

TODO:

- React documentation (not tutorial)
  - Apparently the React tutorial is “not very good”
- FCC
- University of Helsinki’s Full Stack Open course

## Introduction

ReactJs is:

- an open source JavaScript library for creating user interfaces.
- It was created and maintained by Facebook.
- It lets you write readable HTML directly within JavaScript.
- When using React, all content that needs to be rendered is usually defined as React components.

## React - Set Up

React uses its own markup language 'JSX' (a “syntax extension to JavaScript”)

JSX code must be compiled into vanilla JavaScript using a 'transpiler' like Babel.

To use React in your page you have two options:

1. you can just use normal JavaScript syntax in a .js file (natively supported by the browser) plus cdns for React - but the syntax is more complex (writing React as "pure JavaScript" without using JSX). No one with a sound mind would actually do this.
2. Or equivalently, you can use "JSX" code (an actual *syntax extension* to javascript) which is neater/cleaner but requires a transpiler.

For a quick cdn, use:

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js">  
</script>
```

 and add `type = "text/babel"` to any script tag you want to use JSX in.

However, this is slow

Faster/more complicated method is to download (with npm) a preprocessor that will convert your

file to plain js.

[Follow this](#)

Remember - You need CDNs for React as well as for the transpiler! There are three in total!

```
const App = () => {
  const now = new Date()
  const a = 10
  const b = 20

  return (
    <div>
      <p>Hello world, it is {now.toString()}</p>
      <p>
        {a} plus {b} is {a + b}
      </p>
    </div>
  )
}

const App = () => {
  const now = new Date()
  const a = 10
  const b = 20
  return React.createElement(
    'div',
    null,
    React.createElement(
      'p', null, 'Hello world, it is ', now.toString()
    ),
    React.createElement(
      'p', null, a, ' plus ', b, ' is ', a + b
    )
  )
}

ReactDOM.render(
  React.createElement(App, null),
  document.getElementById('root')
)
```

### *Full Stack Open 2019 Method*

<https://create-react-app.dev/>

- Tool is called create-react-app
- Installed in cmd (with npm and node.js already installed) with:  
`npm install create-react-app`
- To create an application:

```
$ npx create-react-app part1
$ cd part1
```

- To run the application

```
$ npm start
```

- By default, the application runs in localhost port 3000 with the address <http://localhost:3000>
- Configured to compile with Babel automatically

For React to run, you always have to import:

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

## Commenting

To add comments use `{ /* */ }`

To write vanilla JavaScript within JSX use curly brackets:

```
{ /*blah blah blah this is some JavaScript */ }
```

Unlike HTML, you can easily embed dynamic content by writing appropriate JavaScript within curly braces.

## React Elements

To create a React element:

```
const bro = <div> Hello World! </div>  
  
const bro = (  
  <div>  
    <h1>This is a block of JSX</h1>  
  </div>  
)
```

```
    <p>Here's a subtitle</p>
  </div>
);
//best practice to use parentheses
```

You cannot use "class" to define HTML classes in JSX (class is a reserved for object constructor functions in JavaScript). Use `className` instead (remember camelCase for everything)

```
const bro = (
  <div className = 'myDiv'>
    <h1>Add a class to this div</h1>
  </div>
);
```

In JSX, every tag can be self closing i.e. instead of `<div> </div>` you can just use `<div />`

Every tag needs to be closed, including `<br />`

These are simple React elements representing HTML tags. *You can also create React elements representing user-defined components (below).*

## Rendering to HTML DOM

You need to render your JSX to the HTML DOM using React's rendering API - ReactDOM. ⇒ It is an object containing a `.render()` method you can use to render React elements to the DOM like this:

```
ReactDOM.render(componentToRender, targetNode).
```

`componentToRender` = React element you want to render, `targetNode` = DOM node that you want to render the component to.

The render must be declared after the JSX element declarations (like how you must declare variables before you use them)!

```
const bro = (  
  <div>  
    <h1>Hello World</h1>  
    <p>Lets render this to the DOM</p>  
  </div>
```

```
ReactDOM.render(bro, document.getElementById('target'));
```

Remember: You have to re-render every time you want the UI to update.

