**INTRO TO JSX**

**Why React?**

**React.js** is a **JavaScript library**. It was developed by **Facebook**.

Few of the reasons to choose program with React:

* **React is *fast***. Apps made in React can **handle complex updates** and still feel quick and responsive.
* **React is *modular*.** Instead of writing large, dense files of code, you can write many smaller, reusable files. Maintenance easy.
* **React is *scalable*.** Large **programs** that display **a lot of changing data** are where **React performs best.**
* **React is *flexible*.** You can use React for interesting projects that have nothing to do with making a web app. People are still figuring out React’s potential. <https://medium.mybridge.co/22-amazing-open-source-react-projects-cb8230ec719f#.o5umedb6v>
* **React is *popular*.**

# Hello World

Take a look at the following line of code:

const h1 = <h1>Hello world</h1>;

What kind of weird hybrid **Script** code is that? Is it JavaScript? HTML? Or something else?

It seems like it must be **Java**, since it starts with **const and ends with ;.** If you tried to run that in an **HTML file**, it wouldn’t work.

However, the code **also contains <h1>Hello world</h1**>, which looks exactly like **HTML**. *That*part wouldn’t work if you tried to run **it in a JavaScript file**.

**Can we create HTML elements in a JavaScript file?**

## Answer

We absolutely can! In a JavaScript file, we can use DOM methods and properties to create (or insert, update, delete, etc) HTML elements stored as strings in a JS variable (or as a string passed to a DOM method/property).

However, when using the JavaScript library React, we will normally use **JSX** to **create HTML elements** inside a JavaScript file which uses different syntax than we would use if we were using DOM methods/properties.

# The Mystery, Revealed

**The answer is…a JavaScript file**! Despite what it looks like, your code **doesn’t actually contain** any **HTML** at all.

The part that looks like HTML, <h1>Hello world</h1>, is something **called JSX.**

# What is JSX?

JSX is **a syntax extension** for **JavaScript**. It was written to **be used with React**. JSX code looks a **lot like HTML**.

What does “syntax extension” mean?

If a JavaScript file contains JSX code, then that **file will have to be compiled**. That means that before the file reaches a web browser, a JSX compiler will translate any JSX into regular JavaScript. Because JSX is not valid JavaScript. Web browsers can’t read it!

## Question

Are there any browsers with a JSX compiler built in?

## Answer

Unfortunately, there are no browsers (currently) with a JSX compiler built in.

# JSX Elements

A basic unit of JSX is called a JSX element.

Here’s an example of a JSX element:

<h1>Hello world</h1>

This JSX element looks exactly like HTML! The only noticeable difference is that you would **find it in a JavaScript file, instead of in an HTML file.**

A JSX element is not the same thing as an HTML element - a JSX element is a description of what we want to see on our page.

# JSX Elements And Their Surroundings

JSX elements are treated as JavaScript expressions. They can go anywhere that JavaScript expressions can go.

That means that a JSX **element can be saved in a variable, passed to a function**, stored in an **object or array**…you name it.

Here’s an example of a JSX element being saved in a variable:

const navBar = <nav>I am a nav bar</nav>;

Here’s an example of several JSX elements being stored in an object:

const myTeam = {

center: <li>Benzo Walli</li>,

powerForward: <li>Rasha Loa</li>,

smallForward: <li>Tayshaun Dasmoto</li>,

shootingGuard: <li>Colmar Cumberbatch</li>,

pointGuard: <li>Femi Billon</li>

};

# Attributes In JSX

**JSX elements** can have **attributes**, just like **HTML elements can**.

A JSX attribute is written using HTML-like syntax: a name, followed by an equals sign, followed by a value. The value should be wrapped in quotes, like this:

my-attribute-name="my-attribute-value"

Here are some JSX elements with attributes:

<a href="http://www.example.com">Welcome to the Web</a>;

const title = <h1 id="title">Introduction to React.js: Part I</h1>;

A single JSX element can have many attributes, just like in HTML:

const panda = <img src="images/panda.jpg" alt="panda" width="500px" height="500px" />;

## Question

Can I create my own JSX element attributes?

## Answer

We can create our own JSX element attributes! When we create our own JSX attributes we should not include whitespace in the attribute name, instead we should use either dashes or camelCase to separate multi-word attributes, or we can use our custom multi-word attribute name as a single word.

const myH1 = <h1 my-custom-attribute="my-custom-attribute-value" myOtherCustomAttribute="my-other-custom-attribute-value" anothercustomattribute="another-custom-attribute-value">Hello React</h1>;

//all of the above attributes `my-custom-attribute`, `myOtherCustomAttribute`, and `anothercustomattribute` are valid JSX

# Nested JSX

You can nest JSX elements inside of other JSX elements, just like in HTML.

Here’s an example of a JSX <h1> element, nestedinside of a JSX <a> element:

<a href="https://www.example.com"><h1>Click me!</h1></a>

To make this more readable, you can use HTML-style line breaks and indentation:

<a href="https://www.example.com">

<h1>

Click me!

</h1>

</a>

If a JSX expression takes up more than one line, then you must **wrap the multi-line JSX expression in parentheses**.

(

<a href="https://www.example.com">

<h1>

Click me!

</h1>

</a>

)

Nested JSX expressions can **be saved as variables**, passed to functions, etc., just like non-nested JSX expressions can! Here’s an example of a nested JSX expression being saved as a variable:

const theExample = (

<a href="https://www.example.com">

<h1>

Click me!

</h1>

</a>

);

We want to use parentheses around multi-line JSX expressions to make sure we are avoiding JavaScript’s automatic semicolon insertion which will add semicolons (based on rules given by the JavaScript specifications) to terminate statements when we don’t necessarily want/expect that behavior in a JSX expression.

# JSX Outer Elements

There’s a rule that we haven’t mentioned: a JSX expression must have exactly one outermost element.

In other words, this code will work:

const paragraphs = (

<div id="i-am-the-outermost-element">

<p>I am a paragraph.</p>

<p>I, too, am a paragraph.</p>

</div>

);

**But this code will not work:**

const paragraphs = (

<p>I am a paragraph.</p>

<p>I, too, am a paragraph.</p>

);

**It’s easy to forget about this rule, and end up with errors that are tough to diagnose.**

If you notice that a JSX expression has multiple outer elements, the solution is usually simple: wrap the JSX expression in a <div></div>.

JSX expressions can be wrapped in an element other than a <div> element to be sure that the first opening tag and final closing tag of a JSX expression belong to the same element.

# Rendering JSX

Now, we understood how to write JSX elements!

To render a JSX expression means to make it appear onscreen.

**React** is a JavaScript library for **building User Interfaces** and **ReactDOM** is the JavaScript library that **allows React to interact with the DOM.**

ReactDOM.render(<h1>Hello world</h1>, document.getElementById('app'));

# ReactDOM.render() I

ReactDOM is the name of a **JavaScript library.** This library contains several React-specific methods, all of which deal with [the DOM](http://www.w3schools.com/js/js_htmldom.asp) in some way or another.

Move slightly to the right, and you can see one of ReactDOM‘s methods: ReactDOM.render().

ReactDOM.render() is the most **common way to render JSX.** It takes a JSX expression, creates a corresponding tree of DOM nodes, and adds that tree to the DOM. That is the way to make a JSX expression appear onscreen.

Move to the right a little more, and you come to this expression:

<h1>Hello world</h1>

This is the first argument being passed to ReactDOM.render(). ReactDOM.render()‘s first argument should be a JSX expression, **and it will be rendered to the screen.**

ReactDOM is not included in the generic React library. When we include the generic React library as a <script> tag in our HTML we will also have to include a <script> tag for the ReactDOM library if we want to use ReactDOM.

# ReactDOM.render() II

Move to the right a little more, and you will see this expression:

document.getElementById('app')

You just learned that ReactDOM.render() **makes its first argument appear onscreen**. But **where** on the screen should **that first argument appear**?

The first argument is appended to whatever element is selected by the second argument.

In the code editor, select **index.html**. See if you can find an element that would be selected by document.getElementById('app').

That element acted as a container for ReactDOM.render()‘s first argument! At the end of the previous exercise, this appeared on the screen:

<main id="app">

<h1>Render me!</h1>

</main>

We can have multiple calls to ReactDOM.render() in a single JavaScript file without issue - having multiple calls to ReactDOM.render() may be especially useful when **using React in an exisiting application/site**. However, when we build an application using only React, we’ll usually use a single call to ReactDOM.render(), not multiple.

# Passing a Variable to ReactDOM.render()

ReactDOM.render()‘s first argument should evaluate to a JSX expression, it doesn’t have to literally be a JSX expression.

The first argument could also be a variable, so long as that variable evaluates to a JSX expression.

In this example, we save a JSX expression as a variable named toDoList. We then pass toDoList as the first argument to ReactDOM.render():

const toDoList = (

<ol>

<li>Learn React</li>

<li>Become a Developer</li>

</ol>

);

ReactDOM.render(

toDoList,

document.getElementById('app')

);

Why use a variable as the first argument in ReactDOM.render() instead of a JSX expression?

Readability! If we are trying to pass a multi-line JSX expression to ReactDOM.render() we may find our JSX code is difficult to read and understand, however, if our JSX expression is a single line, we may want to pass the expression (instead of a variable storing the expression) to ReactDOM.render().

# The Virtual DOM

One special thing about ReactDOM.render**() is that it only updates DOM elements that have changed.**

That means that if you render the exact same thing twice in a row, the second render will do nothing:

const hello = <h1>Hello world</h1>;

// This will add "Hello world" to the screen:

ReactDOM.render(hello, document.getElementById('app'));

// This won't do anything at all:

ReactDOM.render(hello, document.getElementById('app'));

This is significant! Only updating the necessary DOM elements is a large part of what makes React so successful.

React accomplishes this thanks to something called the virtual DOM. Before moving on to the end of the lesson, <https://www.codecademy.com/articles/react-virtual-dom>

In React, for every [DOM object](http://eloquentjavascript.net/13_dom.html), there is a corresponding “virtual DOM object.” A virtual DOM object is a representation of a DOM object, like a lightweight copy.

A virtual DOM object has the same properties as a real DOM object, but it lacks the real thing’s power to directly change what’s on the screen.

Manipulating the DOM is slow. Manipulating the virtual DOM is much faster, because nothing gets drawn onscreen.

# How it helps

When you render a JSX element, every single virtual DOM object gets updated.

This sounds incredibly inefficient, but the cost is insignificant because the virtual DOM can update so quickly.

**Once the virtual DOM has updated, then React compares the virtual DOM with a virtual DOM snapshot that was taken right before the update.**

By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed. This process is called **“diffing.”**

1. The entire virtual DOM gets updated.
2. The virtual DOM gets compared to what it looked like before you updated it. React figures out which objects have changed.
3. The changed objects, and the changed objects only, get updated on the *real* DOM.
4. Changes on the real DOM cause the screen to change.

**ADVANCED JSX**

# class vs className

Let’s understand, some powerful tricks, and some common errors to avoid.

In HTML, it’s common to use class as an attribute name:

<h1 class="big">Hey</h1>

In JSX, you can’t use the word class! You have to use className instead:

<h1 className="big">Hey</h1>

This is because JSX gets translated into JavaScript, and class is a reserved word in JavaScript.

When JSX is rendered, JSX className attributes are automatically rendered as class attributes.

the two main attributes to be aware of (as we’ll use these most often and because their original HTML attribute name is a reserved JavaScript keyword) are:

* class - which will be className in JSX
* for - which will be htmlFor in JSX

# Self-Closing Tags

Another JSX ‘gotcha’ involves self-closing tags.

What’s a self-closing tag?

Most HTML elements use two tags: an opening tag (<div>), and a closing tag (</div>). However, some HTML elements such as <img> and <input> use only one tag. The tag that belongs to a single-tag element isn’t an opening tag nor a closing tag; it’s a self-closing tag.

When you write a self-closing tag in HTML, it is optional to include a forward-slash immediately before the final angle-bracket:

Fine in HTML with a slash:

<br />

Also fine, without the slash:

<br>

But!

In JSX, you have to include the slash. If you write a self-closing tag in JSX and forget the slash, you will raise an error:

Fine in JSX:

<br />

NOT FINE AT ALL in JSX:

<br>

# JavaScript In Your JSX In Your JavaScript

So far, we’ve focused on writing JSX expressions. It’s similar to writing bits of HTML, but inside of a JavaScript file.

Now we do , regular JavaScript, written inside of a JSX expression, written inside of a JavaScript file.

ReactDOM.render(

<h1>2 + 3</h1>,

document.getElementById('app')

);

# Curly Braces in JSX

The code above is not behave as we expect. Instead of adding 2 and 3, it printed out “2 + 3” as a string of text. Reason is  2 + 3 is located in between <h1> and </h1> tags.

Any code in between the tags of a JSX element will be read as JSX, not as regular JavaScript! JSX doesn’t add numbers - it reads them as text, just like HTML.

You need a way to write code that says, “Even though I am located in between JSX tags, treat me like ordinary JavaScript and not like JSX.” You can do this by wrapping your code in curly braces.

ReactDOM.render(

<h1>{2 + 3}</h1>,

document.getElementById('app') );

**20 Digits of Pi in JSX**

You can now inject regular JavaScript into JSX expressions! This will be extremely useful.

const math = <h1>2 + 3 = {2 + 3}</h1>;

ReactDOM.render(

math,

document.getElementById('app') );

# Variables in JSX

When you inject JavaScript into JSX, that JavaScript is part of the same environment as the rest of the JavaScript in your file.

That means that you can access variables while inside of a JSX expression, even if those variables were declared on the outside.

// Declare a variable:

const name = 'Gerdo';

// Access your variable

// from inside of a JSX expression:

const greeting = <p>Hello, {name}!</p>;

**Example:**

const theBestString = 'tralalalala i am da best';

ReactDOM.render(<h1>{theBestString}</h1>, document.getElementById('app'));

# Variable Attributes in JSX

When writing JSX, it’s common to use variables to set attributes.

Here’s an example of how that might work:

// Use a variable to set the `height` and `width` attributes:

const sideLength = "200px";

const panda = (

<img

src="images/panda.jpg"

alt="panda"

height={sideLength}

width={sideLength} />

);

Notice how in this example, the <img />‘s attributes each get their own line. This can make your code more readable if you have a lot of attributes on one element.

Object properties are also often used to set attributes:

const pics = {

panda: "http://bit.ly/1Tqltv5", // panda pic source

owl: "http://bit.ly/1XGtkM3",

owlCat: "http://bit.ly/1Upbczi"

};

const panda = (

<img

src={pics.panda}

alt="Lazy Panda" />

);

const owl = (

<img

src={pics.owl}

alt="Unimpressed Owl" />

);

const owlCat = (

<img

src={pics.owlCat}

alt="Ghastly Abomination" />

);

**Example:**

import React from 'react';

import ReactDOM from 'react-dom';

const goose = 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-goose.jpg';

// Declare new variable here:

const gooseImg = <img src={goose} />;

ReactDOM.render(gooseImg, document.getElementById('app'));

The decision to use a variable to set an attribute vs. using object properties to set attributes will depend on how many attributes with different values we would like to set - if we only need to set a single attribute value (or the same attribute value on multiple elements) we can easily use a variable, but if we need to set multiple attribute values, using object properties may be a better choice.

# Event Listeners in JSX

JSX elements can have event listeners, just like HTML elements can. Programming in React means constantly working with event listeners.

You create an event listener by giving a JSX element a special attribute. Here’s an example:

<img onClick={myFunc} />

An event listener attribute’s name should be something like onClick or onMouseOver: the word on, plus the type of event that you’re listening for. You can see a list of valid event names <https://reactjs.org/docs/events.html#supported-events>

An event listener attribute’s value should be a function. The above example would only work if myFunc were a valid function that had been defined elsewhere:

function myFunc() {

alert('Make myFunc the pFunc... omg that was horrible i am so sorry');

}

<img onClick={myFunc} />

Note that in HTML, event listener names are written in all lowercase, such as onclick or onmouseover. In JSX, event listener names are written in camelCase, such as onClick or onMouseOver.

**Example:**

import React from 'react';

import ReactDOM from 'react-dom';

function makeDoggy(e) {

// Call this extremely useful function on an <img>.

