目錄

QUICK START

ADVANCED GUIDES

REFERENCE

简介

根据 https://facebook.github.io/react/docs 官方文档制作的 React 离线文档.

by 范圣刚

React is flexible and can be used in a variety of projects. You can create new apps with it, but you can also gradually introduce it into an existing codebase without doing a rewrite.

Trying Out React

If you're just interested in playing around with React, you can use CodePen. Try starting from this Hello World example code. You don't need to install anything; you can just modify the code and see if it works.

If you prefer to use your own text editor, you can also download this HTML file, edit it, and open it from the local filesystem in your browser. It does a slow runtime code transformation, so don't use it in production.

Creating a Single Page Application

Create React App is the best way to start building a new React single page application. It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production.

```
npm install -g create-react-app
create-react-app hello-world
cd hello-world
npm start
```

Create React App doesn't handle backend logic or databases; it just creates a frontend build pipeline, so you can use it with any backend you want. It uses webpack, Babel and ESLint under the hood, but configures them for you.

Adding React to an Existing Application

You don't need to rewrite your app to start using React.

We recommend adding React to a small part of your application, such as an individual widget, so you can see if it works well for your use case.

While React can be used without a build pipeline, we recommend setting it up so you can be more productive. A modern build pipeline typically consists of:

- A **package manager**, such as Yarn or npm. It lets you take advantage of a vast ecosystem of third-party packages, and easily install or update them.
- A **bundler**, such as webpack or Browserify. It lets you write modular code and bundle it together into small packages to optimize load time.
- A **compiler** such as Babel. It lets you write modern JavaScript code that still works in older browsers.

Installing React

We recommend using Yarn or npm for managing front-end dependencies. If you're new to package managers, the Yarn documentation is a good place to get started.

To install React with Yarn, run:

```
yarn init
yarn add react react-dom
```

To install React with npm, run:

```
npm init
npm install --save react react-dom
```

Both Yarn and npm download packages from the npm registry.

Enabling ES6 and JSX

We recommend using React with Babel to let you use ES6 and JSX in your JavaScript code. ES6 is a set of modern JavaScript features that make development easier, and JSX is an extension to the JavaScript language that works nicely with React.

The Babel setup instructions explain how to configure Babel in many different build environments. Make sure you install `babel-preset-react` and `babel-preset-es2015` and enable them in your `.babelrc` configuration, and you're good to go.

Hello World with ES6 and JSX

We recommend using a bundler like webpack or Browserify so you can write modular code and bundle it together into small packages to optimize load time.

The smallest React example looks like this:

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

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

This code renders into a DOM element with the id of `root` so you need `<div id="root"></div>` somewhere in your HTML file.

Similarly, you can render a React component inside a DOM element somewhere inside your existing app written with any other JavaScript UI library.

Development and Production Versions

By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.

Create React App

If you use Create React App, `npm run build` will create an optimized build of your app in the `build` folder.

Webpack

Include both `DefinePlugin` and `UglifyJsPlugin` into your production Webpack configuration as described in this guide.

Browserify

Run Browserify with `NODE_ENV` environment variable set to `production` and use UglifyJS as the last build step so that development-only code gets stripped out.

Rollup

Use rollup-plugin-replace plugin together with rollup-plugin-commonjs (in that order) to remove development-only code. See this gist for a complete setup example.

Using a CDN

If you don't want to use npm to manage client packages, the `react` and `react-dom` npm packages also provide single-file distributions in `dist` folders, which are hosted on a CDN:

```
<script src="https://unpkg.com/react@15/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>
```

The versions above are only meant for development, and are not suitable for production. Minified and optimized production versions of React are available at:

```
<script src="https://unpkg.com/react@15/dist/react.min.js"></script>
<script src="https://unpkg.com/react-dom@15/dist/react-dom.min.js"></script>
```

To load a specific version of `react` and `react-dom`, replace `15` with the version number.

If you use Bower, React is available via the `react` package.

The easiest way to get started with React is to use this Hello World example code on CodePen. You don't need to install anything; you can just open it in another tab and follow along as we go through examples. If you'd rather use a local development environment, check out the Installation page.

The smallest React example looks like this:

```
ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

It renders a header saying "Hello, world!" on the page.

The next few sections will gradually introduce you to using React. We will examine the building blocks of React apps: elements and components. Once you master them, you can create complex apps from small reusable pieces.

A Note on JavaScript

React is a JavaScript library, and so it assumes you have a basic understanding of the JavaScript language. If you don't feel very confident, we recommend refreshing your JavaScript knowledge so you can follow along more easily.

We also use some of the ES6 syntax in the examples. We try to use it sparingly because it's still relatively new, but we encourage you to get familiar with arrow functions, classes, template literals, `let`, and `const` statements. You can use Babel REPL to check what ES6 code compiles to.

Consider this variable declaration:

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

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

JSX produces React "elements". We will explore rendering them to the DOM in the next section. Below, you can find the basics of JSX necessary to get you started.

Embedding Expressions in JSX

You can embed any JavaScript expression in JSX by wrapping it in curly braces.

For example, `2 + 2`, `user.firstName`, and `formatName(user)` are all valid expressions:

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

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

Try it on CodePen.

We split JSX over multiple lines for readability. While it isn't required, when doing this, we also recommend wrapping it in parentheses to avoid the pitfalls of automatic semicolon insertion.

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript objects.

This means that you can use JSX inside of `if` statements and `for` loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

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

You may also use curly braces to embed a JavaScript expression in an attribute:

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

Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. Otherwise JSX will treat the attribute as a string literal rather than an expression. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Specifying Children with JSX

If a tag is empty, you may close it immediately with `/>`, like XML:

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

JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
```

```
  </div>
);
```

> **Caveat:**
>
> Since JSX is closer to JavaScript than HTML, React DOM uses `camelCase` property naming convention instead of HTML attribute names.
>
> For example, `class` becomes `className` in JSX, and `tabindex` becomes `tabIndex`.

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX Represents Objects

Babel compiles JSX down to `React.createElement()` calls.

These two examples are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()` performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world'
  }
};
```

These objects are called "React elements". You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

We will explore rendering React elements to the DOM in the next section.

> **Tip:**
>
> We recommend searching for a "Babel" syntax scheme for your editor of choice so that both ES6 and JSX code is properly highlighted.

Elements are the smallest building blocks of React apps.

An element describes what you want to see on the screen:

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

Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

> **Note:**
>
> One might confuse elements with a more widely known concept of "components". We will introduce components in the next section. Elements are what components are "made of", and we encourage you to read this section before jumping ahead.

Rendering an Element into the DOM

Let's say there is a `<div>` somewhere in your HTML file:

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

We call this a "root" DOM node because everything inside it will be managed by React DOM.

Applications built with just React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element into a root DOM node, pass both to `ReactDOM.render()`:

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Try it on CodePen.

It displays "Hello, world" on the page.

Updating the Rendered Element

React elements are immutable. Once you create an element, you can't change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to `ReactDOM.render()`.

Consider this ticking clock example:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Try it on CodePen.

It calls `ReactDOM.render()` every second from a `setInterval()` callback.

> **Note:**
>
> In practice, most React apps only call `ReactDOM.render()` once. In the next sections we will learn how such code gets encapsulated into stateful components.
>
> We recommend that you don't skip topics because they build on each other.

React Only Updates What's Necessary

React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

You can verify by inspecting the last example with the browser tools:

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents has changed gets updated by React DOM.

In our experience, thinking about how the UI should look at any given moment rather than how to change it over time eliminates a whole class of bugs.

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Functional and Class Components

The simplest way to define a component is to write a JavaScript function:

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

This function is a valid React component because it accepts a single "props" object argument with data and returns a React element. We call such components "functional" because they are literally JavaScript functions.

You can also use an ES6 class to define a component:

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

The above two components are equivalent from React's point of view.

Classes have some additional features that we will discuss in the next sections. Until then, we will use functional components for their conciseness.

Rendering a Component

Previously, we only encountered React elements that represent DOM tags:

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

However, elements can also represent user-defined components:

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

When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

For example, this code renders "Hello, Sara" on the page:

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

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

Try it on CodePen.

Let's recap what happens in this example:

1. We call `ReactDOM.render()` with the `<Welcome name="Sara" />` element.
2. React calls the `Welcome` component with `{name: 'Sara'}` as the props.
3. Our `Welcome` component returns a `<h1>Hello, Sara</h1>` element as the result.
4. React DOM efficiently updates the DOM to match `<h1>Hello, Sara</h1>`.

> **Caveat:**
>
> Always start component names with a capital letter.
>
> For example, `<div />` represents a DOM tag, but `<Welcome />` represents a component and requires `Welcome` to be in scope.

Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

For example, we can create an `App` component that renders `Welcome` many times:

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

```
}

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

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

Try it on CodePen.

Typically, new React apps have a single `App` component at the very top. However, if you integrate React into an existing app, you might start bottom-up with a small component like `Button` and gradually work your way to the top of the view hierarchy.

> **Caveat:**
>
> Components must return a single root element. This is why we added a `<div>` to contain all the `<Welcome />` elements.

Extracting Components

Don't be afraid to split components into smaller components.

For example, consider this `Comment` component:

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

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

Try it on CodePen.

It accepts `author` (an object), `text` (a string), and `date` (a date) as props, and describes a comment on a social media website.

This component can be tricky to change because of all the nesting, and it is also hard to reuse individual parts of it. Let's extract a few components from it.

First, we will extract `Avatar`:

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

The `Avatar` doesn't need to know that it is being rendered inside a `Comment`. This is why we have given its prop a more generic name: `user` rather than `author`.

We recommend naming props from the component's own point of view rather than the context in which it is being used.

We can now simplify `Comment` a tiny bit:

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

```
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Next, we will extract a `UserInfo` component that renders an `Avatar` next to user's name:

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

This lets us simplify `Comment` even further:

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

Try it on CodePen.

Extracting components might seem like grunt work at first, but having a palette of reusable components pays off in larger apps. A good rule of thumb is that if a part of your UI is used several times (`Button` , `Panel` , `Avatar`), or is complex enough on its own (`App` , `FeedStory` , `Comment`), it is a good candidate to be a reusable component.

Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this `sum` function:

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

Such functions are called "pure" because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, this function is impure because it changes its own input:

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

React is pretty flexible but it has a single strict rule:

All React components must act like pure functions with respect to their props.

Of course, application UIs are dynamic and change over time. In the next section, we will introduce a new concept of "state". State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.

Consider the ticking clock example from one of the previous sections.

So far we have only learned one way to update the UI.

We call `ReactDOM.render()` to change the rendered output:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Try it on CodePen.

In this section, we will learn how to make the `Clock` component truly reusable and encapsulated. It will set up its own timer and update itself every second.

We can start by encapsulating how the clock looks:

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

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

setInterval(tick, 1000);
```

Try it on CodePen.

However, it misses a crucial requirement: the fact that the `Clock` sets up a timer and updates the UI every second should be an implementation detail of the `Clock`.

Ideally we want to write this once and have the `Clock` update itself:

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

To implement this, we need to add "state" to the `Clock` component.

State is similar to props, but it is private and fully controlled by the component.

We mentioned before that components defined as classes have some additional features. Local state is exactly that: a feature available only to classes.

Converting a Function to a Class

You can convert a functional component like `Clock` to a class in five steps:

1. Create an ES6 class with the same name that extends `React.Component`.
2. Add a single empty method to it called `render()`.
3. Move the body of the function into the `render()` method.
4. Replace `props` with `this.props` in the `render()` body.
5. Delete the remaining empty function declaration.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Try it on CodePen.

`Clock` is now defined as a class rather than a function.

This lets us use additional features such as local state and lifecycle hooks.

Adding Local State to a Class

We will move the `date` from props to state in three steps:

1) Replace `this.props.date` with `this.state.date` in the `render()` method:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2) Add a class constructor that assigns the initial `this.state`:

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

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Note how we pass `props` to the base constructor:

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

Class components should always call the base constructor with `props`.

3) Remove the `date` prop from the `<Clock />` element:

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

We will later add the timer code back to the component itself.

The result looks like this:

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

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Try it on CodePen.

Next, we'll make the `Clock` set up its own timer and update itself every second.

Adding Lifecycle Methods to a Class

In applications with many components, it's very important to free up resources taken by the components when they are destroyed.

We want to set up a timer whenever the `Clock` is rendered to the DOM for the first time. This is called "mounting" in React.

We also want to clear that timer whenever the DOM produced by the `Clock` is removed. This is called "unmounting" in React.

We can declare special methods on the component class to run some code when a component mounts and unmounts:

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

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

These methods are called "lifecycle hooks".

The `componentDidMount()` hook runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

```
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
```

Note how we save the timer ID right on `this`.

While `this.props` is set up by React itself and `this.state` has a special meaning, you are free to add additional fields to the class manually if you need to store something that is not used for the visual output.

If you don't use something in `render()`, it shouldn't be in the state.

We will tear down the timer in the `componentWillUnmount()` lifecycle hook:

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
```

Finally, we will implement the `tick()` method that runs every second.

It will use `this.setState()` to schedule updates to the component local state:

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

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

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

Try it on CodePen.

Now the clock ticks every second.

Let's quickly recap what's going on and the order in which the methods are called:

1) When `<Clock />` is passed to `ReactDOM.render()`, React calls the constructor of the `Clock` component. Since `Clock` needs to display the current time, it initializes `this.state` with an object including the current time. We will later update this state.

2) React then calls the `Clock` component's `render()` method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the `Clock`'s render output.

3) When the `Clock` output is inserted in the DOM, React calls the `componentDidMount()` lifecycle hook. Inside it, the `Clock` component asks the browser to set up a timer to call `tick()` once a second.

4) Every second the browser calls the `tick()` method. Inside it, the `Clock` component schedules a UI update by calling `setState()` with an object containing the current time. Thanks to the `setState()` call, React knows the state has changed, and calls `render()` method again to learn what should be on the screen. This time, `this.state.date` in the `render()` method will be different, and so the render output will include the updated time. React updates the DOM accordingly.

5) If the `Clock` component is ever removed from the DOM, React calls the `componentWillUnmount()` lifecycle hook so the timer is stopped.

Using State Correctly

There are three things you should know about `setState()`.

Do Not Modify State Directly

For example, this will not re-render a component:

```
// Wrong
```

```
this.state.comment = 'Hello';
```

Instead, use `setState()`:

```
// Correct
this.setState({comment: 'Hello'});
```

The only place where you can assign `this.state` is the constructor.

State Updates May Be Asynchronous

React may batch multiple `setState()` calls into a single update for performance.

Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

For example, this code may fail to update the counter:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

To fix it, use a second form of `setState()` that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

```
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

We used an arrow function above, but it also works with regular functions:

```
// Correct
this.setState(function(prevState, props) {
  return {
    counter: prevState.counter + props.increment
  };
});
```

State Updates are Merged

When you call `setState()`, React merges the object you provide into the current state.

