**Require A File**

When you use React.js, every JavaScript file in your application is invisible to every other JavaScript file by default. **ProfilePage.js** and **NavBar.js** can’t see each other.

This is a problem!

On line 9, you just added an instance of the NavBar component class. But since you’re in **ProfilePage.js**, JavaScript has no idea what NavBar means.

If you want to use a variable that’s declared in a different file, such as NavBar, then you have to *import* the variable that you want. To import a variable, you can use an import statement:

import { NavBar } from './NavBar.js';

We’ve used import before, but not like this! Notice the differences between the above line of code and this familiar line:

import React from 'react';

The first important difference is the curly braces around NavBar. We’ll get to those soon!

The second important difference involves the contents of the string at the end of the statement: 'react' vs './NavBar.js'.

If you use an import statement, and the string at the end begins with either a dot or a slash, then import will treat that string as a *filepath*. import will follow that filepath, and import the file that it finds.

If your filepath doesn’t have a file extension, then “.js” is assumed. So the above example could be shortened:

import { NavBar } from './NavBar';

**One final, important note:**  
None of this behavior is specific to React! [Module systems](http://eloquentjavascript.net/10_modules.html) of independent, importable files are a very popular way to organize code. [React’s specific module system comes from ES6](https://hacks.mozilla.org/2015/08/es6-in-depth-modules/" \t "_blank). More on all of that later.

**Instructions**

**1.**

The <NavBar /> on line 9 isn’t going to work until you *import* **NavBar.js**.

In **ProfilePage.js**, on line 3, import NavBar from **NavBar.js**. **ProfilePage.js** and **NavBar.js** are located in the same parent directory.

import React from 'react';

import ReactDOM from 'react-dom';

import {NavBar} from './NavBar'

class ProfilePage extends React.Component {

  render() {

    return (

      <div>

        <NavBar />

        <h1>All About Me!</h1>

        <p>I like movies and blah blah blah blah blah</p>

        <img src="https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-monkeyselfie.jpg" />

      </div>

    );

  }

}

**export**

Alright! You’ve learned how to use import to grab a variable from a file *other than* the file that is currently executing.

When you import a variable from a file that is not the current file, then an import statement isn’t quite enough. You also need an export statement, written in the *other* file, exporting the variable that you hope to grab.

export comes from [ES6’s module system,](http://exploringjs.com/es6/ch_modules.html) just like import does. export and import are meant to be used together, and you rarely see one without the other.

There are a few different ways to use export. In this course, we will be using a style called “named exports.” Here’s how named exports works:

In one file, place the keyword export immediately before something that you want to export. That something can be any top-level var, let, const, function, or class:

// Manifestos.js:

export const faveManifestos = {

futurist: 'http://www.artype.de/Sammlung/pdf/russolo\_noise.pdf',

agile: 'https://agilemanifesto.org/iso/en/manifesto.html',

cyborg: 'http://faculty.georgetown.edu/irvinem/theory/Haraway-CyborgManifesto-1.pdf'

};

You can export multiple things from the same file:

// Manifestos.js:

export const faveManifestos = {

futurist: 'http://www.artype.de/Sammlung/pdf/russolo\_noise.pdf',

agile: 'https://agilemanifesto.org/iso/en/manifesto.html',

cyborg: 'http://faculty.georgetown.edu/irvinem/theory/Haraway-CyborgManifesto-1.pdf'

};

export const alsoRan = 'TimeCube';

In a different file, import the name of the var, let, const, function, or class from the first file:

// App.js:

// Import faveManifestos and alsoRan from ./Manifestos.js:

import { faveManifestos, alsoRan } from './Manifestos';

// Use faveManifestos:

console.log(`A Cyborg Manifesto: ${faveManifestos.cyborg}`);

This style of importing and exporting in JavaScript is known as “named exports.” When you use named exports, you always need to wrap your imported names in curly braces, such as:

import { faveManifestos, alsoRan } from './Manifestos';`

[JavaScript’s ES6 module system](http://exploringjs.com/es6/ch_modules.html) goes beyond named exports and has several advanced syntax features.

