Writing JSX

(https://github.com/learn-co-curriculum/react-hooks-jsx) (https://github.com/learn-co-curriculum/react-hooks-jsx/issues/new)

Learning Goals

- Understand what JSX is used for
- Identify JSX syntax
- Understand what is required to write valid JSX code

What is JSX?

JSX allows us to write HTML-like code in our JavaScript files. JSX is a syntax extension of JavaScript that creates a very special and extremely productive marriage between HTML and JS. It's short for **JavaScript XML** (https://facebook.github.io/jsx/), and was created by Facebook to work hand-in-hand with React.

With JSX, we can instruct React to create DOM elements in JavaScript in an efficient and expressive manner. Ultimately, JSX looks a lot like the end result we see in the browser (i.e. HTML), and is *much* faster to write compared to creating DOM elements using something like `document.createElement()`, especially when incorporating a lot of JavaScript and dynamic content.

Imperative vs Declarative Programming

JSX uses what is referred to as a *declarative* style of programming, whereas creating DOM elements using "vanilla" JavaScript methods like `document.createElement` would be considered *imperative*.

To write imperative code is to write code that describes *how* something is done in detail. To write declarative code is to write *what* you would like to do.

To further explain, imagine you walk into a local restaurant with the intention of ordering a sandwich.

- To order **declaratively** would be to say something like this: "I would like one toasted ham and cheese sandwich." Rather than worry about the details, you are just saying *what* you want and letting the restaurant staff handle the details.
- To order **imperatively** would mean saying something like this instead: "I would like you to take three slices of ham, two slices of cheese and a jar of mayonnaise from the refrigerator and place them together on a clean counter. Please also find two slices of bread. Stack the ham, cheese and bread in this order: bread, ham, ham, ham, cheese, cheese, bread. Remove the top slice of bread and apply a dollop of mayo. Replace top slice of bread and place completed sandwich in oven at 300 degrees for five minutes, then bring it to me on a plate."

In general (and to the relief of restaurant staff everywhere), we prefer the declarative approach when speaking unless we are specifically instructing someone else.

Creating DOM elements using "vanilla" JavaScript is considered **imperative** because we write code for each step explicitly. In plain JavaScript, to render a `div` element on the page, we might end up writing something like:

```
const div = document.createElement("div");
div.id = "card1";
div.className = "card";
div.textContent = "hello world";
document.body.appendChild(div);
```

Five distinct steps are used here. With JSX, however, we just need to write *what* we want, and allow React to figure things out behind the scenes:

```
const div = (
  <div id="card1" className="card">
    hello world
  </div>
);

ReactDOM.render(div, document.body);
```

Both of the examples above will produce a `<div>` element with the appropriate class, id, and text content, but the JSX example is significantly cleaner.

How does this work under the hood? The JSX code above isn't valid JavaScript (JSX is a *syntax extension*, not part of the language), so an intermediate step needs to happen before React can run our code.

When we run this code through Babel (used with React), Babel sees this and understands it to be JSX, **not HTML**. Babel can then **transpile** this code into valid JavaScript, which instructs React how to create this element:

```
const div = React.createElement(
  "div",
  {
    id: "card1",
    className: "card",
  },
  "hello world"
);

ReactDOM.render(div, document.body);
```

React's `createElement` method then takes the parameters provided and creates the actual DOM elements, using similar vanilla JavaScript methods like `document.createElement` under the hood.

While the exact details of how it creates the DOM element differ from traditional DOM manipulation, the end result is the same: a `div` element added to the page with the text 'hello world' inside.

> If you're ever curious about what's being output by Babel after it transpiles our JSX, try pasting your code into the **Babel REPL (https://babeljs.io/en/repl)** to see what the transpiled JavaScript looks like!

What JSX Looks Like

React components return JSX:

```
function Tweet() {
  const currentTime = new Date().toString();
```

```
  // this returns JSX!
  return (
    <div className="tweet">
      <img src="http://twitter.com/some-avatar.png" className="tweet__avatar" />
      <div className="tweet__body">
        <p>We are writing this tweet in JSX. Holy moly!</p>
        <p>{Math.floor(Math.random() * 100)} retweets </p>
        <p>{currentTime}</p>
      </div>
    </div>
  );
}
```

Whoa, isn't this interesting? It's HTML, but in our JavaScript... with JavaScript *inside the HTML!* Looking at this code, there are some important things to point out:

JSX is *not* a String

The JSX in the example is not wrapped in quotes. Think of it as another type in JavaScript. **We are not interpolating HTML strings** like we do with `innerHTML`.

JSX is the Return Value of a Function Component

A function component **must return JSX**.

Every function component you use needs to return *one* JSX element. Although our example displays eight lines of JSX, this is done for readability only. The entire return statement is wrapped in parentheses so it is considered one 'chunk' of JSX code, with *one* top level element:

```
return <div className="tweet">{/*child elements in here*/}</div>;
```

JSX Can Include JavaScript

While writing our pseudo-HTML in JSX, we can also write vanilla JavaScript *in-line*. We do this by wrapping the JavaScript code in **curly braces**.

```
<p>{ Math.floor(Math.random()*100) } retweets</p>
<p>{ currentTime }</p>
```

In the example, we call the `Math.floor()` and `Math.random()` methods directly, which will return a random number when the component is rendered.

