**Hello World**

Take a look at the following line of code:

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

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

It seems like it must be JavaScript, 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.

What’s going on?

**Instructions**

**1.**

Copy the following line into **app.js**:

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

Click Run when you’re finished.

# The Mystery, Revealed

Good!

Take another look at the line of code that you wrote.

Does this code belong in a JavaScript file, an HTML file, or somewhere else?

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.

Click Next to learn about 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?

In this case, it means that JSX is not valid JavaScript. Web browsers can’t read it!

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.

Codecademy’s servers already have a JSX compiler installed, so you don’t have to worry about that for now. Eventually we’ll walk through how to set up a JSX compiler on your personal computer.

**Instructions**

Click Next to continue.

**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.

**Instructions**

**1.**

In **app.js**, write a JSX <p></p> element containing the text, Hello world. Use the example code above as a guide.

<p>Hello world</p>

**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>

};

**Instructions**

**1.**

Create a JSX <article></article> element. Save it in a variable named myArticle.

const myArticle = <article>abc</article>

**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" />;

**Instructions**

**1.**

Declare a constant named p1.

Set p1 equal to a JSX <p></p> element. Write the word foo in between the <p></p> tags.

**2.**

On the next line, declare a constant named p2.

Set p2 equal to another JSX <p></p> element. Write the word bar in between the <p></p> tags.

**3.**

Give the first <p></p> an id attribute of "large".

Give the second <p></p> an id attribute of "small".

const p1 = <p id ="large">foo </p>

const p2 = <p id ="small">bar </p>

**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, *nested* inside 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. This looks strange at first, but you get used to it:

(

<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>

);

**Instructions**

**1.**

Declare a new variable named myDiv. Set myDiv equal to a JSX <div></div> element.

Wrap the <div></div> in parentheses, and use indentation and line breaks like in the examples. In between the <div></div> tags, *nest* an <h1></h1> containing the text Hello world.

const myDiv = <div>

<h1>Hello world</h1>

</div>

**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>

);

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

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>.

**Instructions**

**1.**

Your friend’s blog is down! He’s asked you to fix it.

You immediately diagnose the problem: a JSX expression with multiple outer elements.

Repair your friend’s broken code by wrapping their JSX in a <div></div>.

const blog = (

  <div>

  <img src="pics/192940u73.jpg" />

  <h1>

    Welcome to Dan's Blog!

  </h1>

  <article>

    Wow I had the tastiest sandwich today.  I <strong>literally</strong> almost freaked out.

  </article>

  </div>

);

**Rendering JSX**

You’ve learned how to write JSX elements! Now it’s time to learn how to *render* them.

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

**Instructions**

**1.**

The following code will render a JSX expression:

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

Starting on line 5, carefully copy the code into the code editor. We’ll go over how it works in the next exercise.

JavaScript is case-sensitive, so make sure to capitalize ReactDOM correctly!

import React from 'react';

import ReactDOM from 'react-dom';

// Copy code here:

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

**ReactDOM.render() I**

Let’s examine the code that you just wrote. Start in **previous.js**, on line 5, all the way to the left.

You can see something called ReactDOM. What’s that?

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.

We’ll talk more later about how ReactDOM got into your file. For now, just understand that it’s yours to use.

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.

We’ll discuss the second argument in the next exercise!

**Instructions**

**1.**

Select **app.js**.

Starting on line 5, call ReactDOM.render().

Pass in this expression as a first argument:

<h1>Render me!</h1>

Pass in this expression as a second argument:

document.getElementById('app')

import React from 'react';

import ReactDOM from 'react-dom';

// Write code here:

ReactDOM.render(<h1>Render me!</h1>, document.getElementById('app'));

**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>

**Instructions**

**1.**

In **index.html**, replace this:

<main id="app"></main>

with this span:

<span id="container"></span>

**2.**

Select **app.js**.

You want <h1>Render me!</h1> to be appended to <span id="container"></span>.

On line 5, make that happen by changing the string passed to document.getElementById.

ReactDOM.render(<h1>Render me!</h1>, document.getElementById('container'));

**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')

);

**Instructions**

**1.**

On line 5, declare a variable named myList. Set myList equal to a JSX <ul></ul> element. Wrap your <ul></ul> in parentheses.

Add several <li></li> elements in between your <ul></ul> tags. Put some text in each <li></li>. Use line breaks and indentation similar to the above example.

**2.**

At the bottom of the file, call ReactDOM.render().

For ReactDOM.render()‘s first argument, pass in the variable myList.

For ReactDOM.render()‘s second argument, select an HTML element with an id of app.

