React Recap

Let’s do a quick recap on what we know so far

-React is a JavaScript library

-A component is created using JavaScript functions or classes

-The HTML looking stuff is called JSX. It determines the content of our components

Starting a new react project

```
create-react-app <name of your project>
```

-this generates a new project with all the needed dependencies, like babel(translates any kind of JS into plain old ES5 JS so it can run on any browser)- this is what Chris showed you on Thursday

```
cd <name of your project>
```

```
npm start
```

Components have 3 parts

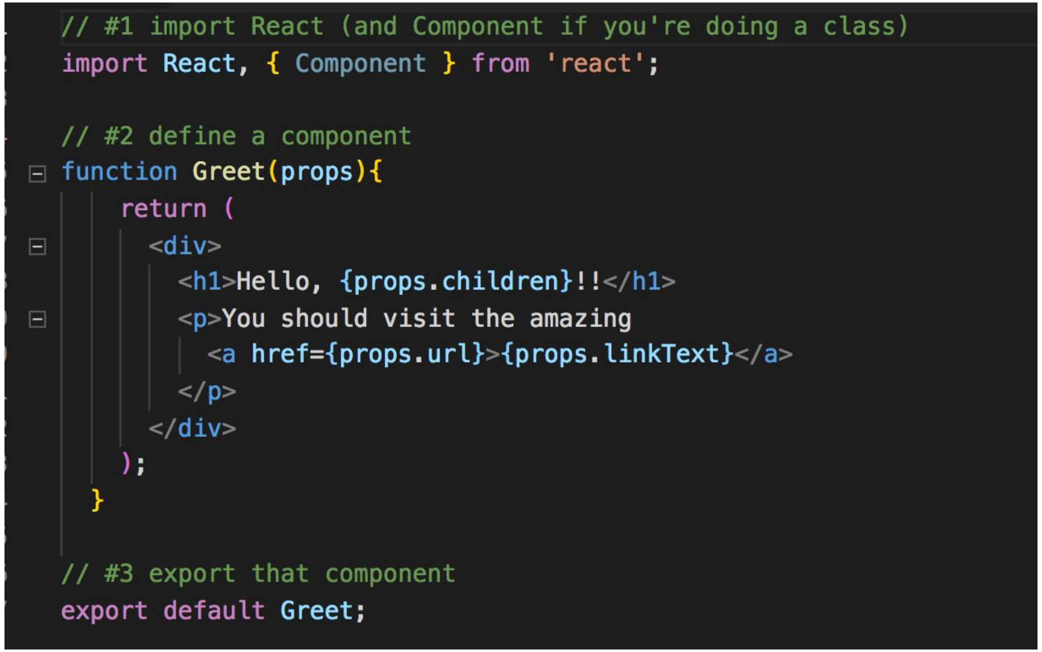
1. Import react (and Component if you’re doing class)
2. Define a component
3. Export the component

   Check out this sweet plug-in for VScode:

https://marketplace.visualstudio.com/items?itemName=xabikos.ReactSnippets

Components can be Class or Functional

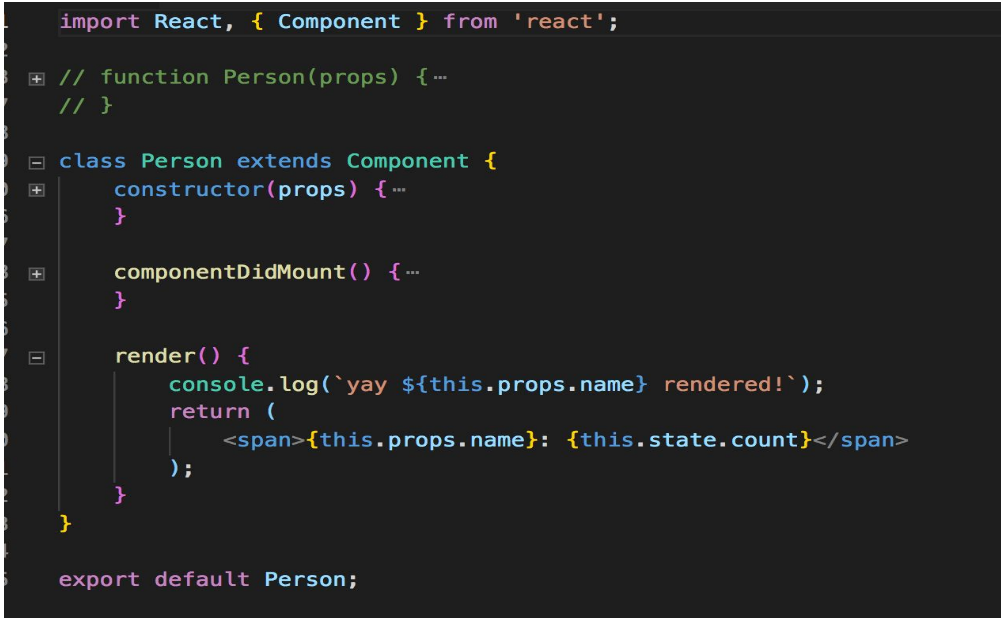
- A functional component is just a JS function. It can only receive props. It does not have state..We'll go over state in a minute…. These are sometimes called stateless components.



```
// #1 import React (and Component if you're doing a class)
import React, { Component } from 'react';

// #2 define a component
function Greet(props){
    return (
      <div>
        <h1>Hello, {props.children}!!</h1>
        <p>You should visit the amazing
          <a href={props.url}>{props.linkText}</a>
        </p>
      </div>
    );
  }

// #3 export that component