For example, your state may contain several independent variables:

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

Then you can update them independently with separate `setState()` calls:

```
  componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
    });

    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

The merging is shallow, so `this.setState({comments})` leaves `this.state.posts` intact, but completely replaces `this.state.comments`.

The Data Flows Down

Neither parent nor child components can know if a certain component is stateful or stateless, and they shouldn't care whether it is defined as a function or a class.

This is why state is often called local or encapsulated. It is not accessible to any component other than the one that owns and sets it.

A component may choose to pass its state down as props to its child components:

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

This also works for user-defined components:

```
<FormattedDate date={this.state.date} />
```

The `FormattedDate` component would receive the `date` in its props and wouldn't know whether it came from the `Clock` 's state, from the `Clock` 's props, or was typed by hand:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

Try it on CodePen.

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

If you imagine a component tree as a waterfall of props, each component's state is like an additional water source that joins it at an arbitrary point but also flows down.

To show that all components are truly isolated, we can create an `App` component that renders three `<Clock>` s:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

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

Try it on CodePen.

Each `Clock` sets up its own timer and updates independently.

In React apps, whether a component is stateful or stateless is considered an implementation detail of the component that may change over time. You can use stateless components inside stateful components, and vice versa.

Handling events with React elements is very similar to handling events on DOM elements. There are some syntactic differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Here, `e` is a synthetic event. React defines these synthetic events according to the W3C spec, so you don't need to worry about cross-browser compatibility. See the `SyntheticEvent` reference guide to learn more.

When using React you should generally not need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class. For example, this `Toggle` component renders a button that lets the user toggle between "ON" and "OFF" states:

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Try it on CodePen.

You have to be careful about the meaning of `this` in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, `this` will be `undefined` when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without `()` after it, such as `onClick={this.handleClick}`, you should bind that method.

If calling `bind` annoys you, there are two ways you can get around this. If you are using the experimental property initializer syntax, you can use property initializers to correctly bind callbacks:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

This syntax is enabled by default in Create React App.

If you aren't using property initializer syntax, you can use an arrow function in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Click me
      </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the `LoggingButton` renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the property initializer syntax, to avoid this sort of performance problem.

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like `if` or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Consider these two components:

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

We'll create a `Greeting` component that displays either of these components depending on whether a user is logged in:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

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

Try it on CodePen.

This example renders a different greeting depending on the value of `isLoggedIn` prop.

Element Variables

You can use variables to store elements. This can help you conditionally render a part of the component while the rest of the output doesn't change.

Consider these two new components representing Logout and Login buttons:

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

In the example below, we will create a stateful component called `LoginControl`.

It will render either `<LoginButton />` or `<LogoutButton />` depending on its current state. It will also render a `<Greeting />` from the previous example:

```
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 = null;
    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')
);
```

Try it on CodePen.

While declaring a variable and using an `if` statement is a fine way to conditionally render a component, sometimes you might want to use a shorter syntax. There are a few ways to inline conditions in JSX, explained below.

Inline If with Logical && Operator

You may embed any expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical `&&` operator. It can be handy for conditionally including an element:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

Try it on CodePen.

It works because in JavaScript, `true && expression` always evaluates to `expression`, and `false && expression` always evaluates to `false`.

Therefore, if the condition is `true`, the element right after `&&` will appear in the output. If it is `false`, React will ignore and skip it.

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator `condition ? true : false`.

In the example below, we use it to conditionally render a small block of text.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

It can also be used for larger expressions although it is less obvious what's going on:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Just like in JavaScript, it is up to you to choose an appropriate style based on what you and your team consider more readable. Also remember that whenever conditions become too complex, it might be a good time to extract a component.

Preventing Component from Rendering

In rare cases you might want a component to hide itself even though it was rendered by another component. To do this return `null` instead of its render output.

In the example below, the `<WarningBanner />` is rendered depending on the value of the prop called `warn`. If the value of the prop is `false`, then the component does not render:

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true}
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(prevState => ({
      showWarning: !prevState.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

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

Try it on CodePen.

Returning `null` from a component's `render` method does not affect the firing of the component's lifecycle methods. For instance, `componentWillUpdate` and `componentDidUpdate` will still be called.

First, let's review how you transform lists in JavaScript.

Given the code below, we use the `map()` function to take an array of `numbers` and double their values. We assign the new array returned by `map()` to the variable `doubled` and log it:

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

This code logs `[2, 4, 6, 8, 10]` to the console.

In React, transforming arrays into lists of elements is nearly identical.

Rendering Multiple Components

You can build collections of elements and include them in JSX using curly braces `{}`.

Below, we loop through the `numbers` array using the Javascript `map()` function. We return an `<li>` element for each item. Finally, we assign the resulting array of elements to `listItems`:

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

We include the entire `listItems` array inside a `<ul>` element, and render it to the DOM:

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

Try it on CodePen.

This code displays a bullet list of numbers between 1 and 5.

Basic List Component

Usually you would render lists inside a component.

We can refactor the previous example into a component that accepts an array of `numbers` and outputs an unordered list of elements.

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

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

When you run this code, you'll be given a warning that a key should be provided for list items. A "key" is a special string attribute you need to include when creating lists of elements. We'll discuss why it's important in the next section.

Let's assign a `key` to our list items inside `numbers.map()` and fix the missing key issue.

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

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

Try it on CodePen.

Keys

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

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

The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys:

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

When you don't have stable IDs for rendered items, you may use the item index as a key as a last resort:

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

We don't recommend using indexes for keys if the items can reorder, as that would be slow. You may read an in-depth explanation about why keys are necessary if you're interested.

Extracting Components with Keys

Keys only make sense in the context of the surrounding array.

For example, if you extract a `ListItem` component, you should keep the key on the `<ListItem />` elements in the array rather than on the root `<li>` element in the `ListItem` itself.

Example: Incorrect Key Usage

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

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

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

Example: Correct Key Usage

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

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

```
  );
}

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

Try it on CodePen.

A good rule of thumb is that elements inside the `map()` call need keys.

Keys Must Only Be Unique Among Siblings

Keys used within arrays should be unique among their siblings. However they don't need to be globally unique. We can use the same keys when we produce two different arrays:

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

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

```
  document.getElementById('root')
);
```

Try it on CodePen.

Keys serve as a hint to React but they don't get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name:

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

With the example above, the `Post` component can read `props.id`, but not `props.key`.

Embedding map() in JSX

In the examples above we declared a separate `listItems` variable and included it in JSX:

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

JSX allows embedding any expressions in curly braces so we could inline the `map()` result:

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

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

Try it on CodePen.

Sometimes this results in clearer code, but this style can also be abused. Like in JavaScript, it is up to you to decide whether it is worth extracting a variable for readability. Keep in mind that if the `map()` body is too nested, it might be a good time to extract a component.

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "controlled components".

Controlled Components

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with `setState()`.

We can combine the two by making the React state be the "single source of truth". Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a "controlled component".

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

```
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange}
 />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Try it on CodePen.

Since the `value` attribute is set on our form element, the displayed value will always be `this.state.value`, making the React state the source of truth. Since `handleChange` runs on every keystroke to update the React state, the displayed value will update as the user types.

With a controlled component, every state mutation will have an associated handler function. This makes it straightforward to modify or validate user input. For example, if we wanted to enforce that names are written with all uppercase letters, we could write `handleChange` as:

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

The textarea Tag

In HTML, a `<textarea>` element defines its text by its children:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

In React, a `<textarea>` uses a `value` attribute instead. This way, a form using a `<textarea>` can be written very similarly to a form that uses a single-line input:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Notice that `this.state.value` is initialized in the constructor, so that the text area starts off with some text in it.

The select Tag

In HTML, `<select>` creates a drop-down list. For example, this HTML creates a drop-down list of flavors:

```
<select>
  <option value="grapefruit">Grapefruit</option>
```

```
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

Note that the Coconut option is initially selected, because of the `selected` attribute. React, instead of using this `selected` attribute, uses a `value` attribute on the root `select` tag. This is more convenient in a controlled component because you only need to update it in one place. For example:

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite La Croix flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Try it on CodePen.

Overall, this makes it so that `<input type="text">`, `<textarea>`, and `<select>` all work very similarly - they all accept a `value` attribute that you can use to implement a controlled component.

Handling Multiple Inputs

When you need to handle multiple controlled `input` elements, you can add a `name` attribute to each element and let the handler function choose what to do based on the value of `event.target.name`.

For example:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
```

```
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Try it on CodePen.

Note how we used the ES6 computed property name syntax to update the state key corresponding to the given input name:

```
this.setState({
  [name]: value
});
```

It is equivalent to this ES5 code:

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Also, since `setState()` automatically merges a partial state into the current state, we only needed to call it with the changed parts.

Alternatives to Controlled Components

It can sometimes be tedious to use controlled components, because you need to write an event handler for every way your data can change and pipe all of the input state through a React component. This can become particularly annoying when you are converting a preexisting codebase to React, or integrating a React application with a non-React library. In these situations, you might want to check out uncontrolled components, an alternative technique for implementing input forms.

Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor. Let's see how this works in action.

In this section, we will create a temperature calculator that calculates whether the water would boil at a given temperature.

We will start with a component called `BoilingVerdict`. It accepts the `celsius` temperature as a prop, and prints whether it is enough to boil the water:

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

Next, we will create a component called `Calculator`. It renders an `<input>` that lets you enter the temperature, and keeps its value in `this.state.value`.

Additionally, it renders the `BoilingVerdict` for the current input value.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {value: ''};
  }

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

  render() {
    const value = this.state.value;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={value}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(value)} />
      </fieldset>
    );
  }
}
```

Try it on CodePen.

Adding a Second Input

Our new requirement is that, in addition to a Celsius input, we provide a Fahrenheit input, and they are kept in sync.

We can start by extracting a `TemperatureInput` component from `Calculator`. We will add a new `scale` prop to it that can either be `"c"` or `"f"`:

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {value: ''};
  }

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

  render() {
    const value = this.state.value;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={value}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

We can now change the `Calculator` to render two separate temperature inputs:

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
```

```
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

Try it on CodePen.

We have two inputs now, but when you enter the temperature in one of them, the other doesn't update. This contradicts our requirement: we want to keep them in sync.

We also can't display the `BoilingVerdict` from `Calculator`. The `Calculator` doesn't know the current temperature because it is hidden inside the `TemperatureInput`.

Lifting State Up

First, we will write two functions to convert from Celsius to Fahrenheit and back:

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

These two functions convert numbers. We will write another function that takes a string `value` and a converter function as arguments and returns a string. We will use it to calculate the value of one input based on the other input.

It returns an empty string on an invalid `value`, and it keeps the output rounded to the third decimal place:

```
function tryConvert(value, convert) {
  const input = parseFloat(value);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

For example, `tryConvert('abc', toCelsius)` returns an empty string, and `tryConvert('10.22', toFahrenheit)` returns `'50.396'`.

Next, we will remove the state from `TemperatureInput`.

Instead, it will receive both `value` and the `onChange` handler by props:

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    const value = this.props.value;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={value}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

If several components need access to the same state, it is a sign that the state should be lifted up to their closest common ancestor instead. In our case, this is the `Calculator`. We will store the current `value` and `scale` in its state.

We could have stored the value of both inputs but it turns out to be unnecessary. It is enough to store the value of the most recently changed input, and the scale that it represents. We can then infer the value of the other input based on the current `value` and `scale` alone.

The inputs stay in sync because their values are computed from the same state:

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {value: '', scale: 'c'};
```

```
  }

  handleCelsiusChange(value) {
    this.setState({scale: 'c', value});
  }

  handleFahrenheitChange(value) {
    this.setState({scale: 'f', value});
  }

  render() {
    const scale = this.state.scale;
    const value = this.state.value;
    const celsius = scale === 'f' ? tryConvert(value, toCelsius) : value;
    const fahrenheit = scale === 'c' ? tryConvert(value, toFahrenheit) : value;

    return (
      <div>
        <TemperatureInput
          scale="c"
          value={celsius}
          onChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          value={fahrenheit}
          onChange={this.handleFahrenheitChange} />
        <BoilingVerdict
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
```

Try it on CodePen.

Now, no matter which input you edit, `this.state.value` and `this.state.scale` in the `Calculator` get updated. One of the inputs gets the value as is, so any user input is preserved, and the other input value is always recalculated based on it.

Lessons Learned

There should be a single "source of truth" for any data that changes in a React application. Usually, the state is first added to the component that needs it for rendering. Then, if other components also need it, you can lift it up to their closest common ancestor. Instead of trying to sync the state between different components, you should rely on the top-down data flow.

Lifting state involves writing more "boilerplate" code than two-way binding approaches, but as a benefit, it takes less work to find and isolate bugs. Since any state "lives" in some component and that component alone can change it, the surface area for bugs is greatly reduced. Additionally, you can implement any custom logic to reject or transform user input.

If something can be derived from either props or state, it probably shouldn't be in the state. For example, instead of storing both `celsiusValue` and `fahrenheitValue`, we store just the last edited `value` and its `scale`. The value of the other input can always be calculated from them in the `render()` method. This lets us clear or apply rounding to the other field without losing any precision in the user input.

When you see something wrong in the UI, you can use React Developer Tools to inspect the props and move up the tree until you find the component responsible for updating the state. This lets you trace the bugs to their source:

Monitoring State in React DevTools

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

In this section, we will consider a few problems where developers new to React often reach for inheritance, and show how we can solve them with composition.

Containment

Some components don't know their children ahead of time. This is especially common for components like `Sidebar` or `Dialog` that represent generic "boxes".

We recommend that such components use the special `children` prop to pass children elements directly into their output:

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

This lets other components pass arbitrary children to them by nesting the JSX:

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

Try it on CodePen.

Anything inside the `<FancyBorder>` JSX tag gets passed into the `FancyBorder` component as a `children` prop. Since `FancyBorder` renders `{props.children}` inside a `<div>`, the passed elements appear in the final output.

While this is less common, sometimes you might need multiple "holes" in a component. In such cases you may come up with your own convention instead of using `children`:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

Try it on CodePen.

React elements like `<Contacts />` and `<Chat />` are just objects, so you can pass them as props like any other data.

Specialization

Sometimes we think about components as being "special cases" of other components. For example, we might say that a `WelcomeDialog` is a special case of `Dialog`.

In React, this is also achieved by composition, where a more "specific" component renders a more "generic" one and configures it with props:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
```

```
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}
```

Try it on CodePen.

