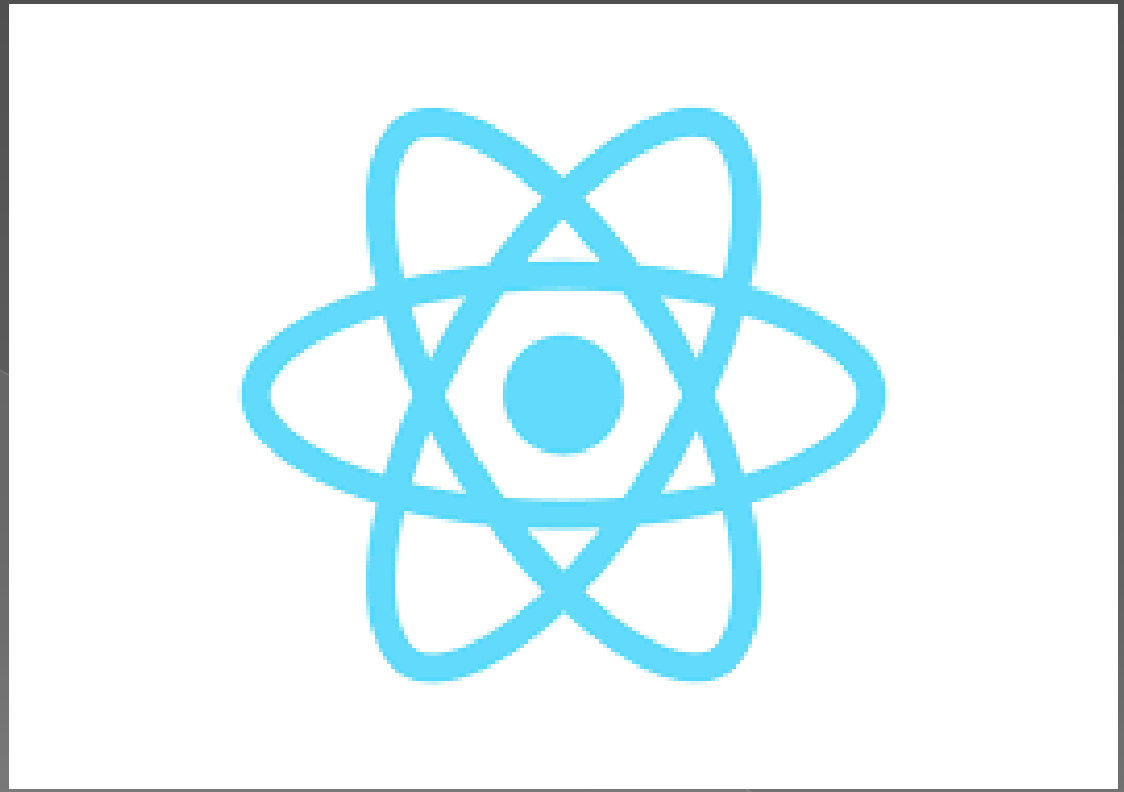
# Web Programming Technologies

# ReactJS

Harshita Maheshwari

# Agenda for Today's Session

- Introduction to ReactJS
- Why ReactJS
- How to install
- Features
- Component – Functional and Class
- Props
- State
- Component LifeCycle Methods
- Event Handling
- Styling with CSS Basic
- Conditional Rendering
- List Rendering
- Fragment
- Form Handling
- Refs
- Error Boundary
- Context

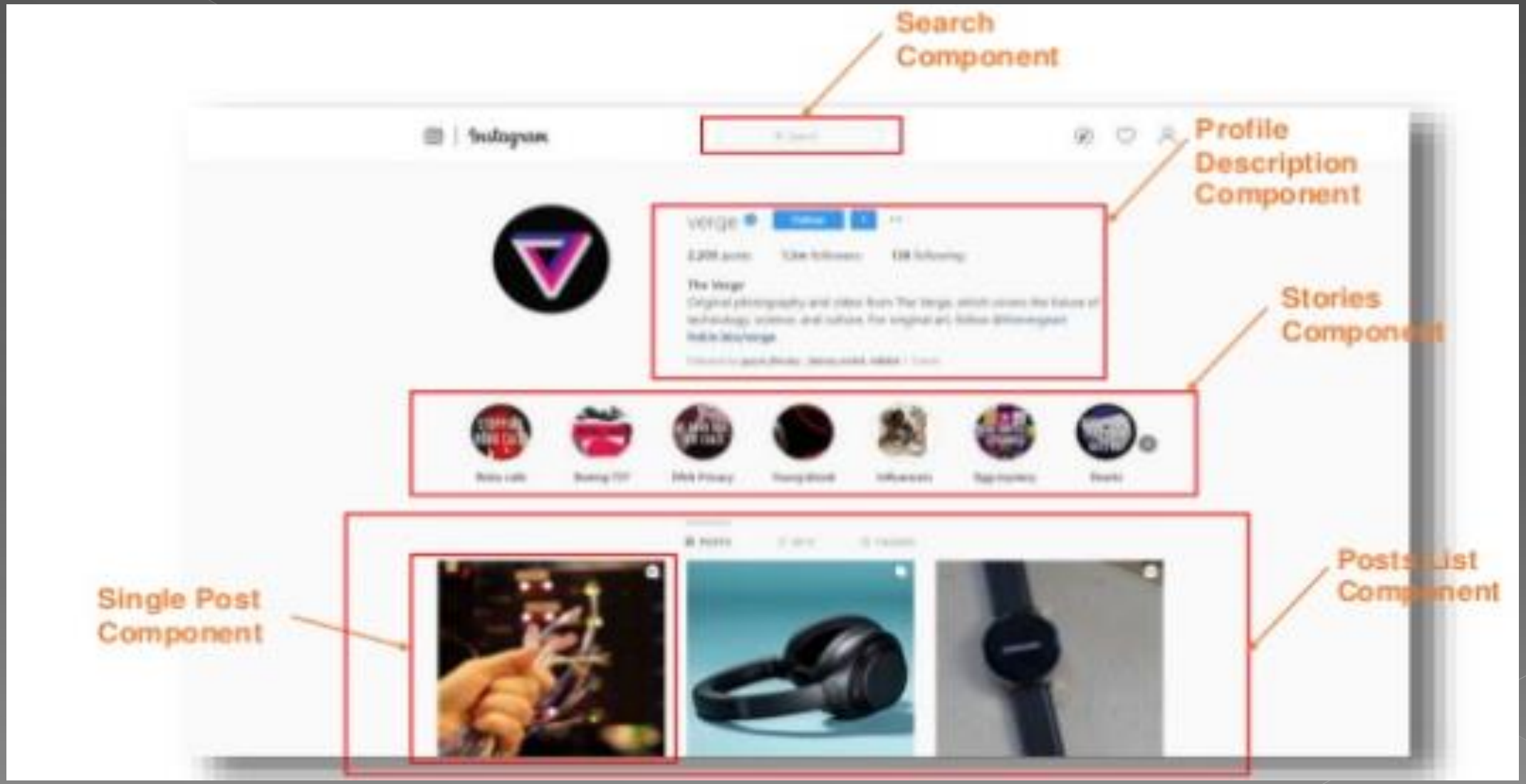


# Introduction

React is a JavaScript library for building fast and interactive user interface for the web as well as mobile applications.

- Open-source
- Reusable component-based front-end library
- In a model view controller architecture, React is the “View” which is responsible for how the app looks and feels.
- Rich ecosystem
- Not  a framework
- Focus on UI
- Created by Jordan Walke, who was a software engineer at Facebook

# Let's see how React works in real time



# Why ReactJS??

- Easy creation of Dynamic web applications
- Performance enhancements
- Reusable components
- Declarative – Tell React what you ant and React will build the actual UI
- Unidirectional Data Flow
- Seamlessly Integrate react into any of your Applications.
- Portion of your webpage or a complete page or even an entire application itself.
- React Native for mobile Applications.

# Prerequisites

- HTML, CSS and JavaScript Fundamentals
- ES6 Features
- JavaScript – “this keyword”, filter, map and reducer
- ES6 – let & const, arrow function, template literals, default parameter, destructing assignment, rest and spread operator.

## Software Requirements:-

1.Node and NPM should be installed in your System

2.Text Editor – (Notepad, Notepad++ or any IDE(visual studio code etc)

