

React Components Lifecycle & Forms




What is Component Lifecycle in React?

React components have three main stages in their lifecycle :

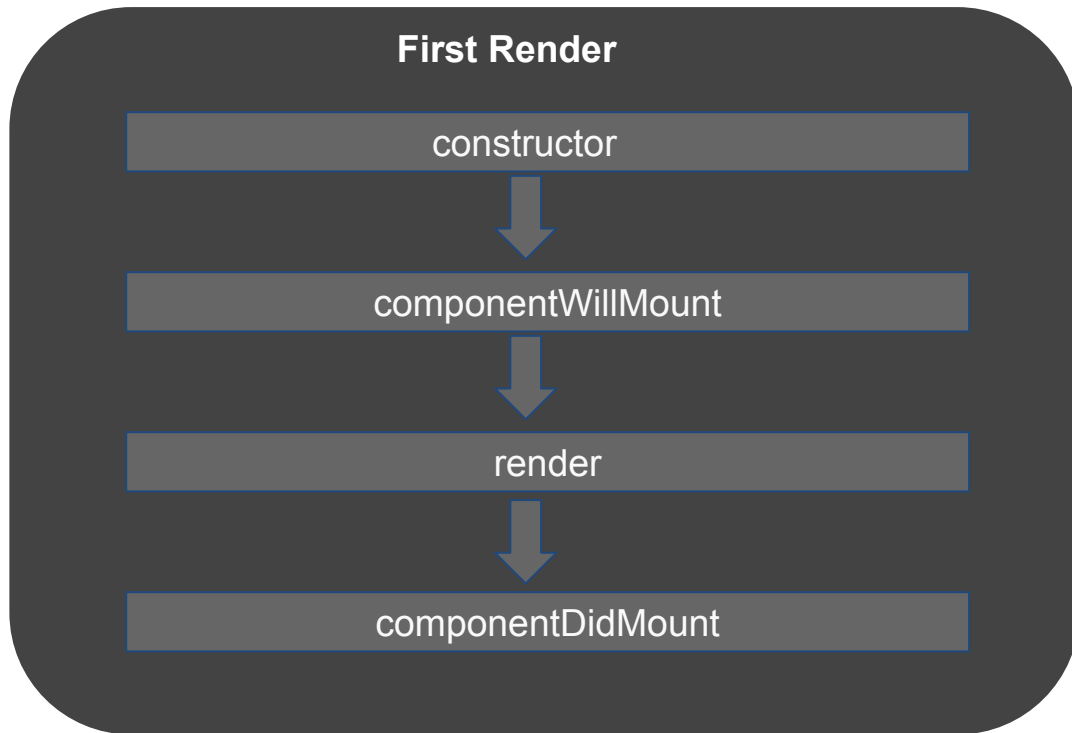
- Mounting or initialization.
- Updation in state or props.
- Unmounting.

Each component has several "lifecycle methods" that you can override to run code at particular times in the process. Methods prefixed with **will** are called right before something happens, and methods prefixed with **did** are called right after something happens.



Mounting

These methods are called when an instance of a component is being created and inserted into the DOM:



Deprecated and New in 16.3.x and onwards



Deprecated -

[_UNSAFE_ComponentWillMount\(\)](#)