Composition works equally well for components defined as classes:

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
              message="How should we refer to you?">
        <input value={this.state.login}
               onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
```

```
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

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

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}
```

Try it on CodePen.

So What About Inheritance?

At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.

Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

If you want to reuse non-UI functionality between components, we suggest extracting it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.

React is, in our opinion, the premier way to build big, fast Web apps with JavaScript. It has scaled very well for us at Facebook and Instagram.

One of the many great parts of React is how it makes you think about apps as you build them. In this document, we'll walk you through the thought process of building a searchable product data table using React.

Start With A Mock

Imagine that we already have a JSON API and a mock from our designer. The mock looks like this:

Our JSON API returns some data that looks like this:

```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"
},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

Step 1: Break The UI Into A Component Hierarchy

The first thing you'll want to do is to draw boxes around every component (and subcomponent) in the mock and give them all names. If you're working with a designer, they may have already done this, so go talk to them! Their Photoshop layer names may end up being the names of your React components!

But how do you know what should be its own component? Just use the same techniques for deciding if you should create a new function or object. One such technique is the single responsibility principle, that is, a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

Since you're often displaying a JSON data model to a user, you'll find that if your model was built correctly, your UI (and therefore your component structure) will map nicely. That's because UI and data models tend to adhere to the same *information architecture*, which means the work of separating your UI into components is often trivial. Just break it up into components that represent exactly one piece of your data model.

You'll see here that we have five components in our simple app. We've italicized the data each component represents.

1. `FilterableProductTable` **(orange):** contains the entirety of the example
2. `SearchBar` **(blue):** receives all *user input*
3. `ProductTable` **(green):** displays and filters the *data collection* based on *user input*
4. `ProductCategoryRow` **(turquoise):** displays a heading for each *category*
5. `ProductRow` **(red):** displays a row for each *product*

If you look at `ProductTable`, you'll see that the table header (containing the "Name" and "Price" labels) isn't its own component. This is a matter of preference, and there's an argument to be made either way. For this example, we left it as part of `ProductTable` because it is part of rendering the *data collection* which is `ProductTable`'s responsibility. However, if this header grows to be complex (i.e. if we were to add affordances for sorting), it would certainly make sense to make this its own `ProductTableHeader` component.

Now that we've identified the components in our mock, let's arrange them into a hierarchy. This is easy. Components that appear within another component in the mock should appear as a child in the hierarchy:

- `FilterableProductTable`
  - `SearchBar`
  - `ProductTable`
    - `ProductCategoryRow`
    - `ProductRow`

Step 2: Build A Static Version in React

See the Pen Thinking In React: Step 2 on CodePen.

Now that you have your component hierarchy, it's time to implement your app. The easiest way is to build a version that takes your data model and renders the UI but has no interactivity. It's best to decouple these processes because building a static version requires a lot of typing and no thinking, and adding interactivity requires a lot of thinking and not a lot of typing. We'll see why.

To build a static version of your app that renders your data model, you'll want to build components that reuse other components and pass data using *props*. *props* are a way of passing data from parent to child. If you're familiar with the concept of *state*, **don't use state at all** to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don't need it.

You can build top-down or bottom-up. That is, you can either start with building the components higher up in the hierarchy (i.e. starting with `FilterableProductTable`) or with the ones lower in it (`ProductRow`). In simpler examples, it's usually easier to go top-down, and on larger projects, it's easier to go bottom-up and write tests as you build.

At the end of this step, you'll have a library of reusable components that render your data model. The components will only have `render()` methods since this is a static version of your app. The component at the top of the hierarchy (`FilterableProductTable`) will take your data model as a prop. If you make a change to your underlying data model and call `ReactDOM.render()` again, the UI will be updated. It's easy to see how your UI is updated and where to make changes since there's nothing complicated going on. React's **one-way data flow** (also called *one-way binding*) keeps everything modular and fast.

Simply refer to the React docs if you need help executing this step.

A Brief Interlude: Props vs State

There are two types of "model" data in React: props and state. It's important to understand the distinction between the two; skim the official React docs if you aren't sure what the difference is.

Step 3: Identify The Minimal (but complete) Representation Of UI State

To make your UI interactive, you need to be able to trigger changes to your underlying data model. React makes this easy with **state**.

To build your app correctly, you first need to think of the minimal set of mutable state that your app needs. The key here is DRY: *Don't Repeat Yourself*. Figure out the absolute minimal representation of the state your application needs and compute everything else you need on-demand. For example, if you're building a TODO list, just keep an array of the TODO items around; don't keep a separate state variable for the count. Instead, when you want to render the TODO count, simply take the length of the TODO items array.

Think of all of the pieces of data in our example application. We have:

- The original list of products
- The search text the user has entered
- The value of the checkbox
- The filtered list of products

Let's go through each one and figure out which one is state. Simply ask three questions about each piece of data:

1. Is it passed in from a parent via props? If so, it probably isn't state.
2. Does it remain unchanged over time? If so, it probably isn't state.
3. Can you compute it based on any other state or props in your component? If so, it isn't state.

The original list of products is passed in as props, so that's not state. The search text and the checkbox seem to be state since they change over time and can't be computed from anything. And finally, the filtered list of products isn't state because it can be computed by combining the original list of products with the search text and value of the checkbox.

So finally, our state is:

- The search text the user has entered
- The value of the checkbox

Step 4: Identify Where Your State Should Live

See the Pen Thinking In React: Step 4 by Kevin Lacker (@lacker) on CodePen.

OK, so we've identified what the minimal set of app state is. Next, we need to identify which component mutates, or *owns*, this state.

Remember: React is all about one-way data flow down the component hierarchy. It may not be immediately clear which component should own what state. **This is often the most challenging part for newcomers to understand,** so follow these steps to figure it out:

For each piece of state in your application:

- Identify every component that renders something based on that state.
- Find a common owner component (a single component above all the components that need the state in the hierarchy).
- Either the common owner or another component higher up in the hierarchy should own the state.
- If you can't find a component where it makes sense to own the state, create a new component simply for holding the state and add it somewhere in the hierarchy above the common owner component.

Let's run through this strategy for our application:

- `ProductTable` needs to filter the product list based on state and `SearchBar` needs to display the search text and checked state.
- The common owner component is `FilterableProductTable`.
- It conceptually makes sense for the filter text and checked value to live in `FilterableProductTable`

Cool, so we've decided that our state lives in `FilterableProductTable`. First, add an instance property `this.state = {filterText: '', inStockOnly: false}` to `FilterableProductTable`'s `constructor` to reflect the initial state of your application. Then, pass `filterText` and `inStockOnly` to `ProductTable` and `SearchBar` as a prop. Finally, use these props to filter the rows in `ProductTable` and set the values of the form fields in `SearchBar`.

You can start seeing how your application will behave: set `filterText` to `"ball"` and refresh your app. You'll see that the data table is updated correctly.

Step 5: Add Inverse Data Flow

See the Pen Thinking In React: Step 5 on CodePen.

So far, we've built an app that renders correctly as a function of props and state flowing down the hierarchy. Now it's time to support data flowing the other way: the form components deep in the hierarchy need to update the state in `FilterableProductTable`.

React makes this data flow explicit to make it easy to understand how your program works, but it does require a little more typing than traditional two-way data binding.

If you try to type or check the box in the current version of the example, you'll see that React ignores your input. This is intentional, as we've set the `value` prop of the `input` to always be equal to the `state` passed in from `FilterableProductTable`.

Let's think about what we want to happen. We want to make sure that whenever the user changes the form, we update the state to reflect the user input. Since components should only update their own state, `FilterableProductTable` will pass callbacks to `SearchBar` that will fire whenever the state should be updated. We can use the `onChange` event on the inputs to be notified of it. The callbacks passed by `FilterableProductTable` will call `setState()`, and the app will be updated.

Though this sounds complex, it's really just a few lines of code. And it's really explicit how your data is flowing throughout the app.

And That's It

Hopefully, this gives you an idea of how to think about building components and applications with React. While it may be a little more typing than you're used to, remember that code is read far more than it's written, and it's extremely easy to read this modular, explicit code. As you start to build large libraries of components, you'll appreciate this explicitness and modularity, and with code reuse, your lines of code will start to shrink. :)

Fundamentally, JSX just provides syntactic sugar for the `React.createElement(component, props, ...children)` function. The JSX code:

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

compiles into:

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

You can also use the self-closing form of the tag if there are no children. So:

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

compiles into:

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

If you want to test out how some specific JSX is converted into JavaScript, you can try out the online Babel compiler%20%7B%0A%20%20return%20%3Cdiv%3EHello%20world!%3C%2Fdiv%3E%3B%0A%7D).

Specifying The React Element Type

The first part of a JSX tag determines the type of the React element.

Capitalized types indicate that the JSX tag is referring to a React component. These tags get compiled into a direct reference to the named variable, so if you use the JSX `<Foo />` expression, `Foo` must be in scope.

React Must Be in Scope

Since JSX compiles into calls to `React.createElement`, the `React` library must also always be in scope from your JSX code.

For example, both of the imports are necessary in this code, even though `React` and `CustomButton` are not directly referenced from JavaScript:

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

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

If you don't use a JavaScript bundler and added React as a script tag, it is already in scope as a `React` global.

Using Dot Notation for JSX Type

You can also refer to a React component using dot-notation from within JSX. This is convenient if you have a single module that exports many React components. For example, if `MyComponents.DatePicker` is a component, you can use it directly from JSX with:

```
import React from 'react';

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

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

User-Defined Components Must Be Capitalized

When an element type starts with a lowercase letter, it refers to a built-in component like `<div>` or `<span>` and results in a string `'div'` or `'span'` passed to `React.createElement`. Types that start with a capital letter like `<Foo />` compile to

`React.createElement(Foo)` and correspond to a component defined or imported in your JavaScript file.

We recommend naming components with a capital letter. If you do have a component that starts with a lowercase letter, assign it to a capitalized variable before using it in JSX.

For example, this code will not run as expected:

```
import React from 'react';

// Wrong! This is a component and should have been capitalized:
function hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Wrong! React thinks <hello /> is an HTML tag because it's not capitalized:
  return <hello toWhat="World" />;
}
```

To fix this, we will rename `hello` to `Hello` and use `<Hello />` when referring to it:

```
import React from 'react';

// Correct! This is a component and should be capitalized:
function Hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Correct! React knows <Hello /> is a component because it's capitalized.
  return <Hello toWhat="World" />;
}
```

Choosing the Type at Runtime

You cannot use a general expression as the React element type. If you do want to use a general expression to indicate the type of the element, just assign it to a capitalized variable first. This often comes up when you want to render a different component based on a prop:

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

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

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

function Story(props) {
  // Wrong! JSX type can't be an expression.
  return <components[props.storyType] story={props.story} />;
}
```

To fix this, we will assign the type to a capitalized variable first:

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

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

function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions

You can pass any JavaScript expression as a prop, by surrounding it with `{}`. For example, in this JSX:

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

For `MyComponent`, the value of `props.foo` will be `10` because the expression `1 + 2 + 3 + 4` gets evaluated.

`if` statements and `for` loops are not expressions in JavaScript, so they can't be used in JSX directly. Instead, you can put these in the surrounding code. For example:

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

String Literals

You can pass a string literal as a prop. These two JSX expressions are equivalent:

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

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

When you pass a string literal, its value is HTML-unescaped. So these two JSX expressions are equivalent:

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

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

This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default to "True"

If you pass no value for a prop, it defaults to `true`. These two JSX expressions are equivalent:

```
<MyTextBox autocomplete />

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

In general, we don't recommend using this because it can be confused with the ES6 object shorthand `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. This behavior is just there so that it matches the behavior of HTML.

Spread Attributes

If you already have `props` as an object, and you want to pass it in JSX, you can use `...` as a "spread" operator to pass the whole props object. These two components are equivalent:

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

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

Spread attributes can be useful when you are building generic containers. However, they can also make your code messy by making it easy to pass a lot of irrelevant props to components that don't care about them. We recommend that you use this syntax sparingly.

Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: `props.children`. There are several different ways to pass children:

String Literals

You can put a string between the opening and closing tags and `props.children` will just be that string. This is useful for many of the built-in HTML elements. For example:

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

This is valid JSX, and `props.children` in `MyComponent` will simply be the string `"Hello world!"`. HTML is unescaped, so you can generally write JSX just like you would write HTML in this way:

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

JSX removes whitespace at the beginning and ending of a line. It also removes blank lines. New lines adjacent to tags are removed; new lines that occur in the middle of string literals are condensed into a single space. So these all render to the same thing:

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

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

<div>

  Hello World
</div>
```

JSX Children

You can provide more JSX elements as the children. This is useful for displaying nested components:

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

You can mix together different types of children, so you can use string literals together with JSX children. This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
  Here is a list:
  <ul>
```

```
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

A React component can't return multiple React elements, but a single JSX expression can have multiple children, so if you want a component to render multiple things you can wrap it in a `div` like this.

JavaScript Expressions

You can pass any JavaScript expression as children, by enclosing it within `{}`. For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>

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

This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

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

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

JavaScript expressions can be mixed with other types of children. This is often useful in lieu of string templates:

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

Functions as Children

Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things. However, `props.children` works just like any other prop in that it can pass any sort of data, not just the sorts that React knows how to render. For example, if you have a custom component, you could have it take a callback as `props.children`:

```
// Calls the children callback numTimes to produce a repeated component
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

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

Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering. This usage is not common, but it works if you want to stretch what JSX is capable of.

Booleans, Null, and Undefined Are Ignored

`false`, `null`, `undefined`, and `true` are valid children. They simply don't render. These JSX expressions will all render to the same thing:

```
<div />

<div></div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

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

This can be useful to conditionally render React elements. This JSX only renders a `<Header />` if `showHeader` is `true`:

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

One caveat is that some "falsy" values, such as the `0` number, are still rendered by React. For example, this code will not behave as you might expect because `0` will be printed when `props.messages` is an empty array:

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

To fix this, make sure that the expression before `&&` is always boolean:

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

Conversely, if you want a value like `false`, `true`, `null`, or `undefined` to appear in the output, you have to convert it to a string first:

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

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like Flow or TypeScript to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special `propTypes` property:

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

Greeting.propTypes = {
  name: React.PropTypes.string
};
```

`React.PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. In this example, we're using `React.PropTypes.string`. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, `propTypes` is only checked in development mode.

React.PropTypes

Here is an example documenting the different validators provided:

```
MyComponent.propTypes = {
  // You can declare that a prop is a specific JS primitive. By default, these
  // are all optional.
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol,

  // Anything that can be rendered: numbers, strings, elements or an array
  // (or fragment) containing these types.
  optionalNode: React.PropTypes.node,

  // A React element.
  optionalElement: React.PropTypes.element,

  // You can also declare that a prop is an instance of a class. This uses
```

```
  // JS's instanceof operator.
  optionalMessage: React.PropTypes.instanceOf(Message),

  // You can ensure that your prop is limited to specific values by treating
  // it as an enum.
  optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

  // An object that could be one of many types
  optionalUnion: React.PropTypes.oneOfType([
    React.PropTypes.string,
    React.PropTypes.number,
    React.PropTypes.instanceOf(Message)
  ]),

  // An array of a certain type
  optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),

  // An object with property values of a certain type
  optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number),

  // An object taking on a particular shape
  optionalObjectWithShape: React.PropTypes.shape({
    color: React.PropTypes.string,
    fontSize: React.PropTypes.number
  }),

  // You can chain any of the above with `isRequired` to make sure a warning
  // is shown if the prop isn't provided.
  requiredFunc: React.PropTypes.func.isRequired,

  // A value of any data type
  requiredAny: React.PropTypes.any.isRequired,

  // You can also specify a custom validator. It should return an Error
  // object if the validation fails. Don't `console.warn` or throw, as this
  // won't work inside `oneOfType`.
  customProp: function(props, propName, componentName) {
    if (!/matchme/.test(props[propName])) {
      return new Error(
        'Invalid prop `' + propName + '` supplied to' +
        ' `' + componentName + '`. Validation failed.'
      );
    }
  },

  // You can also supply a custom validator to `arrayOf` and `objectOf`.
  // It should return an Error object if the validation fails. The validator
  // will be called for each key in the array or object. The first two
  // arguments of the validator are the array or object itself, and the
  // current item's key.
  customArrayProp: React.PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
```

```
    if (!/matchme/.test(propValue[key])) {
      return new Error(
        'Invalid prop `' + propFullName + '` supplied to' +
        ' `' + componentName + '`. Validation failed.'
      );
    }
  })
};
```

Requiring Single Child

With `React.PropTypes.element` you can specify that only a single child can be passed to a component as children.

