表す scale（props で受け取る）
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}

//temperature と scale を state に保存（TemperatureInput からリフトアップ）
class Calculator extends React.Component {
  // state を使うのでコンストラクタを追加
  constructor(props) {
    super(props);
    // ハンドラのバインド
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    // TemperatureInput から state をリフトアップ
    this.state = {
      //state の初期値の設定
      temperature: '',
```

```
      scale: 'c'
    };
  }

  // 入力値（摂氏）を更新するハンドラ
  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  // 入力値（華氏）を更新するハンドラ
  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    // scale が f なら Celsius に変換
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
    // scale が c なら Fahrenheit に変換
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
temperature;

    return (
      <div>
        <TemperatureInput
          // scale を props に設定（c なので摂氏のフィールド）
          scale="c"
          // temperature（摂氏）を props に設定して渡す
          temperature={celsius}
          // 入力が変更された場合のハンドラを props に設定して渡す
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          // scale を props に設定（f なので華氏のフィールド）
          scale="f"
          // temperature（華氏）を props に設定して渡す
          temperature={fahrenheit}
          // 入力が変更された場合のハンドラを props に設定して渡す
          onTemperatureChange={this.handleFahrenheitChange} />

        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}

//華氏から摂氏に変換する関数
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

//摂氏から華氏に変換する関数
function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

```

```
function tryConvert(temperature, convert) {
  //文字列を数値に変換
  const input = parseFloat(temperature);
  //input が無効な値であれば空文字列を返す
  if (Number.isNaN(input)) {
    return '';
  }
  //第2引数に指定した変換関数 convert で摂氏または華氏に変換
  const output = convert(input);
  //小数第 3 位までで四捨五入
  const rounded = Math.round(output * 1000) / 1000;
  //文字列に変換して返す
  return rounded.toString();
}

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



TemperatureInput は state を持たないので関数コンポーネントに書き換えることで、よりシンプルに書くことができます。

以下は TemperatureInput を関数コンポーネントに書き換えて、ついでに入力値を更新する摂氏用と華氏用のハンドラを1つにまとめた例です。コンストラクタを削除して、this.props は、props に書き換える必要があります。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

//関数コンポーネントに書き換え
function TemperatureInput(props) {

  const handleChange = (e) => {
    // 更新された温度（入力値）と scale（単位）を引数に指定
    props.onTemperatureChange(e.target.value, props.scale);
  }

  const temperature = props.temperature;
  const scale = props.scale;
  return (
    <fieldset>
      <legend>Enter temperature in {scaleNames[scale]}:</legend>
      <input value={temperature}
             onChange={handleChange} />
    </fieldset>
  );
}

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    // 変更したハンドラのバインド
    this.handleTemperatureChange = this.handleTemperatureChange.bind(this);
    this.state = {
      temperature: '',
      scale: 'c'
    };
  }

  // 摂氏用と華氏用を統合したハンドラ
  handleTemperatureChange(temperature, scale) {
    //オブジェクトのキー名と値の変数名が同じなのでプロパティの短縮構文を利用
    this.setState({temperature, scale});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
```

```
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          // 変更したハンドラ
          onTemperatureChange={this.handleTemperatureChange} />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          // 変更したハンドラ
          onTemperatureChange={this.handleTemperatureChange} />

        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

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

以下は上記をコンポーネント Calculator のみで行うように書き換えたものです。

```
import React from 'react';
import ReactDOM from 'react-dom';

function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      temperature: '',
      scale: 'c'
    };
  }

  handleChange(e) {
    this.setState({
      temperature: e.target.value,
      scale:e.target.name
    });
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.state.scale;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
temperature;
    return (
      <>
        <fieldset>
          <legend>Enter temperature in Celsius:</legend>
          <input
            value={celsius}
            name='c'
            onChange={this.handleChange} />
        </fieldset>
        <fieldset>
        <legend>Enter temperature in Fahrenheit:</legend>
          <input
            value={fahrenheit}
            name='f'
            onChange={this.handleChange} />
        </fieldset>
        <BoilingVerdict
            celsius={parseFloat(celsius)} />
      </>
    );
  }
}

```

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

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

React：state のリフトアップ

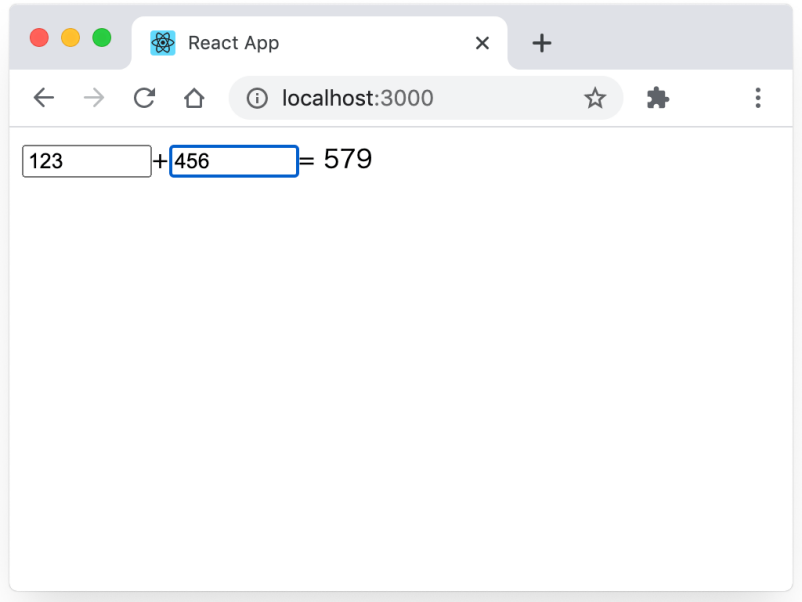
以下は足し算を行うコンポーネントです。

入力された値が変更されると onChange イベントでハンドラ handleChange が呼び出されます。

handleChange はイベントの発生した要素の値（state）を setState を使って更新します。

setState が呼び出されると render メソッドが実行されるので、その際に計算を実行して結果を更新します。（関連項目：複数の入力の処理 ）。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

class MyCalculatorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      //テキストフィールドに入力される値
      input1: 0,
      input2: 0,
    };
    this.handleChange = this.handleChange.bind(this);
  }

  //onChange イベントのハンドラ
  handleChange(event) {
    //イベントの発生した要素
    const target = event.target;
    //イベントの発生した要素の値
    const value = target.value;
    //イベントの発生した要素の name 属性の値
    const name = target.name;
    this.setState({
      // [name] は算出プロパティ
      [name]: value
    });
  }

  render() {
    //計算結果の初期値
    let result = 0;
    //setState() で更新すると render()が呼び出されるので、その際に計算を実行して計算結果を更新
    result = parseFloat(this.state.input1) +  parseFloat(this.state.input2);

    return (
      <div>
        <input name="input1" type="text" value={this.state.input1} onChange={this.handleChange} onKeyUp={this.handleKeyUp} size="10" />
      +
        <input name="input2" type="text" value={this.state.input2} onChange={this.handleChange} onKeyUp={this.handleKeyUp} size="10" />
      =
         <span> {result} </span>
      </div>
    );
  }
}

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



以下は上記のコンポーネントから入力部分（MyInput）を抽出して書き換えてものです。

入力値が変更されると、input 要素のイベントハンドラ handleChange が呼出され、props 経由でで親コンポーネントから受け取った onValueChange を実行します。

onValueChange は MyCalculatorForm コンポーネントの handleValueChange1 または handleValueChange2 を呼び出し（どちらの input 要素かにより決まります）、setState を使って state を更新します。

setState で state が更新されると、render メソッドが呼び出され、計算が実行されてレンダリングされます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//入力部分をコンポーネントに抽出
class MyInput extends React.Component {
  constructor(props) {
    super(props);
    //ローカルの state を削除
    this.handleChange = this.handleChange.bind(this);
    //this.handleKeyUp = this.handleKeyUp.bind(this);
  }

  handleChange(event) {
    //props で親コンポーネントから onValueChange を受け取って実行
    this.props.onValueChange(event.target.value);
  }

  render() {
    return (
      <input
        type="text"
        //props で親コンポーネントから受け取る
        value={this.props.value}
        onChange={this.handleChange}
        size={this.props.size}
      />
    )
  }
}

class MyCalculatorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input1: 0,
      input2: 0,
    };
    this.handleValueChange1 = this.handleValueChange1.bind(this);
    this.handleValueChange2 = this.handleValueChange2.bind(this);
  }

  handleValueChange1(value) {
    this.setState({input1: value});
  }

  handleValueChange2(value) {
    this.setState({input2: value});
  }

  render() {
    //計算結果の初期値
    let result = 0;
    //setState() で更新すると render()が呼び出されるので、その際に計算を実行して計算結果を更新
    result = parseFloat(this.state.input1) +  parseFloat(this.state.input2);

    return (
```

```
      <div>
        <MyInput value={this.state.input1} size="10" onValueChange=
{this.handleValueChange1} />
      +
        <MyInput value={this.state.input2} size="10" onValueChange=
{this.handleValueChange2} />
      =
         <span> {result} </span>
      </div>
    );
  }
}

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

コンテクスト Context

React では、配下のコンポーネントにデータを渡すための手段として props が提供されていますが、props の場合、親コンポーネントから子コンポーネントへ、さらに孫コンポーネントへと言うようにデータをバケツリレーのように渡していかなければなりません。

Context は props のバケツリレーを回避するための API です。Context API を利用すると props を使わずに下の階層のコンポーネントにデータを渡すことができます（コンポーネント間でデータを共有させることができます）。

