**React.js**

**Why do we need react when we already have JS?**

Ans. Because writing applications is much more easier with React.js rather than with JS. JS follows imperative programming model where we define the steps of how to achieve something, whereas React follows Declarative model where we define the end goals and react figures out the best way to achieve it.

**Components**

Components are just custom attributes which you can define in JSX code (which looks like HTML).

Your components must start with upper case.

React handles the uppercase and lower case components differently. Upper case are called as functions.

**JSX**

We use JSX to write React components. It allows to define target HTML code inside of components/ JS code.

JSX code leads to a tree-like code structure that "tells" React how the different components are related and how the UI should look like.

React then executes appropriate commands to manipulate the real DOM to reflect that target structure / code.

function Header() {

return (

<header>

<h1>React Essentials</h1>

<p>This is a react tutorial</p>

</header>

);

}

function App() {

return (

<div>

<Header />

<main>

<h2>Get started!</h2>

</main>

</div>

);

}

**Proper way to set Image source in React**

Consider this way first:

function Test() {

return (

<img src="src/assets/abc.png" />

);

}

The way src is set here is not optimal. It may get lost during the optimization process during build.

This is how to do it:

import reactImg from './assets/abc.png';

function Test() {

return (

<img src={reactImg} />

);

}

- Here, reactImg is a javascript variable pointing to the path of the image.

But it will include an automatically generated path, that will also work when you deploy react app to a server.

- Note that there are no quotes around src attribute value.

**Props**

React allows you to pass data to components via a concept called props.

Let's say when you call one component in another, you also want to pass some data.

You can define custom attributes to your custom components, which the child component then receive.

For eg:

function CoreConcept() {

return (

<li>

<img src='...' alt='...' />

<h3>Title</h3>

<p>Description</p>

</li>

);

}

function App() {

return (

<div>

<CoreConcept />

<CoreConcept />

<CoreConcept />

);

}

Now, let's say we want to display different titles and description.

We can pass these using props. The custom attributes we will be passing is basically props only.

import componentImg from './assets/component-img.png'

function CoreConcept(props) {

return (

<li>

<img src={props.image} alt={props.title} />

<h3>{props.title}</h3>

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

</li>

);

}

function App() {

return (

<div>

<CoreConcept title="Test Value" description="This is a test value" image={componentImg} />

<CoreConcept />

<CoreConcept />

);

}

In normal JS function, we define same number of arguments in a method which are passed.

But in JSX, all the parameters are received in one parameter only, which we typically call as props. But we can give it a custom name.

props parameter will be set by React as it will execute the function.

Remember, we are not calling these component function ourselves, instead we are using them as HTML elements and React is calling them.

So, React converts all the values we pass to an object, which will be received by the child component.

**Get Rid of props keyword**

You can destructure the props parameter received in CoreConcept component, as long as you use the same names.

You then don't need to use props keyword everytime.

This is how the CoreConcept would look like:

function CoreConcept({image, description, title}) {

return (

<li>

<img src={image} alt={title} />

<h3>{title}</h3>

<p>{description}</p>

</li>

);

}

**Children Props**

You have a choice whether to send props as attributes or children. Children props are what is set automatically by React.

The children prop is a special prop that's automatically provided to every component function. It contains the wrapped content as a value.

export default function App() {

return (

<section id="examples">

<h2>Examples</h2>

<menu>

<TabButton>Components</TabButton>

<TabButton>JS</TabButton>

</menu>

</section>

)

}

export default function TabButton({children}) {

return <li><button>{children}</button></li>

}

This method of having one component in another is called as component composition.

Note - while destructuring, you can only use "children" name in the method parameter.

All other props can be named whatever we want (but it should match what we are passing in).

**Reacting to Events/Event Handlers**

React provides us with Prop Functions which usually start with "on" keyword. Also called as event handlers.

They determine what action should happen when an event is fired. Could be a button click or change in text.

Event propagation in react follows the same two phase process as native DOM events: the capturing phase and the bubbling phase. (In order to understand about events, refer to another note on DOM and Events).

**onClick**

There are several ways you can handle this event in differnet languages like HTML, JS, jQuery and React.

Although there are syntactical differences, the underlying concept of handling click events remains consistent.

This is the the TabButton component would look like with an event handler:

export default function TabButton({children}) {

function handleClick() {

alert("clicked");

}

return (

<li><button onClick={handleClick}>Component</button></li>

)

}

Note - You have to pass a function pointer within curly braces.

**Synthetic Events in React**

A synthetic event is a cross-browser wrapper around the browser's native event.

React implements synthetic events that brings consistency and high performance to React apps and interfaces.

**Passing function as values to Props**

export default App() {

    function handleSelect() {

        alert("Select called");

    }

    return (

        <TabButton onSelect={handleSelect}>Components</TabButton>

        <TabButton onSelect={handleSelect}>Test</TabButton>

    );

}

export default function TabButton({children, onSelect}) {

    return (

        <li><button onClick={onSelect}></button></li>

    );

}

This maybe needed when you want to control from parent component how child component behave. Basically we have passed in a callback function here.