```
class MyComponent extends React.Component {
  render() {
    // This must be exactly one element or it will warn.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: React.PropTypes.element.isRequired
};
```

Default Prop Values

You can define default values for your `props` by assigning to the special `defaultProps` property:

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

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
```

```
};

// Renders "Hello, Stranger":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

The `defaultProps` will be used to ensure that `this.props.name` will have a value if it was not specified by the parent component. The `propTypes` typechecking happens after `defaultProps` are resolved, so typechecking will also apply to the `defaultProps`.

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

For example, instead of exposing `open()` and `close()` methods on a `Dialog` component, pass an `isOpen` prop to it.

Adding a Ref to a DOM Element

React supports a special attribute that you can attach to any component. The `ref` attribute takes a callback function, and the callback will be executed immediately after the component is mounted or unmounted.

When the `ref` attribute is used on an HTML element, the `ref` callback receives the underlying DOM element as its argument. For example, this code uses the `ref` callback to store a reference to a DOM node:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.focus = this.focus.bind(this);
  }

  focus() {
    // Explicitly focus the text input using the raw DOM API
    this.textInput.focus();
  }

  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
```

```
        <div>
          <input
            type="text"
            ref={(input) => { this.textInput = input; }} />
          <input
            type="button"
            value="Focus the text input"
            onClick={this.focus}
          />
        </div>
      );
    }
  }
```

React will call the `ref` callback with the DOM element when the component mounts, and call it with `null` when it unmounts.

Using the `ref` callback just to set a property on the class is a common pattern for accessing DOM elements. The preferred way is to set the property in the `ref` callback like in the above example. There is even a shorter way to write it: `ref={input => this.textInput = input}`.

Adding a Ref to a Class Component

When the `ref` attribute is used on a custom component declared as a class, the `ref` callback receives the mounted instance of the component as its argument. For example, if we wanted to wrap the `CustomTextInput` above to simulate it being clicked immediately after mounting:

```
class AutoFocusTextInput extends React.Component {
  componentDidMount() {
    this.textInput.focus();
  }

  render() {
    return (
      <CustomTextInput
        ref={(input) => { this.textInput = input; }} />
    );
  }
}
```

Note that this only works if `CustomTextInput` is declared as a class:

```
class CustomTextInput extends React.Component {
```

```
  // ...
}
```

Refs and Functional Components

You may not use the `ref` attribute on functional components because they don't have instances:

```
function MyFunctionalComponent() {
  return <input />;
}

class Parent extends React.Component {
  render() {
    // This will *not* work!
    return (
      <MyFunctionalComponent
        ref={(input) => { this.textInput = input; }} />
    );
  }
}
```

You should convert the component to a class if you need a ref to it, just like you do when you need lifecycle methods or state.

You can, however, **use the `ref` attribute inside a functional component** as long as you refer to a DOM element or a class component:

```
function CustomTextInput(props) {
  // textInput must be declared here so the ref callback can refer to it
  let textInput = null;

  function handleClick() {
    textInput.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={(input) => { textInput = input; }} />
      <input
        type="button"
        value="Focus the text input"
        onClick={handleClick}
      />
    </div>
```

```
    );
  }
```

Don't Overuse Refs

Your first inclination may be to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy. Often, it becomes clear that the proper place to "own" that state is at a higher level in the hierarchy. See the Lifting State Up guide for examples of this.

Legacy API: String Refs

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `"textInput"`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because string refs have some issues, are considered legacy, and **are likely to be removed in one of the future releases**. If you're currently using `this.refs.textInput` to access refs, we recommend the callback pattern instead.

Caveats

If the `ref` callback is defined as an inline function, it will get called twice during updates, first with `null` and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one. You can avoid this by defining the `ref` callback as a bound method on the class, but note that it shouldn't matter in most cases.

In most cases, we recommend using controlled components to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.

For example, this code accepts a single name in an uncontrolled component:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={(input) => this.input = input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Try it on CodePen.

Since an uncontrolled component keeps the source of truth in the DOM, it is sometimes easier to integrate React and non-React code when using uncontrolled components. It can also be slightly less code if you want to be quick and dirty. Otherwise, you should usually use controlled components.

If it's still not clear which type of component you should use for a particular situation, you might find this article on controlled versus uncontrolled inputs to be helpful.

Default Values

In the React rendering lifecycle, the `value` attribute on form elements will override the value in the DOM. With an uncontrolled component, you often want React to specify the initial value, but leave subsequent updates uncontrolled. To handle this case, you can specify a `defaultValue` attribute instead of `value`.

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
          defaultValue="Bob"
          type="text"
          ref={(input) => this.input = input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

Likewise, `<input type="checkbox">` and `<input type="radio">` support `defaultChecked`, and `<select>` supports `defaultValue`.

Internally, React uses several clever techniques to minimize the number of costly DOM operations required to update the UI. For many applications, using React will lead to a fast user interface without doing much work to specifically optimize for performance. Nevertheless, there are several ways you can speed up your React application.

Use The Production Build

If you're benchmarking or experiencing performance problems in your React apps, make sure you're testing with the minified production build:

- For Create React App, you need to run `npm run build` and follow the instructions.
- For single-file builds, we offer production-ready `.min.js` versions.
- For Browserify, you need to run it with `NODE_ENV=production`.
- For Webpack, you need to add this to plugins in your production config:

```
new webpack.DefinePlugin({
  'process.env': {
    NODE_ENV: JSON.stringify('production')
  }
}),
new webpack.optimize.UglifyJsPlugin()
```

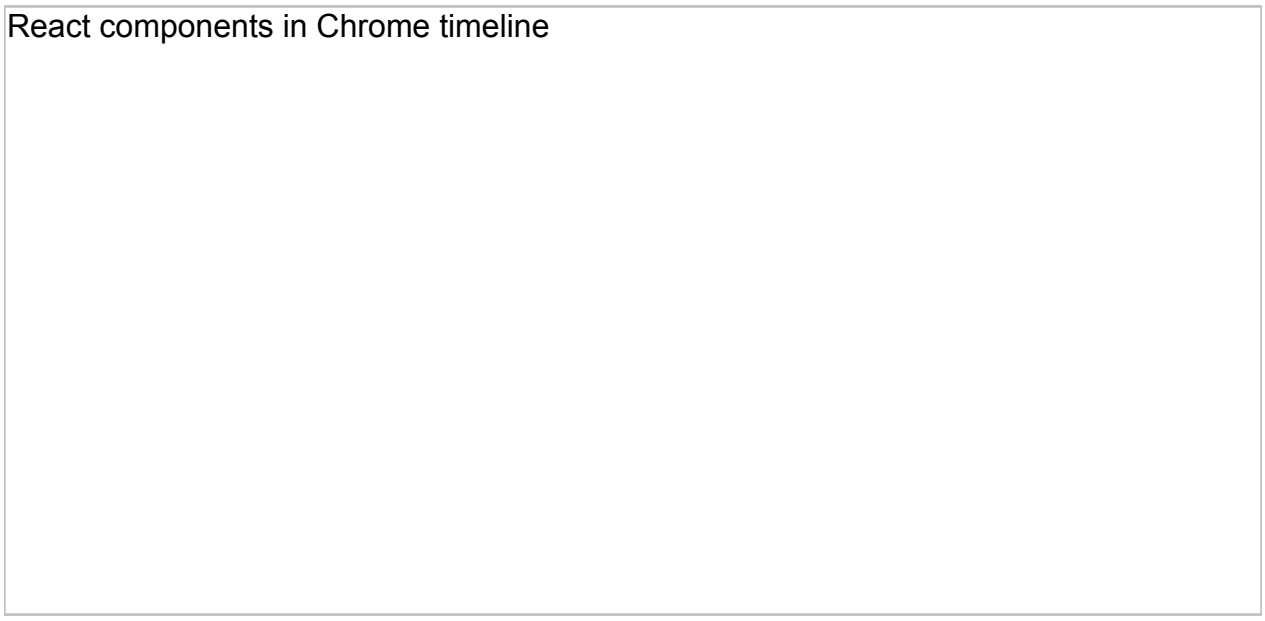
- For Rollup, you need to use the replace plugin *before* the commonjs plugin so that development-only modules are not imported. For a complete setup example see this gist.

```
plugins: [
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  // ...
]
```

The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

Profiling Components with Chrome Timeline

In the **development** mode, you can visualize how components mount, update, and unmount, using the performance tools in supported browsers. For example:



To do this in Chrome:

1. Load your app with `?react_perf` in the query string (for example, `http://localhost:3000/?react_perf`).
2. Open the Chrome DevTools **Timeline** tab and press **Record**.
3. Perform the actions you want to profile. Don't record more than 20 seconds or Chrome might hang.
4. Stop recording.
5. React events will be grouped under the **User Timing** label.

Note that **the numbers are relative so components will render faster in production**. Still, this should help you realize when unrelated UI gets updated by mistake, and how deep and how often your UI updates occur.

Currently Chrome, Edge, and IE are the only browsers supporting this feature, but we use the standard User Timing API so we expect more browsers to add support for it.

Avoid Reconciliation

React builds and maintains an internal representation of the rendered UI. It includes the React elements you return from your components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be

slower than operations on JavaScript objects. Sometimes it is referred to as a "virtual DOM", but it works the same way on React Native.

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM.

In some cases, your component can speed all of this up by overriding the lifecycle function `shouldComponentUpdate`, which is triggered before the re-rendering process starts. The default implementation of this function returns `true`, leaving React to perform the update:

```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

If you know that in some situations your component doesn't need to update, you can return `false` from `shouldComponentUpdate` instead, to skip the whole rendering process, including calling `render()` on this component and below.

shouldComponentUpdate In Action

Here's a subtree of components. For each one, `SCU` indicates what `shouldComponentUpdate` returned, and `vDOMEq` indicates whether the rendered React elements were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.

Since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React did not attempt to render C2, and thus didn't even have to invoke `shouldComponentUpdate` on C4 and C5.

For C1 and C3, `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 `shouldComponentUpdate` returned `true`, and since the rendered elements weren't equivalent React had to update the DOM.

The last interesting case is C8. React had to render this component, but since the React elements it returned were equal to the previously rendered ones, it didn't have to update the DOM.

Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the rendered React elements, and for C2's subtree and C7, it didn't even have to compare the elements as we bailed out on `shouldComponentUpdate`, and `render` was not called.

Examples

If the only way your component ever changes is when the `props.color` or the `state.count` variable changes, you could have `shouldComponentUpdate` check that:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

In this code, `shouldComponentUpdate` is just checking if there is any change in `props.color` or `state.count`. If those values don't change, the component doesn't update. If your component got more complex, you could use a similar pattern of doing a "shallow comparison" between all the fields of `props` and `state` to determine if the component should update. This pattern is common enough that React provides a helper to use this logic - just inherit from `React.PureComponent`. So this code is a simpler way to achieve the same thing:

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Most of the time, you can use `React.PureComponent` instead of writing your own `shouldComponentUpdate`. It only does a shallow comparison, so you can't use it if the props or state may have been mutated in a way that a shallow comparison would miss.

This can be a problem with more complex data structures. For example, let's say you want a `ListOfWords` component to render a comma-separated list of words, with a parent `WordAdder` component that lets you click a button to add a word to the list. This code does *not* work correctly:

```
class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }
```

```
  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}
```

The problem is that `PureComponent` will do a simple comparison between the old and new values of `this.props.words`. Since this code mutates the `words` array in the `handleClick` method of `WordAdder`, the old and new values of `this.props.words` will compare as equal, even though the actual words in the array have changed. The `ListOfWords` will thus not update even though it has new words that should be rendered.

The Power Of Not Mutating Data

The simplest way to avoid this problem is to avoid mutating values that you are using as props or state. For example, the `handleClick` method above could be rewritten using `concat` as:

```
handleClick() {
  this.setState(prevState => ({
    words: prevState.words.concat(['marklar'])
  }));
}
```

ES6 supports a spread syntax for arrays which can make this easier. If you're using Create React App, this syntax is available by default.

```
handleClick() {
  this.setState(prevState => ({
    words: [...prevState.words, 'marklar'],
  }));
};
```

You can also rewrite code that mutates objects to avoid mutation, in a similar way. For example, let's say we have an object named `colormap` and we want to write a function that changes `colormap.right` to be `'blue'`. We could write:

```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

To write this without mutating the original object, we can use Object.assign method:

```
function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}
```

`updateColorMap` now returns a new object, rather than mutating the old one. `Object.assign` is in ES6 and requires a polyfill.

There is a JavaScript proposal to add object spread properties to make it easier to update objects without mutation as well:

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

If you're using Create React App, both `Object.assign` and the object spread syntax are available by default.

Using Immutable Data Structures

Immutable.js is another way to solve this problem. It provides immutable, persistent collections that work via structural sharing:

- *Immutable*: once created, a collection cannot be altered at another point in time.
- *Persistent*: new collections can be created from a previous collection and a mutation such as set. The original collection is still valid after the new collection is created.
- *Structural Sharing*: new collections are created using as much of the same structure as the original collection as possible, reducing copying to a minimum to improve performance.

Immutability makes tracking changes cheap. A change will always result in a new object so we only need to check if the reference to the object has changed. For example, in this regular JavaScript code:

```
const x = { foo: "bar" };
```

```
const y = x;
y.foo = "baz";
x === y; // true
```

Although `y` was edited, since it's a reference to the same object as `x`, this comparison returns `true`. You can write similar code with immutable.js:

```
const SomeRecord = Immutable.Record({ foo: null });
const x = new SomeRecord({ foo: 'bar'  });
const y = x.set('foo', 'baz');
x === y; // false
```

In this case, since a new reference is returned when mutating `x`, we can safely assume that `x` has changed.

