Chap 08 - Let's get classy.

Date : 21/Jan/2023

CLASS BASED COMPONENTS

- It was less maintainable, have more code and a little messy.
- They are no longer used.
- Developers can do almost everything using functional components now.
- Functional component, at the eod is just normal js functions. Similary ↓
- Class based components are just normal js class.

CLASS BASED COMPONENT

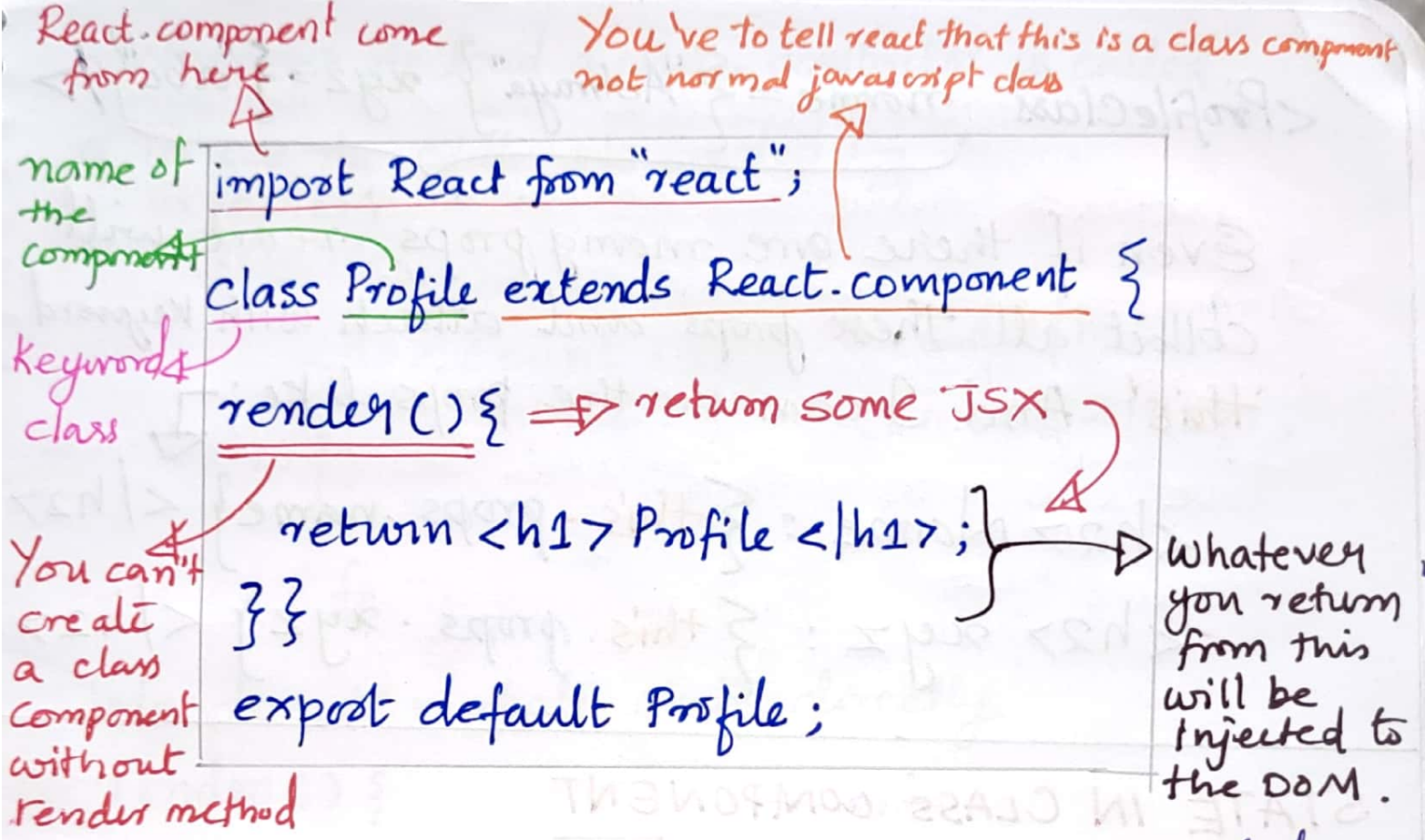
```
import React from "react";
class Profile extends React.component{
    render(){
     return <h1>Profile</h1>
}}
export default Profile;
```

ProfileClass.js

FUNCTIONAL COMPONENT

```
const Profile =()=>{
 return (
    <h1>Profile</h1
 );
}
export default Profile
```

ProfileFunc.js.



render() {} is the only mandatory method for class based components.