// The <img> will become a picture of a doggy.

e.target.setAttribute('src', 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-puppy.jpeg');

e.target.setAttribute('alt', 'doggy');

}

const kitty = (

    <img

        src="https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-kitty.jpg"

        alt="kitty" onClick={makeDoggy} />

);

ReactDOM.render(kitty, document.getElementById('app'));

# JSX Conditionals: If Statements That Don't Work

Here’s a rule that you need to know: you can not inject an if statement into a JSX expression.

This code will break:

(

<h1>

{

if (purchase.complete) {

'Thank you for placing an order!'

}

}

</h1>

)

The reason why has to do with the way that JSX is compiled. You don’t need to understand the mechanics of it for now, but if you’re interested then you can learn more <https://reactjs.org/docs/jsx-in-depth.html>

What if you want a JSX expression to render, but only under certain circumstances? You can’t inject an if statement. What can you do?

You have lots of options.

# JSX Conditionals: If Statements That Do Work

Well, one option is to write an if statement, and not inject it into JSX.

Look at **if.js**. Follow the if statement, all the way from line 6 down to line 18.

**if.js** works, because the words if and else are not injected in between JSX tags. The ifstatement is on the outside, and no JavaScript injection is necessary.

import React from 'react';

import ReactDOM from 'react-dom';

function coinToss() {

// This function will randomly return either 'heads' or 'tails'.

return Math.random() < 0.5 ? 'heads' : 'tails';

}

const pics = {

kitty: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-kitty.jpg',

doggy: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-puppy.jpeg'

};

let img;

// if/else statement begins here:

if (coinToss() === 'heads') {

img = (

<img src={pics.kitty} />

);

} else {

img = (

<img src={pics.doggy} />

);

}

ReactDOM.render(img, document.getElementById('app'));

# JSX Conditionals: The Ternary Operator

There’s a more compact way to write conditionals in JSX: the ternary operator.

The ternary operator works the same way in React as it does in regular JavaScript. However, it shows up in React surprisingly often.

Recall how it works: you write x ? y : z, where x, y, and z are all JavaScript expressions. When your code is executed, x is evaluated as either “truthy” or “falsy.” If x is truthy, then the entire ternary operator returns y. If x is falsy, then the entire ternary operator returns z. <https://stackoverflow.com/questions/6259982/how-do-you-use-the-conditional-operator-in-javascript> Perfect Explanation

Here’s how you might use the ternary operator in a JSX expression:

const headline = (

<h1>

{ age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff' }

</h1>

);

In the above example, if age is greater than or equal to drinkingAge, then headline will equal <h1>Buy Drink</h1>. Otherwise, headline will equal <h1>Do Teen Stuff</h1>.

**Example:**

import React from 'react';

import ReactDOM from 'react-dom';

function coinToss () {

// Randomly return either 'heads' or 'tails'.

return Math.random() < 0.5 ? 'heads' : 'tails';

}

const pics = {

kitty: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-kitty.jpg',

doggy: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-puppy.jpeg'

};

const img = <img src={pics[coinToss() === 'heads' ? 'kitty' : 'doggy']} />;

ReactDOM.render(

    img,

    document.getElementById('app')

);

# JSX Conditionals: &&

Final way of writing conditionals in React: the && operator.

Like the ternary operator, && is not React-specific, but it shows up in React surprisingly often.

In the last two lessons, you wrote statements that would sometimes render a kitty and other times render a doggy. && would not have been the best choice for those lessons.

&& works best in conditionals that will sometimes do an action, but other times do nothing at all.

Here’s an example:

const tasty = (

<ul>

<li>Applesauce</li>

{ !baby && <li>Pizza</li> }

{ age > 15 && <li>Brussels Sprouts</li> }

{ age > 20 && <li>Oysters</li> }

{ age > 25 && <li>Grappa</li> }

</ul>

);

Every time that you see && in this example, either some code will run, or else no code will run.

# .map in JSX

The array method .map() comes up often in React. It’s good to get in the habit of using it alongside JSX.

If you want to create a list of JSX elements, then .map() is often your best bet. It can look odd at first:

const strings = ['Home', 'Shop', 'About Me'];

const listItems = strings.map(string => <li>{string}</li>);

<ul>{listItems}</ul>

In the above example, we start out with an array of strings. We call .map() on this array of strings, and the .map() call returns a new array of <li>s.

On the last line of the example, note that {listItems} will evaluate to an array, because it’s the returned value of .map()! JSX <li>s don’t have to be in an array like this, but they can be.

// This is fine in JSX, not in an explicit array:

<ul>

<li>item 1</li>

<li>item 2</li>

<li>item 3</li>

</ul>

// This is also fine!

const liArray = [

<li>item 1</li>,

<li>item 2<li>,

<li>item 3</li>

];

<ul>{liArray}</ul>

**Example:**

import React from 'react';

import ReactDOM from 'react-dom';

const people = ['Rowe', 'Prevost', 'Gare'];

const peopleLis = people.map(person => <li>{person}</li>

);

ReactDOM.render(<ul>{peopleLis}</ul>, document.getElementById('app'))

**Example: forEach**

We can use .forEach() to create a list of JSX elements from an array as long as we are using .forEach() from outside a JSX expression as .forEach() does not evaluate to a value like .map()does. That said, .forEach() takes a little more code to do the same thing that .map() can do.

const myArr = ['one', 'two', 'three'];

const myArrMadeFromForEach = []; // we create a new array which will evaluate to a value when we inject it into a JSX expression

myArr.forEach((item, i) => myArrMadeFromForEach.push(<li key={item+i}>{item}</li>)); // we push a JSX element containing a value to our 'myArrMadeFromForEach' because `.forEach()` does not return any value, nor does it mutate the array on which it is called

const myList = (

<ul>{myArrMadeFromForEach}</ul> // `myArrMadeFromForEach` will evaluate to an array of `<li>` elements

)

ReactDOM.render(myList, document.getElementById('app'));

**Example :2 using map**

import React from 'react';

import ReactDOM from 'react-dom';

const myArr = ['one', 'two', 'three'];

const myArrCreatedFromMap = myArr.map((item, i) => (<li key={item + i}>{item}</li>)); // `.map()` creates/returns a new array from calling a function on every element in the array it's called on

const myList = (

<ul>{myArrCreatedFromMap}</ul> // `myArrCreatedFromMap` will evaluate to a list of `<li>` elements

)

ReactDOM.render(myList, document.getElementById('app'));

**Example:3 map**

import React from 'react';

import ReactDOM from 'react-dom';

const myArr = ['one', 'two', 'three'];

const myList = (

<ul>{myArr.map((item, i) => <li key={item + i}>{item}</li>)}</ul> // `.map()` creates/returns a new array from calling a function on every element in the array it's called on

)

ReactDOM.render(myList, document.getElementById('app'));

# Keys

When you make a list in JSX, sometimes your list will need to include something called keys:

<ul>

<li key="li-01">Example1</li>

<li key="li-02">Example2</li>

<li key="li-03">Example3</li>

</ul>

A key is a JSX attribute. The attribute’s name is key. The attribute’s value should be something unique, similar to an id attribute.

keys **don’t do anything** that you can see! React uses them internally to keep track of lists. If you don’t use keys when you’re supposed to**, React might accidentally scramble your list-items into the wrong order**.

Not all lists need to have keys. A list needs keys if either of the following are true:

1. The list-items have memory from one render to the next. For instance, when a to-do list renders, each item must “remember” whether it was checked off. The items shouldn’t get amnesia when they render.
2. A list’s order might be shuffled. For instance, a list of search results might be shuffled from one render to the next.

If neither of these conditions are true, then you don’t have to worry about keys. If you aren’t sure then it never hurts to use them!

We can use the index of an element as the key attribute’s value, however, using only the index of an element as the key may cause some issues as the order of the elements may change and potentially render incorrect data specifically when a list can be reordered or filtered. When a list of elements can be reordered or filtered we should use more unique keys for the elements.

**Example:**

**Before:**

const people = ['Rowe', 'Prevost', 'Gare'];

const peopleLis = people.map(person =>

// expression goes here:

<li>{person}</li>

);

**After:**

import React from 'react';

import ReactDOM from 'react-dom';

const people = ['shubham', 'junu darling', 'Gare'];

const peopleLis = people.map((person, i) =>

// expression goes here:

<li key={'person\_' + i}>{person}</li>

);

// ReactDOM.render goes here:

ReactDOM.render(<ul>{peopleLis}</ul>, document.getElementById('app'));

# React.createElement

You can write React code without using JSX at all!

The majority of React programmers do use JSX, and we will use it for the remainder of this tutorial, but you should understand that it is possible to write React code without it.

The following JSX expression:

const h1 = <h1>Hello world</h1>;

can be rewritten without JSX, like this:

const h1 = React.createElement(

"h1",

null,

"Hello, world"

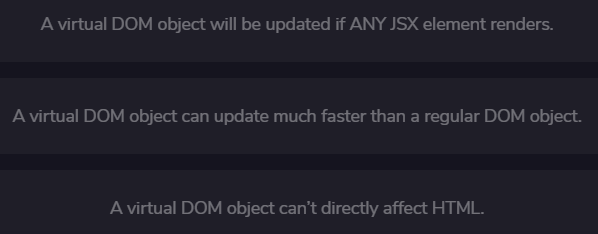
);

When a JSX element is compiled, the compiler transforms the JSX element into the method that you see above: React.createElement(). **Every JSX element is secretly a call to**React.createElement().

We won’t go in-depth into how React.createElement() works, but you can start with the <https://reactjs.org/docs/react-api.html#react.createelement>  if you’d like to learn more!

We would use React.createElementI() instead of JSX when we do not want to set up compilation for our project, which the use of JSX requires!

What’s a difference between a DOM object and a virtual DOM object?



What’s wrong with this code?

let skateboardDog = (

<img src="alfie.jpg" />

<h1>Hiya kids! I'm a dog on a skateboard.</h1>

);

🡪JSX expressions need an outermost element.

Place the following steps in the right order:

a. The screen looks different than it used to.

b. A JSX element renders.

c. The virtual DOM "diffs," comparing its current self with its previous self.

d. The entire virtual DOM updates.

e. Part of the real DOM updates.

**🡪 B,D,C,E,A**

What should you pass to ReactDOM.render() for its first argument?

🡪A JSX expression that you want to render.

What should you pass to ReactDOM.render() for its second argument?

🡪A selector that matches an HTML element.

**YOUR FIRST REACT COMPONENT**

# Hello World, Part II... THE COMPONENT

React applications are made out of components.

A **component** is a **small**, **reusable** chunk of code that is **responsible** **for one job.** That job is often to render some HTML.

This code will **create and render a new React component**:

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponentClass extends React.Component {

render() {

return <h1>Hello world</h1>;

}

};

ReactDOM.render(

<MyComponentClass />,

document.getElementById('app')

);

A lot of that code is probably unfamiliar. You can recognize some JSX in there, as well as ReactDOM.render().

you can declare multiple components within a single file. For example, there can be a file containing the following components,

**class** **Class1** **extends** **React**.**Component** {

render() { ... }

};

**class** **Class2** **extends** **React**.**Component** {

render() { ... }

};

# Import React

Take a look at the code on line 1:

import React from 'react'; // React variable import javascript obj.

This line of code **creates a new variable**. That variable’s name is React, and its value is a particular**, imported JavaScript object**:

// create a variable named React:

import React from 'react';

// evaluate this variable and get a particular, imported JavaScript object:

React // { imported object properties here... }

*This imported object contains* **methods** that you need in order to use React. The **object** is called the **React library.**

Later, we’ll go over where the React library is imported from, and how the importing process works.

For now, just know that you **get the React library** via import React from 'react';.

You’ve already seen one of the **methods** contained in the React library: **React.createElement().** Recall that when a JSX element is compiled, it transforms into a React.createElement() call.

For this reason, you have to import the React library, and save it in a variable named React, before you can use any JSX at all.React.createElement() must be available in order for JSX to work.

 you import multiple objects from a library at once you can do this using “unpacking” when importing from a library.

For example, say that you wanted to import multiple functions from some library, you could do so as follows,

**import** { functionA, functionB, functionC } **from** 'library';

Another, more concrete, example is importing objects from the ‘react-router’ library.

**import** { Router, Switch } **from** 'react-router';

# Import ReactDOM

Now take a look at the code on line 2:

import ReactDOM from 'react-dom';

This line of code is very similar to line 1.

Lines 1 and 2 both import JavaScript objects. In both lines, the **imported object contains React-related methods.**

However, there is a difference!

The methods imported from 'react-dom' are meant for interacting with the DOM. You are already familiar with one of them:ReactDOM.render().

The methods imported from 'react' don’t deal with the DOM at all. They don’t engage directly with anything that isn’t part of React.

To clarify: the **DOM** is **used** in **React** applications, **but** it isn’t part of React. After all, the DOM is also used in countless non-React applications**. Methods imported from 'react' are only for pure React purposes,** such as creating components or writing JSX elements.

# Create a Component Class

React component is a small, reusable chunk of code that is responsible for one job, which often involves rendering HTML. Here’s another fact about components**: every component must come from a component class.**

A component class is **like a factory that creates components**. If you have a component class, then you can use that class to produce as many components as you want.

To make a component class, **you use a base class from the React** library: React.Component.

**The React.Component class provides several methods in addition to render(), some of which are automatically invoked through the lifecycle of the component.**

**The constructor() method is invoked by the component before it is added to the DOM. This is usually used to run code to set up the component such as initializing some state and binding methods to the component.**

**The class also provides a componentDidMount() method which is run right after the component is inserted into the DOM tree, and is invoked only one time.**

**Another method is componentDidUpdate(), which runs immediately after any updates to the component occurs. This is typically used for code such as network requests which should be run if any updates occur to the component.**

React.Component is a JavaScript class. To create your own component class, **you must subclass** React.Component. You can do this by using the syntax class YourComponentNameGoesHere extends React.Component {}.

JavaScript classes and subclassing are a complex topic If you aren’t comfortable with them, here are some good refrences.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

<https://hacks.mozilla.org/2015/07/es6-in-depth-classes/>

<https://hacks.mozilla.org/2015/08/es6-in-depth-subclassing/>

<http://exploringjs.com/es6/ch_classes.html>

import React from 'react';

import ReactDOM from 'react-dom';

class x extends React.Component {

}

On line 4, you know that you are declaring a new component class, which is like a factory for building React components. You know that React.Component is a class, which you must subclass in order to create a component class of your own. You also know that React.Component is a property on the object which was returned by import React from 'react' on line 1.

# Name a Component Class

When you declare a new component class, you need to give that component class a name. On line 4, notice that our component class’s name is MyComponentClass.

Component class variable names must begin with capital letters!

Class names should be nouns in Upper[CamelCase](https://en.wikipedia.org/wiki/CamelCase), with the first letter of every word capitalised. Use whole words — avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

* class Raster {}
* class ImageSprite {}

In addition, to keep the files organized, it is good to keep only one component per file, and name files after the component they contain. So, if you named your file board.js, then the contained class is most likely named Board. This may not always be applicable, such as if a component is declared in the app.js file, however.

In addition, there is a **React-specific reason** why component class names must always be capitalized.

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponentClass extends React.Component {

}

# Component Class Instructions

* On line 1, import React from 'react' creates a JavaScript object. This object contains properties that are needed to make React work, such as React.createElement() and React.Component.
* On line 2, import ReactDOM from 'react-dom'creates another JavaScript object. This object contains methods that help React interact with the DOM, such as ReactDOM.render().
* On line 4, by subclassing React.Component, you create a new component class. This is not a component! A component class is more like a factory that produces components. When you start making components, each one will come from a component class.
* component class a name. That name should be written in UpperCamelCase. In this case, your chosen name is MyComponentClass.

The body of your component class: the pair of curly braces after React.Component, and all of the code between those curly braces.The body will act as a set of instructions, explaining to your component class how it should build a React component.

Here’s what your class body would look like on its own, without the rest of the class declaration syntax. Find it in **app.js**:

{

render() {

return <h1>Hello world</h1>;

}

}

That doesn’t look like a set of instructions explaining how to build a React component! Yet that’s exactly what it is.

Click Next, and we’ll go into how these instructions work.

# The Render Function

A component class is like a factory that builds components. It **builds** these components **by consulting a set of instructions**

For starters, these instructions will be delimited by curly braces, like this:

class ComponentFactory extends React.Component {

// instructions go here, between the curly braces

}

There is only one property that you have to include in your instructions: a render method.

