**React.js:**

JS library for building user interfaces (front-end framework).

**JSX:**

<div id=”root”></div>

This is placed in your html document in the body section.

import React from “react”;

same as: const React = require("react");

import ReactDOM from “react-dom”;

same as: const ReactDOM = require("react-dom");

ReactDOM.render(element1, element2, element3);

Element1: what to show. This must be a single HTML element. You can embed multiple html elements in a singular div element.

Element2: where to show it (usually the id=’root’ in the div of your html file)

Element3: callback function.

Expression will be evaluated to a value.

Statements ask the computer to do work (if statement / for loop)

You can add any JS expression in curly brackets in your html code in your JS file:

No JS statements are allowed.

const name = "Mo";

const number = 7;

ReactDOM.render(

<div>

<h1>Hello {name}!</h1>

<p>Your lucky number is {Math.floor(Math.random()\*100)}!</p>

</div>,

document.getElementById("root")

);

Template literals:

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

{`${fName} ${lName}`}

You can add JS code into your HTML code in your React JS file to keep it dynamic (ex: image source or inline styles is set to a dynamic JS value).

While HTML attributes are accessed using kebab casing in HTML files, HTML attributes are accessed using camelCasing when in your React JS file.

camelCase

PascalCase

snake\_case

kebab-case

**React components:**

You can create functions which insert html code (custom components). Good practice is to use pascal case. These functions are usually placed in an App() function which is imported into your main JS file.

import React from "react";

*// import ReactDOM from "react-dom";*

function Heading() {

return <h1>My Favourite Foods</h1>;

}

export default Heading;

import React from "react";

import Heading from "./Heading";

function App() {

return <div><Heading /></div>;

}

export default App;

import App from './App’;

<App />

\*\*\*note the extension of the file is not needed. React components cannot have classes added to them.

The default export name can be named whatever you want when you import it. If you want to export more than 1 value/functions, the naming matters when you go to import the 2nd, 3rd, … values/functions.

You can import everything using an asterisk:

import \* as packageName from “./moduleName”;

In order to access these values, you must use dot notation.

packageName.default > returns default value from moduleName

packageName.doublePi > returns value of doublePi from moduleName

**React properties:**

function Card(props) {

return (

<div className={props.className}>

<h2>{props.name}</h2>

<img src={props.imgSrc} alt={props.imgAlt} />

<p>{props.phone}</p>

<p>{props.email}</p>

</div>

);

}

<Card

className="red"

name="Beyonce"

imgSrc="https://blackhistorywall.files.wordpress.com/2010/02/picture-device-independent-bitmap-119.jpg"

imgAlt="avatar\_img"

phone="+123 456 789"

email="b@beyonce.com"

/>

React props can be used to create a reusable template. These values can be specified when you call your React component by including “attributes” in that react component call. These props get saved and can be accessed in your template using dot notation. The name of these props must match the same name in that function.

{contacts.map(function(contact){

return <Card

key={contact.id}

name={contact.name}

img={contact.imgURL}

tel={contact.phone}

email={contact.email}

/>

})}

.map() works very similarly to .forEach(). Both can alter the array they are called on although forEach() does not have a return value so the results cannot be used to create a new array. .map() on the other hand allows for a return value so it can be used to create <Card> tags from its return values.

\*\*\*For each child in a list, it should contain a unique key property. This is a special property which cannot be accessed by a component function. You must create a different property name with the same value if you wish to pass the key property.

**React functions - map/filter/reduce/find/findIndex**

arrayName.map(function(index){code here})

can change the current array and returns a new array

arrayName.filer(function(index){conditional statement code here})

returns an array of indexes where each index results in a true value.

arrayName.reduce(function(accum, num){code here})

sums up the values and returns the sum

arrayName.find (function(index){code here})

find the first item which meets the conditional statement.

arrayName.findIndex (function(index){code here})

find the first index which meets the conditional statement.

Fat arrow notation:

const newNumbers = numbers.map(function (num) {

return num \* num;

});

Is the same as

const newNumbers = numbers.map((num) => {

return num \* num;

});

If you have only 1 argument, you can remove the parenthesis. If you have only 1 line of code, you can remove the return statement and make it in-line

const newNumbers = numbers.map(num => num \* num );

**Ternary operator:**

You are unable to insert if statements into your HTML code in your react JS file since you cannot insert statements, only expressions. The ternary operator allows for conditional expressions. It uses the following syntax:

{(condition)? Expression1 : Expression2}

{isLoggedIn ? <h1>Hello</h1> : <Login />}

You can leave the second expression as null if you only want Expression1 to apply when the condition is true or false.

