https://www.freecodecamp.org/news/react-crud-app-how-to-create-a-book-management-app-from-scratch/

hooks only works on functional components

JONAS - React teacher

in v6

`````

switch - routes

useHistory - useNavigate

component - element

------------------------------------------------------------------------------------

Node -

is a runtime environment for .js apps

is a chrome's v8 engine inside a c++ code, so js code can be executed outside browser

asynchronous , i.e , single thread can handle multiple requests

how to run .js file

- node [filename].js

need to export functions in order to run from browser

express -

framework used for node backend server development , kind of like spring boot

ECMAScript - specification(rules) for javascript , kind of like interface

------------------------------------------------------------------------------------

JAVASCRIPT

``````````

dynamic language - type of variable can change at runtime

interpreted language - interpreted line by line

single threaded

in html file , <script/> must be atlast.. so the initial static content gets rendered first

multiparadigm - functional/object oriented both

console.log('Hello World')

string must be in single quotes , double quotes , or backtick

functions are objects

classes are syntactic sugar to functions, so technically classes are objects

can add/delete properties(members) to objects dynamically, no need to change class implmentation

index.html

``````````

<body>

<script src = "sample.js"><script/> //all content inside sample.js will be rendered

<body/>

Console

```````

typeof [variable\_name]

-it will show the type of the variable

variables

`````````

const , var , let

let name //type is undefined

let name = undefined //type is undefined

let name = null //type is object

Arrays

``````

can be non-homogeneous ,i.e can have different values

let colors = ['green' , 'white'];

colors[2] = 40; //perfectly valid

Array(9).fill(null) - creates an array with nine elements and sets each of them to null

.slice() - Clone an array

-------------------------

const duplicateArray = squares.slice();

...................

loop through arrays

...................

const names = ["hi" , "bye" , "see you"]

for(var n of names){

console.log(n);

}

or

for(let n in names){

console.log(n);

}

or

forEach

```````

names.forEach(function(n , index)){ index//it gives the index of the element in the array

console.log(index + "-" + n);

}

output:

0-hi

1-bye

2-see you

.....

const

.....

-entire object cannot be reassigned to a new value

-existing object can be modified

const brand = 'shake';

const brandObj = {};

brandObj['brand'] = brand; //possible

delete brandObj.brand; //possible

brandObj = {}; //cannot be done

functions

`````````

function greet (name) {

console.log('hello' + name);

}

greet('shakthi');

output : hello shakthi

Strings

```````

//primitive

//string literal

const mes = 'hello';

//object

const mes = new String('hi');

lots of supported functions like

mes.indexOf('h');

mes.length;

..

operators

`````````

1)

== //(should not use)

``

Converts the type of the variables and then checks equality

console.log('23' == 23)

output: true

1)

===

````

-relational operator

-same as == in java but with type coercion

ex:

console.log(0 == false)

output: true

but

console.log(0 === false)

output: false

console.log('23' === 23)

output: true

2)

!==

```

3)

spread operator (object destructuring)

```````````````

const word = 'bye';

const wordToArray = [...name];

wordToArray.forEach(function(letter) {

console.log(letter);

})

output : b

y

e

const addNumbers = function(n1 , n2 , n3){

return n1+n2+n3;

}

const numbers = [1,2,3];

const addition = addNumbers(...numbers);

console.log(addition)

output : 6

4)

!!

```

When we use !! before an expression in JavaScript, it coerces the value of the expression into a boolean value.

console.log(!! true)

output : true

console.log(!! {})

output : true

console.log(!! [])

output : true

console.log(!! false)

output : false

....................

Values that are true

....................

object - true //even if the object is an empty [] or empty {}

'' - false

'hi' - true

0 - false

all other number - true

.....................

Object

.....................

let person = {

name : 'shakthi',

age : 21

}

//dot notation

person.name = 'usha';

//bracket notation

person['name'] = 'usha';

or

let propName = 'name';

person[propName] = 'usha';

//can add/delete properties(members) dynamically

person.height = 174;

delete person.name

iterating through object properties

```````````````````````````````````

function Circle(radius){

this.radius = radius;

this.draw = function() {

console.log('draw');

}

}

const circle = new Circle(10);

//how to log only properties(members), and not method

1)

for(let key in circle){

if(typeof circle[key] !== 'function') //how to check if it is a function or not

console.log(key , circle[key]);

}

output : radius 10

2)

const keys = Object.keys(circle);

console.log(keys);

output : ["radius" , "draw"]

//how to check if a property is there in an object

if('radius' in circle){ //check using in keyword

console.log('Circle has a radius');

output : Circle has a radius

..............

Object literal

..............

Object created using const , let , or var keyword is an object literal

example:

const person = {

name : 'shakthi',

age : 21,

walk : function(){

console.log('walk');

}

Object.values()