**Instructions**

**1.**

Select **NavBar.js**.

On line 3, add the word export before the word class. This will export the class NavBar.

Now, when **ProfilePage.js** uses import to grab the variable NavBar from **NavBar.js**, it will get back exactly what it wants: the NavBar component class.

import React from 'react';

export class NavBar extends React.Component {

  render() {

    const pages = ['home', 'blog', 'pics', 'bio', 'art', 'shop', 'about', 'contact'];

    const navLinks = pages.map(page => {

      return (

        <a href={'/' + page}>

          {page}

        </a>

      )

    });

    return <nav>{navLinks}</nav>;

  }

}

**Component Rendering In Action**

Now you’re ready for <ProfilePage /> to render <NavBar />!

All that’s left to do is render <ProfilePage />.

**Instructions**

**1.**

At the bottom of **ProfilePage.js**, use ReactDOM.render() to render an instance of ProfilePage.

For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

Once the result has rendered in the browser, look at the render methods of both ProfilePage and NavBar. Try to figure out exactly which parts of the browser’s display come from which component class.

**2.**

Congratulations! It may not seem like a big deal yet, but you’ve just discovered the key to React’s power.

By nesting components inside of other components, you can build infinitely complex architectures, even if each component is relatively simple. The relationship that you just built is the fundamental relationship of React.js.

import React from 'react';

import ReactDOM from 'react-dom';

import { NavBar } from './NavBar';

class ProfilePage extends React.Component {

  render() {

    return (

      <div>

        <NavBar />

        <h1>All About Me!</h1>

        <p>I like movies and blah blah blah blah blah</p>

        <img src="https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-monkeyselfie.jpg" />

      </div>

    );

  }

}

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

# this.props

Previously, you learned one way that components can interact: a component can render another component.

In this lesson, you will learn another way that components can interact: a component can pass information to another component.

Information that gets passed from one component to another is known as “props.”

**Access a Component's props**

Every component has something called props.

A component’s props is an object. It holds information about that component.

To see a component’s props object, you use the expression this.props. Here’s an example of this.props being used inside of a render method:

render() {

console.log("Props object comin' up!");

console.log(this.props);

console.log("That was my props object!");

return <h1>Hello world</h1>;

}

Most of the information in this.props is pretty useless! But some of it is extremely important, as you’ll see.

**Instructions**

**1.**

Look at **PropsDisplayer.js**.

On line 6, you can see a variable named stringProps. stringProps is equal to a stringified version of this.props.

On line 11, *inject* stringProps in between the <h2></h2> tags.

**2.**

On line 18, call ReactDOM.render().

For ReactDOM.render()‘s first argument, pass in an *instance* of PropsDisplayer.

For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

Click Run, and get ready to see <PropsDisplayer />‘s props object!

… hm. Kind of a letdown, huh.

Despite what you see in the browser, <PropsDisplayer />‘s props object isn’t *really* empty. It has some properties that JSON.stringify doesn’t detect. But even if you could see those properties, the props object still wouldn’t have much of value to show you right now.

But it’s there!

import React from 'react';

import ReactDOM from 'react-dom';

class PropsDisplayer extends React.Component {

  render() {

    const stringProps = JSON.stringify(this.props);

    return (

      <div>

        <h1>CHECK OUT MY PROPS OBJECT</h1>

        <h2>{stringProps}</h2>

      </div>

    );

  }

}

// ReactDOM.render goes here:

ReactDOM.render(  <PropsDisplayer />,document.getElementById('app'))

**Pass `props` to a Component**

You can *pass information* to a React component.

How? By giving that component an *attribute:*

<MyComponent foo="bar" />

Let’s say that you want to pass a component the message, "This is some top secret info.". Here’s how you could do it:

<Example message="This is some top secret info." />

As you can see, to pass information to a component, you need a *name* for the information that you want to pass.

In the above example, we used the name message. You can use any name you want.

