**Hello World, Part II... THE COMPONENT**

React applications are made out of *components.*

What’s a component?

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

Take a look at the code below. 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. However you can recognize some JSX in there, as well as ReactDOM.render().

We are going to unpack that code, one small piece at a time. By the end of this lesson, you’ll understand how to build a React component!

**Instructions**

**1.**

Carefully copy the example code into **app.js**.

**Import React**

Wooo! Your first React component!

Take a look at the code on line 1:

import React from 'react';

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.

**Instructions**

**1.**

Select **new.js**.

On line 1, use import to import the React library. Save the library in a variable named React.

**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.

**Instructions**

**1.**

Select **new.js**.

Import the ReactDOM library on line 2. Store the result in a variable named ReactDOM.

import ReactDOM from 'react-dom'

**Create a Component Class**

You’ve learned that a *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.

What *is* React.Component, and how do you use it to make a component class?

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 beyond the scope of this course. If you aren’t comfortable with them, here are some good resources to consult: [1](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) [2](https://hacks.mozilla.org/2015/07/es6-in-depth-classes/) [3](https://hacks.mozilla.org/2015/08/es6-in-depth-subclassing/) [4](http://exploringjs.com/es6/ch_classes.html).

Look at the code in **app.js**. A lot it is still unfamiliar, but you can understand more than you could before!

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.

**Instructions**

**1.**

Select **new.js**.

Skip line 3. On line 4, declare a new *component class* by writing class x extends React.Component {}.

Don’t put anything between the curly braces just yet!

**Name a Component Class**

Good! Subclassing React.Component is the way to declare a new *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!

This adheres to JavaScript’s class syntax. It also adheres to a broader programming convention in which [class names are written in UpperCamelCase](https://en.wikipedia.org/wiki/Naming_convention_(programming)#Java).

In addition, there is a React-specific reason why component class names must always be capitalized. We’ll get to that soon!

**Instructions**

**1.**

Select **new.js**. Edit your code so that your component class is named MyComponentClass.

class MyComponentClass extends React.Component {}

# Component Class Instructions

Let’s review what you’ve learned so far! Find each of these points in **app.js:**

* 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.
* Whenever you create a component class, you need to give that component class a name. That name should be written in UpperCamelCase. In this case, your chosen name is MyComponentClass.

Something that we haven’t talked about yet is the body of your component class: the pair of curly braces after React.Component, and all of the code between those curly braces.

Like all JavaScript classes, this one needs a body. 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.

# The Render Function

A component class is like a factory that builds components. It builds these components by consulting a set of instructions, which you must provide. Let’s talk about these instructions!

For starters, these instructions should take the form of a class declaration body. That means that they will be delimited by curly braces, like this:

class ComponentFactory extends React.Component {

// instructions go here, between the curly braces

}

The instructions should be written in typical JavaScript [ES2015 class syntax](http://exploringjs.com/es6/ch_classes.html).

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 return statement. Usually, this return statement returns a JSX expression:

class ComponentFactory extends React.Component {

render() {

return <h1>Hello world</h1>;

}

}

Of course, none of this explains the point of a render method. All you know so far is that its name is render, it needs a return statement for some reason, and you have to include it in the body of your component class declaration. We’ll get to the ‘why’ of it soon!

**Instructions**

**1.**

Select **new.js**.

Place the cursor in between the curly braces at the end of line 4, and hit return. Lines 4 through 6 should now look like this:

class MyComponentClass extends React.Component {

}

On line 5, write a render method. For now, make the function’s body empty:

render() {}

**2.**

Now let’s fill out that render method.

Inside of the render method’s body, write a return statement that returns the JSX expression <h1>Hello world</h1>.

import React from 'react'

import ReactDOM from 'react-dom'

class MyComponentClass extends React.Component {

  render() {

    return <h1>Hello world</h1>

  }

}

# 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.”

**Instructions**

**1.**

At the bottom of **new.js**, on line 11, create an instance of MyComponentClass.

mport React from 'react';

import ReactDOM from 'react-dom';

class MyComponentClass extends React.Component {

  render() {

    return <h1>Hello world</h1>;

  }

}

// component goes here:

<MyComponentClass />

# Render A Component

You have learned that a component class needs a set of instructions, which tell the component class how to build components. When you make a new component class, these instructions are the body of your class declaration:

class MyComponentClass extends React.Component

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

render() {

return <h1>Hello world</h1>;

}

}

This class declaration results in a new component class, in this case named MyComponentClass. MyComponentClass has one method, named render. This all happens via standard JavaScript class syntax.

You haven’t learned how these instructions actually work to make components! When you make a component by using the expression <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.

This lesson’s final exercise is to render your component. In order to render a component, that component needs to have a method named render. Your component has this! It inherited a method named render from MyComponentClass.

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.

**Instructions**

**1.**

Select **new.js**.

Render <MyComponentClass />. For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

Oh, hello!

