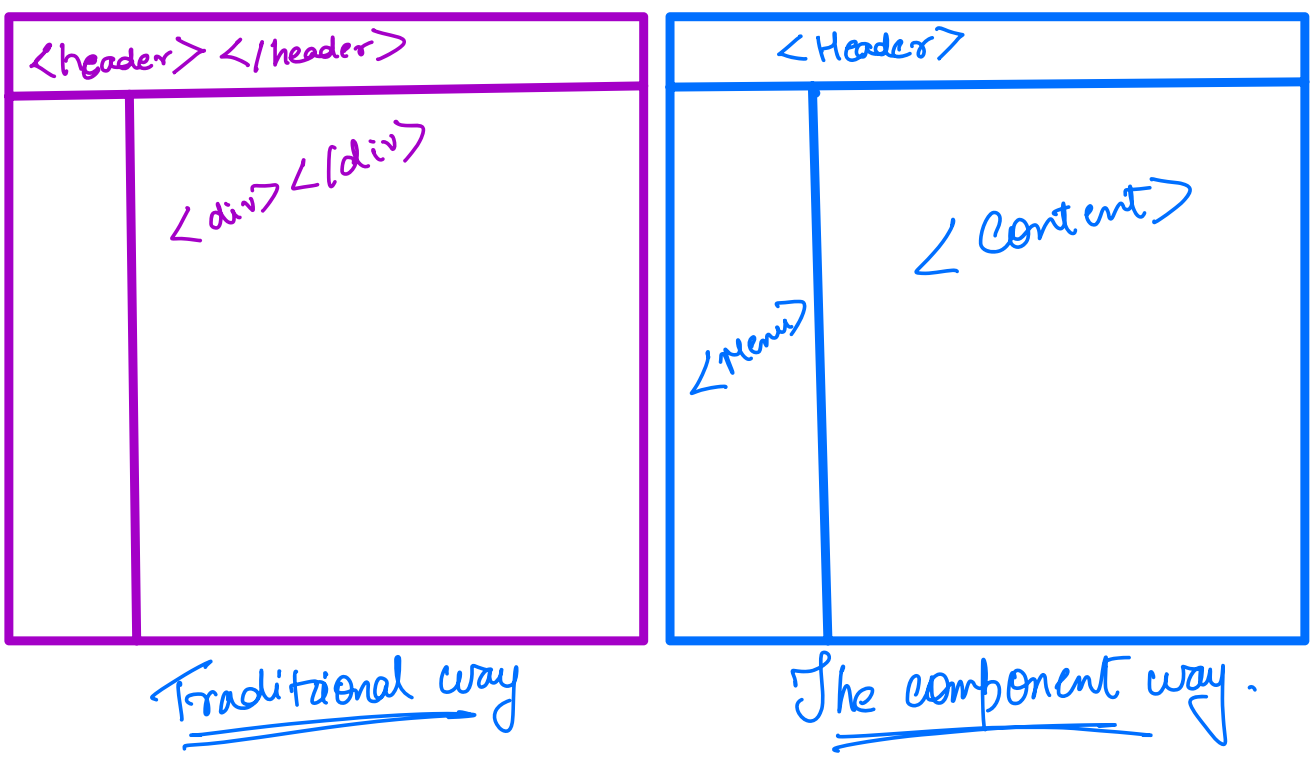
React Theory

The notes is divided into the following topics -:

a. Component Architecture
b. What is JSX?
c. Class based components / Functional components
d. Lifecycle of a React component
e. React hook part 1 - useState()
f. State and props
g. Virtual DOM