If you want to pass information that isn’t a string, then wrap that information in curly braces. Here’s how you would pass an array:

<Greeting myInfo={["top", "secret", "lol"]} />

In this next example, we pass several pieces of information to <Greeting />. The values that aren’t strings are wrapped in curly braces:

<Greeting name="Frarthur" town="Flundon" age={2} haunted={false} />

**Instructions**

**1.**

Inside of the ReactDOM.render call, pass the string "Hello" to <PropsDisplayer />. Give that string a *name* of myProp. Feel free to use the example code as a guide.

ReactDOM.render(

  <PropsDisplayer myProp="Hello"/>,document.getElementById('app')

)

**Render a Component's props**

You just *passed* information to a component’s props object!

You will often want a component to *display* the information that you pass.

Here’s how to make a component display passed-in information:

1 - Find the *component class* that is going to receive that information.  
2 - Include this.props.name-of-information in that component class’s *render* method’s return statement.

**Instructions**

**1.**

Let’s walk through an example!

On line 11, you can see a piece of information being passed to <Greeting />. The information’s *name* is firstName.

How could you make firstName show up on the screen?

By including this.props.firstName somewhere in the Greeting class’s render method’s return statement!

On line 6, in between the returned <h1></h1> tags, add the following expression:

Hi there, {this.props.firstName}!

**2.**

In the ReactDOM.render call, change firstName‘s *value* to a different string.

Click Run. Once the browser refreshes, a new name should appear on the screen.

class Greeting extends React.Component {

  render() {

    return <h1>Hi there, {this.props.firstName}!</h1>;

  }

}

ReactDOM.render(

  <Greeting firstName='Rybu' />,

  document.getElementById('app')

);

**Pass props From Component To Component**

You have learned how to pass a prop to a component:

<Greeting firstName="Esmerelda" />

You have also learned how to access and display a passed-in prop:

render() {

return <h1>{this.props.firstName}</h1>;

}

The most common use of props is to pass information to a component, *from a different component*. You haven’t done that yet, but it’s very similar to what you’ve seen already.

In this exercise, you will pass a prop from one component to another.

**A curmudgeonly clarification about grammar:**  
You may have noticed some loose usage of the words prop and props. Unfortunately, this is pretty inevitable.

props is the name of the object that stores passed-in information. this.props refers to that storage object. At the same time, each piece of passed-in information is called a prop. This means that props could refer to two pieces of passed-in information, or it could refer to the object that stores those pieces of information :(

**Instructions**

**1.**

Your mission is to pass a prop *to* a <Greeting /> component instance, *from* an <App /> component instance.

If <App /> is going to pass a prop *to* <Greeting />, then it follows that <App /> is going to *render* <Greeting />.

Since <Greeting /> is going to be rendered by another component, that means that <Greeting /> needs to use an export statement.

In **Greeting.js**, delete this statement from line 2:

import ReactDOM from 'react-dom';

At the bottom of **Greeting.js**, remove the entire call to ReactDOM.render().

On line 3, add the word export to the beginning of the line, before the word class:

export class Greeting extends...

**2.**

<App /> can’t pass a prop to <Greeting /> until **App.js** imports the variable Greeting! Until then, the characters <Greeting /> in **App.js** might as well be nonsense.

Select **App.js**. Create a new line underneath the line import ReactDOM from 'react-dom';.

On your new line, import the Greeting component class from ./Greeting. Remember to wrap Greeting in curly braces!

**3.**

In **App.js**, add a <Greeting /> instance to App‘s render method, immediately underneath the <h1></h1>.

Give <Greeting /> an attribute with a *name* of “name.” The attribute’s *value* can be whatever you’d like.

When you click Run, <App /> will render <Greeting />, and pass it a prop!

import React from 'react';

export class Greeting extends React.Component {

  render() {

    return <h1>Hi there, {this.props.name}!</h1>;

  }

}

import React from 'react';

import ReactDOM from 'react-dom';

import { Greeting } from './Greeting';

class App extends React.Component {

  render() {

    return (

      <div>

        <h1>

          Hullo and, "Welcome to The Newzz," "On Line!"