export default Greet;
```

Components can be Class or Functional

-Class components: this is a JS class. It had some super powers the Functional component doesn't have, including a required render method. It does have other lifecycle hooks and it can have state
https://reactjs.org/docs/react-component.html#render



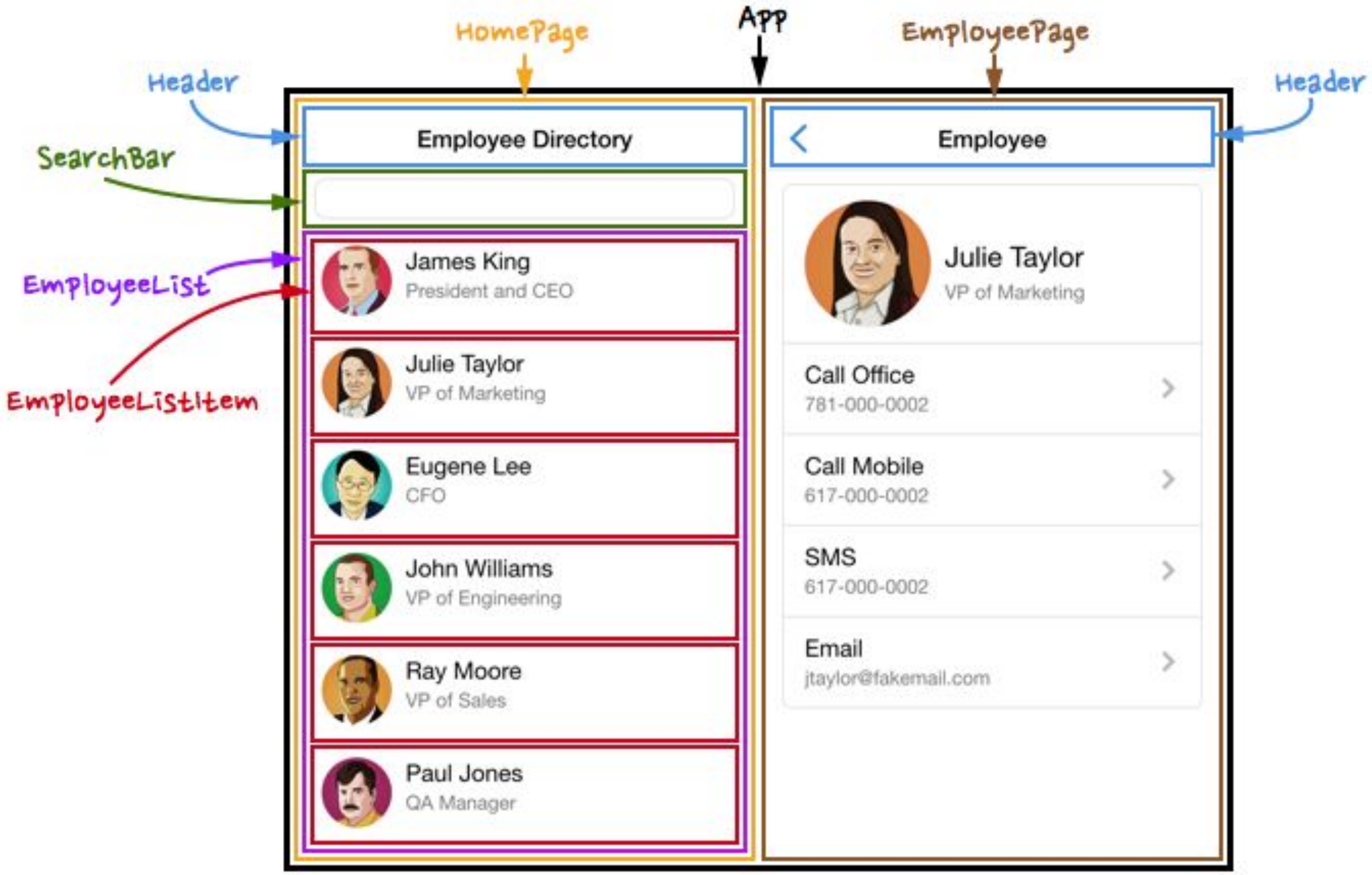
```
import React, { Component } from 'react';

// function Person(props) {…
// }

class Person extends Component {
    constructor(props) {…
    }

    componentDidMount() {…
    }

    render() {
        console.log(`yay ${this.props.name} rendered!`);
        return (
            <span>{this.props.name}: {this.state.count}</span>
        );
    }
}

export default Person;
```