A render method is a **property** whose **name** is **render**, and whose **value** is a **function**. The term “render method” can refer to the entire property, or to just the function part.

class ComponentFactory extends React.Component {

render() {}

}

A render method must contain a returnstatement. Usually, this return statement returns a JSX expression:

class ComponentFactory extends React.Component {

render() {

return <h1>Hello world</h1>;

}

}

**You can return more than one element** in the render function.

In previous versions of React, if you wanted to do so, you had to wrap them inside another element, such as a <div> element. For example, you would have to do something like the following,

render() {

**return** (

<**div**>

<**Element1** />

<**Element2** />

</**div**>

);

}

However, the downside is that this added another element to the DOM, even if you did not want to add the <div> element.

In the current versions, you can utilize what is known as a React.Fragment, which lets you return multiple elements without the need to create a wrapper element. It would look like the following,

render() {

**return** (

<**React.Fragment**>

<**Element1** />

<**Element2** />

</**React.Fragment**>

);

}

# Create a Component Instance

MyComponentClass is now a working component class! It’s ready to follow its instructions and make some React components.

So, let’s make a React component! It only takes one more line:

<MyComponentClass />

To make a React component, you write a JSX element. Instead of naming your JSX element something like h1 or div like you’ve done before, give it the same name as a component class. Voilà, there’s your component instance!

JSX elements can be either HTML-like, or component instances. JSX uses capitalization to distinguish between the two! That is the React-specific reason why component class names must begin with capital letters. In a **JSX element, that capitalized first letter says, “I will be a component instance and not an HTML tag.”**

When running the ReactDOM.render() method, you can only pass in one value as the first argument, which is the component to be rendered. However, you can create more than one component instance at a time, by simply wrapping them inside of another element, such as a <div>. For example, you could do the following to render two components,

ReactDOM.render(

<**div**>

<**ComponentOne** />

<**ComponentTwo** />

</**div**>,

document.getElementById('app')

);

# Render A Component

class MyComponentClass extends React.Component

{ // everything in between these curly-braces is instructions for how to build components

render() {

return <h1>Hello world</h1>;

}

}

What happening here is a class is declared as  MyComponentClass a component. Which is having a **render method** and it has a basic javaScript syntax.

<MyComponentClass />, what do these instructions do?

Whenever you make a component, that component **inherits all of the methods of its component class**. MyComponentClass has one method: MyComponentClass.render()**. Therefore, <MyComponentClass /> also has a method named render.**

You could make a million different <MyComponentClass /> **instances,** and each one would inherit this same exact render method.

Since your component has a render method, all that’s left to do is call it. This happens in a slightly unusual way.

To **call a component’s** render method, you pass that component to ReactDOM.render(). Notice your component, being passed as ReactDOM.render()‘s first argument:

ReactDOM.render(

<MyComponentClass />,

document.getElementById('app')

);

ReactDOM.render() will tell <MyComponentClass />to call its render method.

<MyComponentClass /> will call its render method, which will return the JSX element <h1>Hello world</h1>. ReactDOM.render() will then take that resulting JSX element, and add it to the virtual DOM. This will make “Hello world” appear on the screen.

When the function ReactDOM.render() is first called, all contents of the container are replaced. As a result, if the container element already had some elements within it, they would be completely replaced by the new component.

For instance, say that the container element in the DOM already has contents like so,

<**main** id="app">

<**div**>Hi there!</**div**>

</**main**>

And say that we are rendering a new component into this container,

ReactDOM.render(

<MyComponentClass />,

document.getElementById('app')

);

When it renders the component, all the contents of the container will be replaced with just the <MyComponentClass /> component.

**COMPONENTS AND ADVANCED JSX**

# Use Multiline JSX in a Component

In this lesson, you will learn some common ways that JSX and React components work together. You’ll get more comfortable with both JSX and components, while picking up some new tricks.

Take a look at this HTML:

<blockquote>

<p>

The world is full of objects, more or less interesting; I do not wish to add any more.

</p>

<cite>

<a target="\_blank"

href="https://en.wikipedia.org/wiki/Douglas\_Huebler">

Douglas Huebler

</a>

</cite>

</blockquote>

How might you make a React component that renders this HTML?

Select **QuoteMaker.js** to see one way of doing it.

return <h1>Hello world</h1>;

However, a multi-line JSX expression should always be wrapped in parentheses! That is why here, we return statement has parentheses around it.

**Example:**

import React from 'react';

import ReactDOM from 'react-dom';

class QuoteMaker extends React.Component {

render() {

return (

<blockquote>

<p>

What is important now is to recover our senses.

</p>

<cite>

<a target="\_blank"

href="https://en.wikipedia.org/wiki/Susan\_Sontag">

Susan Sontag

</a>

</cite>

</blockquote>

);

}

};

ReactDOM.render(

<QuoteMaker />,

document.getElementById('app')

);

# Use a Variable Attribute in a Component

Take a look at this JavaScript object named redPanda:

const redPanda = {

src: 'https://upload.wikimedia.org/wikipedia/commons/b/b2/Endangered\_Red\_Panda.jpg',

alt: 'Red Panda',

width: '200px

};

How could you render a React component, and get a picture with redPanda‘s properties?

Select **RedPanda.js** to see one way to do it.

**Note all of the curly-brace JavaScript injections inside of the render function**! These are use JavaScript injections.

You can, and often will, inject JavaScript into JSX inside of a render function.

**Example1:**

import React from 'react';

import ReactDOM from 'react-dom';

const owl = {

title: 'Excellent Owl',

src: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-owl.jpg'

};

// Component class starts here:

class Owl extends React.Component {

render() {

return (

<div>

<h1>{owl.title}</h1>

<img src={owl.src} alt={owl.title}/>

// use JavaScript injections

</div>

) }}

ReactDOM.render(

<Owl />,

document.getElementById('app'));

**Example 2:**

import React from 'react';

import ReactDOM from 'react-dom';

const redPanda = {

src: 'https://upload.wikimedia.org/wikipedia/commons/b/b2/Endangered\_Red\_Panda.jpg',

alt: 'Red Panda',

width: '200px'

};

class RedPanda extends React.Component {

render() {

return (

<div>

<h1>Cute Red Panda</h1>

<img

src={redPanda.src}

alt={redPanda.alt}

width={redPanda.width} />

//use JavaScript injections

</div>

); }}

ReactDOM.render(

<RedPanda />,

document.getElementById('app')

);

# Put Logic in a Render Function

A render() function must have a returnstatement. However, that isn’t all that it can have.

A render() function can also be a fine place to put simple calculations that need to happen right before a component renders. Here’s an example of some calculations inside of a render function:

class Random extends React.Component {

render() {

// First, some logic that must happen

// before rendering:

const n = Math.floor(Math.random() \* 10 + 1);

// Next, a return statement

// using that logic:

return <h1>The number is {n}!</h1>;

}

}

Watch out for this common mistake:

class Random extends React.Component {

// This should be in the render function:

const n = Math.floor(Math.random() \* 10 + 1);

render() {

return <h1>The number is {n}!</h1>;

}

};

In the above example, the line with the const ndeclaration will cause a syntax error, **as is it should not be part of the class declaration** itself, but should occur in a method like render().

**Example:1 LOGIC**

// New component class starts here:

class Friend extends React.Component {

render() {

let friend = friends[2];

    return (

    <div>

<h1>{friend.title}</h1>

<img src={friend.src} />

            </div>

  );

    }

}

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

**Example2:**

import React from 'react';

import ReactDOM from 'react-dom';

const friends = [

{

title: "Yummmmmmm",

src: "https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-monkeyweirdo.jpg"

},

{

title: "Hey Guys! Wait Up!",

src: "https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-earnestfrog.jpg"

},

{

title: "Yikes",

src: "https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-alpaca.jpg"

}

];

// New component class starts here:

class Friend extends React.Component {

render() {

let friend = friends;

  return (

    <div>

<h1>{friend[0].title}</h1>

<img src={friend[0].src} />

             <h1>{friend[1].title}</h1>

<img src={friend[1].src} />

<h1>{friend[2].title}</h1>

<img src={friend[2].src} />

            </div>

 ); }

}

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

# Use a Conditional in a Render Function

How might you use a conditional statement inside of a render() function?

class TodaysPlan extends React.Component {

render() {

let task;

if (!apocalypse) {

task = 'learn React.js'

} else {

task = 'run around'

}

return <h1>Today I am going to {task}!</h1>;

}

}

ReactDOM.render(

<TodaysPlan />,

document.getElementById('app')

);

Notice that the if statement is located inside of the render function, but before the returnstatement. This is pretty much the only way that you will ever see an if statement used in a render function.

**Example:**

import React from 'react';

import ReactDOM from 'react-dom';

const fiftyFifty = Math.random() < 0.5;

// New component class starts here:

class TonightsPlan extends React.Component {

render() {

if (fiftyFifty===true)

{

return (<h1>Tonight I'm going out WOOO</h1>)

}

else

{

return <h1>Tonight I'm going to bed WOOO</h1>

}

}}

ReactDOM.render(

    <TonightsPlan />,

    document.getElementById('app'));

# Use this in a Component

The word this gets used in React a lot!

You are especially likely to see this inside of the body of a component class declaration. Here’s an example:

Example: with getter function

import React from 'react';

import ReactDOM from 'react-dom';

class MyName extends React.Component {

    // name property goes here:

get food() {

  return 'Shubham';

}

render() {

return <h1>My name is {this.food}.</h1>;

}

}

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

The simple answer is that this refers to an **instance** of MyName. It is almost inevitable that this object will be an instance of MyName, but technically it could be something else.

Let’s assume that this refers to an instance of your component class, as will be the case in all examples in this course. MyName has two methods: .food and .render().

Since this will evaluate to an instance of MyName, this.food will evaluate to a call of MyName ‘s .food method. This method will, in turn, evaluate to the string “ice cream.”

Why don’t you need parentheses after this.food? Shouldn’t it be this.food()?

You don’t need those parentheses because .foodis a getter method. You can tell this from the getin the above class declaration body.

There’s nothing React-specific about getter methods, nor about this behaving in this way! However, in React you will see this used in this way almost constantly.

this in JavaScript can be a difficult concept! Here is a good resource for <https://dmitripavlutin.com/gentle-explanation-of-this-in-javascript/>

**Example: with function**

import React from 'react';

import ReactDOM from 'react-dom';

class MyName extends React.Component {

    // name property goes here:

name() {

  return 'Shubham';

}

render() {

return <h1>My name is {this.name()}.</h1>;

}

}

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

# Use an Event Listener in a Component

Render functions often contain event listeners. Here’s an example of an event listener in a render function:

render() {

return (

<div onHover={myFunc}>

</div>

);

}

Recall that an event handler is a function that gets called in response to an event. In the above example, the event handler is myFunc().

In React, you define event handlers as methodson a component class. Like this:

class MyClass extends React.Component {

myFunc() {

alert('Stop it. Stop hovering.');

}

render() {

return (

<div onHover={this.myFunc}>

</div>

);

}

}

Notice that the component class has two methods: .myFunc() and .render(). .myFunc() is being used as an event handler. .myFunc() will be called any time that a user hovers over the rendered <div></div>.

**Example:**

every event call takes argument as a function and elements takes functions as a property

import React from 'react';

import ReactDOM from 'react-dom';

class Button extends React.Component {

scream() {

alert('AAAAAAAAHHH!!!!!');

}

scr() {

return "kutte ke pille"

}

render() {

return(

<div>

// every event call takes argument as a function

<button onClick={this.scream}>AAAAAH</button>;

<h1> {this.scr()}</h1>;

// elements takes functions as a property

</div>

)}}

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

**Example: Project Demo**

import React from 'react';

import ReactDOM from 'react-dom';

class Contact extends React.Component {

constructor(props) {

super(props);

this.state = {

password: 'swordfish',

authorized: false

};

this.authorize = this.authorize.bind(this);

}

authorize(e) {

const password = e.target.querySelector(

'input[type="password"]').value;

const auth = password == this.state.password;

this.setState({

authorized: auth

});

}

render() {

let login = (<form action="#"

onSubmit={this.authorize}>

<input type="password" placeholder="Password"/>

<input type="submit" /> </form>)

const contactInfo = (

<ul>

<li>

client@example.com

</li>

<li>

555.555.5555

</li>

</ul>

);

return (

<div id="authorization">

<h1>{this.state.authorized ? 'Contact' : 'Enter the Password'}</h1>

    {this.state.authorized ? contactInfo : login }

</div>

);

}

}

ReactDOM.render(

<Contact />,

document.getElementById('app')

);

**COMPONENTS RENDER OTHER COMPONENTS**

# Components Interact

A React application can contain dozens, or even hundreds, of components.

Each **component** might be small and **relatively unremarkable** on its own. When **combined**, however, they can form **enormous, fantastically complex ecosyst**ems of information. In other words, **React apps** are made out of **components,** but what makes React special isn’t components themselves. What makes **React special** is the **ways** in which **components interact.**

# A Component in a Render Function

Here is a .render() method that returns an HTML-like JSX element:

class Example extends React.Component {

render() {

return <h1>Hello world</h1>;

}

}

Render methods can also **return another kind of JSX: component instances.**

class OMG extends React.Component {

render() {

return <h1>Whooaa!</h1>;

}

}

class Crazy extends React.Component {

render() {

return <OMG />;

}

}

 Crazy‘s render method returnsan **instance of the OMG component class.**

# Apply a Component in a Render Function

This is new territory! You’ve never seen a component rendered by another component before.

You have seen a component rendered before, though, but not by another component. Instead, you’ve seen a component rendered by ReactDOM.render().

When a component renders another component, what happens is very similar to what happens when ReactDOM.render() renders a component.

# 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 NavBarcomponent 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/). More on all of that later.

Refrences:

<https://hacks.mozilla.org/2015/08/es6-in-depth-modules/>

<http://eloquentjavascript.net/10_modules.html>

<http://exploringjs.com/es6/ch_modules.html>

# export

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 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, 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.

Way-2

SearchBar.js

export default SearchBar;

App.js

import SearchBar from './components/SearchBar/SearchBar';

**THIS.PROPS**

# this.props

Let’s 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.

# 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." />

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.**

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} />

# Render a Component's props

You just passed information to a component’s propsobject!

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

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**.

Example:

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='Groberta' />,

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.

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 :(

**Example:**

**Greeting.js**

import React from 'react';

//import ReactDOM from 'react-dom';

export class Greeting extends React.Component {

render() {

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

}

}

/\*

ReactDOM.render(

<Greeting name='Groberta' />,

document.getElementById('app')

);

\*/

**Apps.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { Greeting } from './Greeting'; // important

class App extends React.Component {

render() {

return (

<div>

<h1>

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

</h1>

<Greeting name="Ruby"> // important

<article>

Latest newzz: where is my phone?

</article>

</div>

);

}

}

ReactDOM.render(

<App />, // important

document.getElementById('app')

);

# Render Different UI Based on props

Do more with props , **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?

**Example1:**

**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')

);

**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 the web

  </h2>

);

} else {

return (

  <h2>

   WELCOME "2" MY WEB SITE BABYYY!!!!!

  </h2>

);

}

}

}

**Example 2:**

**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>;

  }

}

}

/\*

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}.

\*/

**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**.

However, you have to **define** an event **handler before you can pass one** anywhere.

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.

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>

);

}

}

/\*

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).

\*/

# Pass an Event Handler as a prop

. Now we are ready to pass that function to another component.

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

**Example:**

**Button.js**

import React from 'react';

export class Button extends React.Component {

render() {

return (

<button>

Click me!

</button>

);

}

}

**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 talk={this.talk} />;

}

}

ReactDOM.render(

<Talker />,

document.getElementById('app')

);

 By default, functions return undefined. As a result, when passing a function as a prop, there must not be any parentheses following its name.

The correct way to pass the function to the component would be,

# 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 Buttonrenders 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.

**Button.js**

import React from 'react';

export class Button extends React.Component {

render() {

return (

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

Click me!

</button>

);

}}

**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 talk={this.talk}/>;

}

}

ReactDOM.render(

<Talker />,

document.getElementById('app')

);

Yes! You can pass in multiple event handlers as props to the child component, then to choose a specific one to attach to it you can utilize something like a ternary statement.

For example, say that you have a component whose button should run different code based on its state. We can pass the functions to the child component, then from the child component, use a ternary statement that will attach one of the two functions based on its state.

The parent component might look like the following, which passes two event handlers to its child component.