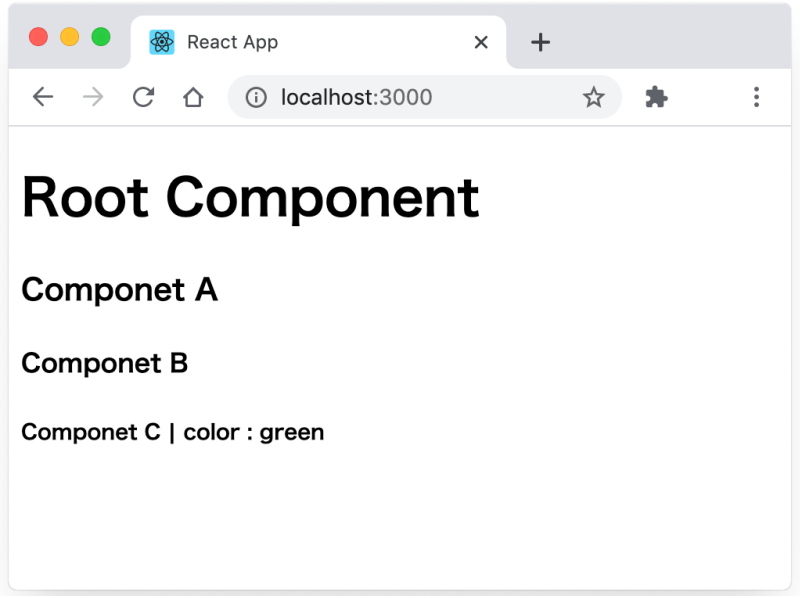
React：コンテクスト

関連ページ：useContext

例えば以下のような RootComponent を一番上の親コンポーネントとして、その子コンポーネントに ComponentA を、ComponentA の子に ComponentB、ComponentB の子に ComponentC を持つ4階層のコンポーネントがある場合、RootComponent の props を ComponentC に渡すには ComponentA 及び ComponentB 経由でバケツリレーのように ComponentC まで伝播させなければなりません。

```
import React from 'react';
import ReactDOM from 'react-dom';

const RootComponent = (props) => {
  //ComponentA に props を渡す
  return (
    <div>
      <h1>Root Component</h1>
      <ComponentA color={props.color} />
    </div>
  )
}

const ComponentA = (props) => {
  //ComponentB に props を渡す
  return (
    <div>
      <h3>Componet A</h3>
      <ComponentB color={props.color} />
    </div>
  )
}

const ComponentB = (props) => {
  //ComponentC に props を渡す
  return (
    <div>
      <h4>Componet B</h4>
      <ComponentC color={props.color} />
    </div>
  )
}

const ComponentC = (props) => {
  //ComponentA → ComponentB 経由で RootComponent から props を受け取る
  return (
    <div>
      <h5>Componet C | color : {props.color}</h5>
    </div>
  )
}

ReactDOM.render(
  <RootComponent color="green"/>,
  document.getElementById('root')
)
```

上記のコードは以下のように表示されます。props.color が RootComponent からバケツリレーされ、ComponentC で表示されます。



以下は Context を使ってデータを RootComponent から ComponentC に渡す（RootComponent と ComponentC でデータを共有する）例です。

ComponentA と ComponentB を経由することなく、データを RootComponent から ComponentC に渡すことができます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//コンポーネントの外で Context オブジェクトを作成
const ColorContext = React.createContext();

const RootComponent = (props) => {
  //Provider コンポーネントで適用範囲を囲み value プロパティに渡したい値を設定
  return (
    <div>
      <h1>Root Component</h1>
      <ColorContext.Provider value={props.color}>
        <ComponentA />
      </ColorContext.Provider>
    </div>
  )
}

const ComponentA = (props) => {
  //props の受け渡しはない
  return (
    <div>
      <h3>Componet A</h3>
      <ComponentB />
    </div>
  )
}

const ComponentB = (props) => {
  //props の受け渡しはない
  return (
    <div>
      <h4>Componet B</h4>
      <ComponentC />
    </div>
  )
}

const ComponentC = (props) => {
  //Consumer コンポーネントで値を受け取る
  return (
    <ColorContext.Consumer>
      {(value) => (
        <div>
          <h5>Componet C | color : {value}</h5>
        </div>
      )}
    </ColorContext.Consumer>
  )
}

ReactDOM.render(
  <RootComponent color="green"/>,
  document.getElementById('root')
)
```

以下は Context API（コンテクスト）を利用するおおまかな流れです。

1. Context オブジェクトを作成
2. Provider コンポーネントに渡したい値を設定
3. Consumer コンポーネントで値を受け取る

Context オブジェクトを作成

コンテクストを利用するには、コンポーネントの外で React.createContext() を使ってコンテクスト（Context）オブジェクトを作成します。

引数にはデフォルト値を設定できます。デフォルト値（defaultValue）は、コンポーネントがツリー内の上位に一致するプロバイダを持っていない場合のみ使用されます。

```
1. const コンテクスト = React.createContext(defaultValue);
```

以下は ColorContext というコンテクストオブジェクトを作成する例です。

```
1. const ColorContext = React.createContext();
```

生成されたコンテクストオブジェクトには Provider と Consumer というオブジェクト（コンポーネント）が含まれています。

Provider コンポーネントに渡したい値を設定

作成したコンテクストオブジェクトには、関連付けされた Provider コンポーネントと Consumer コンポーネントが付属しています。

渡したい値を Provider コンポーネントの value プロパティに設定すると、その値が子孫である Consumer コンポーネントに渡されます。

Provider にはコンテクスト（Context）の適用範囲を決める役割もあります。

また、Provider コンポーネントでは value が変更されると配下のコンポーネントを再レンダーします。

以下は、作成した ColorContext の Provider コンポーネント ColorContext.Provider の value プロパティに、渡したい値 {props.color} を設定しています。

そして ComponentA を囲んで Context の適用範囲をコンポーネントツリーにおける ComponentA 以下に設定しています（9～11行目）。

```
//コンポーネントの外で Context オブジェクトを作成
const ColorContext = React.createContext();

const RootComponent = (props) => {
  //Provider コンポーネントで適用範囲を囲み value プロパティに渡したい値を設定
  return (
    <div>
      <h1>Root</h1>
      <ColorContext.Provider value={props.color}>
        <ComponentA />
      </ColorContext.Provider>
    </div>
  )
}
```

Consumer コンポーネントで値を受け取る

Provider コンポーネントの value プロパティにセットしたデータは、Consumer コンポーネントで受け取る（subscribe・購読する）ことができます。

Consumer コンポーネントの内部は関数を記述します。この関数の引数で Provider コンポーネントから渡されたデータを受け取り、React ノードを返します。

以下のようにアロー関数の引数として Provider コンポーネント側で value プロパティにセットしたデータが渡されるので、その値に基づいて何かをレンダーします。

