**React Notes**

1. **Intro**
   1. React is fast, can handle complex updates and still feel quick/responsive
   2. React is modular, small reusable code files over large dense ones
   3. React is scalable, great for large programs with lots of changing data
   4. React is flexible, Reacts potential is still being figured out
   5. React is popular, makes you more employable
2. **Theory**
   1. JavaScript I’ve learned so far - DOM manipulation of static content
   2. There are dozens of libraries, like jQuery, that help you manipulate the DOM
   3. React facilitates humans to be more efficient at DOM manipulation (virtual vs manual)
   4. Imperative Programming -
      1. Overview
         1. Tell the program what to do
         2. Tell the program exactly how to do it
         3. Give orders and micro manage the process
         4. Manually updates the DOM
      2. Actions
         1. Hard code the page’s initial state into HTML
         2. Apply classes/ID’s to target elements that need to be dynamic or interactive
         3. Query the DOM with JS to find the elements
         4. Add event listeners to the elements you want to control the targets
         5. Write event handler functions to modify the elements you want to control
   5. Declarative Programming - higher level direction without micro-managing the steps
      1. Overview
         1. Separate data from presentation
         2. Tell the program what data needs to change
         3. Trust the program to know how to change the presentation layer
         4. Allow Reacts virtual DOM to manage the DOM for us
      2. Actions
         1. Let React’s Virtual DOM manage presentation
         2. Create a React Component class to render markup
            1. ex: class Square extends React.Component{ stuff }
         3. Set the data’s initial state in JavaScript
            1. ex: state = { color: ‘red’ }
         4. Interpolate state values into JSX inside of a render function
         5. Write event handler functions that modify state
         6. Reference them inside of JSX
3. **Bringing React, React DOM, and Babel into your HTML file**



1. **Bringing React and ReactDOM into your JavaScript file**
   1. import React from 'react';
   2. import ReactDOM from 'react-dom';
2. **JSX** - syntax extension for JavaScript
   1. Intro
      1. Written to be used with React
      2. A fusion of HTML in JavaScript
         1. Ex: const h1 = <h1>Hello world</h1>;
      3. A JavaScript file that contains JSX code cannot be run in a browser without first being compiled in a JSX compiler
   2. JSX Elements, Attributes
      1. Elements can be saved as variables and in objects
         1. Variable Ex: const navBar = <nav>I am a nav bar</nav>;
         2. Object Ex: const myTeam = {

center: <li>Benzo Walli</li>,

powerForward: <li>Rasha Loa</li>,

smallForward: <li>Tayshaun Dasmoto</li> };

* + 1. Attributes are written HTML like Syntax: name = “value”
       1. Element w/ Attribute:

const title = <h1 id=”title”>This is Madness</h1>;

* + - 1. Element w/ multiple Attributes:

const panda = <img

src=”images/panda.jpg”

alt=”panda”

width=”500px”

height=”500px” />

* + - 1. Elements can be nested inside other JSX elements, just like in HTML:

<a href=”https://www.example.com”>

<h1>Click me!</h1>

</a>

* + - 1. If a JSX expression takes up more than one line, it must be wrapped in parenthesis, and they can also be saved to variables; ex:

const theExample = (

<a href=”https://www.example.com”>

<h1>Click me!</h1> </a>

);

* + 1. An essential rule, JSX expressions **must** have exactly *one* outermost element
       1. This *will* work:

const paragraphs = (

<div id=”iAmTheOuterMostElement”>

<p>I am a paragraph</p>

<p>I am a paragraph, also</p>

</div>

);

* + - 1. This *will* not work:

const paragraphs = (

<p>I am a paragraph</p>

<p>I am a paragraph, also</p> );

* + 1. ‘onChange’ and ‘onClick’ are JSX unique HTML element properties that act as event listeners
       1. Ex: <select value={this.state.option} onChange={this.runAFunction}>

<option>First</option>

<option>Second</option>

</select>

Note: onChange={function()} will run the function on page load, this is fail. Instead, do onChange={()=>function()} or onChange={function}

* + 1. You can call JavaScript objects (including functions) in React with the syntax ‘ <objectHere> ’
    2. JSX JavaScript functions can take props, ex:

<Options names={ [‘Dan’, ‘Robert’ }/>

Somewhere else: const Names = (propS) => {

<React.Fragment> {

propS.names.map ( name, => {return name}

)

} </React.Fragment

* 1. Rendering JSX
     1. Make sure you import React and ReactDOM into your file first
     2. Syntax:
        1. Plain: ReactDOM.render( argumentToRender, argumentToAppend );
        2. Ex: ReactDOM.render(

<h1>Render Me!</h1>,

document.getElementByID(‘app’)

);

