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.

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().

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.

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**

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

**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 Welcome to be in scope.

To learn more about the reasoning behind this convention, please read [JSX In Depth](https://reactjs.org/docs/jsx-in-depth.html#user-defined-components-must-be-capitalized).

**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. we can create an App component that renders Welcome many times:

function App() {

return (

<div>

<Welcome name="Sara" />

<Welcome name="Cahal" />

<Welcome name="Edite" />

</div>

);

}

**Extracting Components**

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

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](https://reactjs.org/docs/components-and-props.html#function-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” 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.

*State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.*

**State and Lifecycle**

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

Converting a Function to a Class:

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

* Create an [ES6 class](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes), with the same name, that extends React.Component.
* Add a single empty method to it called render().
* Move the body of the function into the render() method.
* Replace props with this.props in the render() body.
* 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>

);

}

}

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

The render method will be called each time an update happens, but as long as *we render <Clock /> into the same DOM node, only a single instance of the Clock class will be used. This lets us use additional features such as local state and lifecycle methods.*

**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')*

*);*

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](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) 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](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/clearInterval) 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:

* *componentDidMount() {*

*}*

* *componentWillUnmount() {*

*}*

These methods are called “lifecycle methods”.

The componentDidMount() method 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 doesn’t participate in the data flow (like a timer ID).

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

*componentWillUnmount() {*

*clearInterval(this.timerID);*

*}*

Finally, we will implement a method called tick() that the Clock component will run 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')*

*);*

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 method. 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 method 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((state, props) => ({*

*counter: state.counter + props.increment*

*}));*

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

*// Correct*

*this.setState(function(state, props) {*

*return {*

*counter: state.counter + props.increment*

*};*

*});*

**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>;*

*}*

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')

);

**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**

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

*<button onclick="activateLasers()">*

*Activate Lasers*

*</button>*

is slightly different in React:

*<button onClick={activateLasers}>*

*Activate Lasers*

*</button>*

*-------------------------------------------------------------------------------------------------*

*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.**

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')*

*);*

You have to be careful about the meaning of this in JSX callbacks. 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, there are two ways you can get around this. **If you are using the experimental**[**public class fields syntax**](https://babeljs.io/docs/plugins/transform-class-properties/)**,** you can use class fields 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**](https://github.com/facebookincubator/create-react-app)**.**

If you aren’t using class fields 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 class fields syntax, to avoid this sort of performance problem.

**Passing Arguments to Event Handlers**

*<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 and Function.prototype.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.

**Conditional Rendering**

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like [if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) or the [conditional operator](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) to create elements representing the current state, and let React update the UI to match them.

**Inline If with Logical && Operator**

*{*

*unreadMessages.length > 0 &&*

*<h2>*

*You have {unreadMessages.length} unread messages.*

*</h2>*

*}*

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.

*<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:

*{isLoggedIn*

*? (*

*<LogoutButton onClick={this.handleLogoutClick} />*

*)*

*: (*

*<LoginButton onClick={this.handleLoginClick} />*

*)}*

**Preventing Component from Rendering**

 To do this return null instead of its render output.

*function WarningBanner(props) {*

*if (!props.warn) {*

*return null;*

*}*

*return (*

*<div className="warning">*

*Warning!*

*</div>*

*);*

*}*

 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.

**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.**

**List & Keys**

*function NumberList(props) {*

*const numbers = props.numbers;*

*const listItems = numbers.map((number) =>*

*<li>{number}</li>*

*);*

*return (*

*<ul>{listItems}</ul>*

*);*

*}*

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.

*function NumberList(props) {*

*const numbers = props.numbers;*

*const listItems = numbers.map((number) =>*

*<li key={number.toString()}>*

*{number}*

*</li>*

*);*

*return (*

*<ul>{listItems}</ul>*

*);*

*}*

**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.

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 order of items may change. This can negatively impact performance and may cause issues with component state. Check out Robin Pokorny’s article for an [in-depth explanation on the negative impacts of using an index as a key](https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318). If you choose not to assign an explicit key to list items then React will default to using indexes as keys.

Here is an [in-depth explanation about why keys are necessary](https://reactjs.org/docs/reconciliation.html#recursing-on-children) if you’re interested in learning more.

**Extracting Components with Keys**

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

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')*

*);*

**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} />*

*);*

**Embedding Map() in JSX**

JSX allows embedding any expression 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>*

*);*

*}*

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**](https://reactjs.org/docs/components-and-props.html#extracting-components)**.**

**Forms**

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

**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#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”.**

*handleChange(event) {*

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

*}*

*<input type="text" value={this.state.value} onChange={this.handleChange} />*

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:

*handleChange(event) {*

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

*}*

*<textarea value={this.state.value} onChange={this.handleChange} />*

**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 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:

*<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>*

*handleChange(event) {*

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

*}*

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.**

**Note**

You can pass an **array** into the **value** **attribute**, allowing you to select multiple options in a select tag:

*<select* ***multiple={true} value={['B', 'C']}****>*

**The File input Tag**

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.

*<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:

*<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>*

*constructor(props) {*

*super(props);*

*this.state = {*

***isGoing: true,***

***numberOfGuests: 2***

*};*

*handleInputChange(event) {*

*const target = event.target;*

*const value =* ***target.type === 'checkbox'*** *?* ***target.checked*** *:* ***target.value****;*

***const name = target.name;***

*this.setState({*

*[name]: value*

*});*

*}*

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.**

**Controlled Input Null Value**

Specifying the value prop on a controlled component **prevents the user from changing the input unless you desire so**. If you’ve specified a value but the input is still editable, you may have accidentally set value to **undefined or null.**

The following code demonstrates this. (The input is locked at first but becomes editable after a short delay.)

*ReactDOM.render(<input value="hi" />, mountNode);*

*setTimeout(function() {*

*ReactDOM.render(<input value={null} />, mountNode);*

*}, 1000);*

**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.

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.**

**Fully Fledged Solutions**

If you’re looking for a complete solution **including validation, keeping track of the visited fields, and handling form submission,** [**Formik**](https://jaredpalmer.com/formik) is one of the popular choices.

However, it is built on the same principles of controlled components and managing state — so don’t neglect to learn them.

**Lifting State Up**

Often, **several components need to reflect the same changing data.** We recommend lifting the shared state up to their closest common ancestor.

Example:

* React calls the function specified as **onChange** on the DOM <input>. In our case, this is the **handleChange** method in the **TemperatureInput** component.
* The **handleChange** method in the **TemperatureInput** component calls **this.props.onTemperatureChange()** **with the new desired value**. Its props, including **onTemperatureChange**, were provided by its parent component, the Calculator.
* When it previously rendered, the Calculator has specified that **onTemperatureChange** of the **Celsius** **TemperatureInput** is the Calculator’s **handleCelsiusChange** method, and **onTemperatureChange** of the **Fahrenheit** **TemperatureInput** is the Calculator’s **handleFahrenheitChange** method. So either of these two Calculator methods gets called depending on which input we edited.
* Inside these methods, **the Calculator component asks React to re-render itself by calling this.setState()** with the new input value and the current scale of the input we just edited.
* React calls the Calculator component’s render method to learn what the UI should look like. The values of both inputs are recomputed based on the current temperature and the active scale. The temperature conversion is performed here.
* **React calls the render methods of the individual TemperatureInput components with their new props specified by the Calculator**. It learns what their UI should look like.
* React calls the render method of the **BoilingVerdict** component, passing the temperature in Celsius as its props.
* React DOM updates the DOM with the boiling verdict and to match the desired input values. The input we just edited receives its current value, and the other input is updated to the temperature after conversion.

**Every update goes through the same steps so the inputs stay in sync.**

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 temperature 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.

**Composition Vs inheritance**

React has a powerful composition model, and it is recommend to use composition instead of inheritance **to reuse code between components.**

**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>***

*);*

*}*

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 />***

***}*** */>*

*);*

*}*

**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.**

**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!" />***

*);*

*}*

**Composition works equally well for components defined as classes.**

**So what about Inheritance?**

**From the team itself:**

*“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.*** *“*

**Thinking in React**

*“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.

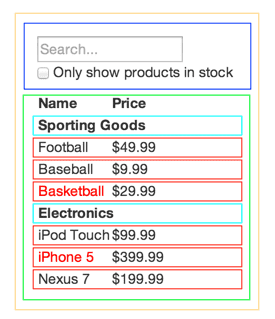
<https://reactjs.org/docs/thinking-in-react.html>

**Always follow Single Responsibility Principle**: a component should ideally only do one thing. If it ends up growing, it should be decomposed into smaller subcomponents.

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.**



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

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.

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.**

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.**