Components continued...

-Components can be **nested** inside of other components- for example: We have the Person component nested inside of our App component

-We want to make components **reusable** so we can use them in different parts of our application. Example: Maybe we have multiple buttons or text fields like comments. We can build one “button” component to use in many different parts of our app

-Components should be **configurable**. Example: In our app, we want give each person a different name.



James King
President and CEO

Julie Taylor
VP of Marketing

Eugene Lee
CFO

John Williams
VP of Engineering

Ray Moore
VP of Sales

Paul Jones
QA Manager

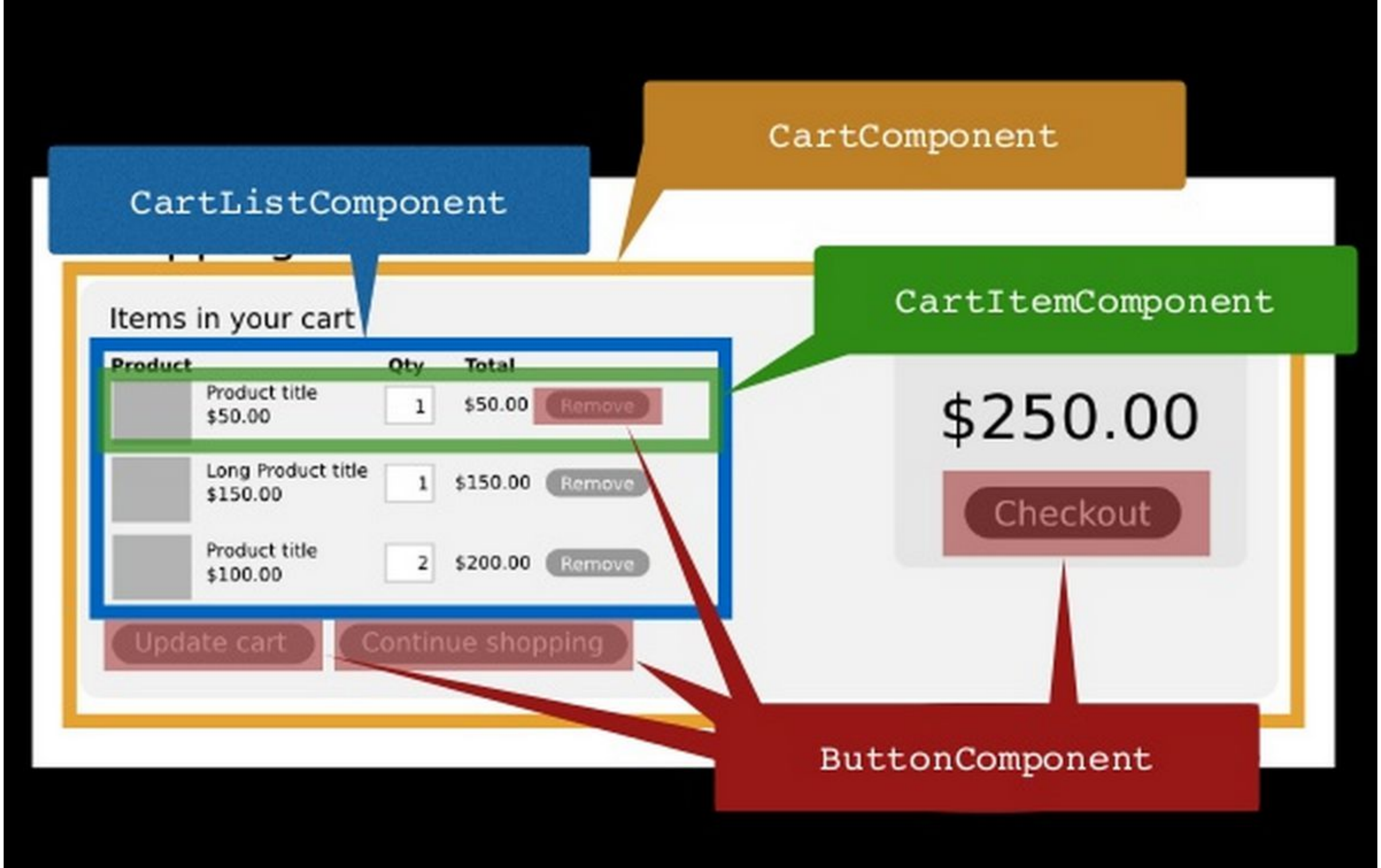
Julie Taylor
VP of Marketing

Call Office
781-000-0002

Call Mobile
617-000-0002

SMS
617-000-0002

Email
jtaylor@fakemail.com


CartComponent
CartListComponent
CartItemComponent
ButtonComponent
Items in your cart
Product
Qty
Total
Product title
$50.00
1
$50.00
Remove
Long Product title
$150.00
1
$150.00
Remove
Product title
$100.00
2
$200.00
Remove
Update cart
Continue shopping
$250.00
Checkout

What’s with props?

-Props make components reusable by giving components the ability to receive data from the parent component in the form of props.

-Props are immutable data that gets passed to children components.

-In our person-counter, the Parent component <App /> can pass down a name prop to each <Person /> component

Passing down a prop

-The parent passes down some information to the child component

<Person name=”Jeffles” />

Prop name

prop value

Receiving a prop

To reference the passed down props, the child has to expect to receive props.

-In functional components, like the <Greet />, we just pass in props as an argument

-In class components, like our <Person /> we use constructor(props) to receive the props.

Constructor and Super

Do I need Constructor in every component?

-Nope, only if your component is class based and you have set an initial state (we’ll get to state in a sec)

Do I have to call super?

-Only if you need access to props inside the constructor of class. You call super(props).

-You are giving the child component access to the parent’s props so you can say, ‘this.props’

-It says “i'm overriding the constructor I've inherited from React.Component, I need to manually trigger all the goodness it sets up for me, including making the `props` available as `this.props` in any of my other methods, such as `render` or `componentDidMount` or `_incrementFarts`, or any other method" ----wise words from Chris.

Without passing super(props)

```
class Person extends Component {

  constructor(props){

    super();

    this.state = {

      firstName: this.props.firstName; // here props would be undefined.

    };

  }

  render () {

    return (

      <p> Name: { this.state.firstName }</p>

    );

  }
```

What the state?

