Agenda:

- Class components in React
- State in React
- Component Lifecycle in React
- Mounting Lifecycle methods
- Unmounting Lifecycle methods
- Updating Lifecycle methods

In the previous session on React, we have seen that React lets us define components as classes or functions. We have discussed about Functic Components in our last session and In this session, we will be starting with understanding Class Components deeply:

Class Components in React :

To define a React component class, We have to create a class and extend React.Component class. Lets say we want to create an Employee Compone and this Component should return the Element which displays Employee Details.

Open Index.js file in your React project, Create Employee class and extend it from React.Component. Output of any Class Component we create dependent on the return value of a Method Called render(). The render() method is the only required method needs to be implemented in a clas component.

lets create a div element which displays employee details like ID, Name , Location, Salary and return the div from this Method. To access the attribute that will be passed to this Component Class, in React we use this.props. Attribute Name. this.props contains the props that were defined by the caller this component.

```
class Employee extends React.Component {
  render(){
    return <div>
      <h2>Employee Details...</h2>
      <p>
        <label>Name : <b>{this.props.Name}</b></label>
      </p>
      </div>;
  }
}
```

Calling the class component and rendering remains as same as the function component, which is shown below in the code:

```
const element=<Employee Name="Anuj Yadav"/>

ReactDOM.render(element,document.getElementById("root"));
```

Now lets create Department Component also as Class Component. So we create a Class, Name it as Department and extend React.Component. We w return an Element which displays Department Information like Department Name, Head of the Department Name and Use this Component in Employe Component.

So, as we are using this new Department Component in our employee component so, we need to change our employee component, in order to rend this component, which is shown below:

```
class Employee extends React.Component {

  render(){
    return <div>
      <h2>Employee Details...</h2>
      <p>
        <label>Name : <b>{this.props.Name}</b></label>
      </p>
      <Department Name={this.props.DeptName}/>
      </div>;
  }
}
```

And the Department component is created as shown below:

```
class Department extends React.Component {

  render(){
    return <div>
      <h2>Department Details...</h2>
      <p>
        <label>Name : <b>{this.props.Name}</b></label>
      </p>
      </div>;
  }
}
```

And now lets render our employee component as shown below:

```
const element=<Employee Name="Anuj Yadav" DeptName="FullStack Developer"/>

ReactDOM.render(element,document.getElementById("root"));
```

So, in this way we can create a class component and can also render the nested components.

As we have seen in the last session, that Props are read only.

Whether we declare a component as a function or a class, it must never modify its own props.

To understand this with an Example, lets go and add a Constructor to the Employee Component Class and inside the Constructor, lets try to log th Property Object.

```
constructor(){
   console.log( this.props);
  }
```

This Code will throw the Error and It is expecting us to call the Base Class Constructor and we do that by using Super();

After adding the Base Class Constructor call, if we look at the Console in the browser, we get props value as undefined. To make sure that props obje can be accessed in constructor, we have to add this parameter to the constructor and pass that to the base class constructor as well. Now if we sav this, we can see that object data in the Console log.

Now if we try to Change the Salary using props object, we can see an error and error says.

React is pretty flexible but it has a single strict rule. Props are read-only. Of course, application UIs are dynamic and change over time. a new conce named “state” allows React components to change their output over time in response to user actions without violating this rule. By now we are clear c how to create function component and A Class Component. Then the obvious question is when to use which one?

If we are expecting features like

1. Managing State of the Components
2. Adding Life Cycle Methods to Components
3. Need to Write Logic for Event Handlers

Then we will go for Class Component and in rest of the cases we can go for Function Component.

So, as just we have seen that we need to manage states in react, so let’s start learning about the State in React :

State in React :

React components will often need *dynamic information* in order to render. For example, imagine a component that displays the score of a basketb game. The score of the game might change over time, meaning that the score is *dynamic*. Our component will have to know the score, a piece dynamic information, in order to render in a useful way.

There are two ways for a component to get dynamic information: `props` and `state`. Besides `props` and `state`, every value used in component should always stay exactly the same.

You spend the last lesson learning about `props`. Now it’s time to learn about `state`. `props` and `state` are all that you need to set up a ecosystem of interacting React components.