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponentClass extends React.Component {

  render() {

    return <h1>Hello world</h1>;

  }

}

// component goes here:

<MyComponentClass />

ReactDOM.render(

  <MyComponentClass />,

  document.getElementById('app')

);

# 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.

The key thing to notice in QuoteMaker is the parentheses in the return statement, on lines 6 and 18. Until now, your render function return statements have looked like this, without any parentheses:

return <h1>Hello world</h1>;

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

**Instructions**

**1.**

Here’s another quote, formatted in the same way:

<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>

In **app.js**, write a React component that renders this HTML. Render your component using ReactDOM.render().

Use **QuoteMaker.js** as a guide! Remember to import React and ReactDOM at the top of the file, and remember to ReactDOM.render() your component at the bottom of the file.

If you don’t like this quote, feel free to use a different one.

import React from 'react';

import ReactDOM from 'react-dom';

class app 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(

  <app />,

  document.getElementById('app')

);

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} />

      </div>

    )

  }

}

ReactDOM.render(

  <Owl />,

  document.getElementById('app')

);

# Put Logic in a Render Function

A render() function must have a return statement. 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 n declaration 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().

**Instructions**

**1.**

Let’s make a render() function with some logic in it.

On lines 1 and 2, import React and ReactDOM.

**2.**

Starting on line 20, create a new component class named Friend. Remember, the component class declaration syntax is class YourClassName extends React.Component {}

Give your component class the following render function:

render() {

return (

<div>

</div>

);

}

**3.**

Inside the body of the render function, before the return statement, declare a new variable named friend.

Set friend equal to either friends[0], friends[1], or friends[2], depending on which friend sounds most appealing to you.

**4.**

Inside of the return statement, inside of the <div></div>, write a nested <h1></h1>.

Inside of the <h1></h1>, inject friend.title.

**5.**

Still inside of the <div></div>, make a new line after the <h1></h1>.

On the new line, write an <img />.

Give the <img /> an attribute of src={friend.src}.

**6.**

At the bottom of the file, use ReactDOM.render() to render an instance of Friend. Use the example code as a guide.

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[0];

    return (

      <div>

        <h1>{friend.title}</h1>

        <img src={friend.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?

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

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

**Instructions**

**1.**

Select **app.js**. You can see a variable named fiftyFifty.

fiftyFifty will equal true half the time and false half the time.

Starting on line 7, write a new component class named TonightsPlan.

If fiftyFifty is true, then TonightsPlan should render this element:

<h1>Tonight I'm going out WOOO</h1>

If fiftyFifty is false, then TonightsPlan should render this element:

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

Use TodaysPlan as a guide, but you don’t have to stick to it exactly. There are many valid ways to solve this problem using a conditional.

**2.**

Render an instance of TonightsPlan and see what fate has in store.

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) {

      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:

class IceCreamGuy extends React.Component {

get food() {

return 'ice cream';

}

render() {

return <h1>I like {this.food}.</h1>;

}

}

In the code, what does this mean?

Once you have a guess, scroll down to see the answer.

…

…

…

…

…

…

The simple answer is that this refers to an instance of IceCreamGuy. The less simple answer is that this refers to the object on which this‘s enclosing method, in this case .render(), is called. It is almost inevitable that this object will be an instance of IceCreamGuy, 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. IceCreamGuy has two methods: .food and .render(). Since this will evaluate to an instance of IceCreamGuy, this.food will evaluate to a call of IceCreamGuy‘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 .food is a getter method. You can tell this from the get in 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 [understanding this in JavaScript](https://dmitripavlutin.com/gentle-explanation-of-this-in-javascript/).

**Instructions**

**1.**

On line 6, add a getter method to your class body. Your getter method should have a name of name, and a return value of a string:

get name() {

return 'whatever-your-name-is-goes-here';

}

**2.**

Inside of the render function, in between the <h1></h1> tags, add the text My name is \_..

In place of \_, call the this object’s name method. Feel free to use the example code as a guide.

import React from 'react';

import ReactDOM from 'react-dom';

class MyName extends React.Component {

  // name property goes here:

get name() {

  return 'whatever-your-name-is-goes-here';

}

  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 methods on 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>.

**Instructions**

**1.**

In **app.js**, find the <button></button> inside of the render function.

Give this <button></button> an onClick attribute. The attribute’s value should be the .scream() method.

Feel free to use the example code as a guide.

**2.**

At the bottom of the file, render a <Button /> using ReactDOM.render(). For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

Once your component renders, click on the button in the browser. Bone-chilling!

import React from 'react';

import ReactDOM from 'react-dom';

class Button extends React.Component {

  scream() {

    alert(document.domain);

  }

  render() {

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

  }

}

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

# Components Recap

Congratulations! You have finished the unit on React components.

React components are complicated. Their syntax is complicated, and the reasoning behind their syntax is especially complicated.

You have learned a lot about both their syntax and their reasoning. You have learned about component classes and component instances. You have learned about React.Component, and about the instructions that you must provide to a component class. You have learned how to import, and how to render a component instance.

You have been introduced to some common ways of using JSX in React components. You have rendered components using multiline JSX expressions, logic inside of the render function, a conditional statement, this, and an event listener.

You have spent a lot of time studying React components in isolation! Now, it’s time to start learning how components fit into with the world around them.