|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Mounting** | | **Updating** | | **Unmounting** | |
| When React creates an instance of a component and inserts it into the DOM (Mounting) | | If props or state of a component are changed, an update of the component is performed and component is re-rendered | | At some point our components will be removed from the DOM again. That process is called "Unmounting" | |
| ***1*** | componentWillReceiveProps() | ***1*** | componentWill***Unmount***() |
| ***1*** | Constructor() | ***2*** | shouldComponent***Update***() |
| ***2*** | componentWill***Mount***() | ***3*** | componentWill***Update***() |
| ***3*** | render() | ***4*** | render() |
| ***4*** | componentDid***Mount***() | ***5*** | componentDid***Update***() |

[**React Lifecycle Methods – A Deep Dive**](https://programmingwithmosh.com/javascript/react-lifecycle-methods/)

**What are React lifecycle methods?**

You can think of React lifecycle methods as the series of events that happen from the birth of a React component to its death.

* **Mounting**–Birth of your component
* **Update**– Growth of your component
* **Unmount**– Death of your component

1. componentWillMount (Legacy, should not be used)
2. componentDidMount
3. componentWillReceiveProps (Legacy, should not be used)
4. shouldComponentUpdate
5. componentWillUpdate
6. componentDidUpdate
7. componentWillUnmount

Common React Lifecycle Methods

**render()**

The *render()* method is the most used lifecycle method. You will see it in all React classes. This is because*render()* is the only required method within a class component in React.

As the name suggests it handles the rendering of your component to the UI. It happens during the **mounting** and **updating** of your component.

Below is an example of a simple *render()* in React.

class Hello extends Component{

render(){

return <div>Hello {this.props.name}</div>

}

}

As you can see in the example above, the *render()* method returns JSX that is displayed in the UI. A *render()* can also return a null if there is nothing to render for that component.

**A *render()* method has to be pure with no side-effects.**

React requires that your *render()* is pure. Pure functions are those that do not have any side-effects and will always return the same output when the same inputs are passed. This means that you can not *setState()* within a*render().*

You cannot modify the component state within the *render()*.

If you do need to modify state that would have to happen in the other lifecycle methods, therefore keeping *render()* pure.

Furthermore, keeping your *render()* simple and clean without state updates makes your app maintainable.

1. *This render method is called always when you call* ***setState().***
2. *When render method is called it returns a new virtual dom structure of the component with values used in setState()*
3. *And the one thing that really makes React still shine in this case is that it doesn't change the real DOM unless the virtual DOM has been changed. So if my render function returns the same thing 2 times in a row, the real DOM doesn't get changed at all*
4. *Implement* ***shouldComponentUpdate()*** *to render only when state or properties change.*
5. *Switch to extending a* ***PureComponent****, which already implements a shouldComponentUpdate method internally for shallow comparisons.*

**componentDidMount()**

***componentDidMount()***is called as soon as the component is mounted and ready. This is a good place to **initiate API calls**, if you need to load data from a remote endpoint.

Unlike the *render()* method, *componentDidMount()***allows the use of *setState()*.**

You can modify the component state within the *componentDidMount()*, but use it with caution.

**componentDidUpdate()**

This lifecycle method is invoked as soon as the updating happens. The most common use case for the *componentDidUpdate()*method is updating the DOM in response to prop or state changes.

**You can call *setState()*in this lifecycle**, but keep in mind that you will need to wrap it in a condition to check for state or prop changes from previous state. Incorrect usage of *setState()* can lead to an infinite loop.

Take a look at the example below that shows a typical usage example of this lifecycle method.

componentDidUpdate(prevProps) {

//Typical usage, don't forget to compare the props

if (this.props.userName !== prevProps.userName) {

this.fetchData(this.props.userName);

}

}

Notice in the above example that we are comparing the current props to the previous props. This is to check if there has been a change in props from what it currently is. In this case, there won’t be a need to make the API call if the props did not change.

**componentWillUnmount()**

As the name suggests this lifecycle method is called just before the component is unmounted and destroyed. If there are any cleanup actions that you would need to do, this would be the right spot.

You **CANNOT** modify the component state in *componentWillUnmount* lifecycle.

This component will never be re-rendered and because of that we cannot call *setState()* during this lifecycle method.

componentWillUnmount() {

window.removeEventListener('resize', this.resizeListener)

}

Common cleanup activities performed in this method include, clearing timers, cancelling api calls, or clearing any caches in storage.

Uncommon React Lifecycle Methods

We now have a good idea of all the commonly used React lifecycle methods. Besides that, there are other lifecycle methods that React offers which are sparingly used or not used at all.

**shouldComponentUpdate()**

This lifecycle can be handy sometimes when you don’t want React to render your state or prop changes.

Anytime *setState()* is called, the component re-renders by default. The *shouldComponentUpdate()* method is used to let React know if a component is not affected by the state and prop changes.

**Caution:**Most importantly, do not always rely on it to prevent rendering of your component, since it can lead to several bugs.

shouldComponentUpdate(nextProps, nextState) {

return this.props.title !== nextProps.title ||

this.state.input !== nextState.input }

As shown in the example above, this lifecycle should always return a boolean value to the question, “***Should I re-render my component?***”

**static getDerivedStateFromProps()**

This is one of the newer lifecycle methods introduced very recently by the React team.

This will be a safer alternative to the previous lifecycle method *componentWillReceiveProps().*

It is called just before calling the *render*() method.

This is a *static* function that does not have access to “*this*“.  *getDerivedStateFromProps()* returns an object to update *state* in response to *prop* changes. It can return a *null* if there is no change to state.