Setting Initial State :

A React component can access dynamic information in two ways: `props` and `state`.

Unlike `props`, a component’s `state` is *not*passed in from the outside. A component decides its own `state`.

To make a component have `state`, give the component a `state` property. This property should be declared inside of a constructor method, a shown below :

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'decent' };
  }

  render() {
    return <div></div>;
  }
}

<Example />
```

Let's take a closer look at constructor code :

```
constructor(props) {
  super(props);
  this.state = { mood: 'decent' };
}
```

`this.state` should be equal to an object, like in the example above. This object represents the initial "state" of any component instance.

How about the rest of the code? `constructor` and `super` are both features of ES6, not unique to React. It is important to note that Rea components *always* have to call `super` in their constructors to be set up properly.

Look at the bottom of the highest code example in this column. `<Example />` has a `state`, and its `state` is equal to `{ mood: 'decent' }`.

Accessing a component's state :

To *read* a component's `state`, use the expression `this.state.name-of-property` as shown below :

```
class TodayImFeeling extends React.Component {
  constructor(props) {
    super(props);
    this.state = { mood: 'decent' };
  }

  render() {
    return (
      <h1>
        I'm feeling {this.state.mood}!
      </h1>
    );
  }
}
```

The above component class reads a property in its `state` from inside of its `render` function.

Just like `this.props`, you can use `this.state` from any property defined inside of a component class's body.

Updating state with this.setState :

A component can do more than just read its own state. A component can also *change* its own state.

A component changes its state by calling the function `this.setState()`.

`this.setState()` takes two arguments: an *object*that will update the component's state, and a callback. You basically never need the callback.

Consider the code shown below :

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      mood:   'great',
```

```
      hungry: false
    };
  }
```

Notice that `<Example />` has a state of :

```
{
  mood:   'great',
  hungry: false
}
```

Now, let's say that `<Example />` were to call:

```
this.setState({ hungry: true });
```

After that call, here is what `<Example />` 's state would be:

```
{
  mood:   'great',
  hungry: true
}
```

The `mood` part of the state remains unaffected!

`this.setState()` takes an object, and *merges*that object with the component's current state. If there are properties in the current state that aren't pa of that object, then those properties remain how they were.

Here, we need to understand that we can call `this.setState` from an another function, let's see how we can do that :

The most common way to call `this.setState()` is to call a custom function that *wraps* a `this.setState()` call.

Consider the .makeSomeFog() function shown below :

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { weather: 'sunny' };
    this.makeSomeFog = this.makeSomeFog.bind(this);
  }

  makeSomeFog() {
    this.setState({
      weather: 'foggy'
    });
  }
}
```

Notice how the method `makeSomeFog()` contains a call to `this.setState()` .

You may have noticed a weird line in there:

```
this.makeSomeFog = this.makeSomeFog.bind(this);
```

This line is necessary because `makeSomeFog()` 's body contains the word `this` .

Due to the way that event handlers are bound in JavaScript, as we have learned in Javascript , we just need to know that in React, whenever you defir an event handler that uses `this` , you need to add `this.methodName = this.methodName.bind(this)` to your constructor function.

Here, we also need to understand that this.setState Automatically calls render :

For this to understand, consider the code shown below :

```
class Toggle extends React.Component {

const green = '#39D1B4';
const yellow = '#FFD712';
```

```
constructor(props) {
    super(props);
    this.state = {
      color: green
    };
    this.changeColor = this.changeColor.bind(this);
}

changeColor() {
    this.setState({
      color: yellow
    });
  }

  render() {
    return (
      <div>
      <div  style={{background:this.state.color}}>
          Change my color
      </div>
      <button onClick={this.changeColor}>Change Color</button>
      </div>
    );
  }
}
```

In the above example when a user clicks on the , the `.changeColor()` method is called. Take a look at `.changeColor()` 's definition.

`.changeColor()` calls `this.setState()` , which updates `this.state.color` . However, even if `this.state.color` is update from `green` to `yellow` , that alone shouldn't be enough to make the screen's color change!

The screen's color doesn't change until `Toggle` *renders.*

Inside of the render function, it's this line:

```
<div style={{background:this.state.color}}>
```

that changes a virtual DOM object's color to the new `this.state.color` , eventually causing a change in the screen.

If you call `.changeColor()` , shouldn't you then *also* have to call `.render()` again? `.changeColor()` only makes it so that, the next time th you render, the color will be different. Why can you see the new background right away, if you haven't re-rendered the component?

Here's why: *Any time that you call this.setState(), this.setState() AUTOMATICALLY calls .render() as soon as the state has changed.*

Think of `this.setState()` as actually being two things: `this.setState()` , immediately followed by `.render()` .

That is why you can't call `this.setState()` from inside of the `.render()` method! `this.setState()` *automatically* calls `.render()` If `.render()` calls `this.setState()` , then an infinite loop is created.

So, this is all we need to learn about state in react now, let's start with lifecycle of a component in react :

The Component's Lifecycle in React :

We've seen that React components can be highly dynamic. They get created, rendered, added to the DOM, updated, and removed. All of these steps a part of a component's *lifecycle*.

The component lifecycle has three high-level parts:

1. *Mounting*, when the component is being initialized and put into the DOM for the first time
2. *Updating*, when the component updates as a result of changed state or changed props
3. *Unmounting*, when the component is being removed from the DOM

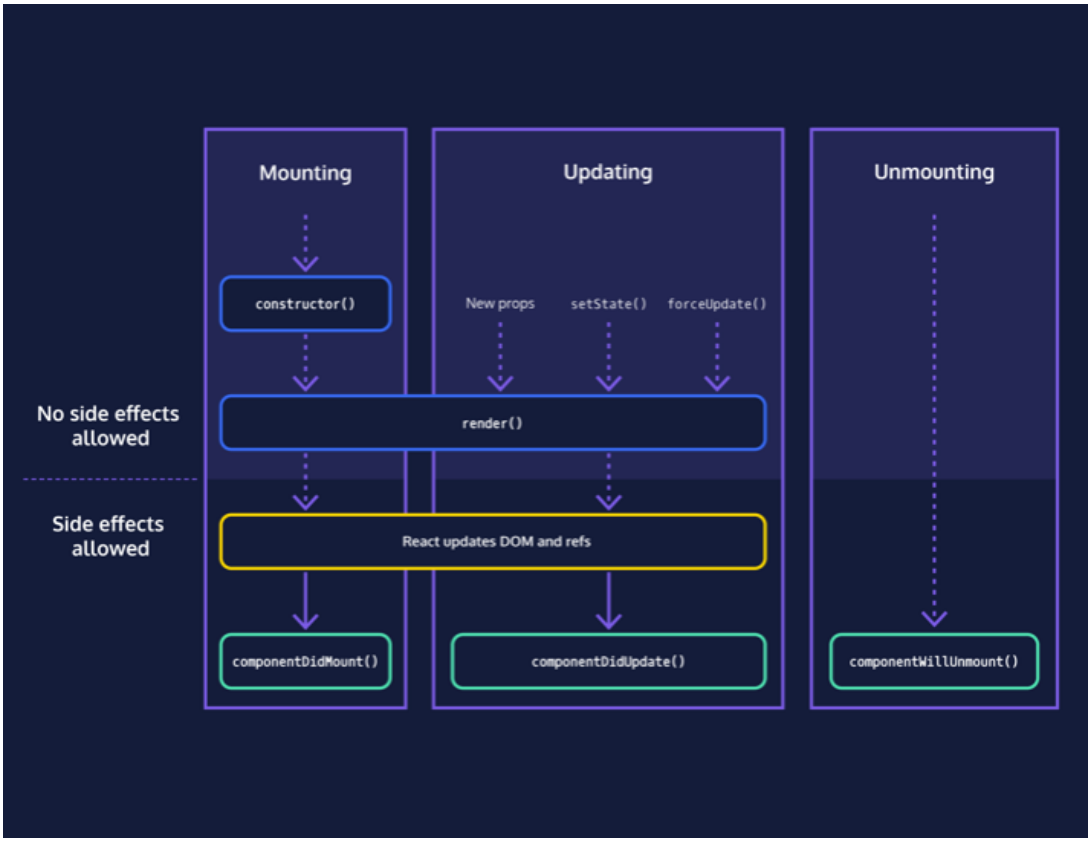
Every React component you've ever interacted with does the first step at a minimum. If a component never mounted, you'd never see it!

Most interesting components are *updated* at some point. A purely static component—like, for example, a logo—might not ever update. But if component's state changes, it updates. Or if different props are passed to a component, it updates.

Finally, a component is *unmounted* when it's removed from the DOM. For example, if you have a button that hides a component, chances are th component will be unmounted. If your app has multiple screens, it's likely that each screen (and all of its child components) will be unmounted. If component is "alive" for the entire lifetime of your app (say, a top-level `<App />` component or a persistent navigation bar), it won't be unmounted. B most components can get unmounted one way or another!

It's worth noting that each component instance has its own lifecycle. For example, if you have 3 buttons on a page, then there are 3 compone instances, each with its own lifecycle. However, once a component instance is unmounted, that's it—it will never be re-mounted, or updated again, unmounted.

Consider the image shown below :



Introduction to Lifecycle Methods :

React components have several methods, called *lifecycle methods*, that are called at different parts of a component's lifecycle. This is how you, th programmer, deal with the lifecycle of a component.

You may not have known it, but you've already used two of the most common lifecyc methods: `constructor()` and `render()` ! `constructor()` is the first method called during the mounting phase. `render()` is called later durir the mounting phase, to render the component for the first time, and during the updating phase, to re-render the component.

Notice that lifecycle methods don't necessarily correspond one-to-one with part of the lifecycle. `constructor()` only executes during the mountir phase, but `render()` executes during both the mounting and updating phase.

With this new understanding, let's build a simple clock component as shown below :