# How to Install

### npx

*npx create-react-app <project_name>*

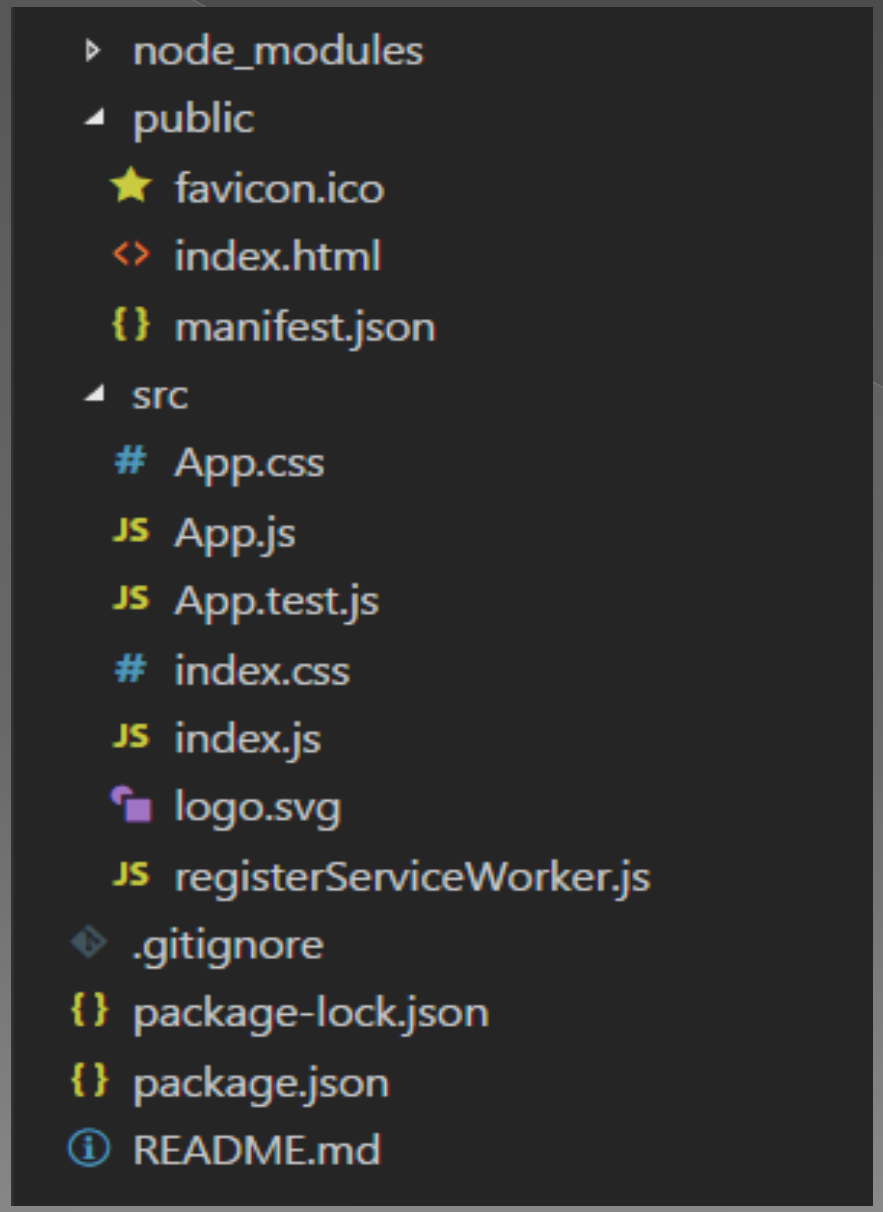
npm package runner

### npm

*npm install create-react-app -g*

*create-react-app<project_name>*

# Folder Structure




src/ - Contains all of our react codebase.

index.js - Base react component.

Package.json – contains dependencies and scripts required for project.

Node_modules – contains all dependencies Files which you have installed.

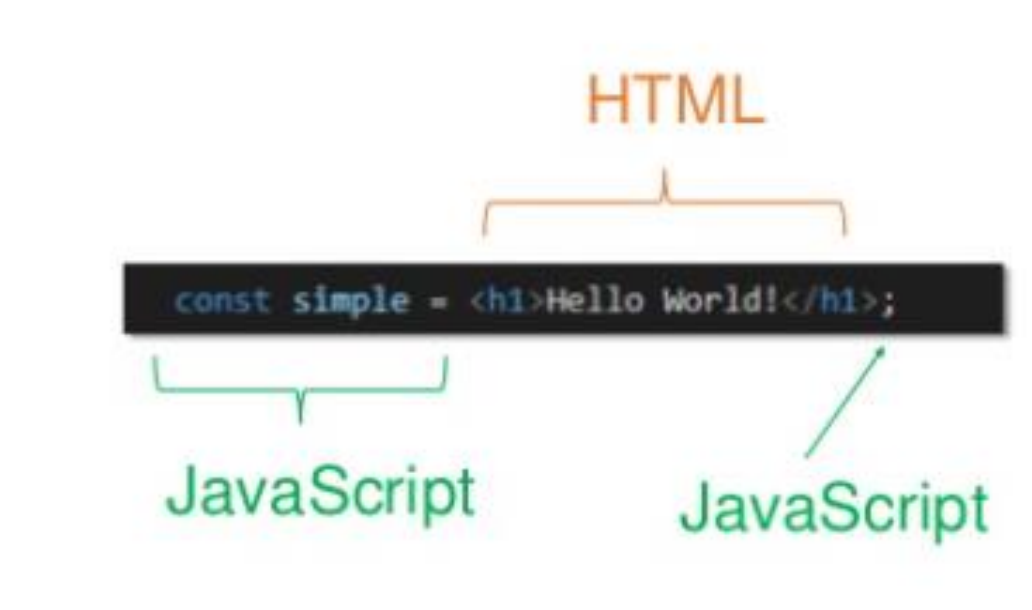
Index.html- Only HTML file as we are creating SPA(single Page Application)

# Features

- JSX
- Virtual DOM
- Performance
- One Way Data Binding
- Extensions
- Debugging

# JSX

- JSX is a syntax extension to JavaScript. It is used with React to describe what the UI should look like.
- By using JSX, we can write HTML structure in the same file that contains JavaScript Code.
- JSX helps in making the code easier to understand and debug as it avoids usage of JS DOM structure which are rather complex.

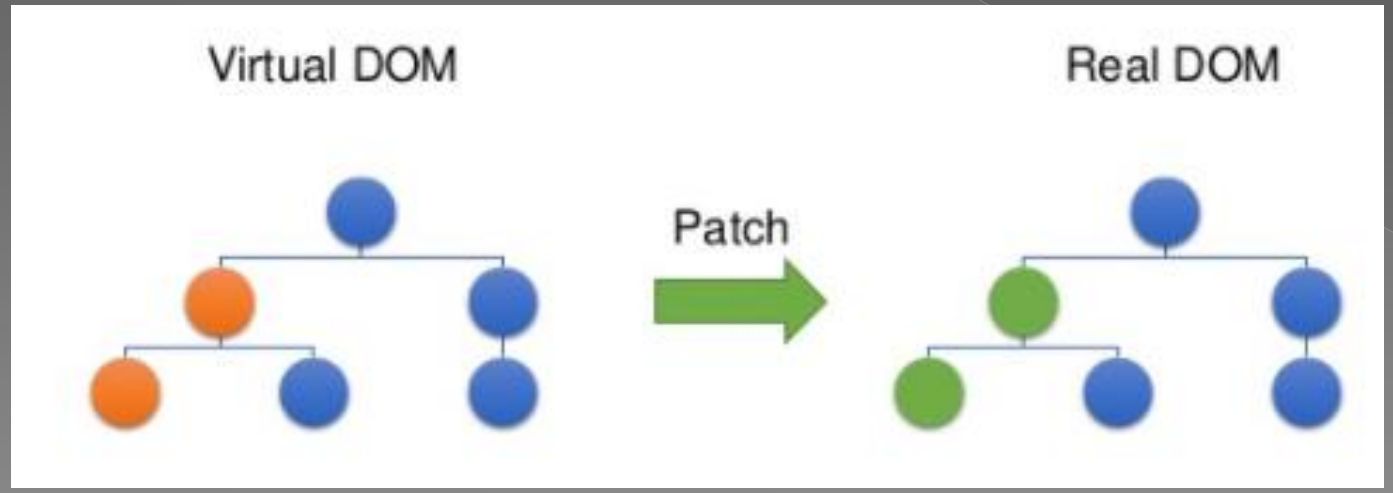



# Virtual DOM

React keeps a lightweight representation of he Real DOM in the memory, and that is known as the Virtual DOM

Manipulating Real DOM is much slower than manipulating virtual DOM, because nothing gets drawn onscreen.

When the state of an abject changes, virtual DOM changes only that object in the Real DOM instead of updating all the objects.



# Performance


React uses Virtual DOM that makes the web apps fast.

Complex User Interface is broken down into individual components allowing multiple users to work on each component simultaneously.

# One-Way Data Binding


React's one way data binding keeps everything modular and fast.

A unidirectional data flow means that when designing a React app you often nest child components within parent components.

# Extension

- React goes beyond simple UI and has many extensions for complete application architecture support.
- It provides server-side rendering.
- Supports mobile app development.
- Extended with flux and Redux ,among others.

# Debugging

- React applications are extremely easy to test due to a large developer community.

- Facebook even provides a small browser extension that makes React debugging faster and easier