**class** **Parent** **extends** **React**.**Component** {

handleClickOne() {

*// Code*

}

handleClickTwo() {

*// Code*

}

render() {

**return** <**Child** onClickOne={this.handleClickOne}

onClickTwo={this.handleClickTwo} />;

}

}

Then, the child component will receive both event handlers, and attach one of them to itself, based on its state.

**class** **Child** **extends** **React**.**Component** {

render() {

**return** (

<**button** onClick={this.state.value == someValue ?

this.props.onClickOne : this.props.onClickTwo}>

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**, 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.

class MyClass extends React.Component {

handleHover() {

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

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

}

}

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 listen for a "hover" event.');

}

render() {

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

}

}

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.

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 />

**Example:**

**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>

</List>

<List type='Living Cat Musician'>

<li>Nora the Piano 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.

**Example:**

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')

);

* **Passed information from parent components (App) to child components (BusinessList, Business**)

**THIS.STATE**

# State

# Dictionary definitions tell us that the word 'dynamic' is characterised by continuous change, activity or progress. This is a fairly accurate description in terms of a 'dynamic information source'.

Dynamic data sources are widely available on smart phones in the form of an 'app'. For example you can get an app to monitor stock prices. It is updated minute by minute.

Another example is real-time traffic updates linked to your Sat-Nav. You subscribe to a traffic monitoring service which then sends the latest information into your navigation system.

# However, you need to be clear that not all web sites are dynamic information sources, some could be classed as static. Although we are constantly adding new worksheets and pages to this web site, the theory mini-web that you are using right now probably won't be updated again for at least another year. That would class it as a static source of information.

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. 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!

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' }.

what values can you store in state?

### Answer

A React component’s state is a plain JavaScript object, so it can store any valid value that is storable within a JavaScript object. This includes booleans, strings, numbers, and even other objects which you can nest within the state.

# Access a Component's state

To read a component’s state, use the expression this.state.name-of-property:

Example:

import React from 'react';

import ReactDOM from 'react-dom';

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'));

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.

# 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.

import React from 'react';

class Example extends React.Component {

constructor(props) {

  super(props);

this.state = {

**mood: 'great',**

**hungry: false**

**};**

**}**

**render() {**

**return <div>{this.setState({ hungry: true });}</div>;**

**}**

**}**

**<Example />**

# 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);

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.

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

Example1:

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'));

Example2:

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'));

can this.setState change multiple properties at once?

### Answer

Yes, you can use this.setState to change single, multiple, or even all properties of state at a time. When setting state for multiple properties, only the properties passed in will be updated.

For example, the following shows how the properties key1 and key3 of a state could be updated.

*/\* Given this state \*/*

**this**.state = {

key1: value1,

key2: value2,

key3: value3

}

*/\* We could use code like the following to update specific properties \*/*

**this**.setState({ key1: newValue1, key3: newValue3 });

# 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.

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**

# Random Color Picker

Random.js

import React from 'react';

import ReactDOM from 'react-dom';

import { Button } from './Button';

class Random extends React.Component {

constructor(props) {

super(props);

this.state = {

color: [92, 132, 153]

}

this.handleClick = this.handleClick.bind(this);

}

componentDidMount() {

this.applyColor();

}

componentDidUpdate(prevProps, prevState) {

this.applyColor();

}

formatColor(ary) {

return 'rgb(' + ary.join(', ') + ')';

}

isLight() {

const rgb = this.state.color;

return rgb.reduce((a,b) => a+b) < 127 \* 3;

}

applyColor() {

const color = this.formatColor(this.state.color);

document.body.style.background = color;

}

chooseColor() {

const random = [];

for (let i = 0; i < 3; i++) {

random.push(Math.floor(Math.random()\*256));

}

return random;

}

handleClick() {

this.setState({

color: this.chooseColor()

});

}

render() {

return (

<div>

<h1 className={this.isLight() ? 'white' : 'black'}>

    Your color is {this.formatColor(this.state.color)}.

</h1>

             <Button

light={this.isLight()}

onClick={this.handleClick} />

        </div>

);

}

}

ReactDOM.render(

<Random />,

document.getElementById('app')

);

Button.js

import React from 'react';

export class Button extends React.Component {

    render() {

        return (

            <button

                className={ this.props.light ? 'light-button' : 'dark-button' }

      onClick={this.props.onClick}>

                Refresh

            </button>

        );

    }

}

1. First, let’s store this random color as state.

Give Random a constructor() method. Give constructor() one parameter, named props.

Inside the body of constructor(), write the line super(props);.

Still inside the body of constructor(), on a new line, set this.state equal to this object:

{ color: [x, y, z] }

Instead of x, y, and z, use three numbers between 0 and 255.

1. In Random, find the method named formatColor. This method transforms an rgb array into something a bit more readable.

Inside of the <h1></h1>, instead of simply using this.state.color, call the formatColorfunction and pass in this.state.color as an argument.

Your color is {this.formatColor(this.state.color)}.

1. Select **Button.js**. Add the word export so that you are exporting the Button component class.
2. On this new line, use import the Buttoncomponent class.**Button.js** and **Random.js** share the same parent directory.

import { Button } from './Button';

1. Now you’re ready to render a <Button />instance!

Inside of Random‘s render method, after the <h1></h1>, add a <Button />.

Give your <Button /> this attribute:

light={this.isLight()}

Hint

</h1>

<Button light={this.isLight()} />

</div>

1. You can see your <Button /> instance in the browser. However, clicking it doesn’t do anything!

You need to define an *event handler* that updates this.state.color to a new random color.

Give Random a new method named handleClick.

Inside of .handleClick()‘s body, call this.setState(). As an argument, pass this.setState() an object with the following property:

color: this.chooseColor()

Hint

handleClick() {

this.setState({

color: this.chooseColor()

});

}

1. That means that you need to bind your new method.

method. On this new line, bind handleClick().

constructor(props) {

super(props);

this.state = {

color: [10, 20, 30]

};

this.handleClick = this.handleClick.bind(this);

1. Great! this.handleClick() will update this.state.color to a new, random color.

Now that you’ve defined an *event handler*, you can pass it to another component as a prop. This is a pattern that you’ll see much more of in the next course.

In Random‘s render method, give <Button />an attribute with a *name* of onClick. Set onClick‘s *value* equal to the handleClickmethod.

Hint

</h1>

<Button

light={this.isLight()}

onClick={this.handleClick} />

</div>

1. Select **Button.js**. In the render function, give the <button></button> an onClick attribute. Set onClick‘s *value* equal to the passed-in prop.

Quiz--------------------------------------------------------------------------------------------------

Fill in the blank, so that clicking on a button will capitalize the button’s inner text

import React from 'react';

class Button extends React.Component {

constructor(props) {

super(props);

this.state = { message: 'Hello world' };

this.capitalize = this.capitalize.bind(this);

}

capitalize() {

this.setState({

message: this.state.message.toUpperCase()

});

}

render() {

return (

<button \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_>

{this.state.message}

</button>

);

}

}

onClick={this.capitalize}

**Why can’t you call this.setState from within the render method**

Ans: this.setState automatically calls render, so it would create an infinite loop

**this.props evaluates to what data type**

Ans : Object.

**Which is the correct way to pass a prop**

<Headline text="Hello world" />

**STATELESS COMPONENTS FROM STATEFUL COMPONENTS**

# Stateless Components Inherit From Stateful Components

Now let’s learn about a **simple version of a programming pattern** and by the end, we’ll have a programming pattern in its **full complexity.**

Our **programming pattern** uses **two React components**: a **stateful** component, and a **stateless** component.

“Stateful” describes any component that has a stateproperty;

“stateless” describes any component that does not.

In our pattern, a stateful component passes its statedown to a stateless component.

# Build a Stateful Component Class

Let’s make a stateful component pass its state to a stateless component.

To make that happen, you need two component classes: a stateful class, and a stateless class.

**Parent.js**

stateFul Component class

import React from 'react';

import ReactDOM from 'react-dom';

class Parent extends React.Component {

constructor(props) {

super(props);

this.state = { name: 'Frarthur' };

}

render() {

return <div></div>;

}

}

# Build a Stateless Component Class

Great! You just made a stateful component class named Parent.

Now, let’s make our stateless component class.

Child.js

StateLess Component Class

import React from 'react';

export class Child extends React.Component {

render() {

return <h1>Hey, my name is {this.props.name}</h1>;

}

}

# Pass a Component's State

A <Parent /> is supposed to pass its state to a <Child />.

Before a <Parent /> can pass anything to a <Child />, you need to import Child into

**Parent.js**.

import React from 'react';

import ReactDOM from 'react-dom';

import { Child } from './Child'; // add on

class Parent extends React.Component {

constructor(props) {

super(props);

this.state = { name: 'Frarthur' };

}

render() {

return <Child name={this.state.name} />;// add on

}}

ReactDOM.render(<Parent />, document.getElementById('app')); // add on

**Can we pass the entire state object to the child component?**

### Answer

Yes, you can do this by using the spread operator in JavaScript, which is done like the following, by adding three dots in front of the object {...this.state}.

*// Given this state for a component.*

**this**.state = {

key1: 'val1',

key2: 'val2'

}

*// We can pass the entire state object to a child component as follows.*

*// Keep in mind that we do not set any attribute to it.*

<Child {...**this**.state} />

*// This is equivalent to doing the following.*

<Child key1={**this**.state.key1} key2={**this**.state.key2} />

# Don't Update props

A **component** can **change its state** by calling this.setState(). You may have been wondering: how does a **component change its props**?

The answer: it doesn’t!

A component **should never update** this.props.

This is potentially confusing. props and state store dynamic information. Dynamic information can change, by definition. If a component can’t change its props, then what are props for?

**A React component should use props to store information that can be changed, but can only be changed by a different component.**

**A React component should use state to store information that the component itself can change.**

**CHILD COMPONENTS UPDATE THEIR PARENTS' STATE**

# Child Components Update Their Parents' state

In the last you **passed information** from a **stateful, parent component** to a **stateless, child component**.

In this you’ll be expanding on that pattern. **The stateless, child component will update the state of the parent component.**

Here’s how that works:

1. **The parent component** class defines a method that calls this.setState().

For an example, look in **Step1.js** at the .handleClick()method.

import React from 'react';

import ReactDOM from 'react-dom';

import { ChildClass } from './ChildClass';

class ParentClass extends React.Component {

constructor(props) {

super(props);

this.state = { totalClicks: 0 };

}

handleClick() {

const total = this.state.totalClicks;

// calling handleClick will

// result in a state change:

this.setState(

{ totalClicks: total + 1 }

);

}

}

1. The parent component binds the newly-defined method to the current instance of the component in its constructor. This ensures that when we pass the method to the child component, it will still update the parent component.

For an example, look in **Step2.js** at the end of the constructor() method.

import React from 'react';

import ReactDOM from 'react-dom';

import { ChildClass } from './ChildClass';

class ParentClass extends React.Component {

constructor(props) {

super(props);

this.state = { totalClicks: 0 };

this.handleClick = this.handleClick.bind(this); // do it

}

handleClick() {

const total = this.state.totalClicks;

// calling handleClick will

// result in a state change:

this.setState(

{ totalClicks: total + 1 }

);

}

// The stateful component class passes down

// handleClick to a stateless component class:

render() {

return (

<ChildClass onClick={this.handleClick} /> // line 28 // do it

);

}

}

1. Once the parent has defined a method that updates its state and bound to it, the parent then passes that method down to a child.

Look in **Step2.js**, at the prop on line 28.

1. The child receives the passed-down function, and uses it as an event handler.

Look in **Step3.js**. When a user clicks on the <button></button>, a click event will fire. This will make the passed-down function get called, which will update the parent’s state.

import React from 'react';

import ReactDOM from 'react-dom';

export class ChildClass extends React.Component {

render() {

return (

// The stateless component class uses

// the passed-down handleClick function,

// accessed here as this.props.onClick,

// as an event handler:

<button onClick={this.props.onClick}> // do it

Click Me!

</button>

);

}

}

# Define an Event Handler

To make a child component update its parent’s state, the first step is something that you’ve seen before: you must define a state-changing method on the parent

In **Parent.js**, define a new function that can change this.state.name:

changeName(newName) {

this.setState({

name: newName

});

}

render() { ...

# Pass the Event Handler

In the last, you defined a function in Parentthat can change Parent‘s state.

Parent must pass this function down to Child, so that Child can use it in an event listener on the dropdown menu.

class Parent extends React.Component {

constructor(props) {

super(props);

this.state = { name: 'Frarthur' };

this.changeName = this.changeName.bind(this); // do it

}

changeName(newName) {

this.setState({

name: newName

});

}

render() {

return <Child name={this.state.name} onChange={this.changeName} /> //do it

}

}

# Receive the Event Handler

You just passed a function down to Child that can change Parent‘s state!

**Child.js**

import React from 'react';

export class Child extends React.Component {

constructor(props) {

super(props);

this.handleChange = this.handleChange.bind(this);

}

handleChange(e) {

const name = e.target.value;

this.props.onChange(name); // **offered a way to change that name**

}

render() {

return (

<div>

<h1>

Hey my name is {this.props.name}! // display the name

</h1>

<select id="great-names" onChange={this.handleChange}>

<option value="Frarthur">

Frarthur

</option>

<option value="Gromulus">

Gromulus

</option>

<option value="Thinkpiece">

Thinkpiece

</option>

</select>

</div>

);

}

}

**Parent.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { Child } from './Child';

class Parent extends React.Component {

constructor(props) {

super(props);

this.state = { name: 'Frarthur' };

this.changeName = this.changeName.bind(this);

}

changeName(newName) {

this.setState({

name: newName

});

}

render() {

return <Child name={this.state.name} onChange={this.changeName} />

}

}

ReactDOM.render(

  <Parent />,

  document.getElementById('app')

);

**CHILD COMPONENTS UPDATE THEIR SIBLINGS' PROPS**

# Child Components Update Sibling Components

Patterns within patterns within patterns!

In lesson 1, you learned your first React programming pattern: a stateful, parent component passes down a prop to a stateless, child component.

In lesson 2, you learned that lesson 1’s pattern is actually part of a larger pattern: a stateful, parent component passes down an event handler to a stateless, child component. The child component then uses that event handler to update its parent’s state.

In this lesson, we will **expand** the **pattern** one last time. A **child component** **updates** its **parent’s** state, and the **parent passes that state** to **a sibling component**.

An understanding of this final pattern will be very helpful in the wild.

# One Sibling to Display, Another to Change

One of the very first things that you learned about components is that **they should only have one job.**

In the last lesson, Child had two jobs:

1 - **Child displayed a name**.

2 **- Child offered a way to change that name**.

You should make like Solomon and divide Child in two: one component for displaying the name, and a different component for allowing a user to change the name.

In the code editor, select **Child.js**. Notice that these lines have vanished:

<h1>

Hey, my name is {this.props.name}!

</h1>

The new version of Child renders a dropdown menu for changing the name, and that’s it.

Select **Sibling.js** in the code editor.

Read through the render function’s return statement.

Here, the name is displayed! Or at least it will be displayed, once you’ve done a little editing.

That brings us to the essential new concept for this lesson: you will have one stateless component displayinformation, and a different stateless component offer the ability to change that information.

**Pass the Right props to the Right Siblings**

Look at **Parent.js** in the code editor.

Three things have changed in this file since the last Lesson:

1. Sibling has been required on line 4.
2. A <Sibling /> instance has been added to the render function on line 27.
3. <Sibling /> and <Child /> have been wrapped in a <div></div>, since JSX expressions must have only one outer element.

# Display Information in a Sibling Component

You’re on the last step!

You’ve passed the name down to <Sibling /> as a prop. Now <Sibling /> has to display that prop.

**Parent.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { Child } from './Child';

import { Sibling } from './Sibling';

class Parent extends React.Component {

constructor(props) {

super(props);

this.state = { name: 'Frarthur' };

this.changeName = this.changeName.bind(this);

}

changeName(newName) {

this.setState({

name: newName

});

}

render() {

return (

<div>

<Child

//name={this.state.name}

onChange={this.changeName} />

<Sibling name={this.state.name} />

</div>

);

}

}

/\*

1.Siblings job is to display the selected name.

That name is stored in Parents state. Parent will have to pass the name to Sibling, so that Sibling can display it.

