THIS.STATE

State

Dynamic information is information that can change.

React components will often need *dynamic information* in order to render. For example, imagine a component that displays the score of a basketball game. The score of the game might change over time, meaning that the score is *dynamic*. Our component will have to know the score, a piece of dynamic information, in order to render in a useful way.

There are two ways for a component to get dynamic information: `props` and `state`. Besides `props` and `state`, every value used in a component should always stay exactly the same.

You just spent a long lesson learning about `props`. Now it's time to learn about `state`. `props` and `state` are all that you need to set up an ecosystem of interacting React components.

Click Next to get started!

Instructions

In this video, the Searchbar component's state includes a `term` value, which changes as a user inputs text in the search bar.

Setting Initial State

A React component can access dynamic information in two ways: `props` and `state`.

Unlike `props`, a component's `state` is *not* passed in from the outside. A component decides its own `state`.

To make a component have `state`, give the component a `state` property. This property should be declared inside of a constructor method, like this:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'decent' };
  }

  render()                                                    {
    return                                            <div></div>;
  }
}

<Example />
```

Whoa, a constructor method? `super(props)`? What’s going on here? Let’s look more closely at this potentially unfamiliar code:

```
constructor(props)                                            {
  super(props);
  this.state       = {       mood:       'decent'        };
}
```

`this.state` should be equal to an object, like in the example above. This object represents the initial “state” of any component instance. We’ll explain that more soon!

How about the rest of the code? `constructor` and `super` are both features of ES6, not unique to React. There is nothing particularly React-y about their usage here. A full explanation of their functionality is outside of the scope of this course, but we’d recommend familiarizing yourself. It is important to note that React components *always* have to call `super` in their constructors to be set up properly.

Look at the bottom of the highest code example in this column. `<Example />` has a `state`, and its `state` is equal to `{ mood: 'decent' }`.

Instructions

1.

In **App.js**, starting on line 6, add a constructor method to the `App` component class. Give your constructor method a single parameter, named `props`. Use the example code as a guide.

Inside of the body of your constructor method, call `super(props)`. On a new line, still inside the body of your constructor, declare a new property named `this.state`. Again, feel free to refer to the example code.

`this.state` should be equal to the following object:

```
{ title: 'Best App' }
```

Make sure *not* to separate `constructor` and `render` with a comma! Methods should never be comma-separated, if inside of a class body. This is to emphasize the fact that classes and object literals are different.

Checkpoint 2 Passed

Hint

Your constructor should contain two lines.

The first is a call to `super()`:

```
super(props);
```

The second should assign the component's state:

```
this.state = { title: 'Best App' };
```

App.js

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

class App extends React.Component {
  // constructor method begins here:
  constructor(props) {
    super(props);
    this.state = { title: 'Best App'}
  }

  render() {
    return (
      <h1>
        {this.state.title}
      </h1>
    );
  }
}

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

Access a Component's state

To *read* a component’s state, use the expression this.state.name-of-property:

```
class TodayImFeeling extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'decent' };
  }

  render() {
    return (
      <h1>
        I'm feeling {this.state.mood}!
      </h1>
    );
  }
}
```

The above component class reads a property in its state from inside of its render function.

Just like this.props, you can use this.state from any property defined inside of a component class’s body.

Instructions

1.

In **App.js**, get rid of the text inside of the <h1></h1>.

Instead, in between the <h1></h1> tags, read your state‘s title property.

Checkpoint 2 Passed

Hint

The <h1> contains a bunch of text, but you don’t need it. When you’re done, it should instead contain {this.state.title} inside.

2.

At the bottom of the file, render `<App />` using `ReactDOM.render()`.

See your component’s `state` on display. Truly, you have the best of apps.

Checkpoint 3 Passed

Hint

Call `ReactDOM.render()` with two arguments: `<App />` and `document.getElementById('app')`.

App.js

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

class App extends React.Component {
  // constructor method begins here:
  constructor(props) {
    super(props);
    this.state = { title: 'Best App'}
  }

  render() {
    return (
      <h1>
        {this.state.title}
      </h1>
    );
  }
}

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

Update state with this.setState

A component can do more than just read its own state. A component can also *change* its own state.

A component changes its state by calling the function `this.setState()`.

`this.setState()` takes two arguments: an *object* that will update the component’s state, and a callback. You basically never need the callback.

In the code editor, take a look at **Example.js**. Notice that `<Example />` has a state of:

```
{
  mood:   'great',
  hungry:                                               false
}
```

Now, let’s say that `<Example />` were to call:

```
this.setState({ hungry: true });
```

After that call, here is what `<Example />`‘s state would be:

```
{
  mood:   'great',
  hungry:                                                true
}
```

The `mood` part of the state remains unaffected!

`this.setState()` takes an object, and *merges* that object with the component’s current state. If there are properties in the current state that aren’t part of that object, then those properties remain how they were.

Example.js

