Let's Get Classy

Class Component

- Class component is a normal Javascript class.

1. How we can create this class based component?
    - Start with the `class` keyword then the name of the `Component` then we will `extends React.Component`.
    - Basically, this `extends React.Component` will make react know that this is a class based component.

    Example :

    ```
    class UserClassComp extends React.Component{}
    ```

    - So, This is the way for react to tell that this is a class based components.
    - Now, We will have a `render()` method inside over here. & this `render()` method will return a piece of `JSX` which will be displayed onto the UI.

    Example :

    ```
    class UserClassComp extends React.Component {

        render() {
            return (
                <div>
                    <h1>This is Class based component.</h1>
                </div>
            )
        }
    }
    ```

2. **What is difference between functional component and class based component?**
    - `Functional component` is basically a function that returns some piece of JSX.
    - `Class based component` is a class which `extends React.Component` & it has a `render` method which returns some piece of JSX.

3. **What is this `React.Component`?**
    - This `React.component` is basically a class which is given to us by React. And this `UserClassComp` component is inheriting some properties from it.
    - We will import it from *React* because it is given to us by *React*.
    - We will also export this component as we do in functional component.

Example :

```
import React from "react";

class UserClassComp extends React.Component {
render() {
    return (
    <div>
        <h1>This is Class based component.</h1>
    </div>
    );
}
}

export default UserClassComp;
```

> We can also write only `Component` instead of `React.Component` because we can destructure it.

Example :

```
import {Component} from "react"

class About extends Component {

}
```

4. **How Class Based Components will receive a `props`?**

    - This class will have a `constructor()` and now, this constructor will receive this props.
    - We will also use `super()` inside the `constructor()`.
    - If we do not write `super()` then it will throw an `error.`

Example :

```
import React from "react";

class UserClassComp extends React.Component {
constructor(props) {
    super(props);

    console.log(props.name);
}

render() {
    return (
    <div>
        <table className="table table-bordered table-hover">
        <thead className="table-primary">
```

```
            <tr>
            <th>Name</th>
            <th>Location</th>
            <th>Contact</th>
            </tr>
        </thead>
        <tbody>
            <tr>
            <td>Aditya Ranjan</td>
            <td>Remote</td>
            <td>91xxxxxxxx</td>
            </tr>
        </tbody>
        </table>
    </div>
    );
}
}

export default UserClassComp;
```

5. **How to access `props/state` in class based component?**
    - We use always `this` keyword in class component to access either `state or props`

    Example :

```
<tr>
    <td>{this.props.name}</td>
    <td>Remote</td>
    <td>91xxxxxxxx</td>
</tr>
```

Constructor Method in React Component:

- **Purpose**: The constructor is used to initialize the state and bind methods to the component instance.
- **Syntax**:

```
constructor(props) {
    super(props);
    // Initialization code here
}
```

- **Key Points**:
    1. `constructor(props)`: Declares the constructor method with `props` as the parameter.

2. `super(props)`: Calls the constructor of the parent class (React.Component) and passes `props` to it. This ensures that `this.props` is accessible in the constructor.

- **Notes**:
  - In JavaScript, when you define a class that extends another class (inherits from a parent class), we often use the super() method with props as an argument in the constructor of the child class. This is commonly seen in React when you create class-based components.
  - The `constructor` is optional in components if you are not initializing state or binding methods. In modern React, you can directly initialize state as a class property outside the constructor.
  - Ensure to call `super(props)` as the first statement in the constructor to properly initialize the component.

or

For more information "CLICK HERE":

> Important : Whenever a new class component is created, this constructor is called and props are extracted here, So now we can use this props anywhere in the code.

State in Class Component

- State is nothing but an object that is privately maintained inside a class component. State consists of any data your application needs to know about, that can change over time.
- state can be changed within the component.
- The state property must be set to a JavaScript object.

Example :

```
this.state = { }
```

- Constructor is a place where we initialize the state.
- state is a reserved keyword in class component.

