

What is a Component in React?

A **component** in React is a reusable piece of UI that can be used multiple times in an application. Components help break down the UI into **small, manageable pieces**, making the code **clean and reusable**.

◆ Types of Components in React

1 Functional Components (Recommended)

- Simple JavaScript functions that return JSX (UI elements).
- Use **React Hooks** (like `useState`, `useEffect`) for managing state.

✓ Example:

```
import React from "react";

function Greeting() {
  return <h1>Hello, Welcome to React!</h1>;
}

export default Greeting;
```

2 Class Components (Older Method)

- Uses ES6 classes.
- Manages state using `this.state`.
- Uses `this.props` to access properties.

✓ Example:

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

class Greeting extends Component {
  render() {
    return <h1>Hello, Welcome to React!</h1>;
  }
}
```

```
}  
}
```

```
export default Greeting;
```

 **Note:** Functional components are preferred because they are simpler and more efficient.

◆ Component Structure

A component typically has:

1. **Imports** → Import React and other dependencies.
 2. **JSX (UI code)** → The HTML-like syntax inside `return()`.
 3. **Props** → Data passed from parent to child.
 4. **State (if needed)** → Stores dynamic values.
-

◆ How to Use a Component?

Once a component is created, it can be used inside another component like this:

✓ **Example:**

```
import Greeting from "../Greeting";
```

```
function App() {  
  return (  
    <div>  
      <Greeting />  
    </div>  
  );  
}
```

```
export default App;
```

Why Use Components?

- ✓ **Reusability** → Write once, use anywhere.
- ✓ **Maintainability** → Clean and modular code.
- ✓ **Faster Development** → Small components are easier to test and debug.

How to Use a Component in React (Same File & Different File)

✓ 1. Using a Component in the Same File

If the component is in the same file, simply **define it and use it** inside another component.

Example:

```
import React from "react";

function Greeting() {
  return <h1>Hello, Welcome to React!</h1>;
}

function App() {
  return (
    <div>
      <Greeting />  {/* Using Greeting component inside App */}
    </div>
  );
}

export default App;
```

 Here, **Greeting** is inside the same file as **App**.

✓ 2. Using a Component from a Different File (Import & Export)

To use a component from another file, you need to **export** it and then **import** it where needed.

Step 1: Create a Component File (e.g., **Greeting.jsx**)

```
import React from "react";

function Greeting() {
  return <h1>Hello, Welcome to React!</h1>;
}

export default Greeting; // Exporting component
```

Step 2: Import and Use in Another File (e.g., **App.jsx**)

```
import React from "react";
import Greeting from "../Greeting"; // Importing the Greeting component

function App() {
  return (
    <div>
      <Greeting /> {/* Using the imported component */}
    </div>
  );
}

export default App;
```

📌 Here, **Greeting.jsx** is in the same folder as **App.jsx**.

✅ 3. Named Export & Import (Alternative Method)

Instead of `export default`, you can use **named exports**.

Step 1: Named Export in **Greeting.jsx**

```
import React from "react";

export function Greeting() { // Named export
  return <h1>Hello, Welcome to React!</h1>;
}
```

Step 2: Named Import in **App.jsx**

```
import React from "react";
import { Greeting } from "../Greeting"; // Named import (Curly Braces Needed)

function App() {
  return (
    <div>
      <Greeting />
    </div>
  );
}

export default App;
```

Difference Between Default & Named Export:

- **Default Export** → No curly braces (`export default ComponentName`).
- **Named Export** → Requires curly braces (`export { ComponentName }`).

Summary

Export Type	Import Syntax
Default Export	<code>import Component from "../Component";</code>
Named Export	<code>import { Component } from "../Component";</code>

Would you like a deeper explanation of `default` vs. `named` exports? 😊

✓ Rules for JSX & React Components

When writing JSX and React components, there are some important rules to follow for clean and error-free code.

JSX Rules (React's Syntax for UI)

① Return a Single Parent Element

- JSX must return **one parent element** (use `<div>` or `<>` for fragments).

✓ **Correct:**

```
return (  
  <div>  
    <h1>Hello</h1>  
    <p>Welcome to React</p>  
  </div>  
);
```

✗ **Wrong:**

```
return (  
  <h1>Hello</h1>  
  <p>Welcome to React</p>  
); // ✗ Error: JSX must have one parent
```

② Use `className` Instead of `class`

- In JSX, `class` is a reserved JavaScript word, so use `className`.

✓ **Correct:**

```
return <h1 className="title">Hello</h1>;
```

✗ **Wrong:**

```
return <h1 class="title">Hello</h1>; // ✗ Error
```

③ Close All Tags Properly

- JSX requires self-closing tags (``, `
`, `<input />`).

✓ **Correct:**

```
return ;
```

✗ **Wrong:**

```
return ; // ✗ Error
```

④ Use Curly Braces `{}` for JavaScript Inside JSX

- To insert variables or expressions inside JSX, use `{}`.

✓ **Correct:**

