React and ReactDOM are often discussed in the same spaces as — and utilized to solve the same problems as — other true web development frameworks. When we refer to React as a "framework", we’re working with that colloquial understanding.

React's primary goal is to minimize the bugs that occur when developers are building UIs. It does this through the use of components — self-contained, logical pieces of code that describe a portion of the user interface. These components can be composed together to create a full UI, and React abstracts away much of the rendering work, leaving you to concentrate on the UI design.

The **src** directory is where we'll spend most of our time, as it's where the source code for our application lives.

The **public** directory contains files that will be read by your browser while you're developing the app; the most important of these is index.html. React injects your code into this file so that your browser can run it.

The package.json file contains information about our project that Node.js/npm uses to keep it organized.

import React from 'react';

The first statement imports the React library itself. Because React turns the JSX we write into React.createElement(), all React components must import the React module. If you skip this step, your application will produce an error.

React components use pascal-case variable names, like HelloWorld, to make it clear that a given JSX element is a React component, and not a regular HTML tag. If you were to rename the App function to app, your browser would show you an error.

Some elements in the expression have attributes, which are written just like in HTML, following a pattern of attribute="value". On line 3, the opening [<div>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div) tag has a className attribute. This is the same as the [class](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class) attribute in HTML, but because JSX is JavaScript, we can't use the word class — it's reserved, meaning JavaScript already uses it for a specific purpose and it would cause problems here in our code. A few other HTML attributes are written differently in JSX than they are in HTML too, for the same kind of reason. We'll cover them as we encounter them.

Let’s open src/index.js, because that's where the App component is being used. This file is the entry point for our app, and it initially looks like this:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

);

Line 7 calls React’s ReactDOM.render() function with two arguments:

* The component we want to render, <App /> in this case.
* The DOM element inside which we want the component to be rendered, in this case the element with an ID of root. If you look inside public/index.html, you'll see that this is a <div> element just inside the <body>.
* Back in App.js, let’s focus on line 9:
* <img src={logo} className="App-logo" alt="logo" />
* Copy to Clipboard
* Here, the <img /> tag's src attribute value is in curly braces. This is how JSX recognizes variables. React will see {logo}, know you are referring to the logo import on line 2 of our app, then retrieve the logo file and render it.

A **prop** is any data passed into a React component. React props are comparable to HTML attributes. Where HTML elements have attributes, React components have props. Props are written inside component calls, and use the same syntax as HTML attributes — prop="value".  In React, dataflow is unidirectional: props can only be passed from Parent components down to Child components; and props are read-only.

* Components can import modules they need and must export themselves at the bottom of their files.
* Component functions are named with PascalCase.
* You can read JSX variables by putting them between curly braces, like {so}.
* Some JSX attributes are different than HTML attributes so that they don't conflict with JavaScript reserved words. For example, class in HTML translates to className in JSX. Note that multi-word attributes are camel-cased.
* Props are written just like attributes inside component calls and are passed into components.

**Note:** Components must always return something. If at any point in the future you try to render a component that does not return anything, React will display an error in your browser.

key is a special prop that's managed by React – you cannot use the word key for any other purpose. **You should always pass a unique key to anything you render with iteration.** Nothing obvious will change in your browser, but if you do not use unique keys, React will log warnings to your console and your app may behave strangely!

<Todo

id={task.id}

name={task.name}

completed={task.completed}

key={task.id}

/>

If you've only written vanilla JavaScript before now, you might be used to having a separate JavaScript file, where you query for some DOM nodes and attach listeners to them. For example:

const btn = document.querySelector('button');

btn.addEventListener('click', () => {

alert("hi!");

});

Copy to Clipboard

In React, we write event handlers directly on the elements in our JSX, like this:

<button

type="button"

onClick={() => alert("hi!")}

>

Say hi!

</button>

Copy to Clipboard

**Note:** This may seem counter-intuitive regarding best-practice advice that tends to advise against use of inline event handlers on HTML, but remember that JSX is actually part of your JavaScript.

In the above example, we're adding an onClick attribute to the <button> element. The value of that attribute is a function that triggers a simple alert.

The onClick attribute has special meaning here: it tells React to run a given function when the user clicks on the button. There are a couple of other things to note:

* The camel-cased nature of onClick is important — JSX will not recognize onclick (again, it is already used in JavaScript for a specific purpose, which is related but different — standard [onclick](https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onclick) handler properties).
* All browser events follow this format in JSX – on, followed by the name of the event.

Let's apply this to our app, starting in the Form.js component.

