React 101
Codecademy
Components Interacting

this.state

State

Dynamic information is information that can change.

React components often need dynamic information in order to render. There are two ways for a component to get dynamic information: `props` and `state`.

Setting Initial 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.

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

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

<Example />
```

`this.state` should be equal to an object, and represents the initial state of any component instance.

React components always have to call super in their constructors to be set up properly.

Access a Component's state

To read a component's state, we 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>
    );
  }
}
```

Update state with this.setState

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. Basically callback is never needed.

`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.

The most common way to call `this.setState()` is to call a custom function that wraps a `this.setState()` call.

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

  makeSomeFog() {
    this.setState({
      weather: 'foggy'
    });
```

```
  }
}
```

```
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' : 'good';
    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'));
```

Due to the way event handlers are bound in JS, this.toggleMood() loses its this when used, therefore the expression this.state.mood and this.setState will not mean what they should unless they have been bound using the bind function.

this.setState() cannot be called from inside the render function.

At this point one would wonder that event though we are changing the state how is it reflecting on the webpage without us rerendering. This happens because when the state is changed the virtual DOM object's color is changed and thus whole screen gets rerendered.

`this.setState()` automatically calls`.render()` as soon as the state is changed.