Two other libraries that can help use immutable data are seamless-immutable and immutability-helper.

Immutable data structures provide you with a cheap way to track changes on objects, which is all we need to implement `shouldComponentUpdate`. This can often provide you with a nice performance boost.

Normally you would define a React component as a plain JavaScript class:

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

If you don't use ES6 yet, you may use the `React.createClass` helper instead:

```
var Greeting = React.createClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

The API of ES6 classes is similar to `React.createClass` with a few exceptions.

Declaring Prop Types and Default Props

With functions and ES6 classes, `propTypes` and `defaultProps` are defined as properties on the components themselves:

```
class Greeting extends React.Component {
  // ...
}

Greeting.propTypes = {
  name: React.PropTypes.string
};

Greeting.defaultProps = {
  name: 'Mary'
};
```

With `React.createClass()`, you need to define `propTypes` as a property on the passed object, and `getDefaultProps()` as a function on it:

```
var Greeting = React.createClass({
  propTypes: {
    name: React.PropTypes.string
  },
```

```
  getDefaultProps: function() {
    return {
      name: 'Mary'
    };
  },

  // ...

});
```

Setting the Initial State

In ES6 classes, you can define the initial state by assigning `this.state` in the constructor:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

With `React.createClass()`, you have to provide a separate `getInitialState` method that returns the initial state:

```
var Counter = React.createClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

Autobinding

In React components declared as ES6 classes, methods follow the same semantics as regular ES6 classes. This means that they don't automatically bind `this` to the instance. You'll have to explicitly use `.bind(this)` in the constructor:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = {message: 'Hello!'};
    // This line is important!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // Because `this.handleClick` is bound, we can use it as an event handler.
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

With `React.createClass()`, this is not necessary because it binds all methods:

```
var SayHello = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
});
```

This means writing ES6 classes comes with a little more boilerplate code for event handlers, but the upside is slightly better performance in large applications.

If the boilerplate code is too unattractive to you, you may enable the **experimental** Class Properties syntax proposal with Babel:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = {message: 'Hello!'};
  }
  // WARNING: this syntax is experimental!
  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

Please note that the syntax above is **experimental** and the syntax may change, or the proposal might not make it into the language.

If you'd rather play it safe, you have a few options:

- Bind methods in the constructor.
- Use arrow functions, e.g. `onClick={(e) => this.handleClick(e)}`.
- Keep using `React.createClass()`.

Mixins

> **Note:**
>
> ES6 launched without any mixin support. Therefore, there is no support for mixins when you use React with ES6 classes.
>
> **We also found numerous issues in codebases using mixins, and don't recommend using them in the new code.**
>
> This section exists only for the reference.

Sometimes very different components may share some common functionality. These are sometimes called cross-cutting concerns. `React.createClass` lets you use a legacy `mixins` system for that.

One common use case is a component wanting to update itself on a time interval. It's easy to use `setInterval()`, but it's important to cancel your interval when you don't need it anymore to save memory. React provides lifecycle methods that let you know

when a component is about to be created or destroyed. Let's create a simple mixin that uses these methods to provide an easy `setInterval()` function that will automatically get cleaned up when your component is destroyed.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var TickTock = React.createClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

ReactDOM.render(
  <TickTock />,
  document.getElementById('example')
);
```

If a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

JSX is not a requirement for using React. Using React without JSX is especially convenient when you don't want to set up compilation in your build environment.

Each JSX element is just syntactic sugar for calling `React.createElement(component, props, ...children)`. So, anything you can do with JSX can also be done with just plain JavaScript.

For example, this code written with JSX:

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

can be compiled to this code that does not use JSX:

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

If you're curious to see more examples of how JSX is converted to JavaScript, you can try out the online Babel compiler%20%7B%0A%20%20return%20%3Cdiv%3EHello%20world!%3C%2Fdiv%3E%3B%0A%7D).

The component can either be provided as a string, or as a subclass of `React.Component`, or a plain function for stateless components.

If you get tired of typing `React.createElement` so much, one common pattern is to assign a shorthand:

```
const e = React.createElement;
```

```
ReactDOM.render(
  e('div', null, 'Hello World'),
  document.getElementById('root')
);
```

If you use this shorthand form for `React.createElement`, it can be almost as convenient to use React without JSX.

React provides a declarative API so that you don't have to worry about exactly what changes on every update. This makes writing applications a lot easier, but it might not be obvious how this is implemented within React. This article explains the choices we made in React's "diffing" algorithm so that component updates are predictable while being fast enough for high-performance apps.

Motivation

When you use React, at a single point in time you can think of the `render()` function as creating a tree of React elements. On the next state or props update, that `render()` function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the state of the art algorithms have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic O(n) algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a `key` prop.

In practice, these assumptions are valid for almost all practical use cases.

The Diffing Algorithm

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements.

Elements Of Different Types

Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from `<a>` to `<img>`, or from `<Article>` to `<Comment>`, or from `<Button>` to `<div>` - any of those will lead to a full rebuild.

When tearing down a tree, old DOM nodes are destroyed. Component instances receive `componentWillUnmount()`. When building up a new tree, new DOM nodes are inserted into the DOM. Component instances receive `componentWillMount()` and then `componentDidMount()`. Any state associated with the old tree is lost.

Any components below the root will also get unmounted and have their state destroyed. For example, when diffing:

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

This will destroy the old `Counter` and remount a new one.

DOM Elements Of The Same Type

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

```
<div className="before" title="stuff" />

<div className="after" title="stuff" />
```

By comparing these two elements, React knows to only modify the `className` on the underlying DOM node.

When updating `style`, React also knows to update only the properties that changed. For example:

```
<div style={{'{{'}}color: 'red', fontWeight: 'bold'}} />

<div style={{'{{'}}color: 'green', fontWeight: 'bold'}} />
```

When converting between these two elements, React knows to only modify the `color` style, not the `fontWeight`.

After handling the DOM node, React then recurses on the children.

Component Elements Of The Same Type

When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls `componentWillReceiveProps()` and `componentWillUpdate()` on the underlying instance.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

Recursing On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will match the two `<li>first</li>` trees, match the two `<li>second</li>` trees, and then insert the `<li>third</li>` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
```

```
  <li>Villanova</li>
</ul>
```

React will mutate every child instead of realizing it can keep the `<li>Duke</li>` and `<li>Villanova</li>` subtrees intact. This inefficiency can be a problem.

Keys

In order to solve this issue, React supports a `key` attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a `key` to our inefficient example above can make the tree conversion efficient:

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Now React knows that the element with key `'2014'` is the new one, and the elements with the keys `'2015'` and `'2016'` have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

```
<li key={item.id}>{item.name}</li>
```

When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. The key only has to be unique among its siblings, not globally unique.

As a last resort, you can pass item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

Tradeoffs

It is important to remember that the reconciliation algorithm is an implementation detail. React could rerender the whole app on every action; the end result would be the same. We are regularly refining the heuristics in order to make common use cases faster.

In the current implementation, you can express the fact that a subtree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will rerender that full subtree.

Because React relies on heuristics, if the assumptions behind them are not met, performance will suffer.

1. The algorithm will not try to match subtrees of different component types. If you see yourself alternating between two component types with very similar output, you may want to make it the same type. In practice, we haven't found this to be an issue.
2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by `Math.random()`) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

With React, it's easy to track the flow of data through your React components. When you look at a component, you can see which props are being passed, which makes your apps easy to reason about.

In some cases, you want to pass data through the component tree without having to pass the props down manually at every level. You can do this directly in React with the powerful "context" API.

Why Not To Use Context

The vast majority of applications do not need to use context.

If you want your application to be stable, don't use context. It is an experimental API and it is likely to break in future releases of React.

If you aren't familiar with state management libraries like Redux or MobX, don't use context. For many practical applications, these libraries and their React bindings are a good choice for managing state that is relevant to many components. It is far more likely that Redux is the right solution to your problem than that context is the right solution.

If you aren't an experienced React developer, don't use context. There is usually a better way to implement functionality just using props and state.

If you insist on using context despite these warnings, try to isolate your use of context to a small area and avoid using the context API directly when possible so that it's easier to upgrade when the API changes.

How To Use Context

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{'{{'}}background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
```

```
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{'{{'}}background: this.context.color}}>
        {this.props.children}
      </button>
    );
  }
}

Button.contextTypes = {
  color: React.PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
```

```
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: React.PropTypes.string
};
```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Parent-Child Coupling

Context can also let you build an API where parents and children communicate. For example, one library that works this way is React Router V4:

```
import { Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React component as props if you'd like to.

Referencing Context in Lifecycle Methods

If `contextTypes` is defined within a component, the following lifecycle methods will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`
- `componentDidUpdate(prevProps, prevState, prevContext)`

Referencing Context in Stateless Functional Components

Stateless functional components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a stateless functional component.

```
const Button = ({children}, context) =>
  <button style={{'{{'}}background: context.color}}>
    {children}
  </button>;

Button.contextTypes = {color: React.PropTypes.string};
```

Updating Context

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

```
class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type:'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' :
'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };

    window.addEventListener('resize', checkMediaQuery);
    checkMediaQuery();
  }

  render() {
    return this.props.children;
  }
}

MediaQuery.childContextTypes = {
  type: React.PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. This blog post has a good explanation of why this is a problem and how you might get around it.

React and Web Components are built to solve different problems. Web Components provide strong encapsulation for reusable components, while React provides a declarative library that keeps the DOM in sync with your data. The two goals are complementary. As a developer, you are free to use React in your Web Components, or to use Web Components in React, or both.

Most people who use React don't use Web Components, but you may want to, especially if you are using third-party UI components that are written using Web Components.

Using Web Components in React

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

> Note:
>
> Web Components often expose an imperative API. For instance, a `video` Web Component might expose `play()` and `pause()` functions. To access the imperative APIs of a Web Component, you will need to use a ref to interact with the DOM node directly. If you are using third-party Web Components, the best solution is to write a React component that behaves as a wrapper for your Web Component.
>
> Events emitted by a Web Component may not properly propagate through a React render tree. You will need to manually attach event handlers to handle these events within your React components.

One common confusion is that Web Components use "class" instead of "className".

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

Using React in your Web Components

```
const proto = Object.create(HTMLElement.prototype, {
  attachedCallback: {
    value: function() {
      const mountPoint = document.createElement('span');
      this.createShadowRoot().appendChild(mountPoint);

      const name = this.getAttribute('name');
      const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
      ReactDOM.render(<a href={url}>{name}</a>, mountPoint);
    }
  }
});
document.registerElement('x-search', {prototype: proto});
```

You can also check out this complete Web Components example on GitHub.

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.

Concretely, **a higher-order component is a function that takes a component and returns a new component.**

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux's `connect` and Relay's `createContainer`.

In this document, we'll discuss why higher-order components are useful, and how to write your own.

Use HOCs For Cross-Cutting Concerns

> **Note**
>
> We previously recommended mixins as a way to handle cross-cutting concerns. We've since realized that mixins create more trouble than they are worth. Read more about why we've moved away from mixins and how you can transition your existing components.

Components are the primary unit of code reuse in React. However, you'll find that some patterns aren't a straightforward fit for traditional components.

For example, say you have a `CommentList` component that subscribes to an external data source to render a list of comments:

```
class CommentList extends React.Component {
  constructor() {
    super();
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }
```

```
  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <div>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </div>
    );
  }
}
```

Later, you write a component for subscribing to a single blog post, which follows a similar pattern:

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
```

```
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

`CommentList` and `BlogPost` aren't identical — they call different methods on `DataSource`, and they render different output. But much of their implementation is the same:

- On mount, add a change listener to `DataSource`.
- Inside the listener, call `setState` whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to `DataSource` and calling `setState` will occur over and over again. We want an abstraction that allows us to define this logic in a single place and share them across many components. This is where higher-order components excel.

We can write a function that creates components, like `CommentList` and `BlogPost`, that subscribe to `DataSource`. The function will accept as one of its arguments a child component that receives the subscribed data as a prop. Let's call the function `withSubscription`:

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
});
```

The first parameter is the wrapped component. The second parameter retrieves the data we're interested in, given a `DataSource` and the current props.

When `CommentListWithSubscription` and `BlogPostWithSubscription` are rendered, `CommentList` and `BlogPost` will be passed a `data` prop with the most current data retrieved from `DataSource`:

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

Note that an HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, an HOC *composes* the original component by *wrapping* it in a container component. An HOC is a pure function with zero side-effects.

And that's it! The wrapped component receives all the props of the container, along with a new prop, `data`, which it uses to render its output. The HOC isn't concerned with how or why the data is used, and the wrapped component isn't concerned with where the data came from.

Because `withSubscription` is a normal function, you can add as many or as few arguments as you like. For example, you may want to make the name of the `data` prop configurable, to further isolate the HOC from the wrapped component. Or you could

accept an argument that configures `shouldComponentUpdate`, or one that configures the data source. These are all possible because the HOC has full control over how the component is defined.

Like components, the contract between `withSubscription` and the wrapped component is entirely props-based. This makes it easy to swap one HOC for a different one, as long as they provide the same props to the wrapped component. This may be useful if you change data-fetching libraries, for example.

Don't Mutate the Original Component. Use Composition.

Resist the temptation to modify a component's prototype (or otherwise mutate it) inside an HOC.

```
function logProps(InputComponent) {
  InputComponent.prototype.componentWillReceiveProps(nextProps) {
    console.log('Current props: ', this.props);
    console.log('Next props: ', nextProps);
  }
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

There are a few problems with this. One is that the input component cannot be reused separately from the enhanced component. More crucially, if you apply another HOC to `EnhancedComponent` that *also* mutates `componentWillReceiveProps`, the first HOC's functionality will be overridden! This HOC also won't work with function components, which do not have lifecycle methods.

Mutating HOCs are a leaky abstraction—the consumer must know how they are implemented in order to avoid conflicts with other HOCs.

Instead of mutation, HOCs should use composition, by wrapping the input component in a container component:

```
function logProps(WrappedComponent) {
  return class extends React.Component {
```

```
    componentWillReceiveProps(nextProps) {
      console.log('Current props: ', this.props);
      console.log('Next props: ', nextProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

This HOC has the same functionality as the mutating version while avoiding the potential for clashes. It works equally well with class and functional components. And because it's a pure function, it's composable with other HOCs, or even with itself.

You may have noticed similarities between HOCs and a pattern called **container components**. Container components are part of a strategy of separating responsibility between high-level and low-level concerns. Containers manage things like subscriptions and state, and pass props to components that handle things like rendering UI. HOCs use containers as part of their implementation. You can think of HOCs as parameterized container component definitions.

