27/02/24 Episode-08 let's get classy 😊

→ We will learn about class based component.

Normal JS class.

→ For interviews
→ understanding of react will become very strong.
→ How lifecycle works.

Syntax :- our class comp. is using some properties of react.component.

```
class Name extends React.Component {
};
```

keyword

component name

This will tell react that, this class is a react component.

React.Component is a class which is given to us by React.
So import it import React from 'react';

our class based component is using some properties of React.Component class

```
class Usex extends React.Component {
  render() {
    return (
      <div className="Usex-comp">
        Hello class based component
      </div>
    );
  }
};
```

Basically our class compo. is inheriting essential functions & properties from React.Component, which are essential for a class based component to work.

This will be rendered on to UI

Class based components have a render method which returns some piece of JSX.

→ What is a class based component?

Ans A class based component is a normal JS class, which has a render method which returns a piece of JS.

When you are using class based component, don't forget to `import React from 'react';`

→ Just a small change in syntax, else everything is same as functional component.

☆ How to receive props in class based components.

→ import React from 'react';

class User extends React.Component {

constructor (props) {

A constructor of a class is called at every class instance

super (props);

Do 2 things :-
1) Initilize the required received func & methods from React.Component. 2) Receives & Stores the props object, received from parent call, and allows us to access those props using this.props properties.

}

you can also do destructuring const {name} = this.props

render () {

< div > {this.prop.name} < /div > } ;

};

→ We can access our props using this.Props only in class compo

→ using - super (props) is imp because without it we will not be able to use this.props.

☆ How to use / create State variables inside class based component

→ Remember one thing :- constructor is the best place to initilize props as well as State variables.

constructor (props) {

super (props);

This is the place where we will declare & initilize all our State variables

this.state = {

State variable 1 → count : 0, ← initial value

State variable 2 → count2 : 1,

};

};

→ Remember: If you want to do destructuring do it inside render() in class based component

```
render() {
    const { count, count2 } = this.state;
    return (
    );
}
```

Behind the scenes react is using a big single object to keep all our state variables.

→ How we do it in func. component

```
import {useState} from 'react';
const user = () => {
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(1);

    return (
        <div> {count}   {count2} </div>
    );
};
```

Above return.

→ In func. component to create state variable we use **useState HOOK.**

→ How to update state variables inside a class based component.

```
<button onClick={() => {
    this.setState({
        count: this.state.count + 1,
        count2: this.state.count2 + 1,
    });
}}>
```

→ you cannot write count++ ✗

→ For updating State variables in class based components we use this.setState() method and it takes an object as an argument.

→ Inside that object you can update your State variables as

state_variable_name : updated value.,

→ Inside that Single object you can update more than one state variable, if you want as per your need.

→ When react will encounter this.setState, it will go into argument which we have passed, it will look for the name/s of the State variable and it will only update that state variables.

→ After updating it will re-render the whole component, and it will only update that State variable on DOM.

→ You can also destructure your React.component like

```
import {component} from 'react';
class user extends component {
    render() {
        return (
    };  );};
```

→ Better defination is written in next page (1)

→ Mount : Mount means component has been loaded on the UI. Loading is same as mounting of a component.

→ As we say a component has been loaded onto the Screen. is equivalent to saying a component is mounted onto the Screen.

★ class based component

Lyfecycle methods

→ Very firstly when ever a component is initiated → then very firstly the class's constructor method is called / executed.

→ Instance / Instiation of a class basically means calling a class based component.

→ In DSA as soon as we create an object of a class, it's constructor is called.

→ Similarly here whenever we call a C.B.C, it's constructor is executed or called.

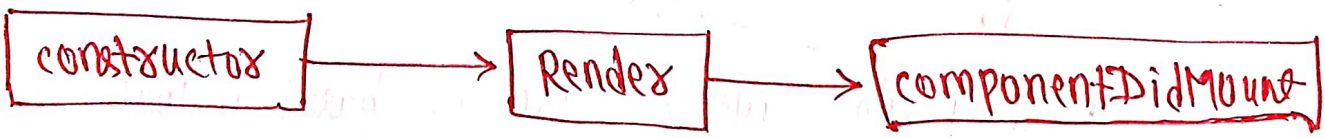
For example:

```
const comp = () => {
    return (
        <classComp />
    );
};
```

Here we are calling this class based component. As soon as react will encounter this line, then an instance of that class based component will be created.