Infact, this is how most of the applications will be written.

Let's say you want to standardize the Button component. But every Button is supposed to do something different on its click depending on which Parent component called it.

So the only way to strandardize this button component is to pass a callback function to the child component.

How to pass custom arguments to these functions?

Ans - use arrow functions while assigning

const handleSelect = (arg) => {

    alert(arg);

}

export default App() {

    return (

        <TabButton onSelect={() => {handleSelect('components');}}>Components</TabButton>

        <TabButton onSelect={() => {handleSelect('state');}}>State</TabButton>

        <TabButton onSelect={() => {handleSelect('prop');}}>Prop</TabButton>

    );

}

export default function TabButton({children, onSelect}) {

    return (

        <li><button onClick={onSelect}></button></li>

    );

}

Note – React Event handlers like onClick only work on html tags, and not on the custom tags created for React Components.

So if you do this - <TabButton onClick={() => {handleSelect('prop')}}>Prop</TabButton>

Nothing would happen on clicking because <TabButton> accepts onSelect and set it on onClick of <button> inside it.

**Define Custom functions inside the App function or outside?**

Ans - Depends

**Defining Functions Inside the App Component**

**Advantages:**

Access to Component State and Props: Functions defined inside the component have access to its state and props, making it easy to work with component-specific data.

Encapsulation: Functions are scoped to the component, providing better encapsulation and reducing the risk of namespace collisions.

Simple Lifecycle Management: Functions defined inside the component are recreated on every render, which can be useful if they rely on the component's state or props.

**Defining Functions Outside the App Component**

**Advantages:**

Reusability: Functions defined outside the component can be reused across different components.

Performance: Functions defined outside the component are not recreated on every render, which can lead to slight performance improvements.

Clean Code: Keeping component functions outside can make the component code cleaner and more readable.

**When to Choose Which**

**Inside the Component:** Use this approach when the function directly interacts with the component's state or props, or when the function is specific to the component's logic and not reused elsewhere.

**Outside the Component:** Use this approach when the function is independent of the component's state and props, can be reused across multiple components, or for utility/helper functions.

**Memoization with useCallback**

To avoid recreating functions on every render while keeping them inside the component, you can use the useCallback hook

**State**

Let's say you have a var/let/const in App() and you're printing it in a div, and also log the value in console.

When you change its value, you'll see the new value printed in the console, but the UI will have previous one.

This is because changing a variable doesn't re-render the UI.

We need special variables called State to trigger the re-render. State comes with a method which asks React to re-render the UI.

import React from 'react';

import { useState } from 'react';

export default function App() {

    const [selectedTopic, setselectedTopic] = useState('Please select a topic');

    function handleSelect(selectedButton) {

        setselectedTopic(selectedButton);

        console.log(selectedTopic) // note - you'll still see the previous topic //and not the currently selected one. This is because state change and re-render //will happen at the end of this function

    }

    return (

        <TabButton onSelect={ () => handleSelect('component') }>Component</TabButton>

        <TabButton onSelect={ () => handleSelect('js') }>JS</TabButton>

        <TabButton onSelect={ () => handleSelect('state') }>State</TabButton>

        {selectedTopic}

    );

}

Note - the useState function can only be declared inside a component.

They should be at the top level, not inside any other function or if/for block.

**Conditional Rendering (applies to both html code and style classes)**

3 ways

- Ternary Operator

- &&

- if block

**&& Example**

import { useState } from 'react';

export default function Players({name, symbol}) {

    const [isEditing, setIsEditing] = useState(false);

    function handleClick() {

        setIsEditing(true);

    }

    return (

        <li>

            <span>

                {!isEditing && <span>{name}</span>}

                {isEditing && <input type="text"></input>}

                <span>{symbol}</span>

            </span>

            <button onClick={handleClick}>Edit</button>

        </li>

    );

}

**If Block**

In React, you can assign JSX code to variables.

Hence in this code, msg is conditional depending on whether show is true or false.

import React from 'react';

export default function App() {

    const [show, setShow] = React.useState(false);

    let msg = <button onClick={() => setShow(true)}>Delete</button>

    if (show) {

        msg = <div data-testid="alert" id="alert">

          <h2>Are you sure?</h2>

          <p>These changes can't be reverted!</p>

          <button onClick={() => setShow(false)}>Proceed</button>

        </div>;

    }

    return (

      <div>

        {msg}

      </div>

    );

}

**Display a list dynamically**

Use map function to display an object dynamically. Search google for examples.

**Fragments in React**

The following JSX code will report error:

export default function App() {

    return (

        <Header></Header>

        <main></main>

    );

}

Error - Atleast 1 parent must be there.

This error happens because App is essentially a JS function and it must return 1 value only, like any other function.

So, we must wrap them in a div to make this work.