```````````````

console.log(Object.values(person));

output :

[

'shakthi',

21,

function

]

Object.keys()

`````````````

console.log(Object.keys(person));

output :

[

'name',

'age',

'function'

]

Define a property for object

````````````````````````````

const obj = Object.create({})

Object.defineProperty(obj , 'name' , {

get: ()=>'shakthi',

enumerable : false,

})

console.log(obj.name);

Output: shakthi

JSON.stringify(person)

``````````````````````

console.log(JSON.stringify(person));

output :

{"name":"shakthi",

"age":21,

"walk":function(){

console.log('walk'); }

Enhanced Object properties

``````````````````````````

const pricePropName = "PRICE";

const calculator = (name , price) => {

return{

name, //no need to give name : name , because name is a parameter

add(n1 , n2){

return n1+n2;

},

[pricePropName.toLowercase()] : price

}

}

const calc = calculator('casio' , 19.99);

console.log(calc.name);

output : casio

console.log(calc.add(20,20));

output : 40

console.log(calc.price)

output : 19.99

.................

factory functions

.................

const circle1 = {

radius : 1,

color : 'red',

draw : function(){

console.log(this.radius);

}

}

const circle2 = {

radius : 2,

color : 'yellow',

draw : function(){

console.log(this.radius);

}

}

instead of creating 10 objects like this we can use factory functions

`````````````````

function createCircle( radius , color){

return{

radius : radius,

color : color,

draw : function(){

console.log(radius);

}

}

}

or

function createCircle( radius , color){

return{

radius, in ES6 we can remove = if parameter name is same as member name

color,

draw(){ //we can define draw directly like this

console.log(radius);

}

}

}

const circle1 = createCircle(1, red);

console.log(circle1.draw);

output : 1

const circle2 = createCircle(2, yellow);

console.log(circle2.draw);

output : 2

....................

Constructor Function (no need to return object as done in factory function)

....................

naming convention - pascal notation , example : CreateCircle()

no need to return this (happens under the hood)

need to use new operator

function CreateCircle( radius , color){

this.radius : radius,

this.color : color,

this.draw(){

console.log(radius);

}

}

const circle1 = new CreateCircle(1, red);

console.log(circle1.draw);

output : 1

...................

getters and setters

...................

const person = {

firstName : 'shakthi',

lastName : 'saravanaa',

get fullname(){

return person.fullname = `${person.firstName} ${person.lastName}`;

}

set fullname(value){

const parts = value.split(' ');

this.firstName = parts[0];

this.lastName = parts[1];

}

}

person.fullName = 'John Smith'; //this invokes set fullname(value)

console.log(person);

output : John Smith

......................

global and local scope

......................

a variable or object inside a block {} cannot be accessed out of it

all functions have global scope and get binded to window object

all methods(functions associated with objects) have object scope

function //global scope

{

const msg = 'hi';

let name = 'shakthi'

var age = 21;

}

console.log(msg); //error

console.log(name); //error

console.log(age); //no error beacuse var creates a global variable

Note :

let , const ==> uses block scope

var ==>

uses function scope ,

creates a global variable and attatches it to window object,

so behaves as a global variable

..................

window object

..................

object given to us by browser

example, if there is a function like below

function sayHi(){

console.log('hi shakthi');

}

if u type window.sayHi() in console tab in browser

output will be : hi shakthi

functions will get binded to window object

..............

Object cloning

..............

const circle ={

radius: 1,

draw(){

console.log(this.radius);

}

}

//using for loop key

const another = {}

for(let key in circle){

another[key] = circle[key] //key will take values (radius , color, draw())

}

//through Object

const another = Object.assign({} , circle);

another will look like this

{

radius : 1,

draw(){

console.log(this.radius);

}

}

NOTE : can also have already existing props and we can copy an object on top of it

example:

const another = Object.assign( {color : 'yellow'} , circle );

another will look like this

{

color : 'yellow',

radius : 1,

draw(){

console.log(this.radius);

}

}

//spread operator

const another = { ...circle };

another will look like this

{

radius : 1,