This method also exists only for rare use cases where the state depends on changes in props in a component.

static getDerivedStateFromProps(props, state) {

if (props.currentRow !== state.lastRow) {

return {

isScrollingDown: props.currentRow > state.lastRow,

lastRow: props.currentRow,

};

}

// Return null to indicate no change to state.

return null;

}

*Keep in mind that this lifecycle method is fired on****every****render.*

An example use case where this method may come in handy would be a <Transition> component that compares its previous and next children to decide which ones to animate in and out.

**getSnapshotBeforeUpdate()**

*getSnapshotBeforeUpdate()* is another new lifecycle method introduced in React recently. This will be a safer alternative to the previous lifecycle method *componentWillUpdate().*

getSnapshotBeforeUpdate(prevProps, prevState) {

// ...

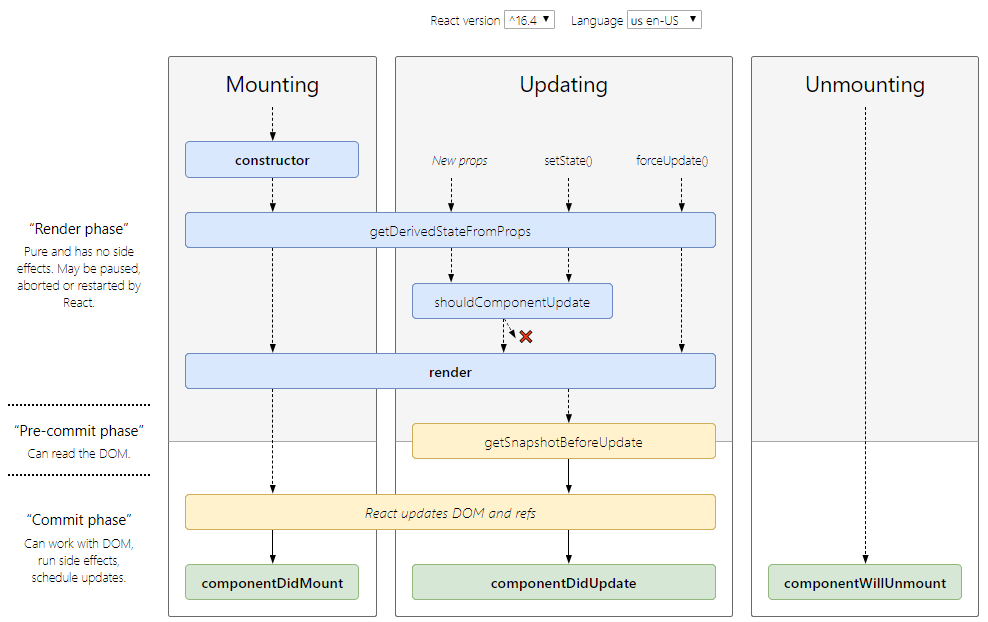
}

It is called right before the DOM is updated. The value that is returned from *getSnapshotBeforeUpdate()* is passed on to *componentDidUpdate().*

Keep in mind that this method should also be used rarely or not used at all.

Resizing the window during an async rendering is a good use-case of when the *getSnapshotBeforeUpdate()*can be utilized.

**React Component Lifecycle Diagram**



Conceptually, React does work in two phases:

1. The **render** phase determines what changes need to be made to e.g. the DOM. During this phase, React calls render and then compares the result to the previous render.
2. The **commit** phase is when React applies any changes. (In the case of React DOM, this is when React inserts, updates, and removes DOM nodes.) React also calls lifecycles like componentDidMount and componentDidUpdate during this phase.

***Recap***

* React component lifecycle has three categories – Mounting, Updating and Unmounting.

1. The *render()*is the most used lifecycle method.
   1. It is a pure function.
   2. You **CANNOT** set state in render()
2. The *componentDidMount*() happens as soon as your component is mounted.
   1. You **CAN SET STATE** here but with caution.
3. The *componentDidUpdate()*happens as soon as the updating happens.”**prevProps”** is used to compare
   1. You **CAN SET STATE** here but with caution.
4. The *componentWillUnmount()*happens just before the component unmounts and is destroyed.
   1. This is a good place to cleanup all the data.
   2. You **CANNOT** set state here.
5. The *shouldComponentUpdate()*can be used rarely.
   1. It can be called if you need to tell React not to re-render for a certain state or prop change.
   2. This needs to be used with caution only for certain performance optimizations.
6. The two “new” lifecycle methods are ***getDerivedStateFromProps****()*and ***getSnapshotBeforeUpdate****().*
   1. They need to be used only occasionally.
   2. Not many examples are out there for these two methods and they are still being discussed and will have more references in the future.

# Strict Mode

StrictMode is a tool for highlighting potential problems in an application. Like Fragment, StrictMode does not render any visible UI. It activates additional checks and warnings for its descendants.

import React from 'react';

function ExampleApplication() {

return (

<div>

<Header />

**<React.StrictMode>**

<div>

<ComponentOne />

<ComponentTwo />

</div>

**</React.StrictMode>**

<Footer />

</div>

);

}

In the above example, strict mode checks will not be run against the Header and Footer components. However, ComponentOne and ComponentTwo, as well as all of their descendants, will have the checks.

StrictMode currently helps with:

1. [Identifying components with **unsafe lifecycles**](https://reactjs.org/docs/strict-mode.html#identifying-unsafe-lifecycles)
2. [Warning about legacy string ref **API usage**](https://reactjs.org/docs/strict-mode.html#warning-about-legacy-string-ref-api-usage)
3. [Warning about deprecated **findDOMNode** usage](https://reactjs.org/docs/strict-mode.html#warning-about-deprecated-finddomnode-usage)
4. [Detecting unexpected **side effects**](https://reactjs.org/docs/strict-mode.html#detecting-unexpected-side-effects)
5. [Detecting legacy **context API**](https://reactjs.org/docs/strict-mode.html#detecting-legacy-context-api)