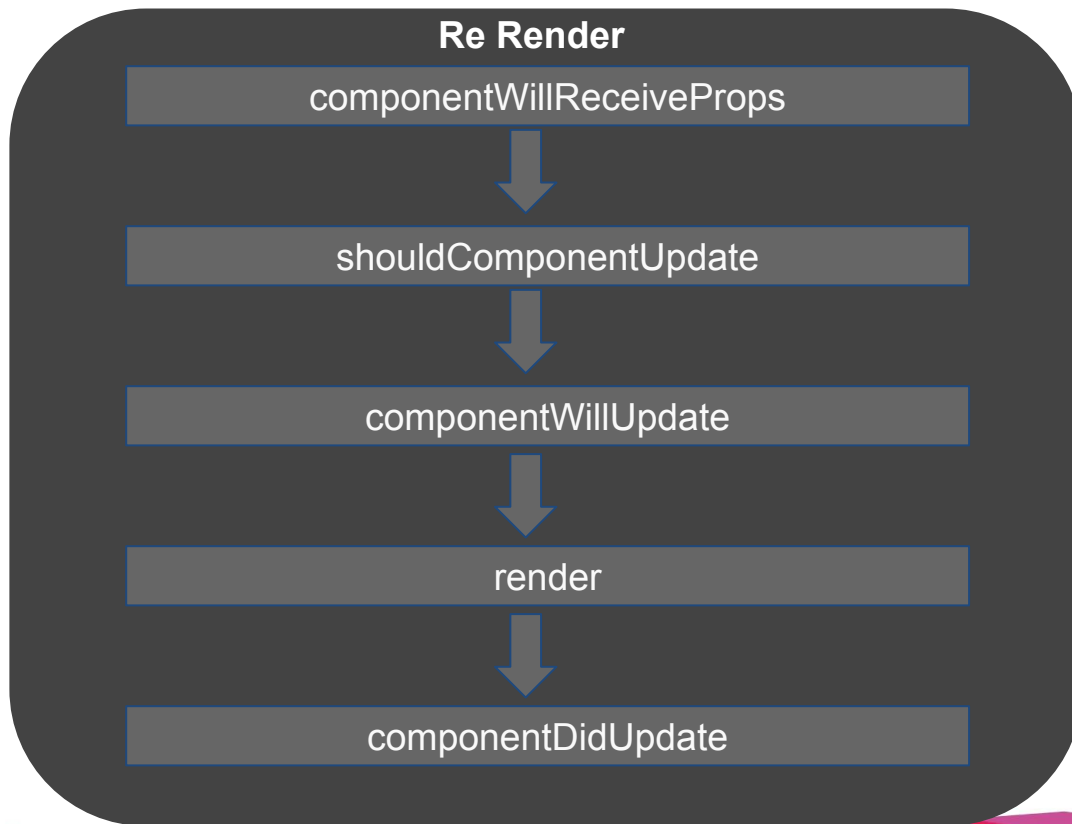
New-

[static getDerivedStateFromProps\(\)](#)



Updating

An update can be caused by changes to props or state. These methods are called when a component is being re-rendered



Deprecated and New in 16.3.x and onwards



Deprecated -

[UNSAFE_componentWillUpdate\(\)](#)

[UNSAFE_componentWillReceiveProps\(\)](#)

New -

[static getDerivedStateFromProps\(\)](#)

[getSnapshotBeforeUpdate\(\)](#)



Unmounting



This method is called when a component is being removed from the DOM:

`componentWillUnmount`

Error Handling



[static getDerivedStateFromError\(\)](#)

[componentDidCatch\(\)](#)



defaultProps(es6)

```
//es6
class Greeting extends React.Component {
  render() {
  }
}

Greeting.defaultProps = {
  name: 'Mary'
};

Greeting.propTypes = {
  name: React.PropTypes.string
},
```


defaultProps(es6)

```
//es6
class Greeting extends React.Component {
  static defaultProps = {
    name: 'Mary'
  };
  static propTypes = {
    name: React.PropTypes.string
  };
  render() {
  }
}
```



getInitialState(es6)

```
class SayHello extends React.Component({
  constructor(props) {
    super(props);
    this.state = {
      message: 'Hello!',
    };
  }
  render: function() {
    //..
  }
})
```



componentWillMount

- Called just before the `render()` is called.
- We have the initial state and default props by this state.
- Is okay to call `this.setState()` here.

```
componentWillMount() {  
  console.log('Component WILL MOUNT!')  
}
```

render

- Available multiple lifecycle stages.
- Never call `setState` or try to access dom nodes inside `render`.
- `Render` returns a single react dom node which can contain multiple dom nodes.

```
render() {  
    return <div> rendering component </div>  
;  
}
```

componentDidMount

- Called after the render method.
- Dom is available to make some changes.

```
componentDidMount() {  
    console.log('Component DID MOUNT!')  
}
```



componentWillReceiveProps

- Gets called when new props are passed to the component.

```
componentWillReceiveProps(newProps) {  
  this.setState({ name: newprops.name })  
}
```



shouldComponentUpdate

- Gets called when props or state changes
- Will stop updation of component if returns false.

```
shouldComponentUpdate(newProps, newState) {  
    return true  
}
```



componentWillUpdate

- Same as `componentWillMount`, just gets called at time state or props are updated.

```
componentWillUpdate(nextProps, nextState) {  
  console.log('Component WILL UPDATE!');  
}
```



componentDidUpdate

- Same as `componentDidMount`, just gets called at time state or props are updated.

```
componentDidUpdate(prevProps, prevState) {  
  console.log('Component DID UPDATE!')  
}
```



componentWillUnmount

- Gets called when component unmounts.
- Is best time when you want to clear any type of listener in your component.

```
componentWillUnmount() {  
  console.log('Component WILL UNMOUNT!')  
}
```



Forms



Forms in React are different than plain HTML forms. A component has its data in its state and/or props received from parent. So we need to design our Form component that reads data from its state and component should be aware of the events (i.e. focus, unfocus, clicks, keypress etc) and update its state accordingly.

