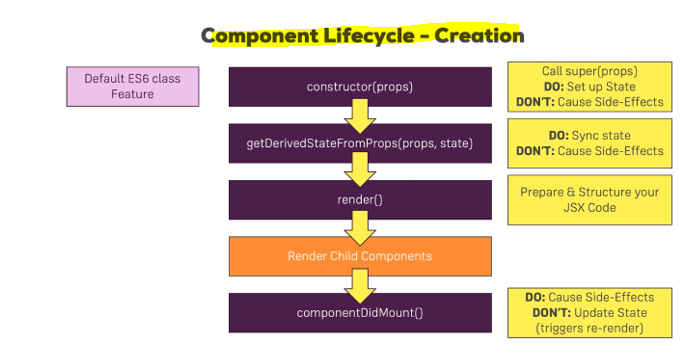
# Component Lifecycle hooks/methods(only available in Class-based components)

**it's only available in class-based components. For functional component handling lifecycle of component we have react hooks.**

We've got these methods which we can add to any class-based component and React will execute them for us and they will run at different points of time and we can do different things with them.

for example

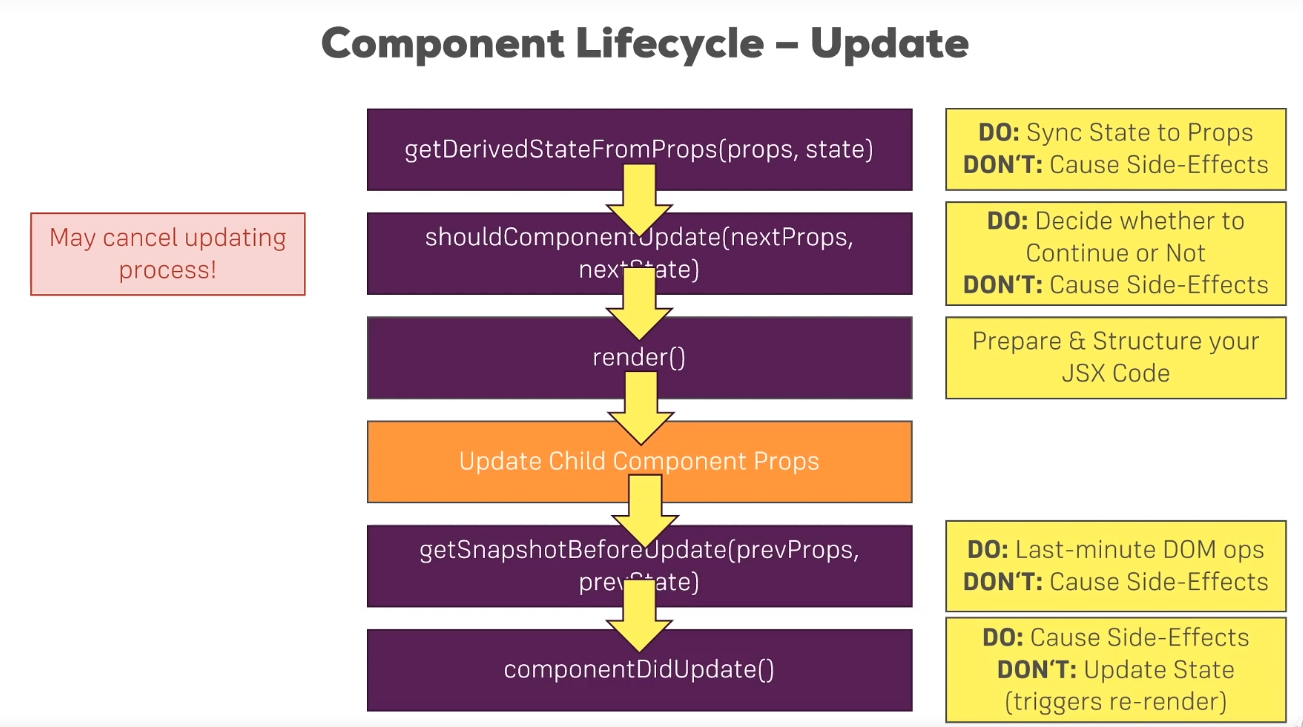
fetch data from the web or to do some cleanup work before a component is removed from the DOM

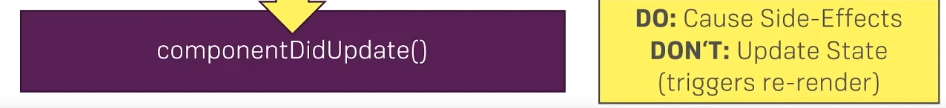




**here, you can cause side effects. That is a typical hook you would use for making an HTTP request to get new data from the web.**

**What you shouldn't do in here is update the state, so don't call set state in here unless it's in, let's say the then block of a promise after you sent an HTTP request but don't call set state in here synchronously. So** **you can definitely set up some code that executes in the future which then updates the state, for example when the response from the server is back but don't do it right away when componentDidMount runs that you immediately call set state because that will trigger a re-render cycle and that is bad for performance.**



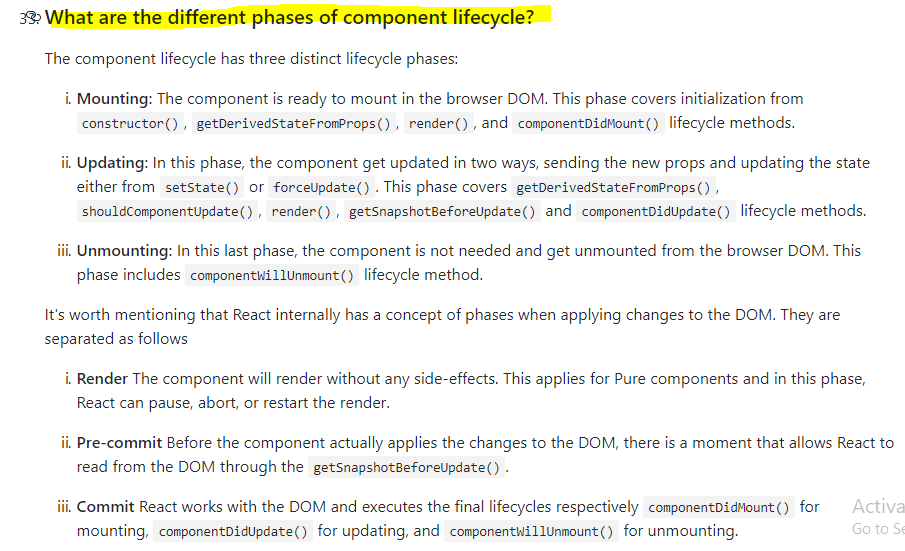


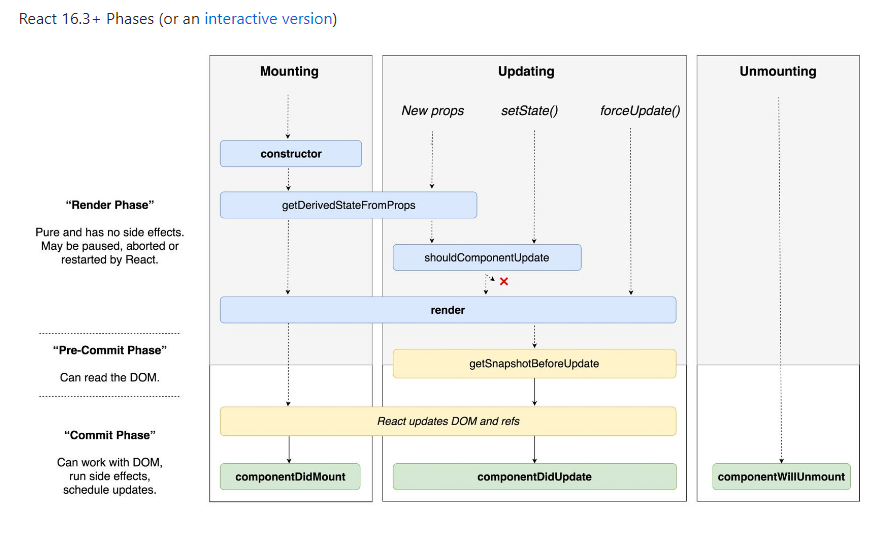
once we're done with the update, componentDidUpdate is called. A lifecycle hook that signals that you are now done with the updating, that the render method has been executed

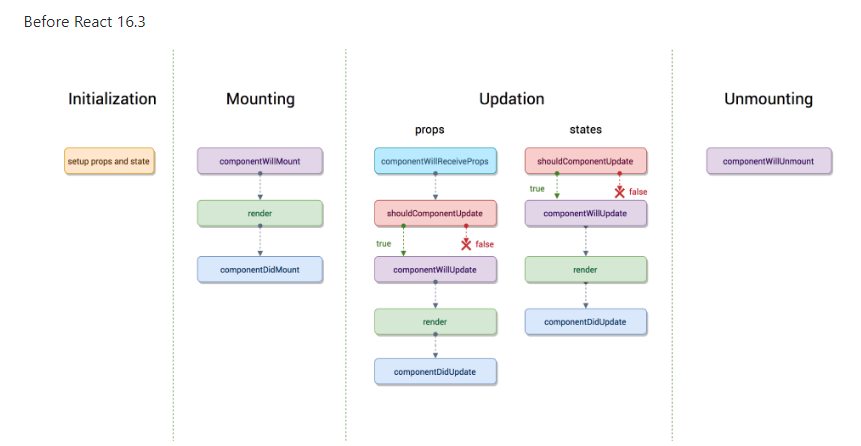
and here you can now cause side effects, so here you could now make an HTTP request, though you'll have to watch out to not enter an infinite loop here

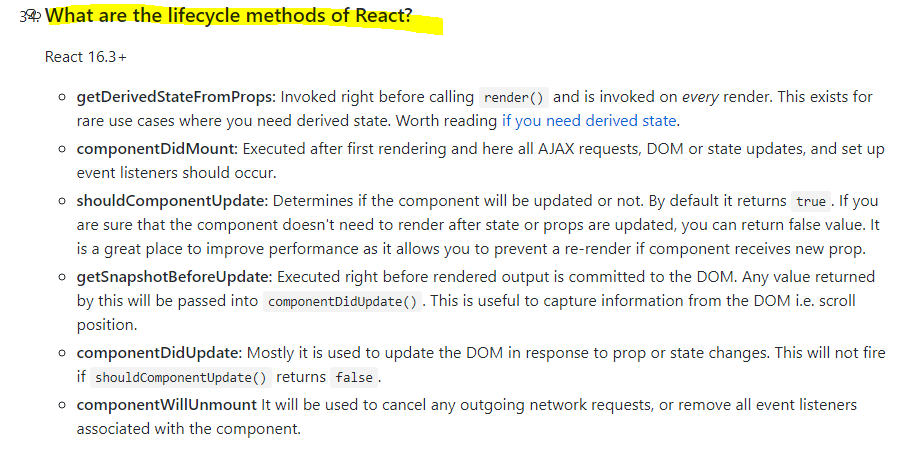
if you make an HTTP request and you get back a response and you then update your component and then this cycle starts .this is a typical problem you'll be facing. Now what you shouldn't do here outside of the, let's say then block of a promise of an HTTP request is updating the state with set state.

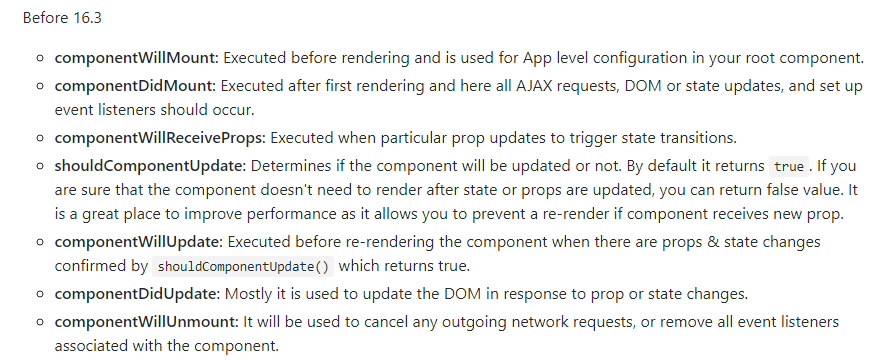
It's fine to do it as a result of some async task you're kicking off here but you should not call it synchronously in componentDidUpdate because that will simply lead to an unnecessary re-render cycle.











# **Very good reference explanation**

<https://medium.com/@baphemot/understanding-reactjs-component-life-cycle-823a640b3e8d>

## **Understanding React — Component life-cycle**

[](https://medium.com/@baphemot)

[Bartosz Szczeciński](https://medium.com/@baphemot)

[Sep 6, 2017](https://medium.com/@baphemot/understanding-reactjs-component-life-cycle-823a640b3e8d) · 8 min read

React provides developers with many methods or “hooks” that are called during the life-cycle of an component, which allows us to update the UI and application state. Knowing when to use which of them is crucial to properly understanding how to work with React.

## Update:

React 16.3 introduced two more life-cycle methods and depracated few of them, be sure to check up the follow-up at <https://medium.com/@baphemot/understanding-react-react-16-3-component-life-cycle-23129bc7a705>

## constructor

constructors are the basic of OOP — this is a special function that will get called whenever a new object is created. It’s very important to call a special function super in cases where our class extends any other class that also has a defined constructor. Calling this special function will call the constructor of our parent class and allow it to initialize itself. This is why we have access to this.props only after we’ve initially called super.

As mentioned, constructors are perfect for setting up our Component — create any fields (variables starting with this.) or initialize state based on props received.

This is also the only place where you are expected to change / set the state by directly overwriting the this.state fields. In all other instances remember to use this.setState

**DO**

* set initial state
* if not using class properties syntax — prepare all class fields and bind functions that will be passed as callbacks

**DON’T**

* cause any side effects (AJAX calls etc.)

## deprecated — componentWillMount

This is a somehow special case — componentWillMount does not differ much from constructor - it is also called once only in the initial mounting life-cycle. Historically there were some reasons to use componentWillMountover constructor [see this react-redux issue](https://github.com/reactjs/react-redux/issues/129) but please keep in mind that the practice described there is since deprecated.

Many will be tempted to use this function in order to send a request to fetch data and expect the data to be available before the initial render is ready. This is not the case — while the request will be initialized before the render, it will not be able to finish before the render is called.

Additionally, with the changes to React Fiber (post React 16 beta release) this function might end up being called multiple times before the initial render is called so might result in triggering multiple side-effects. Due to this fact it is **not** recommended to use this function for any side-effect causing operations.

It is important to note that this function is called when using server-side-rendering while it counterpart — componentDidMount will not be called on the server but on the client in such case. So if some side-effect isdesired on the server part this function should be used as an exception.

A setState used in this function is “free” and will not trigger a re-render.

**DO**

* update state via this.setState
* perform last minute optimization
* cause side-effects (AJAX calls etc.) **in case of server-side-rendering only**

**DON’T**

* cause any side effects (AJAX calls etc.) on client side

## deprecated — componentWillReceiveProps(nextProps)

This function will be called in each update life-cycle caused by changes to props (parent component re-rendering) and will be passed an object map of all the props passed, no matter if the prop value has changed or not since previous re-render phase.

This function is ideal if you have a component whose parts of state are depending on props passed from parent component as calling this.setState here will not cause an extra render call.

Please keep in mind that due to the fact that the function is called with all props, even those that did not change it is expected the developers implement a check to determine if the actual value has changed, for example:

componentWillReceiveProps(nextProps) {

**if**(nextProps.myProp !== **this**.props.myProps) {

// nextProps.myProp has a different value than our current prop

// so we can perform some calculations based on the new value

}

}

Due to the fact that with React Fiber (post 16 beta) this function might be called multiple times before the renderfunction is actually called it is **not**recommended to use any side-effect causing operations here.

**DO**

* sync state to props

**DON’T**

* cause any side effects (AJAX calls etc.)

## shouldComponentUpdate(nextProps, nextState, nextContext)

By default, all class based Components will re-render themselves whenever the props they receiver, their state or context changes. If re-rendering the component is computation heavy (e.g. generating a chart) or is not recommended for some performance reasons, the developer is given access to a special function which will be called in the update cycle.

This function will be called internally with next values of props, state and object. Developer can use those to verify that the change requires a re-render or not and return false to prevent the re-rendering from happening. In other case, you are expected to return true.

**DO**

* use for increasing performance of poor performing Components

**DON’T**

* cause any side effects (AJAX calls etc.)
* call this.setState

## deprecated — componentWillUpdate(nextProps, nextState)

If the shouldComponentUpdate function is not implemented, or it decided that the component should update in this render cycle, another life-cycle function will be called. This function is commonly used to perform state and props synchronization for when parts of your state are based on props.

In cases where shouldComponentUpdate is implemented, this function can be used instead of componentWillReceiveProps as it will be called only when the component will actually be re-rendered.

Similarly to all other componentWill\* functions, this function might end up called multiple times before render so it it **not** advised to perform side-effects causing operations here.

**DO**

* synchronize state to props

**DON’T**

* cause any side effects (AJAX calls etc.)

## componentDidUpdate(prevProps, prevState, prevContext)

This function will be called after render is finished in each of the re-render cycles. This means that you can be sure that the component and all its sub-components have properly rendered itself.

Due to the fact that this is the only function that is guaranteed to be called only once in each re-render cycle it is recommended to use this function for any side-effect causing operations. Similarly to componentWillUpdateand componentWillReceiveProps this function is called with object-maps of **previous** props, state and context, even if no actual change happened to those values. Because of that developers are expected to manually check if given value changed and only then perform various update operations:

componentDidUpdate(prevProps) {

if(prevProps.myProps !== this.props.myProp) {

// this.props.myProp has a different value

// we can perform any operations that would

// need the new value **and**/**or** cause side-effects

// like AJAX calls with the new value - this.props.myProp

}

}

**DO**

* cause side effects (AJAX calls etc.)

**DON’T**

* call this.setState as it will result in a re-render

An exception to the above rule is updating the state based on some DOM property which can be only computed once a component has re-rendered (e.g. position / dimensions of some DOM nodes). Please take extra care to prevent against updating if the value did not in fact change as it might result in a render loop.

## componentDidCatch(errorString, errorInfo)

A new addition in React 16 — this life-cycle method is special in way that it can react to events happening in the child component, specifically to any uncaught errors happening in any of the child components.

With this addition you can make your parent-element handle the error by — for example — setting the error info in state and returning appropriate message in its render, or logging to reporting system, e.g.:

componentDidCatch(errorString, errorInfo) {

this.**setState**({

**error**: errorString

});

ErrorLoggingTool.**log**(errorInfo);

}**render**() {

**if**(this.state.**error**) **return** <ShowErrorMessage **error**={this.state.**error**} />

**return** (

// render normal component output

);

}

When an error happens, the function will be called with:

* errorString — the .toString() message of the error
* errorInfo — an object with a single field componentStack which represent the stack trace back to where the error occured, e.g.:

in Thrower

in div (created **by** App)

in App

## componentDidMount

This function will be called only once in the whole life-cycle of a given component and it being called signalizes that the component — and all its sub-components — rendered properly.

Since this function is guaranteed to be called only once it is a perfect candidate for performing any side-effect causing operations such as AJAX requests.

**DO**

* cause side effects (AJAX calls etc.)

**DON’T**

* call this.setState as it will result in a re-render

An exception to the above rule is updating the state based on some DOM property which can be only computed once a component has re-rendered (e.g. position / dimensions of some DOM nodes). Please take extra care to prevent against updating if the value did not in fact change as it might result in a render loop.

## componentWillUnmount

Use this function to “clean up” after the component if it takes advantage of timers (setTimeout, setInterval), opens sockets or performs any operations we need to close / remove when no longer needed.

**DO**

* remove any timers or listeners created in lifespan of the component

**DON’T**

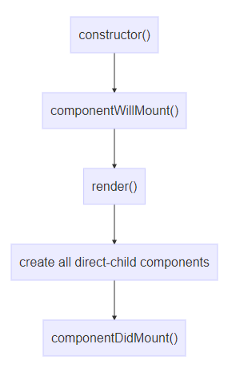
* call this.setState, start new listeners or timers

## Component cycles

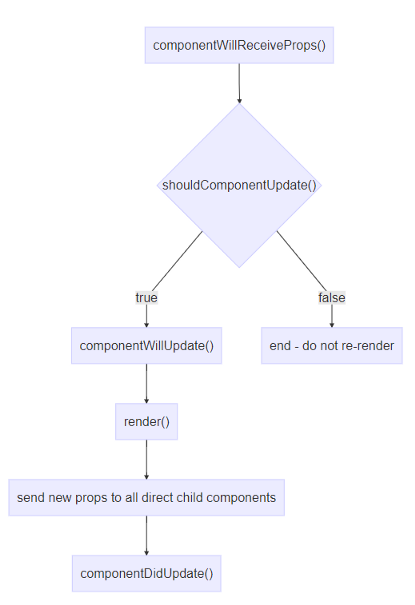
There are multiple reasons a component might re-render, and in each of them different functions are called allowing the developer to update certain parts of the Component.

**Component creation**

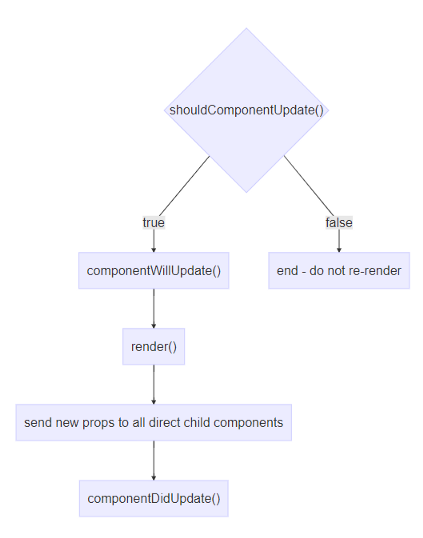
The first cycle is the creation for component, which usually happens the first time a component is encountered in the parsed JSX tree:



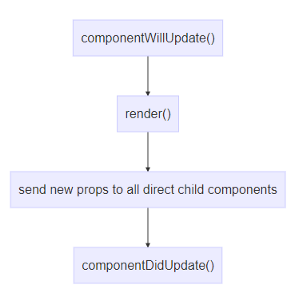
**Component re-rendering due to re-rendering of the parent component**



**Component re-rendering due to internal change (e.g. a call tothis.setState())**



**Component re-rendering due to call to this.forceUpdate**



**Component re-rendering due to catching an error**

Introduced in React 16 as “ErrorBoundaries”. A component can define a special layer which can catch errors and provide a new life-cycle method — componentDidCatch - which allows developers to provide self-repair actions for recovery or graceful handling of errors.

