HACKSERIES 01
React.JS

What is React.JS ?

React or React.js, is a free, open-source JavaScript library developed by Facebook for building user interfaces, particularly for single-page applications. It enables developers to create reusable UI components, enhancing efficiency and maintainability. React utilizes JSX, a syntax extension that allows mixing HTML with JavaScript, facilitating dynamic data rendering without full page reloads. Its component-based architecture promotes modularity and scalability in web development

What is JSX ?

JSX, or JavaScript XML, is a syntax extension for JavaScript, primarily used with React to define user interfaces. It allows developers to write HTML-like code within JavaScript, making it easier to create and visualize UI components. JSX is transpiled into regular JavaScript function calls, enabling dynamic rendering of elements. It supports embedding JavaScript expressions and requires all tags to be properly closed

Important Concepts

Components, JSX, Props, State, Lifecycle Methods, Hooks, Context API, Fragments, Controlled Components, Event Handling

React Elements

React elements look just like HTML, in fact they render the same equivalent HTML elements.

```
<h1>My Header</h1>
<button>My Button</button>
<ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
</ul>
```

React Element Attributes

React elements have different attributes compares to their HTML counterparts. Since JSX is still javascript, we use camelcase. Also, class is a protected keyword in javascript (creating classes) so the HTML class attribute in JSX is className.

```
<div className="my-container">
    <img className="my-container-image" src="my-image.jpeg"/>
</div>
```

React Elements Embedded Javascript

The power of JSX is that it's javascript and HTML. This means you can write javascript to render different attributes directly in your javascript using curly braces {}

React Fragments

React has a special element called a fragment. It's a special element that doesn't actually render into the DOM, but can act as a parent to a group of elements.

```
import { Fragment } from 'react'

<Fragment>
    <h1> My H1 </h1>
    <p> My Paragraph </p>
</Fragment>
```

React Components

Components are the building blocks of your web application. We use them to organize groups of React elements together so they're reusable. There are two kinds of components, class components and functional components but functional components are the de facto standard today. They both follow the same two rules:

1. Component names must be capitalized i.e. MyComponent instead of myComponent

1. They ust return JSX, more specifically one parent level JSX element (more on this later).

Functional Components

Functional components are just javascript functions that return JSX.
Here's how you create a functional component using function declaration:

```
const MyComponent = () => {
    return <h1>My Component</h1>
}

const MyOtherComponent = () => {
    return (
        <div>
            <MyComponent />
            <p> Sample Text </p>
        </div>
    )
}
```

Component Props

We can pass data to our components through custom attributes on the component element. We can choose any name for the attribute as long they don’t overlap with the existing general element attributes (i.e. className, styles, onClick etc.). These properties are then grouped into an object where the attribute name is the key, and the value is the value. Here we are passing a prop title from the App component to our component MyComponent.

```
const MyComponent = (props) => {
    return <h1>{props.title}</h1>
}

const App = () => {
    return (
        <MyComponent title="Hello World" /> // Props == { title: "Hello World" }
    )
}
```

The Children Prop

All component have a special prop called children. Any data (usually components and react elements) sitting between the opening and closing tags of the component get passed in as children.

```
const Greeting = ({ children }) => {
    return children //<h1> Hello World! </h1>
}

const App = () => {
    return (
        <Greeting>
            <h1>Hello World!</h1>
        </Greeting>
    )
}
```

Conditional Rendering

Since our components are written in JSX which is just javascript, we can conditionally render different things with javascript. A basic example is to use an if statement in our functional component.

```jsx
const Greeting = ({large}) => {
    if(large) {
        return <h1> Hello World! </h1>
    }
    return <p> Hello World! </p>
}

const App = () => {
    return (
        <div>
            <Greeting large={true}/> // <h1> Hello World! </h1>
            <Greeting large={false}/> // <p> Hello World! </p>
        </div>
    )
}
```

Context

If we had some data values we wanted to share across multiple sibling or nested child components, we would have to lift that data up to a commonly shared parent and needlessly drill it down through multiple components and their props.
Here, we have to pass the text value from the App component all the way through Parent A and ChildA just so GrandChildA can receive it, even though ParentA and ChildA don't need the text other than to pass it down. The same is true for ParentB in order to get the text value to ChildB. This is called prop drilling.
We can't move the text value down the tree since both GrandChildA and ChildB need it, and App is the lowest common parent between them. This makes our code extremely brittle since moving and the components serving to pass the text prop needlessly complex.
We can solve this with React Context which allows us to lift the data into a special component called Context that allows any of it's children no matter where they are to access it's values without the need for prop drilling.

