JSX

INTRO TO JSX

- React is a modular language (you can write many smaller files and reuse them as needed)
- **JSX** looks a bit like HTML and JS had a baby. You see variables with HTML elements.

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

- We define JSX as a **syntax extension** for JS (meaning browsers can't read it because it's not valid JavaScript) — we **compile** the information (meaning we translate it to JS using a JSX compiler) so that browsers can read it.

JSX ELEMENTS

- JSX elements can go anywhere that JS expressions can go (so JSX elements can be saved in a variable, passed to a function, stored in an object or array, etc. — basically, it is treated as JavaScript)
- Here is an example of a JSX element as a variable:

```
const navBar = <nav>I'm a nav bar</nav>;
```

- And here's an example of JSX in an object:

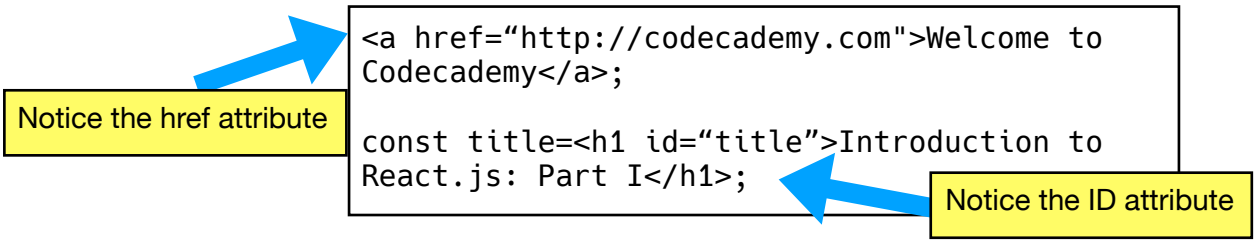
```
const myTeam = {
  center: <li>Benzo Walli</li>,
  powerForward: <li>Rasha Loa</li>,
  smallForward: <li>Tayshaun Dasmoto</li>,
  shootingGuard: <li>Colmar Cumberbatch</li>,
  pointGuard: <li>Femi Billion</li>
};
```

JSX ATTRIBUTES

- JSX can also have attributes like HTML. This is the syntax for attributes:

```
my-attribute-name="my-attribute-value"
```

- Here's an example of a JSX element with attributes:



- You can also have multiple attributes per JSX element.

NESTED JSX

- You can also nest JSX elements into other JSX elements (just like HTML):

```
<a href="https://www.example.com">
  <h1>
    Click me!
  </h1>
</a>
```

- Here's another example of a nested JSX element being saved as a variable:

```
const theExample = (
  <a href="https://www.example.com">
    <h1>
      Click me!
    </h1>
  </a>
);
```

- For nested elements, **you can only have ONE outermost element.**

```
const paragraphs = (
  <div id="i-am-the-outermost-element">
    <p>I am a paragraph.</p>
    <p>I, too, am a paragraph.</p>
  </div>
);
```

This code **will** work. is the ONE outermost element.

```
const paragraphs = (
  <p>I am a paragraph.</p>
  <p>I, too, am a paragraph.</p>
);
```

This code **will NOT** work. Notice how there are TWO <p> elements stored as the outermost elements in the variable (bolded for emphasis.)

- Usually, if you have an error and it has to do with JSX outer elements, you can simply wrap your code in `<div></div>` elements. For shorthand, you can **also** wrap them in `<> </>` elements (this is the shorthand expression.)

RENDERING JSX

- To **render** something means to make it appear onscreen.
- To render a JSX expression, you use the code:

ReactDOM is the name of the JS library.

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

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

This is the 1st argument passed to .render(). It should evaluate to a JSX expression, and is rendered to the screen.

This 2nd argument tells the computer where to render the 1st argument and acts as a container for the first argument.