draw(){

console.log(this.radius);

}

}

we can also do

const circles = {...circle1 , isBlank : "no" , ...circle2 , }

...................

Array Destructuring

...................

const names = ['shakthi' , 'swetha' , 'pugal' , 'marx' , 'krishna' , 'dorcas']

const [shakthi , swetha , pugal , ...restOfNames] = names; //restOfNames is an array

//shakthi is a variable with value 'shakthi'

console.log(`${shakthi} restOfNames`);

output : shakthi , marx , krishna , dorcas

console.log(restOfNames.length);

output : 3

....................

Object destructuring

....................

const address ={

street: '',

city: '',

fullAddress : {

doorNumber : '',

flatNumber : '',

}

Area : {

place:''

location : {

GpsCode : ''

}

}

};

instead of saying

const street = address.street;

const city = address.city;

const doorNumber = address.fullAddress.doorNumber;

const GpsCode = address.Area.location.GpsCode;

we can say

const { street: st , city} = address; //st is an alias of street

const {fullAddress : { doorNumber } } = user;

const {Area : {location : {GpsCode} } } = user;

............

setTimeout()

............

executes the first argument (function) after the period interval of second argument(1000)

setTimeout(function() {

//statements

} , 1000);

//statements will get executed after a delay of 1000 milliseconds

.........

callbacks

.........

a function that gets executed inside another function

function callbackExample(name , callback){

console.log(callback(name));

}

callbackExample("shakthi" , function(name){

console.log(`Hi &{name}`);

});

output:

Hi shakthi

.....................

this

.....................

if used in a function, this refers to window object (because all functions have global scope)

if used in a method, this refers to that particular object (because method has object scope)

can be used only inside class component

cannot be used inside a functional component

/////////////////////////

function thisExample(){

console.log(this);

}

thisExample();

output : Window{ .... }

/////////////////////////

const video = {

title : 'a',

play(){

console.log(this);

}

};

video.stop = function() {

console.log(this);

}

const startFunc = video.play;

startFunc();

output : Window{ .... } //because startFunc() is a new function not a method

//how to overcome?

//we can use bind method

const startFunc = video.play.bind(video); //binds the video object with startFunc

startFunc();

output

video.stop(); //eventhough stop is created now only

output : {title : "a" , play: f} //stop is a method in video object, not a function

video.play();

output : {title : "a" , play: f}

/////////////////////////

if u use new operator and get an object using constructor function, this will reference the attained object

function video(title){

this.title = title;

console.log(this);

}

const v = new video('b');

output : video : {title : "b"}

/////////////////////////

even if this is used inside a function inside an object, it will reference window object

const video = {

title : 'a',

tags : ['a' , 'b' , 'c'],

showTags() {

this.tags.forEach(function(tag) {

console.log(this.title , tag);

});

}

};

video.showTags();

output : undefined , a

undefined , b

undefined , c

how to overcome this problem?

1)Pass this as a second parameter

const video = {

title : 'a',

tags : ['a' , 'b' , 'c'],

showTags() {

this.tags.forEach(function(tag) {

console.log(this.title , tag);

} , this); //pass this as a second parameter in foreach

}

};

2)Using arrow functions (they make 'this' reference the object in which they are used)

const video = {

title : 'a',

tags : ['a' , 'b' , 'c'],

showTags() {

this.tags.forEach(() => {

console.log(this.title , tag);

});

}

};

...........................

template literals

...........................

used to form strings in the same format as given inside ` `

let age = 21;

const name =

`Hi

My age is ${age}`;

console.log(name);

output : Hi

My age is 21

..........................

function default paramater

..........................

const calculatePay = (yearSalary , bonus = {

teamBonus : 0,

employeeBonus : 0

})

=> {

return yearSalary + bonus.teamBonus + bonus.employeeBonus;

}

console.log(22000 , {teamBonus : 1000 , employeeBonus : 1000} );

output : 24000

console.log(22000);

output : 22000

..........................

lambda funtions

..........................

const numbers = [1, -1, 2, 3];

filter

``````

const filtered = numbers.filter(funtion(num) {

return num >= 0;