In Parents render function, pass the name to <Sibling />:

\*/

ReactDOM.render(

<Parent />,

document.getElementById('app')

);

**Child.js**

import React from 'react';

export class Child extends React.Component {

constructor(props) {

super(props);

this.handleChange = this.handleChange.bind(this);

}

handleChange(e) {

const name = e.target.value;

this.props.onChange(name);

}

render() {

return (

<div>

<select

id="great-names"

onChange={this.handleChange}>

<option value="Frarthur">Frarthur</option>

<option value="Gromulus">Gromulus</option>

<option value="Thinkpiece">Thinkpiece</option>

</select>

</div>

);

}

}

**Sibling.js**

import React from 'react';

export class Sibling extends React.Component {

render() {

let name = this.props.name;

return (

<div>

<h1>Hey, my name is {name}!</h1>

<h2>Don't you think {name} is the prettiest name ever?</h2>

<h2>Sure am glad that my parents picked {name}!</h2>

</div>

);

}

}

# Stateless Components Inherit From Stateful Components Recap

You just executed your first complete React programming pattern!

Let’s review. Follow each step in the code editor:

* A stateful component class defines a function that calls this.setState. (**Parent.js**, lines 15-19)
* The stateful component passes that function down to a stateless component. (**Parent.js**, line 24)
* That stateless component class defines a function that calls the passed-down function, and that can take an event object as an argument. (**Child.js**, lines 10-13)
* The stateless component class uses this new function as an event handler. (**Child.js**, line 20)
* When an event is detected, the parent’s state updates. (A user selects a new dropdown menu item)
* The stateful component class passes down its state, distinct from the ability to change its state, to a different stateless component. (**Parent.js**, line 25)
* That stateless component class receives the state and displays it. (**Sibling.js**, lines 5-10)
* An instance of the stateful component class is rendered. One stateless child component displays the state, and a different stateless child component displays a way to change the (**Parent.js**, lines 23-26)

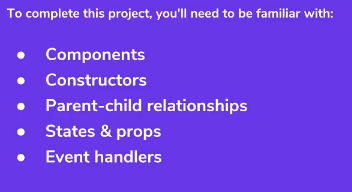
This pattern occurs in React all the time! The more that you see it, the more that its elegance will become clear.

Being introduced to this pattern is your first step towards understanding how React apps fit together! You’ll get more practice using it throughout this course, as well as in the course after this one.

**STATELESS COMPONENTS FROM STATEFUL COMPONENTS**

# Video Player

You just learned your first programming pattern. Let’s put it to use!



1. Take a look at the App component class. This class has one property stored as state: a src containing the address of a video file.App‘s job is to pass this src down to a stateless component, and to pass *the ability to change the* src down to a different stateless component.

Passing src is the easier part, so let’s do that first.

Inside of App‘s *render* function, give <Video /> an *attribute*. Make this attribute’s *name*src, and **the attribute’s *value* equal** to the src property stored in this.state.

In **App.js**:

<Video src={this.state.src} />

1. Let’s make <Video /> play its passed-in video file!

Select **Video.js**. In Video‘s render function, give <video /> a src attribute. Make srcequal to the passed-in video file.

In **Video.js**:

<video controls autostart autoplay src={this.props.src} />

1. Alright, the video player works! Now let’s make the *menu* work as well.

Now App needs to pass the ability to *change* the src down to <Menu />.

If you want to pass the ability to ***change* a piece of state**, then first you need to define a function that calls this.setState.

chooseVideo is going to get passed a string: either 'fast', 'slow', 'cute', or 'eek'. It will use this string to choose a new src, which it will use to update this.state.src.

In **App.js**:

chooseVideo(newVideo) {

this.setState({

src: VIDEOS[newVideo]

});

},

1. If you pass chooseVideo to <Menu />, then you will give <Menu /> the ability to *update* <App />‘s state.

In App‘s render function , give Menu a chooseVideo attribute. Set chooseVideo‘s *value*equal to the chooseVideo function.

In **App.js**:

<Menu chooseVideo={this.chooseVideo} />

1. Currently, if you pass .chooseVideo() to Menuthe value of this will be incorrect when called. In the constructor of App, bind.chooseVideo() to the current value of thisand store it in this.chooseVideo.

In **App.js**:

constructor(props) {

super(props);

this.chooseVideo = this.chooseVideo.bind(this);

}

1. Alright, now you just have to attach this passed-in function to an event listener!

In **Menu.js**:

<form onClick={this.props.chooseVideo}>

1. Try selecting a video in the browser.

It doesn’t work! Do you know why not?

chooseVideo **expects a *string*** as an **argument**. **But event handlers** are automatically **passed *event objects****,* not strings.

You need to *wrap* chooseVideo in a new **function that can take an event object as an argument.**

In **Menu.js**:

handleClick(e) {

var text = e.target.value;

this.props.chooseVideo(text);

},

render() {

1. **Currently, the value of this will be incorrect when you call .handleClick().**

Create a constructor for Menu, and in its body, call super(props). Then, bind.handleClick() to the current value of thisand store it in this.handleClick.

In **Menu.js**:

constructor(props) {

super(props);

this.handleClick = this.handleClick.bind(this);

}

1. Only one more step! You need to use your new wrapper function as an *event handler*.

In Menu‘s render function’s return statement, replace {this.props.chooseVideo} with {this.handleClick}.

In **Menu.js**:

<form onClick={this.handleClick}>

**App.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { Video } from './Video';

import { Menu } from './Menu';

const VIDEOS = {

fast: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_video-fast.mp4',

slow: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_video-slow.mp4',

cute: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_video-cute.mp4',

eek: 'https://s3.amazonaws.com/codecademy-content/courses/React/react\_video-eek.mp4'

};

class App extends React.Component {

constructor(props) {

super(props);

this.state = { src: VIDEOS.fast };

this.chooseVideo = this.chooseVideo.bind(this);

}

     chooseVideo(newVideo) {

this.setState({

src: VIDEOS[newVideo]

});

}

render() {

return (

<div>

<h1>Video Player</h1>

<Menu chooseVideo={this.chooseVideo} />

<Video src={this.state.src} />

</div>

);

}

}

ReactDOM.render(

<App />,

document.getElementById('app')

);

**Video.js**

import React from 'react';

export class Video extends React.Component {

render() {

return (

<div>

<video controls autostart autoplay src={this.props.src}/>

</div>

);

}

}

/\*

step 2:

<video controls autostart autoplay src={this.props.src} />

\*/

**Menu.js**

import React from 'react';

export class Menu extends React.Component {

constructor(props) {

super(props);

this.handleClick = this.handleClick.bind(this);

}

handleClick(e) {

var text = e.target.value;

this.props.chooseVideo(text);

}

render() {

return (

<form onClick={this.handleClick}>

<input type="radio" name="src" value="fast" /> fast

<input type="radio" name="src" value="slow" /> slow

<input type="radio" name="src" value="cute" /> cute

<input type="radio" name="src" value="eek" /> eek

</form>

);

}

}

**Quiz**

**In the programming pattern that you just learned, what are the roles of the three component classes?**

**Ans:**

A **stateful** component class **stores information** as state.

A stateless component class displays that state.  
A different stateless component class displays a way to change that state.

**What will this equal in updateFunction when a user clicks on the ChildExample rendered below**

import React from 'react';

import ReactDOM from 'react-dom';

class ParentExample extends React.Component {

constructor(props) {

super(props);

this.updateFunction = this.updateFunction.bind(this);

}

updateFunction(updatedValue) {

this.setState({value: updatedValue});

},

render() {

return <ChildExample onClick={this.updateFunction} />;

}

});

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

**Ans: The ParentExample that rendered the clicked ChildExample**

**If a user clicks on the button, what ARGUMENT will be passed to logger**

import React from 'react';

import ReactDOM from 'react-dom';

class Example extends React.Component {

logger(param) {

console.log(param);

},

render() {

return <button onClick={this.logger}></button>;

}

});

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

**Ans:** An event object, because logger is being used as an event handler

**What’s the point of stateless components inheriting from stateful components**

Ans: It’s a common pattern used by React programmers. Recognizing it can help you make sense of React programs that you encounter, and can also help you build better React programs yourself

**HTTP Requests**

## Background:

The **internet** is made up of a **bunch of resources** hosted on different **servers**. The term “**resource**” corresponds to any entity on the web, including **HTML files, stylesheets, images, videos, and scripts.** To **access** **content** on the **internet**, your **browser** must **ask** these **servers** for the **resources it wants**, and then **display** these **resources** to you. This **protocol** of **requests** and **responses** enables you view this page in your browser.

## What is HTTP?

HTTP stands for Hypertext Transfer Protocol and is used to **structure requests and responses** over the **internet**. HTTP **requires** **data** to be **transferred** from **one point to another** over the network.

The **transfer** of **resources** happens **using** **TCP** (Transmission Control Protocol). In viewing this webpage, TCP **manages** the **channels** between **your browser and the server** (in this case, LearnCodeWeb.com). TCP is used to manage many types of internet connections in which one computer or device wants to send something to another. **HTTP is the command language that the devices on both sides of the connection** must follow in order to communicate.

## HTTP & TCP: How it Works

When you **type an address** such as [www.LearnWebCode.com](http://www.LearnWebCode.com) into your **browser**, you are **commanding** it to **open a TCP channel** to the **server** that **responds** to that **URL** (or Uniform Resource Locator) A URL is like your home address or phone number because it describes how to reach you.

**In this situation, your computer, which is making the request, is called the client**. The URL you are requesting is the address that belongs to the server.

Once the TCP connection is established, the client sends a HTTP GET request to the server to retrieve the webpage it should display.

**After** the **server** has **sent** the **response**, it **closes** the **TCP** connection. If you open the website in your browser again, or if your browser automatically requests something from the server, a new connection is opened which follows the same process described above.

GET requests are one kind of HTTP method a client can call. You can learn more about the other common ones (POST, PUT and DELETE)

<https://www.codecademy.com/articles/what-is-rest>

Let’s explore an example of how GET requests (the most common type of request) are used to help your computer (the client) access resources on the web.

Suppose you want to check out the latest course offerings from [http://codecademy.com](http://codecademy.com/). After you type the **URL** into your browser, your **browser will extract the http** part and recognize that it is the name of the **network protocol** to use. Then, it takes the domain name from the **URL**, in this case “codecademy.com”, and asks the internet Domain Name Server to return an Internet Protocol (IP) address.

Now the client knows the destination’s IP address. It then opens a connection to the server at that address, using the http protocol as specified. It will initiate a GET request to the server which contains the IP address of the host and optionally a data payload. The GET request contains the following text:

GET / HTTP/1.1

Host: www.codecademy.com

This identifies the type of request, the path on [www.codecademy.com](https://codecademy.com/) (in this case, “/“) and the protocol “HTTP/1.1.” HTTP/1.1 is a revision of the first HTTP, which is now called HTTP/1.0. In HTTP/1.0, every resource request requires a separate connection to the server. HTTP/1.1 uses one connection more than once, so that additional content (like images or stylesheets) is retrieved even after the page has been retrieved. As a result, requests using HTTP/1.1 have less delay than those using HTTP/1.0.

The second line of the request contains the address of the server which is "www.codecademy.com". There may be additional lines as well depending on what data your browser chooses to send.

If the server is able to locate the path requested, the server might respond with the header:

HTTP/1.1 200 OK

Content-Type: text/html

This header is followed by the content requested, which in this case is the information needed to render [www.codecademy.com](https://codecademy.com/).ZXC

The first line of the header, HTTP/1.1 200 OK, is confirmation that the server understands the protocol that the client wants to communicate with (HTTP/1.1), and an [HTTP status code](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) <https://en.wikipedia.org/wiki/List_of_HTTP_status_codes> signifying that the resource was found on the server. The third line, Content-Type: text/html, shows the type of content that it will be sending to the client.

If the server is not able to locate the path requested by the client, it will respond with the header:

HTTP/1.1 404 NOT FOUND

In this case, the server identifies that it understands the HTTP protocol, but the 404 NOT FOUND status code signifies that the specific piece of content requested was not found. This might happen if the content was moved or if you typed in the URL path incorrectly or if the page was removed. You can read more about the 404 status code, commonly called a 404 error, <https://www.codecademy.com/articles/http-errors-404>

What is happening here is that when a user enters transmission protocol and a domain in the address bar. The browser uses that protocol and picks that domain and takes it to DNS, which provides browser the IP address of that domain. As soon the browser gets the IP address of the domain it establishes connection using TCP with that IP address and starts sending  HTTP GET   requests to it using HTTP. Now the Server receive those requests and establishes the TCP with client to provide response for those requests.

## An Analogy:

It can be tricky to understand how HTTP functions because it’s difficult to examine what your browser is actually doing. (And perhaps also because we explained it using acronyms that may be new to you.) Let’s review what we learned by using an analogy that could be more familiar to you.

Imagine the internet is a town. You are a client and your address determines where you can be reached. Businesses in town, such as Codecademy.com, serve requests that are sent to them. The other houses are filled with other clients like you that are making requests and expecting responses from these businesses in town. This town also has a crazy fast mail service, an army of mail delivery staff that can travel on trains that move at the speed of light.

Suppose you want to read the morning newspaper. In order to retrieve it, you write down what you need in a language called HTTP and ask your local mail delivery staff agent to retrieve it from a specific business. The mail delivery person agrees and builds a railroad track (connection) between your house and the business nearly instantly, and rides the train car labeled “TCP” to the address of the business you provided.

Upon arriving at the business, she asks the first of several free employees ready to fulfill the request. The employee searches for the page of the newspaper that you requested but cannot find it and communicates that back to the mail delivery person.

The mail delivery person returns on the light speed train, ripping up the tracks on the way back, and tells you that there was a problem “404 Not Found.” After you check the spelling of what you had written, you realize that you misspelled the newspaper title. You correct it and provide the corrected title to the mail delivery person.

This time the mail delivery person is able to retrieve it from the business. You can now read your newspaper in peace until you decide you want to read the next page, at which point, you would make another request and give it to the mail delivery person.

## What is HTTPS?

Since your HTTP request can be read by anyone at certain network junctures, it might not be a good idea to deliver information such as your credit card or password using this protocol. Fortunately, many servers support HTTPS, short for HTTP Secure, which allows you to encrypt data that you send and receive. You can read more about HTTPS on [Wikipedia](https://en.wikipedia.org/wiki/HTTPS#Difference_from_HTTP).

HTTPS is important to use when passing sensitive or personal information to and from websites. However, it is up to the businesses maintaining the servers to set it up. In order to support HTTPS, the business must apply for a certificate from a [Certificate Authority](https://en.wikipedia.org/wiki/Certificate_authority).

**DNS(CONTAIN ALL IP ADDRESS) - http(set of rules of client)**

**website/server browser/client**

**(TCP rule for server)**

**{TO RESPOND THE REQUEST}**

# Authentication

## Introduction

Authentication is the process used by applications to determine and confirm identities of users. It ensures that the correct content is shown to users. More importantly, it ensures that incorrect content is secured and unavailable to unauthorized users.

In this article, we’ll discuss a few of the common design patterns for these interactions. You’ll need to have some basic understanding of HTTP requests, since these methods all use HTTP requests to exchange information.

## Password Authentication

The most common implementation of authentication requires a user to input their username or email and a password. The application’s server then checks the supplied credentials to determine if the user exists and if the supplied password is correct. If the credentials are correct, the user is logged in and able to use the application as thatthe user.

Typically, upon a successful login, the application will respond with an authentication token (or auth token) for the client to use for additional HTTP requests. This token is then stored on the user’s computer, preventing the need for users to continuously log in.

This token generally expires after a certain amount of time, ensuring the correct user is using the application over time as well.

## API Keys

While it is common to think of authentication as the interaction between a human user and an application, sometimes the user is another application.

Many apps expose interfaces to their information in the form of an API (application program interface). For example, the [Spotify API](https://developer.spotify.com/web-api/) provides endpoints for almost all of its functionality. This allows applications to fetch data from the Spotify music catalog and manage user’s playlists and saved music.

Since these external requests could overwhelm a service and also access user information, they need to be secured using authentication.

The most basic pattern for API access from another application is using an API key.

**Public APIs usually provide a developer portal where you can register your application and generate a corresponding API key.** This key is then unique to your application. When your application makes a request, this key is sent along with it. The API can then verify that your application is allowed access and provide the correct response based on the permission level of your application.

The API can track what type and frequency of requests each application is making. This data can be used to [throttle requests](https://en.wikipedia.org/wiki/Throttling_process_(computing)) from a specific application to a pre-defined level of service. This prevents applications from spamming an endpoint or abusing user data, since the API can easily block that application’s API key and prevent further malicious use of the API by that application.

