ADVANCED JSX

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"
```

Checkpoint 2 Passed

Stuck? Get a hint

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 br

app.js

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

// Write code here:
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.

Checkpoint 2 Passed

Hint

Close the `<img>` and `<br>` tags.

app.js

```
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')
);
```

Checkpoint 2 Passed

Hint

Carefully copy-paste this code, starting on line 5.

```
ReactDOM.render(
  <h1>2                                                    + 3</h1>,
  document.getElementById('app')
);
```

app.js

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

```
// Write code here:
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.

Checkpoint 2 Passed

Hint

Wrap `2 + 3` in curly braces so that it becomes `{2 + 3}`.

app.js

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

// Write code here:
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
```

Checkpoint 2 Passed

Hint

Declare a variable named `math` and set it to `<h1>2 + 3 = 2 + 3</h1>`.

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')`.

Checkpoint 3 Passed

Hint

Call `ReactDOM.render()` with two arguments: `math` and `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`.

Checkpoint 4 Passed

Hint

Find the code that says `2 + 3 = 2 + 3` and wrap the right side in curly braces so that it becomes `2 + 3 = {2 + 3}`.

pi.js

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

const pi = (
  <div>
    <h1>
      PI, YALL!
    </h1>
    <p>
      {Math.PI.toFixed(20)}
    </p>
  </div>
);
```

```
ReactDOM.render(
  pi,
  document.getElementById('app')
);
```

app.js

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

// Write code here:
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`.

Checkpoint 2 Passed

Hint

`ReactDOM.render()`'s first argument should be `<h1>{theBestString}</h1>`.

app.js

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

const theBestString = 'tralalalala i am da best';

ReactDOM.render(<h1>{theBestString}</h1>, document.getElemen
tById('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`.

Checkpoint 2 Passed

Hint

Declare a variable named `gooseImg` and set it to `<img src={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')`.

Checkpoint 3 Passed

Hint

Call `ReactDOM.render()` with two arguments: `gooseImg` and `document.getElementById('app')`.

app.js

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

const goose = 'https://content.codecademy.com/courses/React/react_photo-goose.jpg';

// 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.

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')`.

Checkpoint 2 Passed

Hint

Update the call to `ReactDOM.render()` with two arguments: `kitty` and `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.

Checkpoint 3 Passed

Hint

The `<img />` element already has two attributes: `src` and `alt`. You'll need to add a third one: `onClick={makeDoggy}`.

app.js

```
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://content.codecademy.com/courses/React/react_photo-puppy.jpeg');
  e.target.setAttribute('alt', 'doggy');
}

const kitty = (
  <img
    src="https://content.codecademy.com/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.

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.

Checkpoint 2 Passed

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?

Checkpoint 3 Passed

Hint

Call `ReactDOM.render()` with two arguments: `img` and `document.getElementById('app')`.

If.js

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

let message;

if (user.age >= drinkingAge) {
  message = (
    <h1>
```

```
      Hey, check out this alcoholic beverage!
    </h1>
  );
} else {
  message = (
    <h1>
      Hey, check out these earrings I got at Claire's!
    </h1>
  );
}

ReactDOM.render(
  message,
  document.getElementById('app')
);
```

app.js

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

function coinToss() {
  // This function will randomly return either 'heads' or 't
ails'.
  return Math.random() < 0.5 ? 'heads' : 'tails';
}

const pics = {
  kitty: 'https://content.codecademy.com/courses/React/react
_photo-kitty.jpg',
  doggy: 'https://content.codecademy.com/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 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'`.

Checkpoint 2 Passed

Hint

`x` should be replaced with `coinToss() === 'heads'`. What about `y` and `z`?

app.js

```
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://content.codecademy.com/courses/React/react_photo-kitty.jpg',
  doggy: 'https://content.codecademy.com/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>
);
```

If the expression on the left of the `&&` evaluates as true, then the JSX on the right of the `&&` will be rendered. If the first expression is false, however, then the JSX to the right of the `&&` will be ignored and not rendered.

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.

Checkpoint 2 Passed

Hint

If you wanted to display that list element if `judgmental` was `true`, you'd write something that looks like this:

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

Your solution will look similar.

favorite-foods.js

```
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.

Checkpoint 2 Passed

Hint

Complete the call to `.map()` by adding an `<li>` element on line 8. It should contain `{person}`.

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')`.

Checkpoint 3 Passed

Hint

Call `ReactDOM.render()` with two arguments: `<ul>{peopleLis}</ul>` and `document.getElementById('app')`.

app.js

```
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'));
```

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?

Each `key` must be a unique string that React can use to correctly pair each rendered element with its corresponding item in the array.

So, for each element in `people`, we must generate a unique value. How can you get `.map()` to produce unique keys?

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}
```

Checkpoint 2 Passed

Hint

Start by adding an `i` parameter to `.map()`'s inner function, so that you can access each person's unique index: It will look like this:

```
people.map((person, i) =>
```

Now, add a `key` attribute to each `<li>`, like this:

```
<li key={'person_' + i}>
```

app.js

```
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 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.

Checkpoint 2 Passed

Hint

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"
);
```

Your solution will look similar.

app.js

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

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.