Convention: Pass Unrelated Props Through to the Wrapped Component

HOCs add features to a component. They shouldn't drastically alter its contract. It's expected that the component returned from an HOC has a similar interface to the wrapped component.

HOCs should pass through props that are unrelated to its specific concern. Most HOCs contain a render method that looks something like this:

```
render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
```

```
      <WrappedComponent
        injectedProp={injectedProp}
        {...passThroughProps}
      />
    );
  }
```

This convention helps ensure that HOCs are as flexible and reusable as possible.

Convention: Maximizing Composability

Not all HOCs look the same. Sometimes they accept only a single argument, the wrapped component:

```
const NavbarWithRouter = withRouter(Navbar);
```

Usually, HOCs accept additional arguments. In this example from Relay, a config object is used to specify a component's data dependencies:

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

The most common signature for HOCs looks like this:

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)(Comment);
```

What?! If you break it apart, it's easier to see what's going on.

```
// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is an HOC, which returns a component that is connected
// to the Redux store
const ConnectedComment = enhance(CommentList);
```

In other words, `connect` is a higher-order function that returns a higher-order component!

This form may seem confusing or unnecessary, but it has a useful property. Single-argument HOCs like the one returned by the `connect` function have the signature `Component => Component`. Functions whose output type is the same as its input type are

really easy to compose together.

```
// Instead of doing this...
const EnhancedComponent = connect(commentSelector)(withRouter(WrappedComponent))

// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  connect(commentSelector),
  withRouter
)
const EnhancedComponent = enhance(WrappedComponent)
```

(This same property also allows `connect` and other enhancer-style HOCs to be used as decorators, an experimental JavaScript proposal.)

The `compose` utility function is provided by many third-party libraries including lodash (as `lodash.flowRight`), Redux, and Ramda.

Convention: Wrap the Display Name for Easy Debugging

The container components created by HOCs show up in the React Developer Tools like any other component. To ease debugging, choose a display name that communicates that it's the result of an HOC.

The most common technique is to wrap the display name of the wrapped component. So if your higher-order component is named `withSubscription`, and the wrapped component's display name is `CommentList`, use the display name `WithSubscription(CommentList)`:

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedCompone
nt)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

Caveats

Higher-order components come with a few caveats that aren't immediately obvious if you're new to React.

Don't Use HOCs Inside the render Method

React's diffing algorithm (called reconciliation) uses component identity to determine whether it should update the existing subtree or throw it away and mount a new one. If the component returned from `render` is identical (`===`) to the component from the previous render, React recursively updates the subtree by diffing it with the new one. If they're not equal, the previous subtree is unmounted completely.

Normally, you shouldn't need to think about this. But it matters for HOCs because it means you can't apply an HOC to a component within the render method of a component:

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

The problem here isn't just about performance — remounting a component causes the state of that component and all of its children to be lost.

Instead, apply HOCs outside the component definition so that the resulting component is created only once. Then, its identity will be consistent across renders. This is usually what you want, anyway.

In those rare cases where you need to apply an HOC dynamically, you can also do it inside a component's lifecycle methods or its constructor.

Static Methods Must Be Copied Over

Sometimes it's useful to define a static method on a React component. For example, Relay containers expose a static method `getFragment` to facilitate the composition of GraphQL fragments.

When you apply an HOC to a component, though, the original component is wrapped with a container component. That means the new component does not have any of the static methods of the original component.

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply an HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

To solve this, you could copy the methods onto the container before returning it:

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

However, this requires you to know exactly which methods need to be copied. You can use hoist-non-react-statics to automatically copy all non-React static methods:

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

Another possible solution is to export the static method separately from the component itself.

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

Refs Aren't Passed Through

While the convention for higher-order components is to pass through all props to the wrapped component, it's not possible to pass through refs. That's because `ref` is not really a prop — like `key`, it's handled specially by React. If you add a ref to an element whose component is the result of an HOC, the ref refers to an instance of the outermost container component, not the wrapped component.

If you find yourself facing this problem, the ideal solution is to figure out how to avoid using `ref` at all. Occasionally, users who are new to the React paradigm rely on refs in situations where a prop would work better.

That said, there are times when refs are a necessary escape hatch — React wouldn't support them otherwise. Focusing an input field is an example where you may want imperative control of a component. In that case, one solution is to pass a ref callback as a normal prop, by giving it a different name:

```
function Field({ inputRef, ...rest }) {
  return <input ref={inputRef} {...rest} />;
}

// Wrap Field in a higher-order component
const EnhancedField = enhance(Field);

// Inside a class component's render method...
<EnhancedField
  inputRef={(inputEl) => {
    // This callback gets passed through as a regular prop
    this.inputEl = inputEl
  }}
/>

// Now you can call imperative methods
this.inputEl.focus();
```

This is not a perfect solution by any means. We prefer that refs remain a library concern, rather than require you to manually handle them. We are exploring ways to solve this problem so that using an HOC is unobservable.

`React` is the entry point to the React library. If you use React as a script tag, these top-level APIs are available on the `React` global. If you use ES6 with npm, you can write `import React from 'react'`. If you use ES5 with npm, you can write `var React = require('react')`.

Overview

Components

React components let you split the UI into independent, reusable pieces, and think about each piece in isolation. React components can be defined by subclassing `React.Component` or `React.PureComponent`.

- `React.Component`
- `React.PureComponent`

If you don't use ES6 classes, you may use this helper instead.

- `createClass()`

Creating React Elements

We recommend using JSX to describe what your UI should look like. Each JSX element is just syntactic sugar for calling `React.createElement()`. You will not typically invoke the following methods directly if you are using JSX.

- `createElement()`
- `createFactory()`

See Using React without JSX for more information.

Transforming Elements

`React` also provides some other APIs:

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Typechecking with PropTypes

You can use `React.PropTypes` to run typechecking on the props for a component.

- `React.PropTypes`
- `React.PropTypes.array`
- `React.PropTypes.bool`
- `React.PropTypes.func`
- `React.PropTypes.number`
- `React.PropTypes.object`
- `React.PropTypes.string`
- `React.PropTypes.symbol`
- `React.PropTypes.node`
- `React.PropTypes.element`
- `React.PropTypes.instanceOf()`
- `React.PropTypes.oneOf()`
- `React.PropTypes.oneOfType()`
- `React.PropTypes.arrayOf()`
- `React.PropTypes.objectOf()`
- `React.PropTypes.shape()`
- `React.PropTypes.any`

Validators treat props as optional by default. You can use `isRequired` to make sure a warning is shown if the prop is not provided.

- `isRequired`

Add-Ons

If you're using `react-with-addons.js`, the React Add-Ons will be available via `React.addons`.

- `React.addons`

Reference

`React.Component`

`React.Component` is the base class for React components when they are defined using ES6 classes.

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

See the React.Component API Reference for a list of methods and properties related to the base `React.Component` class.

`React.PureComponent`

`React.PureComponent` is exactly like `React.Component` but implements `shouldComponentUpdate()` with a shallow prop and state comparison.

If your React component's `render()` function renders the same result given the same props and state, you can use `React.PureComponent` for a performance boost in some cases.

> Note
>
> `React.PureComponent`'s `shouldComponentUpdate()` only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only extend `PureComponent` when you expect to have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using immutable objects to facilitate fast comparisons of nested data.
>
> Furthermore, `React.PureComponent`'s `shouldComponentUpdate()` skips prop updates for the whole component subtree. Make sure all the children components are also "pure".

`createClass()`

```
React.createClass(specification)
```

If you don't use ES6 yet, you may use the `React.createClass()` helper instead to create a component class.

```
var Greeting = React.createClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

See Using React without ES6 for more information.

`createElement()`

```
React.createElement(
  type,
  [props],
  [...children]
)
```

Create and return a new React element of the given type. The type argument can be either a tag name string (such as `'div'` or `'span'`), or a React component type (a class or a function).

Convenience wrappers around `React.createElement()` for DOM components are provided by `React.DOM`. For example, `React.DOM.a(...)` is a convenience wrapper for `React.createElement('a', ...)`. They are considered legacy, and we encourage you to either use JSX or use `React.createElement()` directly instead.

Code written with JSX will be converted to use `React.createElement()`. You will not typically invoke `React.createElement()` directly if you are using JSX. See React Without JSX to learn more.

`cloneElement()`

```
React.cloneElement(
  element,
  [props],
  [...children]
)
```

Clone and return a new React element using `element` as the starting point. The resulting element will have the original element's props with the new props merged in shallowly. New children will replace existing children. `key` and `ref` from the original element will be preserved.

`React.cloneElement()` is almost equivalent to:

```
<element.type {...element.props} {...props}>{children}</element.type>
```

However, it also preserves `ref`s. This means that if you get a child with a `ref` on it, you won't accidentally steal it from your ancestor. You will get the same `ref` attached to your new element.

This API was introduced as a replacement of the deprecated `React.addons.cloneWithProps()`.

`createFactory()`

```
React.createFactory(type)
```

Return a function that produces React elements of a given type. Like `React.createElement()`, the type argument can be either a tag name string (such as `'div'` or `'span'`), or a React component type (a class or a function).

This helper is considered legacy, and we encourage you to either use JSX or use `React.createElement()` directly instead.

You will not typically invoke `React.createFactory()` directly if you are using JSX. See React Without JSX to learn more.

`isValidElement()`

```
React.isValidElement(object)
```

Verifies the object is a React element. Returns `true` or `false`.

`React.Children`

`React.Children` provides utilities for dealing with the `this.props.children` opaque data structure.

`React.Children.map`

```
React.Children.map(children, function[(thisArg)])
```

Invokes a function on every immediate child contained within `children` with `this` set to `thisArg`. If `children` is a keyed fragment or array it will be traversed: the function will never be passed the container objects. If children is `null` or `undefined`, returns `null` or `undefined` rather than an array.

`React.Children.forEach`

```
React.Children.forEach(children, function[(thisArg)])
```

Like `React.Children.map()` but does not return an array.

`React.Children.count`

```
React.Children.count(children)
```

Returns the total number of components in `children`, equal to the number of times that a callback passed to `map` or `forEach` would be invoked.

`React.Children.only`

```
React.Children.only(children)
```

Returns the only child in `children`. Throws otherwise.

`React.Children.toArray`

```
React.Children.toArray(children)
```

Returns the `children` opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice `this.props.children` before passing it down.

> Note:
>
> `React.Children.toArray()` changes keys to preserve the semantics of nested arrays when flattening lists of children. That is, `toArray` prefixes each key in the returned array so that each element's key is scoped to the input array containing it.

`React.PropTypes`

`React.PropTypes` exports a range of validators that can be used with a component's `propTypes` object to validate props being passed to your components.

For more information about `PropTypes`, see Typechecking with PropTypes.

`React.PropTypes.array`

```
React.PropTypes.array
```

Validates that a prop is a JavaScript array primitive.

`React.PropTypes.bool`

```
React.PropTypes.bool
```

Validates that a prop is a JavaScript bool primitive.

`React.PropTypes.func`

```
React.PropTypes.func
```

Validates that a prop is a JavaScript function.

`React.PropTypes.number`

```
React.PropTypes.number
```

Validates that a prop is a JavaScript number primitive.

`React.PropTypes.object`

```
React.PropTypes.object
```

Validates that a prop is a JavaScript object.

`React.PropTypes.string`

```
React.PropTypes.string
```

Validates that a prop is a JavaScript string primitive.

`React.PropTypes.symbol`

```
React.PropTypes.symbol
```

Validates that a prop is a JavaScript symbol.

`React.PropTypes.node`

```
React.PropTypes.node
```

Validates that a prop is anything that can be rendered: numbers, strings, elements or an array (or fragment) containing these types.

`React.PropTypes.element`

```
React.PropTypes.element
```

Validates that a prop is a React element.

`React.PropTypes.instanceOf()`

```
React.PropTypes.instanceOf(class)
```

Validates that a prop is an instance of a class. This uses JavaScript's `instanceof` operator.

`React.PropTypes.oneOf()`

```
React.PropTypes.oneOf(arrayOfValues)
```

Validates that a prop is limited to specific values by treating it as an enum.

```
MyComponent.propTypes = {
  optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),
}
```

`React.PropTypes.oneOfType()`

```
React.PropTypes.oneOfType(arrayOfPropTypes)
```

Validates that a prop is an object that could be one of many types.

```
MyComponent.propTypes = {
  optionalUnion: React.PropTypes.oneOfType([
    React.PropTypes.string,
    React.PropTypes.number,
    React.PropTypes.instanceOf(Message)
  ]),
}
```

`React.PropTypes.arrayOf()`

```
React.PropTypes.arrayOf(propType)
```

Validates that a prop is an an array of a certain type.

```
MyComponent.propTypes = {
  optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number),
}
```

`React.PropTypes.objectOf()`

```
React.PropTypes.objectOf(propType)
```

Validates that a prop is an object with property values of a certain type.

```
MyComponent.propTypes = {
  optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number),
}
```

`React.PropTypes.shape()`

```
React.PropTypes.shape(object)
```

Validates that a prop is an object taking on a particular shape.

```
MyComponent.propTypes = {
  optionalObjectWithShape: React.PropTypes.shape({
    color: React.PropTypes.string,
    fontSize: React.PropTypes.number
  }),
}
```

`React.PropTypes.any`

```
React.PropTypes.any
```

Validates that a prop has a value of any data type. Usually followed by `isRequired`.

```
MyComponent.propTypes = {
  requiredAny: React.PropTypes.any.isRequired,
}
```

`isRequired`

```
propType.isRequired
```

You can chain any of the above validators with `isRequired` to make sure a warning is shown if the prop is not provided.

```
MyComponent.propTypes = {
  requiredFunc: React.PropTypes.func.isRequired,
}
```

`React.addons`

```
React.addons
```

`React.addons` exports a range of add-ons when using `react-with-addons.js` .

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. `React.Component` is provided by `React`.

Overview

`React.Component` is an abstract base class, so it rarely makes sense to refer to `React.Component` directly. Instead, you will typically subclass it, and define at least a `render()` method.

Normally you would define a React component as a plain JavaScript class:

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

If you don't use ES6 yet, you may use the `React.createClass` helper instead. Take a look at Using React without ES6 to learn more.

The Component Lifecycle

Each component has several "lifecycle methods" that you can override to run code at particular times in the process. Methods prefixed with `will` are called right before something happens, and methods prefixed with `did` are called right after something happens.

Mounting

These methods are called when an instance of a component is being created and inserted into the DOM:

- `constructor()`
- `componentWillMount()`
- `render()`
- `componentDidMount()`

Updating

An update can be caused by changes to props or state. These methods are called when a component is being re-rendered:

- `componentWillReceiveProps()`
- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

Unmounting

This method is called when a component is being removed from the DOM:

- `componentWillUnmount()`

Other APIs

Each component also provides some other APIs:

- `setState()`
- `forceUpdate()`

Class Properties

- `defaultProps`
- `displayName`
- `propTypes`

Instance Properties

- `props`
- `state`

Reference

`render()`

```
render()
```

The `render()` method is required.