        </h1>

        <Greeting name="Ruby" />

        <article>

          Latest newzz:  where is my phone?

        </article>

      </div>

    );

  }

}

ReactDOM.render(

  <App />,

  document.getElementById('app')

);

**Render Different UI Based on props**

Awesome! You passed a prop from a component to a different component, accessed that prop from the receiver component, and rendered it!

You can do more with props than just display them. You can also use props to make decisions.

In the code editor, look at the Welcome component class.

You can tell from this.props.name on line 5 that Welcome expects to receive a piece of information called *name*. However, Welcome never renders this piece of information! Instead, it uses the information to make a decision.

<Welcome /> instances will render the text WELCOME "2" MY WEB SITE BABYYY!!!!!, unless the user is Mozart, in which case they will render the more respectful  
hello sir it is truly great to meet you  
here on the web.

The passed-in *name* is not displayed in either case! The name is used to *decide* what will be displayed. This is a common technique.

Select **Home.js** and look at the Home component class. What will <Welcome /> render to the screen?

**Instructions**

**1.**

Select **Greeting.js**.

Look in Greeting‘s render function. You can see that Greeting now expects *two* props: name and signedIn.

Notice that this.props.signedIn is *not* located inside of a return statement. This means that Greeting will never display the value of “signedIn.” But Greeting *will* use that value to decide whether to display a friendly greeting or “GO AWAY.”

Look at Greeting until you feel like you understand how it works, and then open **App.js**.

Inside of App‘s render function, on line 12, pass <Greeting /> a second prop of signedIn={false}.

**2.**

How rude! I mean, honestly.

In **App.js**, change the value of signedIn to make <Greeting /> display a friendly greeting again.

import React from 'react';

import ReactDOM from 'react-dom';

export class Greeting extends React.Component {

  render() {

    if (this.props.signedIn == false) {

      return <h1>GO AWAY</h1>;

    } else {

      return <h1>Hi there, {this.props.name}!</h1>;

    }

  }

}

import React from 'react';

import ReactDOM from 'react-dom';

export class Greeting extends React.Component {

  render() {

    if (this.props.signedIn == false) {

      return <h1>GO AWAY</h1>;

    } else {

      return <h1>Hi there, {this.props.name}!</h1>;

    }

  }

}

**Put an Event Handler in a Component Class**

You can, and often will, pass functions as props. It is especially common to pass *event handler* functions.

In the next lesson, you will pass an event handler function as a prop. However, you have to *define* an event handler before you can pass one anywhere. In this lesson, you will define an event handler function.

How do you define an event handler in React?

You define an event handler as a method on the component class, just like the *render* method. Almost all functions that you define in React will be defined in this way, as methods in a class.

Take a look in the code editor. On lines 4 through 8, an *event handler* method is defined, with similar syntax as the render method. On line 12, that event handler method is attached to an *event* (a click event, in this case).

**Instructions**

**1.**

Select **Talker.js**.

Here we have a nice a function named talk that alerts ten thousand “blah”s to your screen. You will eventually use talk as an event handler.

There’s a problem: talk is defined outside of the Talker component class. That’s not how we do things here in React-land!

Rewrite talk, so that it is a method defined in the definition of Talker. Look at **Example.js** if you get stuck! Don’t forget to *NOT* separate talk and render with a comma.

Once you’re done, delete the original talk function before clicking Run.

import React from 'react';

import ReactDOM from 'react-dom';

import { Button } from './Button';

class Talker extends React.Component {

  talk() {

    let speech = '';

    for (let i = 0; i < 10000; i++) {

      speech += 'blah ';

    }

    alert(speech);

  }

  render() {

    return <Button />;

  }

}

ReactDOM.render(

  <Talker />,

  document.getElementById('app')

);

**Pass an Event Handler as a prop**

Good! You’ve defined a new method on the Talker component class. Now you’re ready to *pass* that function to another component.

