**Watch First**

We'll be looking at using React's .createElement() method in the next couple of videos. For starters, here is its signature:

React.createElement( */\* type \*/*, */\* props \*/*, */\* content \*/* );

We'll take a deep dive into what all that entails in just a bit! We'll start things out with a project that's already set up. For now, don't worry about creating a project or coding along. There will be plenty of hands-on work for you to do soon enough! We'll start building our in-class project, Contacts App, in the next section. If you would like to code along for the next few videos, you can use this [React Sandbox](https://codesandbox.io/s/new).

***💡 Trying Out React Code 💡***

*React is an extension of JavaScript (i.e., a JavaScript library), but it isn't built into your browser. You wouldn't be able to test out React code samples in your browser console the way you would if you were learning JavaScript. In just a bit, we'll see how to install and use a React environment!*

**Rendering Elements onto the DOM**

In the previous video, we used ReactDOM's render() method to render our element onto a particular area of a page. In particular, we rendered the element onto a DOM node called root. But where did this root come from?

Apps built with React typically have a single root DOM node. For example, an HTML file may contain a <div> with the following:

<div id="root"></div>

By passing this DOM node into getElementById(), React will end up controlling the entirety of its contents. Another way to think about this is that this particular <div> will serve as a "hook" for our React app; this is the area where React will take over and render our UI!

**QUESTION 1 OF 3**

What will myBio hold when the following code is run?

**import** React **from** 'react';

**const** myBio = React.createElement('div', null, 'My name is Tyler, and I love porcupines.');

* 

a reference to a DOM node

* 

a DOM node itself

* a JavaScript object
* 

a JavaScript class

SUBMIT

**QUESTION 2 OF 3**

React allows a lot of HTML attributes to be passed along to the React element. Look through [all supported HTML attributes](https://facebook.github.io/react/docs/dom-elements.html#all-supported-html-attributes) in the React docs and select which of the following attributes are allowed:

* poster
* id
* marginWidth
* 

for

SUBMIT

I just used React's .createElement() method to construct a "React element". The .createElement() method has the following signature:

React.createElement( */\* type \*/*, */\* props \*/*, */\* content \*/* );

Let's break down what each item can be:

* type – either a string or a React Component

This can be a string of any existing HTML element (e.g. 'p', 'span', or 'header') or you could pass a React *component* (we'll be creating components with JSX, in just a moment).

* props – either null or an object

This is an object of HTML attributes and custom data about the element.

* content – null, a string, a React Element, or a React Component

Anything that you pass here will be the content of the rendered element. This can include plain text, JavaScript code, other React elements, etc.

**.createElement() Returns *One* Root Element**

Recall that React.createElement(); creates a single React element of a particular type. We'd normally pass in a tag such as a <div> or a <span> to represent that type, but the content argument can be *another* React element!

Consider the following example:

**const** element = React.createElement('div', null,

React.createElement('strong', null, 'Hello world!')

);

Here, "Hello world!" will be wrapped in a <div> when this React element renders as HTML. While we can indeed nest React elements, remember the overall call just returns a single element.

**QUESTION 3 OF 3**

Consider the following example in JSX:

**const** greeting = (

<div className='greeting'>

<h2>Hello world!</h2>

</div>

);

If we want to output the same HTML, what goes into 1, 2, and 3 when calling createElement()?

**const** greeting = React.createElement(

\_\_1\_\_,

{ className: 'greeting' },

React.createElement(

\_\_2\_\_,

{},

\_\_3\_\_

)

);

* 

'h2', 'div', 'Hello world!'

* 

div, h2, Hello world!

* 'div', 'h2', 'Hello world!'
* 

Hello world!, div, h2

SUBMIT

**JSX Returns *One* Main Element, Too**

When writing JSX, keep in mind that it must only return a single element. This element may have any number of descendants, but there *must* be a single root element wrapping your overall JSX (typically a <div> or a <span>). Check out the following example:

**const** message = (

<div>

<h1>All About JSX:</h1>

<ul>

<li>JSX</li>

<li>is</li>

<li>awesome!</li>

</ul>

</div>

);

See how there's only one <div> element in the code above and that all other JSX is nested inside it? This is how you have to write it if you want multiple elements. To be completely clear, the following is incorrect and will cause an error:

**const** message = (

<h1>All About JSX:</h1>

<ul>

<li>JSX</li>

<li>is</li>

<li>awesome!</li>

</ul>

);

In this example, we have two sibling elements that are both at the root level (i.e. <h1> and <ul>) . This won't work and will give the error:

*Syntax error: Adjacent JSX elements must be wrapped in an enclosing tag*

Since we know that JSX is really just a syntax extension for .createElement(), this makes sense; .createElement() takes in only one tag name (as a string) as its first argument.

**Intro to Components**

So far we've seen how .createElement() and JSX can help us produce some HTML. Typically, though, we'll use one of React's key features, Components, to construct our UI. Components refer to *reusable* pieces of code ultimately responsible for returning HTML to be rendered onto the page. More often than not, you'll see React components written with JSX.

Since React's main focus is to streamline building our app's UI, there is only one method that is absolutely required in any React component class: render().

Let's go ahead and build our first component class!

***💡 Declaring Components in React 💡***

*In the previous video, we defined the ContactList component like so:*

***class******ContactList******extends******React****.****Component*** *{*

*// ...*

*}*

*In other words, we are defining a component that's really just a JavaScript class that inherits from React.Component.*

*In real-world use (and throughout this course), you may also see declarations like:*

***class******ContactList******extends******Component*** *{*

*// ...*

*}*

*Both ways are functionally the same, but be sure your module imports match accordingly! That is, if you choose to declare components like the example directly above, your import from React will now look like:*

***import*** *React, { Component }* ***from*** *'react';*

**Creating Elements Recap**

In the end, remember that React is only concerned with the View layer of our app. This is what the user sees and interacts with. As such, we can use .createElement() to render HTML onto a document. More often than not, however, you'll use a syntax extension to describe what your UI should look like. This syntax extension is known as JSX, and just looks similar to plain HTML written right into a JavaScript file. The JSX gets transpiled to React's .createElement() method that outputs HTML to be rendered in the browser.

A great mindset to have when building React apps is to [think in components](https://facebook.github.io/react/docs/thinking-in-react.html). Components represent the modularity and reusability of React. You can think of your component classes as factories that produce instances of components. These component classes should follow the [single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle) and just "do one thing". If it manages too many different tasks, it may be a good idea to decompose your component into smaller subcomponents.

Further Research:

* [Rendering Elements](https://facebook.github.io/react/docs/rendering-elements.html) from the React docs