⚛️ React Component Guide

💥 Class-Based vs Function-Based Components

📌 Overview

🆚	**Class-Based Component**	**Function-Based Component**
🏗️ Structure	Uses `class` and `extends React.Component`	Just a function
🧠 State Handling	With `this.state`, `this.setState()`	With `useState()`
⚙️ Lifecycle	Uses lifecycle methods	Uses `useEffect()` and hooks
📥 Props Access	Via `this.props`	Via function parameters `{}`
✅ Preferred Today	❌ Old (still used in some codebases)	✅ Yes, modern standard

💡 Basic Syntax

🧱 Class-Based Component

```
import React from "react";

class Welcome extends React.Component {
  constructor(props) {
    super(props); // ✅ Always call super(props)
    this.state = {
      message: "Welcome to Class Component!",
    };
  }

  render() {
    return <h1>👋 {this.state.message}</h1>;
  }
}

export default Welcome;
```

⚙️ Function-Based Component

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

const Welcome = () => {
  const [message, setMessage] = useState("Welcome to Function Component!");

  return <h1>👋 {message}</h1>;
};

export default Welcome;
```

🧠 Handling `props`

👩‍🏫 In Class-Based:

```
class Greet extends React.Component {
  render() {
    return <h2>👋 Hello, {this.props.name}</h2>;
  }
}
```

🔧 In Function-Based:

```
const Greet = ({ name }) => {
  return <h2>👋 Hello, {name}</h2>;
};
```

🧠 Managing `state`

💼 In Class-Based:

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

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

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

🪄 In Function-Based:

```
import { useState } from "react";

const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>🧮 Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>➕</button>
    </div>
  );
};
```

⌛ Lifecycle Methods vs `useEffect`

⏱️ Class-Based Component Lifecycle:

```
class MyComponent extends React.Component {
  componentDidMount() {
    console.log("🚀 Mounted");
  }

  componentDidUpdate() {
    console.log("🔄 Updated");
  }

  componentWillUnmount() {
    console.log("❌ Unmounted");
  }

  render() {
    return <p>Hello Lifecycle 👋</p>;
  }
}
```

🧙 Function-Based with `useEffect`:

```
import { useEffect } from "react";

const MyComponent = () => {
  useEffect(() => {
    console.log("🚀 Mounted");

    return () => {
      console.log("❌ Unmounted");
    };
  }, []); // empty array = run once

  return <p>Hello Lifecycle 👋</p>;
};
```

💥 Event Handling

📌 Class-Based:

```
class Clicker extends React.Component {
  handleClick = () => {
    alert("Clicked!");
  };

  render() {
    return <button onClick={this.handleClick}>🖱️ Click Me</button>;
  }
}
```

📌 Function-Based:

```
const Clicker = () => {
  const handleClick = () => {
    alert("Clicked!");
  };

  return <button onClick={handleClick}>🖱️ Click Me</button>;
};
```

😈 Bad Practices in Class Components

❌ Modifying state directly:

```
this.state.count = 10; // ❌ Don't do this
```

✅ Correct way:

```
this.setState({ count: 10 }); // ✅
```

❌ Forgetting to bind methods (when not using arrow functions):

```
this.handleClick = this.handleClick.bind(this); // Needed if using normal
functions
```

✅ Best Practices

✅ **Class Component**	✅ **Function Component**
Use arrow functions to auto-bind `this`	Keep components pure and small
Always call `super(props)`	Use hooks (`useState`, `useEffect`) smartly
Use `componentDidMount` for data loading	Use cleanup in `useEffect` for unmount logic
Break big components into smaller ones	Use `useCallback`, `useMemo` for perf

▧ When Should I Use Class Components?

✅ Only when:

- You're working in **legacy codebases**.
- You're maintaining older libraries that still use classes.

🚀 But in modern apps, prefer **function-based components** + **hooks** for:

- Simpler syntax
- Better readability
- More flexible side-effect management

🗒 Final Cheat Sheet Summary

Feature	**Class Component**	**Function Component (Hooks)**
Structure	`class MyComp extends React.Component`	`const MyComp = () => {}`
Props	`this.props.name`	`{ name }` via parameters
State	`this.state`, `this.setState`	`useState` hook
Lifecycle	`componentDidMount`, etc.	`useEffect()`
Events	`this.handleClick = ...`	Direct `const handleClick = ...`
Modern Preferred	❌ Less preferred	✅ Yes (default today)

🎁 Bonus Tip:

👉 Combine hooks like `useState`, `useEffect`, `useContext`, and `useReducer` in function components to **replace everything class components offer** and even more!

🧩 Part 2: React Class vs Function Components – Full Flow & Advanced Patterns

---

🔢 Managing Multiple State Variables

🏗 Class-Based:

```
class Profile extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Darshan",
      age: 22,
      city: "Surat",
    };
  }

  render() {
    return (
      <div>
        <p>👤 Name: {this.state.name}</p>
        <p>🎂 Age: {this.state.age}</p>
        <p>📍 City: {this.state.city}</p>
      </div>
    );
  }
}
```

⚙️ Function-Based (with `useState` x multiple):

```
import { useState } from "react";

const Profile = () => {
  const [name, setName] = useState("Darshan");
  const [age, setAge] = useState(22);
  const [city, setCity] = useState("Surat");

  return (
    <div>
      <p>👤 Name: {name}</p>
      <p>🎂 Age: {age}</p>
      <p>📍 City: {city}</p>
    </div>
    );
  };
```

☑ Or, combine into one object state:

```
const [user, setUser] = useState({
  name: "Darshan",
  age: 22,
  city: "Surat",
});

const updateCity = () => {
  setUser((prev) => ({ ...prev, city: "Ahmedabad" }));
};
```

🛍 Destructuring `props` and `state`

💼 In Class-Based:

```
class User extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Darshan",
      age: 22,
    };
  }

  render() {
    const { name, age } = this.state;
    const { email } = this.props;

    return (
      <div>
        <p>👤 Name: {name}</p>
        <p>🎂 Age: {age}</p>
        <p>📧 Email: {email}</p>
      </div>
    );
  }
}
```

🧩 In Function-Based:

```
const User = ({ email }) => {
  const [user, setUser] = useState({ name: "Darshan", age: 22 });
  const { name, age } = user;

  return (
    <div>
      <p>👤 Name: {name}</p>
      <p>🎂 Age: {age}</p>
      <p>📧 Email: {email}</p>
    </div>
  );
};
```

🔄 Full Class Component Lifecycle 🧭

> Class components have **3 major phases**:
>
> 1. **Mounting**
> 2. **Updating**
> 3. **Unmounting**

📦 1. Mounting Phase

Method	Purpose
`constructor()`	Initialize state & bind methods
`render()`	Return JSX
`componentDidMount()`	Called once after initial render (API calls, DOM ops)

🔁 2. Updating Phase

Method	Purpose
`render()`	Called again on state/prop change
`componentDidUpdate()`	Called after render due to update

❌ 3. Unmounting Phase

Method	Purpose
`componentWillUnmount()`	Cleanup tasks like event listeners, timers, etc

🌟 Full Class Example with All Lifecycle Methods

```
class LifeCycleDemo extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
    console.log("📦 constructor");
  }

  componentDidMount() {
    console.log("🚀 componentDidMount");
  }

  componentDidUpdate(prevProps, prevState) {
    console.log("🔁 componentDidUpdate");
    console.log("Previous State:", prevState.count);
  }

  componentWillUnmount() {
    console.log("🧹 componentWillUnmount");
  }

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

  render() {
    console.log("🎨 render");
    return (
      <div>
        <h1>🔢 Count: {this.state.count}</h1>
        <button onClick={this.increment}>➕ Increment</button>
      </div>
    );
  }
}
```

🔁 Equivalent Function Component with Hooks

```
import React, { useState, useEffect } from "react";

const LifeCycleDemo = () => {
  const [count, setCount] = useState(0);

  // 🚀 ComponentDidMount + ComponentDidUpdate
  useEffect(() => {
    console.log("🔁 useEffect - mount/update");
    return () => {
      console.log("🧹 Cleanup - componentWillUnmount");
    };
  }, [count]); // dependency array

  return (
    <div>
```

```
      <h1>🧮 Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>➕ Increment</button>
    </div>
  );
};
```

🧼 Summary Cheat Sheet: Lifecycle Comparison

🔄 Lifecycle Phase	Class Component	Function Component
Mounting	`constructor()` → `render()` → `componentDidMount()`	`useEffect(() => {}, [])`
Updating	`render()` → `componentDidUpdate()`	`useEffect(..., [deps])`
Unmounting	`componentWillUnmount()`	`return () => {...}` in `useEffect`

🚫 Common Mistakes (BAD practices)

❌ Bad Practice	✅ Fix
Updating state directly `this.state.name = ...`	Always use `this.setState()` or `setState()` hook
Not using cleanup in `useEffect`	Add return function in `useEffect` for cleanup
Using too many `useState()` separately	Consider combining into one object if related
Forgetting to pass `props` to `super()`	Always call `super(props)` in constructor

---

✅ Best Practices

- ✅ Use function components with hooks for all **new code**.
- ✅ Use `useEffect` to handle all side-effects, API calls, subscriptions.
- ✅ Keep components **pure and small**.
- ✅ Group related state in one `useState()` object or use `useReducer()` for complex cases.
- ✅ Always return cleanup in `useEffect()` when needed (e.g., `setInterval`, event listeners).

📦 Scenario: Parent-Child Class Components

```
class Child extends React.Component {
  constructor(props) {
    super(props);
    console.log("👶 Child: constructor");
  }
  componentDidMount() {
    console.log("👶 Child: componentDidMount");
  }

  render() {
    console.log("👶 Child: render");
    return <h3>I am the Child</h3>;
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    console.log("👨‍👧 Parent: constructor");
  }

  componentDidMount() {
    console.log("👨‍👧 Parent: componentDidMount");
  }

  render() {
    console.log("👨‍👧 Parent: render");
    return (
      <div>
        <h2>I am the Parent</h2>
        <Child />
      </div>
    );
  }
}
```

🧠 Full Lifecycle Flow (on initial mount)

🔄 MOUNTING Phase Order:

```
1️⃣ Parent: constructor
2️⃣ Parent: render
3️⃣ Child: constructor
4️⃣ Child: render
5️⃣ Child: componentDidMount
6️⃣ Parent: componentDidMount
```

> ✅ **Render order always goes parent → child** ✅ **componentDidMount runs after both render phases finish**

🔁 If Parent Re-renders (due to state/props update)

```
this.setState({}); // inside Parent
```

🔁 UPDATE Flow:

```
1️⃣ Parent: render
2️⃣ Child: render (if affected or re-rendered)
3️⃣ Child: componentDidUpdate (if state or props changed)
4️⃣ Parent: componentDidUpdate
```

> 🚨 Note: Child will only re-render if:
>
> - It received new props OR
> - Its state changed OR
> - You force update the parent which contains the child

❌ UNMOUNTING Flow (if parent removes child):

```
{this.state.showChild && <Child />}
```

When `showChild` becomes `false`:

```
1️⃣ Child: componentWillUnmount
```

📋 Full Lifecycle Breakdown Table

Phase	Component	Method	Notes
Mounting	Parent	`constructor`	Initialize Parent
Mounting	Parent	`render`	Starts rendering JSX
Mounting	Child	`constructor`	Child is constructed
Mounting	Child	`render`	Child JSX is rendered
Mounting	Child	`componentDidMount`	Runs after child rendered
Mounting	Parent	`componentDidMount`	Runs after child mounted
Updating	Parent	`render`	Rerenders when parent state changes
Updating	Child	`render` (if needed)	Child re-renders if affected
Updating	Child	`componentDidUpdate`	Child update logic

Phase	Component	Method	Notes
Updating	Parent	`componentDidUpdate`	Parent update logic
Unmounting	Child	`componentWillUnmount`	When removed from DOM

🧪 Bonus: Test It Live

```
class Child extends React.Component {
  constructor(props) {
    super(props);
    console.log("Child: constructor");
  }

  componentDidMount() {
    console.log("Child: componentDidMount");
  }

  componentDidUpdate() {
    console.log("Child: componentDidUpdate");
  }

  componentWillUnmount() {
    console.log("Child: componentWillUnmount");
  }

  render() {
    console.log("Child: render");
    return <h3>👶 I am the Child</h3>;
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      showChild: true,
    };
    console.log("Parent: constructor");
  }

  componentDidMount() {
    console.log("Parent: componentDidMount");
  }

  componentDidUpdate() {
    console.log("Parent: componentDidUpdate");
  }

  toggleChild = () => {
    this.setState((prev) => ({ showChild: !prev.showChild }));
  };
  render() {
    console.log("Parent: render");
    return (
      <div>
        <h2>👩‍👧 I am the Parent</h2>
        <button onClick={this.toggleChild}>Toggle Child</button>
        {this.state.showChild && <Child />}
      </div>
    );
  }
}
```

🧠 TL;DR Lifecycle Flow

Event Type	Order
Initial Mount	Parent constructor → Parent render → Child constructor → Child render → Child CDM → Parent CDM
Update State	Parent render → Child render → Child CDU → Parent CDU
Remove Child	Child CWU

⚛️ React Lifecycle Flow with Two Children (Class-Based)

📦 Scenario

We have the following structure:

```
App (Parent)
├── ChildA
└── ChildB
```

✅ Code Setup:

```
// 🔹 ChildA.js
class ChildA extends React.Component {
  constructor(props) {
    super(props);
    console.log("👦 ChildA: constructor");
  }

  componentDidMount() {
    console.log("👦 ChildA: componentDidMount");
  }

  render() {
    console.log("👦 ChildA: render");
    return <div>Child A</div>;
  }
}

// ◇ ChildB.js
class ChildB extends React.Component {
  constructor(props) {
    super(props);
    console.log("👧 ChildB: constructor");
  }

  componentDidMount() {
    console.log("👧 ChildB: componentDidMount");
  }

  render() {
    console.log("👧 ChildB: render");
    return <div>Child B</div>;
  }
}

// ◇ App.js (Parent)
class App extends React.Component {
  constructor(props) {
    super(props);
    console.log("👨 Parent: constructor");
  }

  componentDidMount() {
    console.log("👨 Parent: componentDidMount");
  }

  render() {
    console.log("👨 Parent: render");
    return (
      <div>
        <h1>Parent</h1>
        <ChildA />
        <ChildB />
      </div>
    );
  }
}
```

🧠 Initial Render: Lifecycle Order (Mounting)

```
1️⃣ 👨 Parent: constructor
2️⃣ 👨 Parent: render
3️⃣ 👦 ChildA: constructor
4️⃣ 👦 ChildA: render
5️⃣ 👧 ChildB: constructor
6️⃣ 👧 ChildB: render
7️⃣ 👦 ChildA: componentDidMount
8️⃣ 👧 ChildB: componentDidMount
9️⃣ 👨 Parent: componentDidMount
```

> ✅ Children always **mount after** parent renders ✅ `componentDidMount()` runs **bottom-up** (child first, parent last)

🔁 Update Flow (e.g., Parent `setState()`)

If `App` does `this.setState()`:

```
1️⃣ 👨 Parent: render
2️⃣ 👦 ChildA: render (if affected)
3️⃣ 👧 ChildB: render (if affected)
4️⃣ 👦 ChildA: componentDidUpdate
5️⃣ 👧 ChildB: componentDidUpdate
6️⃣ 👨 Parent: componentDidUpdate
```

> 🔄 Re-render order: **top-down** 🧠 Update lifecycle: **bottom-up**

❌ If One Child is Removed

Let's say `ChildB` is conditionally rendered like:

```
{this.state.showB && <ChildB />}
```

Toggling `showB` to `false`:

```
👧 ChildB: componentWillUnmount
```

🔄 Lifecycle Summary Table

Phase	Order

Phase	Order
Mounting	Parent constructor → render → ChildA constructor → render → ChildB constructor → render → componentDidMount (ChildA → ChildB → Parent)
Updating	Parent render → ChildA render → ChildB render → componentDidUpdate (ChildA → ChildB → Parent)
Unmount	Only affected child's `componentWillUnmount()` runs

🧪 Visual Timeline (Mounting)

```
👨 Parent
 ├── constructor
 └── render
      ├── 👦 ChildA
      │     ├── constructor
      │     └── render
      └── 👧 ChildB
            ├── constructor
            └── render
👦 ChildA: componentDidMount
👧 ChildB: componentDidMount
👨 Parent: componentDidMount
```