When called, it should examine `this.props` and `this.state` and return a single React element. This element can be either a representation of a native DOM component, such as `<div />`, or another composite component that you've defined yourself.

You can also return `null` or `false` to indicate that you don't want anything rendered. When returning `null` or `false`, `ReactDOM.findDOMNode(this)` will return `null`.

The `render()` function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser. If you need to interact with the browser, perform your work in `componentDidMount()` or the other lifecycle methods instead. Keeping `render()` pure makes components easier to think about.

> Note
>
> `render()` will not be invoked if `shouldComponentUpdate()` returns false.

`constructor()`

```
constructor(props)
```

The constructor for a React component is called before it is mounted. When implementing the constructor for a `React.Component` subclass, you should call `super(props)` before any other statement. Otherwise, `this.props` will be undefined in the constructor, which can lead to bugs.

The constructor is the right place to initialize state. If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

It's okay to initialize state based on props if you know what you're doing. Here's an example of a valid `React.Component` subclass constructor:

```
constructor(props) {
  super(props);
  this.state = {
    color: props.initialColor
  };
```

```
}
```

Beware of this pattern, as it effectively "forks" the props and can lead to bugs. Instead of syncing props to state, you often want to lift the state up.

If you "fork" props by using them for state, you might also want to implement `componentWillReceiveProps(nextProps)` to keep the state up-to-date with them. But lifting state up is often easier and less bug-prone.

`componentWillMount()`

```
componentWillMount()
```

`componentWillMount()` is invoked immediately before mounting occurs. It is called before `render()`, therefore setting state in this method will not trigger a re-rendering. Avoid introducing any side-effects or subscriptions in this method.

This is the only lifecycle hook called on server rendering. Generally, we recommend using the `constructor()` instead.

`componentDidMount()`

```
componentDidMount()
```

`componentDidMount()` is invoked immediately after a component is mounted. Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request. Setting state in this method will trigger a re-rendering.

`componentWillReceiveProps()`

```
componentWillReceiveProps(nextProps)
```

`componentWillReceiveProps()` is invoked before a mounted component receives new props. If you need to update the state in response to prop changes (for example, to reset it), you may compare `this.props` and `nextProps` and perform state transitions using `this.setState()` in this method.

Note that React may call this method even if the props have not changed, so make sure to compare the current and next values if you only want to handle changes. This may occur when the parent component causes your component to re-render.

React doesn't call `componentWillReceiveProps` with initial props during mounting. It only calls this method if some of component's props may update. Calling `this.setState` generally doesn't trigger `componentWillReceiveProps`.

`shouldComponentUpdate()`

```
shouldComponentUpdate(nextProps, nextState)
```

Use `shouldComponentUpdate()` to let React know if a component's output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

`shouldComponentUpdate()` is invoked before rendering when new props or state are being received. Defaults to `true`. This method is not called for the initial render or when `forceUpdate()` is used.

Returning `false` does not prevent child components from re-rendering when *their* state changes.

Currently, if `shouldComponentUpdate()` returns `false`, then `componentWillUpdate()`, `render()`, and `componentDidUpdate()` will not be invoked. Note that in the future React may treat `shouldComponentUpdate()` as a hint rather than a strict directive, and returning `false` may still result in a re-rendering of the component.

If you determine a specific component is slow after profiling, you may change it to inherit from `React.PureComponent` which implements `shouldComponentUpdate()` with a shallow prop and state comparison. If you are confident you want to write it by hand, you may compare `this.props` with `nextProps` and `this.state` with `nextState` and return `false` to tell React the update can be skipped.

---

componentWillUpdate()

```
componentWillUpdate(nextProps, nextState)
```

`componentWillUpdate()` is invoked immediately before rendering when new props or state are being received. Use this as an opportunity to perform preparation before an update occurs. This method is not called for the initial render.

Note that you cannot call `this.setState()` here. If you need to update state in response to a prop change, use `componentWillReceiveProps()` instead.

> Note
>
> `componentWillUpdate()` will not be invoked if `shouldComponentUpdate()` returns false.

---

componentDidUpdate()

```
componentDidUpdate(prevProps, prevState)
```

`componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

> Note
>
> `componentDidUpdate()` will not be invoked if `shouldComponentUpdate()` returns false.

---

componentWillUnmount()

```
componentWillUnmount()
```

`componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any DOM elements that were created in `componentDidMount`

`setState()`

```
setState(nextState, callback)
```

Performs a shallow merge of nextState into current state. This is the primary method you use to trigger UI updates from event handlers and server request callbacks.

The first argument can be an object (containing zero or more keys to update) or a function (of state and props) that returns an object containing keys to update.

Here is the simple object usage:

```
this.setState({mykey: 'my new value'});
```

It's also possible to pass a function with the signature `function(state, props) => newState`. This enqueues an atomic update that consults the previous value of state and props before setting any values. For instance, suppose we wanted to increment a value in state by `props.step`:

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

The second parameter is an optional callback function that will be executed once `setState` is completed and the component is re-rendered. Generally we recommend using `componentDidUpdate()` for such logic instead.

`setState()` does not immediately mutate `this.state` but creates a pending state transition. Accessing `this.state` after calling this method can potentially return the existing value.

There is no guarantee of synchronous operation of calls to `setState` and calls may be batched for performance gains.

`setState()` will always lead to a re-render unless `shouldComponentUpdate()` returns `false`. If mutable objects are being used and conditional rendering logic cannot be implemented in `shouldComponentUpdate()`, calling `setState()` only when the new state differs from the previous state will avoid unnecessary re-renders.

`forceUpdate()`

```
component.forceUpdate(callback)
```

By default, when your component's state or props change, your component will re-render. If your `render()` method depends on some other data, you can tell React that the component needs re-rendering by calling `forceUpdate()`.

Calling `forceUpdate()` will cause `render()` to be called on the component, skipping `shouldComponentUpdate()`. This will trigger the normal lifecycle methods for child components, including the `shouldComponentUpdate()` method of each child. React will still only update the DOM if the markup changes.

Normally you should try to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in `render()`.

Class Properties

`defaultProps`

`defaultProps` can be defined as a property on the component class itself, to set the default props for the class. This is used for undefined props, but not for null props. For example:

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

If `props.color` is not provided, it will be set by default to `'blue'`:

```
render() {
  return <CustomButton /> ; // props.color will be set to blue
}
```

If `props.color` is set to null, it will remain null:

```
render() {
  return <CustomButton color={null} /> ; // props.color will remain null
}
```

---

displayName

The `displayName` string is used in debugging messages. JSX sets this value automatically; see JSX in Depth.

---

propTypes

`propTypes` can be defined as a property on the component class itself, to define what types the props should be. It should be a map from prop names to types as defined in `React.PropTypes`. In development mode, when an invalid value is provided for a prop, a warning will be shown in the JavaScript console. In production mode, `propTypes` checks are skipped for efficiency.

For example, this code ensures that the `color` prop is a string:

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.propTypes = {
  color: React.PropTypes.string
};
```

We recommend using Flow when possible, to get compile-time typechecking instead of runtime typechecking. Flow has built-in support for React so it's easy to run static analysis on a React app.

Instance Properties

`props`

`this.props` contains the props that were defined by the caller of this component. See Components and Props for an introduction to props.

In particular, `this.props.children` is a special prop, typically defined by the child tags in the JSX expression rather than in the tag itself.

`state`

The state contains data specific to this component that may change over time. The state is user-defined, and it should be a plain JavaScript object.

If you don't use it in `render()`, it shouldn't be on the state. For example, you can put timer IDs directly on the instance.

See State and Lifecycle for more information about the state.

Never mutate `this.state` directly, as calling `setState()` afterwards may replace the mutation you made. Treat `this.state` as if it were immutable.

If you use React as a script tag, these top-level APIs are available on the `ReactDOM` global. If you use ES6 with npm, you can write `import ReactDOM from 'react-dom'`. If you use ES5 with npm, you can write `var ReactDOM = require('react-dom')`.

Overview

The `react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to. Most of your components should not need to use this module.

- `render()`
- `unmountComponentAtNode()`
- `findDOMNode()`

Browser Support

React supports all popular browsers, including Internet Explorer 9 and above.

> Note
>
> We don't support older browsers that don't support ES5 methods, but you may find that your apps do work in older browsers if polyfills such as es5-shim and es5-sham are included in the page. You're on your own if you choose to take this path.

Reference

`render()`

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

Render a React element into the DOM in the supplied `container` and return a reference to the component (or returns `null` for stateless components).

If the React element was previously rendered into `container`, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React element.

If the optional callback is provided, it will be executed after the component is rendered or updated.

> Note:
>
> `ReactDOM.render()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.
>
> `ReactDOM.render()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.
>
> `ReactDOM.render()` currently returns a reference to the root `ReactComponent` instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root `ReactComponent` instance, the preferred solution is to attach a callback ref to the root element.

`unmountComponentAtNode()`

```
ReactDOM.unmountComponentAtNode(container)
```

Remove a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns `true` if a component was unmounted and `false` if there was no component to unmount.

`findDOMNode()`

```
ReactDOM.findDOMNode(component)
```

If this component has been mounted into the DOM, this returns the corresponding native browser DOM element. This method is useful for reading values out of the DOM, such as form field values and performing DOM measurements. **In most cases, you can attach a ref to the DOM node and avoid using `findDOMNode` at all.** When `render` returns `null` or `false`, `findDOMNode` returns `null`.

> Note:
>
> `findDOMNode` is an escape hatch used to access the underlying DOM node. In most cases, use of this escape hatch is discouraged because it pierces the component abstraction.
>
> `findDOMNode` only works on mounted components (that is, components that have been placed in the DOM). If you try to call this on a component that has not been mounted yet (like calling `findDOMNode()` in `render()` on a component that has yet to be created) an exception will be thrown.
>
> `findDOMNode` cannot be used on functional components.

If you use React as a script tag, these top-level APIs are available on the `ReactDOMServer` global. If you use ES6 with npm, you can write `import ReactDOMServer from 'react-dom/server'`. If you use ES5 with npm, you can write `var ReactDOMServer = require('react-dom/server')`.

Overview

The `ReactDOMServer` class allows you to render your components on the server.

- `renderToString()`
- `renderToStaticMarkup()`

Reference

`renderToString()`

```
ReactDOMServer.renderToString(element)
```

Render a React element to its initial HTML. This should only be used on the server. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.render()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

`renderToStaticMarkup()`

```
ReactDOMServer.renderToStaticMarkup(element)
```

Similar to `renderToString`, except this doesn't create extra DOM attributes such as `data-reactid`, that React uses internally. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save lots of

bytes.

React implements a browser-independent DOM system for performance and cross-browser compatibility. We took the opportunity to clean up a few rough edges in browser DOM implementations.

In React, all DOM properties and attributes (including event handlers) should be camelCased. For example, the HTML attribute `tabindex` corresponds to the attribute `tabIndex` in React. The exception is `aria-*` and `data-*` attributes, which should be lowercased.

Differences In Attributes

There are a number of attributes that work differently between React and HTML:

checked

The `checked` attribute is supported by `<input>` components of type `checkbox` or `radio`. You can use it to set whether the component is checked. This is useful for building controlled components. `defaultChecked` is the uncontrolled equivalent, which sets whether the component is checked when it is first mounted.

className

To specify a CSS class, use the `className` attribute. This applies to all regular DOM and SVG elements like `<div>`, `<a>`, and others.

If you use React with Web Components (which is uncommon), use the `class` attribute instead.

dangerouslySetInnerHTML

`dangerouslySetInnerHTML` is React's replacement for using `innerHTML` in the browser DOM. In general, setting HTML from code is risky because it's easy to inadvertently expose your users to a cross-site scripting (XSS) attack. So, you can set HTML directly from React, but you have to type out `dangerouslySetInnerHTML` and pass an object with a `__html` key, to remind yourself that it's dangerous. For example:

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}
```

```
function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

htmlFor

Since `for` is a reserved word in JavaScript, React elements use `htmlFor` instead.

onChange

The `onChange` event behaves as you would expect it to: whenever a form field is changed, this event is fired. We intentionally do not use the existing browser behavior because `onChange` is a misnomer for its behavior and React relies on this event to handle user input in real time.

selected

The `selected` attribute is supported by `<option>` components. You can use it to set whether the component is selected. This is useful for building controlled components.

style

The `style` attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM `style` JavaScript property, is more efficient, and prevents XSS security holes. For example:

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

Note that styles are not autoprefixed. To support older browsers, you need to supply corresponding style properties:

```
const divStyle = {
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};
```

```
function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes from JS (e.g. `node.style.backgroundImage`). Vendor prefixes other than `ms` should begin with a capital letter. This is why `WebkitTransition` has an uppercase "W".

suppressContentEditableWarning

Normally, there is a warning when an element with children is also marked as `contentEditable`, because it won't work. This attribute suppresses that warning. Don't use this unless you are building a library like Draft.js that manages `contentEditable` manually.

value

The `value` attribute is supported by `<input>` and `<textarea>` components. You can use it to set the value of the component. This is useful for building controlled components. `defaultValue` is the uncontrolled equivalent, which sets the value of the component when it is first mounted.

All Supported HTML Attributes

React supports all `data-*` and `aria-*` attributes as well as these attributes:

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoPlay capture cellPadding cellSpacing challenge
charSet checked cite classID className colSpan cols content contentEditable
contextMenu controls coords crossOrigin data dateTime default defer dir
disabled download draggable encType form formAction formEncType formMethod
formNoValidate formTarget frameBorder headers height hidden high href hrefLang
htmlFor httpEquiv icon id inputMode integrity is keyParams keyType kind label
lang list loop low manifest marginHeight marginWidth max maxLength media
mediaGroup method min minLength multiple muted name noValidate nonce open
optimum pattern placeholder poster preload profile radioGroup readOnly rel
required reversed role rowSpan rows sandbox scope scoped scrolling seamless
selected shape size sizes span spellCheck src srcDoc srcLang srcSet start step
style summary tabIndex target title type useMap value width wmode wrap
```

These RDFa attributes are supported (several RDFa attributes overlap with standard HTML attributes and thus are excluded from this list):

```
about datatype inlist prefix property resource typeof vocab
```

In addition, the following non-standard attributes are supported:

- `autoCapitalize autoCorrect` for Mobile Safari.
- `color` for `<link rel="mask-icon" />` in Safari.
- `itemProp itemScope itemType itemRef itemID` for HTML5 microdata.
- `security` for older versions of Internet Explorer.
- `unselectable` for Internet Explorer.
- `results autoSave` for WebKit/Blink input fields of type `search`.

All Supported SVG Attributes

React supports these SVG attributes:

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
```

```
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

This reference guide documents the `SyntheticEvent` wrapper that forms part of React's Event System. See the Handling Events guide to learn more.

Overview

Your event handlers will be passed instances of `SyntheticEvent`, a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.

If you find that you need the underlying browser event for some reason, simply use the `nativeEvent` attribute to get it. Every `SyntheticEvent` object has the following attributes:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

> Note:
>
> As of v0.14, returning `false` from an event handler will no longer stop event propagation. Instead, `e.stopPropagation()` or `e.preventDefault()` should be triggered manually, as appropriate.

Event Pooling

The `SyntheticEvent` is pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event callback has been invoked. This is for performance reasons. As such, you cannot access the event in an asynchronous way.