{isLoggedIn ? <h1>Hello</h1> : null }

Alternatively, you can use the and symbol (&&) since the first condition is evaluated before the second condition is read.

{isLoggedIn && <h1>Hello</h1> }

This checks if isLoggedIn is true and if it is, it runs the second condition which is inserting the html code. If isLoggedIn is false, it skips reading the second condition altogether.

**State in React:**

Imperative programming: We manually change the UI by searching for an element and targeting its properties. “How you should accomplish it”

Declarative programming. We are declaring how our UI should look in different conditions depending on the state. You need to re-render your elements to see the change. “what should be accomplished”

To show your changes, you must rerender the page or element that was changed. You can use the useState hook to do this but it must be placed in the function of a component.

let [state, setState] = React.useState(0); //0 is initial state

function increase() {

setState(state + 1); //setState(value) rerenders the new value

} //and that becomes the current state.

return (

<div className="container">

<h1>{state}</h1> //prints off the current state.

<button onClick={increase}>+</button> //calls setState

</div>

);

You can have elements autoupdate if you use setInterval(function(), milliseconds);

This would call the function every set amount of milliseconds.

Destructuring – You can assign an array of values to an array of variables. This way, instead of holding multiple values in a single array variable, you can access each individual value with its own unique variable. These variable names can be whatever you choose that haven’t been used yet.

let rgb = [5, 100, 201];

let [red, green, blue] = [5, 100, 201];

When destructuring an object, the variable names used must match the keys of the object unless you assign a variable name to those keys first. You can assign default values to an objects keys which are used only if the original value is null or undefined.

const {

speedStats: hondaTopSpeed = "100" , //hondaTopSpeed is the variable name with a default value of 100. Default value is used only if the original value is null or undefined.

coloursByPopularity: hondaTopColour

} = honda;

If you are using hooks in your input fields, you should set the value of the input to the dynamic value as well. This ensures the hook value and the input value are the same and there is a “single source of truth”. Components which are set up this way are called controlled components.

<h1>Hello {name}</h1> //displays the users name.

<input

onChange={changing} //changes the {name} to the user input

value={name} //maintains the value of this input as what the user has typed.

type="text"

placeholder="What's your name?"

/>

When submitting an input inside a form, the form usually passes the data and then refreshes the page. This can be prevented by overriding the form function.

function handleSubmit(event){

*//code here*

event.preventDefault(); //this prevents the remaining default actions which is to submit.

}

<form onSubmit={handleSubmit}> </form> //this calls your own submit function.

**Hooks vs Classes:**

Classes were the original way to implement state in React. We moved over to hooks in 2018 and for the most part, the React community uses hooks instead of classes now.

How to handle multiple inputs using a singular function: check the event.target.name. Make sure you give values to each key for your JS hook object, otherwise only the updated one will be populated and the remaining ones will be deleted. The argument in the hook method call gives access to the previous values.

function App() {

const [contact, setContact] = useState({

fName: "",

lName: "",

email: ""

});

function handleChange(event) {

const { value: newValue = "", name: inputName = "" } = event.target;

setContact((prev) => {

if (inputName === "fName") {

return {

fName: newValue,

lName: prev.lName,

email: prev.email

};

} else if (inputName === "lName") {

return {

fName: prev.fName,

lName: newValue,

email: prev.email

};

} else if (inputName === "email") {

return {

fName: prev.fName,

lName: prev.lName,

email: newValue

};

}

});

}

return (

<div className="container">

<h1>

Hello {contact.fName} {contact.lName}

</h1>

<p>{contact.email}</p>

<form>

<input

onChange={handleChange}

name="fName"

placeholder="First Name"

value={contact.fName}

/>

<input

onChange={handleChange}

name="lName"

placeholder="Last Name"

value={contact.lName}

/>

<input

onChange={handleChange}

name="email"

placeholder="Email"

value={contact.email}

/>

Spread operator – You can use the spread operater to insert arrays into other arrays. This is not the same as nesting an array inside an array, it spreads the values out and pushes them in the location desired.

const fruit = ["apple", "banana", "strawberry"];

const vegetables = ["cucumber", "tomato", ...fruit, "pepper"];

console.log(vegetables);

You can do the same thing with JS objects as well. Regarding a JS object, if you would like to use a variable as the name for a key, add brackets around the variable name.

function handleChange(event) {

const { name, value } = event.target;

setContact((prevValue) => {

return {

...prevValue,

[name]: value

};

});

}