REACTJS

A study material for the students of GLS University

Prepared by : Harshita Maheshwari

Subject : MERN Stack

Introduction to ReactJS

ReactJS, an open-source JavaScript library developed by Facebook in 2013.

It is a powerful tool for building fast and scalable user interfaces.

It powers dynamic single-page and multi-page web applications for companies like Facebook, Instagram, and Netflix.

Library vs. Framework: What's the Difference?

1

Library

A collection of reusable code for specific tasks. You, the developer, are in control, calling the library's functions when needed.

- Examples: React (UI), Lodash (utilities)

2

Framework

Provides a comprehensive structure and guides your application's architecture. The framework dictates the flow, calling your code as needed (Inversion of Control).

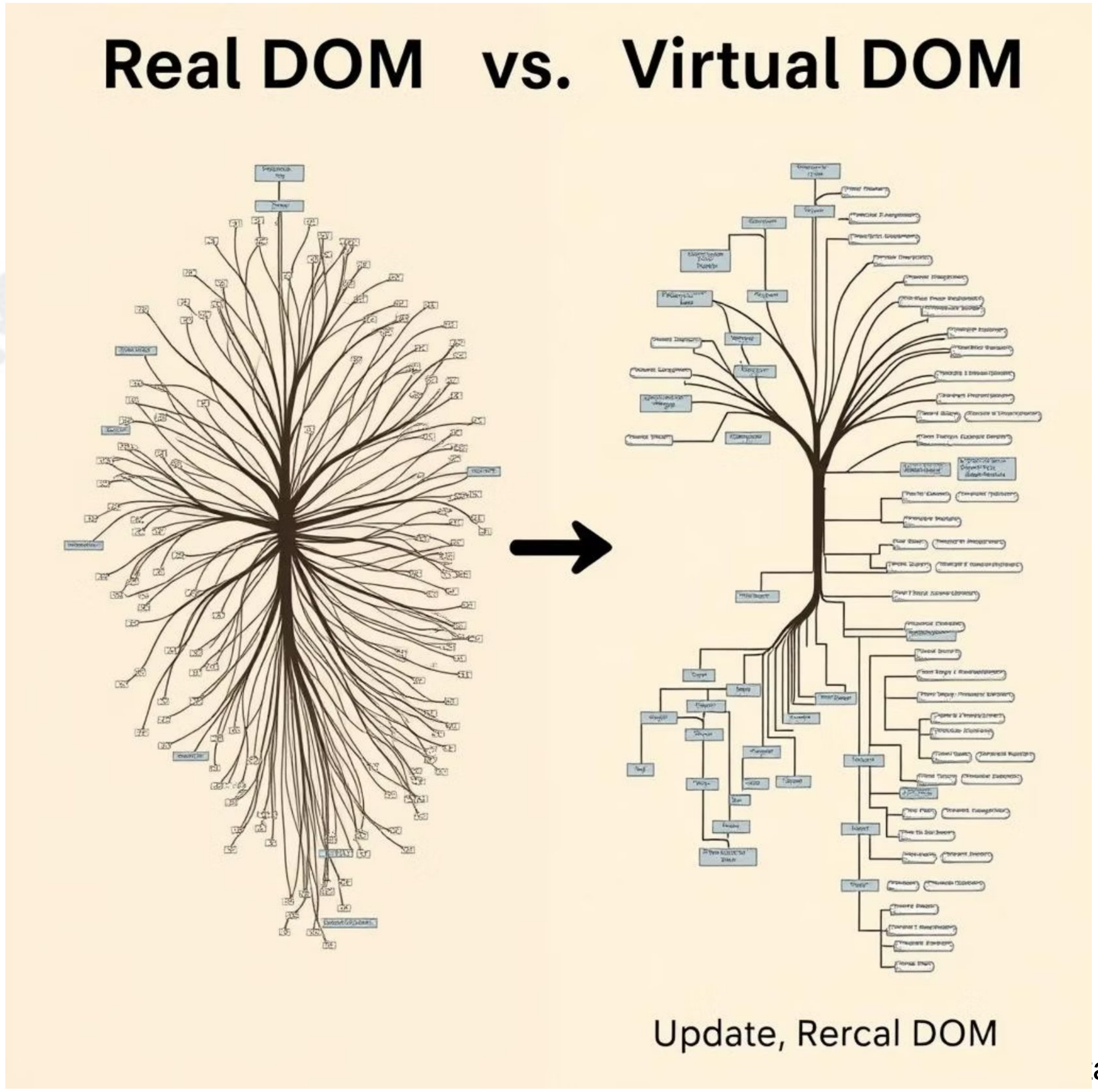
- Examples: Angular, Vue (full-stack)

Think of it this way: with a library, you're building with tools from a toolbox. With a framework, you're building within a pre-designed house blueprint.

Why ReactJS Emerged

In 2011, Facebook faced challenges with its News Feed: slow Document Object Model (DOM) updates and complex UI state made development difficult.

React was created to solve these problems by introducing the **Virtual DOM** for efficient UI rendering. It was open-sourced in 2013 and quickly gained traction, becoming a cornerstone of modern web development.



Key Features of ReactJS

Component-Based Architecture

Build UIs with self-contained, reusable components, making complex UIs manageable.

Virtual DOM

React creates an in-memory representation of the UI, comparing changes and updating only what's necessary for speed.

JSX

An HTML-like syntax embedded in JavaScript, making UI creation more intuitive and readable.

One-Way Data Binding

Data flows predictably from parent to child components, simplifying debugging and maintaining consistency.

Advantages of ReactJS

High Performance
Efficient UI updates thanks to the Virtual DOM.

Easy to Learn
Accessible for those familiar with JavaScript and basic JSX.

Vast Ecosystem
Rich with third-party libraries, tools, and a strong community.

SEO Friendly
Supports server-side rendering (e.g., Next.js) for better SEO.

Strong Community & Updates
Constant evolution and extensive support resources.

Disadvantages of ReactJS

- **Rapid Evolution:** Constant updates mean continuous learning is required to stay current.
- **UI Layer Only:** React focuses solely on the UI; you'll need other tools for full application development (routing, state management).
- **JSX Learning Curve:** The HTML-like JSX syntax might initially confuse newcomers.
- **Optimization Needed:** Without proper optimization, performance can suffer in complex applications.

Prerequisites

HTML, CSS and JavaScript Fundamentals
ES6 Features

Software Requirements:-
1.Node and NPM should be installed in your System
2.Text Editor – (Notepad, Notepad++ or any IDE(visual studio code etc)

Create react project

npx create-react-app appname – Older Way
create-react-app is now considered outdated for modern React development.

The **new and faster alternative is Vite**, which is lightweight and much faster for both development and build processes.

Create a React Project using **Vite** (Modern Way)

1. Make sure Node.js is installed
Check by running:
node -v
npm –v

2. Create the Project using Vite **:npm create vite@latest my-react-app**

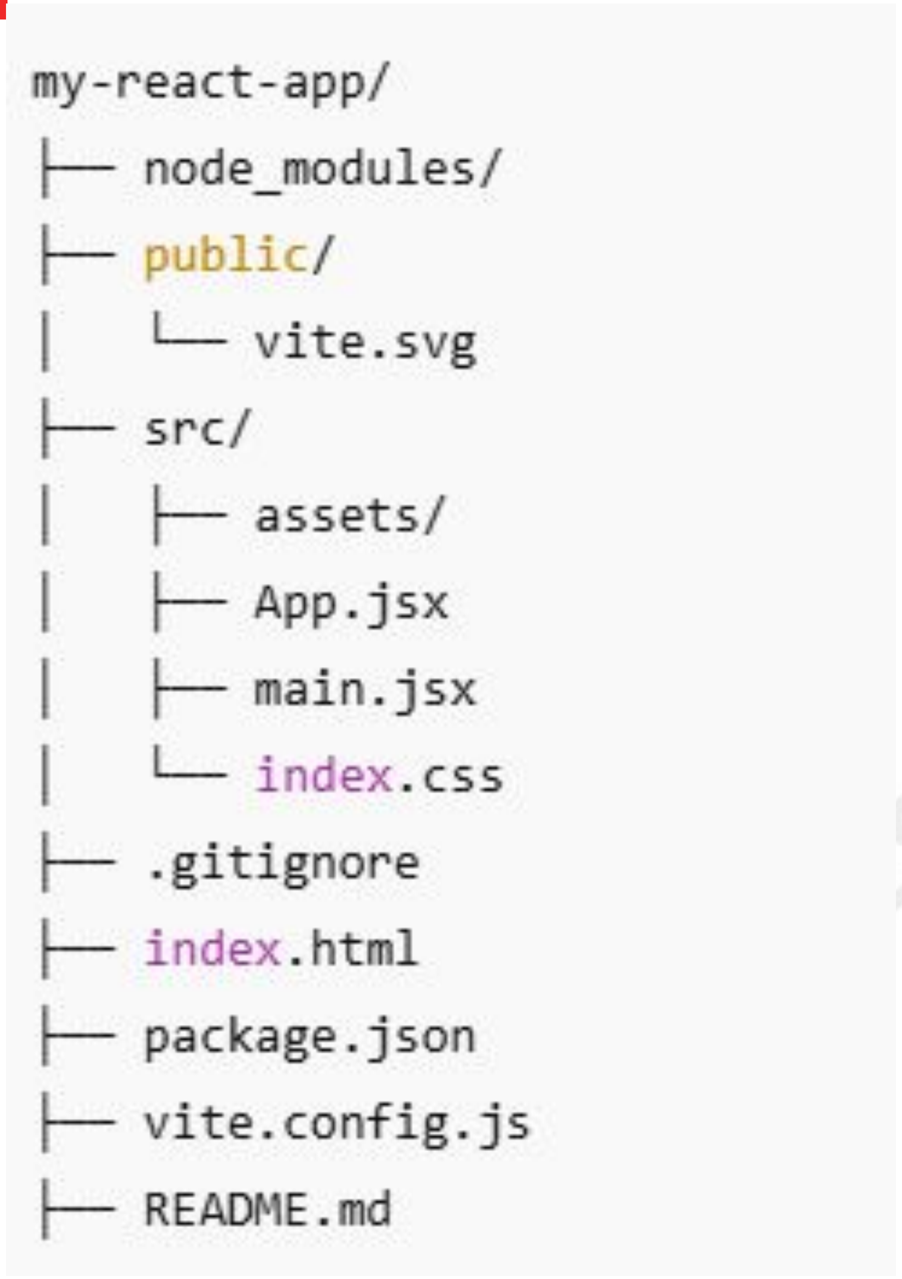
3. Navigate into the project folder : **cd my-react-app**

4. Install dependencies: **npm install**

5. Run the development server: **npm run dev**

Your project will run at http://localhost:5173

Folder Structure



```
my-react-app/
├── node_modules/
├── public/
│   └── vite.svg
├── src/
│   ├── assets/
│   ├── App.jsx
│   ├── main.jsx
│   └── index.css
├── .gitignore
├── index.html
├── package.json
├── vite.config.js
├── README.md
```

node_modules: Auto-generated folder where all the dependencies (React, Vite, etc.) are stored.

public: Contains static files you don't want Vite to process.

src: Where you write all your React code (components, styles, logic).

assets (Optional): Place for images, custom styles, fonts, etc.

App.jsx: Main React component of your application.

main.jsx: The entry point of your app.This is where the React app is linked with the root element of the HTML (index.html).

index.css: You can write global CSS styles here.

index.html: The HTML template for your React app.

package.json: Holds project details and manages dependencies.

.gitignore: Tells Git which files/folders to ignore when uploading to GitHub.

README.md: A markdown file that describes the project. Good for documentation, instructions, or project info.

vite.config.js: Vite’s configuration file.

JSX

JSX = JavaScript + HTML

It allows you to write HTML-like code inside JavaScript, which React converts into actual DOM elements.
File Extension - .jsx

Why Use JSX?

Cleaner and more readable
describe the UI structure like HTML
Helps in embedding dynamic logic (like if, map, etc.)

