<https://www.w3.org/TR/wai-aria/> **…..Accessible rich internet applications**

**React** is a JavaScript library for building user interfaces.

React has been designed from the start for gradual adoption, and **you can use as little or as much React as you need.**

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('root')

);

It displays a heading saying “Hello, world!” on the page

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.

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React [separates concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) with loosely coupled units called “components” that contain both

You can put any valid [JavaScript expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Expressions) inside the curly braces in JSX.

After compilation, JSX expressions become regular JavaScript function calls and evaluate to 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>;

}

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. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.

For example, class becomes [className](https://developer.mozilla.org/en-US/docs/Web/API/Element/className) in JSX, and tabindex becomes [tabIndex](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/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](http://stackoverflow.com/questions/7381974/which-characters-need-to-be-escaped-on-html) 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)](https://en.wikipedia.org/wiki/Cross-site_scripting) attacks.

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!'

}

};

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

Elements are what components are “made of”

React elements are [immutable](https://en.wikipedia.org/wiki/Immutable_object). 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);

It calls ReactDOM.render() every second from a [setInterval()](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) callback.

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.

## 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.

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.

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” (which stands for properties) 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](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes) 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

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”

**Note: Always start component names with a capital letter.**

React treats components starting with lowercase letters as DOM tags. For example, <div />represents an HTML div tag, but <Welcome /> represents a component and requires Welcometo be in scope.

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.

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

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

Whether you declare a component [as a function or a class](https://reactjs.org/docs/components-and-props.html#functional-and-class-components), it must never modify its own props. Consider this sum function:

function sum(a, b) {

return a + b;

}

Such **functions are called**[**“pure”**](https://en.wikipedia.org/wiki/Pure_function)**because they do not attempt to change their inputs, and always return the same result for the same inputs.**

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

We [mentioned before](https://reactjs.org/docs/components-and-props.html#functional-and-class-components) 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

1. Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), 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 {

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')

);

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 Clockcomponent. 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 the component’s tick() method 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 the 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.

### 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((state, props) => ({

counter: state.counter + props.increment

}));

We used an [arrow function](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) above, but it also works with regular functions:

// Correct

this.setState(function(state, props) {

return {

counter: state.counter + props.increment

};

});

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.

You can use stateless components inside stateful components, and vice versa

* 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](https://www.w3.org/TR/DOM-Level-3-Events/)

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](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), 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(state => ({

isToggleOn: !state.isToggleOn

}));

}

render() {

return (

<button onClick={this.handleClick}>

{this.state.isToggleOn ? 'ON' : 'OFF'}

</button>

);

}

}

ReactDOM.render(

<Toggle />,

document.getElementById('root')

);

In JavaScript, class methods are not [bound](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind) 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](https://www.smashingmagazine.com/2014/01/understanding-javascript-function-prototype-bind/). 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, you can use an [arrow function](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions) 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 class fields syntax, to avoid this sort of performance problem.

## Passing Arguments to EventHandlers

Inside a loop it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

The above two lines are equivalent, and use [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) and [Function.prototype.bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind)respectively.

In both cases, the e argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with bind any further arguments are automatically forwarded.

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

## 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:

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>

);

It is not recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state. If you choose not to assign an explicit key to list items then React will default to using indexes as keys.

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

For example, if you [extract](https://reactjs.org/docs/components-and-props.html#extracting-components) a ListItem component, you should keep the key on the <ListItem /> elements in the array rather than on the <li> element in the ListItem itself.

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>

);

}

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

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

## 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()](https://reactjs.org/docs/react-component.html" \l "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.

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state

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>

Essay:

<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

in HTML, an <input type="file"> lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the [File API](https://developer.mozilla.org/en-US/docs/Web/API/File/Using_files_from_web_applications).

<input type="file" />

Because its value is read-only, it is an **uncontrolled** component in React.

