**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.

# 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>

# 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...

# 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.

# 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.

# 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>;

# 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" /> );

# 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.

# 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 ifstatement is on the outside, and no JavaScript injection is necessary.

This is a common way to express conditionals in JSX.

# 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>.

# 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.

# .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>

# 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!

# 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!

# JSX Recap

Congratulations! You have completed the unit on JSX.

You have learned a wide variety of JSX concepts. If you don't feel like you've mastered them all yet, that's okay! These concepts will come up again and again throughout this course, and the following courses.

You are now ready to put your JSX knowledge to use! It's time to move on to the next major topic: React Components.