JSX Rules -

Rule	Example
Use camelCase for attributes	className, onClick, htmlFor
Only one parent element returned	Wrap in <div> or <>...</>
Use curly braces {} for JavaScript	{name}, {items.map()}, {condition && ...}
Self-close tags if empty	<img />, <input />

TSX

TSX = TypeScript + HTML

It’s just like JSX, but with **TypeScript support** (i.e., typing, interfaces, and error checking).

File extension: .tsx

Use TSX when:

- You want type safety
- You're building larger apps
- You're working in a team
- You're using TypeScript in your project

Components

Components are the building blocks that comprise a React application representing a part of the user interface.

A component used in one area of the application can be reused in another area. This helps speed up the development process.

A component can contain several other components.

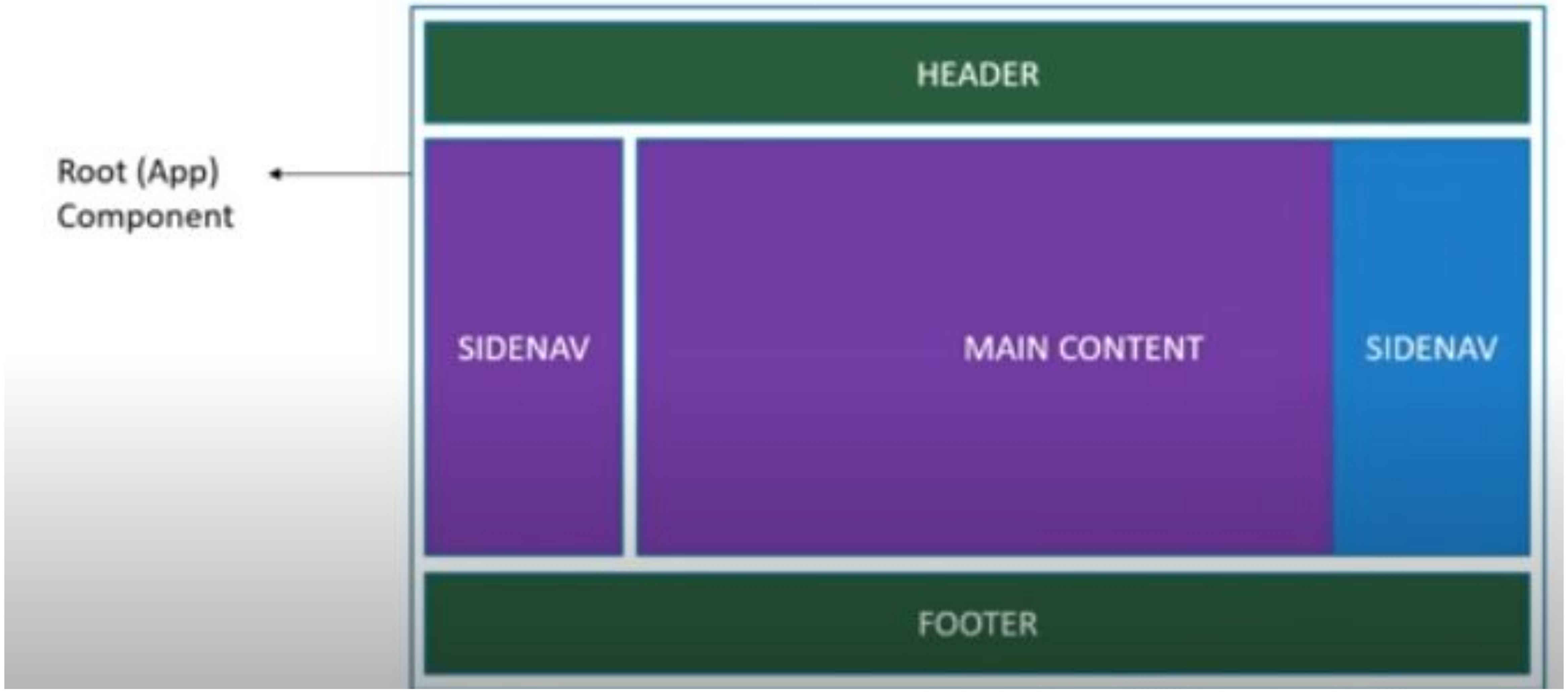
Think of components like **LEGO blocks** — you assemble small blocks (components) to build a big structure (a complete app).

Why Use Components?
To split UI into small, manageable parts
To reuse code across the app
To organize logic and styles better

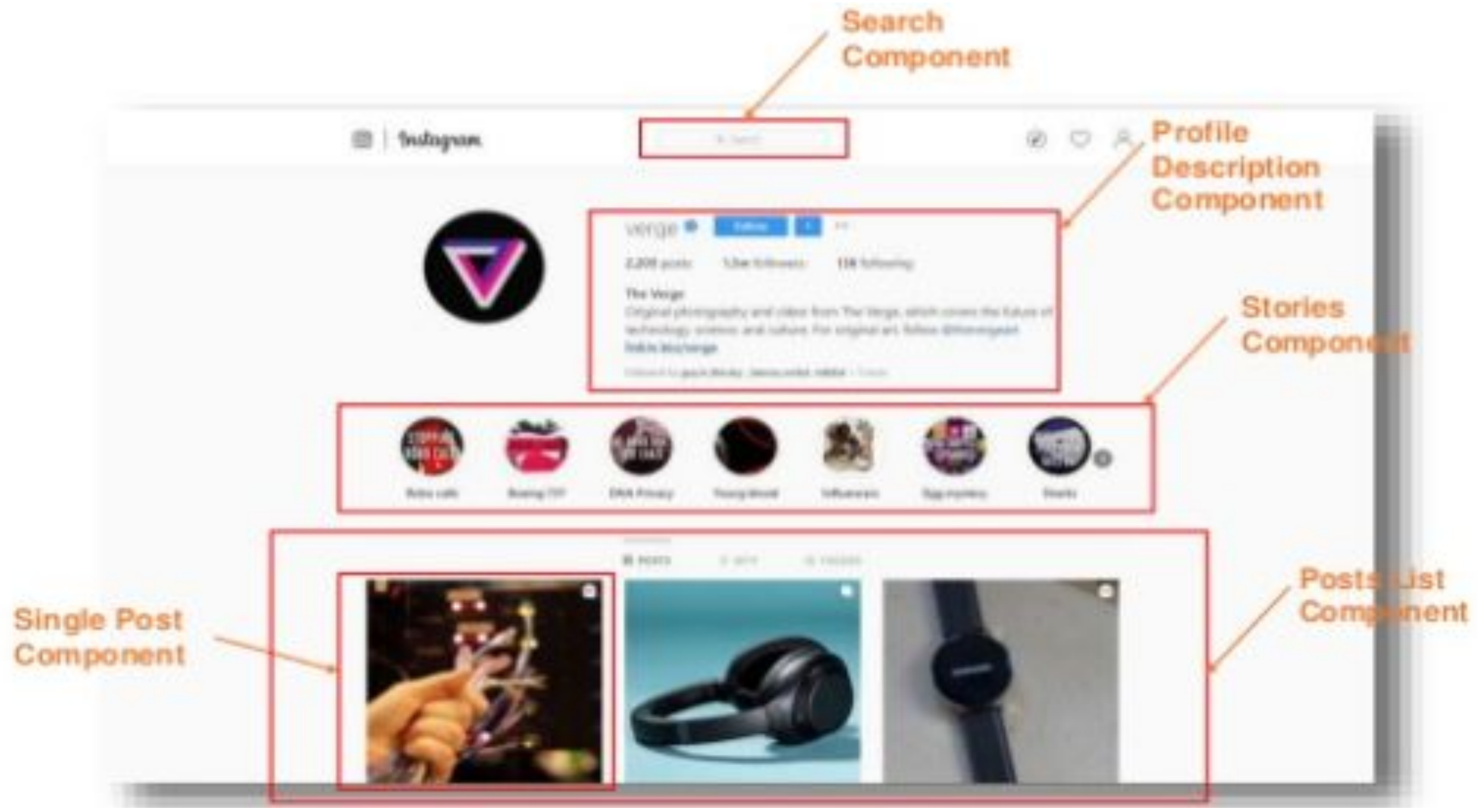
Components

A component is combination of
1. Template using HTML
2. User Interactivity using JS
3. Applying Styles using CSS

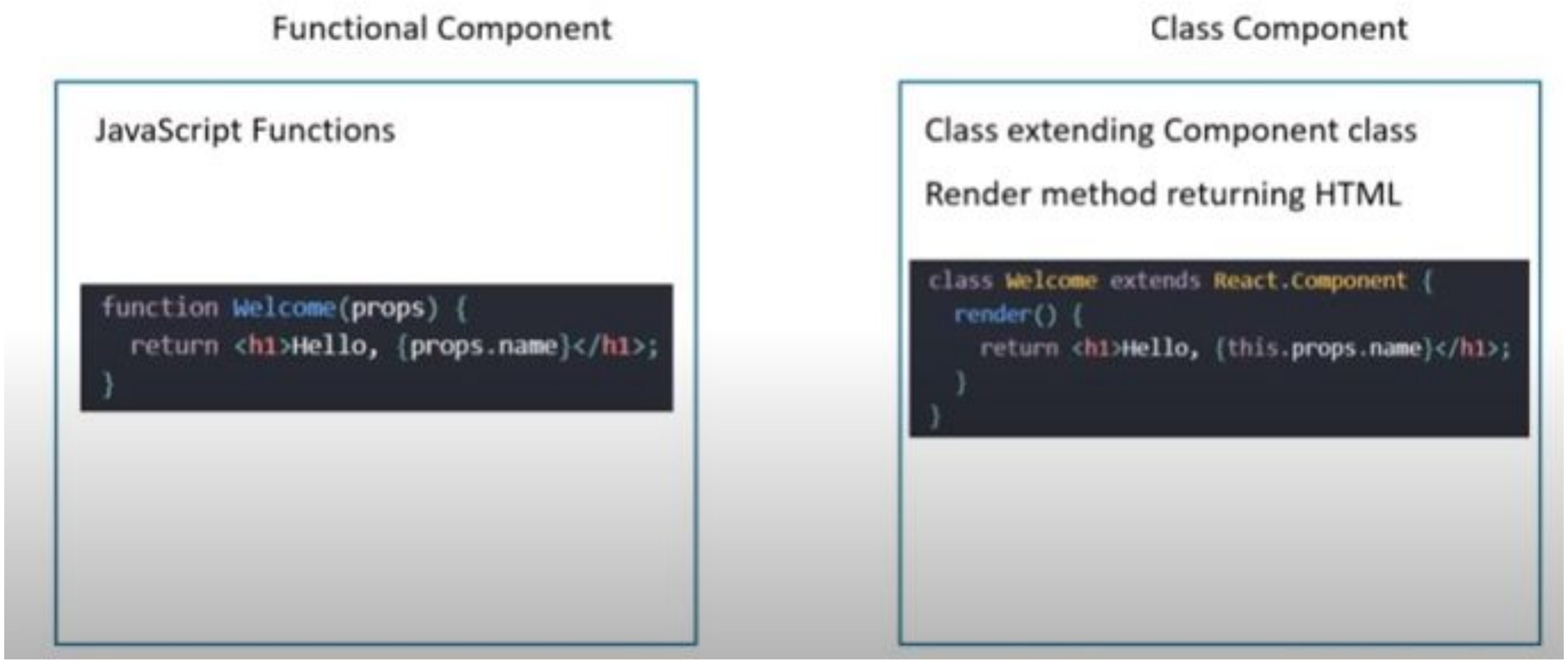
Note: Always start component names with a capital letter. React treats components starting with lowercase letters as DOM tags.