## 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**](https://codepen.io/gaearon/pen/wgedvV?editors=0010)

Note how we used the ES6 [**computed property name**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Object_initializer#Computed_property_names) 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](https://reactjs.org/docs/state-and-lifecycle.html#state-updates-are-merged), we only needed to call it with the changed parts

In React, sharing state is accomplished by moving it up to the closest common ancestor of the components that need it. This is called “lifting state up”

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

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**](https://codepen.io/gaearon/pen/gwZOJp?editors=0010)

React elements like <Contacts /> and <Chat /> are just objects, so you can pass them as props like any other data. This approach may remind you of “slots” in other libraries but there are no limitations on what you can pass as props in React.

**inheritance between** two classes, where one class extends another class establishes "IS A" relationship.**Composition** on the other end contains an instance of another class in your class establishes "Has A" relationship.

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!" />

);

}

Composition works equally well for components defined as classes

[**single responsibility principle**](https://en.wikipedia.org/wiki/Single_responsibility_principle), : a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

Web accessibility (also referred to as [**a11y**](https://en.wiktionary.org/wiki/a11y)) is the design and creation of websites that can be used by everyone. Accessibility support is necessary to allow assistive technology to interpret web pages

**Fragments**

class Columns extends React.Component {

render() {

return (

<React.Fragment>

<td>Hello</td>

<td>World</td>

</React.Fragment>

);

}

}

Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once

[Code-Splitting](https://webpack.js.org/guides/code-splitting/) is a feature supported by bundlers like Webpack and Browserify (via [factor-bundle](https://github.com/browserify/factor-bundle)) which can create multiple bundles that can be dynamically loaded at runtime.

Code-splitting your app can help you “lazy-load” just the things that are currently needed by the user, which can dramatically improve the performance of your app. While you haven’t reduced the overall amount of code in your app, you’ve avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

## Route-based code splitting

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Loadable from 'react-loadable';

const Loading = () => <div>Loading...</div>;

const Home = Loadable({

loader: () => import('./routes/Home'),

loading: Loading,

});

const About = Loadable({

loader: () => import('./routes/About'),

loading: Loading,

});

const App = () => (

<Router>

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/about" component={About}/>

</Switch>

</Router>

);

# Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level

Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

**If you only want to avoid passing some props through many levels,**[**component composition**](https://reactjs.org/docs/composition-vs-inheritance.html)**is often a simpler solution than context.**

### React.createContext

const {Provider, Consumer} = React.createContext(defaultValue);

Creates a { Provider, Consumer } pair. When React renders a context Consumer, it will read the current context value from the closest matching Provider above it in the tree.

The defaultValue argument is **only** used by a Consumer when it does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them. Note: passing undefined as a Provider value does not cause Consumers to use defaultValue.

### Provider

<Provider value={/\* some value \*/}>

A React component that allows Consumers to subscribe to context changes.

Accepts a value prop to be passed to Consumers that are descendants of this Provider. One Provider can be connected to many Consumers. Providers can be nested to override values deeper within the tree.

### Consumer

<Consumer>

{value => /\* render something based on the context value \*/}

</Consumer>

A React component that subscribes to context changes.

Requires a [function as a child](https://reactjs.org/docs/render-props.html#using-props-other-than-render). The function receives the current context value and returns a React node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the value argument will be equal to the defaultValue that was passed to createContext().

All Consumers that are descendants of a Provider will re-render whenever the Provider’s valueprop changes. The propagation from Provider to its descendant Consumers is not subject to the shouldComponentUpdate method, so the Consumer is updated even when an ancestor component bails out of the update.

Changes are determined by comparing the new and old values using the same algorithm as [Object.is](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is#Description).

## Forwarding refs to DOM components

**Ref forwarding is an opt-in feature that lets some components take a ref they receive, and pass it further down (in other words, “forward” it) to a child.**

in the example below, FancyButton uses React.forwardRef to obtain the ref passed to it, and then forward it to the DOM button that it renders:

const FancyButton = React.forwardRef((props, ref) => (

<button ref={ref} className="FancyButton">

{props.children}

</button>

));

// You can now get a ref directly to the DOM button:

const ref = React.createRef();

<FancyButton ref={ref}>Click me!</FancyButton>;

This way, components using FancyButton can get a ref to the underlying button DOM node and access it if necessary—just like if they used a DOM button directly.