```
import React from 'react';

class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      mood:   'great',
      hungry: false
    };
  }

  render() {
    return <div></div>;
  }
}
```

```
<Example />
```

Call this.setState from Another Function
The most common way to call `this.setState()` is to call a custom function that *wraps* a `this.setState()` call. `.makeSomeFog()` is an example:

```
class       Example       extends       React.Component          {
  constructor(props)                                              {
    super(props);
    this.state       = {       weather:       'sunny'       };
    this.makeSomeFog           = this.makeSomeFog.bind(this);
  }

  makeSomeFog()                                                   {
    this.setState({
      weather:                                              'foggy'
    });
  }
}
```
Notice how the method `makeSomeFog()` contains a call to `this.setState()`.

You may have noticed a weird line in there:

```
this.makeSomeFog = this.makeSomeFog.bind(this);
```
This line is necessary because `makeSomeFog()`'s body contains the word `this`. We'll talk about it more soon!

Let's walk through how a function wrapping `this.setState()` might work in practice. In the code editor, read **Mood.js** all the way through.

Here is how a `<Mood />`'s state would be set:

1. A user triggers an *event* (in this case a click event, triggered by clicking on a `<button></button>`).
2. The event from Step 1 is being listened for (in this case by the `onClick` attribute on line 20).
3. When this listened-for event occurs, it calls an *event handler* function (in this case, `this.toggleMood()`, called on line 20 and defined on lines 11-14).

4. Inside of the body of the *event handler*, `this.setState()` is called (in this case on line 13).
5. The component's `state` is changed!

Due to the way that event handlers are bound in JavaScript, `this.toggleMood()` loses its `this` when it is used on line 20. Therefore, the expressions `this.state.mood` and `this.setState` on lines 7 and 8 won't mean what they're supposed to… *unless* you have already bound the correct `this` to `this.toggleMood`.

That is why we must bind `this.toggleMood` to `this` on line 8.

For an in-depth explanation of this kind of binding trickery, begin with [the React docs](). For the less curious, just know that in React, whenever you define an event handler that uses `this`, you need to add `this.methodName = this.methodName.bind(this)` to your constructor function.

Look at the statement on line 12. What does that do?

Line 12 declares a `const` named `newMood` equal to the opposite of `this.state.mood`. If `this.state.mood` is "good", then `newMood` will be "bad," and vice versa.

A `<Mood />` instance would display "I'm feeling good!" along with a button. Clicking on the button would change the display to "I'm feeling bad!" Clicking again would change back to "I'm feeling good!", etc. Try to follow the step-by-step walkthrough in **Mood.js** and see how all of this works.

One final note: you *can't* call `this.setState()` from inside of the render function! We'll explain why in the next exercise.

Instructions

1.
In the code editor, select **Toggle.js**.

Before the render method, give `Toggle` a `constructor()` method. `Toggle`'s `constructor()` m ethod should have one parameter, named `props`.

Inside the body of the your constructor method, call `super(props);`

On a new line, still inside the body of your constructor method, set `this.state` equal to this object: `{ color: green }`. Use the example as a guide.

Don't put `green` in quotes! `green` should not be a string, it should be a reference to the variable declared on line 4.

Checkpoint 2 Passed

Hint

Open up **Toggle.js** and find the `Toggle` class.

Give it a `constructor()` like this:

```
class        Toggle        extends        React.Component        {
  constructor(props)                                            {
    super(props);
    //          More          code          goes          here
  }

  //                                                          ...
}
```

Inside of the constructor, set `this.state` to `{ color: green }`, and make sure that `green` is NOT in quotes.

2.

Inside of `Toggle`'s render method, give the `<div></div>` the following attribute:

```
style={{background: this.state.color}}
```

Make sure to include the double curly braces! We'll explain those in a later lesson.

Checkpoint 3 Passed

Hint

Find `Toggle`'s render method, and give the `<div></div>` the following attribute:

```
style={{background: this.state.color}}
```

The resulting line will look like this:

```
<div style={{background: this.state.color}}>
```

3.

On line 2, import the `ReactDOM` library from `react-dom`.

At the bottom of the file, render `<Toggle />` using `ReactDOM.render()`.

Click Run and see if the background color reflects the `state`.

Checkpoint 4 Passed
Hint
On line 2, import the `ReactDOM` library with `import ReactDOM from 'react-dom';`.

At the bottom of the file, render `<Toggle />` using `ReactDOM.render()`. Call it with two arguments: `<Toggle />` and `document.getElementById('app')`.
4.
In between `constructor()` and `render()`, define a new method named `changeColor()`.

`changeColor()` should set the state's `color` to yellow if it's currently green, and vice versa.
Checkpoint 5 Passed
Hint
`.toggleMood()` in **Mood.js** is a good place to look for help.
5.
You just wrote a component class method that called `this.setState()`. When you write a component class method that uses `this`, then you need to *bind* that method inside of your constructor function!

Add the following line to the end of `constructor()`:

```
this.changeColor = this.changeColor.bind(this);
```
Checkpoint 6 Passed
Hint
Find `Toggle`'s constructor and add the following line to the end, before the `}`:

```
this.changeColor = this.changeColor.bind(this);
```
6.
In **Toggle.js**, in the render method, underneath the `<h1></h1>`, add this JSX element:

```
<button>
  Change                                                    color
</button>
```
Checkpoint 7 Passed
Hint
Find the `<h1></h1>` in **Toggle.js**, inside of the `render()` method. Add this JSX button:

```
<button>
  Change                                                   color
</button>
```
7.
Now let’s make the button actually work!

Give the `<button></button>` an `onClick` attribute with a value of `{this.changeColor}`.

Hit Run and let the browser refresh. Does clicking on the button change the color?

Checkpoint 8 Passed

Hint

Give your newly-added the `<button></button>` an `onClick` attribute with a value of `{this.changeColor}`. It will look something like this:

```
<button onClick={this.changeColor}>
```

Mood.js

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

class Mood extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'good' };
    this.toggleMood = this.toggleMood.bind(this);
  }

  toggleMood() {
    const newMood = this.state.mood == 'good' ? 'bad' : 'goo
d';
    this.setState({ mood: newMood });
  }

  render() {
    return (
      <div>
        <h1>I'm feeling {this.state.mood}!</h1>
        <button onClick={this.toggleMood}>
          Click Me
        </button>
      </div>
    );
  }
}

ReactDOM.render(<Mood />, document.getElementById('app'));
```

Toggle.js

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

const green = '#39D1B4';
const yellow = '#FFD712';

class Toggle extends React.Component {

  constructor(props){
    super(props);
    this.state = {color:green}
    this.changeColor = this.changeColor.bind(this);
  }

  changeColor() {
    const newColor = this.state.color == green ? yellow : gr
een;
    this.setState( { color: newColor });
  }

  render() {
    return (
      <div style={{background: this.state.color}}>
        <h1>
          Change my color
        </h1>
        <button onClick = {this.changeColor}>
          Change color
        </button>
      </div>
    );
  }
}

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

`this.setState` Automatically Calls `render`

There's something odd about all of this.

Look again at **Toggle.js**.

When a user clicks on the `<button></button>`, the `.changeColor()` method is called. Take a look at `.changeColor()`'s definition.

`.changeColor()` calls `this.setState()`, which updates `this.state.color`. However, even if `this.state.color` is updated from `green` to `yellow`, that alone shouldn't be enough to make the screen's color change!

The screen's color doesn't change until `Toggle` *renders*.

Inside of the render function, it's this line:

```
<div style={{background:this.state.color}}>
```

that changes a virtual DOM object's color to the new `this.state.color`, eventually causing a change in the screen.

If you call `.changeColor()`, shouldn't you then *also* have to call `.render()` again? `.changeColor()` only makes it so that, the next time that you render, the color will be different. Why can you see the new background right away, if you haven't re-rendered the component?

Here's why: *Any time that you call this.setState(), this.setState() AUTOMATICALLY calls .render() as soon as the state has changed.*

Think of `this.setState()` as actually being two things: `this.setState()`, immediately followed by `.render()`.

That is why you can't call `this.setState()` from inside of the `.render()` method! `this.setState()` *automatically* calls `.r` `ender()`. If `.render()` calls `this.setState()`, then an infinite loop is created.

Toggle.js

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

const green = '#39D1B4';
const yellow = '#FFD712';

class Toggle extends React.Component {

  constructor(props){
    super(props);
    this.state = {color:green}
    this.changeColor = this.changeColor.bind(this);
  }

  changeColor() {
    const newColor = this.state.color == green ? yellow : gr
een;
    this.setState( { color: newColor });
  }

  render() {
    return (
      <div style={{background: this.state.color}}>
        <h1>
          Change my color
        </h1>
        <button onClick = {this.changeColor}>
          Change color
        </button>
      </div>
    );
  }
}

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

Mood.js

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

class Mood extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'good' };
    this.toggleMood = this.toggleMood.bind(this);
  }

  toggleMood() {
    const newMood = this.state.mood == 'good' ? 'bad' : 'goo
d';
    this.setState({ mood: newMood });
  }

  render() {
    return (
      <div>
        <h1>I'm feeling {this.state.mood}!</h1>
        <button onClick={this.toggleMood}>
          Click Me
        </button>
      </div>
    );
  }
}

ReactDOM.render(<Mood />, document.getElementById('app'));
```

Components Interacting Recap
In this unit, you learned how to use `import` and `export` to access one file from another. You learned about `props` and `state`, and the countless variations on how they work.

Before this unit, you learned about JSX, components, and how they can work together.

A React app is basically just a lot of components, setting `state` and passing `props` to one another. Now that you have a real understanding of how to use `props` and `state`, you have by far the most important tools that you need!

In future lessons, the focus will shift slightly outward. In addition to learning more new skills, you’ll also learn your first *programming patterns*. These large-scale strategies will help you combine what you’ve learned and really start building.