# Component

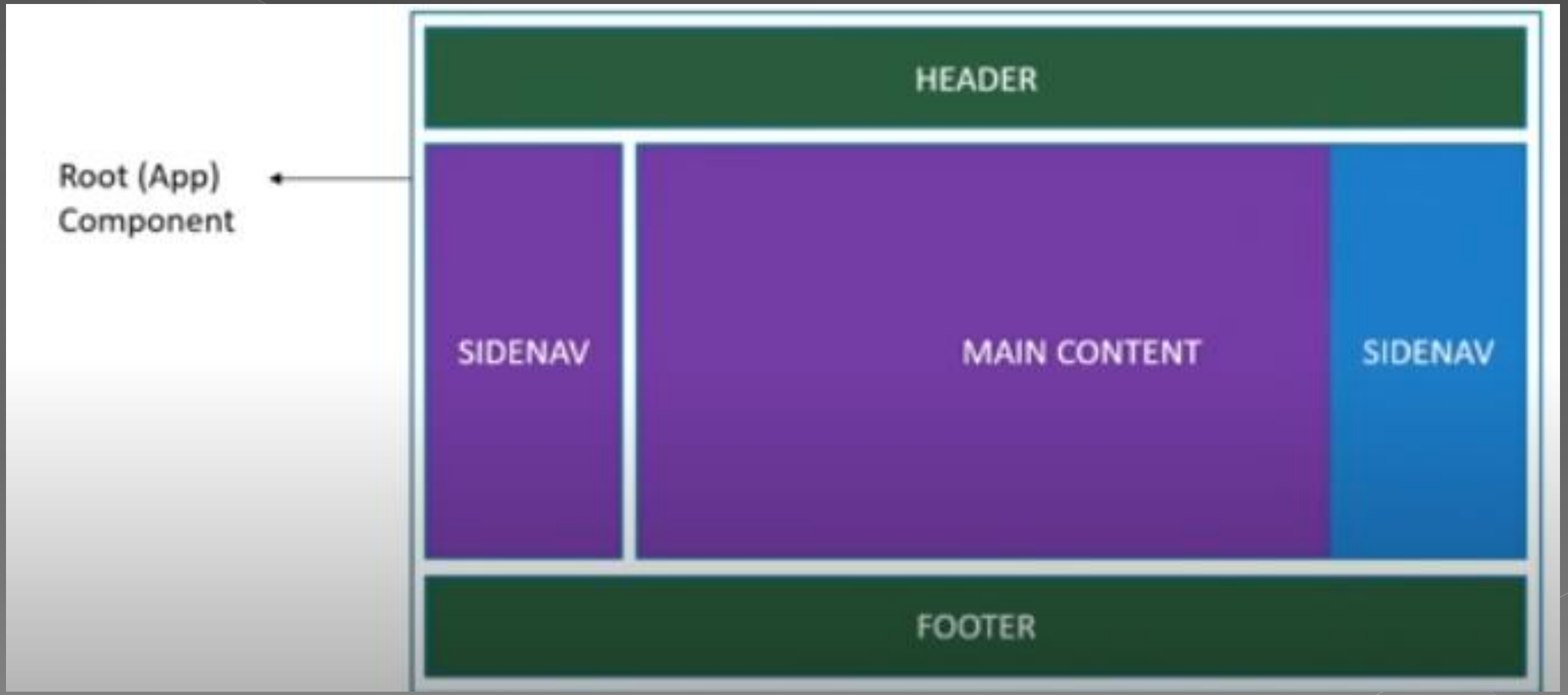
# What is React Component?

- Components are the building blocks of any React app.
- It allow us to split the UI into independent and reusable pieces.

A component is combination of

1. Template using HTML
2. User Interactivity using JS
3. Applying Styles using CSS

Note: Always start component names with a capital letter. React treats components starting with lowercase letters as DOM tags.


Root (App)
Component
HEADER
SIDENAV
MAIN CONTENT
SIDENAV
FOOTER

# Component Types

infoway TECHNOLOGIES

## Functional Component

JavaScript Functions

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

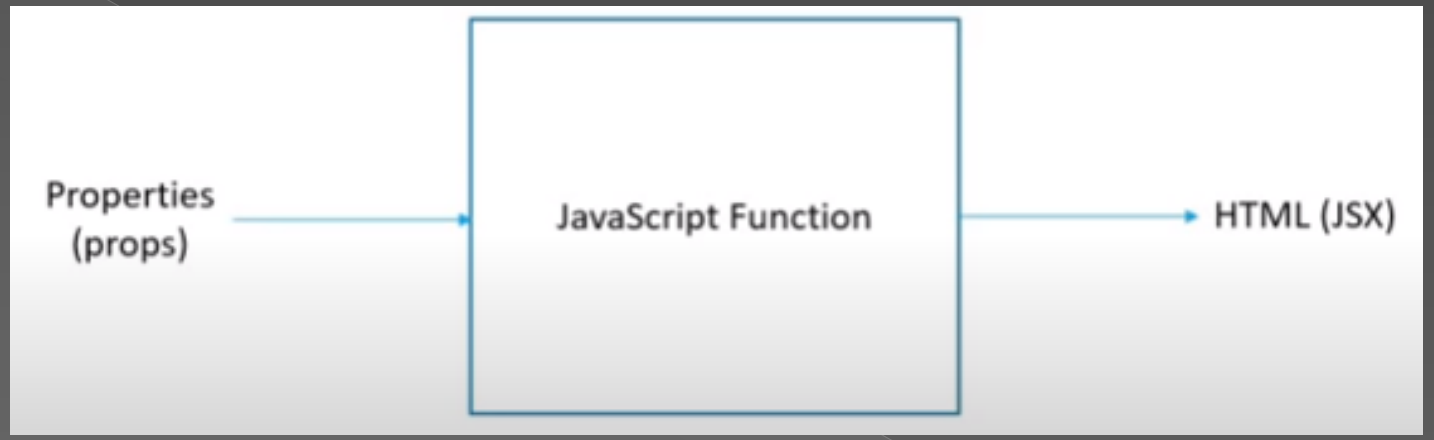
## Class Component

Class extending Component class

Render method returning HTML

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

# Functional Component




```
Employee.js- const Employee=(data)=> {
  return <div><p>Name : {data.name}</p>
    <p>Salary : {data.salary}</p></div>;
}

In App.js  -
<Employee name="Smith" Salary="20000" />
```

# Class Component


```
class Employee extends Component
{
render(){
return <div> <h2>Employee Details...</h2>
<p> <label>Name : <b>{this.props.Name}</b></label> </p>
 <Department Name={this.props.DeptName}/> </div>;
} }
```

# Functional vs. Class Component

| Functional | Class |
| --- | --- |
| Simple Function | More feature rich |
| Use Func components as much as possible | Maintain their own private data-state |
| Absence of ‘this’ keyword | Complex UI logic |
| Solution without using state | Provide LifeCycle Hooks |

# Props

- Props is short for properties, that allow us to pass argument or data to components.
- Props are passed to components in the way similar to the HTML tag attributes.

- They are used to –
  - Pass custom data to your component
  - Trigger state changes

**For Example-**

```
Employee.js- const Employee=(props)=> {
  return <div><p>Name : {props.name}</p>
    <p>Salary : {props.salary}</p></div>;
}

In App.js  -
<Employee
name="Smith" Salary="20000" />
```

# State

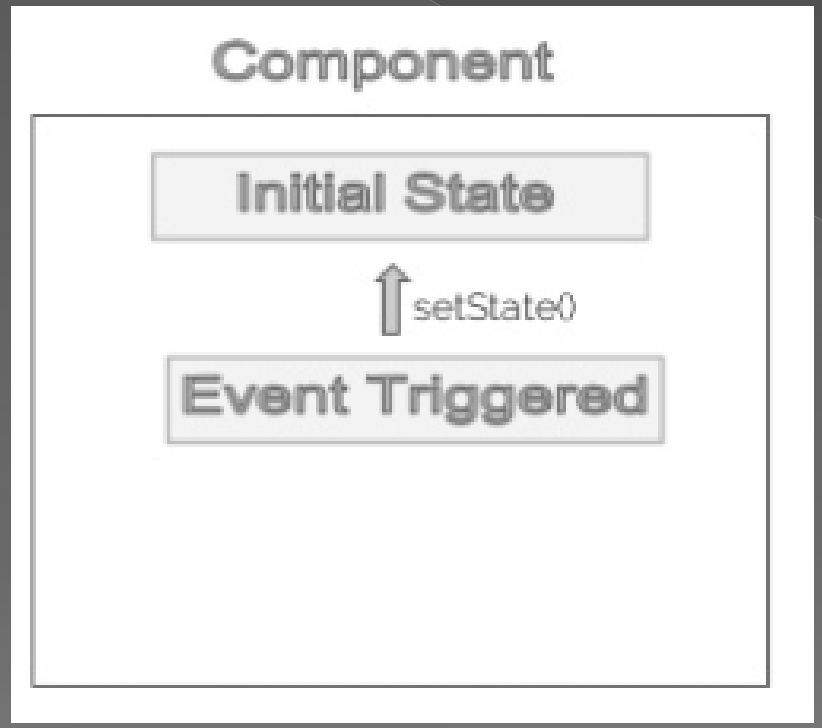
- State of a component is an object that holds some data that may change throughout the component lifecycle..
- We define initial state and then we just have to notify that the state is changed and the react will automatically Render those changes on the Front end behind the scenes.

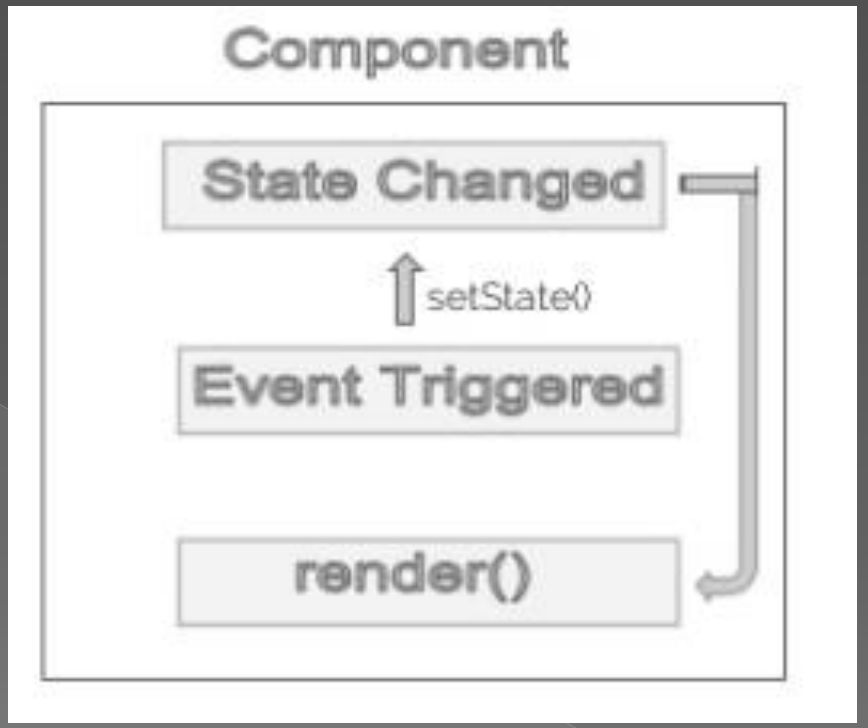
```
import React,{ Component } from 'react';
class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
        Id:101,
        Name:"Ram"
    }
  }
  render(){
  return (
    <div className="App">
      <h2>{this.state.Id}</h2>
      <h2>{this.state.Name}</h2>
    </div>
  );
  }}
export default App;
```

