* Learn about Component Lifecycle methods of React.

 - Mounting:

After you do your initialization then you have prepared the code with basic requirements, states and props, you need your component to mount in the browser. This is done via browser DOM and the phase gives you the right React.js Hooks methods for a before and after fitting.

Here are the critical terms you should be adept in:

**Render:**

Render is what mounts the component onto the browser in this state. It is a classic method that gives the same output every time the same input is provided. It’s a standard function that is used extensively in the React.js coding framework.

**componentWillMount:**

This is a critical function to remember, as it is executed just before the reach component is about to mount. The mounting on the DOM is done after this stage, wherein you can enter all the things that you want to the program to do. It is also executed once in a lifecycle of a component and occurs before you render the program for the first time. It is used for initializing the states or props as well, making it a robust component to leverage.

**componentDidMount:**

This is the final React.js Hook method that is executed after the component mounts the DOM. It’s performed once in the lifecycle and occurs after the first rendering. Engineers can access the DOM via this method and initialize the appropriate JS libraries. You can access the DOM efficiently using this component. You can also initialize several other libraries that can be incorporated into the final output. You can make the right API calls under this method so that you can retrieve the data the right way.

 - Updating:

Updating starts when the component has been adopted on the browser. This can then grow by receiving new updates from the program. The user can interact with the program and the component can be updated accordingly. Developers can typically update the component in a few main ways. They can either send new props to the command or update the state entirely. Depending on the complexity or the scale of work, they can choose either method and get the program running. Here are Hook methods that are critical to understanding:

**shouldComponentUpdate:**

The method tells the program about the state of rendering when it is updated. If new props or rules are being updated, a rendering can be done or skipped. This is important to code in properly, as there are evolving states in the program as well. Updating the method as true/false is the proper approach. The default here is true, which can be changed as per the code.

**componentWillUpdate:**

This is executed when the prior method returns the answer of true. It’s then used to prepare the upcoming render, in the case where some previous calculation is necessary before returning a response. For more complex programs, this method can be used as well.

**componentDidUpdate:**

This is then executed when the updated component has been updated in the DOM as well. You can initiate new libraries to reload so that you can maintain an updated program throughout the process. Rendering can be triggered accordingly, as per the core requirement.

 - Unmounting:

The final stage of unmounting is essential as it doesn’t require the component and gets unmounted from the DOM. As the final state, it is designed to produce the outcome via unmounting. Here is the essential method used in unmounting:

**componentWillUnmount:**

This is the last method in the lifecycle as it pertains to the core unmounting and removal from the DOM. The cleaning up of the component is also performed here. This is used in the logging out of users when they want to clear out the program from their browser.

* Create an example code for each lifecycle method in the above three phases.
* Create a document in which all the methods should be explained with a code sample written in the document or pasted from your text editor.

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

### **constructor**

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

The constructor method is called, by React, every time you make a component:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### **getDerivedStateFromProps**

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

### **Example:**

The getDerivedStateFromProps method is called right before the render method:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));

### **render**

The render() method is required, and is the method that actual outputs HTML to the DOM.

### **Example:**

A simple component with a simple render() method:

class Header extends React.Component {

render() {

return (

<h1>This is the content of the Header component</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### **componentDidMount**

The componentDidMount() method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

### **Example:**

At first my favorite color is red, but give me a second, and it is yellow instead:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

render() {

return (

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

## Updating

The next phase in the lifecycle is when a component is updated.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

### **getDerivedStateFromProps**

Also at updates the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.

This is still the natural place to set the state object based on the initial props.

The example below has a button that changes the favorite color to blue, but since the getDerivedStateFromProps() method is called, which updates the state with the color from the favcol attribute, the favorite color is still rendered as yellow:

### **Example:**

If the component gets updated, the getDerivedStateFromProps() method is called:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

static getDerivedStateFromProps(props, state) {

return {favoritecolor: props.favcol };

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));

### **shouldComponentUpdate**

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true.

The example below shows what happens when the shouldComponentUpdate() method returns false:

### **Example:**

Stop the component from rendering at any update:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

shouldComponentUpdate() {

return false;

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### **Example:**

shouldComponentUpdate() method returns true:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

shouldComponentUpdate() {

return true;

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### **render**

The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.

The example below has a button that changes the favorite color to blue:

### **Example:**

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

changeColor = () => {

this.setState({favoritecolor: "blue"});

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<button type="button" onClick={this.changeColor}>Change color</button>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### **getSnapshotBeforeUpdate**

In the getSnapshotBeforeUpdate() method you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

When the component has been mounted, a timer changes the state, and after one second, the favorite color becomes "yellow".

This action triggers the update phase, and since this component has a getSnapshotBeforeUpdate() method, this method is executed, and writes a message to the empty DIV1 element.

Then the componentDidUpdate() method is executed and writes a message in the empty DIV2 element:

### **Example:**

getSnapshotBeforeUpdate() method to find out what the state object looked like before the update:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

getSnapshotBeforeUpdate(prevProps, prevState) {

document.getElementById("div1").innerHTML =

"Before the update, the favorite was " + prevState.favoritecolor;

}

componentDidUpdate() {

document.getElementById("div2").innerHTML =

"The updated favorite is " + this.state.favoritecolor;

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<div id="div1"></div>

<div id="div2"></div>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

### **componentDidUpdate**

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

The example below might seem complicated, but all it does is this:

When the component is mounting it is rendered with the favorite color "red".

When the component has been mounted, a timer changes the state, and the color becomes "yellow".

This action triggers the update phase, and since this component has a componentDidUpdate method, this method is executed and writes a message in the empty DIV element:

### **Example:**

The componentDidUpdate method is called after the update has been rendered in the DOM:

class Header extends React.Component {

constructor(props) {

super(props);

this.state = {favoritecolor: "red"};

}

componentDidMount() {

setTimeout(() => {

this.setState({favoritecolor: "yellow"})

}, 1000)

}

componentDidUpdate() {

document.getElementById("mydiv").innerHTML =

"The updated favorite is " + this.state.favoritecolor;

}

render() {

return (

<div>

<h1>My Favorite Color is {this.state.favoritecolor}</h1>

<div id="mydiv"></div>

</div>

);

}

}

ReactDOM.render(<Header />, document.getElementById('root'));

## Unmounting

The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

* componentWillUnmount()

### **componentWillUnmount**

The componentWillUnmount method is called when the component is about to be removed from the DOM.

### **Example:**

class Container extends React.Component {

constructor(props) {

super(props);

this.state = {show: true};

}

delHeader = () => {

this.setState({show: false});

}

render() {

let myheader;

if (this.state.show) {

myheader = <Child />;

};

return (

<div>

{myheader}

<button type="button" onClick={this.delHeader}>Delete Header</button>

</div>

);

}

}

class Child extends React.Component {

componentWillUnmount() {

alert("The component named Header is about to be unmounted.");

}

render() {

return (

<h1>Hello World!</h1>

);

}

}

ReactDOM.render(<Container />, document.getElementById('root'));