Example :

```
constructor(props) {
    super(props);

    this.state = {
        count : 0
    }
  }
```

Accessing the state :

```
<h3>Count : {this.state.count}</h3>
```

Notes :

- this state is basically a big object which will contain all these state variables
- suppose If I have to create a `count2` then we will create inside this state only rather than creating another state.

Example : **Correct state creation**

```
constructor(props) {
    super(props);

    this.state = {
        count : 0,
        count2 : 1
    }
  }
```

Example : **Wrong state creation**

```
constructor(props) {
    super(props);

    this.state = {
        count : 0
    }
    this.state = {
        count2 : 0
    }
  }
```

Update the state :

- setState method updates the state value within the component.
- Setstate method has two parameters: 1. Object 2. Callback function

> **Notes :**

- Never directly update the state value. Otherwise React never re-renders the value. Also throws an error.
- whenever you need to execute some code after the state has been changed,
  - *do not place that code right after the setState method,*
  - *instead place that code within the `callback function` that is passed as a `second argument` to the setState method*

Example :

```
// correct way:

increase(){
    this.setState({
        counter : this.state.counter + 1
    }, ()=>console.log("count -", this.state.counter ))
}
---------------------

// Wrong way :

increase(){
    this.setState({
        counter : this.state.counter + 1
    })
    console.log(this.state.counter)
}
```

- Whenever you have to update the state, based on the previous state value, make sure to pass in a callBack function as an argument instead of the regular object. The function has access to the previous state which can be used to calculate the new state and as it turns out the second parameter to this function is the props object so if your new state is dependent on props as well, `“props.addValue”` may be you can go with function parameter approach and make use of props

Example :

```
increase(){
    this.setState((prevState) => ({
        counter : prevState.counter + 1
    })
    )
}


// Second parameter :

increase(){
    this.setState((prevState, props) => ({
        counter : prevState.counter + props.addValue
    })
    )
}
```

- why we need prevState?
    - react may group multiple setState calls into a single update for better performance.

Binding Event Handler

- “This” keyword within our event handler is Undefined. And that is the reason , event binding is necessary in react class component.
- There is a several way to bind the events : 3rd & 4th approach is best approach

1. Binding in render :

```
onClick={this.clickhandler.bind(this)}
```

   - every update to the state will cause the component to re-render. It will generate a brand new event handler on every render although it will troublesome on large applications

2. Binding with arrow function in render :

```
onClick={()=>this.clickHandler()}
```

   - we are calling the event handler and return that value that is why parentheses are required. It also has some performance implications in some scenarios.

3. Binding in the class constructor :- Deals with binding the event handler in the constructor as opposed to binding in the render method

   - Under the constructor :

```
this.clickHandler = this.clickHandler.bind(this)
```

   - Under the event :

```
onClick = {this.clickHandler}
```

4. Use an arrow function as a class property

   - Create a click function using an arrow function

```
clickHandler = () => {
this.setState({
message : “GoodBye”
})
}
```

```
onClick = {this.clickHandler}
```