Root (App)
Component
HEADER
SIDENAV
MAIN CONTENT
SIDENAV
FOOTER

Let’s see how React works in real time



Types of Components



Functional Component

JavaScript Functions

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

Class Component

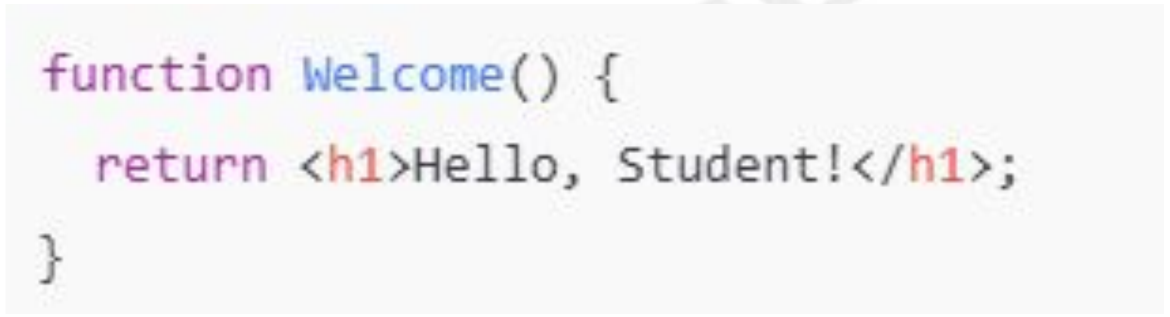
Class extending Component class

Render method returning HTML

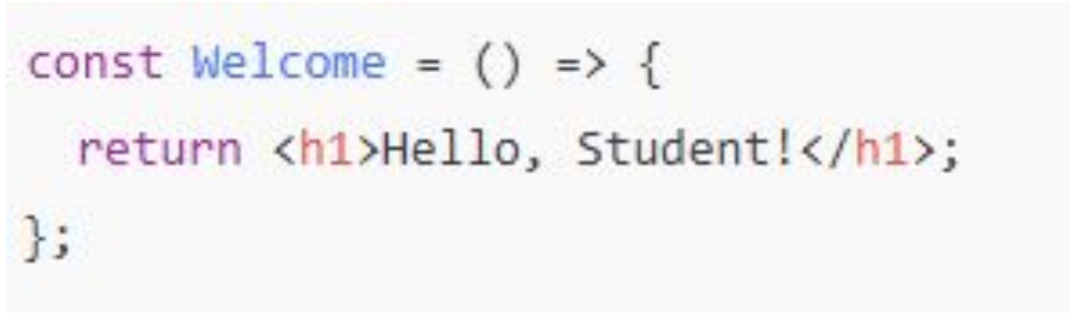
```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Functional Component

A Functional Component is a simple JavaScript function that returns JSX (UI code). It represents a part of the UI in a reusable way.
Preferred in modern React (v16.8 and later)
Simpler Syntax and Shorter code
Supports Hooks like useState, useEffect, etc.
Performance optimized compared to class components



```
function Welcome() {
  return <h1>Hello, Student!</h1>;
}
```



```
const Welcome = () => {
  return <h1>Hello, Student!</h1>;
};
```

Class Component

A Class Component is a React component defined using ES6 class syntax, and it extends React.Component. It must contain a render() method that returns JSX.

```
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>;
} }
< Employee Name="Harshita“ DeptName=“FCAIT” />
```

render()

Class components uses render function.
React renders HTML to the web page by using a function called render().
The purpose of the function is to display the specified HTML code inside the specified HTML element.
In the render() method, we can read props and state and return our JSX code to the root component of our app.
In the render() method, we cannot change the state, and we cannot cause side effects(such as making an HTTP request to the webserver).

props

Props is short for properties, that allow us to pass argument or data to components.
Props are **read-only** inputs passed from a **parent component to a child** component.
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

How Props Works??

1. Pass props in parent component
```
<Comp1 name="Harshita" />
```

2. Receive props in child component

Functional Component -
```
function Comp1(props) {
                    return <h1>Hello, {props.name}</h1>
                }
```

Class Component -
```
class Comp1 extends React.Component {
                render() {
                 return <p>Welcome, {this.props.name}</p>;
                  }
                }
```

Props with Children

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: **props.children.**

```
Ex:- <Employee>
          <h1> Employee of the year </h1>
                 <Home/>
     </Employee>
```

Class Component – {this.props.children} // Employee of the year and Home Component Data
Function Component – {props.children}

Note :- to access particular children - ex. First one -> props.children[0] (0 is index here)

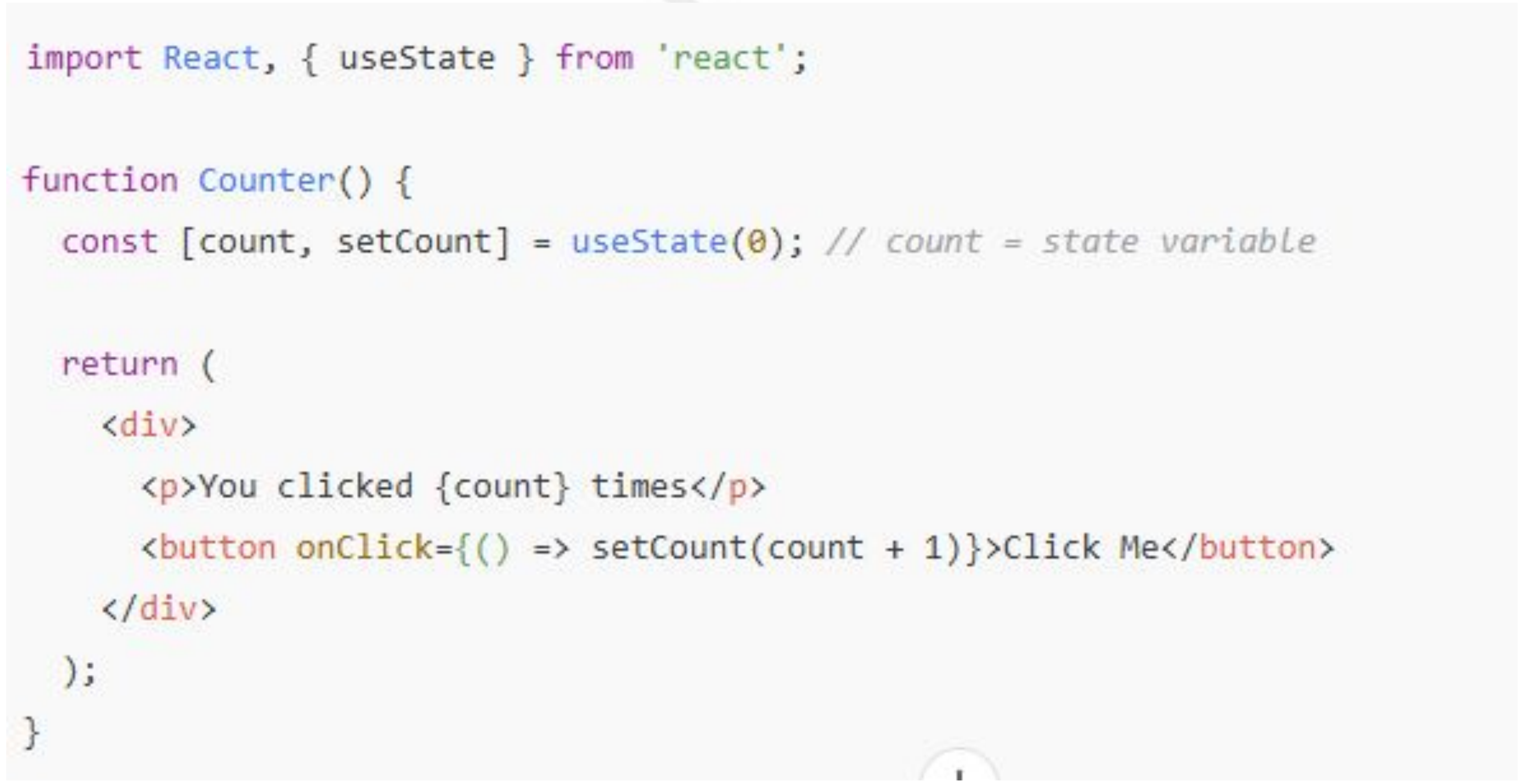
state

State is a built-in object in React used to store data that changes over time in a component.

state allows a component to manage its own data internally.

State in Functional Components (v16.8)

Hooks are a new addition in React 16.8.
Hooks are nothing but the function that let you "hook into" React state and lifecycle features from function components.
useState hook is to manage state in functional components.

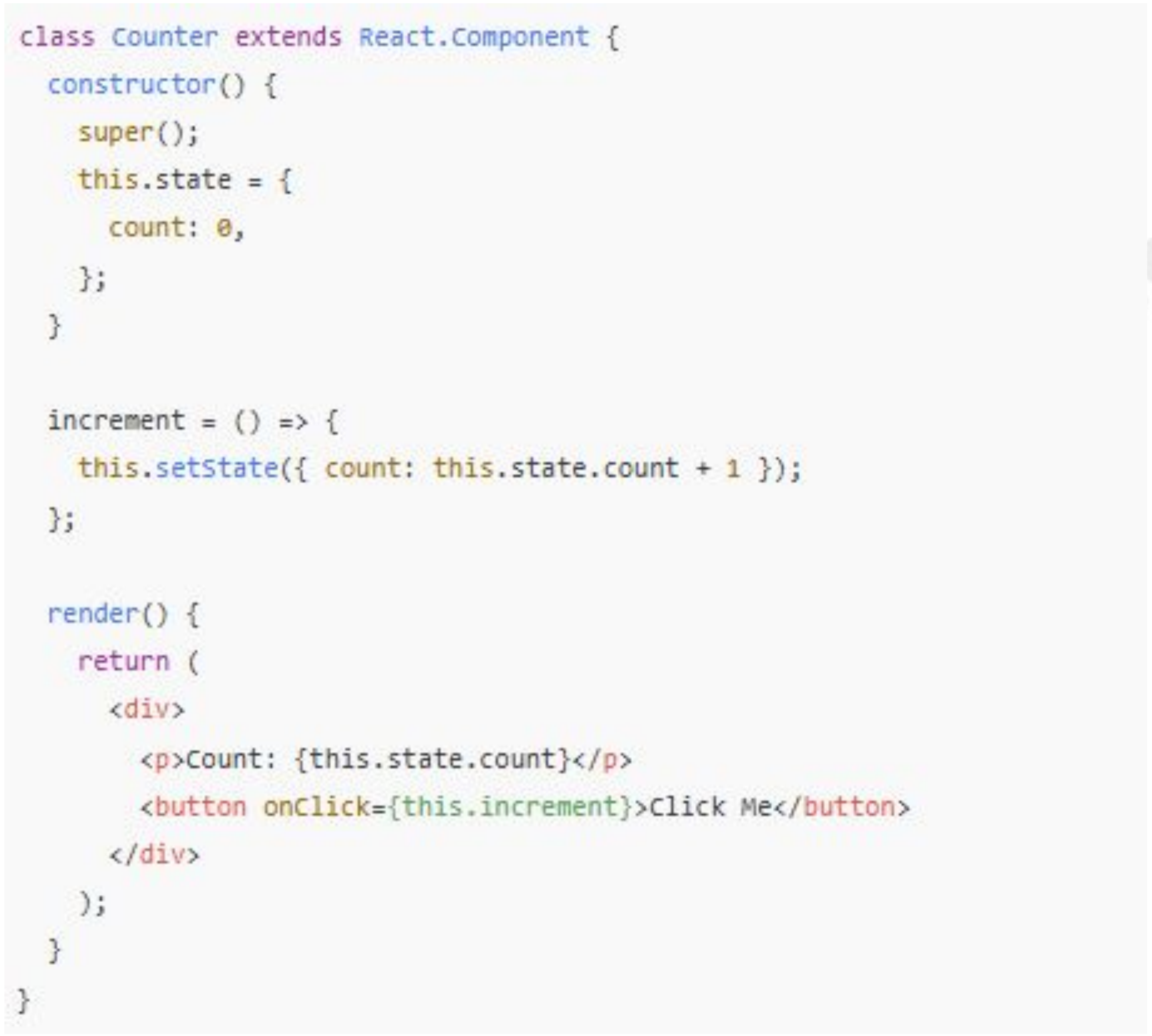