→ After executing the constructor render() { } method gets executed.

→ After executed render() method, componentDidMount() method gets executed.

→ 

→
```
class Parent extends React.Component {
    render(){
        return (
            <childCompo/>
        );
    };
};
```

→ Flow or lifecycle of above class based component will be :

1. constructor of parent class will be called.
2. Render of parent class will be called.
3. constructor of child class will be called.
4. Render of child class will be called.
5. componentDidMount of child will be called.
6. componentDidMount of ~~child~~ parent class will be called.

→ When react will execute parent (class based component) then lifecycle methods of parent class will start executing.

→ As soon as it will encounter child (class based component) then lifecycle methods of parent class will be paused, and lifecycle methods of child (class based component) will start getting executed.

→ After completing the lifecycle methods of child (C.B.C), lifecycle methods of parent (C.B.C) will resume.

Note : constructor will be ~~executed~~ executed only once at the ~~inten~~ instentiation of the class based component.

Similarly componentDidMount will will only be executed once after completing the ~~initial~~ Mounting (Initial Rendering)

↓

Mounting means putting element into DOM.

→ ~~Loading~~ is ~~also~~ ~~same~~, and putting elements into the Dom is done just once & then react simply updates the Dom.

(a)

→ Mounting is when react renders the component for the first time and actually builds the initial DOM for from those instruction. (this DOM did not existed already)

→ A "re-render" is when React calls the function component for the ~~first time~~ again to get a new set of instructions on already mounted component.

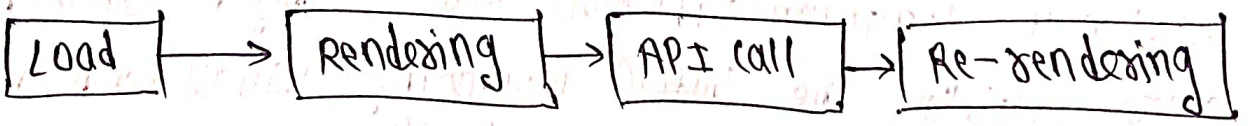
→ In a class based component, if a state changes then react calls the render() method of that component is only render method of that class based component is executed, constructor or componentDidMount does not gets executed again

→ In class based component we use componentDidMount for making API calls, as we use useEffect() in functional component