Based on how we design our Form we can have two different type of forms

Uncontrolled and **Controlled** forms.



Uncontrolled Forms



Uncontrolled forms are similar to the tradition HTML.

```
class Form extends Component {  
  render( ) {  
    return (  
      <form>  
        <input type="text" />  
      </form>  
    )  
  }  
}
```

Controlled Forms



A controlled input accepts its current value as a prop, as well as a callback to change that value. You could say it's a more “React way” of approaching this.

```
<input value={someValue} onChange={handleChange} />
```

the value of this input (**someValue**) has to live in the state somewhere. Typically, the component that renders the input (Form component) saves that in its state

Controlled Forms cond..



```
class Form extends Component {  
  constructor() {  
    super()  
    This.state = { name: 'John' }  
  }  
  render () {  
    return (  
      <input type="text" value={this.state.name} />  
    )  
  }  
}
```

Handling Form Events



We can handle form events by providing a handler callback to our controlled inputs. In below example we are checking what user is typing and then converting it into upper case.

```
class Form extends Component {
  constructor() {
    super()
    This.state = { name: '' }
    this.handleChange = this.handleChange.bind(this)
  }
  handleChange(event) {
    this.setState({ name: event.target.value.toUpperCase() })
  }
  render () {
    return (
      <input type="text" value={this.state.name} onChange={this.handleChange} />
    )
  }
}
```


Form validation and error handling



In below example, we're going to create a simple signup form and onSubmit we're going to validate if user has entered 'name', password and confirm password should be same. .

```
import React, { Component } from 'react'

export default class Form extends Component {
  constructor() {
    super()
    this.state = {
      name: '',
      pass: '',
      cnf_pass: '',
      message: ''
    }
  }
}
```

example



```
handleChange(e) {  
  let state = {}  
  state[e.target.name] = e.target.value  
  this.setState(state)  
}  
  
handleSubmit (event) {  
  event.preventDefault();  
  if(!this.state.name || !this.state.pass || !this.state.cnf_pass){  
    this.setState({message: 'All fields are required'})  
  }  
  else if(this.state.pass !== this.state.cnf_pass){  
    this.setState({message: 'passwords do not match!'})  
  }  
  else {  
    this.setState({message: "You're registered"})  
  }  
}
```

example



```
render () {  
  return (  
    <form onSubmit={this.handleSubmit.bind(this)}>  
      <p>  
        { this.state.message }  
      </p>  
      <input type='text' name='name' onChange={this.handleChange.bind(this)}  
value={this.state.name} placeholder="name"/>  
      <input type='password' name='pass' onChange={this.handleChange.bind(this)}  
value={this.state.pass} placeholder="password"/>  
      <input type='password' name='cnf_pass' onChange={this.handleChange.bind(this)}  
value={this.state.cnf_pass} placeholder="confirm your password"/>  
      <input type="submit" value="Signup"/>  
    </form>  
  )  
}
```

Keys



React **keys** are useful when working with dynamically created components or when your lists are altered by users. Setting the **key** value will keep your components uniquely identified after the change.

Let's create a list of users dynamically

```
class Keys extends Component {
  constructor(props) {
    super(props);
    this.state = {
      users: ['Chandler', 'Monica', 'Ross', 'Pheobe', 'Rachel']
    }
  }
  render() {
    let userList = this.state.users.map((user,i) => {
      return <li key={i}> {user} </li>
    })
    return (
      <div>
        {userList}
      </div>
    );
  }
}
```

Keys contd..



If we add or remove some elements in the future or change the order of the dynamically created elements, React will use the key values to keep track of each element.



Exercise

Create a cart application having a form component to add new items in cart, a itemlist component to display added items and CartTotal to show total amount.

My Cart

Enter item and price separated by a - (hyphen)

mango	2	30	-	+	⊗
Orange	1	35	-	+	⊗
Apple	4	50	-	+	⊗

Total

295