```
const App = (props) => {  
  const { counter } = props  
  return (  
    <div>{counter}</div>  
  )  
}
```

```
let counter = 1
```

```
const refresh = () => {  
  ReactDOM.render(<App counter={counter} />,  
    document.getElementById('root'))  
}
```

```
refresh()  
counter += 1  
refresh()  
counter += 1  
refresh()
```

If you want to set a delay between each call to the ReactDOM.render-method, use `setInterval(function, delay)`

```
setInterval(() => {  
  refresh()
```

```
    counter += 1
  }, 1000)
```

This is not a good way to re-render components (better way is to use state).

## React Components

React 'components' are (conceptually) like JS functions → they accept inputs (called "props") and return React elements (determining what is shown on the screen). By using components, you “split the UI” into independent, reusable pieces.

There are two ways to create a React component.

1) use a JavaScript function ⇒ creates a stateless 'functional' component.

We call these “function” components, because they are literally JavaScript functions.

'Stateless' because it can receive data and render it, but does not manage or track changes to that data (can't add state).

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

```
const DemoComponent = function() {
  return (
    <div className='customClass' />
  );
};
```

2) Using the ES6 class syntax (for object constructor functions) to define a component:

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

```

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

    render() {
      return (
        <h1>Hi</h1>
      );
    }
  }
}

```

Class constructor calls `super()`, which calls the constructor of the parent class (React.Component). (Constructor is a special method called during the initialisation of objects created using class). Best practice is to call a component's constructor with super, and pass props to both. (This makes sure the component is initialized properly).

React elements can also represent user-defined components:

```

const element = <div />;
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".*

```

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

```

```

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

```

```

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

```

You can render dynamic content inside a component:

```
const App = () => {
  const now = new Date()
  const a = 10
  const b = 20

  return (
    <div>
      <p>Hello world, it is {now.toString()}</p>
      <p>
        {a} plus {b} is {a + b}
      </p>
    </div>
  )
}
```

Remember: functions in React must begin with a capital!

Remember: a React component must return only one root element! (“a root element is stipulated”) (e.g. wrapped inside a <div/>),

or an array of components!

```
const App = () => {
  return [
    <h1>Greetings</h1>,
    <Hello name="Maya" age={26 + 10} />,
    <Footer />
  ]
}
```

You can avoid having pointless extra “<div> </div>”s in the DOM tree by wrapping with fragments (empty elements) instead:

```
const App = () => {
  const name = 'Peter'
  const age = 10
```



```

return (
  <>
    <h1>Greetings</h1>
    <Hello name="Maya" age={26 + 10} />
    <Hello name={name} age={age} />
    <Footer />
  </>
)
}

```

## Component composition/Multiple components

A component (*parent*) can render other components as *children*

```

const ChildComponent = () => {
  return (
    <div>
      <p>I am the child</p>
    </div>
  );
};

class ParentComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h1>I am the parent</h1>
        <ChildComponent />
      </div>
    );
  }
}

```

```
};
```

You can use a component multiple times! Here we're defining a component "Hello" and using it inside another component "App":

```
const Hello = () => {  
  return (  
    <div>  
      <p>Hello world</p>  
    </div>  
  )  
}
```

```
const App = () => {  
  return (  
    <div>  
      <h1>Greetings</h1>  
      <Hello />  
      <Hello />  
      <Hello />  
    </div>  
  )  
}
```

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

To render React components to the DOM (by making a call to the ReactDOM API) - create a React element representing the component

```
let my-element = < component-name />;  
ReactDOM.render(element, document.getElementById("target"));
```

or just go

```
ReactDOM.render(< component-name />, document.getElementById("target"))
```

Composing complex applications is kept manageable by using many specialized reusable

components (the core philosophy of React).

By convention: the root component at the top of the component tree of the application is called *App*.

## Props

You can pass data to components using “props”.

You simply give the function defining the component a parameter/argument that represents the “props” object. (it’s standard practice to call the object itself “props”).

```
const Hello = (props) => {  
  return (  
    <div>  
      <p>Hello {props.name}</p>  
    </div>  
  )  
}
```

The function (defining the component) receives an object which has fields containing all the props the user of the component defines WHEN THE COMPONENT IS USED!