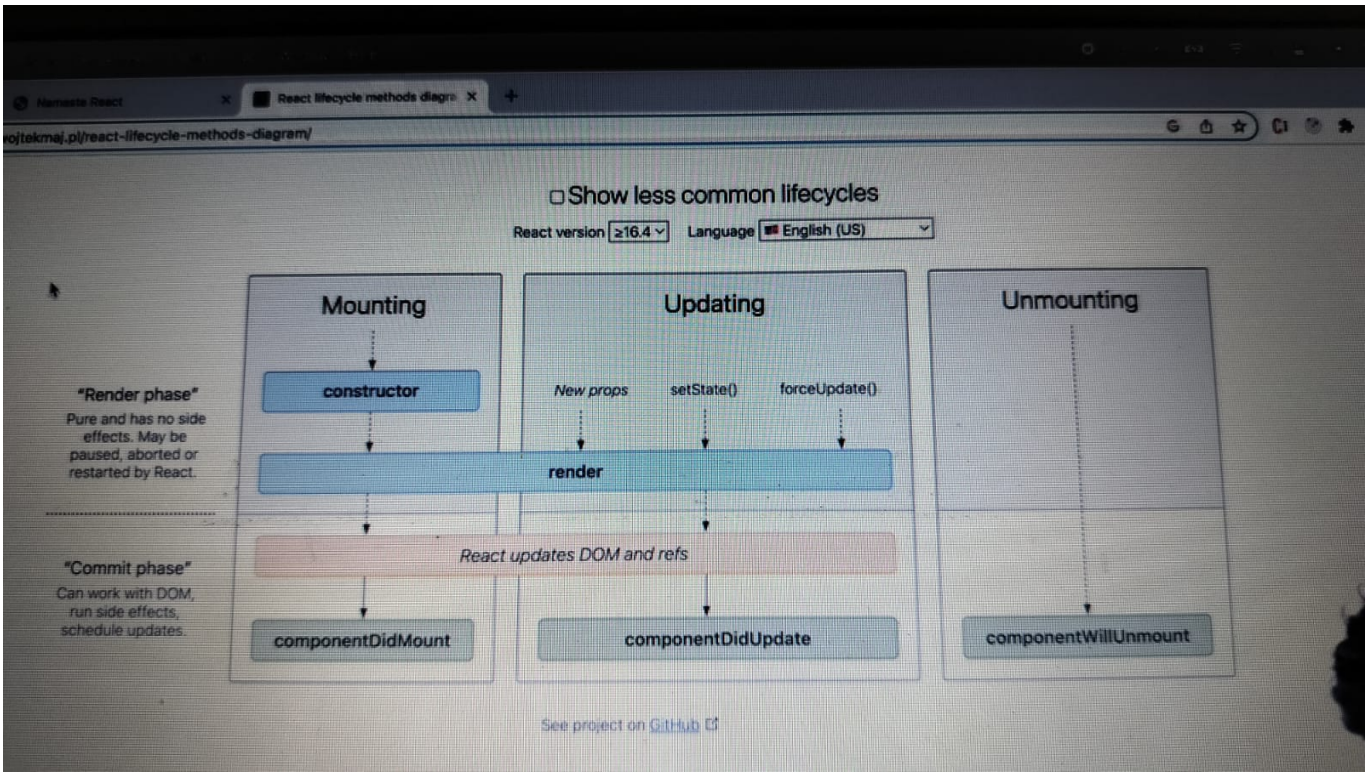
Life Cycle methods of Class Based Component

> `Loading and Mounting`, both word are same thing.

In React, Components go through a series of lifecycle methods from initialization to destruction, categorized into three main phases: `Mounting`, `Updating`, and `Unmounting`. Each phase offers specific lifecycle methods that allow developers to execute code at different points in the component's lifecycle.

Those methods are *Constructor, render, componentDidMount, ComponentDidUpdate, ComponentWillUnmount:*

Let's understand this diagram & explore phases :



- ***Mounting Phase:***

  Constructor(): The constructor method is the first to be called when a component is created. It's where we typically initialize the component's state and bind event handlers.

  Render(): The render method is responsible for rendering the component's UI. It must return a React element (typically JSX) representing the component's structure.

  ComponentDidMount(): This method is called immediately after the component has been rendered to the DOM. It is often used to perform tasks like fetching data from an API & DOM Manipulation.

  - *DOM Manipulation*: When we need to interact with the DOM directly, such as selecting elements, setting attributes, or applying third-party libraries that require DOM elements to be present, we can safely do so in componentDidMount. This is because the component is guaranteed to be in the DOM at this point.

Example :

```
class MyComponent extends React.Component {
componentDidMount() {
    // Fetch data from an API
    fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => {
        // Update the component's state with the fetched data
        this.setState({ data });
    })
    .catch(error => {
        // Handle any errors
        console.error(error);
    });
    }
render() {
    // Render component based on state
    return (
    <div>{/* Display data from this.state.data */}</div>
    );
}
}
```

- ***Updating Phase:***

  render(): Re-renders the component with updated state or props.

  componentDidUpdate(prevProps, prevState, snapshot): This method is called after the component has been updated (re-rendered) due to changes in state or props. It's often used for side effects, like updating the DOM in response to state or prop changes.