```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0); // count = state variable

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click Me</button>
    </div>
  );
}
```

State in Class Components (Old Syntax)



```
class Counter extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Click Me</button>
      </div>
    );
  }
}
```

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 this.props – class components	useState Hook – Functional Components This.state – Class Component

Event Handling

Handling events with React elements is very similar to handling events on DOM elements.
There are some syntactic differences:
React events are named using camelCase, rather than lowercase.
With JSX you pass a function as the event handler, rather than a string.

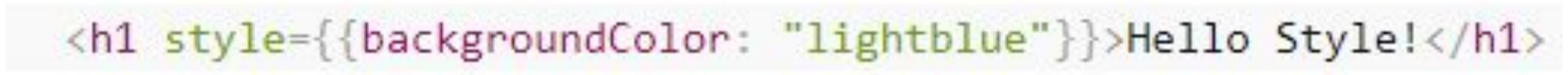
In HTML: <button onclick=“handleClick()">Click Me</button>

In React

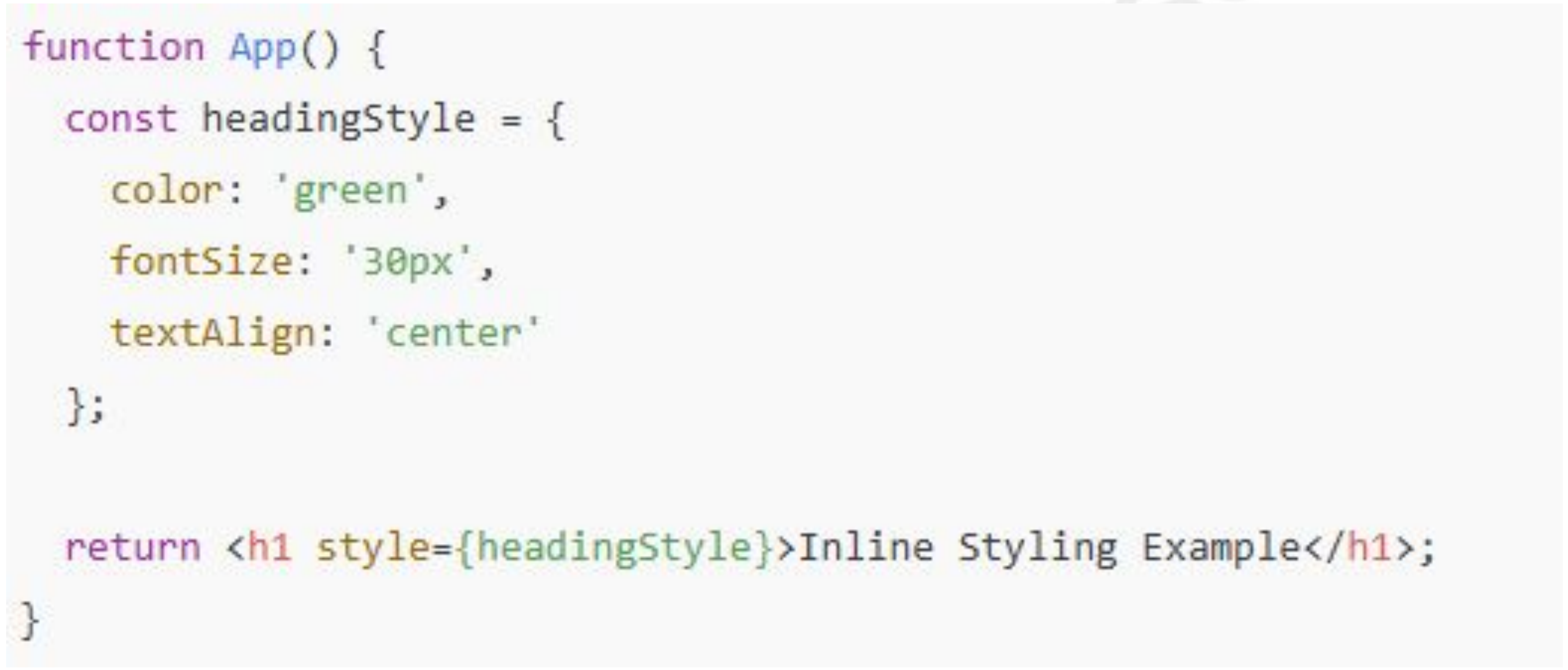
```
 <button onClick={handleClick}>Click Me</button> // Function Component
 <button onClick={() => clickHandler(“Hello")}>Click Me!!</button>
<button onClick={() => console.log(“Hello")}>Click Me!!</button>

 <button onClick={this.handleClick}>Click Me</button>    // Class Component
```

Inline styling



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



```
function App() {
  const headingStyle = {
    color: 'green',
    fontSize: '30px',
    textAlign: 'center'
  };

  return <h1 style={headingStyle}>Inline Styling Example</h1>;
}
```

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>
 </>
```

Conditional Rendering

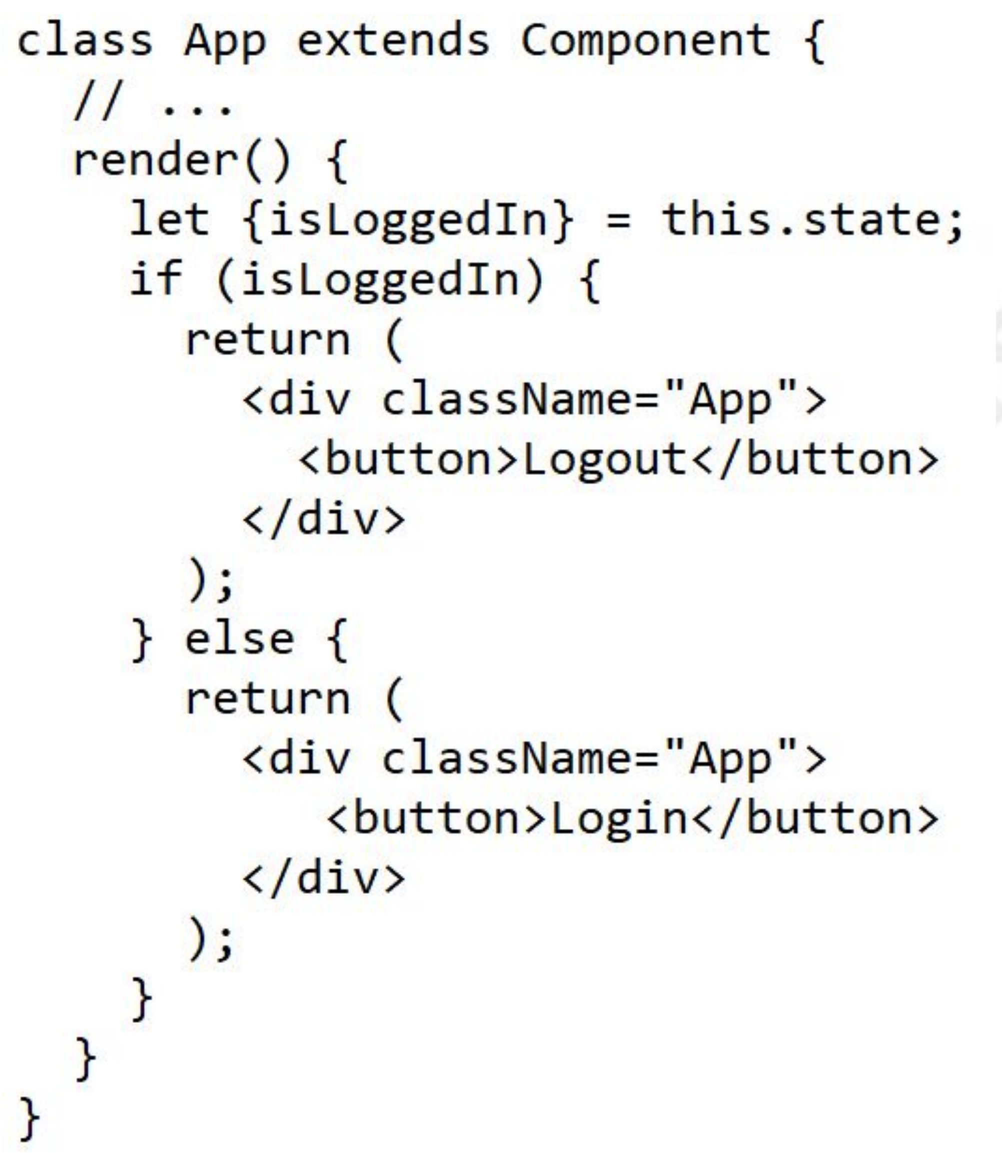
Conditional rendering is a term to describe the ability to render different user interface (UI) markup if a condition is true or false. In React, it allows us to render different elements or components based on a condition. This concept is applied often in the following scenarios:

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

Conditional Rendering Approaches:

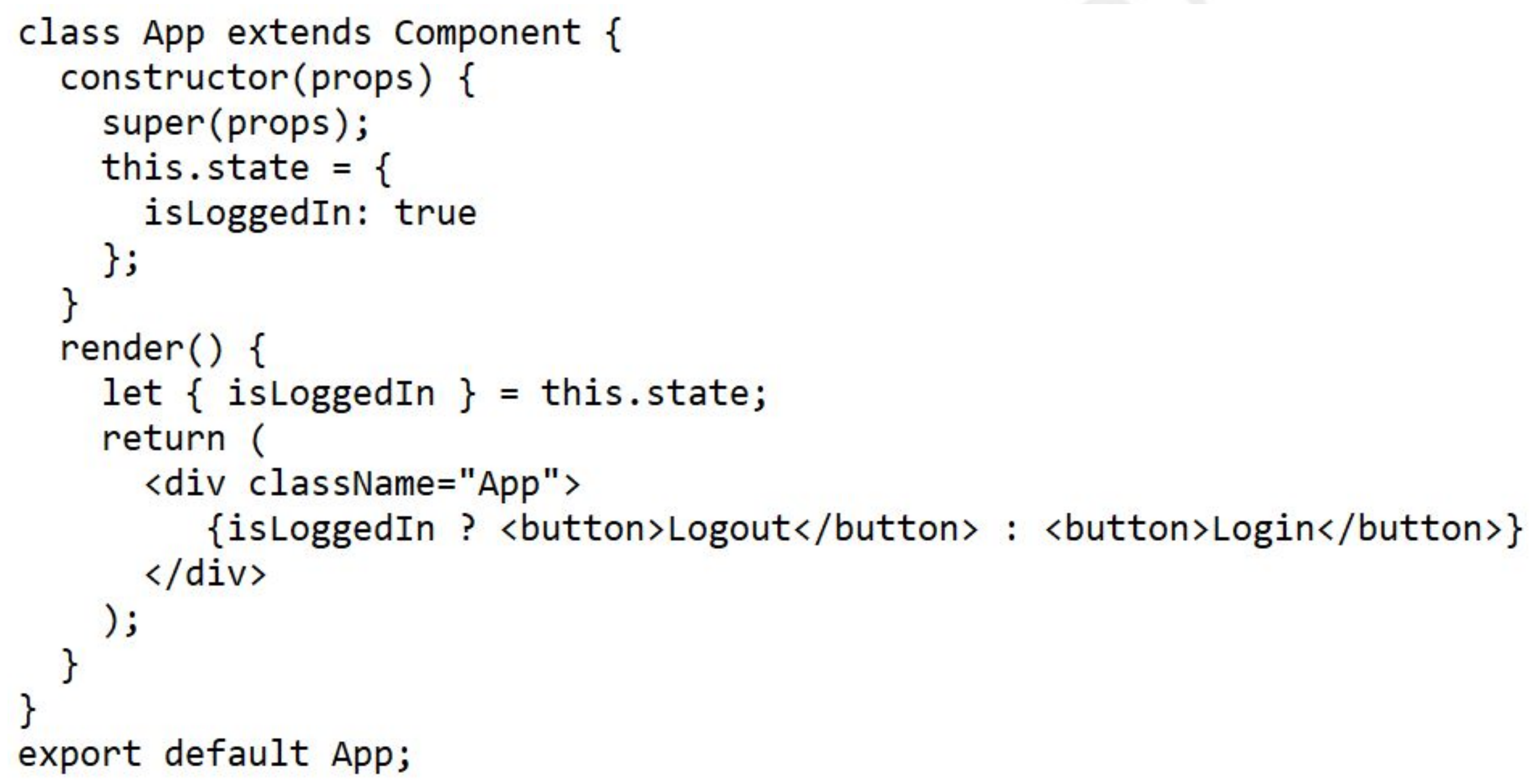
If/else
Ternary conditional operator
Short Circuit Operator