For example:

```
const Hello = (props) => {  
  return (  
    <div>  
      <p>  
        Hello {props.name}, you are {props.age} years old  
      </p>  
    </div>  
  )  
}  
  
const App = () => {
```

```

const name = 'Peter'
const age = 10

return (
  <div>
    <h1>Greetings</h1>
    <Hello name="Maya" age={26 + 10} />
    <Hello name={name} age={age} />
  </div>
)
}

```

Another example:

```

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

const App = () => {

  return (
    <div>
      <Welcome user = 'Mark' />
    </div>
  )
}

```

Like any argument, you can modify the “props” object’s values however you like (e.g. calling array methods like `.join()` for example)

```

const List = (props) => {
  return <p>{props.tasks.join(", ")}</p>
};

class ToDo extends React.Component {
  constructor(props) {
    super(props);
  }
}

```

```

    }
    render() {
      return (
        <div>
          <h1>To Do Lists</h1>
          <h2>Today</h2>
          <List tasks = {["join the gym", "join the army", "go to
dinner"]} />
          <h2>Tomorrow</h2>
          <List tasks = {["walk dog", "wonder about life", "do a
pullup"]} />
        </div>
      );
    }
  };
};

```

You can use ES6's destructuring assignment on the props object:

```

const Hello = ({ name, age }) => {
  return (
    <div>
      <p>
        Hello {name}, you are {age} years old
      </p>
    </div>
  )
}

```

To pass props to an ES6 class component: use the `this` keyword to refer to the class component itself within itself, to access props use `this.props.propName`.

```

class ReturnTempPassword extends React.Component {
  constructor(props) {
    super(props);
  }
}

```

```

render() {
  return (
    <div>
      <p>Your temporary password is:
    </strong>{this.props.tempPassword}</p>
    </div>
  );
}
};

class ResetPassword extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div>
        <h2>Reset Password</h2>
        <h3>We've generated a new temporary password for you.</h3>
        <h3>Please reset this password from your account settings
        ASAP.</h3>
        <ReturnTempPassword tempPassword = "mypassword" />
      </div>
    );
  }
};

```

Best practice is to call the parent class React.component's constructor method with super (inside the component's own constructor method), and pass props to both (makes sure the component is initially correctly).

Pass props to the base constructor

Class components should always call the base constructor with props.

## Default props

You can use default props (only assigned "if necessary" i.e. you haven't manually set it when the component is called) by accessing the component's `defaultProps` property (which is itself an object) e.g.

```
const ShoppingCart = (props) => {  
  return (  
    <div>  
      <h1>Shopping Cart Component</h1>  
    </div>  
  )  
};
```

```
ShoppingCart.defaultProps = {items: 0};
```

## propTypes

To specify that a component should only receive props' objects properties/methods of a particular type, you have to import PropTypes, and modify the component's propTypes attribute.

(remember, .propTypes property has a lowercase 'p', PropTypes...isRequired has an uppercase 'P')

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

```
const ShoppingCart = (props) => {  
  return (  
    <div>  
      <h1>Shopping Cart Component</h1>  
    </div>  
  )  
};
```

```
ShoppingCart.propTypes = {handleclick: PropTypes.func.isRequired,
```

```
free: PropTypes.bool.isRequired};
```

## State

**State** is any information your component needs to know (like props except private and fully controlled by the component), that can change over time.

- Apps should respond to changes in state and present an updated UI ("state management").
- **State** allows you to track important data in your app and render a UI in response to changes in data.
- State is 'local' or 'encapsulated' → it is not accessible to any component other than the one that owns and sets it

A *stateless component* does not use internal state, a *stateful* component maintains its own internal state.

To give a React component **state**

- You have to use ES6 class syntax to define the component
- Declare a state 'property' in the class' **constructor** function
- when the class is called to create an object - 'initialising the component' - it will be created with a **state** property (obviously)

The **state** object must be set as = a javascript object (object literal) , like this:

```
class StatefulComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "John"
    };
  }
  render() {
    return (
```



```

    <div>
      <h1>{this.state.name}</h1>
    </div>
  );
}
};

```

React uses a Virtual DOM to track changes behind the scenes. When state data updates, it triggers a re-render of components using that data.