```
mport React from 'react';
import ReactDOM from 'react-dom';

class Clock extends React.Component {
constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return <div>{this.state.date.toLocaleTimeString()}</div>;
  }

  // Add your methods in here.
}

ReactDOM.render(<Clock />, document.getElementById('app'));
```

Mounting Lifecycle Methods :

React supports three mounting lifecycle methods for component classes: `componentWillMount()`, `render()` and `componentDidMount()`. `componentWillMount()` will be called first followed by the `render()` method and final the `componentDidMount()` method.

componentDidMount :

We've made a clock component, but it's static. Wouldn't it be nice if it updated?

At a high level, we'd like to update `this.state.date` with a new date once per second.

JavaScript has a helpful function, `setInterval()`, that will help us do just this. It lets us run a function on a set interval. In our case, we'll make function that updates `this.state.date`, and call it every second.

We'll want to run some code that looks like this:

```
const oneSecond = 1000;
setInterval(() => {
  this.setState({ date: new Date() });
}, oneSecond);
```

We have the code we want to run—that's great. But where should we put this code? In other words, where in the component's lifecycle should it go?

Remember, the component lifecycle has three high-level parts:

1. *Mounting*, when the component is being initialized and put into the DOM for the first time
2. *Updating*, when the component updates as a result of changed state or changed props
3. *Unmounting*, when the component is being removed from the DOM

It's certainly not in the unmounting phase—we don't want to start our interval when the clock disappears from the screen! It's also probably not usef during the updating phase—we want the interval to start as soon as the clock appears, and we don't want to wait for an update. It probably makes sens to stick this code somewhere in the mounting phase.

We've seen two functions: the `render()` and the constructor. Can we put this code in either of those places?

- `render()` isn't a good candidate. For one, it executes during the mounting phase and the updating phase—too often for us. It's also generally a bad idea to set up any kind of side-effect like this in `render()`, as it can create subtle bugs in the future.
- `constructor()` is also not great. It does only execute during the mounting phase, so that's good, but you should generally avoid side-effects like this in constructors because it violates something called the Single Responsibility Principle. In short, it's not a constructor's responsibility to start side-effects.

If it's not `render()` or the constructor, then where? Enter a new lifecycle method, `componentDidMount()`.

`componentDidMount()` is the final method called during the mounting phase. The order is:

1. The constructor
2. `render()`
3. `componentDidMount()`

In other words, it's called after the component is rendered. This is where we'll want to start our timer.

Consider the code shown below :

```
export class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return <div>{this.state.date.toLocaleTimeString()}</div>;
  }
  componentDidMount() {
    const oneSecond = 1000;
    setInterval(() => {
      this.setState({ date: new Date() });
    }, oneSecond);
```