```
const name = "John";  
return <h1>Hello, {name}!</h1>;
```

✗ **Wrong:**

```
const name = "John";  
return <h1>Hello, name!</h1>; // ✗ Error: `name` is treated as text
```

⑤ Boolean Attributes Don't Need Values

- In JSX, you don't need to assign `true` to boolean attributes like `checked`, `disabled`, `required`.

✓ **Correct:**

```
return <input type="checkbox" checked />;
```

✗ **Wrong:**

```
return <input type="checkbox" checked="true" />; // ✗ Unnecessary
```

React Component Rules

① Component Names Must Start with a Capital Letter

- React treats lowercase names as **HTML elements**, not components.

✅ **Correct:**

```
function Header() {  
  return <h1>Welcome</h1>;  
}
```

❌ **Wrong:**

```
function header() { // ❌ React will not recognize this as a  
  component  
  return <h1>Welcome</h1>;  
}
```

② Components Must Be Exported & Imported Correctly

✅ **Default Export & Import:**

```
export default function Header() {  
  return <h1>Welcome</h1>;  
}
```

```
import Header from "../Header";
```

✅ **Named Export & Import:**

```
export function Header() {  
  return <h1>Welcome</h1>;  
}
```

```
import { Header } from "../Header";
```

③ Props Should Be Passed Correctly


- Props allow components to receive data dynamically.

✅ **Correct:**


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

`<Greeting name="Alice" />` //  Passes "Alice" as a prop

 **Wrong:**

`<Greeting name={Alice} />` //  Error: Alice is not a string or variable

4 Use `useState` for Dynamic Changes


- Components should **not modify variables directly**; use `useState` instead.

 **Correct:**


```
import { useState } from "react";  
  
function Counter() {  
  const [count, setCount] = useState(0);  
  
  return <button onClick={() => setCount(count + 1)}>Count:  
    {count}</button>;  
}
```

 **Wrong:**

```
let count = 0;  
return <button onClick={() => count++}>Count: {count}</button>; //
```

 Won't update UI

Summary of JSX & Component Rules

Rule	Example
 Use one parent element	<code><div>...</div></code>

✓ Use <code>className</code> instead of <code>class</code>	<code><h1 className="title">Hello</h1></code>
✓ Close all tags properly	<code></code>
✓ Use <code>{ }</code> for dynamic values	<code><h1>Hello, {name}!</h1></code>
✓ Component names start with a capital letter	<code>function Header() { ... }</code>
✓ Pass props correctly	<code><Greeting name="John" /></code>
✓ Use <code>useState</code> for changes	<code>const [count, setCount] = useState(0);</code>

Would you like a **practical example** of these rules? 😊

💀💀 Props :-

Yes! ✅ **Props in React are just like parameters in functions.**

- They **allow components to receive data** from their parent component.
 - They **help make components reusable** by passing different values.
 - They **are immutable** (cannot be changed inside the component).
-

✅ **Props = Function Parameters**

✅ **Example:** Regular function with parameters:

```
function greet(name) {  
  return `Hello, ${name}!`;  
}
```

```
console.log(greet("Alice")); // Output: Hello, Alice!
```

✅ **Same concept in React with props:**

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

// Usage:

```
<Greeting name="Alice" /> // Output: Hello, Alice!  
<Greeting name="Bob" /> // Output: Hello, Bob!
```

✅ **Props + Conditional Rendering (Dynamic Data)**

When you need to show **different UI based on conditions**, props help pass dynamic values.

✅ **Example:**

```
function Message(props) {  
  return (  
    <div>
```

```
    {props.isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please log  
in</h1>}  
  </div>  
);  
}
```

// Usage:

```
<Message isLoggedIn={true} /> // Shows: Welcome back!  
<Message isLoggedIn={false} /> // Shows: Please log in
```

Summary

- **Props = Function parameters** → Used to pass dynamic values.
- **Props make components reusable** → One component can work with multiple values.
- **Conditional rendering with props** → Change UI dynamically based on conditions.

Would you like an example with multiple props? 😊

Conditional Rendering in React

Conditional Rendering means displaying different UI elements **based on conditions**. It works just like **if-else statements** in JavaScript but inside JSX.

1. Conditional Rendering Using **if-else**

We can use a simple **if-else** statement inside a function to render different content.

```
function WelcomeMessage(props) {
  if (props.isLoggedIn) {
    return <h1>Welcome back, {props.username}!</h1>;
  } else {
    return <h1>Please log in.</h1>;
  }
}

function App() {
  return (
    <div>
      <WelcomeMessage isLoggedIn={true} username="Alice" />
      <WelcomeMessage isLoggedIn={false} />
    </div>
  );
}

export default App;
```

Explanation:

- If **isLoggedIn** is **true**, it displays "Welcome back, Alice!".
 - If **isLoggedIn** is **false**, it displays "Please log in.".
 - Props (**isLoggedIn**, **username**) **pass dynamic data** to change the output.
-

✓ 2. Conditional Rendering Using the Ternary Operator (? :)

A shorter way to handle conditions inside JSX.