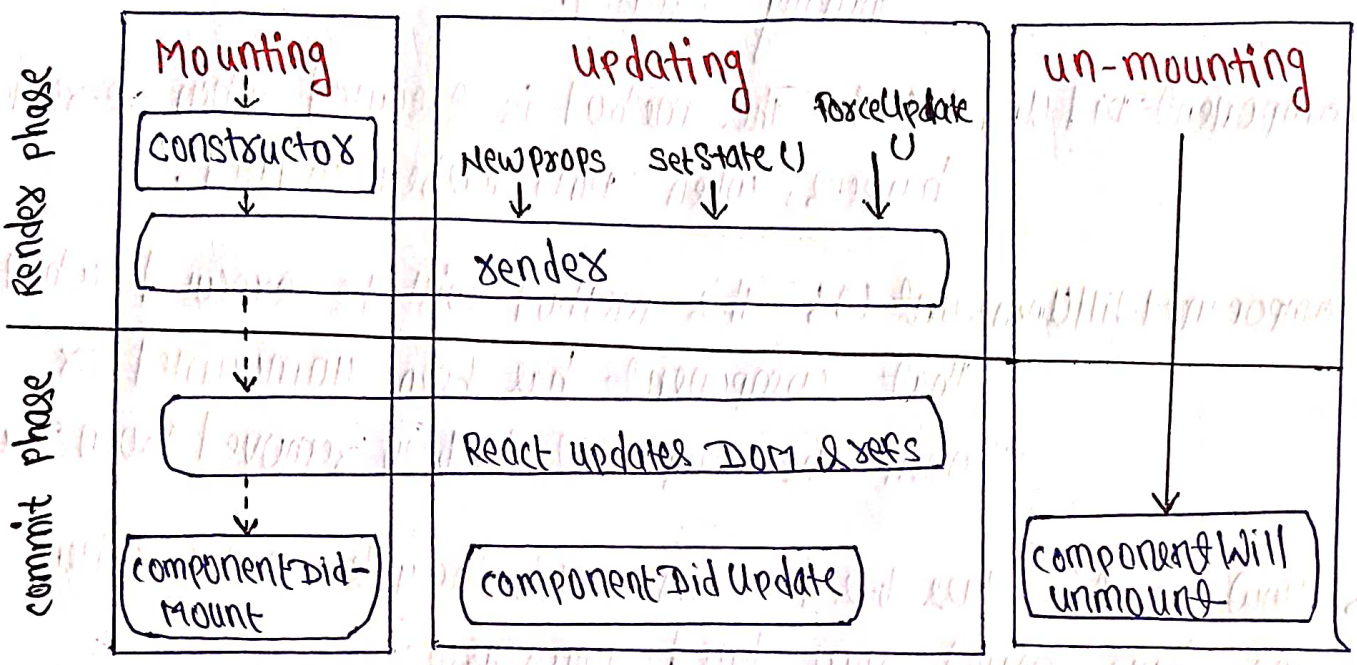
→ The reason why we use componentDidMount is that it will executed after the ~~mounting~~ of the component.

↓
Initial rendering / 1st rendering only.

component has been placed onto the DOM for the first time.



28/02/24 → React lifecycle method.



→ Disclamase :- Do dont compare the react lifecyle method of class based components to functional components.

→ React class based component's life cycle is divided into 3 parts

1.) Mounting : Putting the component onto Dom for 1st time

2.) Updating : Making changes into DOM or simply re-rendering with updated state variables.

3.) un-Mounting : Removing component from DOM, or when that component has been removed from the screen.

→ There are 2 phase in mounting and updating :-

1.) Render phase : In this phase constructor and render method of the component is executed.

2.) commit phase : In this phase, DOM is updated and componentDidMount or componentDidUpdate or componentwillunmount method is called/executed.

→ componentDidMount(): This method is executed when mounting has been completed, when DOM has been initially created.

→ componentDidUpdate(): This method is executed when re-rendering happens, when DOM has been updated.

→ componentWillUnmount(): This method will be executed when that component has been unmounted or when the component has been removed from DOM.

→ How React's class based component's lifecycle works when we have nested class based component.

```
import React from 'react';
class Parent extends React.Component {
    constructor (props) {
        super (props);
    }
    componentDidMount () {
    };
    componentDidUpdate (){
    };
    componentWillUnmount (){
    };
    render () {
        return (
            <child />
        ) <child />
};  };
```

→ When react have more than one child components ~~inside~~ inside any parent components, then it will ~~their~~ combine or batch their Rendering and commit phases.

→ Rendering phase of both child components will take place 1st and then commit phase of both child components will take place together.

→ And then commit phase of parent component will take place.
See logs in code for more better understanding.

→ But why React batches or combines the rendering phase of all the components together & commit Phase of all the components together ?

↳ Because Dom manipulation is a very expensive operation in react. So to fast our Dom manipulation Process, react basically checks what all component it want to add into Dom and the it do Dom updation only once.

→ which increase our Dom updation/manipulation process.