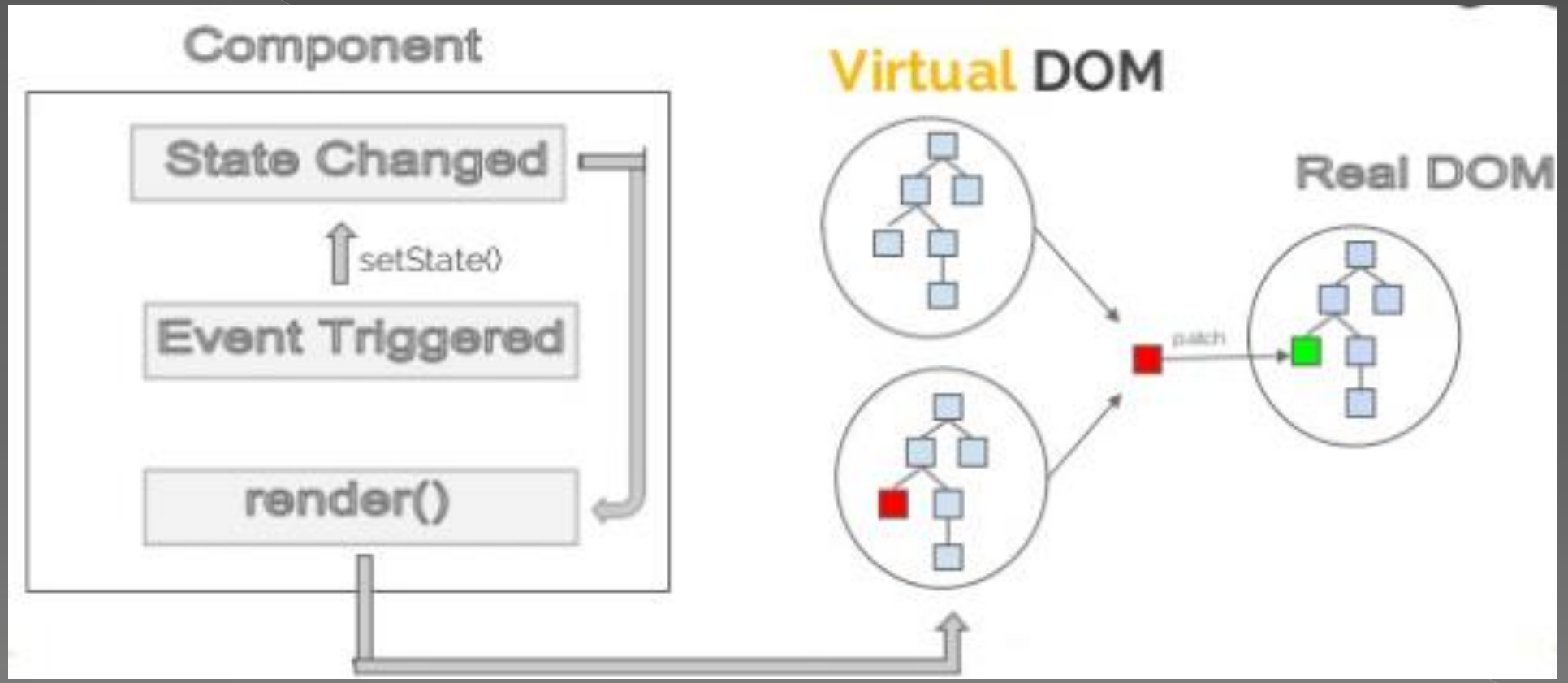
# Props vs. state

| props | state |
| --- | --- |
| Props get passed to the component | State is managed within the component |
| Functional parameters | Variable declared in the function body |
| Props are immutable | States can be changed |
| props – functional components<br>this.props – class components | useState Hook – Functional Components<br>This.state – Class Component |

# setState Function




Component
State Changed
setState()
Event Triggered
render()
Virtual DOM
Real DOM
patch

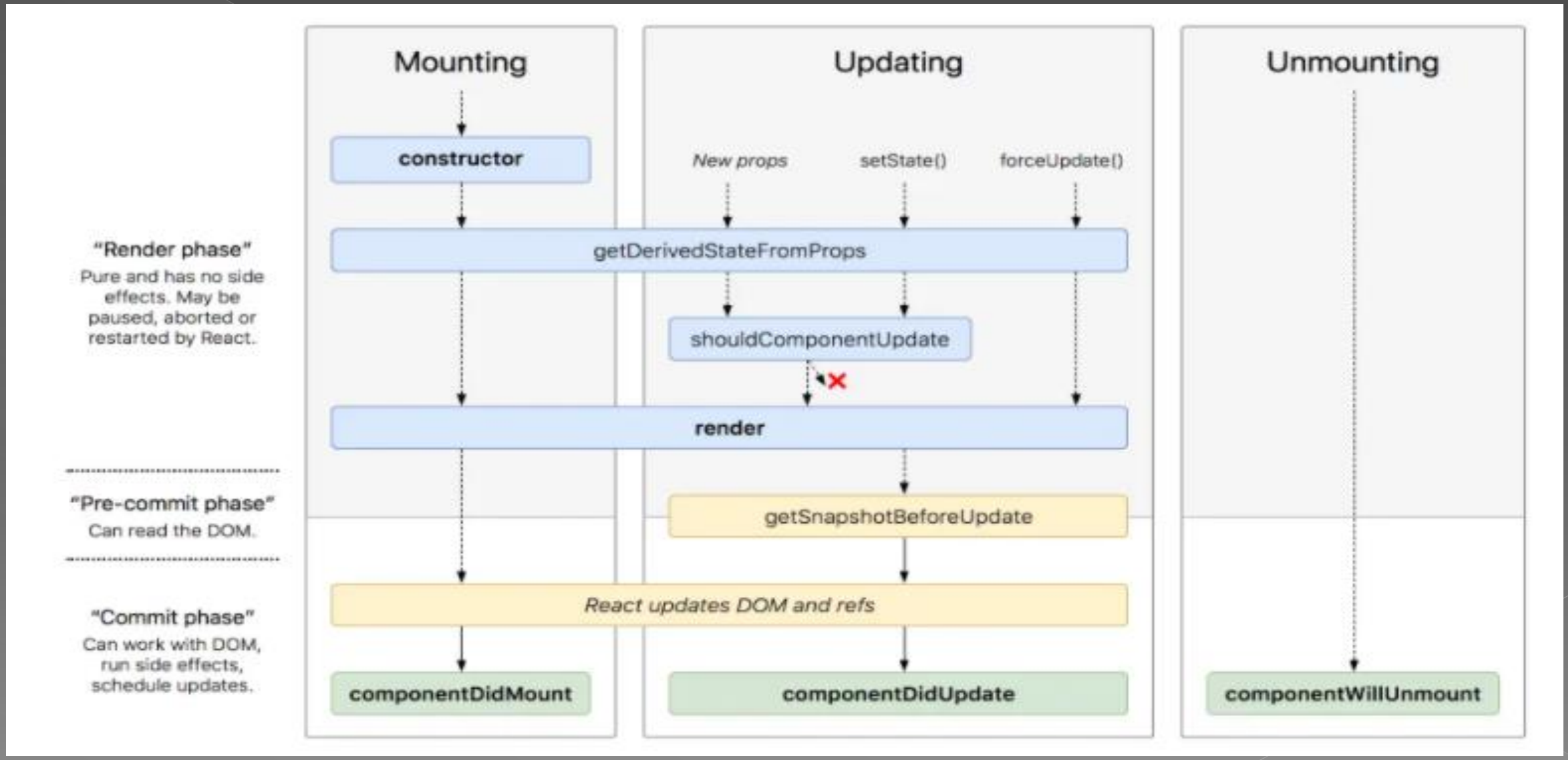
# Component Life Cycle Methods

# Life Cycle Phases

| | |
|---|---|
| Mounting | When an instance of a component is being created and inserted into the DOM |
| Updating | When a component is being re-rendered as a result of changes to either its props or state |
| Unmounting | When a component is being removed from the DOM |
| Error Handling | When there is an error during rendering, in a lifecycle method, or in the constructor of any child component |

