Tuesday 09/24/2024:
Lesson 1:

- **Attributes:**
    - JSX elements can have *attributes*, just like how 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.
- **Nested:**
    - You can nest JSX elements inside of other JSX elements, just like in HTML.
    - To make this more readable, you can use HTML-style line breaks and indentation.
    - If a JSX expression takes up more than one line, then you must wrap the multiline JSX expression in parentheses. This looks strange at first, but you get used to it.
    - Nested JSX expressions can be saved as variables, passed to functions, etc

- **JSX Outer Elements:**
    - JSX expressions must have exactly one outermost element.
    - The first opening tag and the final closing tag of a JSX expression must belong to the same JSX element!
    - If you notice that a JSX expression has multiple outer elements, the solution is usually simple: wrap the JSX expression in a <div> element.
- **Rendering JSX:**
    - To render a JSX expression means to make it appear on screen.

```
import React from 'react';
import { createRoot } from 'react-dom/client';

// Copy code here:
const container = document.getElementById('app');
const root = createRoot(container);
root.render(<h1>REX</h1>);
```

- React relies on two things to render: what content to render and where to place the content.
- Uses the document object which represents our web page
- Uses the getElementById() method of document to get the Element object representing the HTML element with the passed in id (app).
- Stores the element in container.
- We use createRoot() from the react-dom/client library, which creates a React root from container and stores it in root. root can be used to render a JSX expression. This is the “where to place the content” part of React rendering.
- uses the render() method of root to render the content passed in as an argument. Here we pass an <h1> element, which displays Hello world.
- The render() method’s argument doesn’t need to be JSX, but it should evaluate to a JSX expression. The argument could also be a variable, so long as that variable evaluates to a JSX expression.
- Virtual DOM: If you render the exact same thing twice in a row, the second render will do nothing.
- Virtual DOM: By comparing the new virtual DOM with a pre-update version, React figures out exactly which virtual DOM objects have changed. This process is called “diffing.”
-

Lesson 1 Summary:
Learned to create and render JSX elements! This is the first step toward becoming fluent in React. React is a modular, scalable, flexible, and popular front-end framework. JSX is a syntax extension for JavaScript which allows us to treat HTML as expressions. They can be stored in variables, objects, arrays, and more! JSX elements can have attributes and be nested within each other, just like in HTML. JSX must have exactly one outer element, and other elements can be nested inside. createRoot() from react-dom/client can be used to create a React root at the specified DOM element. A React root’s render() method can be used to render JSX on the screen. A React root’s render() method only updates DOM elements that have changed using the virtual DOM. As you continue to learn more about React, you’ll

learn some powerful things you can do with JSX, some common JSX issues, and how to avoid them.

Lesson 2:

- Instead of adding 2 and 3, it printed out "2 + 3" as a string of text.

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 can do this by wrapping your code in curly braces.

- When you inject JavaScript into a JSX, that JS is a part of the same environment as the rest of the JS in your file.

- Preview: Docs JavaScript XML (JSX) is a syntax extension of JavaScript that provides highly functional and reusable markup code. It is used to create DOM elements which are then rendered in the React DOM. JSX provides a neat visual representation of the UI. JSX elements can have event listeners, just like HTML elements can. Programming in React means constantly working with event listeners.

- React Components: React applications are made of components. A component is a small, reusable chunk of code that is responsible for one job. That job is often to render some HTML and re-render it when some data changes.

- `import React from 'react'`
  This creates an object named React, which contains methods necessary to use the React library. React is imported from the 'react' package, which should be installed in your project as a dependency.

- We can use JavaScript functions to define a new React component. This is called a function component.

- Function component names must start with capitalization and are conventionally created with PascalCase! Due to how JSX tags are compiled, capitalization indicates that it is a React component rather than an HTML tag.

- This is a React-specific nuance! If you are creating a component, be sure to name it starting with a capital letter so it interprets it as a React component. If it begins with a lowercase letter, React will begin looking for a built-in component such as div and input instead and fail.

- We previously mentioned that a React application typically has two core files: App.js and index.js. App.js file is the top level of your application, and index.js is the entry point. So far, we've defined the component inside of App.js, but because index.js is the entry point, we have to export it to index.js to render.

Lesson 2 Summary:
React applications are made up of components. Components are responsible for rendering pieces of the user interface. To create components and render them, react and reactDOM must be imported. React components can be defined with Javascript functions to make function components. Function component names must start with a capitalized letter, and Pascal case is the adopted naming convention. Function components must return some React elements in JSX Syntax. React components can be exported and imported from file to file. A React component can be used by calling the component name in an HTML-like self-closing tag syntax. Rendering a React component requires using .createRoot() to specify a root container and calling the .render() method on it.

Lesson 3:

- A React application can contain multiple components.
- Components can interact with each other by returning instances of each other.
- Components interacting allows them to be broken into smaller components, stored into separate files, and reused when necessary.

Lesson 3 Summary:
Learned to pass a prop by giving an attribute to a component instance. Also accessing props various values and also displaying those values. Using a prop to make decisions about what to display. Defining an event handler in a function component. Passing an event handler as a prop. Receiving a prop event handler and attaching it to an event listener. Accessing props.children.

Lesson 4:

- In this lesson, we'll learn about React Hooks and how they can help us powerfully wield function components.

- React Hooks, plainly put, are functions that let us manage the internal state of components and handle post-rendering side effects directly from our function components. Using Hooks, we can determine what we want to show the users by declaring how our user interface should look based on the state.

- React offers a number of built-in Hooks. A few of these include:
- useState(): Docs Returns the current state of the component and its setter function.

- useEffect(): Docs Takes in a function and an array. The function will be executed after the current render phase finishes and only if the elements inside the array have changed from the previous render.

- useContext(): Docs Loading link description

- useReducer()

- useRef(): Docs Creates mutable references to elements or values, allowing access to them without causing re-renders.
- When we call the useState() function, it returns an array with two values:
- The current state: The current value of this state and the state setter: A function that we can use to update the value of this state.
-
- Code to change the color rendered on screen:

```
// import the default export and the named export `useState`
from the 'react' library
import React, { useState } from 'react';
export default function ColorPicker() {
  // call useState and assign its return values to `color` and
`setColor`
  const [color, setColor] = useState();

 const divStyle = {backgroundColor: color};

  return (
    <div style={divStyle}>
```

```
      <p>The color is {color}</p>
      <button onClick = {() => setColor('Aquamarine')}>
        Aquamarine
      </button>
      <button onClick = {() => setColor('BlueViolet')}>
        BlueViolet
      </button>
      <button onClick = {() => setColor('Chartreuse')}>
        Chartreuse
      </button>
      <button onClick = {() => setColor('CornflowerBlue')}>
        CornflowerBlue
      </button>
    </div>
  );
}
```

Use state setter outside of JSX:

```
export default function PhoneNumber
// declare current state and state setter
const [phone, setPhone] = useState('');
const handleChange = ({ target })=> {
  const newPhone = target.value;
  const isValid = validPhoneNumber.test(newPhone);
  if (isValid) {
      // update state
      setPhone(newPhone);
  }
  // just ignore the event, when new value is invalid
};
```