```
function WelcomeMessage({ isLoggedIn, username }) {
  return (
    <h1>{isLoggedIn ? `Welcome back, ${username}!` : "Please log
in."}</h1>
  );
}

function App() {
  return (
    <div>
      <WelcomeMessage isLoggedIn={true} username="Alice" />
      <WelcomeMessage isLoggedIn={false} />
    </div>
  );
}

export default App;
```

📌 Here,

- If `isLoggedIn` is `true`, it shows "Welcome back, Alice!".
- If `false`, it shows "Please log in.".
- This method **reduces lines of code** while keeping readability.

✓ 3. Conditional Rendering Using `&&` (Short-Circuit Evaluation)

If you **only need to show something** when a condition is `true`, use `&&`.

```
function Notification({ unreadMessages }) {
  return (
    <div>
```

```

        <h1>Dashboard</h1>
        {unreadMessages > 0 && <p>You have {unreadMessages} unread
messages!</p>}
      </div>
    );
  }

function App() {
  return (
    <div>
      <Notification unreadMessages={5} />
      <Notification unreadMessages={0} />
    </div>
  );
}

export default App;

```

📌 Here,

- If `unreadMessages > 0`, it shows "You have 5 unread messages!".
- If `unreadMessages = 0`, it **hides** the message (nothing is displayed).



How Props & Conditional Rendering Work Together

Props **pass dynamic values** to a component, and **conditional rendering** decides what to show based on those values.

✅ Example: Combining Props & Conditional Rendering

```

function UserStatus({ name, isOnline }) {
  return (
    <h1>
      {name} is {isOnline ? "Online ✅" : "Offline ❌"}
    </h1>
  );
}

function App() {

```



```

return (
  <div>
    <UserStatus name="Alice" isOnline={true} />
    <UserStatus name="Bob" isOnline={false} />
  </div>
);
}

export default App;

```

Explanation:

- `UserStatus` gets `name` and `isOnline` as **props**.
- If `isOnline` is **true**, it shows "Alice is Online .
- If `isOnline` is **false**, it shows "Bob is Offline .

Final Summary

Concept	Example	Usage
if-else inside function	<pre>if (props.isLoggedIn) { return <h1>Welcome</h1>; }</pre>	More readable, but longer
Ternary Operator (?:)	<pre>{props.isLoggedIn ? <h1>Welcome</h1> : <h1>Please Log In</h1>}</pre>	Shorter & more readable inside JSX
Short-Circuit (&&)	<pre>{unreadMessages > 0 && <p>You have {unreadMessages} messages!</p>}</pre>	Hides elements when condition is false
Props + Conditional Rendering	<pre><UserStatus isOnline={true} /></pre>	Pass dynamic values and render different UI


In React, **array rendering** refers to the process of displaying lists of data by iterating over an array and rendering UI elements dynamically. This is commonly done using the `.map()` function.

♦ Basic Example: Rendering an Array of Strings

If you have an array of names and want to display them as a list:

```
const names = ["Alice", "Bob", "Charlie"];

function NameList() {
  return (
    <ul>
      {names.map((name, index) => (
        <li key={index}>{name}</li>
      ))}
    </ul>
  );
}
```

 **Key Attribute:** Each item in the list should have a unique `key` to help React efficiently update and render components.

♦ Rendering an Array of Objects

If the data is more complex, like an array of objects:

```
const users = [
  { id: 1, name: "Alice", age: 25 },
  { id: 2, name: "Bob", age: 30 },
  { id: 3, name: "Charlie", age: 22 }
];

function UserList() {
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>
          {user.name} - Age: {user.age}
        </li>
      ))}
    </ul>
  );
}
```

```
    ))}  
  </ul>  
);  
}
```

◆ Conditional Rendering within Arrays

You can filter items before rendering:

```
function AdultsList() {  
  return (  
    <ul>  
      {users  
        .filter(user => user.age >= 25)  
        .map(user => (  
          <li key={user.id}>{user.name} - {user.age}</li>  
        ))}  
    </ul>  
  );  
}
```

◆ Using Components for Array Rendering

For better modularity, render each item using a separate component:

```
function UserItem({ user }) {  
  return <li>{user.name} - {user.age}</li>;  
}  
  
function UserList() {  
  return (  
    <ul>  
      {users.map(user => (  
        <UserItem key={user.id} user={user} />  
      ))}  
    </ul>  
  );  
}
```

◆ Handling Empty Arrays

If the array is empty, you can show a fallback message:

```
function UserList() {  
  return (  
    <ul>  
      {users.length > 0 ? (  
        users.map(user => <li key={user.id}>{user.name}</li>)  
      ) : (  
        <p>No users available</p>  
      )}  
    </ul>  
  );  
}
```

🏆 Key Takeaways:

- ✓ Use `.map()` to iterate over arrays.
- ✓ Always provide a **unique key** for each item.
- ✓ Use separate components for complex lists.
- ✓ Filter or conditionally render items when needed.

Want to see a specific example? 😊