We *also* used a variable, `currentTime`, which holds the String value of the current date and time. In our example, `currentTime` is a variable within the `Tweet` component. It's common to write variables inside React components in order to organize functionality based on a component's responsibility.

This is similar to how we use curly braces when interpolating variables into strings:

```
const myString = `The temperature today is ${getTemperature()}`;
```

In both JSX and string interpolation, curly braces are an "escape hatch" in the syntax. Curly braces allow us to use variables/functions within the JSX to make our templates dynamic.

Keep this in mind: Any time you want to use JavaScript variables or call functions from within a JSX element, **you must use curly braces** like we did in the example above.

JSX Works With Expressions, Not Statements

JSX is an extension of JavaScript, wrapping a lot of underlying function calls in a syntactically appealing style. This is why JSX code is considered *declarative*. When *we're* writing in JSX, it is equivalent to saying "Make an h1 element with this content inside" and letting React work on the element creation and function calls. Because we follow the proper syntax, React knows that when we write:

```
<h1 id="header">Hello!</h1>
```

Babel must convert this JSX into regular, imperative JavaScript before React renders the component:

```
React.createElement("h1", { id: "header" }, "Hello!");
```

Which is then committed to the actual DOM as an `h1` DOM node. We never need to see this — all *we* write is the JSX:

```
<h1 id="header">Hello!</h1>
```

In order for our JSX to be converted into regular JavaScript, the JSX we write **must be in the form of an expression, not a statement (https://2ality.com/2012/09/expressions-vs-statements.html)**. For instance, the following `if` **statement** is not valid in JSX:

```
<h1 id="header">{if (true) {
  "Hello"
} else {
  "Goodbye"
}}</h1>
```

However, the ternary **expression** does work:

```
<h1 id="header">{true ? "Hello" : "Goodbye"}</h1>
```

The reason for this is that statements don't have a return value, and expressions do.

You can also call functions from within JSX, if you need to express your code with an `if` statement:

```
function Header() {
  function getHeaderText(isHello) {
    if (isHello) {
      return "Hello";
    } else {
      return "Goodbye";
    }
  }
```

```
  return <h1 id="header">{getHeaderText(true)}</h1>;
}
```

A Component Can Render Another Component Using JSX

If we have a component (a function that returns JSX), like this:

```
function Header() {
  return <h1>Hello</h1>;
}
```

We can embed that component inside another component using JSX:

```
function Page() {
  return (
    <div>
      <Header />
      <p>Some great content in here</p>
    </div>
  );
}
```

When we're writing *HTML elements* in JSX, the element names must be all *lowercase*, just like we typically write normal HTML elements. When we're writing *components* in JSX, the name of the component must be *capitalized*. This is how React can differentiate our `<Header>` component from a normal HTML `<header>` element.

A Component Must Return One JSX Element

In all the lesson examples we've seen so far, each component is returning a `div` that contains content or child elements. However, we can actually use any HTML element we would normally use to contain content. The following are all valid components:

```
function PlainDiv() {
  return <div>I am one line, so I do not need the parentheses</div>;
}

const Photo = () => {
  return (
    <figure>
      <img
        className="image"
        src="https://s3.amazonaws.com/ironboard-learn/sunglasses.gif"
      />
      <figcaption>Whoa</figcaption>
    </figure>
  );
};

const Table = () => (
  <table>
    <tr>
      <th>ID</th>
      <th>Name</th>
    </tr>
    <tr>
      <th>312213</th>
      <th>Tim Berners-Lee</th>
    </tr>
  </table>
);

function ParentComponent() {
  return (
    <main>
```

```
        <PlainDiv />
        <Photo />
        <Table />
      </main>
    );
  }
```

Each of these are valid components, but *all* of these components have *one* returned JSX element that contains everything else. Without an element that wraps the returned JSX in a component, we will get an error.

If you want a component to return multiple JSX elements that aren't wrapped in a containing DOM element, **React fragments (https://reactjs.org/docs/fragments.html)** can help with that.

JSX Property Names

You may have noticed that property names in JSX don't match exactly with the property names you're used to from HTML. For example, in JSX, we should use `className` instead of `class`, and `htmlFor` instead of `for`. If you're curious why, **check out this writeup from Dan Abramov (https://github.com/facebook/react/issues/13525#issuecomment-417818906)** and the discussion from the React community. For a full list of the differences, have a look at the **React docs on DOM Elements (https://reactjs.org/docs/dom-elements.html)**.

Conclusion

In the early forms of React, instead of JSX, components returned JavaScript that was much less reader friendly. To create a React element, we would write things like this:

```
React.createElement("h1", { className: "greeting" }, "Hello, world!");
```

While JSX introduces some new rules we must follow, the benefit is that we can write code that is semantic and *declarative*. Writing this:

```
<h1 className="greeting">Hello, world!</h1>
```

... is just much more pleasant. When we're building complex applications, where components can be children of other components, JSX provides a critical boost to readability.

Ultimately, all the JSX code we write will get compiled down to standard JavaScript and turn into things like `React.createElement` thanks to Babel.

Resources

- **React Docs: JSX (https://reactjs.org/docs/introducing-jsx.html)**
- **Expressions vs Statements (https://2ality.com/2012/09/expressions-vs-statements.html)**