Here is a step-by-step explanation of what happens in the above example:

1. We create a [React ref](https://reactjs.org/docs/refs-and-the-dom.html) by calling React.createRef and assign it to a ref variable.
2. We pass our ref down to <FancyButton ref={ref}> by specifying it as a JSX attribute.
3. React passes the ref to the (props, ref) => ... function inside forwardRef as a second argument.
4. We forward this ref argument down to <button ref={ref}> by specifying it as a JSX attribute.
5. When the ref is attached, ref.current will point to the <button> DOM node.

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

<MyButton color="blue" shadowSize={2}>

Click Me

</MyButton>

compiles into:

React.createElement(

MyButton,

{color: 'blue', shadowSize: 2},

'Click Me'

)

<div className="sidebar" />

compiles into:

React.createElement(

'div',

{className: 'sidebar'},

null

)

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" />;

}

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.

### 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) {

// Correct! JSX type can be a capitalized variable.

const SpecificStory = components[props.storyType];

return <SpecificStory story={props.story} />;

}

### 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.

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

<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />

### Spread Attributes

const Button = props => {

const { kind, ...other } = props;

const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";

return <button className={className} {...other} />;

};

const App = () => {

return (

<div>

<Button kind="primary" onClick={() => console.log("clicked!")}>

Hello World!

</Button>

</div>

);

};

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:

### 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](https://developer.mozilla.org/en-US/docs/Glossary/Falsy), 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String#String_conversion) first:

<div>

My JavaScript variable is {String(myVariable)}.

</div>

**a higher-order component is a function that takes a component and returns a new component.**

Note that a HOC doesn’t modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC composes the original component by wrapping it in a container component. A HOC is a pure function with zero side-effects.

HOC ….. <https://reactjs.org/docs/higher-order-components.html> …

## 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**](https://reactjs.org/blog/2016/07/13/mixins-considered-harmful.html)**.**

This section exists only for the reference.

Sometimes very different components may share some common functionality. These are sometimes called [cross-cutting concerns](https://en.wikipedia.org/wiki/Cross-cutting_concern). createReactClass lets you use a legacy mixinssystem 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](https://reactjs.org/docs/react-component.html#the-component-lifecycle) 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 createReactClass = require('create-react-class');

var TickTock = createReactClass({

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.

## 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.

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 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.

Refs provide a way to access DOM nodes or React elements created in the render method.

**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.

The value of the ref differs depending on the type of the node:

* When the ref attribute is used on an HTML element, the ref created in the constructor with React.createRef() receives the underlying DOM element as its current property.
* When the ref attribute is used on a custom class component, the ref object receives the mounted instance of the component as its current.
* **You may not use the ref attribute on function components** because they don’t have instances.
* function MyFunctionComponent() {
* return <input />;
* }
* class Parent extends React.Component {
* constructor(props) {
* super(props);
* this.textInput = React.createRef();
* }
* render() {
* // This will \*not\* work!
* return (
* <MyFunctionComponent ref={this.textInput} />
* );
* }
* }

ref updates happen before componentDidMount or componentDidUpdate lifecycle methods.

### Caveats with callback refs

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.

The term [“render prop”](https://cdb.reacttraining.com/use-a-render-prop-50de598f11ce) refers to a simple technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

<DataProvider render={data => (

<h1>Hello {data.target}</h1>

)}/>

Libraries that use render props include [React Router](https://reacttraining.com/react-router/web/api/Route/Route-render-methods) and [Downshift](https://github.com/paypal/downshift).

It’s important to remember that just because the pattern is called “render props” you don’t have to use a prop named *render* to use this pattern. In fact, [any prop that is a function that a component uses to know what to render is technically a “render prop”](https://cdb.reacttraining.com/use-a-render-prop-50de598f11ce).

Although the examples above use render, we could just as easily use the children prop!