## OAuth

For many applications, a generic developer-level API key is not sufficient. As mentioned earlier, APIs sometimes have the ability to provide access to user-level data. However, most services only provide this private data if the user enables it.

For example, Facebook doesn’t want Tinder to access all of their users’ data, just the users who have opted in to allowing the sharing of data to better help them find a match in their area.

A basic approach to this problem might be to have the user provide their login credentials to the intermediary application, but this is not very secure and would give full access to the requesting application, when the requesting application might only need a very limited set of privileges to function.

**OAuth** defines a more elegant approach to this problem. It was developed in November 2006 by lead Twitter developer Blaine Cook and version 1.0 was published in April 2010.

OAuth is an **open standard** and is commonly used to grant permission for applications to access user information without forcing users to give away their passwords.

An open standard is a publicly available definition of how some functionality should work. However, the standard does not actually build out that functionality.

As a result, each API is required to implement their own version of OAuth and therefore may have a slightly different implementation or flow. However, they’re all based around the same OAuth specification.

This can make using a new OAuth API a little more frustrating. However, with time you will begin noticing the similarities between API authentication flows and be able to use them in your applications with increasing ease. Below is a summary of the standard OAuth flow.

## Generic OAuth Flow

Many applications implementing OAuth will first ask the user to select which service they would like to use for credentials:

authenticationLogin

After selecting the service, the user will be redirected to the service to login. This login confirms the user’s identity and typically provides the user with a list of permissions the originating application is attempting to gain on the user’s account.

If the user confirms they want to allow this access, they will be redirected back to the original site, along with an access token. This access token is then saved by the originating application.

Like a developer API key, this access token will be included on requests by the application to prove that the user has granted access and enable access to the appropriate content for that user. When a user returns to the application, the token will be retrieved and they will not have to re-authenticate.

## OAuth 2

Since OAuth evolved out of Twitter, there were important use cases not originally considered as part of the specification. Eventually, this led to the creation of a new version of the specification, called OAuth 2.

Among other improvements, OAuth 2 allows for different authentication flows depending on the specific application requesting access and the level of access being requested.

OAuth 2 is still an open standard, so each API will have its own flow based on its particular implementation. Below, we’ll discuss a few of the common OAuth 2 flows and how they are used.

## Client Credentials Grant

Sometimes an application will not need access to user information but may implement the added security and consistency of the OAuth 2 specification. This type of grant is used to access application-level data (similar to the developer API key above) and the end user does not participate in this flow.

Instead of an API key, a client ID and a client secret (strings provided to the application when it was authorized to use the API) are exchanged for an access token (and sometimes a refresh token). We will discuss refresh tokens in more depth later.

This flow is similar to our first example, where an email and password were exchanged for an authentication token.

**It is essential to ensure the client secret does not become public information, just like a password. As a result, developers should be careful not to accidentally commit this information to a public git repository. Additionally, to ensure integrity of the secret key, it should not be exposed on the client-side and all requests containing it should be sent server-side.**

Similar to the previously-mentioned keys, the returned access token is included on requests to identify the client making the requests and is subject to API restrictions.

This access token is often short-lived, expiring frequently. Upon expiration, a new access token can be obtained by re-sending the client credentials or, preferably, a refresh token.

Refresh tokens are an important feature of the OAuth 2 updates, encouraging access tokens to expire often and, as a result, be continuously changed (in the original OAuth specification, access tokens could last for time periods in the range of years). When a refresh token is used to generate a new access token, it typically expires any previous access tokens.

## Authorization Code Grant

This flow is one of the most common implementations of OAuth and will look familiar if you’ve ever signed into a web application with Google or Facebook. It is similar to the OAuth flow described earlier with an added step linking the requesting application to the authentication.

A user is redirected to the authenticating site, verifies the application requesting access and permissions, and is redirected back to the referring site with an authorization code.

The requesting application then takes this code and submits it to the authenticating API, along with the application’s client ID and client secret to receive an access token and a refresh token. This access token and refresh token are then used in the same manner as the previous flow.

To avoid exposing the client ID and secret, this step of the flow should be done on the server side of the requesting application.

Since tokens are tied both to users and requesting applications, the API has a great deal of control over limiting access based on user behavior, application behavior, or both.

## Implicit Grant

The previous two methods cause the client secret key to be exposed, so they need to be handled server-side. Some applications may need to access an OAuth API but don’t have the necessary server-side capabilities to keep this information secure.

The Implicit Grant OAuth flow was designed for this very use case. This flow prompts the user through similar authorization steps as the Authorization Code flow, but does not involve the exchange of the client secret.

The result of this interaction is an access token, and typically no refresh token. The access token is then used by application to make additional requests to the service, but is not sent to the server side of the requesting application.

This flow allows applications to use OAuth APIs without fear of potentially exposing long-term access to a user or application’s information.

## Conclusion

OAuth provides powerful access to a diverse set of sites and information. By using it correctly, you can reduce sign-up friction and enrich user experience in your applications.

# STYLE

# Advanced React Techniques

In this unit, you will learn a variety of useful techniques that React programmers are expected to know.

You’ll learn how to **make a stateless functional component**, how to make a **propType,** how to **write a form, and** how to **use styles**.

You’ll also be introduced to your second programming pattern: dividing components **into presentational components and container components.**

# Inline Styles

There are many different ways to use styles in React. This lesson is focused on one of them: inline styles.

An inline style is a style that’s written as an attribute, like this:

<h1 style={{ color: 'red' }}>Hello world</h1>

Notice the double curly braces. What are those for?

The **outer** curly braces **inject JavaScript into JSX**. They say, “everything between us should be read as JavaScript, not JSX.”

The **inner** curly braces **create a JavaScript object** literal. They make this a valid JavaScript object:

{ color: 'red' }

If you inject an object literal into JSX, and your entire injection is only that object literal, then you will end up with double curly braces. There’s nothing unusual about how they work, but they look funny and can be confusing.

Example:

import React from 'react';

import ReactDOM from 'react-dom';

const styleMe = <h1 style={{background: 'lightblue', color: 'darkred'}}>Please style me! I am so bland!</h1>;

ReactDOM.render(

    styleMe,

    document.getElementById('app')

);

# Make A Style Object Variable

That’s all that you need to apply basic styles in React! Simple and straightforward.

One problem with this approach is that it becomes obnoxious if you want to use more than just a few styles. An alternative that’s often nicer is to store a style object in a variable, and then inject that variable into JSX.

Look in the code editor for an example. The style object is defined on lines 3-6, and then injected on line 11.

If you aren’t used to using modules, then this code may have made you twitch uncontrollably:

const style = {

color: 'darkcyan',

background: 'mintcream'

};

Defining a variable named style in the top-level scope would be an extremely bad idea in many JavaScript environments! In React, however, it’s totally fine.

Remember that every file is invisible to every other file, except for what you choose to expose via module.exports. You could have 100 different files, all with global variables named style, and there could be no conflicts.

import React from 'react';

import ReactDOM from 'react-dom';

const styles = {

background: 'lightblue',

color: 'darkred'

};

const styleMe = <h1 style={styles}>Please style me! I am so bland!</h1>;

ReactDOM.render(

    styleMe,

    document.getElementById('app')

);

# Style Name Syntax

In regular **JavaScript**, style names are written in **hyphenated-lowercase**:

const styles = {

'margin-top': "20px",

'background-color': "green"

};

**In React,** those same names are instead written in **camelCase**:

const styles = {

marginTop: "20px",

backgroundColor: "green"

};

This has zero effect on style property values, only on style property names.

import React from 'react';

import ReactDOM from 'react-dom';

const styles = {

background: 'lightblue',

color: 'darkred',

marginTop: '100px', // changes name write in camel case

fontSize: '50px'

};

const styleMe = <h1 style={styles}>Please style me! I am so bland!</h1>;

ReactDOM.render(

  styleMe,

  document.getElementById('app')

);

# Style Value Syntax

In the last exercise, you learned how style names are slightly different in React than they are in regular JavaScript.

In this exercise, you will learn how style values are slightly different in React than they are in regular JavaScript.

In regular JS, style values are almost always strings. Even if a style value is numeric, you usually have to write it as a string so that you can specify a unit. For example, you have to write "450px" or "20%".

In React, if you write a style value as a number, then the unit "px" is assumed.

How convenient! If you want a font size of 30px, you can write:

{ fontSize: 30 }

If you want to use units other than “px,” you can use a string:

{ fontSize: "2em" }

Specifying “px” with a string will still work, although it’s redundant.

A few specific styles will not automatically fill in the “px” for you. These are styles where you aren’t likely to use “px” anyway, so you don’t really have to worry about it.

<https://reactjs.org/docs/dom-elements.html>

import React from 'react';

import ReactDOM from 'react-dom';

const styles = {

background: 'lightblue',

color: 'darkred',

marginTop: 100, // changed value string to number

fontSize: 50

};

const styleMe = <h1 style={styles}>Please style me! I am so bland!</h1>;

ReactDOM.render(

    styleMe,

    document.getElementById('app')

);

# Share Styles Across Multiple Components

What if you want to reuse styles for several different components?

One way to make styles reusable is to keep them in a separate JavaScript file. This file should export the styles that you want to reuse, via export. You can then import your styles into any component that wants them.

style.js

/\*

Step 1

No, definitely not impressive enough.

You need to export your styles from styles.js, so that you can import them into your other files.

In styles.js, export your styles:

\*/

const fontFamily = 'Comic Sans MS, Lucida Handwriting, cursive';

const background = 'pink url("https://media.giphy.com/media/oyr89uTOBNVbG/giphy.gif") fixed';

const fontSize = '4em';

const padding = '45px 0';

const color = 'green';

export const styles = {

fontFamily: fontFamily,

background: background,

fontSize: fontSize,

padding: padding,

color: color

};

**AttentionGrabber.js**

/\*

STEP-2

These styles are all you’ve got! You’re going to have to use them in both Home.js and AttentionGrabber.js.

In Home.js, create a new line after import { AttentionGrabber } from './AttentionGrabber';.

On your new line, import styles from styles.js.

styles.js, Home.js, and AttentionGrabber.js all share the same parent directory.

Create a new line after import React from 'react';. On your new line, import your exported styles variable again

\*/

import React from 'react';

import { styles } from './styles';

// DEFINE NEW VARIABLE

const h1Style = {

color: styles.color,

fontSize: styles.fontSize,

fontFamily: styles.fontFamily,

padding: styles.padding,

margin: 0

};

//In AttentionGrabber‘s render function, give the <h1> a style attribute of {h1Style}.

export class AttentionGrabber extends React.Component {

    render() {

        return <h1 style={h1Style}>WELCOME TO MY HOMEPAGE!</h1>;

    }

}

**home.js**

/\*

STEP-2

These styles are all you’ve got! You’re going to have to use them in both Home.js and AttentionGrabber.js.

In Home.js, create a new line after import { AttentionGrabber } from './AttentionGrabber';.

On your new line, import styles from styles.js.

styles.js, Home.js, and AttentionGrabber.js all share the same parent directory.

\*/

import React from 'react';

import ReactDOM from 'react-dom';

import { AttentionGrabber } from './AttentionGrabber';

import { styles } from './styles';

// DEFINE NEW VARIABLE

const divStyle = {

background: styles.background,

height: '100%'

};

//In Home‘s render function, give the <div> a style attribute of {divStyle}

export class Home extends React.Component {

render() {

return (

<div style={divStyle}>

<AttentionGrabber />

<footer>THANK YOU FOR VISITING MY HOMEPAGE!</footer>

</div>

);

}

}

ReactDOM.render(

    <Home />,

    document.getElementById('app')

);

**CONTAINER COMPONENTS FROM PRESENTATIONAL COMPONENTS**

# Separate Container Components From Presentational Components: Explanation

Here, you will learn your second programming pattern: **separating presentational components from display components.**

Click Run. In the browser, navigate to https://localhost:8000.

You are looking at an rendered <GuineaPigs />component.

<GuineaPigs />‘s job is to render a photo carousel of guinea pigs. It does this perfectly well! And yet, it has a problem: it does too much stuff.

We can break <GuineaPigs /> into smaller components, but before we do: how do we know that GuineaPigsdoes too much stuff? How can you tell when a component has too many responsibilities?

Separating container components from presentational components helps to answer that question. It shows you when it might be a good time to divide a component into smaller components. It also shows you how to perform that division.

**GuineaPigs.js**

import React from 'react';

import ReactDOM from 'react-dom';

const GUINEAPATHS = [

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-1.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-2.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-3.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-4.jpg'

];

export class GuineaPigs extends React.Component {

constructor(props) {

super(props);

this.state = { currentGP: 0 };

this.interval = null;

this.nextGP = this.nextGP.bind(this);

}

nextGP() {

let current = this.state.currentGP;

let next = ++current % GUINEAPATHS.length;

this.setState({ currentGP: next });

}

componentDidMount() {

this.interval = setInterval(this.nextGP, 5000);

}

componentWillUnmount() {

clearInterval(this.interval);

}

render() {

let src = GUINEAPATHS[this.state.currentGP];

return (

<div>

<h1>Cute Guinea Pigs</h1>

<img src={src} />

</div>

);

}

}

ReactDOM.render(

<GuineaPigs />,

document.getElementById('app')

);

# Separate Container Components From Presentational Components: Apply

Separating container components from presentational components is a popular React programming pattern.

Here’s the basic idea behind it: if a component has to have state, make calculations based on props, or manage any other complex logic, then that component shouldn’t also have to render HTML-like JSX.

Instead of rendering HTML-like JSX, the component should render another component. It should be thatcomponent’s job to render HTML-like JSX.

Following this pattern separates your business logicfrom your presentational logic, which is a [Good Thing](http://www.dictionary.com/browse/good-thing).

<https://medium.com/@learnreact/container-components-c0e67432e005>

GuineaPigsContainer.js

import React from 'react';

import ReactDOM from 'react-dom';

import { GuineaPigs } from '../components/GuineaPigs';

const GUINEAPATHS = [

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-1.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-2.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-3.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-4.jpg'

];

class GuineaPigsContainer extends React.Component {

constructor(props) {

super(props);

this.state = { currentGP: 0 };

this.interval = null;

this.nextGP = this.nextGP.bind(this);

}

nextGP() {

let current = this.state.currentGP;

let next = ++current % GUINEAPATHS.length;

this.setState({ currentGP: next });

}

componentDidMount() {

this.interval = setInterval(this.nextGP, 5000);

}

componentWillUnmount() {

clearInterval(this.interval);

}

render() {

let src = GUINEAPATHS[this.state.currentGP];

return <GuineaPigs src={src} />;

}

}

ReactDOM.render(

<GuineaPigsContainer />,

document.getElementById('app')

);

GuineaPigs.js

import React from 'react';

/\*

Look at the GuineaPigs component class, starting on line 11.

This is going to be your

presentational component class.

That means that its only job will be to render HTML-like JSX.

\*/

import React from 'react';

export class GuineaPigs extends React.Component {

render() {

const src = this.props.src;

return (

<div>

<h1>Cute Guinea Pigs</h1>

<img src={src} />

</div>

);

}

}

**STATELESS FUNCTIONAL COMPONENTS**

# Stateless Functional Components

In the code editor, take a look at GuineaPigs from the last lesson.

Notice that its instructions object only has one property:render().

When you separate a container component from a presentational component, the presentationalcomponent will always end up like this: one render()function, and no other properties.

**If you have a component class with nothing but a render function, then you can rewrite that component class in a very different way. Instead of using React.Component, you can write it as a JavaScript function!**

is a **stateless functional component** the same as a **stateless component**?

### Answer

Although similar in some aspects, a **stateless functional component** is not the same as a **stateless component**. Some of the differences are as follows:

A **stateless functional component** can be seen just as a function which returns JSX based on the prop values that are passed to it. It is not a class and does not extend the React.Component class. It also has no state.

The general structure typically looks like the following

const Example = **(props)** => {

**return** ...

}

A **stateless component** is a class and extends the React.Component class. It has no state and no life-cycle unlike regular stateful components. These are typically used by a parent component for rendering something, like text or information, without the need for it to perform any actions.

The general structure is similar to a regular stateful component, but usually just has a render()method,

**class** **Example** **extends** **React**.**Component** {

render() {

**return** ...

}

}

A **component** class written as a **function** is called a **stateless functional component.** Stateless functional components have some advantages over typical component classes. We’ll cover those advantages in this lesson.