If else



```
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>
      );
    }
  }
}
```

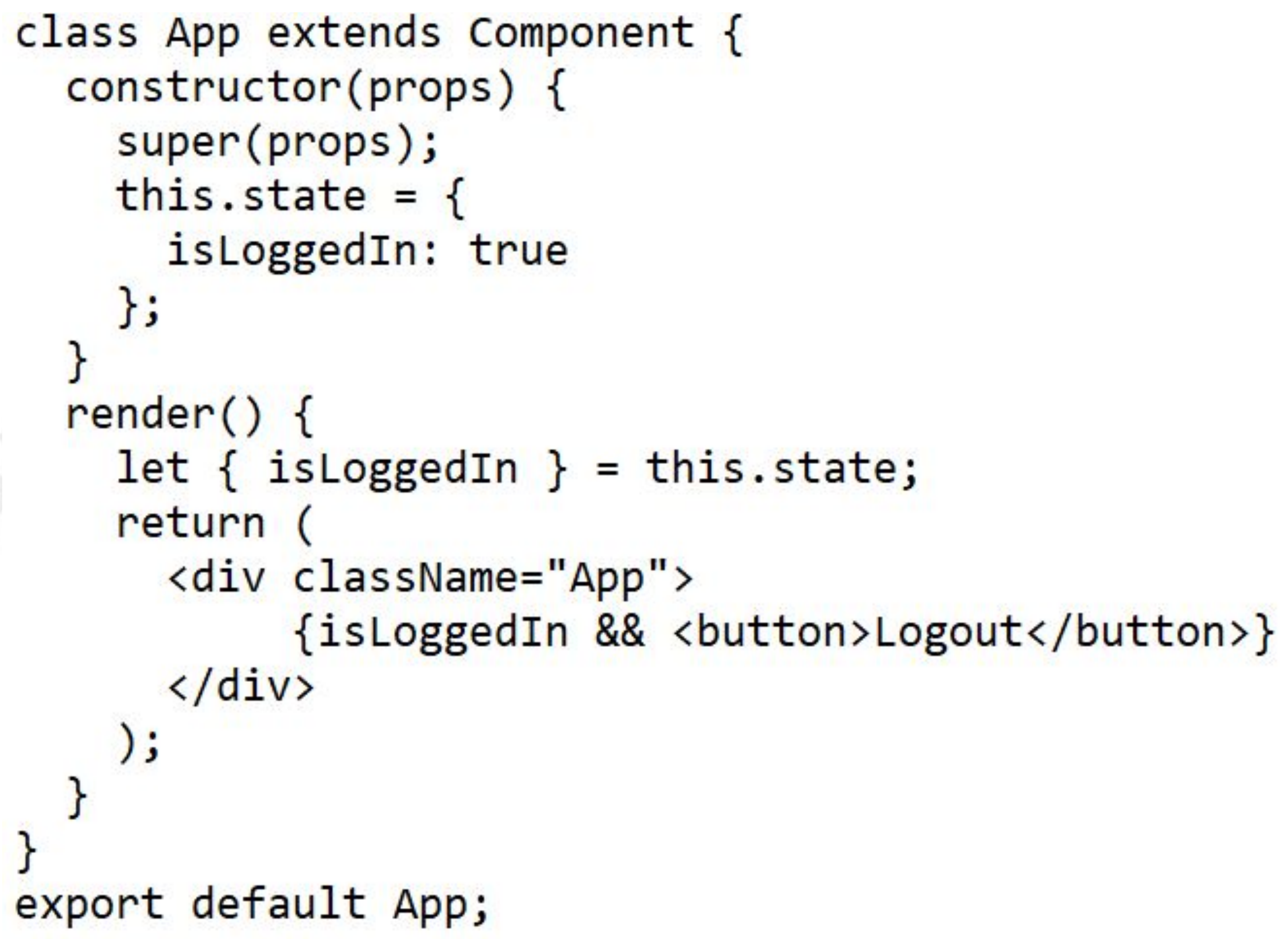
Ternary operator approach



```
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.

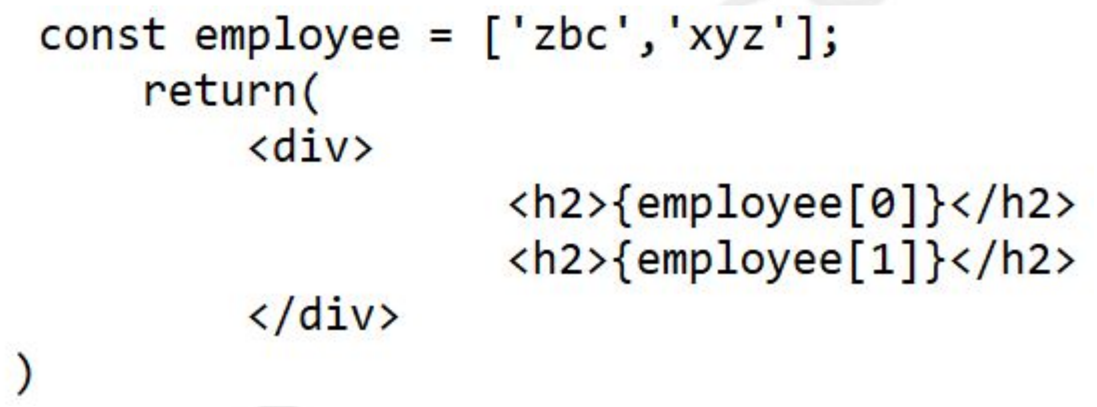


```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isLoggedIn: true
    };
  }
  render() {
    let { isLoggedIn } = this.state;
    return (
      <div className="App">
           {isLoggedIn && <button>Logout</button>}
      </div>
    );
  }
}
export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

List Rendering

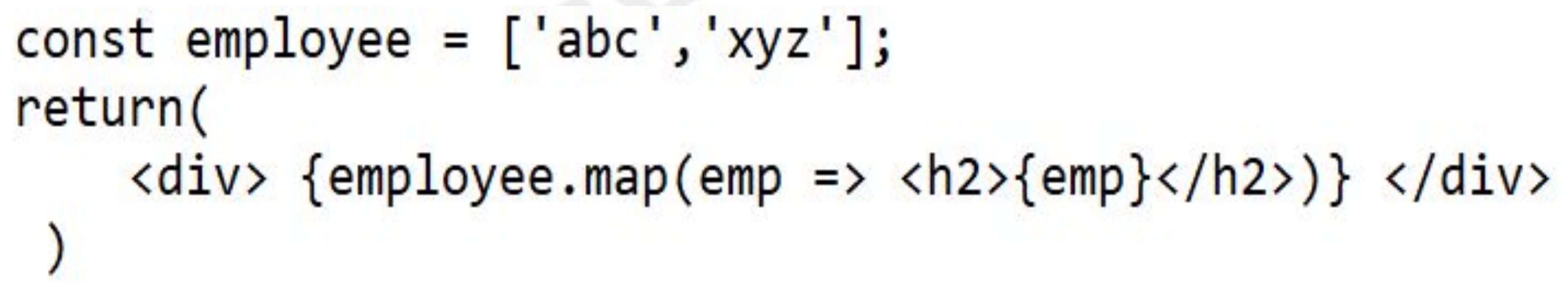
In React, List Rendering refers to displaying a list of items using the .map() function, and the key prop helps React identify which items have changed, added, or removed.

Using array index:



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

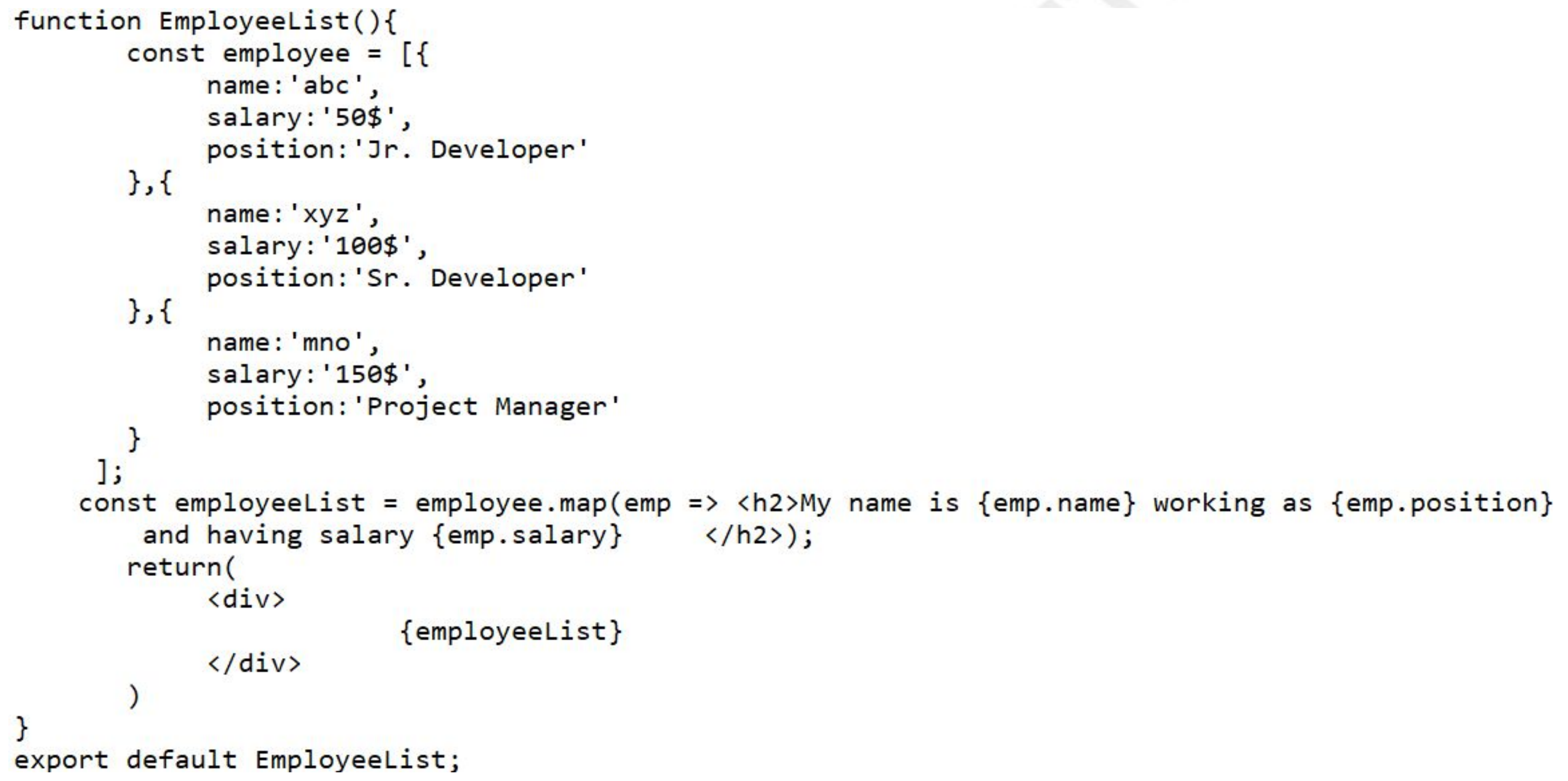
Array.map():



```
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;
```