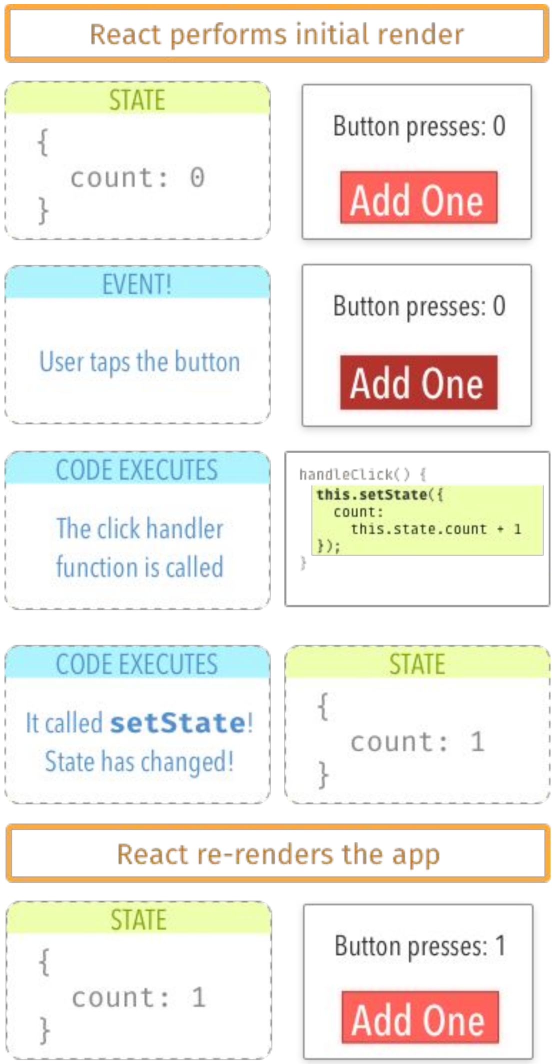
-Think of the state of matter: we can have gas, liquid or solid.

-State is mutable data, meaning it can be changed

-State is managed by the component it is in

-When state is changed (setState), it almost instantly caused the component to re-render

-props vs state is one of the hard concepts in React



```
handleClick() {
  this.setState({
    count:
      this.state.count + 1
  });
}
```

Code- along Scoreboard.

At the top level (`App.js`), there's a piece of state that is an array of playerScores:

```
this.state = {
 scores: [
   {
     id: 1,
     name: 'alice',
     score: 1001
   },
   {
     id: 2,
     name: 'bob',
     score: 20
   },
   {
     id: 3,
     name: 'carol',
     score: 500
   },
 ]
};
```

How would we render each score?

Once we do, how do we solve the error in the console?

-

Let’s add an increment button to increase the score

-Whats up with ‘onClick’?

It’s an event listener!

-Why do we pass in an anonymous function?

-What happens if we don’t wrap it up in an anonymous function?

-let’s console.log it

What does it mean to increase the score of one of the objects?

One rule about state: you can't update it manually. That is, you are never, ever, ever, ever allowed to just re-assign something in state. It breaks React's rendering optimizations.

This.state = “12” is not how it works, because it does not guarantee a rerender

Calling setState will not mutate the state and it will intentionally cause a re-render

What's the solution? Make a copy, then call `this.setState`, passing it the copy.

And when you want to copy an array, but transform/replace at least one of the items… we use map and sprinkles(spread operator).

Can we refactor a bit?

-Lets make a scorecard component

-What’s with bind?

-If you don’t to use anonymous functions, we have to use bind

https://reactjs.org/docs/handling-events.html

Exercises

```
small exercise: make it so you can "decrease score"
-----------------------------------------------------
-add a "decrement" button to what is rendered in the `ScoreCard`
-add an `onClick` to the button
   - for now, have it console.log
-create a method in `App.js` that can decrement a score by id
-pass a handleDecrement prop to `ScoreCard`, setting it to the decrement method
-connect the decrement button's onclick to the handleDecrement prop

medium exercise: add a button to App.js that adds another score to state
------------------------------------------------------------------------
-for now, set the name to "jeff" or "bruce"
-start the score at 0

large exercise: add a button to each ScoreCard that removes its score object from App's
this.state
-------------------------------------------------------------------------------------------
-------
```

Another walkthrough: Walky-Talky aka a Todo App

-Let’s identify:

- -the parts that should be components
- who has state (that is, who needs to keep track of variables between renders)
- who needs to show what variables

- who is responding to user interaction
- how the data is passed around from component to component
  - focus on the fact that children can't talk to each other, they have to go through the parent