});

console.log(filtered)

output : [1,2,3]

map

```

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

return {value : num};

}

);

can also be written as

const items = numbers.map(num => {

return {value: num};

}

);

console.log(items);

output :value : 1

value : -1

value : 2

value : 3

if u want to return an object without using the return keyword

const items = numbers.map(num => ({value: num}) );

cannot be written as

const items = numbers.map(num => {value: num} );

because {} is used to represent code block we must encapsulte using()

on the other hand

const items = numbers.map(num => num\*2 );

console.log(items);

output : 2

-2

4

6

because num\*2 evaluates to a value which will be returned automatically

.....................

class

.....................

use PascalNotation

need to export classes in different modules and import them back in the package that we want to use the exported class

class Person{

constructor(name){ //no need to declare members like in java

this.name = name;

}

walk(){

console.log("walk");

}

}

class Teacher extends Person{

constructor(name , degree){

super(name);

this.degree = degree;

}

teach(){

console.log("teach");

}

}

const teacher = new Teacher("Mosh" , "MSc");

....................

Named and default exports

....................

Named export

````````````

use only export

export class Teacher extends Person{ ... }

and import like this

import { Teacher } from ./Teacher;

Default export

``````````````

use default export

export default class Teacher extends Person{ ... }

and import like this

import Teacher from ./Teacher;

..........................

oops

..........................

//constructor function

function circle(radius){

this.radius = radius;

let defaultLocation = { x:0 , y:0 }; //scopes the variable to the function, therefore behaves like a private variable

let computeOptimumLocation = function(factor){

// ...

}

this.draw = function(){

computeOptimumLocation(0.1);

//defaultLocation is accessible

//this.radius is accessible and can only be reference using this

};

Object.defaultProperty(this , 'defaultLoaction' , {

get: function() { //line 1

return defaultLocation;

}

set: function(value) { //line 2

defaultLocation = value

});

}

const circle = new circle(10);

circle.draw();

circle.defaultLocation = 1; //line 2 will be triggered

circle.defaultLocation !error //cannot be accessed since defaultLocation is declared using let

....................................................................................

React

....................................................................................

-Not a framework like Angular,

-Library

DOM element

```````````

- Used in browser

- Real DOM

- used by browser

- costly

React element

`````````````

-

render(){

}

- virutual DOM

- When a small piece of element is changed, it compares itself with real DOM and updates it

- cheaper than Real DOM

index.html

``````````

<div id = "appRoot" ></div>

-application start point

create a .js file example

'use strict';

function LikeButton() {

const [liked, setLiked] = React.useState(false);

if (liked) {

return 'You liked this!';

}

return React.createElement(

'button',

{

onClick: () => setLiked(true),

},

'Like'

);

}

const rootNode = document.getElementById('appRoot');

const root = ReactDOM.createRoot(rootNode);

root.render(React.createElement(LikeButton));

-manifest.json - has meta data attributes

Babel

`````

Compiler to convert React code to javascript code that the browser can understand

index.js

````````

ReactDOM.render(element , document.getElementById('root'));

Explanation of the above statement

- ReactDOM gets a reference to document.getElementById('root') and renders element inside that reference

state = {}

``````````

-property in React.Component

-special property

-basically it is an object that has all the data that the component needs

class{

render{

(

this.handleIncrement = {handleIncrement()};

className = "Name"

onClick(){() => this.handleIncrement(product)}

)

lifecycle hooks cannot be used inside stateless functional components

lifecycle hooks can be used only inside class components

Class components must start with Uppercase or else it will be identified as a tag

JSX

```

const element = <h1>Hello, world!</h1>;

(<h1>Hello, world!</h1>) is JSX

JSX prevents XSS-(Cross Site Scripting) attacks - https://reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks

element is immutable

npm

---

package manager

used to install third party apps

------------------------

Before deploying to a live website, make sure to replace development.js with production.min.js! Development builds of React provide more helpful error messages, but slow down your website a lot.

The Babel <script> compiler is fine for learning and creating simple demos. However, it makes your website slow and isn’t suitable for production. When you’re ready to move forward, remove the Babel <script> tag and remove the type="text/babel" attribute you’ve added in this step. Instead, in the next section you will set up a JSX preprocessor to convert all your <script> tags from JSX to JS.

