YOUR FIRST REACT COMPONENT

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**.

Checkpoint 2 Passed

Hint

Carefully copy the example code into **app.js** and click Run.

app.js

```
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')
);
```

Import React

Wooo! Your first React component!

In the last exercise, we started by importing from `react`. The line that did this is:

```
import React from 'react';
```

This creates an object named `React` which contains methods necessary to use 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 this is how we import the React library.

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.

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

Checkpoint 2 Passed

Hint

To import React, you'll write a line like this:

```
import React from 'react';
```

new.js

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

class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>
  }
}
```

Import ReactDOM

In order to create our first component, we next imported the ReactDOM:

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

This line of code is very similar to line 1.

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.

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

Checkpoint 2 Passed

Hint

Your import will look similar to the import of React. But instead of importing from `'react'`, you’ll import from `'react-dom'`, and you’ll save it in a variable called `ReactDOM` instead of `React`.

new.js

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

class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>
  }
}
```

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: we can use a JavaScript class to define a new React component. We can also define components with JavaScript functions, but we’ll focus on *class components* first.

All class components will have some methods and properties in common (more on this later). Rather than rewriting those same properties over and over again every time, we extend the `Component` class from the React library. This way, we can use code that we import from the React library, without having to write it over and over again ourselves.

After we *define* our class component, we can use it to *render* as many instances of that component as we want.

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 2 3 4.

Take another look at the code from the first exercise:

```
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 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.

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!
Checkpoint 2 Passed

Hint

On line 4, write class x extends React.Component {} to create a new component class.

new.js

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

class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>
  }
}
```

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 our finished component, our component class’s name was MyComponentClass:

```
class    MyComponentClass    extends    React.Component    {
  render()                                                 {
    return               <h1>Hello                world</h1>;
  }
}
```

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.

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

Instructions

1.

Edit your code so that your component class is named `MyComponentClass`.

Checkpoint 2 Passed

Hint

Your component class was called `x`, but it should be renamed to `MyComponentClass`.

new.js

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

class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>
  }
}
```

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.

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

app.js

```
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')
);
```

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.

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.

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() {}
```

Checkpoint 2 Passed

Hint

When you’re all done, lines 4 through 6 should look like this:

```
class MyComponentClass extends React.Component {
  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>`.

Checkpoint 3 Passed

Hint

The `render()` method should contain just one line between the curly braces:

```
return <h1>Hello world</h1>;
```

new.js

```
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.

On line 11, create an *instance* of `MyComponentClass`.
Checkpoint 2 Passed

Hint

If you wanted to create an instance of a component called `FooBar`, you'd do something like this:

```
<FooBar />
```

On line 11, you’ll do something similar, but with a different name.

new.js

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

class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
}

// component goes here:
ReactDOM.render(
  <MyComponentClass />,
  document.getElementById('app')
);
```

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.

Use `ReactDOM.render` to render `<MyComponentClass />`. For the second argument, pass in `document.getElementById('app')`.

```
Oh, hello!
Checkpoint 2 Passed
```

Hint

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

new.js

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

class MyComponentClass extends React.Component {
  render() {
    return <h1>Hello world</h1>;
  }
}

// component goes here:
ReactDOM.render(
  <MyComponentClass />,
  document.getElementById('app')
);
```