You can pass a method in the exact same way that you pass any other information. Behold, the mighty JavaScript.

**Instructions**

**1.**

Select **Talker.js**.

You want to pass talk from here to <Button />.

If you want to pass *any* prop to <Button />, that means that you need to give <Button /> an *attribute*. Let’s start there.

Inside of Talker‘s render method, give <Button /> the following attribute:

foo="bar"

During the next two checkpoints, you’ll replace those values with the values that you need! Keep them as foo and "bar" for now.

**2.**

Your goal is to pass talk *from* <Talker /> *to* <Button />.

What prop *name* should you choose?

It doesn’t really matter! prop attributes will work with just about any name, so long as the name follows the [JavaScript variable name rules.](https://mathiasbynens.be/notes/javascript-identifiers)

Since you’re going to pass a function named talk, you might as well use talk as your *name*. Inside of the render method, change your attribute name from foo to talk.

**3.**

What should your prop *value* be?

Your prop *value* should be the information that you want to pass! In this case, you want to pass the method named talk.

Inside of the render method, change your attribute’s *value* to talk. HINT: you will need to use both curly braces and this in order to successfully access talk.

import React from 'react';

import ReactDOM from 'react-dom';

import { Button } from './Button';

class Talker extends React.Component {

  talk() {

    let speech = '';

    for (let i = 0; i < 10000; i++) {

      speech += 'blah ';

    }

    alert(speech);

  }

  render() {

    return <Button talk={this.talk} />;

  }

}

ReactDOM.render(

  <Talker />,

  document.getElementById('app')

);

**Receive an Event Handler as a prop**

Great! You just passed a function from <Talker /> to <Button />.

In the code editor, select **Button.js**. Notice that Button renders a <button></button> element.

If a user clicks on this <button></button> element, then you want your passed-in talk function to get called.

That means that you need to attach talk to the <button></button> as an *event handler*.

How do you do that? The same way that you attach any event handler to a JSX element: you give that JSX element a special *attribute*. The attribute’s *name* should be something like onClick or onHover. The attribute’s *value* should be the event handler that you want to attach.

**Instructions**

**1.**

In **Button.js**, add an onClick attribute to the render method’s <button></button>.

The onClick attribute’s *value* should be the passed-down talk function. Since you *named* your prop talk in the last exercise, you can access this prop via this.props.talk.

Click Run. Once the browser refreshes, click on the button. Ew, how annoying!

import React from 'react';

export class Button extends React.Component {

  render() {

    return (

      <button onClick={this.props.talk}>

        Click me!

      </button>

    );

  }

}

**handleEvent, onEvent, and this.props.onEvent**

Let’s talk about naming things.

When you pass an *event handler* as a prop, as you just did, there are two names that you have to choose.

Both naming choices occur in the *parent* component class - that is, in the component class that defines the event handler and passes it.

The first name that you have to choose is the name of the event handler itself.

Look at **Talker.js**, lines 6 through 12. This is our event handler. We chose to name it talk.

The second name that you have to choose is the name of the prop that you will use to *pass* the event handler. This is the same thing as your attribute name.

For our prop name, we also chose talk, as shown on line 15:

return <Button talk={this.talk} />;

These two names can be whatever you want. However, there is a naming convention that they often follow. You don’t have to follow this convention, but you should understand it when you see it.

Here’s how the naming convention works: first, think about what *type of event* you are listening for. In our example, the event type was “click.”

If you are listening for a “click” event, then you name your *event handler* handleClick. If you are listening for a “keyPress” event, then you name your event handler handleKeyPress:

class MyClass extends React.Component {

handleHover() {

alert('I am an event handler.');

alert('I will be called in response to "hover" events.');

}

}

Your prop name should be the word on, plus your event type. If you are listening for a “click” event, then you name your prop onClick. If you are listening for a “keyPress” event, then you name your prop onKeyPress:

class MyClass extends React.Component {

handleHover() {

alert('I am an event handler.');

alert('I will listen for a "hover" event.');

}

render() {

return <Child onHover={this.handleHover} />;

}

}

**Instructions**

**1.**

In **Talker.js** on line 6, change the event handler’s *name* from talk to handleClick.