```
function onClick(event) {
```

```
  console.log(event); // => nullified object.
  console.log(event.type); // => "click"
  const eventType = event.type; // => "click"

  setTimeout(function() {
    console.log(event.type); // => null
    console.log(eventType); // => "click"
  }, 0);

  // Won't work. this.state.clickEvent will only contain null values.
  this.setState({clickEvent: event});

  // You can still export event properties.
  this.setState({eventType: event.type});
}
```

> Note:
>
> If you want to access the event properties in an asynchronous way, you should call `event.persist()` on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

Supported Events

React normalizes events so that they have consistent properties across different browsers.

The event handlers below are triggered by an event in the bubbling phase. To register an event handler for the capture phase, append `Capture` to the event name; for example, instead of using `onClick`, you would use `onClickCapture` to handle the click event in the capture phase.

- Clipboard Events
- Composition Events
- Keyboard Events
- Focus Events
- Form Events
- Mouse Events
- Selection Events
- Touch Events
- UI Events
- Wheel Events

Reference

Clipboard Events

Event names:

```
onCopy onCut onPaste
```

Properties:

```
DOMDataTransfer clipboardData
```

Composition Events

Event names:

```
onCompositionEnd onCompositionStart onCompositionUpdate
```

Properties:

```
string data
```

Keyboard Events

Event names:

```
onKeyDown onKeyPress onKeyUp
```

Properties:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

Focus Events

Event names:

```
onFocus onBlur
```

These focus events work on all elements in the React DOM, not just form elements.

Properties:

```
DOMEventTarget relatedTarget
```

Form Events

Event names:

```
onChange onInput onSubmit
```

For more information about the onChange event, see Forms.

Mouse Events

Event names:

```
onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp
```

The `onMouseEnter` and `onMouseLeave` events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

Selection Events

Event names:

```
onSelect
```

Touch Events

Event names:

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

Properties:

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

UI Events

Event names:

```
onScroll
```

Properties:

```
number detail
DOMAbstractView view
```

Wheel Events

Event names:

```
onWheel
```

Properties:

```
number deltaMode
number deltaX
number deltaY
number deltaZ
```

Media Events

Event names:

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
onTimeUpdate onVolumeChange onWaiting
```

Image Events

Event names:

```
onLoad onError
```

Animation Events

Event names:

```
onAnimationStart onAnimationEnd onAnimationIteration
```

Properties:

```
string animationName
string pseudoElement
float elapsedTime
```

Transition Events

Event names:

```
onTransitionEnd
```

Properties:

```
string propertyName
string pseudoElement
float elapsedTime
```

Other Events

Event names:

```
onToggle
```

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `TransitionGroup` and `CSSTransitionGroup`, for dealing with animations and transitions that are usually not simple to implement, such as before a component's removal.
- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update.
- `update`. Use `kolodny/immutability-helper` instead.

Deprecated Add-ons

`LinkedStateMixin` has been deprecated.

Using React with Add-ons

If using npm, you can install the add-ons individually from npm (e.g. `npm install react-addons-test-utils`) and import them:

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

When using a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

Importing

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a shouldComponentUpdate() hook where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these hooks.

See these articles for an introduction to React performance tooling:

- "How to Benchmark React Components"
- "Performance Engineering with React"
- "A Deep Dive into React Perf Debugging"

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the minified production build. The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`
- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
- `printExclusive()`
- `printWasted()`
- `printOperations()`
- `printDOM()`

Reference

`start()`

`stop()`

```
Perf.start()
// ...
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements()`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

> Note
>
> Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive()`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. If no argument's passed, defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

`printExclusive()`

```
Perf.printExclusive(measurements)
```

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

`printWasted()`

```
Perf.printWasted(measurements)
```

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

`printOperations()`

```
Perf.printOperations(measurements)
```

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

`printDOM()`

```
Perf.printDOM(measurements)
```

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

Importing

```
import ReactTestUtils from 'react-addons-test-utils' // ES6
var ReactTestUtils = require('react-addons-test-utils') // ES5 with npm
var ReactTestUtils = React.addons.TestUtils; // ES5 with react-with-addons.js
```

Overview

`ReactTestUtils` makes it easy to test React components in the testing framework of your choice. At Facebook we use Jest for painless JavaScript testing. Learn how to get started with Jest through the Jest website's React Tutorial.

> Note:
>
> Airbnb has released a testing utility called Enzyme, which makes it easy to assert, manipulate, and traverse your React Components' output. If you're deciding on a unit testing utility to use together with Jest, or any other test runner, it's worth checking out: http://airbnb.io/enzyme/

- `Simulate`
- `renderIntoDocument()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`

Shallow Rendering

Shallow rendering lets you render a component "one level deep" and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

- `createRenderer()`
- `shallowRenderer.render()`
- `shallowRenderer.getRenderOutput()`

Call `createRenderer()` in your tests to create a shallow renderer. You can think of this as a "place" to render the component you're testing, and from which you can extract the component's output.

`shallowRenderer.render()` is similar to `ReactDOM.render()` but it doesn't require DOM and only renders a single level deep. This means you can test components isolated from how their children are implemented.

After `shallowRenderer.render()` has been called, you can use `shallowRenderer.getRenderOutput()` to get the shallowly rendered output.

You can then begin to assert facts about the output. For example, if you have the following component:

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

Then you can assert:

```
const renderer = ReactTestUtils.createRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

Shallow testing currently has some limitations, namely not supporting refs.

We also recommend checking out Enzyme's Shallow Rendering API. It provides a nicer higher-level API over the same functionality.

Reference

`Simulate`

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

Simulate an event dispatch on a DOM node with optional `eventData` event data.

`Simulate` has a method for every event that React understands.

Clicking an element

```
// <button ref="button">...</button>
const node = this.refs.button;
ReactTestUtils.Simulate.click(node);
```

Changing the value of an input field and then pressing ENTER.

```
// <input ref="input" />
const node = this.refs.input;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

> Note
>
> You will have to provide any event property that you're using in your component (e.g. keyCode, which, etc...) as React is not creating any of these for you.

`renderIntoDocument()`

```
renderIntoDocument(element)
```

Render a React element into a detached DOM node in the document. **This function requires a DOM.**

> Note:
>
> You will need to have `window`, `window.document` and `window.document.createElement` globally available **before** you import `React`. Otherwise React will think it can't access the DOM and methods like `setState` won't work.

`mockComponent()`

```
mockComponent(
  componentClass,
  [mockTagName]
)
```

Pass a mocked component module to this method to augment it with useful methods that allow it to be used as a dummy React component. Instead of rendering as usual, the component will become a simple `<div>` (or other tag if `mockTagName` is provided) containing any provided children.

`isElement()`

```
isElement(element)
```

Returns `true` if `element` is any React element.

`isElementOfType()`

```
isElementOfType(
  element,
  componentClass
)
```

Returns `true` if `element` is a React element whose type is of a React `componentClass`.

`isDOMComponent()`

```
isDOMComponent(instance)
```

Returns `true` if `instance` is a DOM component (such as a `<div>` or `<span>`).

`isCompositeComponent()`

```
isCompositeComponent(instance)
```

Returns `true` if `instance` is a user-defined component, such as a class or a function.

`isCompositeComponentWithType()`

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

Returns `true` if `instance` is a component whose type is of a React `componentClass`.

`findAllInRenderedTree()`

```
findAllInRenderedTree(
  tree,
  test
)
```

Traverse all components in `tree` and accumulate all components where `test(component)` is `true`. This is not that useful on its own, but it's used as a primitive for other test utils.

`scryRenderedDOMComponentsWithClass()`

```
scryRenderedDOMComponentsWithClass(
  tree,
  className
)
```

Finds all DOM elements of components in the rendered tree that are DOM components with the class name matching `className`.

`findRenderedDOMComponentWithClass()`

```
findRenderedDOMComponentWithClass(
  tree,
  className
)
```

Like `scryRenderedDOMComponentsWithClass()` but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

`scryRenderedDOMComponentsWithTag()`

```
scryRenderedDOMComponentsWithTag(
  tree,
  tagName
)
```

Finds all DOM elements of components in the rendered tree that are DOM components with the tag name matching `tagName`.

`findRenderedDOMComponentWithTag()`

```
findRenderedDOMComponentWithTag(
```

```
  tree,
  tagName
)
```

Like `scryRenderedDOMComponentsWithTag()` but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

`scryRenderedComponentsWithType()`

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

Finds all instances of components with type equal to `componentClass`.

`findRenderedComponentWithType()`

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

Same as `scryRenderedComponentsWithType()` but expects there to be one result and returns that one result, or throws exception if there is any other number of matches besides one.

Shallow Rendering

`createRenderer()`

```
createRenderer()
```

Call this in your tests to create a shallow renderer.

`shallowRenderer.render()`

```
shallowRenderer.render(
  element
)
```

Similar to `ReactDOM.render` but it doesn't require DOM and only renders a single level deep. See Shallow Rendering.

`shallowRenderer.getRenderOutput()`

```
shallowRenderer.getRenderOutput()
```

After `shallowRenderer.render()` has been called, returns shallowly rendered output.

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent ng-animate library.

Importing

```
import ReactCSSTransitionGroup from 'react-addons-css-transition-group' // ES6
var ReactCSSTransitionGroup = require('react-addons-css-transition-group') // ES5
with npm
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup; // ES5 with react-w
ith-addons.js
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={item} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ));

    return (
```

```
        <div>
          <button onClick={this.handleAdd}>Add Item</button>
          <ReactCSSTransitionGroup
            transitionName="example"
            transitionEnterTimeout={500}
            transitionLeaveTimeout={300}>
            {items}
          </ReactCSSTransitionGroup>
        </div>
      );
    }
  }
```

> Note:
>
> You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```
.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}
```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```
render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}
```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```
.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}
```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

> Note:
>
> The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version `0.13`. To maintain backwards compatibility, the default value is set to `false`.

> However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into transitionName you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the enter and leave classes are provided, the enter-active and leave-active classes will be determined by appending '-active' to the end of the class name. Here are two examples using custom classes:

```
// ...
<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
    appear: 'appear',
    appearActive: 'appearActive'
  } }>
  {item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    leave: 'leave',
    appear: 'appear'
  } }>
  {item2}
</ReactCSSTransitionGroup>
// ...
```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the Getting Started section above to see the difference.

```
render() {
  const items = this.state.items.map((item, i) => (
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  ));

  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      {items}
    </div>
  );
}
```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```
import ReactCSSTransitionGroup from 'react-addons-css-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}
        transitionLeaveTimeout={300}>
        <img src={props.imageSrc} key={props.imageSrc} />
      </ReactCSSTransitionGroup>
    </div>
  );
}
```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

> Note:
>
> When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: ReactTransitionGroup

Importing

```
import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with npm

var ReactTransitionGroup = React.addons.TransitionGroup; // ES5 with react-with-addons.js
```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the example above), special lifecycle hooks are called on them.

- `componentWillAppear()`
- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a `<ul>`:

```
<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>
```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `<ul>` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `<span>` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
```

```
  </ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()`

```
componentWillAppear(callback)
```

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()`

```
componentDidAppear()
```

This is called after the `callback` function that was passed to `componentWillAppear` is called.

`componentWillEnter()`

```
componentWillEnter(callback)
```

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()`

```
componentDidEnter()
```

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()`

```
componentWillLeave(callback)
```

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

`componentDidLeave()`

```
componentDidLeave()
```

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

Importing

```
import createFragment from 'react-addons-create-fragment' // ES6
var createFragment = require('react-addons-create-fragment') // ES5 with npm
var createFragment = React.addons.createFragment; // ES5 with react-with-addons.js
```

Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

`Array<ReactNode> createFragment(object children)`

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment'

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
```

```
    });
  } else {
    children = createFragment({
      left: props.leftChildren,
      right: props.rightChildren
    });
  }
  return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

> Note
>
> The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
React.createClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements shouldComponentUpdate, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

> Note:
>
> This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using immutable objects to facilitate fast comparisons of nested data.

> Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

> Note: `shallowCompare` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare' // ES6
var shallowCompare = require('react-addons-shallow-compare') // ES5 with npm
var shallowCompare = React.addons.shallowCompare; // ES5 with react-with-addons.js
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.
It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.
`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

> Note: `LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin' // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin') // ES5 with npm
var LinkedStateMixin = React.addons.LinkedStateMixin; // ES5 with react-with-addons.js
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. This is because data only flows one direction in the Von Neumann model of computing. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See our forms documentation for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

> Note:
>
> `LinkedStateMixin` is just a thin wrapper and convention around the `onChange` / `setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var NoLink = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange} />;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var WithLink = React.createClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var WithoutMixin = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(newValue) {
    this.setState({message: newValue});
  },
  render: function() {
    var valueLink = {
      value: this.state.message,
      requestChange: this.handleChange
    };
    return <input type="text" valueLink={valueLink} />;
  }
});
```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```
var LinkedStateMixin = require('react-addons-linked-state-mixin');

var WithoutLink = React.createClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
```

```
  });
```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

> Note: `PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin' // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin') // ES5 with npm
var PureRenderMixin = React.addons.PureRenderMixin; // ES5 with react-with-addons.
js
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
React.createClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements shouldComponentUpdate, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

> Note:
>
> This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using immutable objects to facilitate fast comparisons of nested data.
>
> Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

> Note: `update` is a legacy add-on. Use kolodny/immutability-helper instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
var update = React.addons.update; // ES5 with react-with-addons.js
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like Clojure. However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's Immutable-js and the Advanced Performance section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
  }),
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

`update()`

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `{$push: array}` `push()` all the items in `array` on the target.
- `{$unshift: array}` `unshift()` all the items in `array` on the target.
- `{$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.

- `{$set: any}` replace the target entirely.
- `{$merge: object}` merge the keys of `object` with the target.
- `{$apply: function}` passes in the current value to the function and updates it with the new returned value.

Examples

Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

In the minified production build of React, we avoid sending down full error messages in order to reduce the number of bytes sent over the wire.

We highly recommend using the development build locally when debugging your app since it tracks additional debug info and provides helpful warnings about potential problems in your apps, but if you encounter an exception while using the production build, this page will reassemble the original text of the error.