Understanding class based components

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props); // Call parent class constructor (if any)
    this.state = { /* initial state object */ };
  }

  // ... rest of the component methods
}
```

Explanation:

1. extends React.Component**:** This line establishes that your class inherits from the base React.Component class, which provides functionalities needed for React components.
2. constructor(props)**:** The constructor is a special method that is invoked when a new instance of the component is created.
    - If your current component is called by any other component (Let's say any parent component) with some props then, the props argument holds the properties passed down from the parent component,
3. In React class-based components, the super(props) call within the constructor serves two important purposes:

1. Inheriting Properties from React.Component:

- When you extend your class component from React.Component, you inherit various methods and functionalities from the base class i.e, React.Component .
- These inherited functionalities include lifecycle methods like render, componentDidMount, etc., which are essential for building React components.
- By calling super(props), you ensure that these **inherited properties are properly initialized** for your component instance.

2. Receiving Props from Parent Components:

- When a component is used within another component (called a parent component), the parent can pass data to the child component through **props**.
- These props are essentially key-value pairs that hold the information the child component needs to function.
- By calling super(props), your component **receives and stores the props object** passed down from its parent. This allows you to access those props within your component using the this.props property.

4. this.state = ...**:** This line is where you initialize the component's internal state. The object you assign to this.state defines the initial values for your state variables.

Mount vs Render : A must read

Understanding why we can make componentDidMount async but we cannot make the call back function of useEffect as async function?

While both `componentDidMount` and `useEffect` are used to perform side effects in React components, their approach to handling asynchronous operations differs for following key reasons:

1. Return Value Expectation:

- **componentDidMount:** This method **doesn't have a return value expectation**. You can use regular synchronous or asynchronous code within it without causing errors.
- **useEffect:** This hook expects its **callback function to return either a cleanup function or nothing**. This means directly making the callback function `async` is problematic as it creates a promise-like object instead of a function or `undefined`.
- As async function will return a promise, which useEffect does not expects

Here's why making `useEffect` callback `async` is not recommended:

1. **Unexpected Behavior:** If you directly use an `async` function inside `useEffect`, it won't work as intended. The function will still be treated as a regular function, and the `await` keyword will have no effect.

2. **Missing Cleanup:** `useEffect` relies on the return value to perform cleanup tasks when the component unmounts or re-renders with changed components. An `async` function doesn't meet this expectation and can lead to memory leaks or unintended side effects.

Difference in componentDidMount and useEffect()

Here's an explanation of why `componentDidMount` and `useEffect` are not the same, focusing on the key differences and their implications:

1. Timing of Execution (Synchronous and asynchronous):

- **componentDidMount** (Class Components): Executes synchronously after the first render. This means it runs before the user sees the results of the initial render, potentially blocking the display of content until it completes.
- **useEffect** (Functional Components): Executes asynchronously after the first render **and subsequent renders**. After the initial render, React updates the DOM and then calls effects. Therefore, users see the UI change before effects run. This creates a smoother user experience.

2. State and Prop Capture:

- **componentDidMount:** Captures the state and props as they were during the first render. If the state or props subsequently change, `componentDidMount` won't run again, so it won't see those updated values. For this purpose we use componentDidUpdate() method.
- **useEffect:** By default, captures the most recent version of state and props on every render cycle. This means it always operates with the latest data available. You can

control this behavior using the dependency array in the second argument of `useEffect`.

3. Cleanup Management:

- **componentDidMount:** Doesn't provide a built-in mechanism for cleaning up side effects (e.g., subscriptions, timers, event listeners). You would need to manage this manually in `componentWillUnmount`.
- **componentWillUnmount :** componentWillUnmount will be used for cleanup.
- **useEffect:** Allows you to return a cleanup function from the callback. This function runs when the component. This ensures a cleaner and more predictable management of side effects.

We can use useEffect hook :

1. Once after initial rendering.
2. Once after initial rendering and when the state variables changes which are mentioned in dependency array.
3. We can also use useEffect hook after every re rendering.
4. We can also use it for a cleanup function.

→We can use componentDidMount for only once after the initial rendering r mounting only.

→For using after every rending we have a separate method known as componentDidUpdate.

→For cleanup purpose we have another method known as componentWillUpdate.

Whereas we can use all the above 3 methods usecase in a single useEffect hook

CleanUp function in useEffect()

```
useEffect(() => {
```

```
  // Your main effect code here


  // Return the cleanup function
  return () => {
    // Your cleanup logic here
  };
}, [/* dependency array */]);
```