But, this leads to extra html tags on the page.

So, the alternate is to wrap them in a <Fragment>

import { Fragment } from 'react';

export default function App() {

    return (

        <Fragment>

            <Header></Header>

            <main></main>

        </Fragment>

    );

}

Modern React version go beyond one level and now we don't need to import <Fragment> and directly use:

<>

    <Header></Header>

    <main></main>

</>

**When should you split component?**

Be careful to understand when and how to split your components.

Like in our App component, we may be referring to 2 Custom React Components.

Let's say the state of App() changes, both will get re-rendered. But we may not want that.

**Forward Props to child**

Sometimes, while calling your custom react components, you may want to apply the html attributes (passed in as props) directly to another html tag in the child component.

You can do this using spread/rest operator.

export default function App() {

    return (

        <Input type="text" placeholder="Initial text" />

    );

}

export default function Input({props}) {

    return (

        <input {...props}></input>

    );

}

**JSX Slots**

Expanding in the concept above, we can also pass jsx elements also inside the prop which gives us a lot of flexibility while building UI's.

So, we can do something like:

export default function App() {

    return (

        <Button buttons={

            <>

                <TabButton

                  isSelected={selectedTopic === 'components'}

                  onClick = { () => handleSelect('components') }

                >

                    Components

                </TabButton>

                <TabButton

                  isSelected={selectedTopic === 'JSX'}

                  onClick = { () => handleSelect('JSX') }

                >

                    JSX

                </TabButton>

            </>

        }

    );

}

export default function TabButton({children, buttons}) {

    <>

        <menu>

            {buttons}

        </menu>

        {children}

    </>

}

**public vs assets for image storage**

Content on public is available directly through URL access. But assets one is not available.

**Updating State based on Previous/Old value**

You should know that whenever we update the state using its corresponding 'set...' method, the state is not changed immediately.

It's changed after the current handler function is complete. Therefore, the value of state is going to be false in both statements below:

import { useState } from 'react';

export default function Players({name, symbol}) {

    const [isEditing, setIsEditing] = useState(false);

    function handleClick() {

        setIsEditing(true);

        console.log(isEditing); // false

        setIsEditing(false);

        console.log(isEditing); // false

    }

    return (

        <li>

            <button onClick={handleClick}>Edit</button>

        </li>

    );

}

Now, let's imagine a scenario where you want to toggle the value of isEditing on every click of the button.

In this case, if you do this:

function handleClick() {

    setIsEditing(!isEditing);

}

Then you may get unexpected error. Because what if in the same function you do setIsEditing(!isEditing) again?

You would expect the isEditing value to be false at the end your function execution and expect React to not re-render the UI.

But that won't happen because as I mentioned above, the first call of setIsEditing would not be called until at the end of the function, and React will re-render the UI in this case. Essentially, in both the calls of setIsEditing, it's the 'false' value which is passed.

If you want the latest value of isEditing to be passed to setIsEditing (true for the 2nd call), then pass the value in parameters as a function like below:

function handleClick() {

    setIsEditing( (editing) => !editing ) // false to true

    setIsEditing( (editing) => !editing ) // true to false

}

No re-rendering occurs in this case. React's team suggests to do it in this way.

Basically React will inject the parameter itself here. Its value would be the current value of the object.

**Two way binding**

Two way binding in React can be achieved by calling the useState method of a variable whenever the input/textarea value change (using onChange event).

Such a component is called a Controlled input component. Controlled input component are a React pattern in which we store the input element's value in a state,

and everytime state changes, we update the user display.

This can be done:

import { React, useState } from 'react';

export default function App() {

    const [feedback, setFeedback] = useState(); //will set initally empty value

                                                //otherwise you may set a placeholder value, or any value received in a prop

    function handleFeedback(event) {

        setFeedback(event.target.value);

    }

    return (

        <>

            <h2>Please share your feedback</h2>

            <p>

                <label>Your feedback</label>

                <textarea value={feedback} onChange={handleFeedback} />

            </p>

            {feedback} //will update as soon as you type

            <Review feedback={feedback} />

        </>

    );

}

**Updating States in immutable way**

Objects & Arrays (which are objects only in JS) are reference values in JS.

You should therefore not mutate them directly - instead create a deep copy first.

So, let's say you have this:

const student = {

    name: "Test Value",

    address: "NY",

    age: 26

}

const [student, setStudent] = useState(student);

function handleStudent(nameValue) {

    setStudent(student.name);

}

But this is not the recommended way and could lead to bugs and side effects.

Objects don't hold values but references. So when you modify like above, you've modified the object even before the schedules update by React.

We should always update objects in immutable way, which means we should create a new object and then update those.

This is how we can do this:

function handleStudent(nameValue) {

    setStudent( stud => (...student, name: nameValue))

}

In order to understand why we need to do this, we have to understand how JS copies objects.

Reference - <https://dpericich.medium.com/how-to-update-complex-state-objects-in-react-c8cad3363566>