```
const ComponentC = (props) => {
  //Consumer コンポーネントの内部は関数を記述
  return (
    <ColorContext.Consumer>
      {(value) => (
        <div>
          <h5>Componet C | color : {value}</h5>
        </div>
      )}
    </ColorContext.Consumer>
  )
}
```

以下のように何らかの処理を記述することもできます。

```
const ComponentC = (props) => {
  return (
    <ColorContext.Consumer>
      {(value) => {
       //何らかの処理
       return (
          <div>
            <h5>Componet C | color : {value}</h5>
          </div>
        )}}
    </ColorContext.Consumer>
  )
}
```

Class.contextType

クラスの contextType プロパティには React.createContext() により作成された Context オブジェクトを指定することができます。

これにより、this.context を使ってそのコンテクストタイプの最も近い現在値を利用できます（Consumer コンポーネントは不要）。

contextType はクラスのプロパティなので、クラスコンポーネントで使うことができます。

```
import React from 'react';
import ReactDOM from 'react-dom';

//コンポーネントの外で Context オブジェクト ColorContext を作成
const ColorContext = React.createContext();

const RootComponent = (props) => {
  //Provider コンポーネントで適用範囲を囲み value プロパティに渡したい値を設定
  return (
    <div>
      <h1>Root</h1>
      <ColorContext.Provider value={props.color}>
        <ComponentA />
      </ColorContext.Provider>
    </div>
  )
}

const ComponentA = (props) => {
  return (
    <div>
      <h3>Componet A</h3>
      <ComponentB />
    </div>
  )
}

const ComponentB = (props) => {
  return (
    <div>
      <h4>Componet B</h4>
      <ComponentC />
    </div>
  )
}

//クラスコンポーネントに変更
class ComponentC extends React.Component{
  render() {
    //this.context を使ってそのコンテクストタイプの最も近い現在値を利用できる
    let value = this.context;
    return (
      <div>
        <h5>Componet C | color : {value}</h5>
      </div>
    )
  }
}
// ComponentC の contextType プロパティに Context オブジェクトを指定
ComponentC.contextType = ColorContext;

ReactDOM.render(
  <RootComponent color="green"/>,
  document.getElementById('root')
)
```

実験的な public class fields syntax を使用している場合は、static クラスフィールドを使用することで contextType を初期化することができます。

```
class ComponentC extends React.Component{
  //ComponentC.contextType = ColorContext; の代わりに static クラスフィールドを使用
  static contextType = ColorContext;
  render() {
    let value = this.context;
    return (
      <div>
        <h5>Componet C | color : {value}</h5>
      </div>
    )
  }
}
```

Context で state を使う

Context は state と組み合わせて使うことがよくあります。

以下の例では App コンポーネントを一番上の親として、その子コンポーネントに ComponentA、ComponentA の子コンポーネントに ComponentB があります。

App コンポーネントは count という state を持ち、この値を ComponentB で利用するには props を使う場合、以下のように ComponentA 経由で props を伝播させる必要があります。

```
import React from 'react';
import ReactDOM from 'react-dom';

//一番上の親コンポーネント
class App extends React.Component {
  constructor(props) {
    super(props);
    // state プロパティの初期化（初期値の設定）
    this.state = {
      count: 0,
    };
  }

  render() {
    //count を ComponentA に props で渡す
    return (
      <div>
        <ComponentA count={this.state.count}/>
      </div>
    );
  }
}

//App の子コンポーネント
const ComponentA = (props) => {
  //count を ComponentB に props で渡す
  return (
    <div>
      <ComponentB count={props.count}/>
    </div>
  )
}

//ComponentA の子コンポーネント
const ComponentB = (props) => {
  //props を受け取る
  return (
    <div>
      <div> Count: {props.count}</div>
    </div>
  )
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
)
//Count: 0 と表示される
```

Context を使うと以下のように App コンポーネントの state を ComponentA を経由せずに ComponentB に渡すことができます。

```jsx
import React from 'react';
import ReactDOM from 'react-dom';

//Context を作成
const CountContext = React.createContext();

class App extends React.Component {
  constructor(props) {
    super(props);
    // state プロパティの初期化（初期値の設定）
    this.state = {
      count: 0,
    };
  }

  render() {
    //CountContext の Provider コンポーネントで渡したい値を設定
    return (
      <div>
        <CountContext.Provider value={this.state.count}>
          <ComponentA />
        </CountContext.Provider>
      </div>
    )
  }
}

//何もしない中間のコンポーネント
const ComponentA = (props) => {
  return (
    <div>
      <ComponentB />
    </div>
  )
}

const ComponentB = (props) => {
  //CountContext の Consumer コンポーネントで値を受け取る
  return (
    <CountContext.Consumer>
      {(value) => (
        <div>
          <div> Count: {value}</div>
        </div>
      )}
    </CountContext.Consumer>
  )
}