**2.**

In Talker‘s render method, change the prop‘s *name* from talk to onClick.

Change the prop’s *value* to reflect the fact that the event handler is now named handleClick, not talk.

**3.**

Select **Button.js**.

Change Button‘s render function so that it expects a prop named onClick, instead of a prop named talk.

**4.**

One major source of confusion is the fact that names like onClick have special meaning, but only if they’re used on HTML-like elements.

Look at **Button.js**. When you give a <button></button> an attribute named onClick, then the name onClick has special meaning. As you’ve learned, this special onClick attribute creates an *event listener*, listening for clicks on the <button></button>:

// Button.js

// The attribute name onClick

// creates an event listner:

<button onClick={this.props.onClick}>

Click me!

</button>

Now look at **Talker.js**. Here, when you give <Button /> an attribute named onClick, then the name onClick doesn’t do anything special. The name onClick does *not* create an event listener when used on <Button /> - it’s just an arbitrary attribute name:

// Talker.js

// The attribute name onClick

// is just a normal attribute name:

<Button onClick={this.handleClick} />

The reason for this is that <Button /> is not an HTML-like JSX element; it’s a *component instance*.

Names like onClick only create event listeners if they’re used on HTML-like JSX elements. Otherwise, they’re just ordinary prop names.

import React from 'react';

export class Button extends React.Component {

  render() {

    return (

      <button onClick={this.props.onClick}>

        Click me!

      </button>

    );

  }

}

import React from 'react';

export class Button extends React.Component {

  render() {

    return (

      <button onClick={this.props.onClick}>

        Click me!

      </button>

    );

  }

}

**this.props.children**

Every component’s props object has a property named children.

this.props.children will return everything in between a component’s opening and closing JSX tags.

So far, all of the components that you’ve seen have been *self-closing tags*, such as <MyComponentClass />. They don’t have to be! You could write <MyComponentClass></MyComponentClass>, and it would still work.

this.props.children would return everything in between <MyComponentClass> and </MyComponentClass>.

Look at **BigButton.js**. In *Example 1,* <BigButton>‘s this.props.children would equal the text, “I am a child of BigButton.”

In *Example 2,* <BigButton>‘s this.props.children would equal a <LilButton /> component.

In *Example 3,* <BigButton>‘s this.props.children would equal undefined.

If a component has more than one child between its JSX tags, then this.props.children will return those children in an array. However, if a component has only one child, then this.props.children will return the single child, *not* wrapped in an array.

**Instructions**

**1.**

Select **App.js**.

Notice that App renders two <List></List> instances, and that each <List></List> has at least one <li></li> child.

Now open **List.js**, and take a look at the List component class.

Think about the fact that each List instance is going to be rendered with two JSX tags:

<List> // opening tag

</List> // closing tag

…and that there will be at least one <li></li> child in between those tags:

<List> // opening tag

<li></li> // child

</List> // closing tag

Click Run.

**2.**

You can see two list *titles* in the browser, but no list *items!* How can you make the list-items appear?

In **List.js**, in the render function, in between <ul></ul> tags, add {this.props.children}.

**3.**

BONUS: Each <List></List> instance is passed a singular title: “Living Musician” and “Living Cat Musician,” respectively. Somehow, each <List></List> counts its list-items and automatically adds an “s” to the end of its title if the count is greater than one. We could add a second piano cat, and the second list title would automatically pluralize.

