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.”

Click Next to enter props-land!

Instructions

In this video, you can see the Post component passing a prop to the Content component. The prop contains a string `'../images/atom.png'`, which is used to display an image.

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.

Checkpoint 2 Passed

Hint

The once-empty `<h2></h2>` will become `<h2>{stringProps}</h2>` when you're done with it.

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!

Checkpoint 3 Passed

Hint

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

PropsDisplayer.js

```
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
    myProp = "Hello"
  />,
  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.

Checkpoint 2 Passed

Hint

If you wanted to pass `myProp` with a value of `"Goodbye"`, you'd do something like this:

```
<PropsDisplayer myProp="Goodbye" />
```

Your code will look similar.

PropsDisplayer.js

```
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
    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}!
```

Checkpoint 2 Passed

Hint

The empty `<h1></h1>` will become `<h1>Hi there, {this.props.firstName}!</h1>` when you're done with it.

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.

Checkpoint 3 Passed

Hint

Find the value of the `firstName` prop, and change it to anything other than "Groberta". How about "Porthos"?

Greeting.js

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

class Greeting extends React.Component {
  render() {
    return <h1>Hi there, {this.props.firstName}!</h1>;
  }
}

ReactDOM.render(
  <Greeting firstName='Shraga' />,
  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...
```

Checkpoint 2 Passed

Hint

We're going to remove anything about `ReactDOM` from **Greeting.js**. Make sure to remove the `import` (but keep the import of `React`) and delete the call to `ReactDOM.render()`.

You'll also need to export the `Greeting` class by adding the word `export` before it.

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!

Checkpoint 3 Passed

Hint

To import `Fleeting` from `./Fleeting`, you'd write something like this:

```
import { Fleeting } from './Fleeting';
```

Your solution will look similar.

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!

Checkpoint 4 Passed

Hint

Between the `</h1>` and `<article>`, you’ll want to render a `<Greeting />` instance. Make sure to pass the `name` prop with a string value of your choice.

Greeting.js

```
import React from 'react';

export class Greeting extends React.Component {
  render() {
    return <h1>Hi there, {this.props.name}!</h1>;
  }
}
```

App.js

```
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="Shraga" />
        <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}`.

Checkpoint 2 Passed

Hint

In **App.js**, `<Greeting />` is already being passed a `name` prop. You’ll add another one: `signedIn={false}`.

2.

How rude! I mean, honestly.

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

Checkpoint 3 Passed

Hint

In **App.js**, `signedIn={false}` will become `signedIn={true}`.

Welcome.js

```
import React from 'react';

export class Welcome extends React.Component {
  render() {
    if (this.props.name === 'Wolfgang Amadeus Mozart') {
      return (
        <h2>
          hello sir it is truly great to meet you here on th
e web
        </h2>
      );
    } else {
      return (
        <h2>
```

```
          WELCOME "2" MY WEB SITE BABYYY!!!!!
        </h2>
      );
    }
  }
}
```

Home.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Welcome } from './Welcome';

class Home extends React.Component {
  render() {
    return <Welcome name='Ludwig van Beethoven' />;
  }
}

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

Greeting.js

```
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>;
    }
  }
}
```

App.js

```
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="Alison"
          signedIn={true}
        />
        <article>
          Latest:  where is my phone?
        </article>
      </div>
    );
  }
}

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

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.

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.
Checkpoint 2 Passed

Hint

The code starts looking like this:

```
function talk () {
  // ...
}

class Talker extends React.Component {
  render() {
    // ...
  }
}
```
It should end up looking like this:

```
class Talker extends React.Component {
  talk() {
    // ...
  }
  render() {
    // ...
  }
}
```

Example.js

```
import React from 'react';

class Example extends React.Component {
  handleEvent() {
    alert(`I am an event handler.
      If you see this message,
      then I have been called.`);
  }

  render() {
    return (
      <h1 onClick={this.handleEvent}>
        Hello world
      </h1>
    );
  }
}
```

Talker.js

```
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')
);
```

Button.js

```
import React from 'react';

export class Button extends React.Component {
  render() {
    return (
      <button>
        Click me!
      </button>
    );
  }
}
```

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.