```
  }
}
```

Unmounting Lifecycle Method :

React supports one unmounting lifecycle method, `componentWillUnmount` , which will be called right before a component is removed from the DOM.

`componentWillUnmount()` is used to do any necessary cleanup (canceling any timers or intervals, for example) before the component disappears.

Note that the `this.setState()` method should not be called inside `componentWillUnmount()` because the component will not be re-rendered.

ComponentWillUnmount :

Our clock is working, but it has an important problem. We never told the interval to stop, so it'll keep running that function forever (or at least, until th user leaves/refreshes the page).

When the component is unmounted (in other words, removed from the page), that timer will keep on ticking, trying to update the state of a compone that's effectively gone. This means your users will have some JavaScript code running unnecessarily, which will hurt the performance of your app.

React will log a warning that looks something like this:

```
Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it indicates a memory leak i
```

Imagine if the clock gets mounted and unmounted hundreds of times—eventually, this will cause your page to become sluggish because of all of th unnecessary work. You'll also see warnings in your browser console. Even worse, this can lead to subtle, annoying bugs.

All this bad stuff can happen if we fail to clean up a side-effect of a component. In our case this is a call to `setInterval()` , but components can hav lots of other side-effects: loading external data with AJAX, doing manual tweaking of the DOM, setting a global value, and more. We try to limit our sid effects, but it's difficult to build an interesting app with truly zero side-effects.

In general, when a component produces a side-effect, you should remember to clean it up.

JavaScript gives us the `clearInterval()` function. `setInterval()` can return an ID, which you can then pass into `clearInterval()` to clear Here's the code we'll want to use:

```
const oneSecond = 1000;
this.intervalID = setInterval(() => {
  this.setState({ date: new Date() });
}, oneSecond);

// Some time later...
clearInterval(this.intervalID);
```

At a high level, we want to continue to set up our `setInterval()` in `componentDidMount()` , but then we want to clear that interval when the clo is unmounted.

Let's introduce a new lifecycle method: `componentWillUnmount()` . `componentWillUnmount()` is called in the unmounting phase, right before th component is completely destroyed. It's a useful time to clean up any of your component's mess.

Consider the code shown below :

```
export class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }
  render() {
    return (
      <div>
        {this.props.isPrecise
          ? this.state.date.toISOString()
          : this.state.date.toLocaleTimeString()}
      </div>
    );
  }
  componentDidMount() {
    const oneSecond = 1000;
    this.intervalID = setInterval(() => {
      this.setState({ date: new Date() });
```

```
    }, oneSecond);
  }
  componentWillUnmount() {
    clearInterval(this.intervalID);
  }
}
```

Updating Lifecycle Method :

When a component updates, `shouldComponentUpdate()` gets called after `componentWillReceiveProps()` , but still before the rendering begins. automatically receives two arguments: `nextProps` and `nextState` .

`shouldComponentUpdate()` should return either true or false. The best way to use this method is to have it return false *only under certain condition* If those conditions are met, then your component will not update.

componentDidUpdate :

Remember the three parts of a component's lifecycle:

1. *Mounting*, when the component is being initialized and put into the DOM for the first time
2. *Updating*, when the component updates as a result of changed state or changed props
3. *Unmounting*, when the component is being removed from the DOM

We've looked at mounting (`constructor()` , `render()` , and `componentDidMount()`). We've looked at unmountir (`componentWillUnmount()`). Let's finish by looking at the updating phase.

An update is caused by changes to props or state. You've already seen this happen a bunch of times. Every time you've called `setState()` with ne data, you've triggered an update. Every time you change the props passed to a component, you've caused it to update.

When a component updates, it calls several methods, but only two are commonly used.

The first is `render()` , which we've seen in every React component. When a component's props or state changes, `render()` is called.

The second, which we haven't seen yet, is `componentDidUpdate()` . Just like `componentDidMount()` is a good place for mount-phas setup, `componentDidUpdate()` is a good place for update-phase work.

So, we have learnt about lifecycle in react and its some main methods.

Conclusion:

We have learnt about

- class component in react
- state in react
- lifecycle of components in react and about some main lifecycle methods in react.

In next session, we will build a project based upon these class components, states and lifecycle methods in React.

Thank You !