In the render method, before the return statement, you can write any javascript you want, i.e. declare a function, alter the data in `state` or `props`, assign data to variables, which you have access to in the return statement.

```

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'freeCodeCamp'
    }
  }
  render() {
    const name = this.state.name;
    return (
      <div>
        <h1>{name}</h1>
      </div>
    );
  }
};

```

An example using 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>
    );
  }
}

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

```

You want to make your component fully encapsulated and reusable.

Do not/NEVER modify state directly after it has been declared (it will not re-render the component), instead use `this.setState({})` elsewhere inside the class.

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

`setState` takes an object literal as an argument (you can assign new properties however you like).

*Never mutate this.state directly, as calling `setState()` afterwards may replace the mutation you made. Treat this.state as if it were immutable*

`this.setState` does a shallow merge → as in when you update on property in state the other properties remain the same

```

// Wrong
this.state.comment = 'Hello';
// Correct
this.setState({comment: 'Hello'});

```

State updates may be asynchronous/ `this.props` and `this.state` may be updated asynchronously (React may bundle together multiple `this.setState()` calls for efficiency) - so DO NOT rely on their values for calculating the next state. Instead pass `setState()` a callback function that receives the previous state as the first argument, and props (at the time the update is applied) as the second argument and returns the object that you'd normally pass into `setState` by itself.

CONFUSINGLY: if you want to use a single-line arrow function (with just a return line), you cannot have a single line arrow function returning just an object literal because an object also uses curly brackets (like an arrow function pointing towards a normal function would). So just slap some round brackets around the object literal and you can use a single return line.

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

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

this.setState((state, props) => {
  return {counter: state.counter + props.increment}
});

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

## Adding lifecycle methods to a class

Set up - when the clock is rendered to the DOM ("mounting")

Clear - when the DOM produced by the component is removed ("unmounting")

Declare special methods on the component class to run code when the component mounts and unmounts = these methods are called "lifecycle methods" (the lifecycle of the component).

```

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>
    );
  }
}

```

`componentDidMount()` method runs AFTER the component has been rendered to the DOM,  
`componentWillUnmount()` method runs when the component is removed from the DOM.

```

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')
);

```

- When the component is used/rendered (a new instance of Clock), when the component is inserted into the DOM `componentDidMount()` is called
- It gives the component an attribute called "timerID" - an Interval
  - Other than `this.props` and `this.state` (which are reserved/have special meanings), you are free to add any additional fields to the class (`this`) you want

- The timerID setInterval() calls the `tick()` method every second
- The tick method just updates the component's state!! It uses `this.setState()` rather than modifying the state object directly! This is what you're supposed to do!
- tick() sets the state's date property to = a new Date object (representing the exact time and date when the Date class is called) every time it is called, as in will be a second later each time

## useState()

React Hooks are a new way to access the core features of React (such as state) without having to use classes.

Part of the newer versions of React, available from 16.8.0 onwards.

Before the addition of hooks, there was no way to add state to functional components.

Components requiring state had to be defined as class components (using JS class syntax).

Functional components are (apparently) the future of React.

useState is one of the built-in react hooks (the state hook)

useState should be used only inside functional components

You should use useState if we need an internal state and don't need to implement more complex logic such as lifecycle methods.

useState(initialState) returns a tuple - a stateful value - we'll call it count (the current state of the counter) and a function to update it - we'll call it setCounter (method that will allow us to update the counter's state). You can use the setCounter method to update the state of count anywhere

Some rules:

- The useState function cannot be called from inside a loop, a conditional expression, or any place that is not inside a functional component

```
import React, { useState } from 'react'
import ReactDOM from 'react-dom'

const App = (props) => {
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    () => setCounter(counter + 1),
    1000
  )

  return (
    <div>{counter}</div>
  )
}
```

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

- Imports the `useState` function in the first line
- The function body that defines the component begins with a function call (`useState(0)`)
- The function call adds state to the component
- The function returns two variables as an array
- We assign those two variables to *counter* and *setCounter* using a destructuring assignment
- *counter* is assigned to 0/set to an initial value of 0
- *setCounter* is assigned to a function (that will later be used to modify state)
- The component calls the *setTimeout()* function (just executes that code a second later), which calls a function that increases counter state by 1
- When *setCounter* (the state-modifying function) is called, React re-renders the component (the body of the component function gets re-executed)
- The process begins again (with counter now being 1)
- The value of the counter state is incremented by 1, which continues for as long as the application is running

You can call the stateful value and the function that modifies it whatever you want:

```
const App = (props) => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)

  return (
    <div>
      <div>
        {left}
        <button onClick={() => setLeft(left + 1)}>
          left
        </button>
        <button onClick={() => setRight(right + 1)}>
          right
        </button>
      </div>
    </div>
  )
}
```



```

        {right}
      </div>
    </div>
  )
}

```

You can implement the same functionality by saving the click count of both the *left* and *right* buttons into a single object:

```

const App = (props) => {
  const [clicks, setClicks] = useState({
    left: 0, right: 0
  })

  const handleLeftClick = () => {
    const newClicks = {
      left: clicks.left + 1,
      right: clicks.right
    }
    setClicks(newClicks)
  }

  const handleRightClick = () => {
    const newClicks = {
      left: clicks.left,
      right: clicks.right + 1
    }
    setClicks(newClicks)
  }

  return (
    <div>
      <div>
        {clicks.left}
        <button onClick={handleLeftClick}>left</button>
        <button onClick={handleRightClick}>right</button>
        {clicks.right}
      </div>
    </div>
  )
}

```

```

    </div>
  </div>
)
}

```

//add “complex states” notes here: it’s all using useState React hook rather than setState with class components

The initial value doesn’t have to be an integer. You can set it as an array, for example. Uses concat rather than push to copy the array (as you are not allowed to modify state directly) → as in copy the array “allClicks” or whatever it’s called (with the spread operator, slice(0), or [].concat(allClicks)), modify your copy, THEN pass the copy into function that updates allClicks (doing this will update allClicks to be the same as the copy, and re-render the component).

```

const App = (props) => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)
  const [allClicks, setAll] = useState([])

  const handleLeftClick = () => {
    setAll(allClicks.concat('L'))
    setLeft(left + 1)
  }

  const handleRightClick = () => {
    setAll(allClicks.concat('R'))
    setRight(right + 1)
  }

  return (
    <div>
      <div>
        {left}
        <button onClick={handleLeftClick}>left</button>
        <button onClick={handleRightClick}>right</button>
        {right}

        <p>{allClicks.join(' ')}</p>
      </div>
    </div>
  )
}

```

```

    </div>
  </div>
)
}

```

## Events

- You must define event handlers in React Elements with camelCase, rather than lower case
- In JSX, you pass a function to the event handler rather than a string

```

//incorrect
let myElement = <button onclick="activateLasers()">Activate Lasers</button>

//correct
let myElement = <button onClick={activateLasers}>Activate Lasers</button>

```

Good idea to/common practice is to give the component (defined with class) a method (which is passed to the event handler/is the event handler). A cool example:

```

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 can define any methods you'd like in the component class.

The method typically needs to use the `this` keyword so it can access the class' properties (like `state` and `props`). In javascript, class methods are not bound to `this` by default. You have to explicitly bind `this` in the constructor to the class methods (they will become bound once the component is initialised and the constructor method is called).

```
this.handleClick = this.handleClick.bind(this)
```

Use `this.methodName.bind(this)` in order to refer to the method later without `()`.

(otherwise, you can use the experimental public class fields syntax enabled by default in create-react-app, or use an arrow function in the callback).

To pass arguments to event handlers:

```

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

```

Example using a functional component:

- NB: don't have to use/bind `this`

```

const App = (props) => {
  const [ counter, setCounter ] = useState(0)

```

```

const handleClick = () => {
  console.log('clicked')
}

return (
  <div>
    <div>{counter}</div>

    <button onClick={handleClick}>
      plus
    </button>
  </div>
)
}

```

```

const App = (props) => {
  const [ counter, setCounter ] = useState(0)

  return (
    <div>
      <div>{counter}</div>
      <button onClick={() => setCounter(counter + 1)}>
        plus
      </button>

      <button onClick={() => setCounter(0)}>
        zero
      </button>
    </div>
  )
}