Checkpoint 2 Passed

Hint

Give the `<Button />` a new prop: `foo="bar"`.

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.

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`.

Checkpoint 3 Passed

Hint

Change `foo="bar"` to `talk="bar"`.

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`.

Checkpoint 4 Passed

Hint

Change `talk="bar"` to `talk={this.talk}`.

Talker.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Button } from './Button';

class Talker extends React.Component {
  handleClick() {
    let speech = '';
    for (let i = 0; i < 10000; i++) {
      speech += 'blah ';
    }
    alert(speech);
  }

  render() {
    return <Button onClick={this.handleClick}/>;
  }
}

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

Button.js

```
import React from 'react';

export class Button extends React.Component {
  render() {
    return (
      <button onClick={this.props.onClick}>
        Click me!
      </button>
    );
  }
}
```

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!

Checkpoint 2 Passed

Hint

In **Button.js**, find the `<button></button>` element. Add an attribute to it: `onClick={this.props.talk}`.

Button.js

```
import React from 'react';

export class Button extends React.Component {
  render() {
    return (
      <button onClick={this.props.onClick}>
        Click me!
      </button>
    );
  }
}
```

Talker.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Button } from './Button';

class Talker extends React.Component {
  handleClick() {
    let speech = '';
    for (let i = 0; i < 10000; i++) {
      speech += 'blah ';
    }
    alert(speech);
  }

  render() {
    return <Button onClick={this.handleClick}/>;
  }
}

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

---

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`.

Checkpoint 2 Passed

Hint

Open **Talker.js** and look at line 6 where the `talk()` method is defined. Rename it 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`.

Checkpoint 3 Passed

Hint

Open **Talker.js** and look at the `render()` method. Rename `<Button />`'s `talk` attribute to `onClick` and set it to `{this.handleClick}`.

3.

Select **Button.js**.

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

Checkpoint 4 Passed

Hint

Open **Button.js**. Change usages of `this.props.talk` to `this.props.onClick`.

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.

Talker.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Button } from './Button';

class Talker extends React.Component {
  handleClick() {
    let speech = '';
    for (let i = 0; i < 10000; i++) {
      speech += 'blah ';
    }
    alert(speech);
  }

  render() {
    return <Button onClick={this.handleClick}/>;
  }
}

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

Button.js

```
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.

Checkpoint 2 Passed

Hint

No need to edit the code here; just 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}`.

Checkpoint 3 Passed

Hint

Find the `<ul></ul>` tags and put `{this.props.children}` inside.

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!

BigButton.js

```
import React from 'react';
import { LilButton } from './LilButton';

class BigButton extends React.Component {
  render() {
    console.log(this.props.children);
    return <button>Yo I am big</button>;
  }
}


// Example 1
<BigButton>
  I am a child of BigButton.
</BigButton>


// Example 2
<BigButton>
  <LilButton />
</BigButton>


// Example 3
<BigButton />
```

App.js

```
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>
          <li>Soso the Guitar Cat</li>
        </List>
      </div>
    );
  }
}

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

List.js

```
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! :(

Checkpoint 2 Passed

Hint

No need to edit the code here! Just click Run.

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!

Checkpoint 3 Passed

Hint

On line 15, set `Button.defaultProps` to an object. That object should look like this:

```
{ text: 'I am a button' }
```

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 :(

Checkpoint 4 Passed

Hint

Give the `<Button />` an empty `text` attribute to override the `defaultProps` you set up in the previous step.

Button.js

```
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=""/>,
  document.getElementById('app')
);
```

---

this.props Recap
That completes our lesson on `props`. Great job sticking with it!

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!