Click on **Example.js** to see a stateless functional component in action.

// A component class written in the usual way:

export class MyComponentClass extends React.Component {

render() {

return <h1>Hello world</h1>;

}

}

// The same component class, written as a stateless functional component:

export const MyComponentClass = () => {

return <h1>Hello world</h1>;

}

// Works the same either way:

ReactDOM.render(

<MyComponentClass />,

document.getElementById('app')

);

Example 2 : Friend.js

import React from 'react';

import ReactDOM from 'react-dom';

// The same component class, written as a stateless functional component:

export const Friend = () => {

return <img src='https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-octopus.jpg' />;

}

ReactDOM.render(

    <Friend />,

    document.getElementById('app')

);

# Stateless Functional Components and Props

Stateless functional components usually have propspassed to them.

To access these props, give your stateless functional component a parameter. This parameter will automatically be equal to the component’s propsobject.

It’s customary to name this parameter props. Read **Example.js** to see how it works.

Not only are stateless functional components more concise, but they will subtly influence how you think about components in a positive way. They emphasize the fact that components are basically functions! A component takes two optional inputs, props and state, and outputs HTML and/or other components.

You’ll be seeing a lot of stateless functional components in the next React course!

can a stateless functional component have more parameters, other than props and state?

### Answer

No, stateless functional components in React can only accept props and state as optional parameters. As a result, you cannot do something like the following, which adds other parameters other than propsor state,

**export** const MyComponentClass = **(props, param2, param3)** => {

**return** ...

}

If you need to pass any values to the component, you should always do so by passing them as props, without the need for adding other parameters.

Open **GuineaPigs.js**.

After you divided GuineaPigs into GuineaPigsand GuineaPigsContainer, GuineaPigs was left with only a render function. That means that GuineaPigs can be rewritten as a *stateless functional component!*

Rewrite the GuineaPigs component class as a stateless functional component. Use **Example**as a guide.

Make sure to delete the original GuineaPigsclass when you’re done.

Click Run and make sure that your guinea pigs are still working!

**Example.js**

// Normal way to display a prop:

export class MyComponentClass extends React.Component {

render() {

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

}

}

// Stateless functional component way to display a prop:

export const MyComponentClass = (props) => {

return <h1>{props.title}</h1>;

}

// Normal way to display a prop using a variable:

export class MyComponentClass extends React.component {

render() {

let title = this.props.title;

return <h1>{title}</h1>;

}

}

// Stateless functional component way to display a prop using a variable:

export const MyComponentClass = (props) => {

let title = props.title;

return <h1>{title}</h1>;

}

GuineaPigs.js

import React from 'react';

export const GuineaPigs = (props) => {

let src = props.src;

return (

<div>

<h1>Cute Guinea Pigs</h1>

<img src={src} />

</div>

);

}

**PROPTYPES**

# propTypes

In this lesson, you will learn to use an important React feature called propTypes.

propTypes are useful for two reasons. The first reason **is prop validation.**

**Validation can ensure that your props are doing what they’re supposed to be doing**. If props are missing, or if they’re present but they aren’t what you’re expecting, then a warning will print in the console.

This is useful, but reason #2 is arguably more useful: documentation.

Documenting props makes it easier to glance at a file and quickly understand the component class inside. When you have a lot of files, and you will, this can be a huge benefit.

# Apply PropTypes

In the code editor, take a look at MessageDisplayer‘s render function.

Notice the expression this.props.message. From this expression, you can deduce that MessageDisplayerexpects to get passed a prop named message. Somewhere, at some time, this code is expected to execute:

<MessageDisplayer message="something" />

If a component class expects a prop, then you can give that component class a propType!

The first step to making a propType is to search for a property named propTypes on the instructions object. If there isn’t one, make one! You will have to declare it after the close of your component declaration, since it will be a static property.

See the example of a propTypes property on lines 11-13. **Notice that the value of propTypes is an object, not a function!**

The second step is to add a property to the propTypesobject. For each prop that your component class expects to receive, there can be one property on your propTypes object.

MessageDisplayer only expects one prop: message. Therefore, its propTypes object only has one property.

Example.js

import React from 'react';

export class MessageDisplayer extends React.Component {

render() {

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

}

}

// This propTypes object should have

// one property for each expected prop:

MessageDisplayer.propTypes = {

message: React.PropTypes.string

};

# Add Properties to PropTypes

In the code editor, look at the property on MessageDisplayer‘s propTypes object:

message: React.PropTypes.string

What are the properties on propTypes supposed to be, exactly?

The name of each property in propTypes should be the name of an expected prop. In our case, MessageDisplayer expects a prop named message, so our property’s name is message.

The value of each property in propTypes should fit this pattern:

React.PropTypes.expected-data-type-goes-here

Since message is presumably going to be a string, we chose React.PropTypes.string. You can see this on line 12. Notice the difference in capitalization between the propTypes object and React.PropTypes!

Each property on the propTypes object is called a propType.

Select the next file in code editor, **Runner.js**. Find Runner‘s propTypes object.

Runner has six propTypes! Look at each one. Note that bool and func are abbreviated, but all other datatypes are spelled normally.

If you add .isRequired to a propType, then you will get a console warning if that prop isn’t sent.

Try to find all six props from the propTypes object in Runner‘s render function: this.props.message, this.props.style, etc.

**Example(add multiple prop type)**

import React from 'react';

export class Runner extends React.Component {

render() {

  let miles = this.props.miles;

let km = this.props.milesToKM(miles);

let races = this.props.races.map(function(race, i){

return <li key={race + i}>{race}</li>;

});

return (

<div style={this.props.style}>

<h1>{this.props.message}</h1>

{ this.props.isMetric &&

<h2>One Time I Ran {km} Kilometers!</h2> }

{ !this.props.isMetric &&

<h2>One Time I Ran {miles} Miles!</h2> }

<h3>Races I've Run</h3>

<ul id="races">{races}</ul>

</div>

);

}

}

Runner.propTypes = {

message: React.PropTypes.string.isRequired,

style: React.PropTypes.object.isRequired,

isMetric: React.PropTypes.bool.isRequired,

miles: React.PropTypes.number.isRequired,

milesToKM: React.PropTypes.func.isRequired,

races: React.PropTypes.array.isRequired

};

BestSeller.js (component class)

import React from 'react';

export class BestSeller extends React.Component {

render() {

return (

<li>

Title: <span>

{this.props.title}

</span><br />

Author: <span>

{this.props.author}

</span><br />

Weeks: <span>

{this.props.weeksOnList}

</span>

</li>

);

}

}

BestSeller.propTypes = {

title: React.PropTypes.string.isRequired,

author: React.PropTypes.string.isRequired,

weeksOnList: React.PropTypes.number.isRequired

};

BookList.js (container class)

import React from 'react';

import ReactDOM from 'react-dom';

import { BestSeller } from './BestSeller';

export class BookList extends React.Component {

render() {

return (

<div>

<h1>Best Sellers</h1>

<div>

<ol>

<BestSeller

title="Glory and War Stuff for Dads"

author="Sir Eldrich Van Hoorsgaard"

weeksOnList={10} />

<BestSeller

title="The Crime Criminals!"

author="Brenda Sqrentun"

weeksOnList={2} />

<BestSeller

title="Subprime Lending For Punk Rockers"

author="Malcolm McLaren"

weeksOnList={600} />

</ol>

</div>

</div>

);

}

}

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

# PropTypes in Stateless Functional Components

Remember stateless functional components? You can see some familiar ones in **Example.js**.

How could you write propTypes for a stateless functional component?

// Usual way:

class Example extends React.component{

}

Example.propTypes = {

};

...

// Stateless functional component way:

const Example = (props) => {

// ummm ??????

It turns out the process is fairly similar. To write propTypes for a stateless functional component, you define a propTypes object as a property of the stateless functional component itself. Here’s what that looks like:

const Example = (props) => {

return <h1>{props.message}</h1>;

}

Example.propTypes = {

message: React.PropTypes.string.isRequired

};

Example.js

// Normal way to display a prop:

export class MyComponentClass extends React.Component {

render() {

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

}

}

// Stateless functional component way to display a prop:

export const MyComponentClass = (props) => {

return <h1>{props.title}</h1>;

}

// Normal way to display a prop using a variable:

export class MyComponentClass extends React.component {

render() {

let title = this.props.title;

return <h1>{title}</h1>;

}

}

// Stateless functional component way to display a prop using a variable:

export const MyComponentClass = (props) => {

let title = props.title;

return <h1>{title}</h1>;

}

GuineaPigs.JS

import React from 'react';

export const GuineaPigs = (props) => {

  let src = props.src;

return (

<div>

<h1>Cute Guinea Pigs</h1>

<img src={src} />

</div>

)

}

GuineaPigs.propTypes = {

src: React.PropTypes.string.isRequired

};

**GuineaPigsContainer.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { GuineaPigs } from '../components/GuineaPigs';

const GUINEAPATHS = [

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-1.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-2.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-3.jpg',

'https://s3.amazonaws.com/codecademy-content/courses/React/react\_photo-guineapig-4.jpg'

];

export class GuineaPigsContainer extends React.Component {

constructor(props) {

super(props);

this.state = { currentGP: 0 };

}

nextGP() {

let current = this.state.currentGP;

let next = ++current % GUINEAPATHS.length;

this.setState({ currentGP: next });

}

componentDidMount() {

this.interval = setInterval(this.nextGP, 5000);

}

componentWillUnmount() {

clearInterval(this.interval);

}

render() {

let src = GUINEAPATHS[this.state.currentGP];

return <GuineaPigs src={src} />;

}

});

ReactDOM.render(

<GuineaPigsContainer />,

document.getElementById('app')

);

**REACT FORMS**

# React Forms

Think about how forms work in a typical, non-React environment. A user types some data into a form’s input fields, and the server doesn’t know about it. The server remains clueless until the user hits a “submit” button, which sends all of the form’s data over to the server simultaneously.

In React, as in many other JavaScript environments, this is not the best way of doing things.

The problem is the period of time during which a form thinks that a user has typed one thing, but the server thinks that the user has typed a different thing. What if, during that time, a third part of the website needs to know what a user has typed? It could ask the form or the server and get two different answers. In a complex JavaScript app with many moving, interdependent parts, this kind of conflict can easily lead to problems.

In a React form, you want the server to know about every new character or deletion, as soon as it happens. That way, your screen will always be in sync with the rest of your application.

**Example.js**

import React from 'react';

export class Example extends React.Component {

  constructor(props) {

    super(props);

    this.state = { userInput: '' };

    this.handleChange = this.handleChange.bind(this);

  }

  handleChange(e) {

   this.setState({

   userInput: e.target.value

   });

  }

  render() {

   return (

   <input

   onChange={this.handleChange}

   type="text" />

   );

  }

}

# Input onChange

A traditional form doesn’t update the server until a user hits “submit.” But you want to update the server any time a user enters or deletes any character. You can see in **Example.js**

what does onChange mean?

### Answer

onChange means to invoke some function on each change of an input field. It is similar to onClick in that it follows a similar naming convention, and it is applied in a similar way to an element. A “change” happens whenever there is a change to the value of the input form, such as by typing in a character, deleting characters, or copying and pasting text into the field.

# Write an Input Event Handler

In this exercise, you will define a function that gets called whenever a user enters or deletes any character.

This function will be an event handler. It will listen for change events. You can see an example of an event handler listening for change events in **Example.js**.

The e parameter of the function is an optional parameter of the input event handler which equals to a JavaScript Event object that contains information regarding what action or event just happened. When the handleUserInput() function is invoked, the action that occurs, like a change in the input field, is passed as the value of e. Then, the information about the action can be accessed, such as e.target.value, which would be the value of the text input element.

# Set the Input's Initial State

Good! Any time that someone types or deletes in <input />, the .handleUserInput() method will update this.state.userInput with the <input />‘s text.

Since you’re using this.setState, that means that Input needs an initial state! What should this.state‘s initial value be?

Well, this.state.userInput will be displayed in the <input />. What should the initial text in the <input />be, when a user first visits the page?

The initial text should be blank! Otherwise it would look like someone had already typed something. You can see in **Example.js**

**Input.js**

import React from 'react';

import ReactDOM from 'react-dom';

export class Input extends React.Component {

constructor(props) {

super(props);

this.state = { userInput: '' };

this.handleUserInput = this.handleUserInput.bind(this);

}

 handleUserInput(e) {

   this.setState({

   userInput: e.target.value

   });

  }

render() {

return (

<div>

<input type="text" onChange={this.handleUserInput}/>

<h1>I am an h1.</h1>

</div>

);

}

}

ReactDOM.render(

  <Input />,

  document.getElementById('app')

);

Every time we change the input of the text field, whether by adding or deleting some text, it will trigger the onChange attribute’s function, handleUserInput(). When this function is invoked, it will update the state of the component to the current value of the text field using setState(). Any time setState() is called for a component, it will trigger a call to the render() method, which will re-display the text to the latest value of the text field.

# Update an Input's Value

When a user types or deletes in the <input />, then that will trigger a change event, which will call handleUserInput. That’s good!

handleUserInput will set this.state.userInput equal to whatever text is currently in the input field. That’s also good!

There’s only one problem: you can set this.state.userInput to whatever you want, but <input /> won’t care. You need to somehow make the <input />‘s text responsive to this.state.userInput.

Easy enough! You can control an <input />‘s text by setting its value attribute.

Input.js

import React from 'react';

import ReactDOM from 'react-dom';

export class Input extends React.Component {

constructor(props) {

super(props);

this.state = { userInput: '' };

this.handleUserInput = this.handleUserInput.bind(this);

}

 handleUserInput(e) {

   this.setState({

   userInput: e.target.value

   });

  }

render() {

return (

<div>

<input type="text" onChange={this.handleUserInput} value={this.state.userInput}/>

<h1>{this.state.userInput}</h1>

</div>

);

}

}

ReactDOM.render(

  <Input />,

  document.getElementById('app')

);

, can we set a placeholder value for the input field?

### Answer

Yes, there are a few ways you can do this. The first method was covered in the previous exercise, which is done by setting an initial value to the state. This method places some actual text in the field, so a user would have to edit it. For instance,

this.state = { userInput: 'Some initial text' };

Another method is to utilize the input HTML element’s attribute placeholder, which sets some placeholder text in the field, but unlike the previous method, it will not be an actual value. It can be done like so, by adding the parameter,

<input placeholder="placeholder text" />

# Controlled vs Uncontrolled

There are two terms that will probably come up when you talk about React forms: controlled component and uncontrolled component. Like automatic binding, controlled vs uncontrolled components is a topic that you should be familiar with, but don’t need to understand deeply at this point.

An uncontrolled component is a component that maintains its own internal state. A controlled component is a component that does not maintain any internal state. Since a controlled component has no state, it must be controlled by someone else.

Think of a typical <input type='text' /> element. It appears onscreen as a text box. If you need to know what text is currently in the box, then you can ask the <input />, possibly with some code like this:

let input = document.querySelector('input[type="text"]');

let typedText = input.value; // input.value will be equal to whatever text is currently in the text box.

The important thing here is that the <input /> keeps track of its own text. You can ask it what its text is at any time, and it will be able to tell you.

The fact that <input /> keeps track of information makes it an uncontrolled component. It maintains its own internal state, by remembering data about itself.

A controlled component, on the other hand, has no memory. If you ask it for information about itself, then it will have to get that information through props. Most React components are controlled.

In React, when you give an <input /> a valueattribute, then something strange happens: the <input /> BECOMES controlled. It stops using its internal storage. This is a more ‘React’ way of doing things.

<https://reactjs.org/docs/forms.html>

 Do controlled and uncontrolled just mean stateless and stateful?

### Answer

No, controlled and uncontrolled cannot be used interchangeably with the terms stateless and stateful. Controlled and uncontrolled are always used in the context of React forms, which can be stateless or stateful. However, controlled and uncontrolled components are usually attributed with a stateless and a stateful nature, respectively.

A controlled component has its state controlled from outside. Because of this, controlled components are usually stateless as they do not store their own state and receive it from a parent component.

Uncontrolled components are essentially stateful, as they store and maintain their own internal state.

**MOUNTING LIFECYCLE METHODS**

# What's a Lifecycle Method?

**Lifecycle methods** are methods that **get called** at certain **moments** in a **component’s life**.

You can write a lifecycle method that gets **called** **right before** **a component renders** for the first time.

