Component Based Architecture

- Declare a state and tie it to a specific UI component
- Think small to build big! Single purpose components and isolated scope make it easier to debug and maintain
- Small components are re-usable through your app (DRY - don't repeat yourself)
- You might have a component for different features on your site such as:
  - Navbar
  - Search form
  - Create comment form
  - Slider

What does a component look like?

Minimum requirements

- extends the Component class
- defines the 'render' function

```
class HelloWorld extends Component {
  render() {
    return <h1>Hello World!</h1>
  }
};
```

Gotchas

- `render` should return a *single* HTML element.
- Every self-closing tag in JSX needs to have an ending `/`: `<br>` needs to be `<br/>`, etc.

What about the data

- We have two ways of getting data into components
  - Props
  - State

Props (aka: Properties)

- props are the data that is passed into your component
- props are immutable - they can't be changed!
- think of props as being the default data for your component
- can't pass objects/arrays into props!
- Component Docs
- Does not take objects

Example

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

ReactDOM.render(<HelloWorld name="Tom" />)
```

State

- But how do you make UI changes if your data is immutable? With state!
- An object defined on the Component that can be used in the render function to define data that will change.

Example

```
class HelloFriend extends Component {
    constructor(props) {
      // props.name is default data; in our first example we passed in 'David'
      // need to call super(props) when using 'this' in constructor
      super(props);
      this.state = {
        name: props.name
      }

      setTimeout(this.updateName.bind(this), 2000)
    }

    updateName() {
        this.setState({name: 'Jeff'});
    }

    render() {
      return <h1>Hello {this.state.name}!</h1>
    }
};
```

Proptypes

- Defined as the 'propTypes' property on the Component
- Essentially a dictionary where you define what props your component needs
- and what type(s) they should be
  - a type is a `Number`, `Object`, `String`, etc.
- `id` prop should of type `Number`
- `id` is also required!
- `message` is not required, and it is of type `String`

```
MyCoolComponent.propTypes = {
  id: React.PropTypes.number.isRequired,
  message: React.PropTypes.string
}
```

Example

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

class MyCoolComponent extends Component {
  constructor(props) {

  }

  render() {
    return <div id={this.props.id}>{this.props.message}</h1>
  }
};

MyCoolComponent.propTypes = {
  id: PropTypes.number.isRequired,
  message: PropTypes.string
}
```

Exercise: Contact Cards

Props and Proptypes

- Define a 'ContactCard' component that takes in a contact name, mobile number, work phone number and email as properties and displays the results in a visually appealing way.
- The name, mobile number and email are required, but the work phone number is optional.
- Create a page that display at least 3 different contact cards with different information.

Exercise: Decrement

State

- Define a 'Decrement' component that will display a number and an "Increment" button next to it.
- The start number should come from the props.
- The number should have a `number` prop type and be required on the component.
- Clicking on the "Decrement" button should subtract 1 to the number.
- When the number reaches zero, clicking on "Decrement" should show an alert to the user "Cannot be less than zero" and not decrement the number any more.