```

- The value of counter is increased AND the component is re-rendered every time the button is clicked

Or

```

const App = (props) => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () =>
    setCounter(counter + 1)

  const setToZero = () =>
    setCounter(0)

  return (
    <div>
      <div>{counter}</div>

      <button onClick={increaseByOne}>
        plus
      </button>

      <button onClick={setToZero}>
        zero
      </button>
    </div>
  )
}

```

The event handler must reference a function and not a function call (function call will re-render the page and get stuck in a loop)

## Passing state to child components (lifting the state up)

Passing the state as props to child components.  
You want to write small, reusable components.

Best practice in React is to “lift the state up” as high as possible in the component heirarchy (ideally to the App root component)

If components reflect the same changing data, lift the state up to their closest common ancestor.

A common pattern is to have a stateful component containing the state important to your app, that then renders child components. You want these components to have access to some pieces of that state, which are passed in as props.

For example, maybe you have an App component that renders a Navbar, among other components. In your App, you have state that contains a lot of user information, but the Navbar only needs access to the user's username so it can display it. You pass that piece of state to the Navbar component as a prop.

*unidirectional data flow*. State flows in one direction down the tree of your application's components, from the stateful parent component to child components. The child components only receive the state data they need.

```
class MyApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'CamperBot'
    }
  }
  render() {
    return (
      <div>
        <Navbar name = {this.state.name} />
      </div>
    );
  }
}
```

```
};
```

```
class Navbar extends React.Component {  
  constructor(props) {  
    super(props);  
  }  
  render() {  
    return (  
      <div>  
        <h1>Hello, my name is: {this.props.name} </h1>  
      </div>  
    );  
  }  
};
```

```
const Display = ({ counter }) => {  
  return (  
    <div>{counter}</div>  
  )  
}
```

```
const App = (props) => {  
  const [counter, setCounter] = useState(0)  
  const setToValue = (value) => setCounter(value)  
  
  return (  
    <div>  
  
      <Display counter={counter}/>  
      <button onClick={() => setToValue(counter + 1)}>  
        plus
```



```

    </button>
    <button onClick={() => setToValue(0)}>
      zero
    </button>
  </div>
)
}

```

## Conditional Rendering

If you want to render different React elements depending on the state of the application = conditional rendering

Just use **if** or the **conditional (ternary) operator**

```

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')
);

```

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }
}

```

```

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

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

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

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

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

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

```

If you want a component to hide itself, have it return null instead of a React element:

```

function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }
  return (
    <div className="warning">
      Warning!
    </div>
  );
}

```

```

    </div>
  );
}

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

  handleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

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

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

```

## Create a controlled input

Form control elements for text input (`input` and `textarea`) maintain their own state as the user types

You can move this mutable state into a React components state

Initialise the component's state with an "input" property that holds an empty string (representing the text a user types into the input field)

```
class ControlledInput extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: ''
    };
    this.handleChange = this.handleChange.bind(this);
    // remember to bind any new method to this
  }

  handleChange (event) {
    this.setState({input: event.target.value});
  }
  // don't forget this for event handlers in class components!
  render() {
    return (
      <div>
        <input value = {this.state.input} onChange =
{this.handleChange} />
        <h4>Controlled Input:</h4>
        <p>{this.state.input}</p>
      </div>
    );
  }
};
```

The component's state is updated with every change to the input's value (every keystroke)! Very neat!

This can be done with the HTML form element as well. The below renders a form which updates part of the state (the submit property) with the value of the input (in the form) when the submit button (in the form) is pressed (plus the state's input property is updated every time something is typed in the input element). We then pass that to a h1 element:

```
class MyForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      input: '',
      submit: ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({
      input: event.target.value
    });
  }
  handleSubmit(event) {
    this.setState(function (state) {
      return {submit: state.input}
    });
  }
  render() {
    return (
      <div>
        <form onSubmit={this.handleSubmit}>
```

```
        <input value = {this.state.input} onChange =  
{this.handleChange} />  
        <button type='submit'>Submit!</button>  
    </form>  
    <h1>{this.state.submit}</h1>  
</div>  
    );  
}  
};
```