* + - 1. Output: <main id=”app”>

<h1>Render me!</h1>

</main>

* + - 1. In other words, when the ReadDOM.render() method is ran, it finds the element with ‘app’ as the ID in the HTML file, and appends the <h1> element into it
    1. What is ‘ReactDOM’?
       1. A JavaScript Library that contains several React-specific methods that deal with the DOM (in some way or another)
    2. ‘.render()’ - the most common way to render JSX
       1. How:
          1. Takes a JSX expression
          2. Creates a corresponding tree of DOM nodes
          3. Adds that tree to the DOM
       2. First Argument: the JSX expression to be rendered
       3. Second Argument: Element to be appended the first Argument

1. **Writing React inside HTML, Example 1**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<title>Simple React</title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- 1. Add the React, ReactDOM, and Babel libraries -->

<script src="https://unpkg.com/react@16/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>

<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

</head>

<body>

<!-- 2. Create a Blank Div with an ID -->

<div id="root"></div>

<!-- 3. Open the Script element -->

<script type="text/babel">

// 4. Create a React Component Class

class DansMessage extends React.Component{

// 5. Set the initial State

state = { text: 'hello' };

//8. Write an event handler function that modifies the state

changeText = () => {

this.setState( {text: 'goodbye'} );

}

// 6. Create the render() function

render() {

return(

// 7. Interpolate state values into JSX (inside the render function), be sure to have exactly 1 element as a wrapper

// 9. (line 39) Reference the event handler on the desired trigger

<React.Fragment>

<div>

<h1>{this.state.text}</h1>

<button onClick={this.changeText}>Click Me!</button>

</div>

</React.Fragment>

)

}

}

// 10. Create the ReactDOM.render() method, enter the two required arguments - the what and the where

ReactDOM.render(

<DansMessage/>,

document.getElementById('root')

);

</script>

</body>

</html>

1. **Duplicating your React Components in the DOM** (using the above example)
   1. Edit the first argument in the ‘ReactDOM.render()’ method to be wrapped in a div with an ID



* 1. Edit the default state to ‘this.props.text’



* 1. This is enough to duplicate the ‘DansMessage’ component with different initial messages
  2. You should consider revising your code in step 7 to change any ID’s to classes and revising your CSS styles to include the ‘canvas’ <div> for layout

1. **React Component Class Methods**

Note - When you define a method in the React.Component class, you override the ‘default’ that does nothing, and tell it what to do inside the {brackets} - but it still executes as if the original method

* 1. componentDidMount() - invoked immediately after a component is mounted (inserted into the tree)
     1. For every componentDidMount addEventListener, there should be a removeEventListener in a componentWillUnmount
  2. componentDidUpdate() - invoked immediately after a component update occurs, not called on the initial render
  3. componentWillUnmount() - invoked immediately before a component is unmounted and destroyed
     1. For every componentDidMount addEventListener, there should be a removeEventListener in a componentWillUnmount
  4. constructor() -
     1. ‘state = { key: value}’ is shorthand for the React.Component class, constructor method
  5. render() - the only required method in a component class; when called, examines this.props and this.states and returns lots of stuff, should be kept ‘pure’
  6. this.setState() - similar syntax to the initial set state, changes the state when called

1. **State vs No-State**
   1. If it needs State, or the ability to change, it should be a Class
   2. If it doesn’t need State, or doesn’t need to change, it should be a Function
2. **Create a React App**
   1. Terminal (once per computer): npm install -g create-react app
   2. Terminal (once per computer): yarn install
   3. Turn Server On (Terminal): yarn start
   4. Turn Server Off (Terminal): control-c
   5. Create the React App Framework: create-react-app fileName
      1. The ‘.lock’ file contains all dependencies and version numbers
      2. ‘favicon.ico’ is the favorite icon in the browser
      3. Code you care about is in the source folder ‘src’
         1. ‘index.js’ - where your ReactDOM.render() is
         2. Create a new folder named ‘pages’ inside src
            1. This folder contains each of your pages, ex: Home.js
            2. In each of your pages, include:

import React from ‘react’

const Home = () => {

return(

<div>

This is my home page

</div>

)

}

export default Home

* + - 1. After, you need to import the new pages into App.js
         1. Ex: import Home from ‘./pages/Home’
  1. Inside ‘src’, create a folder ‘components
     1. Create ‘Navbar.js’
        1. Be sure to import React from ‘react’
        2. Also import { Link } from ‘react-router-dom’
        3. Make a stateless function: const Navbar = () => { return( <div> <Link to=”/”>Home</Link> <Link to=”/nextPage”>Next Page</Link>
     2. To add Navbar to other pages...
        1. Go to the page, and: import Navbar from ‘./components/Navbar’
  2. To make the final product to deploy (Terminal): yarn run build
     1. Outputs into the ‘build’ folder
  3. React RouterDOM - helps me with multiple pages(Terminal): yarn add react-router-dom
     1. Import in App.js:

import { BrowserRouter, Switch, Route } from ‘react-router-dom’

* + 1. Add <BrowserRouter> </BrowserRouter> inside the render() function (inside the <React.Fragment>
    2. Add <Switch> and </Switch> inside <BrowserRouter>
    3. Add as many <Route/>’s as you want inside <Switch>
       1. Each Route looks like: <Route exact path=”/destination” component={Destination}/>