ReactDOM.render(
  <App />,
  document.getElementById('root')
)
//前述の例と同様に Count: 0 と表示される
```

ネストしたコンポーネントからコンテクストを更新

コンテクストを通して下に関数を渡すことで、Consumer コンポーネント側でコンテクストを更新することができます。

以下は、前述の ComponentB に App コンポーネントの state の count を増減するボタンを配置してコンテクストを更新（count を増減）する例です。

まず、App コンポーネントに count の値を増減する2つのメソッドを定義します（16～28行目）。

そして定義した2つのメソッドを state に追加（11～12行目）し、Provider コンポーネントの value に this.state（state 全体）を設定してコンテクストに渡します（35行目）。

```
class App extends React.Component {
  constructor(props) {
    super(props);
    //メソッドのバインド
    this.incrementCount = this.incrementCount.bind(this);
    this.decrementCount = this.decrementCount.bind(this);

    this.state = {
      count: 0,
      //count の値を更新するメソッドを追
      increment: this.incrementCount,
      decrement: this.decrementCount
    };
  }

  //count の値を増やすメソッド
  incrementCount() {
    this.setState((state) => {
      return { count: state.count + 1 }
    });
  }

  //count の値を減らすメソッド
  decrementCount() {
    this.setState((state) => {
      return { count: state.count - 1 }
    });
  }

  render() {
    //Provider コンポーネントで value に this.state を設定
    //state は全てプロバイダへ渡される
    return (
      <div>
        <CountContext.Provider value={this.state}>
          <ComponentA />
        </CountContext.Provider>
      </div>
    )
  }
}
```

続いて ComponentB の Consumer コンポーネントに count の値をクリックして更新する2つの button 要素を追加します。

Consumer コンポーネントの内部の関数の引数では、state の値を受け取り、onClick イベントハンドラに App コンポーネントで定義したメソッドを指定します。

count の値を表示する div 要素は Count: {count} に変更します。

```
const ComponentB = (props) => {
  //Consumer コンポーネントで値を受け取る
  return (
    <CountContext.Consumer>
      {({count, increment, decrement}) => (
        <div>
          <div>Count: {count}</div>
          <button onClick={increment}>Increment: +1</button>
          <button onClick={decrement}>Decrement: -1</button>
        </div>
      )}
    </CountContext.Consumer>
  )
}
```

上記の例では Consumer コンポーネントの内部の関数の引数で、state の値をそれぞれ受け取りましたが、以下のようにオブジェクトで受け取ることもできます。

```
const ComponentB = (props) => {
  //Consumer コンポーネントで state オブジェクトを value に受け取る
  return (
    <CountContext.Consumer>
      {(value) => (
        <div>
          <div>Count: {value.count}</div>
          <button onClick={value.increment}>Increment: +1</button>
          <button onClick={value.decrement}>Decrement: -1</button>
        </div>
      )}
    </CountContext.Consumer>
  )
}
```

以下が全体です。アプリケーションを作成する場合、通常はコンポーネントごとにファイルを分けて作成するので以下もそのようにしています。

src/index.js（App を表示するエントリーポイント）

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

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

コンテクストも1つのファイルに記述して App.js と ComponentB.js でインポートしています。

src/count-context.js

```
import React from 'react';

//Context を作成してエクスポート
const CountContext = React.createContext();
export default CountContext;
```

src/App.js

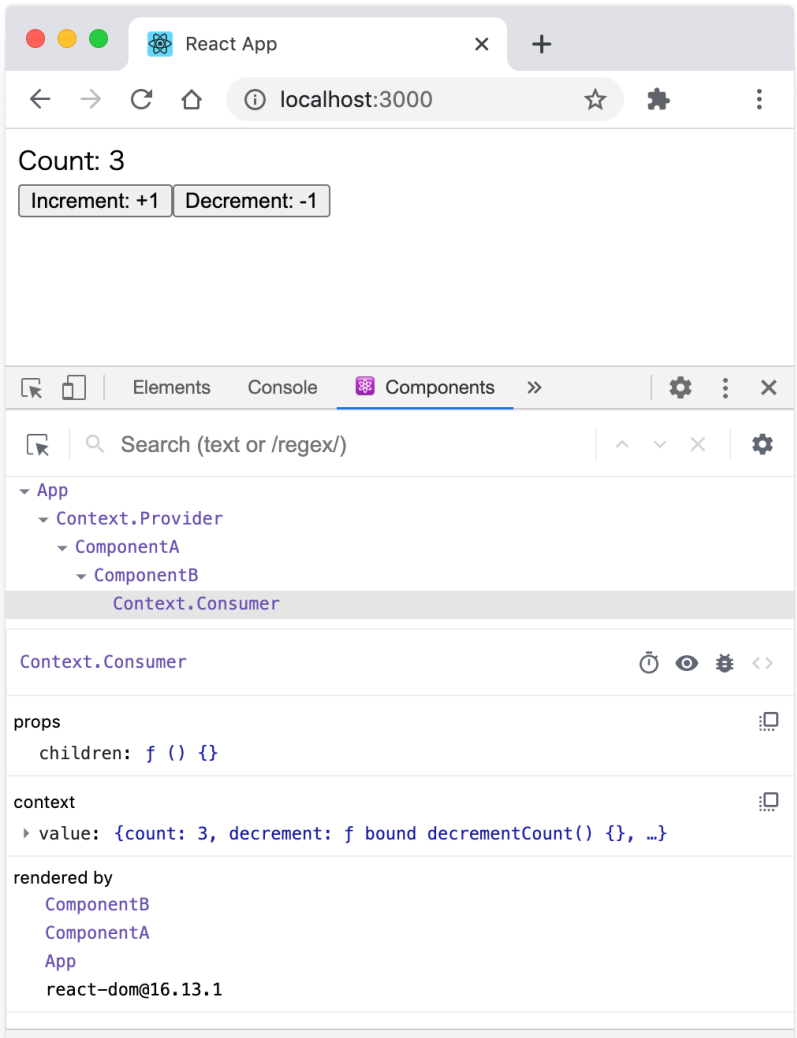
```
import React from 'react';
import CountContext from './count-context'
import ComponentA from './ComponentA';

export default class App extends React.Component {
  constructor(props) {
    super(props);
    //メソッドのバインド
    this.incrementCount = this.incrementCount.bind(this);
    this.decrementCount = this.decrementCount.bind(this);

    this.state = {
      count: 0,
      //count の値を更新するメソッド
      increment: this.incrementCount,
      decrement: this.decrementCount
    };
  }