PROPS IN CLASS COMPONENTS

In About.js, Suppose we called class component "ProfileClass.js" as :- and if we pass props inside it :-

```
<ProfileClass name={"Ashraya"} />
```

So, in class component, we have 'this' keyword.

So, in 'ProfileClass.js' ⇩

```
<h2> Name : {this.props.name} </h2>
```

<ProfileClass name={"Ashraya"} xyz={"abc"}/>

Even if there are many props, react will collect all these props and attach with keyword 'this'. And I can use the props like :-

<h2> Name : {this.props.name} </h2>

<h2> xyz : {this.props.xyz} </h2>

STATE IN CLASS COMPONENT

```
class Profile extends React.component {
    constructor (props); {
        super (props);
        this.state = {
            count : 0,
            count2 : 0,
        };
    }
```

creates & initialize the component's state.

This is necessary because it passes the received props to the constructor of the base component class, allowing the component access it's properties.

Constructor function is called only once, when the component is first created & initialised, making it an ideal place to set the initial state of the component using this.state = {..}.

- Whenever we load a class, constructor is called.
- To use the state we created :

```
render() {
  return (
    <h2> Count : {this.state.count} </h2>
    <h2> Count2: {this.state.count2} </h2>
  ); };
```

- setCount fn

We do not mutate state directly

```
render() {
  return (
    <h2> Count : {this.state.count} </h2>
    <button
      onClick = {() => {
        this.setState ({
          Count : 1, count2 = 2,
        })
      }) > SetCount </button>
```

React Lifecycle

- first constructor is called
- Then, component is rendered

React component lifecycle refers to the series of methods that get executed at different stages of a component's existence in React application.

The lifecycle methods can be divided into 3 phases :→

1. Mounting Phase
2. Updating Phase
3. Unmounting Phase

Mounting Phase

→ These methods are called when an instance of a component is being created & inserted into the DOM :

- constructor()
- static getDerivedStateFromProps()
- render()
- componentDidMount() → This method is called immediately after the first render of of a component. It is executed only once during the lifecycle of a component.

Updating Phase :

→ These methods are called when a component is updated in response to changes in its props or state :-

- **shouldComponentUpdate ()** -

  This method is called before a component is updated. It returns a boolean value indicating whether the component should be updated or not.

- **componentWillUpdate ()** -

  This method is called just before a component is updated. It is only executed if shouldComponentUpdate() is true

- **render ()** - Used to rended the component after it has been updated.

- **componentDidUpdate ()** - This method is called immedietly after a component is updated. It is only executed if ShouldComponentUpdate() is true.

Unmounting Phase

→ The phase where component is being removed from the DOM.

- componentWillUnmount() :- This method is called just before a component is removed from the DOM.

Best Place to make API call in class Components

→ componentDidMount () { . . }

This is, because during mounting phase.

1) Constructor ( ) → is called

then, 2) render ( ) → is called.

atlast, 3) componentDidMount () → is called

Eg:-

```
class Apidatafetchexample extends React.Compment{
    state = { data: [], loading: true, error: null};
  componentDidMount () {
      this.fetchData (); }
  fetchData = () => {
      fetch ("https://api.example/data")
```

```
.then(response => response.json())
.then(data => this.setstate({data, loading:false}));
. catch(data
. then(
. catch(error => this.setState({error,
                                loading: false}));
};
```

CORE - BASIC OF CLASS COMPONENT

About.js (Parent Component)

```
class About extends Component{
 constructor(props){
   super(props);
console.log("Parent-constructor");
 }
componentDidMount(){
console.log("Parent-componentDidMount");
 }
render(){
 console.log("Parent-render");
 return(
 <Profile />
 );}}
```

Profile.js (Child Component)

```
class Profile extends Component{
 constructor(props){
   super(props);
console.log("Child-constructor");
 }
componentDidMount(){
console.log("Child-componentDidMount");
 }
render(){
 console.log("Child-render");
 return(
    <h1>Profile class</h1>
 );
}}
```

—▷ In above eg.

