JSX is a syntax extension for JavaScript. It was written to be used with React. JSX code looks a lot like HTML. In this case, it means that JSX is not valid JavaScript. Web browsers can't read it!

JavaScript is case-sensitive

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.

A basic unit of JSX is called a JSX *element*.

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>

};

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"

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" />;

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

<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>

)

A JSX expression must have exactly *one* outermost element.

The first opening tag and the final closing tag of a JSX expression must belong to the same JSX element!

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

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.

ReactDOM.render() is the most common way to *render* JSX. That is the way to make a JSX expression appear onscreen.

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

ReactDOM.render()'s first argument should be a JSX expression, and it will be rendered to the screen. The first argument is *appended* to whatever element is selected by the *second* argument.

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 variablenamed 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') );

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.

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.

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>

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 />

<img src="images/jenkins.png" />

NOT FINE AT ALL in JSX: <br>

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.

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

Everything inside of the curly braces will be treated as regular JavaScript.

The curly braces themselves won't be treated as JSX nor as JavaScript. They are *markers* that signal the beginning and end of a JavaScript injection into 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.

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

// 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.

JSX elements can have *event listeners*, just like HTML elements can.

You create an event listener by giving a JSX element a special *attribute*.

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.

An event listener attribute's *value* should be a function. The 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.

you can not inject an if statement into a JSX expression.

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

The if statement is on the outside, and no JavaScript injection is necessary.

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

img = <img src={pics.kitty} />;

} else {

img = <img src={pics.doggy} />;

}

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

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

const headline = ( <h1> { age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff' } </h1> );

We're going to cover one final way of writing conditionals in React: the && operator.

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.

If you want to create a list of JSX elements, .map() is often your best bet.

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.

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.

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

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

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

);

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

You can write React code without using JSX at all!

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" );

A React component is a small, reusable chunk of code that is responsible for one job, which often involves rendering HTML.

Every component must come from a component class.

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

// create a variable named React:

import React from 'react';

// evaluate this variable and get a particular, imported JavaScript object: React // { imported object properties here... }

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.

import ReactDOM from 'react-dom';

The methods imported from 'react-dom' are meant for interacting with the DOM. 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.

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.

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 {}.

Component class variable names must begin with capital letters!

class MyComponentClass extends React.Component {

// everything in between curly-braces is instructions 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. A render method must contain a return statement.

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

class Crazy extends React.Component { render() { return <OMG />; } }

In the above example, Crazy's render method returns an instance of the OMG component class.

Usually, this return statement returns a JSX expression:

<MyComponentClass />

To make a React component, you write a JSX element, give it the same name as a component class. Voilà, there's your component instance!

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.

JSX elements can be either HTML-like, or component instances. JSX uses capitalization to distinguish between the two!

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') );

<MyComponentClass /> will call its render method, which will return the JSX element <h1>Hello world</h1>.

However, a multi-line JSX expression should always be wrapped in parentheses!





A render() function can put simple calculations that need to happen right before a component renders.

class Random extends React.Component {

render() {

// First, some logic that must happen // before rendering:

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

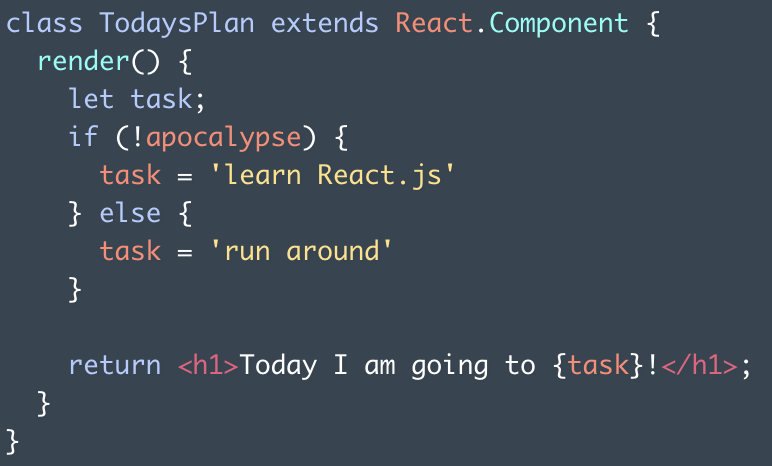
// Next, a return statement // using that logic:

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

}

}

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.



this refers to the object on which this's enclosing method, in this case .render(), is called.

class IceCreamGuy extends React.Component {

get food() { return 'ice cream'; }

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

}

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.