- ***Unmounting Phase:***

  componentWillUnmount(): This method is called just before the component is removed from the DOM. It's used to clean up resources or perform any necessary cleanup.

  - > *Mounting* means showing onto the UI.
  - > *Unmounting* means disabling like removing from the UI.

    Here's why and when we should use componentWillUnmount:

    Cleanup Resources: If your component has allocated any resources, such as event listeners, subscriptions, timers, or manual DOM manipulations, it's essential to release these resources to prevent memory leaks. componentWillUnmount is the appropriate place to do this.

    Cancel Pending Requests: If your component has initiated any asynchronous requests, such as AJAX calls or timers, you should cancel or clean them up to avoid unexpected behavior after the component is unmounted.

Here's an example of using componentWillUnmount to remove an event listener when a component is unmounted:

```
class MyComponent extends React.Component {
constructor() {
    super();
    this.handleResize = this.handleResize.bind(this);
}
componentDidMount() {
    // Add a window resize event listener when the component
is mounted
    window.addEventListener('resize', this.handleResize);
}
componentWillUnmount() {
    // Remove the window resize event listener when the compo
nent is unmounted
    window.removeEventListener('resize', this.handleResize);
}
handleResize(event) {
    // Handle the resize event
    console.log('Window resized:', event);
}
render() {
    return <div>My Component</div>;
}
}
```

In this example, the component adds a resize event listener to the window when it's mounted, and it removes that listener in the componentWillUnmount method.

This ensures that the event listener is properly cleaned up when the component is unmounted, preventing memory leaks or unexpected behavior.

By using componentWillUnmount, we can ensure that any cleanup tasks are executed reliably when the component is no longer needed, helping to maintain the integrity of our application and avoiding potential issues.

> **Notes :**

- First of all the Mounting Cycle happens then the update cycle happens

```
-----Mounting -----------
constructor with dummy data
Render with dummy data
    <HTML Dummy>
componentDidMount
    <API Call>
    <this.setState> --> state variable is updated.

--------Updating--------
```

```
render(API Data)
    <HTML (new API Data)>
componentDidUpdate
```

For more information "CLICK HERE"

1. **How lifecycle methods are called when there are multiple children?**
    - Let's understand below case :
        - what happens is,
            - parent constructor is called, parent render is called then the first child constructor is called, then the first child render is called. So the render phase is completed.
        - It will batch the render phase,
            - So now, the second child constructor is called, then second child render is called
        - Once the render phase is batched and completed.
        - then the commit phase will happen for
            - first child componentDidMount and then second child componentDidMount and finally parent componentDidMount.

    > Render phase first batched up for all childrens then starts the commit phase.

    Correct way

    ```
    Parent constructor
    Parent render

        first child constructor
        first child render

        second child constructor
        second child render

        first child componentDidMount
        second child componentDidMount

    parent componentDidMount
    ```

2. **Why react is doing that like this. why first batched all renders and then started the commit phase?**
    - This is called React's optimization strategies.
    - React is batching up the render phase for multiple children's, because once the commit phase starts, react tries to update the DOM and DOM manipulation is the most expensive thing, when we are updating a component because it takes a lot of time.
    - So, react want to batch up the render first for all these childrens, then finally starts commit phase.

3. **How to make an API Call in class based component?**

- We make a `componentDidMount()` as a `async` function and `await` for `fetchData` function.

Example :

```
async componentDidMount() {
    const data = await fetch("https://api.github.com/users/adityaranjan");

    const json = await data.json();

    this.setState({
        userInfo: json,
    });

    // console.log(json);
    // console.log("Child component did mount");
}
```

4. **What will happen if I write `count` as a dependency array inside the useEffect?**

   - Every time my count changes, then useEffect will be called.