<About /> is the Parent component

<Profile /> is the child component of <About />

—▷ In what order the above code will execute?

① Parent - Constructor

⇩

② Parent - render (In render() of About it sees <Profile /> and it will trigged the lifecycle method of this childeren component)

⇩

③ Child - constructor

⇩

④ Child - render

⇩

⑤ Child - componentDidMount

⇩

⑥ Parent - componentDidMount.

Another Case

If <About/> component have 2 children :→
First child & Second child.

Let's see how it will be executed :-

About.js

```
class About extends Component {
    constructor (props) {
        super (props);
    console.log ("Parent-constructor");
    }
    componentDidMount () {
        console.log ("Parent- ~~render~~ componentDidMount");
    }
    render () {
        console.log ("Parent-render");
    return ( <>
    <Profile name = {"First child"} />
    <Profile name = {"Second child"} />
    </> ); }}
```

Profile class .js

```
class Profile extends React.component {
    constructor (props) {
        Super (props);
        this.state = {
            count : 0,
        };
    console.log ("Child - constructor " + this.props.name);
}
componentDidMount () {
    console.log ("Child - componentDidMount" +
                this.props.name);
}
render () {
    console.log ("Child - render" + this.props.name)
    return (
        <h1> Profile Class </h1>
    )
};
```

Order of Execution :-

1. Parent - Constructor
2. Parent - Render
3. First child - constructor
4. First child - Render
5. Second child - constructor
6. Second child - Render