# Life Cycle Methods

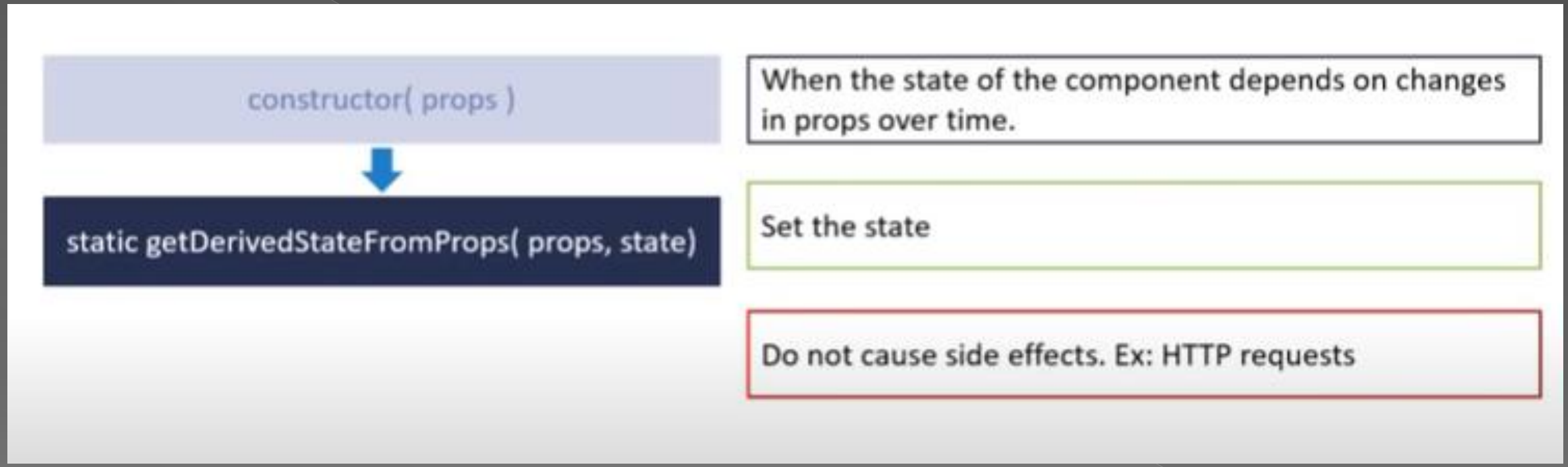
| | |
|---|---|
| Mounting | *constructor, static getDerivedStateFromProps, render* and *componentDidMount* |
| Updating | *static getDerivedStateFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate* and *componentDidUpdate* |
| Unmounting | *componentWillUnmount* |
| Error Handling | *static getDerivedStateFromError* and *componentDidCatch* |


Mounting
Updating
Unmounting
constructor
New props
setState()
forceUpdate()
getDerivedStateFromProps
shouldComponentUpdate
render
getSnapshotBeforeUpdate
React updates DOM and refs
componentDidMount
componentDidUpdate
componentWillUnmount
"Render phase"
Pure and has no side effects. May be paused, aborted or restarted by React.
"Pre-commit phase"
Can read the DOM.
"Commit phase"
Can work with DOM, run side effects, schedule updates.

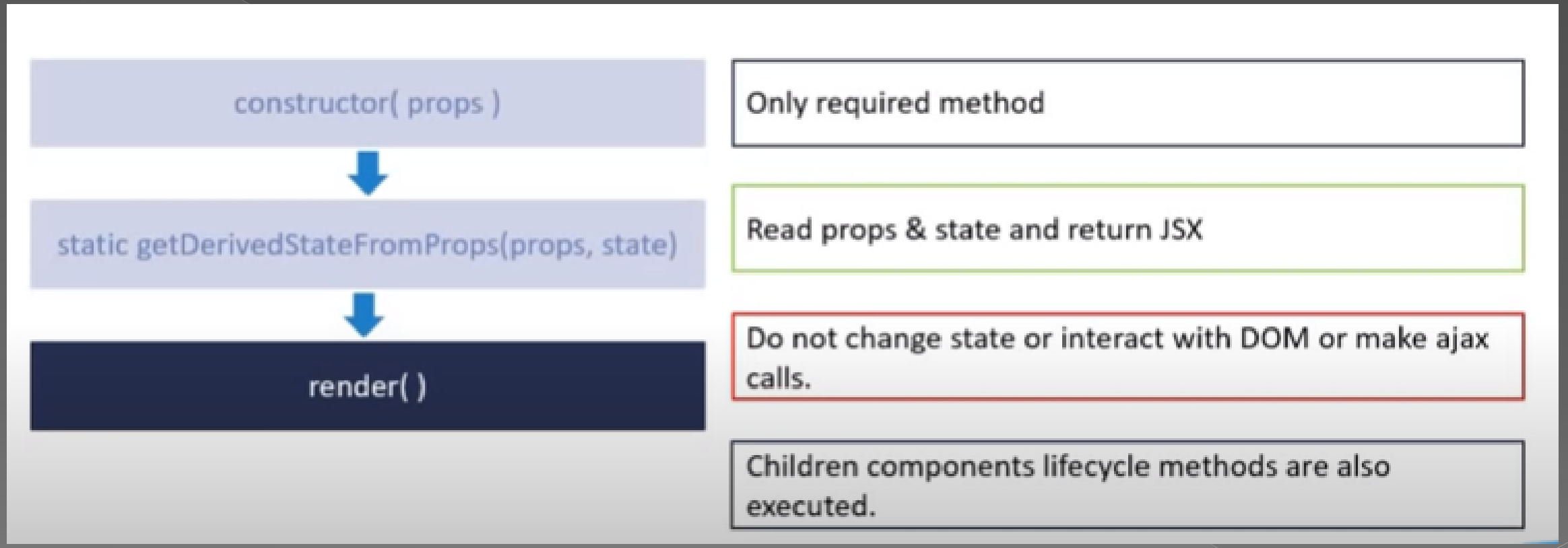
# Mounting Life Cycle Phases

| | |
|---|---|
| constructor( props ) | A special function that will get called whenever a new component is created. |
| | Initializing state<br>Binding the event handlers |
| | Do not cause side effects. Ex: HTTP requests |
| | super(props)<br>Directly overwrite this.state |

