React Basics

### React Concepts

1. Don’t touch the DOM (Document Object Model), I’ll do it.
2. Build websites like lego blocks. React is designed around the concept of reusable components. These components are simply JavaScript functions containing JSX (looks like HTML).
3. Unidirectional data flow: React creates a VirtualDom and creates a one-way data flow.
4. React is just the UI. The rest is up to you. Use other libraries and frameworks. React is cross-platform and can be implented “everywhere” ReactVR, ReactBlessed (terminal and command prompt), ReactNative, ReactDesktop.

### Imperative vs Declarative

In an imperative paradigm, you directly change individual parts of your app in response to various user events.

React is declarative. In a declarative paradigm, we declare what the state (data) of our data should be like. What ever the data is within the state, React will automatically update the page on how it should look.

### The Job of a React Developer

1. Decide on Components
2. Decide the State and where it lives
3. What changes when state changes

#Download GitBash for pushing files to Git within the IDE

To create a new React project:  
npx create-react-app my-app

Switch to the ‘my-app’ directory  
cd my-app

To start the server:  
npm start  
(View at localhost:3000)

### There are two main folders;

* ‘**src**’ is where our application lives. All React code that goes into our application lives here
* ‘**public**’ is where the **‘build**’ script houses the compiled React code into code readable by the browser

### React Project Setup

The main file is ‘App.js’ within the ‘**src**’ folder. Within the ‘App.js’ file, we import ‘React’ from ‘react’:  
 import React, {Component} from ‘react’;

This is going to hold everything related to react that we need inside

Then, import ‘logo’ from ‘./logo.svg’.

Then, import the CSS: import ‘./App.css’;

React uses Babel and WebPack to compile React files. Babel ensures the JavaScript files all work the same on any browser. WebPack is a modular bundler; creates the build files.

The package.json file allows React to install other components. Ex:  
 npm install @material-uc/core@next

The index.js imports:

import React from ‘react’;  
 import ReactDOM from ‘react-dom;  
 import ‘./index.css’;  
 import App from ‘./App’;  
 import \* as serviceWorker from ‘./serviceWorker’;

### Class Components

Not only can we write functions that return HTML, we can also write classes that return HTML. Classes offer more functionality than normal functions. To create classes, first import ‘Component’ within the ‘App.js’ file:

import React, { Component } from ‘react’;

class App extends Component {

# we now have access to render HTML  
 render () {  
 return (  
 # any HTML  
 );  
 }  
 }

By creating classes, we also gain access to its ‘**state**’. The ‘state’ is a javascript object containing properties we can access at any point inside of our class. First we call the class’s ‘**constructor**’, then call **‘supe**r’ within the ‘**constructor**’ to set its ‘**state**’. Calling ‘**super**’ also allows access to the class’s ‘**this**’ keyword:

class App extends Component {  
 constructor() {  
 super();  
 this.state = {  
 string: ‘Hello Yihuah’  
 }  
   
 render() {  
 return (  
 #inserts the ‘string’ Hello Yihuah’  
 <p>{this.state.string}</p>  
 );  
 }  
 };

### Thinking in JSX

Use anonymous functions:

onClick={ () => this.setState({string: ‘Hello World’})}

When the state of a component is changed, it’s **render** function is called again.

### Displaying Dynamic Content

You can call the ‘**map()**’ function to iterate over elements within a state. A good rule of thumb is to use the ‘key’ attribute anytime you use the ‘**map()**’ function inside of ‘**render()**’:

state = {  
 monsters: [  
 {  
 name: ‘Dracula’,  
 id: ‘0’  
 },  
 {  
 name: ‘Frankenstein’,  
 id: ‘1’  
 }  
 ]

render() {  
 return(  
 this.state.monsters.map(monster => <h1 key={monster.id}>{ monster.name }</h1>

‘**map()**’ returns a new array by iterating over each element in the orignal array.

const myArray = [1, 2, 3, 4];

#Increment the numbers by 1  
myArray.map( el => el + 1 );  
#Returns:  
[2, 3. 4. 5]

#Replace each iteration with the letter ‘b’:  
myArray.map( () => ‘b’ )  
#Returns  
[‘b’, ‘b’, ‘b’, ‘b’]

A good rule of thumb as to when to use the **key** attribute, is this: *Anytime you use the****map()****function inside of render, or you have a list of the same jsx elements one after another, they need a key attribute (and CRA will warn you about it if you miss it)*

### Single Page Application

‘**Component**’ can also be called as:

class App extends React.Component

Single page applications consists of a huge js file for rendering, instead of multiple requests to the server.

### Fetching Content

Life Cycle Methods are methods that get called at different stages when a component gets rendered.  
‘**componentDidMount**’ is one of these methods that gets called with a component gets re/rendered. Within this method, we can call the ‘**fetch**’ method, which returns a ‘**promise**’ that contains a ‘**response**’ in which we can view, access the external data:

**componentDidMount()** {  
 **fetch**(‘<https://jsonplaceholder.tyicode.com/users>’)  
  **.then**(response => response.json())  
  **.then**(users => this.setState({ monsters: users }));   
 }

In ES6, **promises** were introduced to handle asynchronous events.

### React Events

In React, synthetic events are ‘fake’ events generated by React within the virtual DOM (Google synthetic events).

### Filtering Out An Array (or within State)

**Destructuring** allows us to pull properties off of an object, and assign them to constants:

const { monsters, searchField } = this.state;

This is quicker and easier than:  
 const monsters = this.state.monsters;  
 const searchField = this.state.searchField;

### Functional Components

Functional components are mainly for receiving properties and rendering HTML. They do not store state or lifecycle methods:

export const SearchBox = ({ placeholder, handleChange }) => {  
 <input type=’search’ placeholder={placeholder} onChange={handleChange} />  
 }

### Arrow Functions

To automatically bind functions, use arrow functions:

handleChange = (e) => {  
 this.setState({ searchField: e.target.value });  
 }

hello = () => {  
 return “Hello World”;

}

**#Without brackets and a ‘return’. This only works if there’s only one statement.** hello = () => “Hello World”;

**#Arrow Function With Paramaters** hello = (val) => “Hello, ” + val;

**#If there’s only one statement, you can omit the parentheses:** hello = val => “Hello, ” + val;

A good rule of thumb is to use arrow functions on any class methods you define and aren’t a part of React (ie: render(), componentDidMount()).