🎭 JSX Conditional Rendering in React

In React, conditional rendering means **displaying different UI elements based on logic** like a user's login state, loading data, or errors.

☑️ 1. Using `if...else` outside JSX 🧠

📌 Use this **outside the** `return()` when you have more than 1 condition and need full control.

```
if (isLoggedIn) {
  return <Dashboard />;
}
return <Login />;
```

📝 **Explanation**: Just like plain JavaScript! Ideal for **clear logic** and **early returns**.

❓ 2. Ternary Operator (`condition ? true : false`) ⚖️

📌 Inline rendering. Choose between **two components** or UI fragments.

```
{isLoggedIn ? <Dashboard /> : <Login />}
```

📝 **Explanation**: If `isLoggedIn` is `true`, it renders `<Dashboard>`. Else, it renders `<Login>`.

🧠 Tip: Nesting ternaries can get messy. Avoid this:

```
{isLoggedIn ? (isAdmin ? <AdminPanel /> : <UserPanel />) : <Login />}
```

⚡ 3. Logical AND (`&&`) for single condition ☑️

📌 Render something **only if a condition is** `true`.

```
{hasError && <ErrorMessage />}
```

📝 **Explanation**: If `hasError` is `true`, show the error. If `false`, React ignores the expression.

🧠 Best for optional elements.

⛔ 4. Logical OR (`||`) for default fallback 🪄

📌 Show fallback UI when a variable is falsy (like `""`, `null`, or `undefined`).

```
{username || "Guest"}
```

📝 **Explanation**: If `username` has a value, it shows it. If it's `""` or `null`, it shows `"Guest"`.

🌀 5. IIFE (Immediately Invoked Function Expression) 🧩

📌 For **complex logic** inside JSX (multiple `if`, `switch`, etc.)

```
{(() => {
  if (loading) return <Spinner />;
  if (error) return <Error />;
  return <Content />;
})()}
```

📝 **Explanation**: Runs a small inline function that returns UI based on multiple conditions. ☑️ Keeps your JSX readable for **multi-condition logic**.

🔀 6. Switch Case Pattern (via function) 🎯

📌 Cleaner alternative to nested if-else or ternary chains.

```
const renderPage = () => {
  switch (page) {
    case "home":
      return <Home />;
    case "about":
      return <About />;
    default:
      return <NotFound />;
  }
};

return <div>{renderPage()}</div>;
```

📝 **Explanation**: Define a render function outside `return()`, and invoke it in JSX.

🛡️🔐 7. Optional Chaining with AND (`user?.prop && <Component />`)

📌 Safely access nested values **without throwing errors**.

```
{user?.isAdmin && <AdminPanel />}
```

📝 **Explanation**: If `user` exists AND `user.isAdmin` is `true`, show the `<AdminPanel />`.

☑️ Prevents "Cannot read property of undefined" errors.

🎨 Bonus: Styling Short Conditional UI

☑️ You can also use fragments (`<> </>`) or wrap JSX conditionally.

```
{isLoggedIn ? (
  <>
    <Navbar />
    <Dashboard />
  </>
) : (
  <Login />
)}
```

🧠 Pro Tips

- ☑️ Keep JSX readable — use helper functions for complex conditions.
- ❌ Avoid deeply nested ternaries — hard to debug and maintain.
- 🔁 If the same condition is reused, store it in a variable for clarity.

⚛️ React Conditional Rendering + Shimmer UI Guide 🔄✨

🛠️ Use Case: Show Loader While Data Is Fetching

🔍 Scenario:

You're building a restaurant listing app. Initially, the restaurant list is empty because data is fetched **asynchronously** from an API.

🌀 During this time, we want to show a **Shimmer UI** (like a skeleton loader) ☑️ Once the data arrives, we display the **actual list of restaurants**.

🔄 Component Lifecycle Flow:

```
// 🏁 Initial render:
const [listOfRestaurants, setListOfRestaurants] = useState([]);

// 📦 listOfRestaurants = [] ➡ Show Shimmer UI
// ✅ After API success ➡ setListOfRestaurants(data) ➡ Show actual UI
```

💡 Pseudo Code Logic:

```
if (listOfRestaurants.length === 0) {
  // ⏳ Still loading...
  return <Shimmer />;
} else {
  // ✅ Data loaded
  return <RestaurantList data={listOfRestaurants} />;
}
```

⚠️ Common Error: `listOfRestaurants` is `undefined`

```
{listOfRestaurants.length === 0 && <Shimmer />}
```

🧪 ❗ Will throw:

> ❌ `Cannot read properties of undefined (reading 'length')`

✅ Solution 1: Optional Chaining

```
{listOfRestaurants?.length === 0 && <Shimmer />}
```

✅ If `listOfRestaurants` is `undefined`, `?.length` will **safely return `undefined`**, preventing crash.

✅ Solution 2: Early Return Pattern

```
if (!listOfRestaurants) return <Shimmer />;
```

✅ If `listOfRestaurants` is falsy (`undefined` or `null`), return loader early — before JSX renders.

💎 Final Pattern — Clean & Safe:

```
const Body = () => {
  const [listOfRestaurants, setListOfRestaurants] = useState(null);

  useEffect(() => {
    fetchRestaurants();
  }, []);

  const fetchRestaurants = async () => {
    const data = await fetch(API_URL);
    const json = await data.json();
    setListOfRestaurants(json?.data?.restaurants || []);
  };

  if (!listOfRestaurants) return <Shimmer />;

  return (
    <div className="restaurant-list">
      {listOfRestaurants.map((res) => (
        <RestaurantCard key={res.id} {...res} />
      ))}
    </div>
  );
};
```

💥 BONUS: Empty Filter Result

```
{filteredRestaurants.length === 0 ? (
  <h3>😞 No restaurants match your filter</h3>
) : (
  <RestaurantList data={filteredRestaurants} />
)}
```

🚨 React Anti-Patterns To Avoid ❌

🚫 1. Creating Components Inside Components

```
const Parent = () => {
  const Child = () => <p>I'm inside! ❌</p>;
  return <Child />;
};
```

❌ BAD: Every render recreates `Child`. ☑ DO THIS:

```
// Outside
const Child = () => <p>I'm outside ☑</p>;
```

🚫 2. `useState` Inside `if/else`

```
if (condition) {
  const [name, setName] = useState(""); // ❌
}
```

❌ React loses track of hook execution. ☑ Hooks must run in **top-level scope** of component.

🚫 3. `useState` in Loops

```
for (let i = 0; i < 5; i++) {
  const [val, setVal] = useState(""); // ❌
}
```

❌ Creates multiple unexpected states.

🚫 4. `useState` Outside Component

```
const [count, setCount] = useState(0); // ❌ Invalid Hook Call
```

Hooks MUST be inside functional components or custom hooks.

🧠 Important React Notes

🔧 **Concept**	☑ **Best Practice**
Loader while fetching	Use `Shimmer` or spinner
Avoid crash on `undefined`	Use `?.` or early return
Multiple side-effects	You can use multiple `useEffect`s
Store images	Use `/assets/` folder

🔧 Concept	☑ Best Practice
Hook location	Always top-level, inside component only