We need the createContext function from React and pass it the initial value we want to share. It returns us back an object that contains two components:

1. the Provider component which we wrap around the portion of the component tree we want to access the value.
2. The Consumer component which has access to the values from the created context which we place in any component that needs the value.

```
import { createContext } from 'react';

const TextContext = createContext('');

const GrandChildA = () => { // I need text
    return (
        <div>
            <h1> Grand Child A </h1>
            <TextContext.Consumer>
          {text => <p> {text} </p>}
        </TextContext.Consumer>
        </div>
    )
}

const ChildA = () => { // I don't need text
    return (
        <div>
            <h1> Child A </h1>
            <GrandChildA />
        </div>
    )
}
```

```jsx
const ParentA = () => { // I don't need text
    return (
        <div>
            <h1> Parent A </h1>
            <ChildA />
        </div>
    )
}

const ChildB = () => { // I need the text
    return (
        <div>
            <h1> Child B </h1>
            <TextContext.Consumer>
          {text => <p> {text} </p>}
        </TextContext.Consumer>
        </div>
    )
}

const ParentB = () => { // I don't need text
    return (
        <div>
            <h1> Parent B </h1>
            <ChildB />
        </div>
    )
}

 const App = () => {
   return (
     <TextContext.Provider value="Hello World">
              <ParentA />
              <ParentB />
     </TextContext.Provider>
   )
 }
```

Hooks

Hooks were introduced in React version 16.8 as a way to extend additional functionality into functional components. Previously this functionality was only available to class components, but through hooks we can super charge our functional components!
To better understand hooks, we need to understand the React component lifecycle. There are three main phases of any React component:

1. The mounting phase when a component is created and inserted into the DOM. This is the initial render and only happens once in a components lifecycle.
2. The updating phase is when a component re-renders due to updates. This happens either due to prop changes or state changes (more below).
3. The final phase is the un-mounting phase, when a component is removed from the DOM.

Hooks are normally called at the top of our components.

useState

useState hook allows us to store values scoped to a component. Any changes to those values will cause the component and any of it's child components to re-render.
As mentioned above, components re-render in the updating phase (2) due to prop changes and state changes. State is data stored inside of a component that can be updated/changed. When this state data changes, this will trigger a re-render of the component. While we can store and change data in a variable, those changes will not trigger a re-render. With the useState hook, it does allow us to trigger re-renders on changes to that data.

```
import { useState } from 'react';

const MyComponent = () => {
    const [value, setValue] = useState(initialValue);
}
```

useEffect

useEffect is a hook that allows us to create side effects in our functional components.
useEffect takes two arguments:

1. The first argument is a callback function called the effect function that contains the side effect code we want to run.
2. The second argument is an array called the dependency array which contains values from outside the scope of the effect function. Whenever one of these values changes, useEffect will run the effect function.

```
import { useEffect } from 'react'

const MyComponent = () => {
    useEffect(() => {
        // side effect code here
    }, [// dependencies go here]);
}
```

The effect function will run:

1. Once when the component mounts.
2. Whenever any value in the dependency array changes.

useRef

useRef is a hook that stores a value in a component like useState except changes to that value won't cause the component to re-render.
It accepts one argument as the initial value and returns a reference object.

```
import { useRef } from 'react';

const MyComponent = () => {
    const ref = useRef(initialValue);

    // ...remaining component code
}
```

useCallback

The useCallback hook is a performance improvement hook that prevents functions from being needlessly recreated between re-renders.
Whenever a component renders or re-renders, any functions that are created are recreated. In the component below, we create a hideUser function that we use in the button.
The useCallback hook signature takes two arguments:

1. The function we want to persist between re-renders.
2. A dependency array containing values that tells useCallback when to recreate the function when any of them change.

```
import { useMemo } from 'react'

const MyComponent = () => {
    const computedValue = useMemo(
        () => { //...computationally expensive function },
        [//dependencies]
    )
}
```