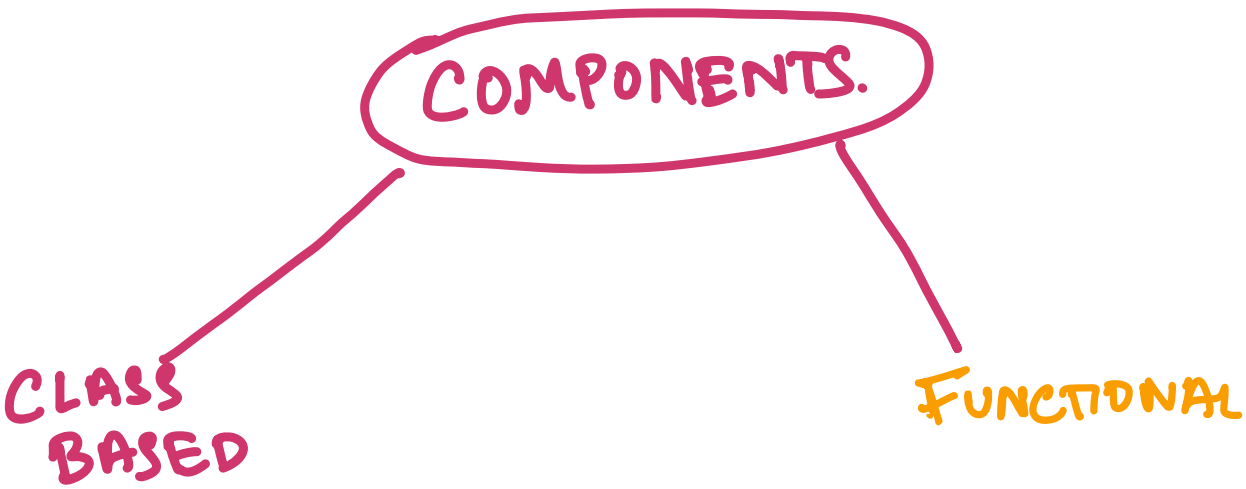
In React, we think about our web applications in components.

A component can be defined as an individual, isolated piece of UI which has its own structure, styling and logic. Basically, its a reusable piece of HTML+CSS+JS.



These individual components combine together to create a dynamic web application. Each of these components have the ability to function on their own, but often communicate with each other via states and props (which we will read later about).

Any UI feature that you create in React will be built with the mental model of a component in our heads.