Feel free to use the example code as a guide.

const myList = (

  <ul>

    <li>Learn React</li>

    <li>Become a Developer</li>

  </ul>

);

ReactDOM.render(

  myList,

  document.getElementById('app')

);

# 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, [read this article about the Virtual DOM](https://www.codecademy.com/articles/react-virtual-dom).

**class vs className**

This lesson will cover more advanced JSX. You’ll learn some powerful tricks, and some common errors to avoid.

Grammar in JSX is mostly the same as in HTML, but there are subtle differences to watch out for. Probably the most frequent of these involves the word class.

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.

**Instructions**

**1.**

On line 5, declare a new variable named myDiv. Set myDiv equal to a JSX <div></div> element.

In between the <div></div> tags, write the text I AM A BIG DIV.

Give your <div></div> the following attribute:

className="big"

**2.**

Underneath your <div></div>, call ReactDOM.render.

For ReactDOM.render()‘s first argument, pass in myDiv.

For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

If your rendered <div></div> has a *class* of "big", then it should look big in the browser!

const myDiv =

<div className="big">

I AM A BIG DIV

</div>

ReactDOM.render(myDiv,document.getElementById('app'))

**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>

**Instructions**

**1.**

In **app.js**, fix the broken JSX by adding slashes to all of the self-closing tags.

Hint

Close the <img> and <br> tags.

const profile = (

  <div>

    <h1>I AM JENKINS</h1>

    <img src="images/jenkins.png" />

    <article>

      I LIKE TO SIT

      <br/>

      JENKINS IS MY NAME

      <br/>

      THANKS HA LOT

    </article>

  </div>

);

**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.

In this lesson, we’re going to add something new: regular JavaScript, written inside of a JSX expression, written inside of a JavaScript file.

Whoaaaa…

**Instructions**

**1.**

Starting on line 5, carefully write the following code. What do you think will appear in the browser?

ReactDOM.render(

<h1>2 + 3</h1>,

document.getElementById('app')

);

ReactDOM.render(

  <h1>2 + 3</h1>,

  document.getElementById('app')

);

**Curly Braces in JSX**

The code in the last exercise didn’t behave as one might expect. Instead of adding 2 and 3, it printed out “2 + 3” as a string of text. Why?

This happened because 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*.

**Instructions**

**1.**

Add a pair of curly braces to the code from last exercise, so that your JSX expression looks like this:

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

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

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.

In the code editor, you can see a JSX expression that displays the first twenty digits of pi.

Study the expression and notice the following:

* The code is written in a JavaScript file. By default, it will all be treated as regular JavaScript.
* Find <div> on line 5. From there up through </div>, the code will be treated as JSX.
* Find Math. From there up through (20), the code will be treated as regular JavaScript again.
* 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, similar to the quotation marks that signal the boundaries of a string.

**Instructions**

**1.**

Select **app.js**.

Declare a new variable named math. Set math equal to a JSX <h1></h1> element.

Put the following text inside of the <h1>:

2 + 3 = 2 + 3

**2.**

At the bottom of the file, call ReactDOM.render().

For ReactDOM.render()‘s first argument, pass in math.

For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

**3.**

As you probably expected, the equation was displayed as a string.

Insert a pair of curly braces into the <h1></h1>, so that the browser displays 2 + 3 = 5.

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>;

**Instructions**

**1.**

Replace ReactDOM.render()‘s first argument with a JSX <h1></h1>.

Using curly braces, set the <h1></h1>‘s inner text equal to theBestString.

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",

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" />

);

**Instructions**

**1.**

On line 7, declare a new variable named gooseImg. Set its value equal to a JSX <img /> element.

Give the <img /> an attribute with a name of src. Set the attribute’s value equal to the variable goose.

**2.**

Use ReactDOM.render() to render gooseImg.

ReactDOM.render()‘s first argument should be gooseImg.

ReactDOM.render()‘s second argument should be document.getElementById('app').

// Declare new variable here:

const gooseImg = <img src={goose} />

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

# 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 [here](http://facebook.github.io/react/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.

**Instructions**

**1.**

Look at line 17. ReactDOM.render() is being passed two null arguments.

Render kitty by replacing the first null with kitty, and the second null with document.getElementById('app').

**2.**

Add an onClick attribute to the <img /> element. Set onClick‘s value equal to the makeDoggy function.

Remember, since attributes are a part of JSX expressions, you will need to inject JavaScript in order to use makeDoggy.

Click Run, and then click on the browser image to change the kitty into a doggy.