   Now, how can we achieve something like `count` inside my class based component?

   - We can achieve that functionality by using `componentDidUpdate()`.
   - We know that componentDidUpdate() is called after every update.

   Example :

```
  componentDidUpdate(prevProps, prevState) {
        if(this.state.count !== prevState.count){

        }
    }
```

5. **Why `dependency array` is an `array`?**

   - Because there can be multiple state variables where might, if/else want to do.
   - So, now we can write multiple state variables like

   Example :

```
useEffect(()=>{

},[count, count2, count3])
```

6. **Suppose If I want to do something, when my `count` changes and also something else, when my `count2` changes, how will I write that in functional component?**

- There is a 2 separate conditions for `count` and `count2`. So we will write `two useEffect` separately like below:

Example :

```
// for Count
-------------
useEffect(()=>{

},[count])
```

```
// for Count2
-------------
useEffect(()=>{

},[count2])
```

How to write same thing in a class based component?

Example :

```
  componentDidUpdate(prevProps, prevState) {
        if(this.state.count !== prevState.count){

        }

        if(this.state.count2 !== prevState.count2){

        }
    }
```

7. **What will happen if we create a `setInterval()` in class based component and do not clean up?**

   - It will run all the time, even if we moved from one component to another component. It will not stop.

Example :

```
componentDidMount() {
this.timer = setInterval(() => {
    console.log("Namaste React");
}, 1000);
}
```

- So, We need to clean up this using `componentWillUnmount()`. This way we can stop it.

```
componentWillUnmount() {
    clearInterval(this.timer);
}
```

8. **How to clean up things in `useEffect()`?**

   - There is something known as `return`.
   - We can `return a function` from `useEffect()`
   - This function is basically called when we are unmounting the component.

   Example : this `return a function` wrote inside the callback function of `useEffect()`

```
return ()=>{

}
```

   Like this below one:

```
useEffect(()=>{
    const timer = setInterval(() => {
        console.log("Namaste React");
    }, 1000);

    return ()=>{
        clearInterval(timer);
    }
},[])
```

9. **Why can't we have the callback function of useEffect async ?**

   - In React, the `useEffect` hook is designed to handle side effects in functional components. It's a powerful and flexible tool for managing asynchronous operations, such as data fetching, API calls, and more. However, useEffect itself cannot directly accept an async callback function. This is because useEffect expects its callback function to return either nothing (i.e., undefined) or a cleanup function, and it doesn't work well with Promises returned from async functions.

     There are a few reasons for this:

     Return Value Expectation: The primary purpose of the useEffect callback function is to handle side effects and perform cleanup. React expects us to either return nothing (i.e., undefined) from the callback or return a cleanup function. An async function returns a Promise, and it doesn't fit well with this expected behavior.

Execution Order and Timing: With async functions, we might not have fine-grained control over the execution order of the asynchronous code and the cleanup code. React relies on the returned cleanup function to handle cleanup when the component is unmounted or when the dependencies specified in the useEffect dependency array change. If you return a Promise, React doesn't know when or how to handle cleanup.

To work with async operations within a useEffect, we can use the following pattern:

```
useEffect(() => {
const fetchData = async () => {
    try {
    // Perform asynchronous operations
    const result = await someAsyncOperation();
    // Update the state with the result
    setState(result);
    } catch (error) {
    // Handle errors
    console.error(error);
    }
};
fetchData(); // Call the async function
return () => {
    // Cleanup code, if necessary
    // This function will be called when the component unmoun
ts or when dependencies change
};
}, [/* dependency array */]);
```

In this pattern, we define an async function within the useEffect callback, perform our asynchronous operations, and then call that function. Additionally, we return a cleanup function from the useEffect to handle any necessary cleanup tasks when the component unmounts or when specified dependencies change.

By using this approach, we can effectively manage asynchronous operations with useEffect while adhering to React's expectations for the callback function's return value.

10.