  //count の値を増やすメソッド
  incrementCount() {
    this.setState((state) => {
      return { count: state.count + 1 }
    });
  }

  //count の値を減らすメソッド
  decrementCount() {
    this.setState((state) => {
      return { count: state.count - 1 }
    });
  }

  render() {
    //Provider コンポーネントで value に this.state を設定（state は全てプロバイダへ渡される）
    return (
      <div>
        <CountContext.Provider value={this.state}>
          <ComponentA />
        </CountContext.Provider>
      </div>
    )
  }
}
```

src/ComponentA.js

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

//何もしない中間のコンポーネント
const ComponentA = (props) => {
  return (
    <div>
      <ComponentB />
    </div>
  )
}

export default ComponentA;
```

src/ComponentB.js

```
import React from 'react';
import CountContext from './count-context'

const ComponentB = (props) => {
  //Consumer コンポーネントで state を受け取る
  return (
    <CountContext.Consumer>
      {(value) => (
        <div>
          <div>Count: {value.count}</div>
          <button onClick={value.increment}>Increment: +1</button>
          <button onClick={value.decrement}>Decrement: -1</button>
        </div>
      )}
    </CountContext.Consumer>
  )
}

export default ComponentB;
```

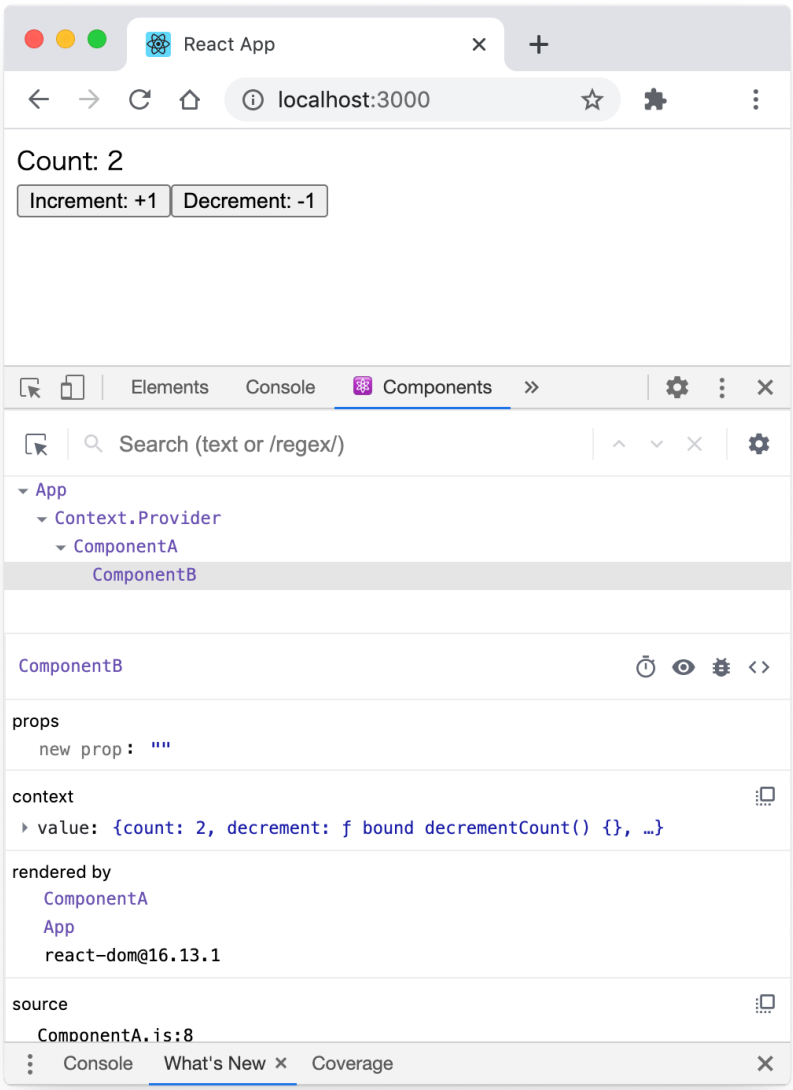


contextType を使う

以下は ComponentB をクラスコンポーネントに書き換えて Class.contextType を使う例です。

src/ComponentB.js

```jsx
import React from 'react';
import CountContext from './count-context'

//クラスコンポーネントに変更
class ComponentB extends React.Component{
  render() {
    //this.context を使ってそのコンテクストタイプの現在値を利用
    let value = this.context;
    return (
      <div>
        <div>Count: {value.count}</div>
        <button onClick={value.increment}>Increment: +1</button>
        <button onClick={value.decrement}>Decrement: -1</button>
      </div>
    )
  }
}
// ComponentB の contextType プロパティに Context オブジェクト CountContext を指定
ComponentB.contextType = CountContext;

export default ComponentB;
```