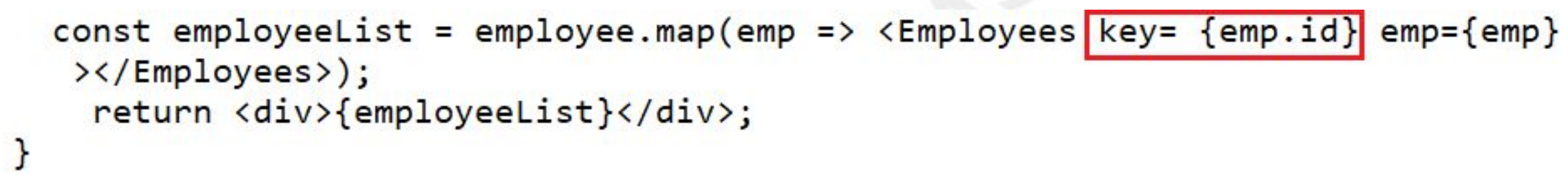
Render List in Sub-Component

```
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>;
}
```

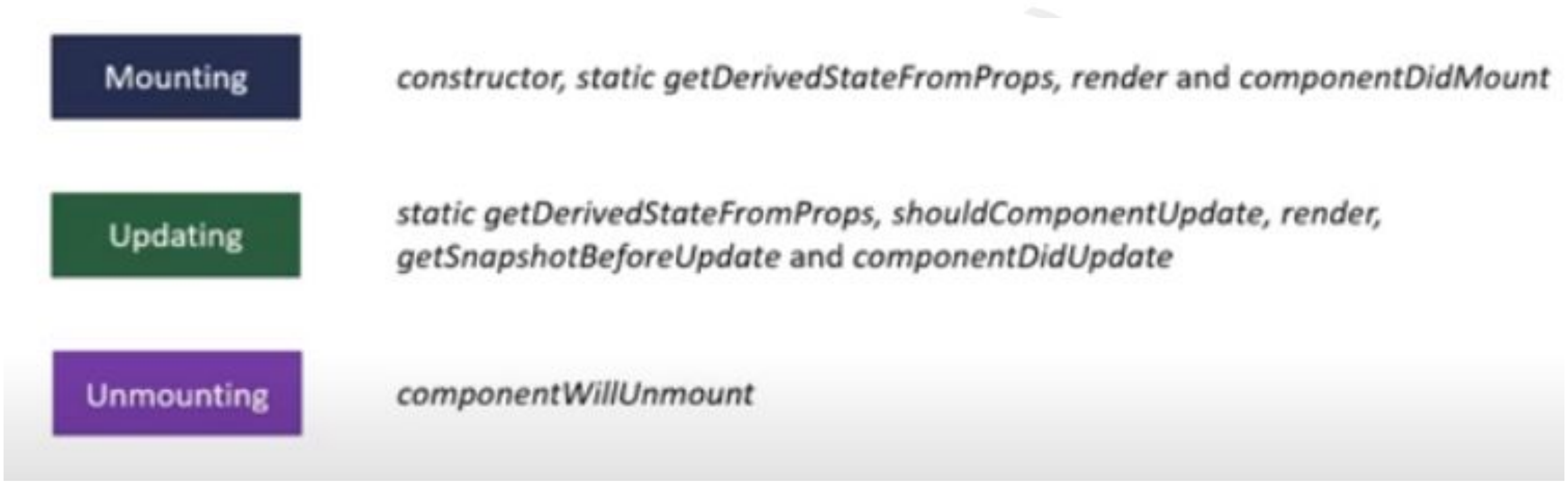
The important point that needs to be kept in mind when using key prop is that key prop cannot be used in child component.
Importance of key prop:
Key prop helps to easily identify which item is added, updated or removed.
Key prop helps in updating user interface efficiently.
Keys provide stable identity to elements.

Class Component Lifecycle Methods

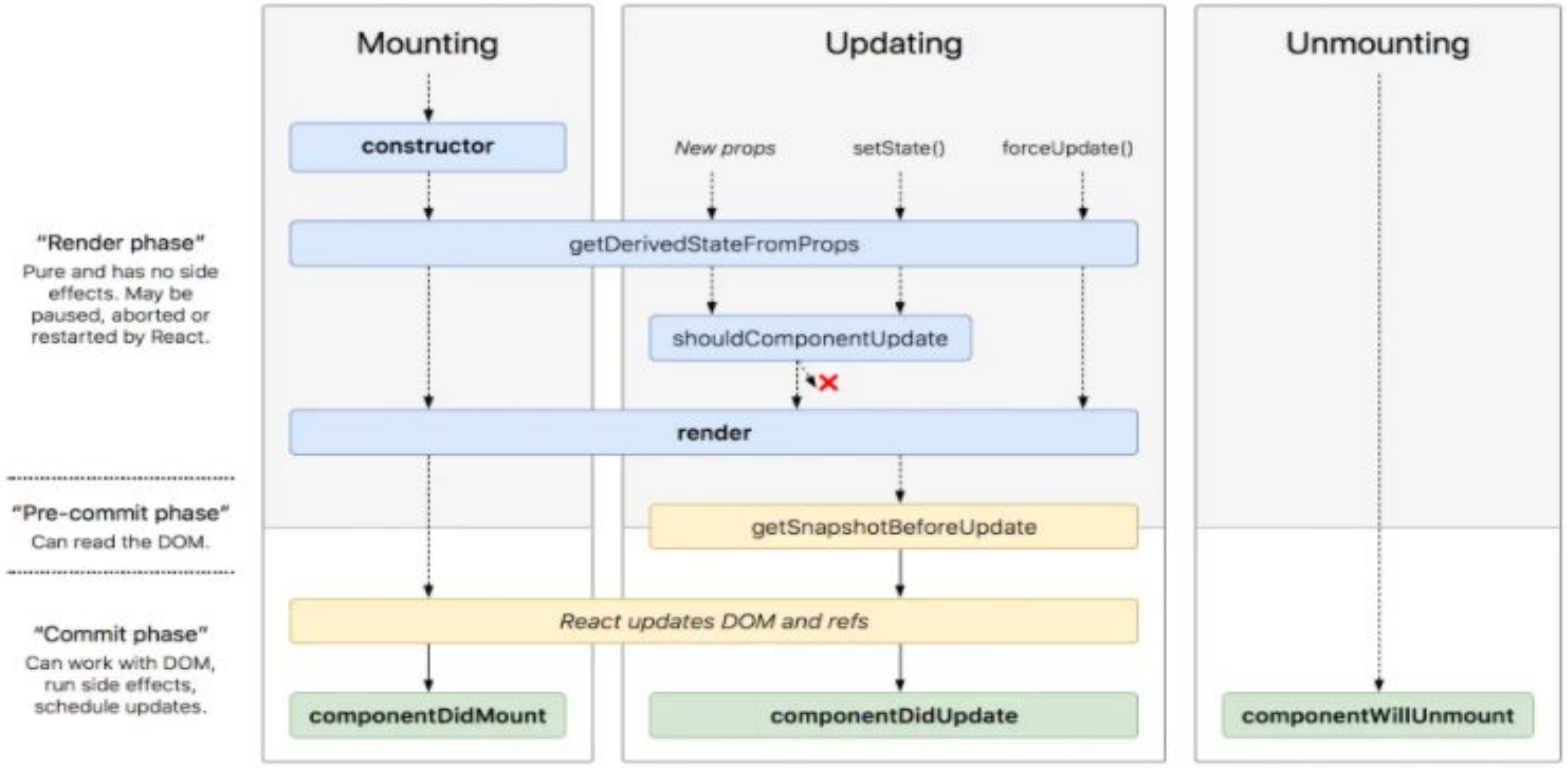
Each component has several “lifecycle methods” that you can override to run code at particular times in the process.

Three Phases of a React Component:

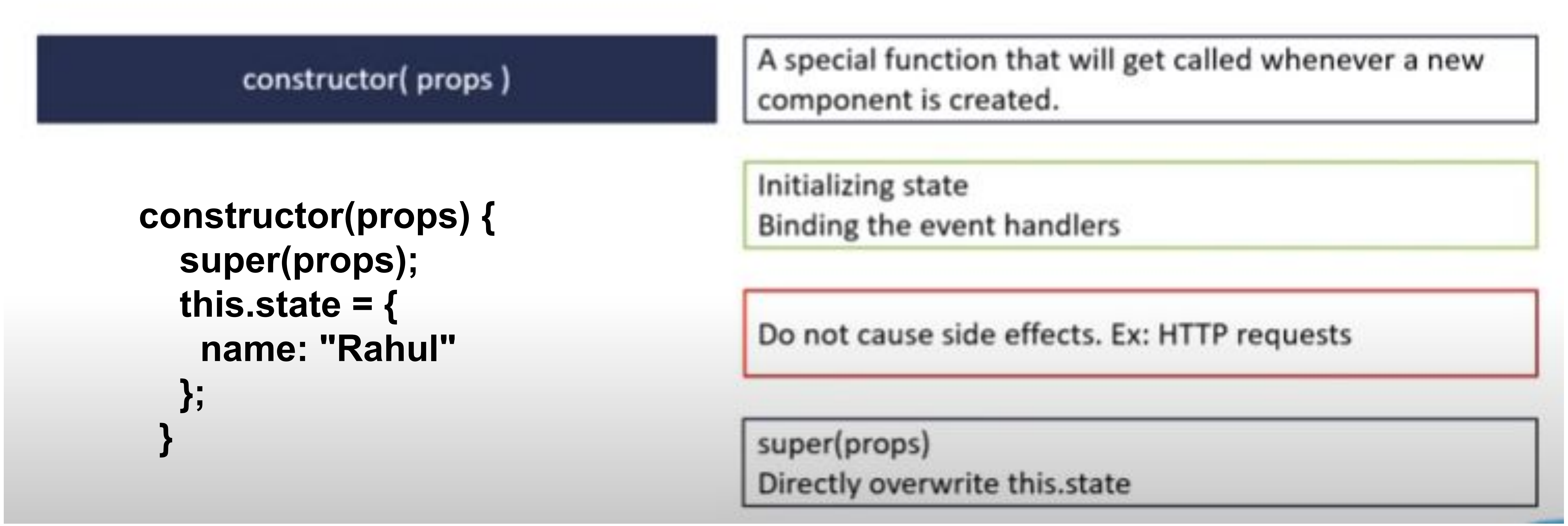
1. **Mounting** – Component is being created and inserted into the DOM.
2. **Updating** – Component is being re-rendered due to changes in props or state.
3. **Unmounting** – Component is being removed from the DOM.



Mounting	*constructor, static getDerivedStateFromProps, render* and *componentDidMount*
Updating	*static getDerivedStateFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate* and *componentDidUpdate*
Unmounting	*componentWillUnmount*


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 Phase



constructor(props)

A special function that will get called whenever a new component is created.