<Mouse children={mouse => (

<p>The mouse position is {mouse.x}, {mouse.y}</p>

)}/>

And remember, the children prop doesn’t actually need to be named in the list of “attributes” in your JSX element. Instead, you can put it directly inside the element!

<Mouse>

{mouse => (

<p>The mouse position is {mouse.x}, {mouse.y}</p>

)}

</Mouse>

you’ll see this technique used in the [react-motion](https://github.com/chenglou/react-motion) API.

Since this technique is a little unusual, you’ll probably want to explicitly state that childrenshould be a function in your propTypes when designing an API like this.

Mouse.propTypes = {

children: PropTypes.func.isRequired

};

### Be careful when using Render Props with React.PureComponent

Using a render prop can negate the advantage that comes from using [React.PureComponent](https://reactjs.org/docs/react-api.html" \l "reactpurecomponent)if you create the function inside a render method. This is because the shallow prop comparison will always return false for new props, and each render in this case will generate a new value for the render prop.

To get around this problem, you can sometimes define the prop as an instance method, like so:

class Mouse extends React.PureComponent {

// Same implementation as above...

}

class MouseTracker extends React.Component {

// Defined as an instance method, `this.renderTheCat` always

// refers to \*same\* function when we use it in render

renderTheCat(mouse) {

return <Cat mouse={mouse} />;

}

render() {

return (

<div>

<h1>Move the mouse around!</h1>

<Mouse render={this.renderTheCat} />

</div>

);

}

}

### PropTypes

Here is an example documenting the different validators provided:

import PropTypes from 'prop-types';

MyComponent.propTypes = {

// You can declare that a prop is a specific JS type. By default, these

// are all optional.

optionalArray: PropTypes.array,

optionalBool: PropTypes.bool,

optionalFunc: PropTypes.func,

optionalNumber: PropTypes.number,

optionalObject: PropTypes.object,

optionalString: PropTypes.string,

optionalSymbol: PropTypes.symbol,

// Anything that can be rendered: numbers, strings, elements or an array

// (or fragment) containing these types.

optionalNode: PropTypes.node,

// A React element.

optionalElement: PropTypes.element,

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

// JS's instanceof operator.

optionalMessage: PropTypes.instanceOf(Message),

// You can ensure that your prop is limited to specific values by treating

// it as an enum.

optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// An object that could be one of many types

optionalUnion: PropTypes.oneOfType([

PropTypes.string,

PropTypes.number,

PropTypes.instanceOf(Message)

]),

// An array of a certain type

optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// An object with property values of a certain type

optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// An object taking on a particular shape

optionalObjectWithShape: PropTypes.shape({

color: PropTypes.string,

fontSize: 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: PropTypes.func.isRequired,

// A value of any data type

requiredAny: 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: 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 PropTypes.element you can specify that only a single child can be passed to a component as children.

import PropTypes from 'prop-types';

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: PropTypes.element.isRequired

};

### 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={this.input} />

</label>

<input type="submit" value="Submit" />

</form>

);

}

# Portals

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

ReactDOM.createPortal(child, container)

sometimes it’s useful to insert a child into a different location in the DOM:

render() {

// React does \*not\* create a new div. It renders the children into `domNode`.

// `domNode` is any valid DOM node, regardless of its location in the DOM.

return ReactDOM.createPortal(

this.props.children,

domNode

);

}

A typical use case for portals is when a parent component has an overflow: hidden or z-index style, but you need the child to visually “break out” of its container. For example, dialogs, hovercards, and tooltips.

If you have an event handler such as onClick or onScroll and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be done by using:

* **throttling**: sample changes based on a time based frequency (eg [\_.throttle](https://lodash.com/docs#throttle))
* **debouncing**: publish changes after a period of inactivity (eg [\_.debounce](https://lodash.com/docs#debounce))
* **requestAnimationFrame throttling**: sample changes based on [requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) (eg [raf-schd](https://github.com/alexreardon/raf-schd))

\_.debounce, \_.throttle and raf-schd provide a cancel method to cancel delayed callbacks. You should either call this method from componentWillUnmount or check to ensure that the component is still mounted within the delayed function.