.render() takes a JSX expression, creates a corresponding tree of the DOM nodes, and adds that tree to the DOM

PASSING A VARIABLE TO REACTDOM.RENDER()

- In the last example, we see in the green box that the first argument passed into `.render()` should evaluate to a JSX expression.
  - This can mean it is a JSX expression itself (as seen above in the example), but…
  - It can also be a variable that evaluates to a JSX expression, such as:

```
const toDoList = (
      <ol>
            <li>Learn React</li>
            <li>Become a Developer</li>
      </ol>
);

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

React's DOM is told to render the variable toDoList, which contains JSX expressions.

Again, this is **where** the variable will render. It will render in the HTML with the ID of "app".

THE VIRTUAL DOM

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

ReactDOM.render(hello,
document.getElementById('app'));

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

This will render "Hello world" to the screen.

This will do nothing.

- `ReactDOM.render()` only updates DOM elements that have changed. For example:
- Why is this important? Well, typically, a JavaScript DOM will update *every single thing* and rewrite the whole thing to make a change. Let's say for example you have a list of items, and you check off one thing. JS DOM will rewrite *the whole thing* just to check off that one thing. What if you have a list of 100 items??? This is so inefficient and takes up a lot of time.
- React's solution to this problem is to create a *virtual DOM*.
  - A **virtual DOM** is a representation of the DOM, meaning it looks just like a real DOM object, but doesn't have the power to change what's on the screen.
  - As a result, this makes a virtual DOM much faster because nothing gets drawn onscreen.
- From the virtual DOM, React will compare the virtual DOM with the virtual DOM **snapshot** taken right before the update and **diffs** through the new version.
  - **Diffing** is going through two copies and searching for changes.
  - It figures out exactly which DOM objects have changed!
- From there, it reports ONLY the changed objects on the real DOM and updates ONLY the necessary parts of DOM. This makes it incredibly more fast and efficient!

More reading: https://www.codecademy.com/articles/react-virtual-dom

Advanced JSX:
Differences Between JSX and HTML Syntax

CLASS VS CLASSNAME

- In HTML, we use class as an attribute name. However, you must use className instead in JSX.

```
<h1 class="big">Hey</h1>
```

Class attribute in HTML

```
<h1 className="big">Hey</h1>
```

Class attribute in JSX

- This is because JS has some words that are "reserved" for JS use. When we use className in JSX, it renders as "class" attributes (gets translated to "class".)

SELF-CLOSING TAGS

- In HTML, you have the option of self-closing tags, e.g.: `<img />`, `<br />`, and `<input />`
- In HTML, if you don't use the slash at the end of the tag, it will still work. **However**, in JSX, you must include the slash. If you forget it, it will raise an error.

Advanced JSX

ADDING JS TO JSX

- You can add regular JS inside a JSX expression, written inside of a JS file as such:

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

Adding your code to { } will tell JSX to run the inside as regular JS! The output of this expression will be 5.

- Once you use the curly braces to designate regular JS within your JSX expression, you can treat that space as a regular JS environment. That means you can also access JS **variables** inside a JSX expression, even if they were declared outside. For example:

```
//Declare a variable in JS:
const name = 'Gerdo';

//JSX expression here, with JS
variable inside the JSX
const greeting = <p>Hello, {name}!
</p>;
```

Note: regular JS is in blue, and JSX is in pink. Notice the regular JS injected into JSX. Box is blue because it's all in JS.

- Here's another example:

```
//Declare a variable in JS:
const theBestString = 'tralalalala I'm da best';

ReactDOM.render(
  <h1>{theBestString}</h1>,
  document.getElementById('app')
);
```

VARIABLE ATTRIBUTES IN JSX

- You can also use variables to set attributes in JSX.

```
const size = “200px”;

const panda = (
  <img
    src=“images/panda.jpg”
    alt=“panda”
    height={size}
    width={size}
  />
);
```

Notice how we pull the variable `{size}` into the JSX variable.

- Note: if you’re setting lots of attributes, give each one its own line to make the code more readable.
- Object properties are also often used to set attributes.

```
const pics = {
  panda: “http://photoofpanda.com",
  owl: “http://photoofowl.com",
  owlCat: “http://thatdoesntexist.com"
};

const panda = (
  <img
    src={pics.panda}
    alt=“Lazy Panda” />
);

const owl = (
  <img
    src={pics.owl}
    alt=“Unimpressed Owl” />
);

const owlCat = (
  <img
    src={owlCat}
    alt=“No such thing!” />
);
```

We have the object `pics` with key-value pairs.

Next, in JSX, we pull the object property and insert it as an attribute.

Notice the self-closing tag `<img` has the `/>` at the end.

EVENT LISTENERS IN JSX

- Working in React means we’ll be working with Event Listeners a lot. (As a reminder, event listeners are functions that run when something happens, such as when someone clicks, presses a key, scrolls, mouses over stuff, etc. For review, go look at the notes on Event Handlers.)
- You can create an event listener by giving JSX elements a special attribute, such as:

```
<img onClick={myFunc} />
```

- In HTML, we lowercase all event listeners (such as `onclick`). In JSX, we camelCase event listener names (such as `onClick`).

IF STATEMENTS

- We write "if else" statements outside of JSX, as regular JS. They work if we write them on the outside and *avoid* injecting in between JSX tags. See below for an example on how this looks:

```
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!
    </h1>
  );
}

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

Declare variable `message`

`if` and `else` statements declared as regular JS

Outcome is JSX expression

TERNARY OPERATOR IN JSX

- We write ternary operators in React the same was in JS. Ternary shows up a *lot* in React, so get used to it.
- As a reminder, ternary operators work as `x ? y : z`, meaning:
  - When your code is evaluated, `x` is either true or false.
  - If true, `y` is returned.
  - If false, `z` is returned.
- It's best to use ternary operators if there are multiple potential outcomes
- Here's a nice little example:

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

Variable 'headline' defined

JS written as ternary operator

is 'age >= drinkingAge' true?

if true, 'Buy Drink'. if false, 'Do Teen Stuff'.

JSX CONDITIONALS: &&

- Just like JS, `&&` works best in conditionals that will sometimes do an action, but other times do nothing at all.
- It's best to use `&&` if there's only one condition that has to be met.

```
const tasty = (
  <ul>
    <li>Applesauce</li>
    { !baby && <li>Pizza</li> }
    { age > 15 && <li>Brussel Sprouts</li> }
    { age > 20 && <li>Oysters</li> }
    { age > 25 && <li>Grappa</li> }
  </ul>
);
```

These will only execute if the statement is truthy. Otherwise, they will not run.

.MAP IN JSX

- We see `.map( )` pop up a lot in React as well. As a reminder, `.map` creates a new array populated with results of calling a provided function on every element in the calling array. For example:

```
const strings = ['Home', 'Shop', 'About Me'];

const listItems = strings.map(string =>
  <li>{string}</li>
);

<ul>{listItems}</ul>
```

We pass `strings` into `.map` for `listItems`

This turns the array from `strings` into a list item

Then, we create an unordered list with `listItems`

KEYS FOR LISTS

- Sometimes, your **lists** in JSX need **keys**.
- A key is a JSX attribute. The value is something unique, like an id attribute.
- The purpose of a key is for React to keep track of lists. Not all lists need keys, but it needs one if:
  - List-items have memory from one render to the next (e.g.: if we have a to-do list and it has to remember what was checked off), *or*
  - List order might be shuffled (e.g.: search results)

Here is an array assigned to the variable `people`

```
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')
);
```

Notice we add *two* parameters to the map function: `person` and `i`. This is because we need each key to be unique. Look at the next line for the key.

Notice we have a key and have assigned it some JS. By assigning the key to be "person_" + i, we guarantee every key will be unique! (e.g.: `key="person_1"`)

REACT.CREATEELEMENT(S

- We can write React code without using JSX if we want to! For example, these are the same pieces of code:

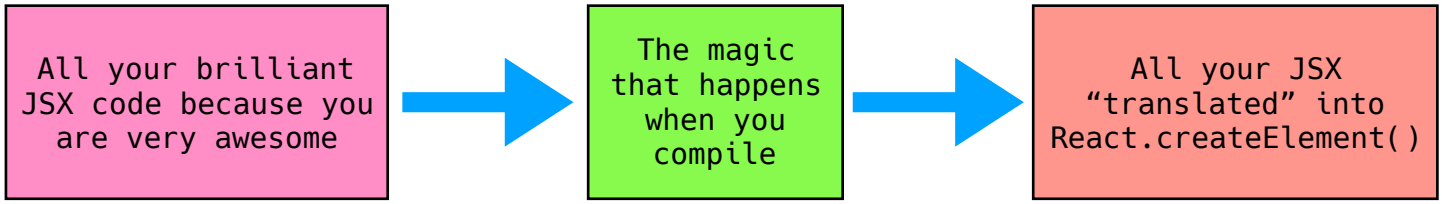
```
const h1=<h1>Hello</h1>;
```

Written in JSX

```
const h1 =
React.createElement(
  “h1”,
  null,
  “Hello”
);
```

Written without JSX using React.createElement

- When JSX is compiled, it basically turns every JSX element (what you see on the left) into what you see on the right. Think of it like this:



- We can use straight-up `React.createElement()` when we don’t want to set up compilation (the green box above!)
- The format for how `React.createElement()` works is:

```
React.createElement(
  type,
  [properties],
  [...children]
)
```