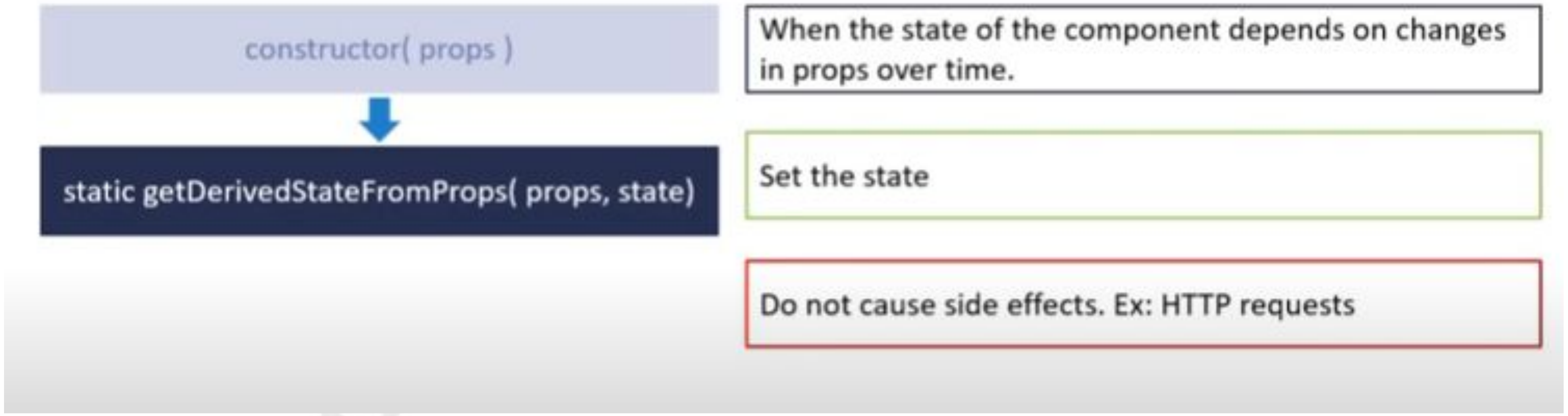
Initializing state
Binding the event handlers

Do not cause side effects. Ex: HTTP requests

super(props)
Directly overwrite this.state

```
constructor(props) {
    super(props);
    this.state = {
      name: "Rahul"
    };
  }
```

Mounting Phase


constructor(props)

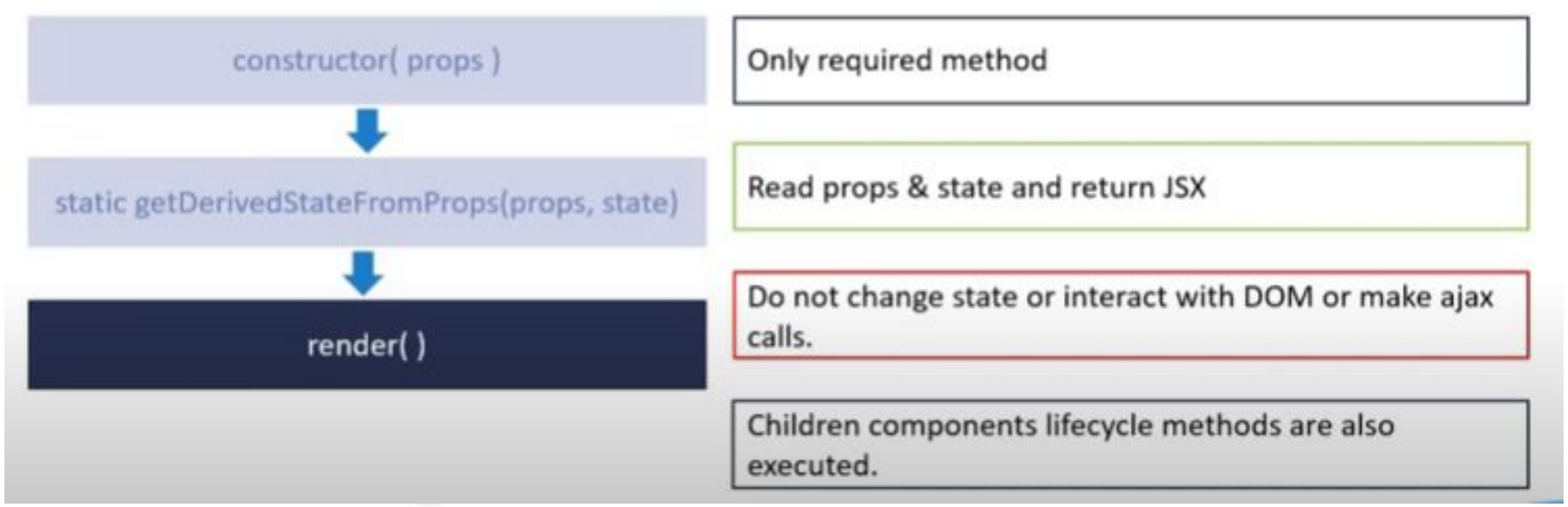
static getDerivedStateFromProps(props, state)

When the state of the component depends on changes in props over time.

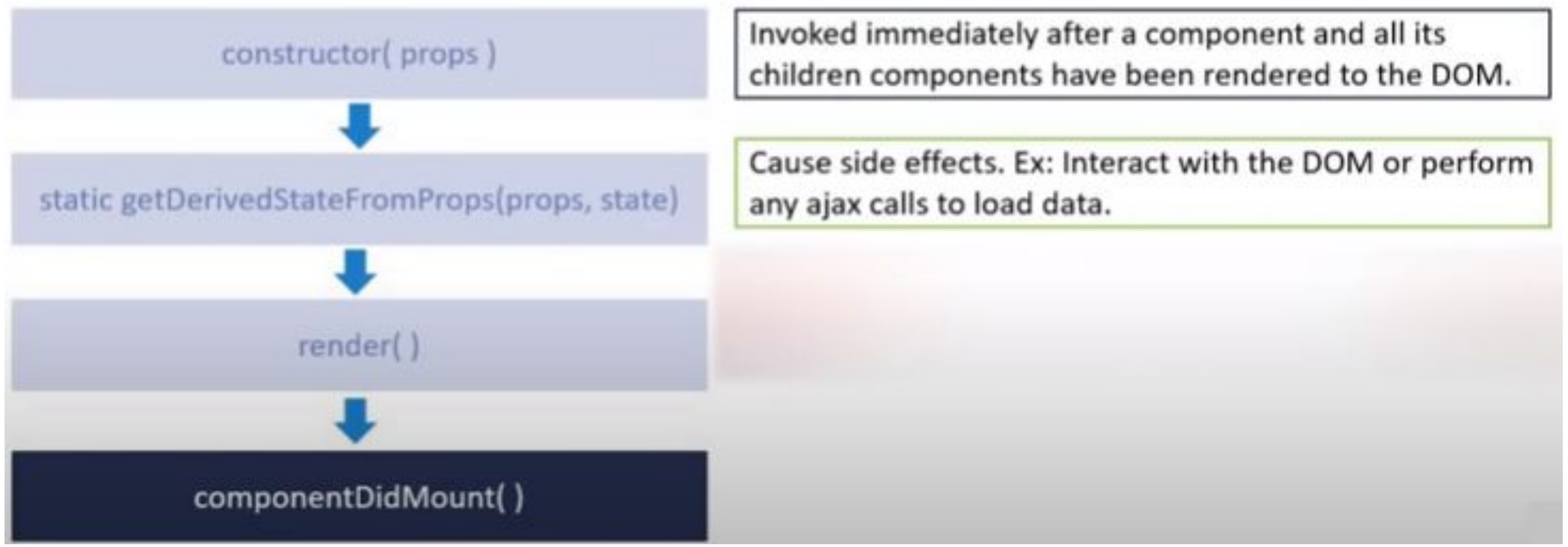
Set the state

Do not cause side effects. Ex: HTTP requests

Mounting Phase


constructor(props)
static getDerivedStateFromProps(props, state)
render( )
Only required method
Read props & state and return JSX
Do not change state or interact with DOM or make ajax calls.
Children components lifecycle methods are also executed.

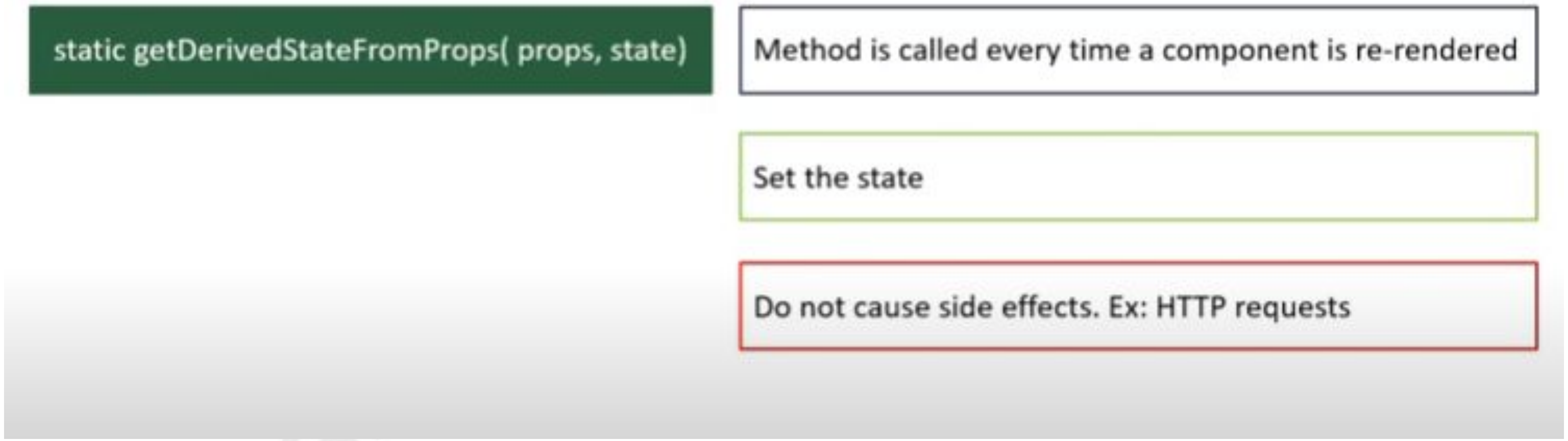
Mounting Phase



Invoked immediately after a component and all its children components have been rendered to the DOM.

Cause side effects. Ex: Interact with the DOM or perform any ajax calls to load data.

Updating Phase

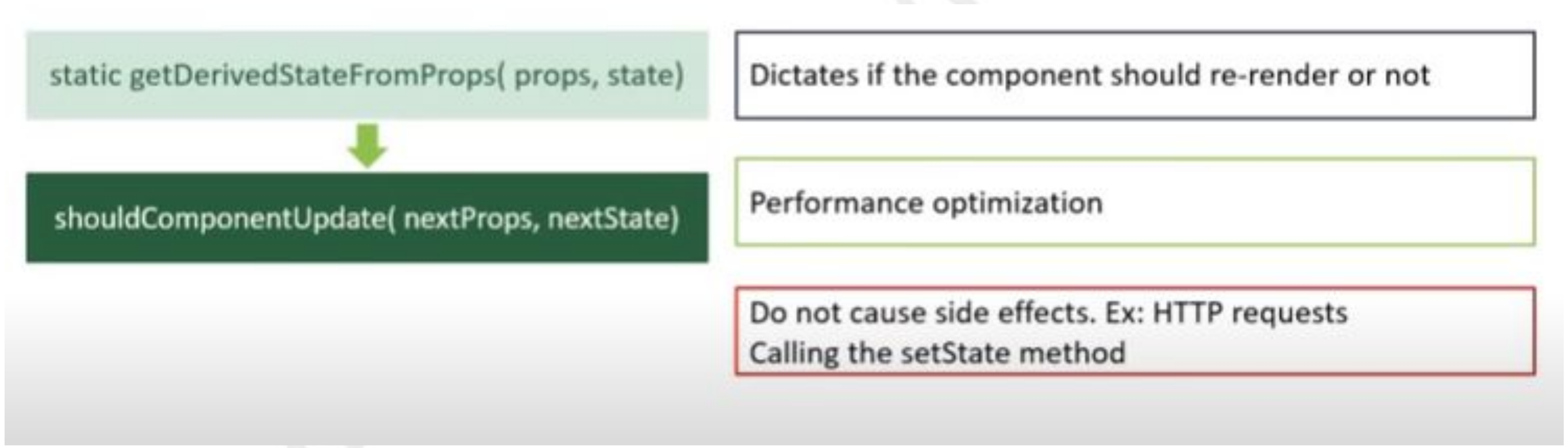

static getDerivedStateFromProps(props, state)