------------------------

React components are JavaScript functions that return markup

React component names must always start with a capital letter

HTML tags must be lowercase

Adding styles

-------------

In React, you specify a CSS class with className

<img className="avatar" />

/\* In your CSS \*/

.avatar {

border-radius: 50%;

}

<img id="avatar" />

/\* In your CSS \*/

#avatar {

border-radius: 50%;

}

<img class = "avatar" />

/\* In your CSS \*/

img {

border-radius: 50%;

}

<button onClick={handleClick}>

Click me

</button>

Notice how onClick={handleClick} has no parentheses at the end! Do not call the event handler function: you only need to pass it down. React will call your event handler when the user clicks the button.

export default function Square() {

return <button className="square">X</button>;

}

The export JavaScript keyword makes this function accessible outside of this file

The default keyword tells other files using your code that it’s the main function in your file

index.js - bridge between the component you created in the App.js file and the web browser

.......................

import {StrictMode} from 'react';

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

import './styles.css';

import App from './App';

Lines 1-5 brings all the necessary pieces together:

React

React’s library to talk to web browsers (React DOM)

the styles for your components

the component you created in App.js.

The remainder of the file brings all the pieces together and injects the final product into index.html in the public folder.

.......................

React components need to return a single JSX element and not multiple adjacent JSX elements like two buttons. To fix this you can use fragments (<> and </>) to wrap multiple adjacent JSX elements

To collect data from multiple children, or to have two child components communicate with each other, declare the shared state in their parent component instead. The parent component can pass that state back down to the children via props. This keeps the child components in sync with each other and with the parent component.

state is private to a component that defines it, you cannot update the parent's state directly from child's state.

Instead, you’ll pass down a function from the parent component to the child component, and you’ll have child call that function when a child is clicked.

whenever usestate function is invoked , it will trigger a re-render of the components that use the child's state (parent) as well as its child components

Problem in passing function as a prop from parent to child component

--------------------------------------------------------------------

function Square({ value, onSquareClick }) {

return (

<button className="square" onClick={onSquareClick}>

{value}

</button>

);

}

export default function Board() {

const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick() { Instead we can give write function handleClick(i) {

const nextSquares = squares.slice();

nextSquares[0] = "X"; //always only updates 0th index of squares[]. Instead we can give write nextSquares[i] = "X";

setSquares(nextSquares);

}

return (

<>

<div className="board-row">

<Square value={squares[0]} onSquareClick={handleClick} /> /Instead we can give write <Square value={squares[0]} onSquareClick={() => handleClick(0)} />

<Square value={squares[1]} onSquareClick={handleClick} />

<Square value={squares[2]} onSquareClick={handleClick} />

</div>

<div className="board-row">

<Square value={squares[3]} onSquareClick={handleClick} />

<Square value={squares[4]} onSquareClick={handleClick} />

<Square value={squares[5]} onSquareClick={handleClick} />

</div>

<div className="board-row">

<Square value={squares[6]} onSquareClick={handleClick} />

<Square value={squares[7]} onSquareClick={handleClick} />

<Square value={squares[8]} onSquareClick={handleClick} />

</div>

</>

)

}

<Square value={squares[0]} onSquareClick={handleClick} />

-no problem, handleClick in Square(child) component gets invoked only when user clicks a square button

<Square value={squares[0]} onSquareClick={handleClick(0)} />

-problem

The handleClick(0) call will be a part of rendering the board component. Because handleClick(0) alters the state of the board component by calling setSquares, your entire board component will be re-rendered again. But handleClick(0) is now a part of rendering of the board component, and so you’ve created an infinite loop:

Why didn’t this problem happen earlier?

When you were passing onSquareClick={handleClick}, you were passing the handleClick function down as a prop. You were not calling it! But now you are calling that function right away—notice the parentheses in handleClick(0)—and that’s why it runs too early. You don’t want to call handleClick until the user clicks!

To fix this, you could create a function like handleFirstSquareClick that calls handleClick(0), a function like handleSecondSquareClick that calls handleClick(1), and so on. Instead of calling them, you would pass these functions down as props like onSquareClick={handleFirstSquareClick}. This would solve the infinite loop.

However, defining nine different functions and giving each of them a name is too verbose. Instead, let’s do this:

<Square value={squares[0]} onSquareClick={() => handleClick(0)} />

() => handleClick(0) is an arrow function, which is a shorter way to define functions. When the square is clicked, the code after the => “arrow” will run, calling handleClick(0).

By default, all child components re-render automatically when the state of a parent component changes. This includes even the child components that weren’t affected by the change.

Which of these are state? Identify the ones that are not:

Does it remain unchanged over time? If so, it isn’t state.

Is it passed in from a parent via props? If so, it isn’t state.

Can you compute it based on existing state or props in your component? If so, it definitely isn’t state!