#### Throttle

Throttling prevents a function from being called more than once in a given window of time. The example below throttles a “click” handler to prevent calling it more than once per second.

import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {

constructor(props) {

super(props);

this.handleClick = this.handleClick.bind(this);

this.handleClickThrottled = throttle(this.handleClick, 1000);

}

componentWillUnmount() {

this.handleClickThrottled.cancel();

}

render() {

return <button onClick={this.handleClickThrottled}>Load More</button>;

}

handleClick() {

this.props.loadMore();

}

}

#### Debounce

Debouncing ensures that a function will not be executed until after a certain amount of time has passed since it was last called. This can be useful when you have to perform some expensive calculation in response to an event that might dispatch rapidly (eg scroll or keyboard events). The example below debounces text input with a 250ms delay.

import debounce from 'lodash.debounce';

class Searchbox extends React.Component {

constructor(props) {

super(props);

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

this.emitChangeDebounced = debounce(this.emitChange, 250);

}

componentWillUnmount() {

this.emitChangeDebounced.cancel();

}

render() {

return (

<input

type="text"

onChange={this.handleChange}

placeholder="Search..."

defaultValue={this.props.value}

/>

);

}

handleChange(e) {

// React pools events, so we read the value before debounce.

// Alternately we could call `event.persist()` and pass the entire event.

// For more info see reactjs.org/docs/events.html#event-pooling

this.emitChangeDebounced(e.target.value);

}

emitChange(value) {

this.props.onChange(value);

}

}

#### requestAnimationFrame throttling

[requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) is a way of queuing a function to be executed in the browser at the optimal time for rendering performance. A function that is queued with requestAnimationFrame will fire in the next frame. The browser will work hard to ensure that there are 60 frames per second (60 fps). However, if the browser is unable to it will naturally limit the amount of frames in a second. For example, a device might only be able to handle 30 fps and so you will only get 30 frames in that second. Using requestAnimationFrame for throttling is a useful technique in that it prevents you from doing more than 60 updates in a second. If you are doing 100 updates in a second this creates additional work for the browser that the user will not see anyway.

import rafSchedule from 'raf-schd';

class ScrollListener extends React.Component {

constructor(props) {

super(props);

this.handleScroll = this.handleScroll.bind(this);

// Create a new function to schedule updates.

this.scheduleUpdate = rafSchedule(

point => this.props.onScroll(point)

);

}

handleScroll(e) {

// When we receive a scroll event, schedule an update.

// If we receive many updates within a frame, we'll only publish the latest value.

this.scheduleUpdate({ x: e.clientX, y: e.clientY });

}

componentWillUnmount() {

// Cancel any pending updates since we're unmounting.

this.scheduleUpdate.cancel();

}

render() {

return (

<div

style={{ overflow: 'scroll' }}

onScroll={this.handleScroll}

>

<img src="/my-huge-image.jpg" />

</div>

);

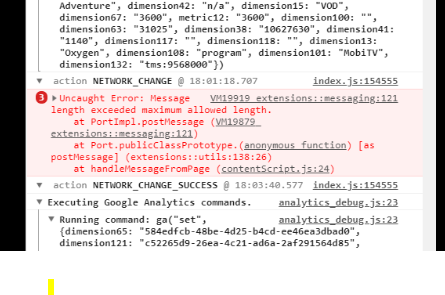
}

}

const { episode\_number: episodeNumber } = this.props.item;

var episodeNumber = this.props.item.episode\_number;

above two lines are same nearly.



Using native XMLHttpRequest in react:

var request = new XMLHttpRequest();

request.open('GET', '/api/content', true);

request.onload = function() {

if (this.status >= 200 && this.status < 400) {

var data = JSON.parse(this.response);

} else {

console.error('Response received and there was an error');

}

};

request.onerror = function() {

console.error('Request error');

};

request.send();