Recall that an event handler is a function that gets called in response to an event. 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> ); }

}

.myFunc() will be called any time that a user hovers over the rendered <div></div>.

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

When you use React.js, every JavaScript file in your application is invisible to every other JavaScript file by default.

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.

import { NavBar } from './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. importwill 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';

You also need an export statement, written in the *other* file, exporting the variable that you hope to grab.

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.

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.

render() {

console.log(this.props);

return <h1>Hello world</h1>;

}

You can pass information to a React component by giving that component an attribute:

<MyComponent foo="bar" />

If you want to pass information that isn't a string, then wrap that information in curly braces.

<Greeting myInfo={["top", "secret", "lol"]} />

Here's how to make a component display passed-in information:

1 - Find the component class that is going to receive that information.  
2 - Include this.props.name-of-informationin that component class's render method's return statement.

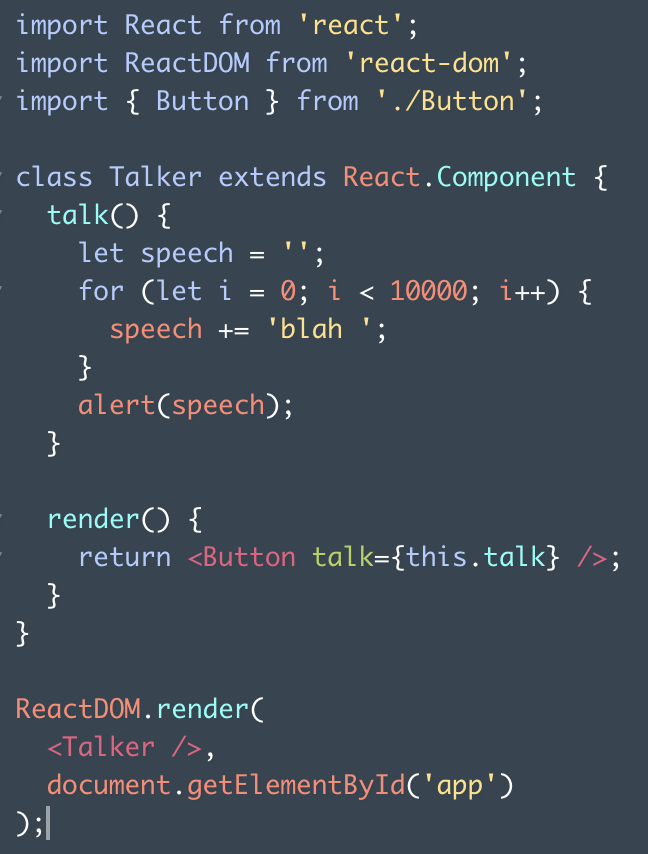


props is the name of the object that stores passed-in information. this.props refers to that storage object. At the same time, each piece of passed-in information is called a prop.

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."

You can, and often will, pass functions as props. However, you have to define an event handler before you can pass one anywhere. You can pass a method in the exact same way that you pass any other information.

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.





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.

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.

These two names can be whatever you want. However, there is a naming convention that they often follow.

Here's how the naming convention works: first, think about what type of event you are listening for. If you are listening for a "click" event, then you name your event handler handleClick. Your prop name should be the word on, plus your event type. If you are listening for a "click" event, then you name your prop onClick.

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} />; }

}

Every component's props object has a property named children.

this.props.children would return everything in between <MyComponentClass> and </MyComponentClass>.

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.

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:

The defaultProps property should be equal to an object. 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.

In the ReactDOM.render() call, give <Button /> the following attribute:

text=""

Your new prop should override the default, making the <button></button> blank again.

A React component can access dynamic information in two ways: props and state.

Unlike props, a component's state is notpassed 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 TodayImFeeling extends React.Component {

constructor(props) {

super(props);

this.state = { mood: 'decent' };

}

render() { return ( <h1> I'm feeling {this.state.mood}! </h1> ); }

}

<Example />

Methods should never be comma-separated, if inside of a class body.

this.state should be equal to an object, like in the example above. This object represents the initial "state" of any component instance.

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' }.

To read a component's state, use the expression

this.state.name-of-property:

Just like this.props, you can use this.state from any property defined inside of a component class's body.

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.

If there are properties in the current state that aren't part of that object, then those properties remain how they were.

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);

This line is necessary because makeSomeFog()'s body contains the word this. 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.

Any time that you call this.setState(), this.setState() AUTOMATICALLY calls .render() as soon as the state has changed.

Think of this.setState() as actually being two things: this.setState(), immediately followed by .render().

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.