JSX を深く理解する

React：JSX を深く理解する

React 要素の型

JSX タグの先頭の部分は、React 要素の型を表しています。

例えば、以下の MyButton のような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るため、以下の場合であれば、MyButton がスコープになければなりません。

```
1. //大文字で始まる型
2. <MyButton color="blue" shadowSize={2}>
3.   Click Me
4. </MyButton>
```

スコープにあること

JSX は React.createElement の呼び出しへとコンパイルされるため（React の createElement メソッドが使われるため）、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。

言い換えると JSX を使う場合は、React ライブラリがインポートされている必要があります。

以下の場合、React ライブラリと CustomButton（React コンポーネント）の両方ともがインポートされている必要があります。

```
1. import React from 'react';  //React（ライブラリ）のインポート
2. import CustomButton from './CustomButton';  //CustomButton のインポート
3.
4. function WarningButton() {
5.   return <CustomButton color="red" />;
6.   // 以下と同じこと（等価）
7.   // return React.createElement(CustomButton, {color: 'red'}, null);
8. }
```

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るのでインポートする必要はありません。

JSX 型にドット記法を使用する

JSX では JSX 型に、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。

例えば、以下のような MyComponents とそのプロパティとしてコンポーネントが定義されている場合、

MyComponents.js

```
import React from 'react';

const MyComponents = {
  //MyButton コンポーネント
  MyButton: function MyButton(props) {
    return <button  color={props.color}>My Button</button>;
  },
  //MyTitle コンポーネント
  MyTitle: function MyTitle(props) {
    return <h1>{props.title}</h1>;
  }
}

export default MyComponents;
```

以下のようにドット記法を使って JSX 内から利用することができます。

App.js

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

function BlueButton() {
  //ドット記法を使って React コンポーネントを参照
  return <MyComponents.MyButton color="blue" />;
}

function HelloTitle() {
  //ドット記法を使って React コンポーネントを参照
  return <MyComponents.MyTitle title="Hello" />;
}

function App() {
  return (
    <>
      <HelloTitle />
      <BlueButton />
    </>
  );
}

export default App;
```

ユーザ定義のコンポーネントの名前は大文字で始める

ある要素の型が小文字から始まっている場合、それは や のような組み込みのコンポーネントを参照していて、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。

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

上記は以下のようにコンパイルされます。

```
React.createElement(
  'div',  //文字列に変換されて渡される
  {className: 'sidebar'}
)
```

<MyButton /> のように大文字で始まる型は React.createElement(MyButton) 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。

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

上記は以下のようにコンパイルされます。

```
React.createElement(
  MyButton,  //コンポーネントを参照
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

実行時に型を選択

プロパティ（props）の値に応じて異なるコンポーネントを表示し分けしたい場合がありますが、式を React の要素（JSX）の型として使用することはできないので、以下のような使い方はできません。

```
import React from 'react';
import {BlueButton, ClickButton} from './Buttons';

const buttons = {
  blue: BlueButton,
  click: ClickButton
};

//誤り
function Button(props) {
  //間違った使い方（エラーになりコンパイルされない）
  return <buttons[props.buttonType] value={props.value} />;
}
```

式を使って要素の型を示すには式を大文字から始まる変数に代入してその変数を JSX の型に指定します。

```
import React from 'react';
import {BlueButton, ClickButton} from './Buttons';

const buttons = {
  blue: BlueButton,
  click: ClickButton
};

function Button(props) {
  //式を大文字から始まる変数に代入
  const SpecificButton = buttons[props.buttonType];
  //変数を JSX の型に指定
  return <SpecificButton value={props.value} />;
}

function App() {
  return (
    <Button buttonType="blue" value="Blue Type Button"/>
  );
}

export default App;
```

Buttons.js

```
import React from 'react';

function MyButton(props) {
  return <button color={props.color} buttonType={props.buttonType} >{props.value}
</button>;
}

function BlueButton(props) {
  return <MyButton color="blue" value={props.value} buttonType={props.buttonType}
/>;
}

function ClickButton(props) {
  return <MyButton color={props.color} value="Click" buttonType=
{props.buttonType} />;
}

export { MyButton, BlueButton, ClickButton };
```

JSX における props

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が、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。

if 文や for 文を使う場合は、JSX の近くで間接的に利用します。

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>偶数</strong>;
  } else {
    description = <i>奇数</i>;
  }
  return <div>{props.number} は {description} です。</div>;
}
```

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。以下の JSX の式は等しいものです。

```
//文字列リテラルを渡す
<MyComponent message="hello world" />

//文字列を式として渡す
<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。

```
//以下の4つ JSX の式は等しいもので message は「<3」になります
<MyComponent message="&lt;3" />  //文字列リテラルのエスケープは復元される
<MyComponent message='&lt;3' />  //文字列リテラルのエスケープは復元される
<MyComponent message={"<3"} />
<MyComponent message={'<3'} />

//以下の message は「&lt;3」になります
<MyComponent message={"&lt;3"} />
<MyComponent message={'&lt;3'} />
```

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 となるので以下は等しいものとなります。

但し、ES6 におけるオブジェクトの簡略表記（プロパティの短縮構文）では、{foo} は {foo: true} ではなく {foo: foo} を意味するため、値を省略する記法は混乱を招く可能性があり推奨されていません。

```
//値を省略（推奨されない）
<MyTextBox autocomplete />

//以下のように記述したほうが良い
<MyTextBox autocomplete={true} />
```