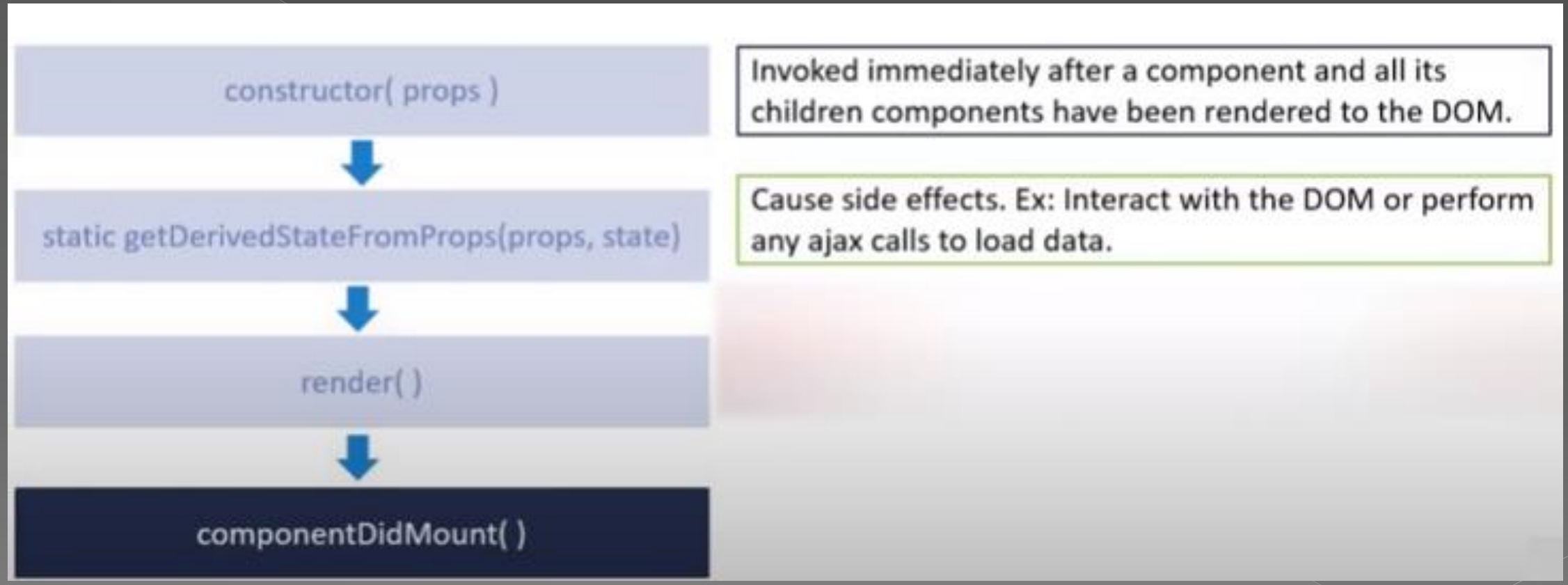
# Mounting Life Cycle Phases



# Mounting Life Cycle Phases



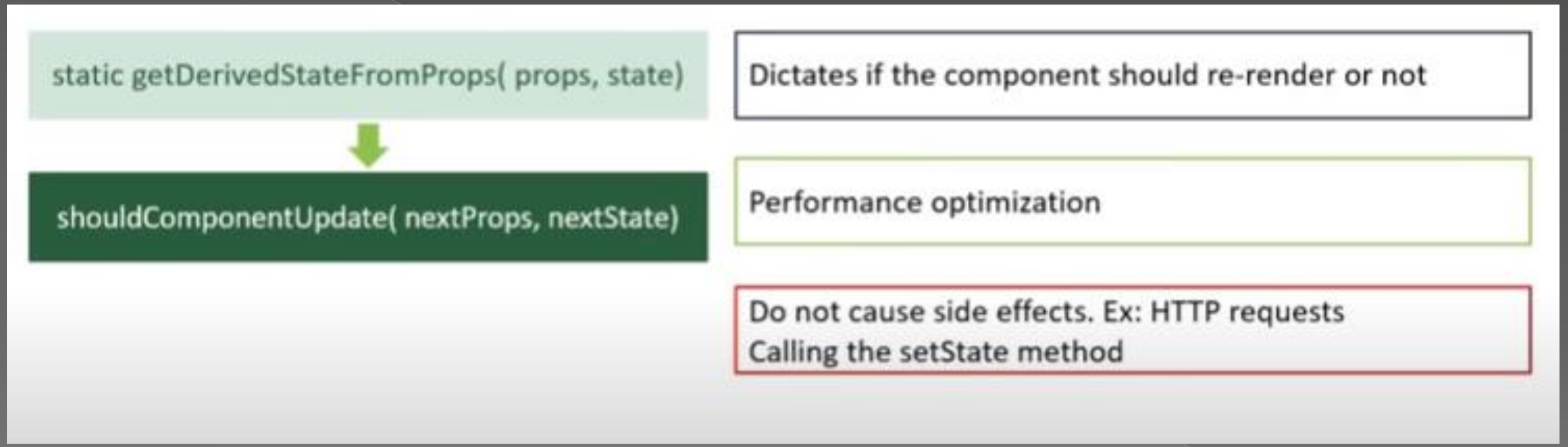
# Mounting Life Cycle Phases



# Updating Life Cycle Phases

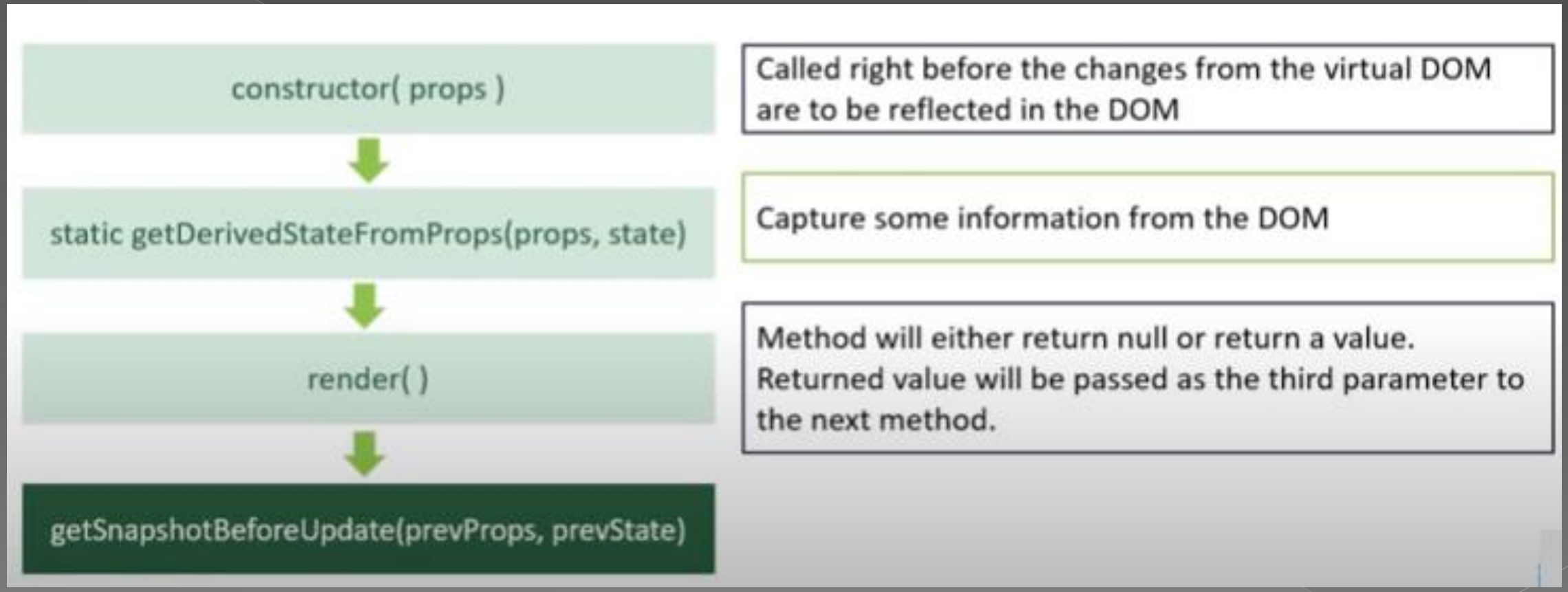
| | |
|---|---|
| static getDerivedStateFromProps( props, state) | Method is called every time a component is re-rendered |
| | Set the state |
| | Do not cause side effects. Ex: HTTP requests |