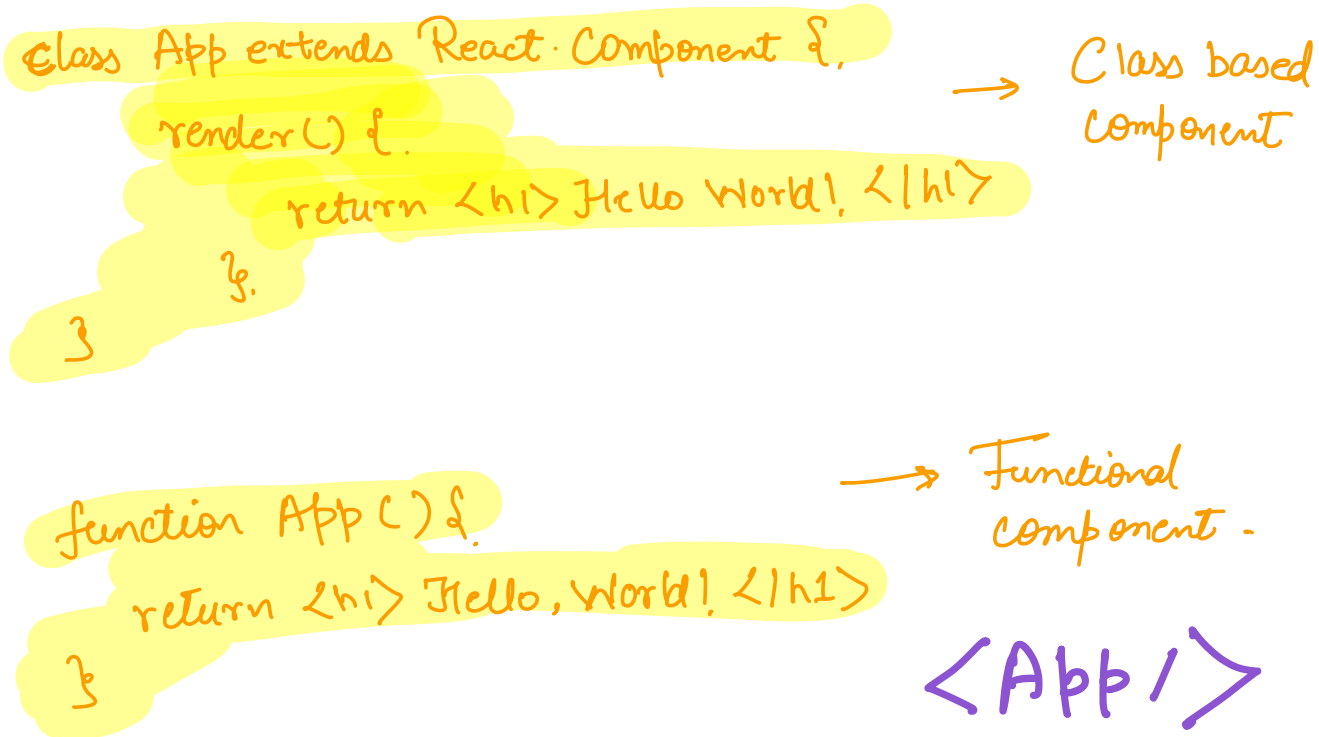


What is JSX?

Simply put, JSX is the HTML sort of structure that gets returned from inside of a component, whether it is class based or functional. The difference between traditional HTML and JSX stems out as -:

> I. One cannot use the word **class** inside of JSX, instead use **className**.
>
> II. You can write Javascript expressions inside of JSX which will compute at runtime.

Behind the scene, React is running React.createElement() function whenever we return some JSX from inside of a component.



We will be dealing with functional components from now on.

Class based components are **stateful,** while functional components are called **stateless** in nature.

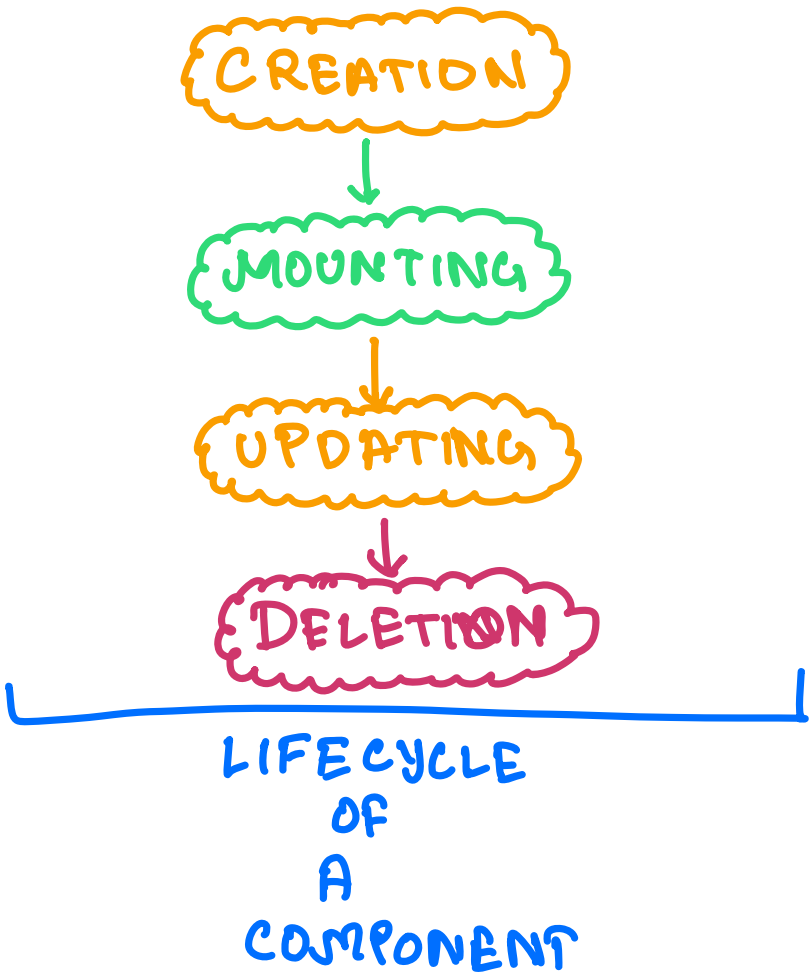
Functional components provide a much more intuitive and convenient way of defining our components. Both regular function as well as arrow function can be used to define a functional component.

One thing to always note for both the types of components is that they must always return only one single JSX element. If you wish to add multiple JSX elements in your component, wrap them up under a single JSX element like a container HTML tag, <></> or React.Fragment.