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

Great work! You’ve learned how to use curly braces to inject JavaScript into a JSX expression!

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 [here](http://facebook.github.io/react/tips/if-else-in-JSX.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. In the next few lessons, we’ll explore some simple ways to write conditionals (expressions that are only executed under certain conditions) in JSX.

# JSX Conditionals: If Statements That Do Work

How can you write a conditional, if you can’t inject an if statement into JSX?

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

This is a common way to express conditionals in JSX.

**Instructions**

**1.**

Select **app.js**.

Starting on line 16, write an if/else statement, using **if.js** as a guide.

If (coinToss() === 'heads'), then execute img = <img src={pics.kitty} />.

Inside of the else clause, execute img = <img src={pics.doggy} />.

In other words, if the coin lands heads then you should get a picture of a kitty, and if the coin lands tails then you should get a picture of a doggy.

Hint

Remember: semi-colons are used in JavaScript, but not within JSX expressions!

// like this

img = (

<img src={pics.kitty} />

);

/// NOT like this

img = (

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

)

**2.**

At the bottom of the file, call ReactDOM.render().

For ReactDOM.render()‘s first argument, pass in img.

For ReactDOM.render()‘s second argument, pass in document.getElementById('app').

Click Run. Refresh the browser several times. Does the picture change?

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. [Here’s](http://stackoverflow.com/questions/6259982/how-to-use-the-ternary-operator-in-javascript) a nice explanation if you need a refresher.

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>.

**Instructions**

**1.**

Take a look at **app.js**.

On line 14, replace x, y, and z with the following three expressions. You have to decide which is x, which is y, and which is z:

'doggy'

'kitty'

coinToss() === 'heads'

'heads' should return 'kitty', and 'tails' should return 'doggy'.

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: &&

We’re going to cover one 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.

**Instructions**

**1.**

You’ve been building a React website all about your favorite foods!

You’re excited to share your website with your friends, and yet at the same time, you fear the cruel, icy harshness of their judgment.

On line 13, use the && operator to make it so that this expression:

<li>Nacho Cheez Straight Out The Jar</li>

… will only appear if !judgmental. Feel free to use the example code as a guide.

Once you click Run, then every time that you refresh the browser, there will be a 50% chance that judgmental will be true. Refresh until you see both versions of your list.

import React from 'react';

import ReactDOM from 'react-dom';

// judgmental will be true half the time.

const judgmental = Math.random() < 0.5;

const favoriteFoods = (

  <div>

    <h1>My Favorite Foods</h1>

    <ul>

      <li>Sushi Burrito</li>

      <li>Rhubarb Pie</li>

      {!judgmental && <li>Nacho Cheez Straight Out The Jar</li>}

      <li>Broiled Grapefruit</li>

    </ul>

  </div>

);

ReactDOM.render(

  favoriteFoods,

  document.getElementById('app')

);

# .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>

**Instructions**

**1.**

You can see that a .map() call is partially set up.

On line 8, write an expression to complete the .map() call. This expression should consist of an <li></li>, containing the person parameter. Feel free to use the first example as a guide.

**2.**

On line 12, call ReactDOM.render().

For ReactDOM.render()‘s first argument, write a <ul></ul>. In between the <ul> tags, use curly braces to inject the peopleLis variable.

For ReactDOM.render()‘s second argument, use document.getElementById('app').

import React from 'react';

import ReactDOM from 'react-dom';

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

const peopleLis = people.map(person =>

  // expression goes here:

<li>{person}</li>

);

// ReactDOM.render goes here:

ReactDOM.render(<ul>{peopleLis}</ul>, 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!

**Instructions**

**1.**

On line 8, give your <li></li> a key attribute.

What should key‘s *value* be? In this case, the answer is surprisingly tricky.

.map() is going to loop over your expression on line 8, creating a new <li></li> with a new key attribute on each loop. You need every key to be unique! How can you get .map() to produce a unique key, each time it loops?

First, add an i parameter to .map()‘s inner function, so that you can access each person’s unique index:

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

Now, you can get a unique key on each loop, by adding the following attribute to your <li></li>:

key={'person\_' + i}

import React from 'react';

import ReactDOM from 'react-dom';

const people = ['Rowe', 'Prevost', '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 [documentation](http://facebook.github.io/react/docs/top-level-api.html#react.createelement) if you’d like to learn more!

**Instructions**

**1.**

Take a look at the following JSX element:

const greatestDivEver = <div>i am div</div>;

In **app.js**, create the element above without using JSX. Use the example as a guide.

const greatestDivEver = React.createElement("div",null,"i am div");