Method is called every time a component is re-rendered

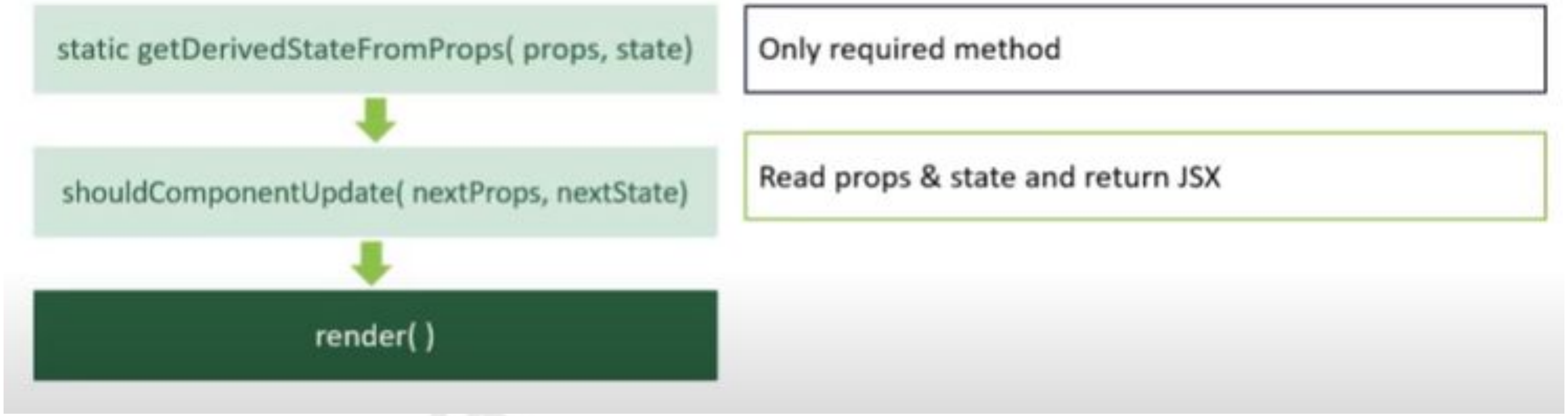
Set the state

Do not cause side effects. Ex: HTTP requests

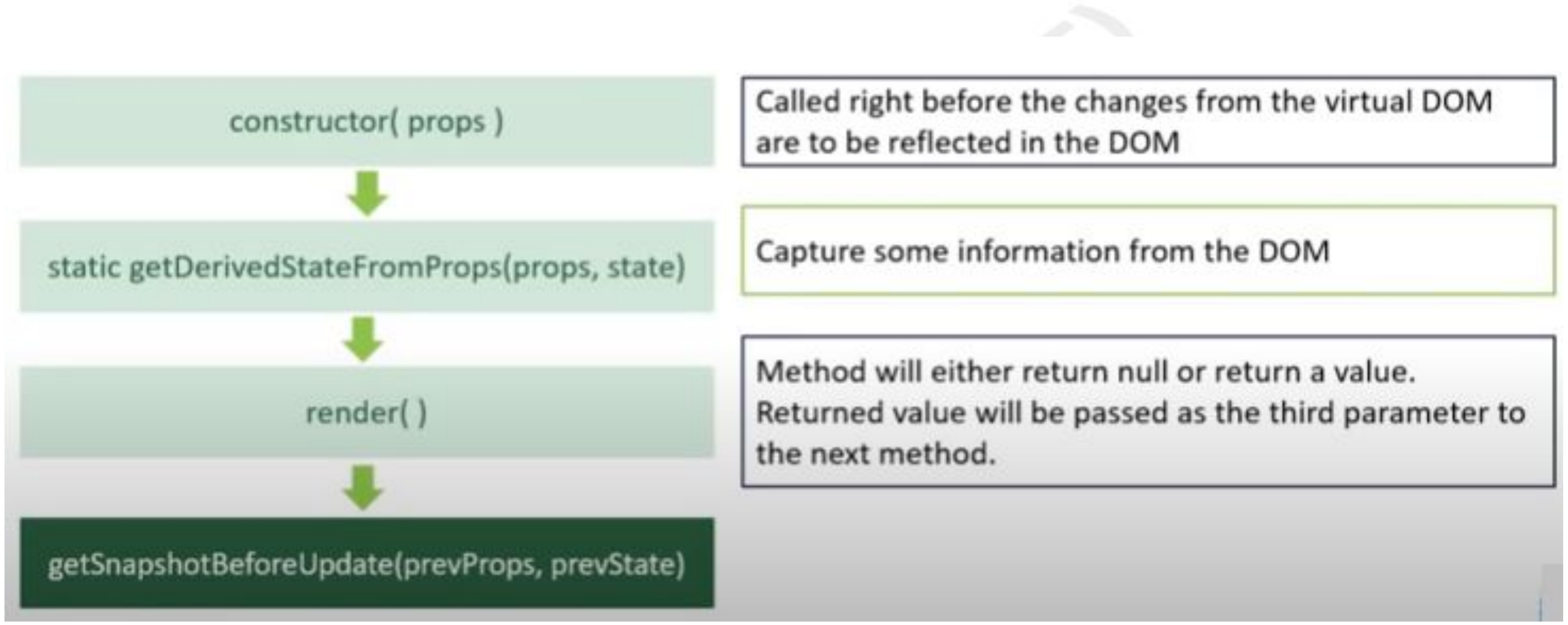
Updating Phase


static getDerivedStateFromProps(props, state)
shouldComponentUpdate(nextProps, nextState)
Dictates if the component should re-render or not
Performance optimization
Do not cause side effects. Ex: HTTP requests
Calling the setState method

Updating Phase


static getDerivedStateFromProps(props, state)
shouldComponentUpdate(nextProps, nextState)
render( )
Only required method
Read props & state and return JSX

Updating Phase

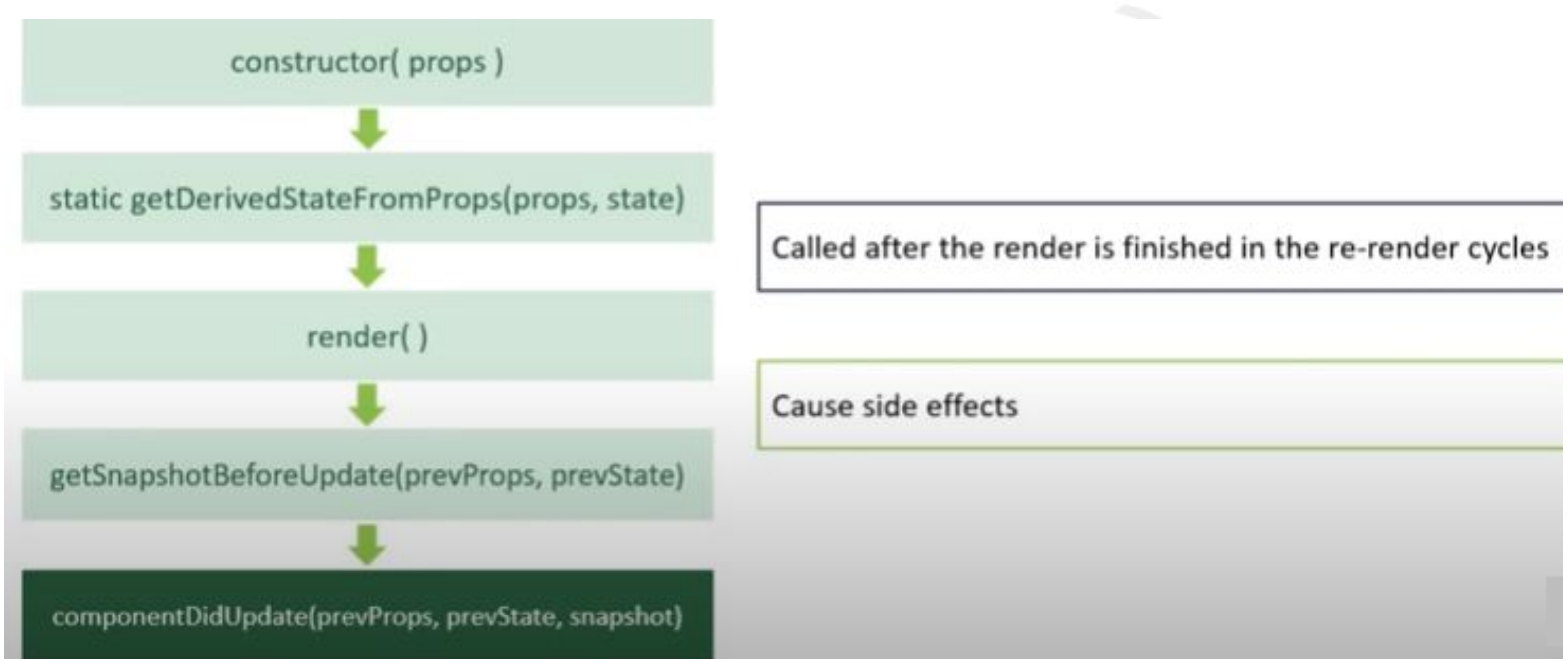

constructor(props)
static getDerivedStateFromProps(props, state)
render( )
getSnapshotBeforeUpdate(prevProps, prevState)

Called right before the changes from the virtual DOM are to be reflected in the DOM

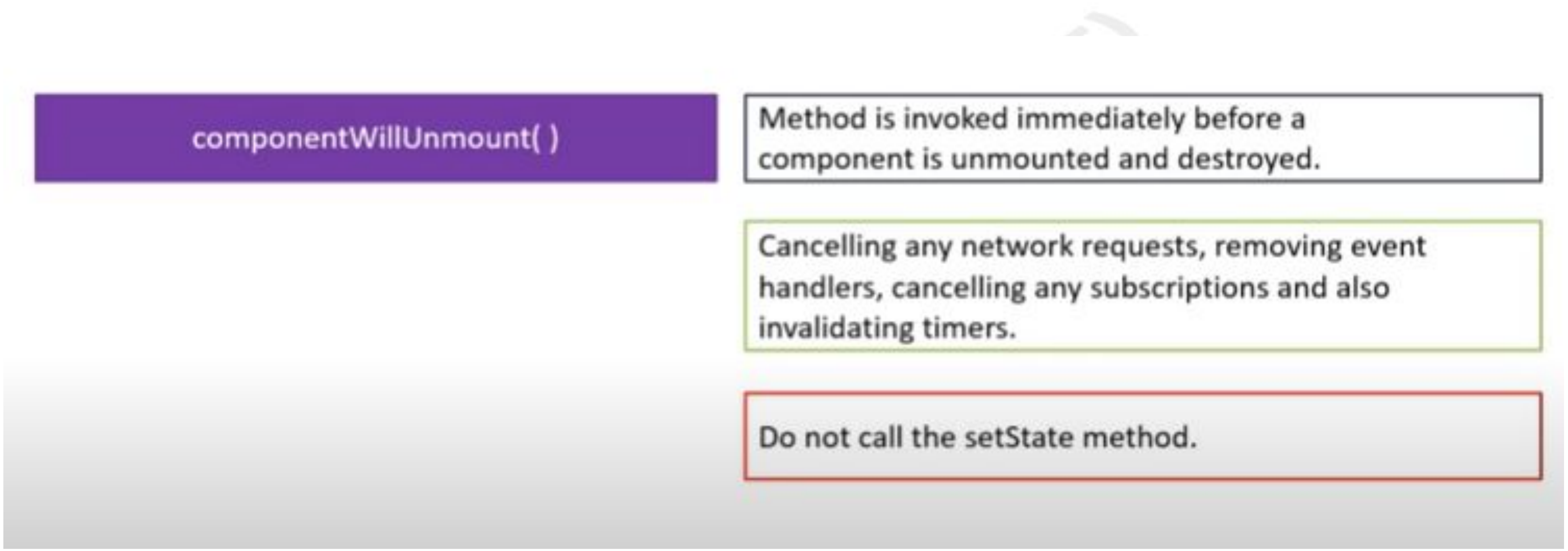
Capture some information from the DOM

Method will either return null or return a value. Returned value will be passed as the third parameter to the next method.

Updating Phase


constructor(props)
static getDerivedStateFromProps(props, state)
render( )
getSnapshotBeforeUpdate(prevProps, prevState)
componentDidUpdate(prevProps, prevState, snapshot)
Called after the render is finished in the re-render cycles
Cause side effects

Unmounting Phase



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.

Thank you