Components can also be nested within each other, like in HTML. The concept of parent-child components also prevails here.

A component goes through multiple phases in its lifetime. Class based components provide a way to tap into these phases of a component via

lifecycle methods. These methods can be executed during the execution of these phases.



In the context of a functional component, we don't have lifecycle methods. Rather we have hooks. Hooks are simple JS functions which 'hook into' these lifecycle methods, and provide a similar interfacing with a component.

There are multiple hooks in the latest versions of React, but we will be studying a few major of them.

States and props

React's engine makes our UI reactive, i.e. our UI responds to data change, This feature is not possible in vanilla JS, and this makes React extremely powerful.

Because we are dealing with only Javascript inside of a component, we can define normal variables as part of the component data. But these static declaration are not reactive.

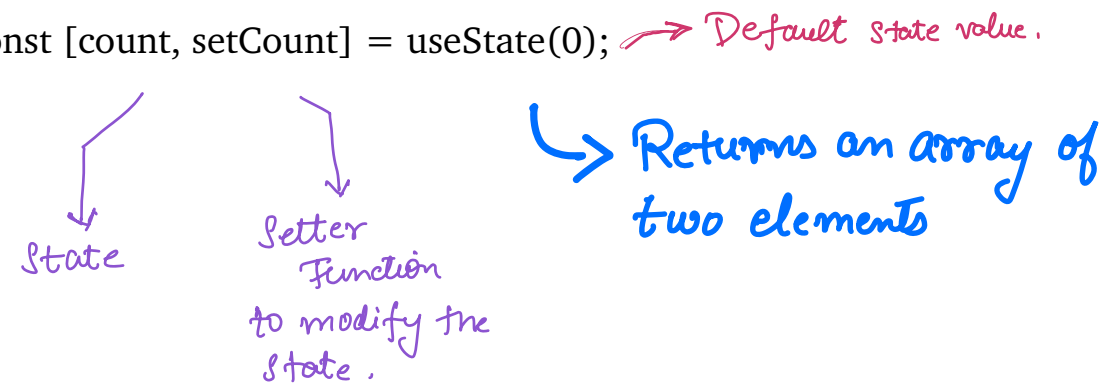
In order to make a reactive data inside of a component, we define a state.

States are internal data of a component, both static as well as reactive.

Props are component states which are passed from a parent component to a child component.

In order to create a reactive state data, we use the **useState()** hook.

```
const [count, setCount] = useState(0);
```



In order to pass state data as props to child component,

```
<Child val={count} />
```

You can pass multiple props to a child component. The props passed to a child component are grouped inside of a prop object.

```
function Child(props) {
    const {val} = props;
    return <p>{val}</p>
}
```

Props are immutable, while states are mutable in nature. This means that we cannot change the value of the props being passed down by the parent component inside of the child component.

Understanding the render cycle of React

React works on the foundation of *reactivity to data. This means that whenever there is a state or props change in the UI component, the React engine will try to re-render the component to sync up with the updated data of the component.*

The state or props change can occur due to any reason, be it user triggered event like a button click, or any external event like fetching data from an api call.

During the re-rendering cycle, the entire function body of the component re runs and the updated value of the states or props is propagated through the JSX.

In React.StrictMode, the re-rendering sequence happens twice.

If neither the states nor the props of any component has changed, React will not trigger that component's re-rendering. But there is a catch to this as well.

``If the state of the parent changes, all of the child components inside of it will re-render, even if the states and props of the child components did not change.``

React does this to ensure that no components are out of sync with their data. This is a safety mechanism to ensure coherence amongst the states of a component.

Understanding Virtual DOM

The concept of virtual DOM is what makes React so efficient in DOM manipulations. We have learned previously that directly performing operations on the actual DOM is expensive, so React spawns a virtual DOM which is an exact replica of the current DOM, but it is much more efficient and speedy to perform operations on the virtual DOM.

The way React uses this virtual DOM is that, whenever any component updates, React compares the old virtual DOM with the new updated virtual DOM. Then it searches for the exact differences between the two, and updates just those nodes in the actual DOM, effectively reducing the operations on the actual DOM.