You can write a lifecycle method that gets **called** **right after** a **component renders,** every time except for the first time.

You **can attach lifecycle** methods to **a lot of different moments** in a component’s life. This has powerful implications!

**Mounting Lifecycle Methods**

There are three categories of lifecycle methods: *mounting*, *updating*, and *unmounting*.

*mounting* lifecycle methods.

A component “mounts” when it renders for the first time. This is when *mounting lifecycle methods* get called.

There are three *mounting lifecycle methods:*

* componentWillMount
* render
* componentDidMount

When a component *mounts*, it automatically calls these three methods, in order.

**componentWillMount**

The first *mounting* lifecycle method is called componentWillMount.

When a component renders for the first time, componentWillMount gets called right *before* render.

Look at **Example.js**, and follow these steps:

1. On lines 14-17, <Example /> is rendered for the first time. <Example />‘s *mounting* period begins.
2. <Example /> calls the first mounting lifecycle method, componentWillMount.
3. componentWillMount executes, and an alert appears on the screen. (lines 5-7)
4. After componentWillMount has finished, <Example />calls the second mounting lifecycle method: render.
5. <h1>Hello world</h1> appears on the screen (lines 9-11)
6. Two seconds later, <Example /> renders again (lines 20-22). componentWillMount does NOT get called, because mounting lifecycle events only execute the first time that a component renders.

You can call this.setState from within componentWillMount!

Look at **Example2.js** for an example of this.setStateinside of componentWillMount. See if you can follow how <Example2 /> would render <h1>Hello world</h1>.

**Example1.js**

1. import React from 'react';
2. import ReactDOM from 'react-dom';

3

1. export class Example extends React.Component {

5.

6. componentWillMount() {

1. alert('component is about to mount!');
2. }
3. render() {
4. return <h1>Hello world</h1>;
5. }
6. }
7. ReactDOM.render(
8. <Example />,
9. document.getElementById('app')
10. );
11. setTimeout(() => {
12. ReactDOM.render(
13. <Example />,
14. document.getElementById('app')
15. );
16. }, 2000);

**Example2.js**

import React from 'react';

import ReactDOM from 'react-dom';

export class Example2 extends React.Component {

constructor(props) {

super(props);

this.state = { text: '' };

}

componentWillMount() {

this.setState({ text: 'Hello world' });

}

render() {

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

}

}

ReactDOM.render(

<Example2 />,

document.getElementById('app')

);

**Code Flashy.js**

import React from 'react';

import ReactDOM from 'react-dom';

export class Flashy extends React.Component {

componentWillMount() {

alert('AND NOW, FOR THE FIRST TIME EVER... FLASHY!!!!');

}

render() {

alert('Flashy is rendering!');

return (

<h1 style={{ color: this.props.color }}>

OOH LA LA LOOK AT ME I AM THE FLASHIEST

</h1>

);

}

}

ReactDOM.render(

<Flashy color='red' />,

document.getElementById('app')

);

setTimeout(() => {

ReactDOM.render(

<Flashy color='green' />,

document.getElementById('app')

);

}, 2000);

# render

render is a lifecycle method!

We won’t go over render here - we’ve already talked about it plenty. However, you should understand how render fits into the mounting period. Whenever a component mounts, componentWillMount is called first, followed by render, followed by componentDidMount.

render belongs to two categories: mounting lifecycle methods, and updating lifecycle methods. We’ll cover updating lifecycle methods in the next lesson. But first, there’s one final mounting lifecycle method!

# componentDidMount

The final mounting lifecycle method is called componentDidMount.

When a component renders for the first time, componentDidMount gets called right after the HTML from render has finished loading. Look in the code editor for an example of componentDidMount.

componentDidMount gets used a lot!

If your React app uses AJAX to fetch initial data from an API, then componentDidMount is the place to make that AJAX call. More generally, componentDidMount **is a good place to connect a React app to external applications, such as web APIs or JavaScript frameworks**.componentDidMount is also the **place to set timers** using setTimeout or setInterval.

If that sounds vague, don’t worry. You’ll put lifecycle methods into practice in this course’s final project, and then more in the next course! Not to mention in the real world…

Introducing the componentDidMount() method, can you set the component’s state in this method?

### Answer

Yes, you can set the component’s state in this method, using the setState() method. This is not possible in some other lifecycle methods, such as render(), because calling setState() within them can result in infinite loops.

Calling setState() in componentDidMount() will not end up in an infinite loop, but will just invoke an additional rendering. This additional rendering will not be visible as both of the renders will happen be right before the browser updates the screen, but, one drawback to doing this is that the additional render can take some additional time and cause the site to display slower.

Instead of setting the state initially within componentDidMount(), you may consider setting the initial state in the constructor() method instead.

**Flashy2.js**

import React from 'react';

import ReactDOM from 'react-dom';

export class Flashy extends React.Component {

componentWillMount() {

alert('AND NOW, FOR THE FIRST TIME EVER... FLASHY!!!!');

}

componentDidMount() {

alert('YOU JUST WITNESSED THE DEBUT OF... FLASHY!!!!!!!');

}

render() {

alert('Flashy is rendering!');

return (

<h1 style={{ color: this.props.color }}>

OOH LA LA LOOK AT ME I AM THE FLASHIEST

</h1>

);

}

}

ReactDOM.render(

<Flashy color='red' />,

document.getElementById('app')

);

setTimeout(() => {

ReactDOM.render(

<Flashy color='green' />,

document.getElementById('app')

);

}, 2000);

**UPDATING/UNMOUNTING LIFECYCLE METHODS**

**Updating Lifecycle Methods**

There are two categories that we haven’t yet discussed: *updating* and *unmounting* lifecycle methods.

What is *updating*?

The **first time that a component instance renders**, it **does not update**. A **component updates every time** that it renders, ***starting with the second render*.**

There are five updating lifecycle methods:

* componentWillReceiveProps
* shouldComponentUpdate
* componentWillUpdate
* render
* componentDidUpdate

Whenever a component instance *updates*, it automatically calls all five of these methods, in order.

1. Updating Lifecycle Methods

# componentWillReceiveProps

The first updating lifecycle method is called componentWillReceiveProps.

When a component instance updates, componentWillReceiveProps gets called before the rendering begins.

**As one might expect, componentWillReceiveProps only gets called if the component will receive props:**

// componentWillReceiveProps will get called here:

ReactDOM.render(

<Example prop="myVal" />,

document.getElementById('app')

);

// componentWillReceiveProps will NOT get called here:

ReactDOM.render(

<Example />,

document.getElementById('app')

);

Look in the code editor for an example of componentWillReceiveProps. Read it through and try to figure out how it works.

componentWillReceiveProps automatically gets passed one argument: an object called nextProps. nextProps is a preview of the upcoming props object that the component is about to receive.

On line 6, nextProps.text will evaluate to "Hello world".

import React from 'react';

export class Example extends React.Component {

componentWillReceiveProps(nextProps) {

alert("Check out the new props.text that "

    + "I'm about to get: " + nextProps.text); // line six

}

render() {

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

}

}

// The first render won't trigger

// componentWillReceiveProps:

ReactDOM.render(

    <Example text="Hello world" />,

    document.getElementById('app')

);

// After the first render,

// subsequent renders will trigger

// componentWillReceiveProps:

setTimeout(() => {

    ReactDOM.render(

        <Example text="Hello world" />,

        document.getElementById('app')

    );

}, 1000);

This is a common use of componentWillReceiveProps: comparing incoming props to current props or state, and deciding what to render based on that comparison.

<https://reactjs.org/docs/components-and-props.html>

Does componentWillReceiveProps always run even if the props didn’t change?

### Answer

Yes, the method will run anytime the component receives any props, whether or not the props are new or the same as before.

To prevent the method from running its code anytime props are passed in, and only when new props are passed in, you can override the method code, for instance by comparing this.props with nextPropsand only running if they are different

# shouldComponentUpdate

The second updating lifecycle method is called shouldComponentUpdate.

When a component updates, shouldComponentUpdategets called after componentWillReceiveProps, but still before the rendering begins.

Look at **Example.js** in the code editor. Read it through and try to figure out how shouldComponentUpdate works.

shouldComponentUpdate **should return either true or false.**

If shouldComponentUpdate returns true, then nothing noticeable happens. But if shouldComponentUpdatereturns **false**, then the component will not update! **None of the remaining lifecycle methods for that updating period will be called, including render.**

**The best way to use shouldComponentUpdate is to have it return false only under certain conditions.** If those conditions are **met**, then your component will not update.

shouldComponentUpdate automatically receives two arguments: nextProps and nextState. It’s typical to compare nextProps and nextState to the current this.props and this.state, and use the results to decide what to do. See how Example.js does this on lines 10-19.

import React from 'react';

export class Example extends React.Component {

constructor(props) {

super(props);

this.state = { subtext: 'Put me in an <h2> please.' };

}

shouldComponentUpdate(nextProps, nextState) {

if ((this.props.text == nextProps.text) &&

(this.state.subtext == nextState.subtext)) {

alert("Props and state haven't changed, so I'm not gonna update!");

return false;

} else {

alert("Okay fine I will update.")

return true;

}

}

render() {

return (

<div>

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

<h2>{this.state.subtext}</h2>

</div>

);

}

}

# componentWillUpdate

The third updating lifecycle method is componentWillUpdate.

componentWillUpdate gets called in between shouldComponentUpdate and render.

componentWillUpdate receives two arguments**: nextProps and nextState**. Read Example in the code editor to see it in action.

***You cannot call this.setState from the body of componentWillUpdate!*** Which begs the question, why would you use it?

The main purpose of componentWillUpdate is to **interact with things outside of the React architecture**. If you need to do **non-React setup** before a component renders, such as **checking the window size or interacting with an API, then componentWillUpdate is a good place to do that.**

If that sounds abstract, that’s okay! All of the lifecycle methods might feel a bit theoretical, until you’ve used them in real-life scenarios.

import React from 'react';

export class Example extends React.Component {

componentWillUpdate(nextProps, nextState) {

alert('Component is about to update! Any second now!');

}

render() {

return <h1>Hello world</h1>;

}

}

# componentDidUpdate

The last updating lifecycle method is componentDidUpdate.

When a component instance updates, componentDidUpdate gets called after any rendered HTML has finished loading.

Look at Example for an example of componentDidUpdate.

componentDidUpdate automatically gets passed two arguments: prevProps and prevState. **prevProps and prevState are references to the component’s propsand state before the current updating period began**. You can compare them to the current props and state.

componentDidUpdate is usually used for **interacting with things outside of the React environment, like the browser or APIs.** It’s similar to componentWillUpdate in that way, except that it gets called after render instead of before.

import React from 'react';

export class Example extends React.component {

componentDidUpdate(prevProps, prevState) {

alert('Component is done rendering!');

}

render() {

return <h1>Hello world</h1>;

}

}

**Example:**

**app.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { TopNumber } from './TopNumber';

import { Display } from './Display';

import { Target } from './Target';

import { random, clone } from './helpers';

const fieldStyle = {

position: 'absolute',

width: 250,

bottom: 60,

left: 10,

height: '60%',

};

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

game: false,

targets: {},

latestClick: 0

};

this.intervals = null;

this.hitTarget = this.hitTarget.bind(this);

this.startGame = this.startGame.bind(this);

this.endGame = this.endGame.bind(this);

}

componentDidUpdate(prevProps, prevState) {

if (this.state.latestClick < prevState.latestClick) {

this.endGame();

}

}

createTarget(key, ms) {

ms = ms || random(500, 2000);

this.intervals.push(setInterval(function(){

let targets = clone(this.state.targets);

let num = random(1, 1000\*1000);

targets[key] = targets[key] != 0 ? 0 : num;

this.setState({ targets: targets });

}.bind(this), ms));

}

hitTarget(e) {

if (e.target.className != 'target') return;

let num = parseInt(e.target.innerText);

for (let target in this.state.targets) {

let key = Math.random().toFixed(4);

this.createTarget(key);

}

this.setState({ latestClick: num });

}

startGame() {

this.createTarget('first', 750);

this.setState({

game: true

});

}

endGame() {

this.intervals.forEach((int) => {

clearInterval(int);

});

this.intervals = [];

this.setState({

game: false,

targets: {},

latestClick: 0

});

}

componentWillMount() {

this.intervals = [];

}

render() {

let buttonStyle = {

display: this.state.game ? 'none' : 'inline-block'

};

let targets = [];

for (let key in this.state.targets) {

targets.push(

<Target

number={this.state.targets[key]}

key={key} />

);

}

return (

<div>

<TopNumber number={this.state.latestClick} game={this.state.game} />

<Display number={this.state.latestClick} />

<button onClick={this.startGame} style={buttonStyle}>

New Game

</button>

<div style={fieldStyle} onClick={this.hitTarget}>

{targets}

</div>

</div>

);

}

}

ReactDOM.render(

<App />,

document.getElementById('app')

);

**topnumber.js**

import React from 'react';

import ReactDOM from 'react-dom';

const yellow = 'rgb(255, 215, 18)';

export class TopNumber extends React.Component {

constructor(props) {

super(props);

this.state = { 'highest': 0 };

}

componentWillUpdate(nextProps, nextState) {

if (document.body.style.background != yellow && this.state.highest >= 950\*1000) {

document.body.style.background = yellow;

} else if (!this.props.game && nextProps.game) {

document.body.style.background = 'white';

}

}

componentWillReceiveProps(nextProps) {

if (nextProps.number > this.state.highest) {

this.setState({

highest: nextProps.number

});

}

}

render() {

return (

<h1>

Top Number: {this.state.highest}

</h1>

);

}

}

TopNumber.propTypes = {

number: React.PropTypes.number,

game: React.PropTypes.bool

};

**target.js**

import React from 'react';

import { random } from './helpers';

export class Target extends React.Component {

shouldComponentUpdate(nextProps, nextState) {

return this.props.number != nextProps.number;

}

render() {

let visibility = this.props.number ? 'visible' : 'hidden';

let style = {

position: 'absolute',

left: random(0, 100) + '%',

top: random(0, 100) + '%',

fontSize: 40,

cursor: 'pointer',

visibility: visibility

};

return (

<span style={style} className="target">

{this.props.number}

</span>

)

}

}

Target.propTypes = {

number: React.PropTypes.number.isRequired

};

# componentWillUnmount

A component’s unmounting period occurs when the **component is removed** from the **DOM**. This could happen if the **DOM is rerendered without the component**, or if **the user navigates to a different website** or **closes their web browser**.

componentWillUnmount is **the only unmounting lifecycle method**!

componentWillUnmount gets **called right before** a **component is removed** from the DOM. If a component initiates any methods that require cleanup, then componentWillUnmount is where you should put that cleanup.

You can see an example in **Example.js**.

import React from 'react';

export class Example extends React.Component {

componentWillUnmount() {

alert('Goodbye world');

}

render() {

return <h1>Hello world</h1>;

}

}

**Enthused.js**

import React from 'react';

export class Enthused extends React.Component {

componentDidMount() {

this.interval = setInterval(() => {

this.props.addText('!');

}, 15);

}

componentWillUnmount(prevProps, prevState) {

clearInterval(this.interval);

}

render() {

return (

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

Stop!

</button>

);

}

}

**App.js**

import React from 'react';

import ReactDOM from 'react-dom';

import { Enthused } from './Enthused';

export class App extends React.Component {

constructor(props) {

super(props);

this.state = {

enthused: false,

text: ''

};

this.toggleEnthusiasm = this.toggleEnthusiasm.bind(this);

this.addText = this.addText.bind(this);

this.handleChange = this.handleChange.bind(this);

}

toggleEnthusiasm() {

this.setState({

enthused: !this.state.enthused

});

}

setText(text) {

this.setState({ text: text });

}

addText(newText) {

let text = this.state.text + newText;

this.setState({ text: text });

}

handleChange(e) {

this.setText(e.target.value);

}

render() {

let button;

if (this.state.enthused) {

button = (

<Enthused toggle={this.toggleEnthusiasm} addText={this.addText} />

);

} else {

button = (

<button onClick={this.toggleEnthusiasm}>

Add Enthusiasm!

</button>

);

}

return (

<div>

<h1>Auto-Enthusiasm</h1>

<textarea rows="7" cols="40" value={this.state.text}

onChange={this.handleChange}>

</textarea>

{button}

<h2>{this.state.text}</h2>

</div>

);

}

}

ReactDOM.render(

<App />,

document.getElementById('app')

);