Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/React: Lifecycle Methods/6045ae5b70ff4c89cfddbf44

TUTORIAL

React: Lifecycle Methods

React components are highly dynamic. They get created, rendered, added to the DOM, updated, and removed. All of these steps are part of a component's lifecycle.

The component lifecycle has three high-level parts:

- **Mounting:** when the component is being initialized and added to the DOM for the first time
- **Updating:** when the component updates as a result of changed state or changed props
- **Unmounting:** when the component is being removed from the DOM.

Every React component that we have ever interacted with does the first step at a minimum. If a component never mounted, we would have never seen it!

Most interesting components are updated at some point. A purely static component—like, for example, a brand logo - might not ever update. But if a 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 there is a button that hides a component, chances are that component will be unmounted. If the app has multiple screens, it's likely that each screen (and all of its child components) will be unmounted. If a component is "alive" for the entire lifetime of your app, it won't be unmounted. But most components can get unmounted one way or another!

For more information see the React Official Docs and see this diagram for clear understanding of the components' lifecycle.

Introduction to lifecycle methods:

React components have several methods, called lifecycle methods, that are called at different parts of a component's lifecycle.

We've already learnt about two of the most common method lifecycle methods:constructor() and render()!

constructor() is the first method called during the mounting phase. render() is called later during 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 life cycle. constructor() only executes during the mounting phase, but render() executes during both the mounting and updating phase.

To understand the working of lifecycle methods let's make an <MyClock /> component.

```
class MyClock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      time:new Date()
    };
  }
  render(){
    return (
      <h3> {this.state.time.toLocaleTimeString()} </h3>
    );
  }
}
```

Let's start with the first lifecycle method!

componentDidMount() method:

The MyClock component we just created was just a static clock.It does not get updated on its own.

We would like to update this.state.time every second with a new Date.

For this we have the setInterval() method available in Javascript that calls the passed function after an interval.

So the code will be like:

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

But where should we put this code in our class?

render() isn't a good candidate.It executes during the mounting phase and the updating phase - that is too often. 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 ,that's good, but side-effects like this should be avoided in constructors because it violates the Single Responsibility Principle i.e. it's not a constructor's responsibility to start side-effects.

It's not the render() or constructor().Hence there is a new lifecycle method componentDidMount()

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

- The constructor
- render()
- componentDidMount()

In other words ,the method is called after the component is rendered.This is where we run statements that require that the component is already placed in the DOM.

We'll write our setInterval() code in the componentDidMount() function as below:

```
class MyClock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      time:new Date()
    };
  }

  componentDidMount(){
    setInterval(()=>{
      this.setState({
        time:new Date()
      })
    },1000);

  }

  render(){
    return (
      <h3> {this.state.time.toLocaleTimeString()} </h3>
    );
  }
}
```

Output as follows:

componentWillUnmount() method:

The clock is now workingBut it has a problem. We never told it to stop. It will keep running that functions until the user leaves or refreshes the page.

When the component is removed from the page, the timer will keep on ticking, trying to update the state of a component that's effectively gone. This means the users will have some JavaScript code running unnecessarily, which will hurt the performance of the 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
in your application. To fix, cancel all subscriptions and
asynchronous tasks in the componentWillUnmount method.
```

Imagine if the clock gets mounted and unmounted hundreds of times - eventually, this will cause the page to become sluggish because of all of the unnecessary work. Also warnings in the browser console will be shown. 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 this case this is a call to setInterval(), but components can have lots of other side-effects: loading external data with AJAX, doing manual tweaking of the DOM, setting a global value, and more.

JavaScript gives us the clearInterval() function. setInterval() can return an ID, which you can then pass into clearInterval() to clear it.So we'll update the code like this

```
    this.interval_id = setInterval(()=>{
      this.setState({
        time:new Date()
      })
    },1000);
```

Now, we want to continue to set up our setInterval() in componentDidMount(), but then we want to clear that interval when the clock is unmounted.

So there comes a new lifecycle method: componentWillUnmount(). componentWillUnmount() is called in the unmounting phase, right before the component is completely destroyed. It’s a useful time to clean up any of the component’s mess. Like:

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

In the MyClock component class it will look like:

```
class MyClock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      time:new Date()
    };
  }

  componentDidMount(){
    this.interval_id = setInterval(()=>{
      this.setState({
        time:new Date()
      })
    },1000);
  }

  componentWillUnmount(){
    clearInterval(this.interval_id);
  }

  render(){
    return (
      <h3> {this.state.time.toLocaleTimeString()} </h3>
    );
  }
}
```

componentDidUpdate() method:

We’ve looked at mounting (constructor(), render(), and componentDidMount()). We’ve looked at unmounting (componentWillUnmount()). Let’s look at the updating phase.

An update is caused by changes to props or state. We’ve already seen this happen a bunch of times. Every time setState() is called with new data, it has triggered an update. Every time props passed to a component are changed, it has 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 is new , is componentDidUpdate(). Just like componentDidMount() is a good place for mount-phase setup, componentDidUpdate() is a good place for update-phase work.

Lets see its working.

The componentDidUpdate method is called after the component is updated in the DOM.

```
class MyClock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      time:new Date(),
      isPrecise:false
    };
    this.togglePreciseMode =
this.togglePreciseMode.bind(this);
  }

  startInterval(){
    let delay=1000;
    if(this.state.isPrecise === true){
      delay=100;
    }
    console.log(delay);
    this.interval_id = setInterval(()=>{
      this.setState({
        time:new Date()
      })
    },delay);
  }

  componentDidMount(){
    this.startInterval();
  }

  componentDidUpdate(prevProps){
    if(this.state.isPrecise === prevProps.isPrecise){
      return;
    }
    clearInterval(this.interval_id);
    this.startInterval();
  }

  componentWillUnmount(){
    clearInterval(this.interval_id);
  }

  togglePreciseMode(){

    var isPrecise=this.state.isPrecise;
    this.setState({
      isPrecise:!isPrecise
    })
  }

  render(){
    var time = !this.state.isPrecise ?
this.state.time.toLocaleTimeString() :
this.state.time.toISOString() ;
    return (
      <div>
        <center>
          <h3> {time} </h3>
          <button onClick={this.togglePreciseMode}>Toggle
Precise Mode</button>
        </center>
      </div>
    );
  }
}
```

In the above example we can see that componentDidUpdate() receives an argument 'prevProps' which were the previous properties of the component before it got updated.

In the above example we can also see that if the precise mode is enabled the clock shows updates after every 100 microseconds and if it disabled it shows updates after every second.

Output as follows: