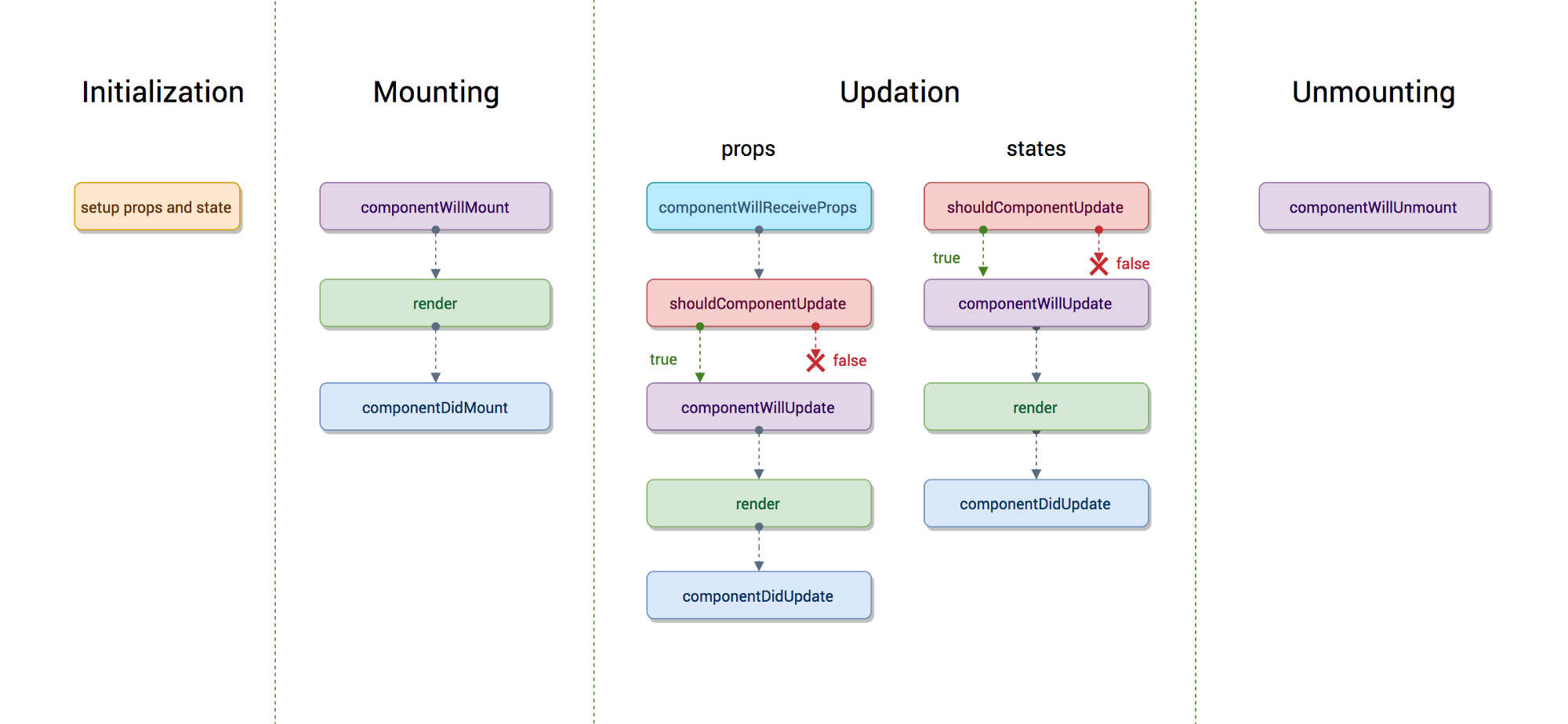
**Four phases of a React component**

The React component, like anything else in the world, goes through the following phases

* **Initialization**
* **Mounting**
* **Update**
* **Unmounting**

The following image is the visual representation of the phases and the methods of ReactJs lifecycle.



## **Lifecycle Methods**

* **componentWillMount** is executed before rendering, on both the server and the client side.
* **componentDidMount** is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as **setTimeout** or **setInterval**. We are using it to update the state so we can trigger the other lifecycle methods.
* **componentWillReceiveProps** is invoked as soon as the props are updated before another render is called. We triggered it from **setNewNumber** when we updated the state.
* **shouldComponentUpdate** should return **true** or **false** value. This will determine if the component will be updated or not. This is set to **true** by default. If you are sure that the component doesn't need to render after **state** or **props** are updated, you can return **false** value.
* **componentWillUpdate** is called just before rendering.
* **componentDidUpdate** is called just after rendering.
* **componentWillUnmount** is called after the component is unmounted from the dom. We are unmounting our component in **main.js**.

In the following example, we will set the initial **state** in the constructor function. The **setNewnumber** is used to update the **state**. All the lifecycle methods are inside the Content component.

### App.jsx

import React from 'react';

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

data: 0

}

this.setNewNumber = this.setNewNumber.bind(this)

};

setNewNumber() {

this.setState({data: this.state.data + 1})

}

render() {

return (

<div>

<button onClick = {this.setNewNumber}>INCREMENT</button>

<Content myNumber = {this.state.data}></Content>

</div>

);

}

}

class Content extends React.Component {

componentWillMount() {

console.log('Component WILL MOUNT!')

}

componentDidMount() {

console.log('Component DID MOUNT!')

}

componentWillReceiveProps(newProps) {

console.log('Component WILL RECIEVE PROPS!')

}

shouldComponentUpdate(newProps, newState) {

return true;

}

componentWillUpdate(nextProps, nextState) {

console.log('Component WILL UPDATE!');

}

componentDidUpdate(prevProps, prevState) {

console.log('Component DID UPDATE!')

}

componentWillUnmount() {

console.log('Component WILL UNMOUNT!')

}

render() {

return (

<div>

<h3>{this.props.myNumber}</h3>

</div>

);

}

}

export default App;

### main.js

import React from 'react';

import ReactDOM from 'react-dom';

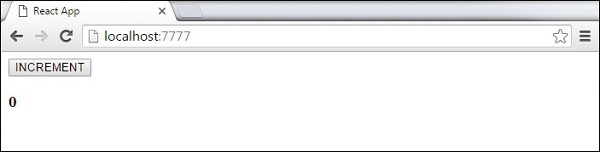
import App from './App.jsx';

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

setTimeout(() => {

ReactDOM.unmountComponentAtNode(document.getElementById('app'));}, 10000);

After the initial render, we will get the following screen.



ReactJs lifecycle phases and methods

To visualize the implementation of these lifecycle hooks we will create a music player React app named Contra music player. Let’s start our discussion on these phases.

**1) Initialization**

In this phase the React component prepares for the upcoming tough journey, by setting up the initial states and default props, if any.

Contra Music player app’s initializations would look like

The component is setting up the initial state in the constructor, which can be changed later by using thesetState method.

The defaultProps is defined as a property of Component to define all the default value of props, which can be overridden with new prop values.

By rendering like<ContraMusicPlayer/> Contra Music player will start with *volume* 70% in *paused* state with *dark theme*.

By rendering like <ContraMusicPlayer theme="light"/> Contra Music player will start with *volume* 70% in *paused* state with *light theme*.

**2) Mounting**

After preparing with basic needs, state and props, our React Component is ready to mount in the browser DOM. This phase gives hook methods for before and after mounting of components. The methods which gets called in this phase are

* **componentWillMount**is executed just before the React Component is about to mount on the DOM. Hence, after this method the component **will**mount. All the things that you want to do before a component mounts has to be defined here.  
  This method is executed once in a lifecycle of a component and before first render.  
  ***Usage:***componentWillMount is used for initializing the states or props, there is a huge debate going on to merge it with the constructor.
* **render**mounts the component onto the browser. This is a pure method, which means it gives the same output every time the same input is provided.

The render method for our music player may look like this

* **componentDidMount** this is the hook method which is executed after the component **did**mount on the dom.   
  This method is executed once in a lifecycle of a component and after the first render.  
  As, in this method, we can access the DOM, we should initialize JS libraries like D3 or Jquery which needs to access the DOM.  
  ***Usage:***In our Contra music player app, we want to draw the sound wave graphs of the full song, this is the right method to integrate with the D3 or other third party Javascript libraries.

**3) Update**

This phase starts when the react component has taken birth on the browser and grows by receiving new updates. The component can be updated by two ways, sending new props or updating the state.

Let’s see the list of hook methods when the current state is updated by calling setState

* **shouldComponentUpdate**tells the React that when the component receives new props or state is being updated, should React re-render or it can skip rendering?   
  This method is a question, **should** the **Component** be **Update**d?   
  Hence this method should return true or false, and accordingly the component would be re-rendered or skipped. By default, this method return true. ***Usage:***The example is one of the cases where I would like to re-render the component only when the props status changes.

This method is generally used when rendering is a very heavy method, then you should avoid render every now and then. For example, suppose for every render, the component generates thousand prime numbers, let’s consider some app has this kind of logic, then we can control when it is required then only the component is rendered.

* **componentWillUpdate**is executed only after the shouldComponentUpdatereturns true. This method is only used to do the preparation for the upcoming render, similar to componentWillMount or constructor.  
  There can be some use case when there needs some calculation or preparation before rendering some item, this is the place to do so.
* **render**And then the component gets rendered.
* **componentDidUpdate**is executed when the new updated component has been updated in the DOM. This method is used to re trigger the third party libraries used to make sure these libraries also update and reload themselves.

**The list of methods that will get called when the parent sends new props are as follows:**

* **componentWillReceiveProps**gets executed when the props have changed and is not first render. Sometimes state depends on the props, hence whenever props changes the state should also be synced. This is the method where it should be done.  
  The similar method for the state doesn’t exist before state change because the props are read only within a component and can never be dependent on the state.  
  ***Usage:***This is how the state can be kept synced with the new props.

The rest of the methods behave exactly same defined above, in terms of state as well.

* **shouldComponentUpdate**
* **componentWillUpdate**
* **render**
* **componentDidUpdate**

**4) Unmounting**

In this phase, the component is not needed and the component will get unmounted from the DOM. The method which is called in this phase

* **componentWillUnmount**This method is the last method in the lifecycle. This is executed just before the component gets removed from the DOM. ***Usage:***In this method, we do all the cleanups related to the component.   
  For example, on logout, the user details and all the auth tokens can be cleared before unmounting the main component.