**Understanding class based components**

class MyComponent extends React.Component {

constructor(props) {

super(props); // Call parent class constructor (if any)

this.state = { /\* initial state object \*/ };

}

// ... rest of the component methods

}

**Explanation:**

1. extends React.Component**:** This line establishes that your class inherits from the base React.Component class, which provides functionalities needed for React components.
2. constructor(props)**:** The constructor is a special method that is invoked when a new instance of the component is created.
   * + If your current component is called by any other component (Let’s say any parent component) with some props then, the props argument holds the properties passed down from the parent component,
3. In React class-based components, the super(props) call within the constructor serves two important purposes:

**1. Inheriting Properties from React.Component:**

* + - When you extend your class component from React.Component, you inherit various methods and functionalities from the base class i.e, React.Component .
    - These inherited functionalities include lifecycle methods like render, componentDidMount, etc., which are essential for building React components.
    - By calling super(props), you ensure that these **inherited properties are properly initialized** for your component instance.

**2. Receiving Props from Parent Components:**

* + - When a component is used within another component (called a parent component), the parent can pass data to the child component through **props**.
    - These props are essentially key-value pairs that hold the information the child component needs to function.
    - By calling super(props), your component **receives and stores the props object** passed down from its parent. This allows you to access those props within your component using the this.props property.

1. this.state = ...**:** This line is where you initialize the component's internal state. The object you assign to this.state defines the initial values for your state variables.

[Mount vs Render](https://reacttraining.com/blog/mount-vs-render)  : A must read

**Understanding why we can make componentDidMount async but we cannot make the call back function of useEffect as async function?**

While both componentDidMount and useEffect are used to perform side effects in React components, their approach to handling asynchronous operations differs for two key reasons:

**1. Execution Timing:**

* **componentDidMount:** This lifecycle method runs **synchronously** after the component **finishes rendering** for the **first time**. This means any code within it executes immediately, blocking further rendering until it finishes.
* As it blocks the further rending of until the componentDidMount method get completed, we can make it aync to use await inside it.
* **useEffect:** This hook is **asynchronous (It does not runs synchronously with the code)** as it will runs **after the initial render** and **potentially again** depending on its dependency array. It doesn't block rendering, allowing the UI to update while the effect runs.

**2. Return Value Expectation:**

* **componentDidMount:** This method **doesn't have a return value expectation**. You can use regular synchronous or asynchronous code within it without causing errors.
* **useEffect:** This hook expects its **callback function to return either a cleanup function or nothing**. This means directly making the callback function async is problematic as it creates a promise-like object instead of a function or undefined.
* As async function will return a promise, which useEffect does not expects

**Here's why making useEffect callback async is not recommended:**

1. **Unexpected Behavior:** If you directly use an async function inside useEffect, it won't work as intended. The function will still be treated as a regular function, and the await keyword will have no effect.
2. **Missing Cleanup:** useEffect relies on the return value to perform cleanup tasks when the component unmounts or re-renders with changed components. An async function doesn't meet this expectation and can lead to memory leaks or unintended side effects.

**Difference in componentDidMount and useEffect()**

Here's an explanation of why componentDidMount and useEffect are not the same, focusing on the key differences and their implications:

**1. Timing of Execution (Synchronous and asynchronous):**

* **componentDidMount** (Class Components): Executes synchronously after the first render. This means it runs before the user sees the results of the initial render, potentially blocking the display of content until it completes.
* **useEffect** (Functional Components): Executes asynchronously after the first render **and subsequent renders**. After the initial render, React updates the DOM and then calls effects. Therefore, users see the UI change before effects run. This creates a smoother user experience.

**2. State and Prop Capture:**

* **componentDidMount:** Captures the state and props as they were during the first render. If the state or props subsequently change, componentDidMount won't run again, so it won't see those updated values. For this purpose we use componentDidUpdate() method.
* **useEffect:** By default, captures the most recent version of state and props on every render cycle. This means it always operates with the latest data available. You can control this behavior using the dependency array in the second argument of useEffect.

**3. Cleanup Management:**

* **componentDidMount:** Doesn't provide a built-in mechanism for cleaning up side effects (e.g., subscriptions, timers, event listeners). You would need to manage this manually in componentWillUnmount.
* **componentWillUnmount :** componentWillUnmount will be used for cleanup.
* **useEffect:** Allows you to return a cleanup function from the callback. This function runs when the component. This ensures a cleaner and more predictable management of side effects.

We can use useEffect hook :

1. Once after initial rendering.
2. Once after initial rendering and when the state variables changes which are mentioned in dependency array.
3. We can also use useEffect hook after every re rendering.
4. We can also use it for a cleanup function.

🡪We can use componentDidMount for only once after the initial rendering r mounting only.

🡪For using after every rending we have a separate method known as componentDidUpdate.

🡪For cleanup purpose we have another method known as componentWillUpdate.

Whereas we can use all the above 3 methods usecase in a single useEffect hook

**CleanUp function in useEffect()**

useEffect(() => {

// Your main effect code here

// Return the cleanup function

return () => {

// Your cleanup logic here

};

}, [/\* dependency array \*/]);