# Updating Life Cycle Phases

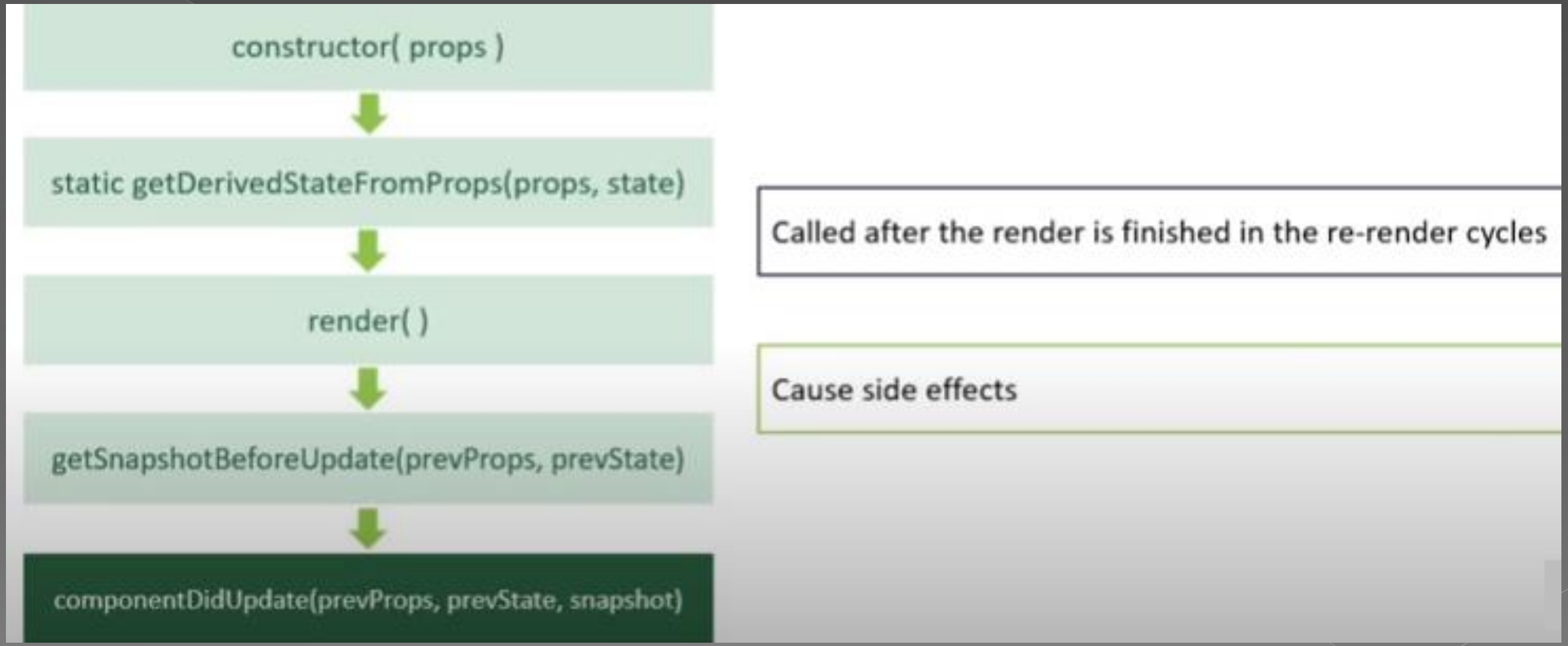


# Updating Life Cycle Phases

# Updating Life Cycle Phases



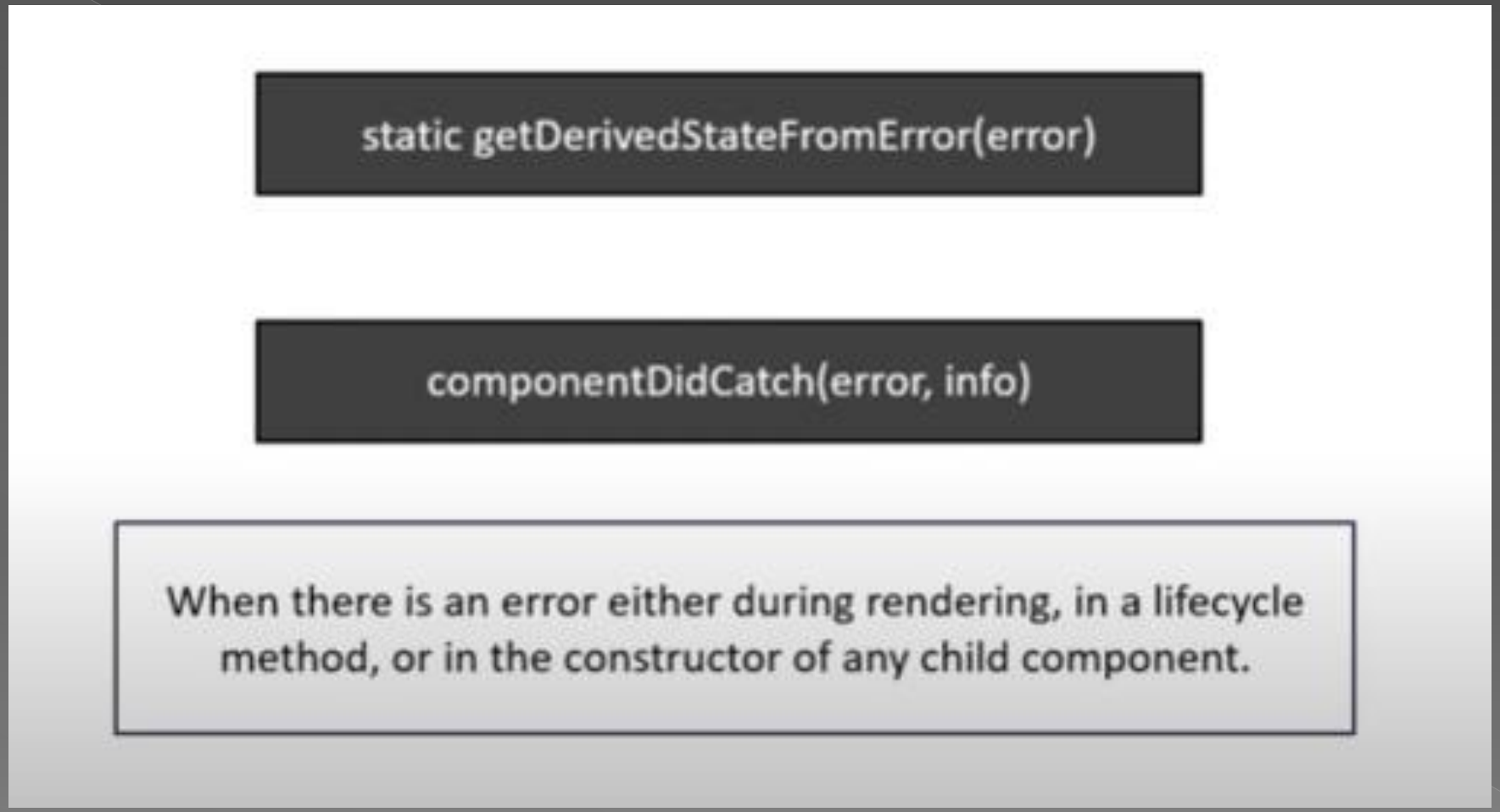
# Updating Life Cycle Phases



# Unmounting Life Cycle Phases

| | |
|---|---|
| componentWillUnmount( ) | Method is invoked immediately before a component is unmounted and destroyed. |
| | Cancelling any network requests, removing event handlers, cancelling any subscriptions and also invalidating timers. |
| | Do not call the setState method. |

# Error Handling Life Cycle Phases



# Event Handling

- Just like HTML, React can perform actions based on user events.
- React has the same events as HTML: click, change, mouseover etc.
- React events are written in camelCase syntax.
- React event handlers are written inside curly braces.
- For methods in React, the this keyword should represent the component that owns the method.
- That is why you should use arrow functions. With arrow functions, this will always represent the object that defined the arrow function.

```
class App extends Component {
clickHandler = () => { alert("Button Clicked"); }
render() {
return (
<button onClick={this.clickHandler}>Click Me</button>
        );
    }
 }

export default App
```

# Event Handling

infoway TECHNOLOGIES

## Passing Arguments : -

- **Make an anonymous arrow function**

```
<button onClick={() => this.clickHandler("Hello")}>
Click Me!!</button>
```

- **Bind the event handler to this**

```
<button onClick={this.clickHandler.bind(this, "Hello")}>Click
   Me!!</button>
```

# Styling with CSS Basics

- CSS Stylesheets
- Inline Styling
- CSS Modules
- CSS in JS Libraries (Styled Component)

# Inline Stylesheet

```
<h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>
```

```
render() {
  const mystyle = {
    color: "white",
    backgroundColor: "DodgerBlue",
    padding: "10px",
    fontFamily: "Arial"
  };
  return (
    <div>
    <h1 style={mystyle}>Hello Style!</h1>
    <p>Add a little style!</p>
    </div>
  );
}
```

# CSS Modules

The CSS inside a module is available only for the component that imported it.

Create the CSS module with the .module.css.

For Example : -

**mystyle.module.css**

```
.bigblue {
  color: DodgerBlue;
  padding: 40px;
  font-family: Arial;
  text-align: center;
}
```

**App.js**

```
import styles from './mystyle.module.css';
```

```
render() {
  return <h1 className={styles.bigblue}>Hello Car!</h1>;
}
```

# Conditional Rendering

Conditional rendering is a term to describe the ability to render different user interface (UI) markup if a condition is true or false.  In React, it allows us to render different elements or components based on a condition. This concept is applied often in the following scenarios:

- Rendering external data from an API.
- Showing or hiding elements.
- Toggling application functionality.
- Implementing permission levels.
- Handling authentication and authorization.

## Conditional Rendering Approaches:

1. If/else
2. Element Variables
3. Ternary conditional operator
4. Short Circuit Operator

# If/else Approaches

```
class App extends Component {
  // ...
  render() {
    let {isLoggedIn} = this.state;
    if (isLoggedIn) {
      return (
        <div className="App">
          <button>Logout</button>
        </div>
      );
    } else {
      return (
        <div className="App">
           <button>Login</button>
        </div>
      );
    }
  }
}
```

# Element Variables

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }
  render() {
    let { isLoggedIn } = this.state;
    let AuthButton;
    if (isLoggedIn) {
      AuthButton = <button>Logout</button>;
    } else {
      AuthButton = <button>Login</button>;
    }
    return (
      <div className="App">
        {AuthButton}
      </div>
    );
  }
}
export default App;
```

# Using Ternary Operators

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }
  render() {
    let { isLoggedIn } = this.state;
    return (
      <div className="App">
         {isLoggedIn ? <button>Logout</button> : <button>Login</button>}
      </div>
    );
  }
}
export default App;
```

# Using Logical && (Short Circuit Evaluation)

Short circuit evaluation is a technique used to ensure that there are no side effects during the evaluation of operands in an expression. The logical && helps you specify that an action should be taken only on one condition, otherwise, it would be ignored entirely.

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }
  render() {
    let { isLoggedIn } = this.state;
    return (
      <div className="App">
           {isLoggedIn && <button>Logout</button>}
      </div>
    );
  }
}
export default App;
```

# List Rendering

infoway TECHNOLOGIES

## Using array index:

```
const employee = ['zbc','xyz'];
    return(
        <div>
                <h2>{employee[0]}</h2>
                <h2>{employee[1]}</h2>
        </div>
  )
```

## Array.map():

```
const employee = ['abc','xyz'];
return(
    <div> {employee.map(emp => <h2>{emp}</h2>)} </div>
 )
```

# List Rendering

Now we will see example of having an array with key-value pair in each array having multiple values:

```
function EmployeeList(){
       const employee = [{
            name:'abc',
            salary:'50$',
            position:'Jr. Developer'
       },{
            name:'xyz',
            salary:'100$',
            position:'Sr. Developer'
       },{
            name:'mno',
            salary:'150$',
            position:'Project Manager'
       }
     ];
    const employeeList = employee.map(emp => <h2>My name is {emp.name} working as {emp.position}
        and having salary {emp.salary}     </h2>);
       return(
            <div>
                        {employeeList}
            </div>
       )
}
export default EmployeeList;
```

# Render List in Sub-Component

```
import Employees from './Employees';
function EmployeeList(){
        const employee = [{
              name:'abc',
              salary:'50$',
              position:'Jr. Developer'
          },{
              name:'xyz',
              salary:'100$',
              position:'Sr. Developer'
          },{
              name:'mno',
              salary:'150$',
              position:'Project Manager'
          }
       ];
        const employeeList = employee.map(emp =>
              <Employees emp={emp}></Employees>
              );
              return <div>{employeeList}</div>;
}
export default EmployeeList;
```

```
function Employees({emp}){
        return(
        <div>
                <h2>My name is {emp.name} working as
                    {emp.position} and having salary
                    {emp.salary} </h2>
        </div>
        )
}
export default Employees;
```

# List and Key Props

When we fetch list in sub-component then we will find there are errors related to keys in console.

It will show a warning related to keys as each child in a list should have a unique key prop. This error can be resolved by defining the key to each list item generated using JSX. The key defined should not be the same for any list item.

```
  const employeeList = employee.map(emp => <Employees key= {emp.id} emp={emp}
   ></Employees>);
    return <div>{employeeList}</div>;
}
```

The important point that needs to be kept in mind when using key prop is that key prop cannot be used in child component.

Importance of key prop:

- Key prop helps to easily identify which item is added, updated or removed.
- Key prop helps in updating user interface efficiently.
- Keys provide stable identity to elements.

# Fragment

We know that we make use of the render method inside a component whenever we want to render something to the screen. We may render a single element or multiple elements, though rendering multiple elements will require a 'div' tag around the content as the render method will only render a single root node inside it at a time.

when we are trying to render more than one root element we have to put the entire content inside the 'div' tag which is not loved by many developers. So in React 16.2 version, **Fragments** were introduced, and we use them instead of the extraneous 'div' tag.

Syntax:

```
<React.Fragment>
        <h2>Child-1</h2>
        <p> Child-2</p>
</React.Fragment>
```

**Shorthand Fragment :**