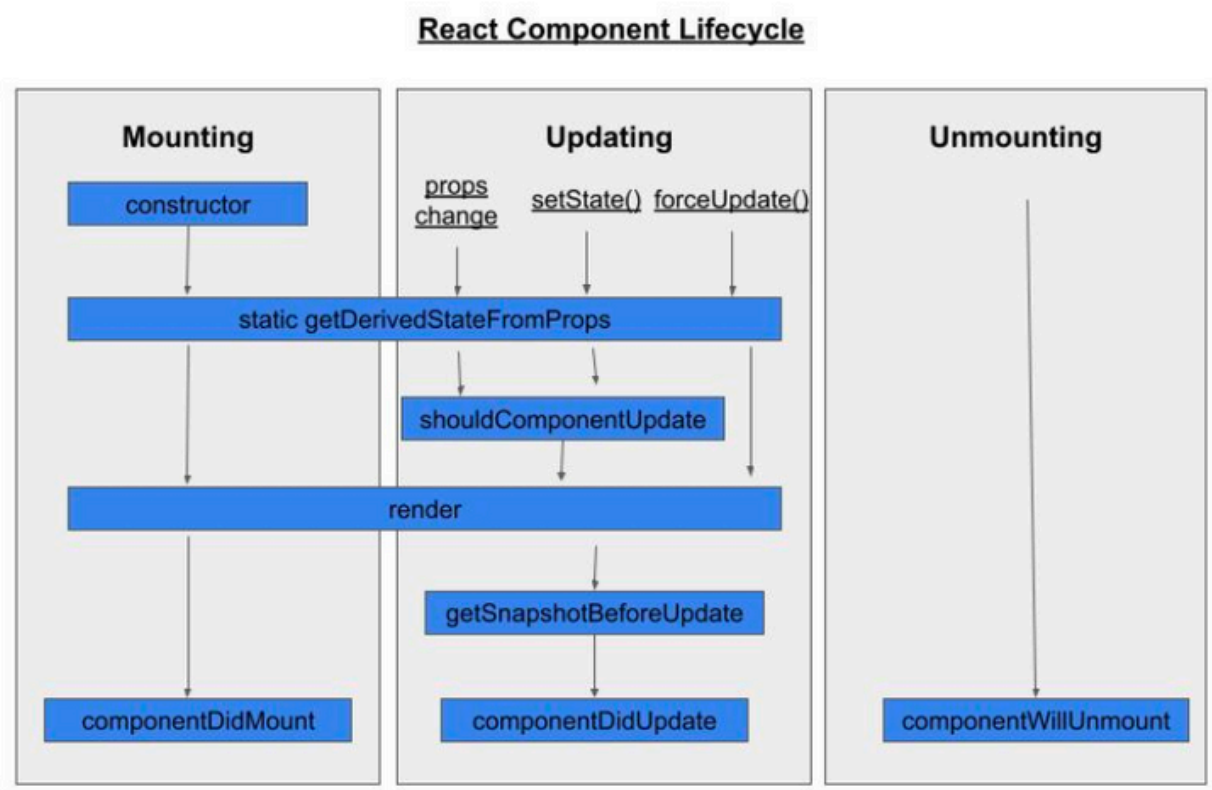
useContext

useContext allows us to access Context without need to use it’s Consumer component.

```
import { createContext } from 'react';

const TextContext = createContext('');

const ChildA = () => {
    return (
        <div>
            <TextContext.Consumer>
          {text => <p> {text} </p>}
        </TextContext.Consumer>
        </div>
    )
}

const ParentA = () => {
    return (
        <div>
            <h1> Parent A </h1>
            <ChildA />
        </div>
    )
}

const App = () => {
  return (
    <TextContext.Provider value="Hello World">
            <ParentA />
    </TextContext.Provider>
  )
}
```

Lifecycle Methods

Class components also have methods that hook into React's rendering and re-rendering cycles called lifecycle methods. Many methods are now deprecated or considered UNSAFE as the React team is pushing forward with functional components + hooks. This cheatsheet will only reference the commonly used ones.



Conclusion :

React.js simplifies building dynamic web applications with its component-based design and efficient tools like hooks. Its evolving ecosystem ensures scalability and adaptability. Mastering React unlocks endless opportunities for creating seamless, interactive user interfaces. Happy coding! 🚀