See if you can figure out how the instances of the List component class are automatically pluralizing their titles!

import React from 'react';

import ReactDOM from 'react-dom';

import { List } from './List';

class App extends React.Component {

  render() {

    return (

      <div>

        <List type='Living Musician'>

          <li>Sachiko M</li>

          <li>Harvey Sid Fisher</li>

        </List>

        <List type='Living Cat Musician'>

          <li>Nora the Piano Cat</li>

        </List>

      </div>

    );

  }

}

ReactDOM.render(

  <App />,

  document.getElementById('app')

);

import React from 'react';

export class List extends React.Component {

  render() {

    let titleText = `Favorite ${this.props.type}`;

    if (this.props.children instanceof Array) {

      titleText += 's';

    }

    return (

      <div>

        <h1>{titleText}</h1>

        <ul>{this.props.children}</ul>

      </div>

    );

  }

}

**defaultProps**

Take a look at the Button component class.

Notice that on line 8, Button expects to receive a prop named text. The received text will be displayed inside of a <button></button> element.

What if nobody passes any text to Button?

If nobody passes any text to Button, then Button‘s display will be blank. It would be better if Button could display a default message instead.

You can make this happen by giving your component class a property named defaultProps:

class Example extends React.Component {

render() {

return <h1>{this.props.text}</h1>;

}

}

Example.defaultProps;

The defaultProps property should be equal to an object:

class Example extends React.Component {

render() {

return <h1>{this.props.text}</h1>;

}

}

// Set defaultProps equal to an object:

Example.defaultProps = {};

Inside of this object, write properties for any default props that you’d like to set:

class Example extends React.Component {

render() {

return <h1>{this.props.text}</h1>;

}

}

Example.defaultProps = { text: 'yo' };

If an <Example /> doesn’t get passed any text, then it will display “yo.”

If an <Example /> *does* get passed some text, then it will display that passed-in text.

**Instructions**

**1.**

Click Run.

What a sad, textless button! :(

**2.**

On line 15, give the Button component class a defaultProps property. Make this property equal to an object.

Give the defaultProps object one property, so that text‘s default value is equal to 'I am a button'.

The button’s appearance should change. Much better!

**3.**

In the ReactDOM.render() call, give <Button /> the following attribute:

text=""

Your new prop should override the default, making the <button></button> sad again :(

import React from 'react';

import ReactDOM from 'react-dom';

class Button extends React.Component {

  render() {

    return (

      <button>

        {this.props.text}

      </button>

    );

  }

}

// defaultProps goes here:

Button.defaultProps = {text: 'I am a button'}

ReactDOM.render(

  <Button text="abc"/>,

  document.getElementById('app')

);

# this.props Recap

That completes our lesson on props!

props is quite possibly the longest and most difficult lesson in all of our React courses. Congratulations on getting this far!

Here are some of the skills that you have learned:

* Passing a prop by giving an attribute to a component instance
* Accessing a passed-in prop via this.props.prop-name
* Displaying a prop
* Using a prop to make decisions about what to display
* Defining an event handler in a component class
* Passing an event handler as a prop
* Receiving a prop event handler and attaching it to an event listener
* Naming event handlers and event handler attributes according to convention
* this.props.children
* getDefaultProps

That’s a lot! Don’t worry if it’s all a bit of a blur. Soon you’ll get plenty of practice!

# State

[Dynamic information](http://www.teach-ict.com/as_a2_ict_new/ocr/AS_G061/311_data_info_knowledge/static_dynamic_data/miniweb/pg4.htm) 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.

# 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](https://hacks.mozilla.org/2015/07/es6-in-depth-classes/) [yourself](http://exploringjs.com/es6/ch_classes.html). 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.

class App extends React.Component {

  // constructor method begins here:

constructor(props) {

   super(props);

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

}

  render() {

    return (

      <h1>

        Wow this entire app is just an h1.

      </h1>

    );

  }

}

# 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.

**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.

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.

**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](https://facebook.github.io/react/docs/handling-events.html). 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() method 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.

**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 *Introduction to React.js Part II.*

**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.

**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.

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);

**6.**

In **Toggle.js** in the render method, underneath the <h1></h1>, add this JSX element:

<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?

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 : green;

    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 .render(). If .render() calls this.setState(), then an infinite loop is created.

# 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.