```
<>
    <h2>Child-1</h2>
     <p> Child-2</p>
</>
```

# React Form Handling

# Creating Form

infoway TECHNOLOGIES

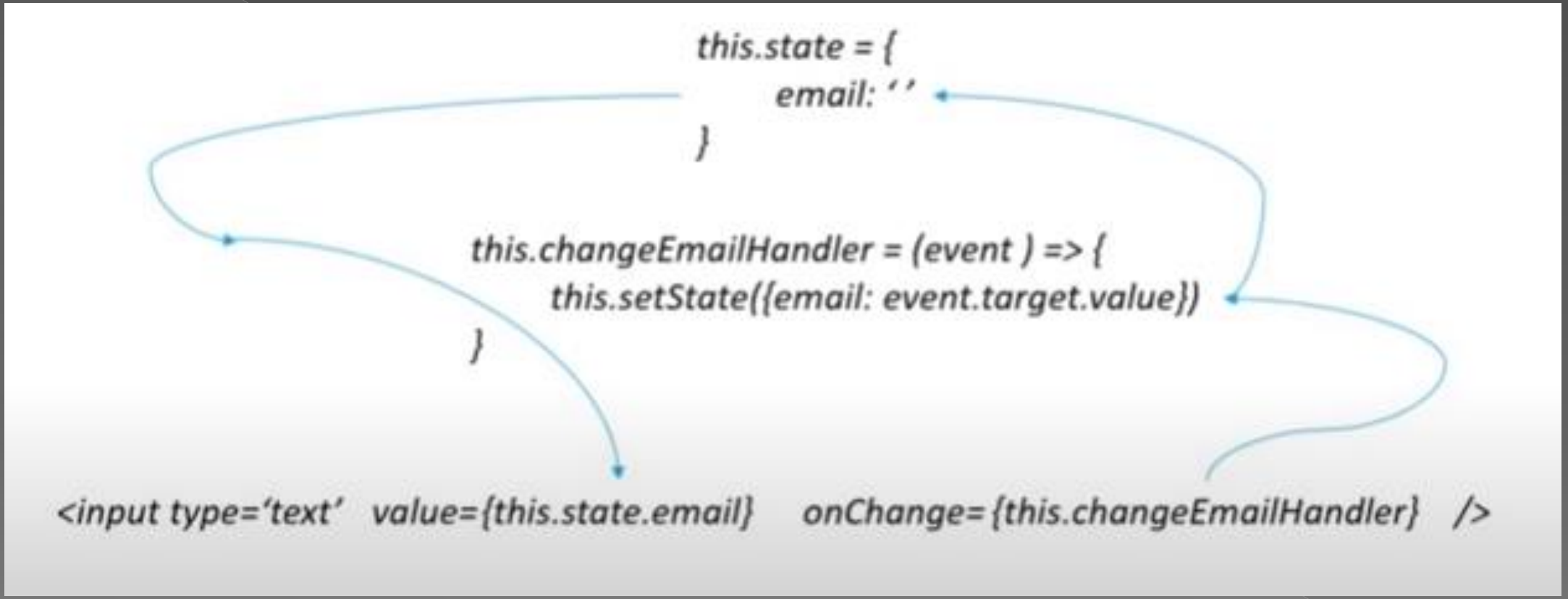
React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

- Uncontrolled component: Where form data is handled by the DOM itself. We will use ref to get the input values and Perform Operations using this data.

- Controlled component: An input form element whose value is controlled by React in this way is called a "controlled input or Controlled Component".

# Controlled Components



```
import React, { Component } from 'react'
class Form extends Component {
        constructor(props) {
                super(props)
                this.state = {
                        username: '',
                }
        }
        handleUsernameChange = event => {
                this.setState({
                        username: event.target.value
                })
        }
        handleSubmit = event => {
                alert(`${this.state.username}`)
                event.preventDefault()
        }
        render() {
                return (
                        <form onSubmit={this.handleSubmit}>
                        <div>
                        <label>Username </label>
                        <input type="text" value={this.state.username} onChange={this.handleUsernameChange}/>
                        </div>
                        <button type="submit">Submit</button>
                        </form>
                )
        }
}
export default Form
```

# Uncontrolled Components

To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

```
import React, { Component } from 'react';
class App extends Component {
  constructor(props) {
      super(props);
      this.updateSubmit = this.updateSubmit.bind(this);
      this.input = React.createRef();
  }
  updateSubmit(event) {
      alert('You have entered the UserName and Age successfully.');
      event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.updateSubmit}>
        <h1>Uncontrolled Form Example</h1>
        Name: <input type="text" ref={this.input} />
        Age:  <input type="text" ref={this.input} />
       <input type="submit" value="Submit" />
      </form>
    );
  }
}
export default App;
```

# Controlled vs uncontrolled component

| Controlled | Uncontrolled |
| --- | --- |
| It does not maintain its internal state. | It maintains its internal states. |
| Here, data is controlled by the parent component. | Here, data is controlled by the DOM itself. |
| It accepts its current value as a prop. | It uses a ref for their current values. |
| It allows validation control. | It does not allow validation control. |
| It has better control over the form elements and data. | It has limited control over the form elements and data. |

# Refs

**Refs is the shorthand used for references in React. It is similar to keys in React. It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements.**

## When to Use Refs:

- Refs can be used in the following cases:
- When we need DOM measurements such as managing focus, text selection, or media playback.
- It is used in triggering imperative animations.
- When integrating with third-party DOM libraries.
- It can also use as in callbacks.

## When to not use Refs:

- Its use should be avoided for anything that can be done declaratively. For example, instead of using open() and close() methods on a Dialog component, you need to pass an isOpen prop to it.
- You should have to avoid overuse of the Refs.

# How to create Refs

Refs can be created by using React.createRef().

**How to access Refs:**

```
const node = this.callRef.current;
```

## Refs current Properties:

- The ref value differs depending on the type of the node:
- When the ref attribute is used in HTML element, the ref created with React.createRef() receives the underlying DOM element as its current property.
- If the ref attribute is used on a custom class component, then ref object receives the mounted instance of the component as its current property.
- The ref attribute cannot be used on function components because they don't have instances.

# Callback refs

it gives more control when the refs are set and unset. Instead of creating refs by createRef() method.

```
<input type="text" ref={element => this.callRefInput = element} />
```

It can be accessed as below:

```
this.callRefInput.value
```

# Forwarding Ref from one component to another component

```
import React, { Component } from 'react';
import { render } from 'react-dom';

const TextInput = React.forwardRef((props, ref) => (
  <input type="text" placeholder="Hello World" ref={ref} />
));

const inputRef = React.createRef();

class CustomTextInput extends Component {
  handleSubmit = e => {
    e.preventDefault();
    console.log(inputRef.current.value);
  };
  render() {
    return (
      <div>
        <form onSubmit={e => this.handleSubmit(e)}>
          <TextInput ref={inputRef} />
          <button>Submit</button>
        </form>
      </div>
    );
  }
}
```

# Error Boundary

A class component that implements either one or both the life cycle methods getDerivedStateFromError or componentDidCatch becomes an error boundary.

The static method getDerivedStateFromError method is used to render a fallback UI after an error is thrown and the componentDidCatch method is used to log the error information.

Error boundaries do not catch errors inside event handlers.

Error boundaries do not catch errors also for:

- Any Asynchronous code we write (e.g. setTimeout )
- For Server side rendering code
- For Errors thrown in the error boundary component class itself (rather than its children)

```
import React, { Component } from 'react'
export class ErrorBoundary extends Component {
        constructor(props) {
                super(props)
                this.state = {
                        hasError: false
                }
        }
        static getDerivedStateFromError(error) {
                return { hasError: true }
        }
        componentDidCatch(error, info) {
                console.log(error)
                console.log(info)
        }
        render() {
                if (this.state.hasError) {
                        return <h1>Something went wrong.</h1>
                }
                return this.props.children
        }
}
export default ErrorBoundary
```

```
import React from 'react'
function Hero ({ heroName }) {
  if (heroName === 'Joker') {
    throw new Error(' Not a hero!')
  }
  return <h1>{heroName}</h1>
}
export default Hero
```

```
class App extends Component {
        render() {
           return (
                <div className="App">
                  <Hero heroName="Batman" />
                  <Hero heroName="Superman" />
                  <ErrorBoundary>
                     <Hero heroName="Joker" />
                  </ErrorBoundary>
                </div>
                )
        }
}
```

# Context

Context allows passing data through the component tree without passing props down manually at every level.

In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

There are three main steps to use the React context into the React application:

- Create context
- Setup a context provider and define the data which you want to store.
- Use a context consumer whenever you need the data from the store

# React Context API

**The React Context API is a component structure, which allows us to share data across all levels of the application. The main aim of Context API is to solve the problem of prop drilling (also called "Threading"). The Context API in React are given below.**

- **React.createContext :**

```
const MyContext = React.createContext(defaultValue);
```

- **Context.provider**

```
<MyContext.Provider value={/* some value */}>
```

- **Context.Consumer**

```
<MyContext.Consumer>
        {value => /* render something which is based on the context value */}
</MyContext.Consumer>
```

- **Class.contextType**

UserContext.js

```
import React from 'react'

const UserContext = React.createContext('default')

const UserProvider = UserContext.Provider
const UserConsumer = UserContext.Consumer

export { UserProvider, UserConsumer }
```

Columns.js

```
import React, { Component } from 'react'
import UserConsumer from './UserContext'
export default class ColumnFragment extends Component {
    render(){
    return (
        <>
           <UserConsumer>
                {(uname)=><div>Hello {uname}</div>}
            </UserConsumer>
        </>
    )
}
}
```

```
import { UserProvider } from './Components/UserContext';
class App extends Component{
  render(){
  return (
    <div>
     <UserProvider value="Infoway">
          <TableFragment/>
      </UserProvider>
    </div>
  );
}
}
export default App;
```

# Class.contextType

UserContext.js

```
import React from 'react'
const UserContext=React.createContext();
const UserProvider=UserContext.Provider
const UserConsumer=UserContext.Consumer
export {UserProvider, UserConsumer}
export default UserContext;
```

```
import React, { Component } from 'react'
import UserContext from './UserContext'
export default class ColumnFragment extends Component {
    static contextType=UserContext;
    render(){
    return (
        <>
          {this.context.Id}{this.context.Name}
        </>
    )
}
}
// ColumnFragment.contextType=UserContext;
```

```
import { UserProvider } from './Components/UserContext';
class App extends Component{
  constructor(props) {
    super(props)
      this.state = {
       data:{Id:101,Name:"Ram"}
    }
  }
  changedata=()=>{
    this.setState({
      data:{Id:102,Name:"Seema"}
    })
  }
  render(){
  return (
    <div>
     <UserProvider value={this.state.data}>
          <TableFragment/>
      </UserProvider>
    <button onClick={this.changedata}>Change Data of Context</button>
    </div>
  );
}
}
export default App;
```

# END

# Thank you !!