属性の展開

props オブジェクトがあらかじめ存在していて、それを JSX に渡す場合はスプレッド構文（ ... ）を使用することで、props オブジェクトそのものを渡すことができます。

以下の App1 と App2 の JSX は等しいものです。

```
function Greeting(props) {
  return <h1>Hello, {props.firstName} {props.lastName} !</h1>;
}

function App1() {
  return <Greeting firstName="Jimi" lastName="Hendrix" />;
}

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

以下はスプレッド構文で props オブジェクトそのもの（全ての props）を渡す例です。

以下の場合レンダリングされる際に、MyButton が呼び出されて props として {className: "linkButton", children: "Click"} が渡されます。

```
import React from 'react';
import ReactDOM from 'react-dom';

function MyButton(props) {
  //スプレッド構文で全ての props を渡す
  return  <button { ...props } />;
};

ReactDOM.render(
  (
    <MyButton className="linkButton">
      Click
    </MyButton>
  ),
  document.getElementById('root')
);
```

上記の場合クラス属性と子要素 が渡されて以下のようにレンダリングされます。

```
<button class="linkButton">Click</button>
```

以下のように分割代入を使いコンポーネントが利用する特定のプロパティを取り出して、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>
  );
};
```

上記の場合、props として {kind: "primary", children: "Hello World!", onClick: ƒ} を受け取りますが、分割代入とスプレッド演算子を利用して、className には props.kind の値により PrimaryButton または SecondaryButton を設定しています。そして残りの props を ...other オブジェクトで渡しています。

DOM 中の <button> 要素にはクラス属性として PrimaryButton が渡され、 onClick 属性の関数や children プロパティの文字列（Hello World!）が渡されます。

JSX における子要素

開始タグと終了タグの両方を含む JSX 式の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。

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。

以下の場合 props.children は単なる文字列 "Hello world!" です。

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

JSX は行の先頭と末尾の空白文字を削除し、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。

```
<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、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 JSX と HTML は似ています。以下のような例は 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。

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return (
    //配列
    [
      <h1>Apple</h1>,
      <div>Banana</div>,
      <p>Orange</p>
    ]
  )
}

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

以下が上記により出力される HTML です。

```
<div id="root">
  <h1>Apple</h1>
  <div>Banana</div>
  <p>Orange</p>
</div>
```

子要素としての JavaScript 式

JSX では任意の JavaScript の式を { } 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 の式は等しいものです。

```
<MyComponent>15</MyComponent>

<MyComponent>{ 5*3 }</MyComponent>
```

子要素を表示する際に JavaScript の関数などの式が使えるので、以下のように map() を使ってリストを表示することなどができます。

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

function FruitList() {
  const fruits = ['Apple', 'Banana', 'Orange'];
  return (
    <ul>
      {fruits.map((value) => <Item key={value} value={value} />)}
    </ul>
  );
}

function UserList() {
  const users = [
    { id: '001', name: 'Foo'},
    { id: '002', name: 'Bar'},
    { id: '003', name: 'Boo'}
  ];
  return (
    <ul>
      {users.map((value) => <Item key={value.id} value={value.name} />)}
    </ul>
  );
}
```

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 要素として評価されます。

子要素として渡される props.children に関数を渡して props.children を通してコールバックを定義することもできます。

以下の DoChildFunc コンポーネントは props.val で渡された値を引数として props.children で渡される関数を実行します。ShowVal コンポーネントでは、引数 val で受け取った値を表示する p 要素を返す関数を子要素として定義しています。

```
import React from 'react';
import ReactDOM from 'react-dom';

function DoChildFunc(props) {
  //子要素として渡された関数 props.children() を div 要素内で実行して返す
  return <div>{props.children(props.val)}</div>;
}

function ShowVal(props) {
  return (
    <DoChildFunc val={props.val}>
      { function(val) {
          return <p>The val is {val}. </p>;
        }
      }
    </DoChildFunc>
  );
}

ReactDOM.render(
  <ShowVal val="Foo"/>,
  document.getElementById('root')
)
```

以下は上記により出力される HTML です。

```
<div id="root">
  <div>
    <p>The val is Foo. </p>
  </div>
</div>
```

以下の Repeat コンポーネントは props.numTimes で渡された回数だけ子要素で渡された関数 props.children() を実行して要素の配列を生成して返します（React コンポーネントは要素の配列を返すことができます）。

```
import React from 'react';
import ReactDOM from 'react-dom';

function Repeat(props) {
  //要素（JSX）を格納する配列の初期化
  let items = [];
  //上記配列に props.numTimes の回数分 props.children() を実行して要素を追加
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  //上記で作成した要素の配列を返す
  return <div>{items}</div>;
}

function ListOfNThings(props) {
  return (
    <Repeat numTimes={props.numTimes}>
      {(index) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}

const App = () => {
  return (
    <ListOfNThings numTimes={3}/>
  );
};

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

ListOfNThings コンポーネントは引数にインデックスを受け取り、key 属性と本文にそのインデックスを設定した div 要素を返す関数を子要素として定義しています。

子要素の関数を function 文を使って記述すると以下のようになります。

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

以下は上記により出力される HTML です。

```
<div id="root">
  <div>
    <div>This is item 0 in the list</div>
    <div>This is item 1 in the list</div>
    <div>This is item 2 in the list</div>
  </div>
</div>
```