**Note:** We decided to name our callback prop addTask to make it easy to understand what the prop will do. Another common convention you may well come across in React code is to prefix callback prop names with the word on, followed by the name of the event that will cause them to be run. For instance, we could have given our form a prop of onSubmit with the value of addTask.

Это JSX — расширение языка JavaScript

Вместо того, чтобы искусственно разделить технологии, помещая разметку и логику в разные файлы, React [разделяет ответственность](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE%D1%82%D0%B2%D0%B5%D1%82%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8) с помощью слабо связанных единиц, называемых «компоненты», которые содержат и разметку, и логику.

**JSX это тоже выражение**

После компиляции каждое JSX-выражение становится обычным вызовом JavaScript-функции, результат которого — объект JavaScript.

Из этого следует, что JSX можно использовать внутри инструкций if и циклов for, присваивать переменным, передавать функции в качестве аргумента и возвращать из функции.

function getGreeting(user) {

if (user) {

return <h1>Здравствуй, {formatName(user)}!</h1>; }

return <h1>Здравствуй, незнакомец.</h1>;}

Чтобы использовать строковый литерал в качестве значения атрибута, используются кавычки:

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

Если же в значении атрибута требуется указать JavaScript-выражение, то на помощь приходят фигурные скобки:

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

**Использование дочерних элементов в JSX**

Если тег пуст, то его можно сразу же закрыть с помощью /> точно так же, как и в XML:

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

Но JSX-теги могут и содержать дочерние элементы:

const element = (

<div>

<h1>Здравствуйте!</h1>

<h2>Рады вас видеть.</h2>

</div>

);

Элементы — мельчайшие кирпичики React-приложений.

Элемент описывает то, что вы хотите увидеть на экране:

const element = <h1>Hello, world</h1>;

В отличие от DOM-элементов, элементы React — это простые объекты, не отнимающие много ресурсов. React DOM обновляет DOM, чтобы он соответствовал переданным React-элементам.

Элементы — это то, «из чего сделаны» компоненты, и мы рекомендуем вам дочитать эту главу, прежде чем двигаться дальше.

Допустим, в вашем HTML-файле есть <div>:

<div id="root"></div>

Мы назовём его «корневым» узлом DOM, так как React DOM будет управлять его содержимым.

Обычно в приложениях, написанных полностью на React, есть только один корневой элемент. При встраивании React в существующее приложение вы можете рендерить во столько независимых корневых элементов, во сколько посчитаете нужным.

Для рендеринга React-элемента в корневой узел DOM вызовите [ReactDOM.render()](https://ru.reactjs.org/docs/react-dom.html" \l "render) с React-элементом и корневым DOM-узлом в качестве аргументов:

const element = <h1>Hello, world</h1>;

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

Элементы React [иммутабельны](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B8%D0%B7%D0%BC%D0%B5%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82" \t "_blank). После создания элемента нельзя изменить его потомков или атрибуты. Элемент похож на кадр в фильме: он отражает состояние интерфейса в конкретный момент времени.

React DOM сравнивает элемент и его дочернее дерево с предыдущей версией и вносит в DOM только минимально необходимые изменения.

Во многом компоненты ведут себя как обычные функции JavaScript. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают React-элементы, описывающие, что мы хотим увидеть на экране.

Проще всего объявить React-компонент как функцию:

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

Эта функция — компонент, потому что она получает данные в одном объекте («пропсы») в качестве параметра и возвращает React-элемент. Мы будем называть такие компоненты «функциональными», так как они буквально являются функциями.

Ещё компоненты можно определять как [классы ES6](https://developer.mozilla.org/ru-RU/docs/Web/JavaScript/Reference/Classes):

class Welcome extends React.Component {

render() {

return <h1>Привет, {this.props.name}</h1>;

}

}

С точки зрения React, эти два компонента эквивалентны.

Но элементы могут описывать и наши собственные компоненты:

const element = <Welcome name="Алиса" />;

Когда React встречает подобный элемент, он собирает все JSX-атрибуты и дочерние элементы в один объект и передаёт их нашему компоненту. Этот объект называется «пропсы» (props).

В приложениях, написанных на React с нуля, как правило, есть один компонент App, который находится на самом верху. В случае, если вы переписываете существующее приложение на React, имеет смысл начать работу с маленького компонента типа Button и постепенно двигаться «вверх» по иерархии.

Пропсы следует называть так, чтобы они имели смысл в первую очередь с точки зрения самого компонента, а уже во вторую тех компонентов, которые его рендерят.