COMMIT PHASE STARTS

7. First child - componentDidMount
8. Second child - componentDidMount
9. Parent - Componentdidmount

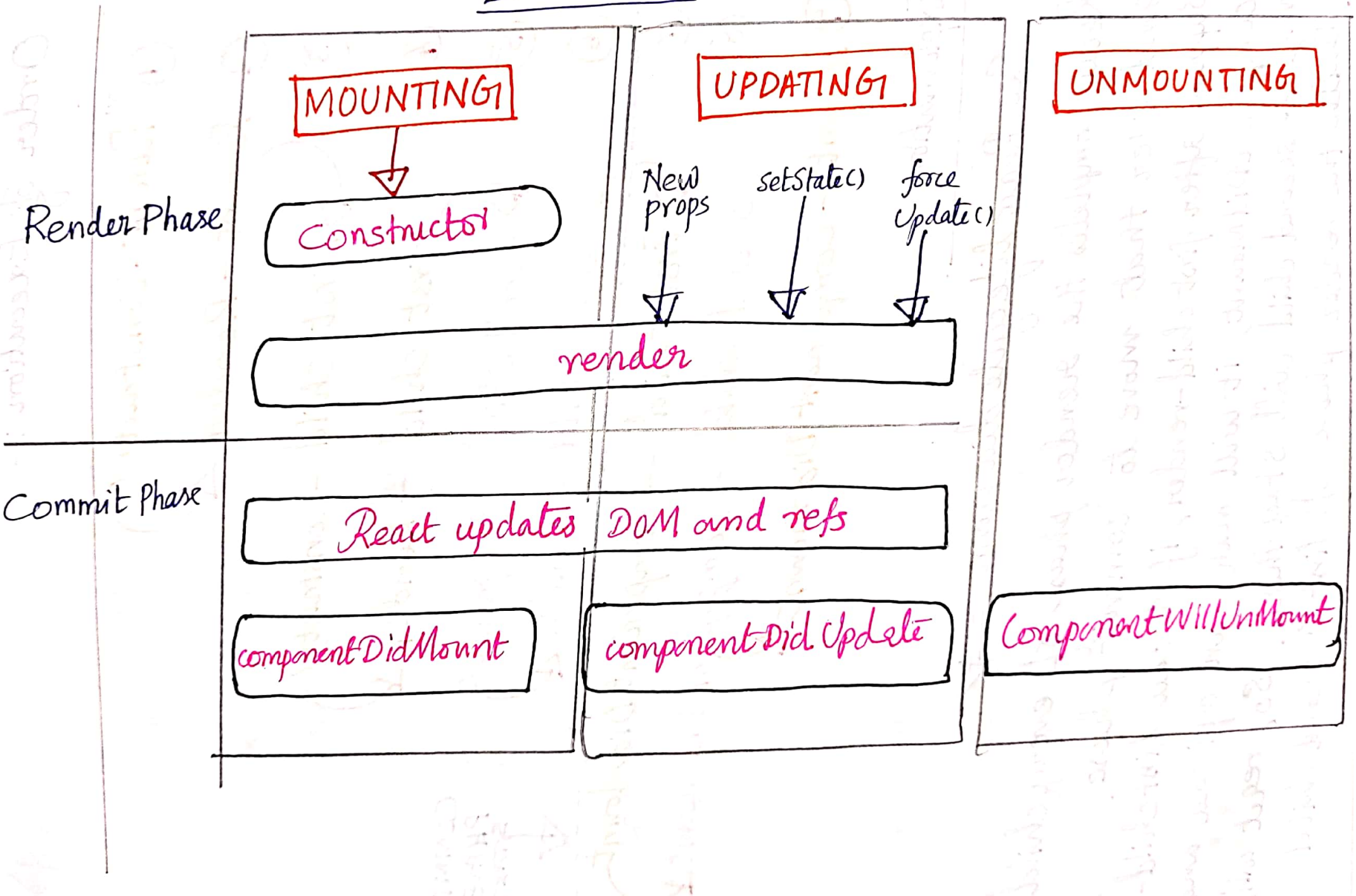
Explanation

See React lifecycle diagram ⟹

React completes the render phase of every child and after that move to commit phase.

Suppose after first-child-render, if I call firstchild-componentDidMount, it will make an api call and my ~~fir~~ second child will stay there. So, react will batch the render phase of first & second child

LIFECYCLE



React do rendering in 2 phases :-

1. Render Phase
2. Commit Phase

First of all, react finishes the RENDER PHASE.

Render Phase : → is fast

→ includes constructor and render method.

Commit Phase

→ Phase where react modifies the DOM.

→ componentDidMount is called after the initial render has finished.

→ Commit Phase is slow.

Making an API Call

→ Let's use github user api.

→ make an api call in the child component

Profileclass.js :-

* class Profile extends React.component {

```
constructor(props) {
    super(props);
    this.state = {
        userInfo : {
            name : " ",
            location : " ",
        }};
    console.log("Child - Constructor");
}

async componentDidMount() {
    const data = await fetch("https://api-github..
                              .com/users/Ashrayaa");
    const json = await data.json();
    console.log(json);
    this.setstate({
        userInfo : json,
    });
    console.log("Child - componentDidMount");
}
```

```
render() {
  console.log("Child - render");
  return (
    <h1> Name : {this.state.userInfo?.name} </h1>
    <img src={this.state.userInfo?.avatar_url} </h1>
    <h2> Location: {this.state.userInfo.location}
                                              </h2>
```

Sequence of method called in above code

I have parent 'About.js' inside one child 'ProfileClass.js'.

1. Parent - Constructor
2. Parent - Render
3. Child - constructor
4. Child - render
5. ~~API call~~
5. Parent - componentDidMount } is called before making api call.

This is because React finishes render cycle first and then it goes to commit cycle. As Child - componentDidmount, will take some time for the data to load, Parent - componentDidMount is called before. So, hence this Sequence.

① Parent - constructor

② Parent render

③ Child constructor

④ child render

⑤ DOM is updated

⑥ json is logged in console

⑦ Parent - componentDidMount

⑧ Child - componentDidMount

It is called before but is been put into the wait cycle. Because we are using async.

⑨ Child - render

* setState trigger next render. It will trigger reconciliation process. So, the child will be rendered once again when we have the data.

This re-render cycle is known as "UPDATING"

* ComponentDidMount is called after first render.
* ComponentDidUpdate is called after every next render
* Before the component is unmounted from the DOM, ComponentWillUnmount will be called.

NB: Never compare React Lifecyclemethod with Functional components
In modern react code, they removed the concepts of lifecycle method.

ComponentDidUpdate

→ is called after every subsequent render.

→ In functional component, we use dependency array in useEffect which indicates when should useEffect should be called.

Eg:-
```
useEffect(() => {
    // API call
}, [count, count2]);
```

This means that whenever the count and count2 gets updated, UseEffect gets executed.

Earlier, in class-based components, this is done like below: ↓ Which is hectic!

```
componentDidUpdate (prevProps, prevState) {
  if (
    this.state.count !== prevState.count ||
    this.state.count2 !== prevState.count2)
  {
    //API call.
    //code
  }
}
```