**Conditional Rendering in React**

In React, conditional rendering allows you to display different content based on certain conditions. Here’s a detailed explanation of the concepts and examples provided:

**Key Topics to Learn**

1. **How to return different JSX depending on a condition.**
2. **How to conditionally include or exclude JSX.**
3. **Common syntax shortcuts for conditional rendering in React.**

**1. Conditionally Returning JSX**

**Example Setup**

You have a PackingList component that renders multiple Item components. Each item has a name and an isPacked prop, which determines whether it is packed or not.

**Goal**

Add a checkmark (✅) next to items that are packed (isPacked=true).

**Using if/else for Conditional Rendering**

* **Approach**: Check the condition using an if/else statement and return different JSX based on the condition.

javascript

CopyEdit

function Item({ name, isPacked }) {

if (isPacked) {

return <li className="item">{name} ✅</li>;

}

return <li className="item">{name}</li>;

}

* **Explanation**:
  + If isPacked is true, the item’s name is displayed with a ✅.
  + Otherwise, it simply displays the name.

**2. Conditionally Returning Nothing**

**Use Case**

If an item is packed, you might want to hide it completely from the list.

**Solution: Return null**

* In React, returning null means the component will render nothing.

javascript

CopyEdit

function Item({ name, isPacked }) {

if (isPacked) {

return null; // Does not render anything for packed items

}

return <li className="item">{name}</li>;

}

* **Explanation**:
  + If isPacked is true, the Item component returns null, excluding it from the DOM.
  + If isPacked is false, it renders the name as a list item.

**3. Conditionally Including or Excluding Components**

**Approach**

Instead of returning null from the Item component, you can handle the conditional logic in the parent component (PackingList).

javascript

CopyEdit

function PackingList() {

return (

<section>

<h1>Sally Ride's Packing List</h1>

<ul>

{/\* Only include Item components where isPacked is false \*/}

<Item isPacked={false} name="Space suit" />

<Item isPacked={false} name="Photo of Tam" />

</ul>

</section>

);

}

* **Explanation**:
  + In this example, you manually decide which items to render based on their isPacked status before passing them to the Item component.

**4. Common Syntax for Conditional Rendering**

React supports several JavaScript syntax options for conditions:

**a. if/else**

* Most straightforward way to handle conditions.
* Example:

javascript

CopyEdit

function Item({ name, isPacked }) {

if (isPacked) {

return <li>{name} ✅</li>;

} else {

return <li>{name}</li>;

}

}

**b. Ternary Operator (? :)**

* Useful for inline conditional rendering.
* Example:

javascript

CopyEdit

function Item({ name, isPacked }) {

return (

<li>{name} {isPacked ? '✅' : ''}</li>

);

}

**c. Logical && Operator**

* If the condition is true, the JSX after && is rendered.
* Example:

javascript

CopyEdit

function Item({ name, isPacked }) {

return (

<li>

{name}

{isPacked && ' ✅'}

</li>

);

}

**5. Returning JSX Trees Based on Conditions**

**Example: Adding Checkmarks**

In this example, the JSX returned changes based on the isPacked condition.

javascript

CopyEdit

if (isPacked) {

return <li className="item">{name} ✅</li>;

}

return <li className="item">{name}</li>;

* **Explanation**:
  + If the isPacked prop is true, the item is rendered with a checkmark.
  + Otherwise, the plain item name is rendered.

**6. Practical Example: Full Packing List**

Here’s how the full example works:

javascript

CopyEdit

function Item({ name, isPacked }) {

if (isPacked) {

return <li className="item">{name} ✅</li>;

}

return <li className="item">{name}</li>;

}

export default function PackingList() {

return (

<section>

<h1>Sally Ride's Packing List</h1>

<ul>

<Item isPacked={true} name="Space suit" />

<Item isPacked={true} name="Helmet with a golden leaf" />

<Item isPacked={false} name="Photo of Tam" />

</ul>

</section>

);

}

**Best Practices**

1. **Keep It Simple**:
   * Use if/else or the ternary operator for clarity when the logic is simple.
2. **Use null Sparingly**:
   * Avoid surprising future developers by not rendering components. Use this only when necessary.
3. **Conditionally Include Components in Parent**:
   * Move conditional logic to the parent component for better readability and separation of concerns.
4. **Choose the Right Syntax**:
   * Use if/else for complex conditions and ternary or && for concise logic.

**Default Props in React**

**Default props** in React allow you to define default values for props in a component. This is useful when a parent component does not provide a specific prop, ensuring the component has a fallback value to work with.

**Why Use Default Props?**

1. **Avoid Undefined Props**: Prevent errors caused by missing or undefined props.
2. **Ensure Consistency**: Provide consistent behavior when props are not explicitly passed.
3. **Simplify Components**: Reduce the need for conditional logic to handle missing props.

**Defining Default Props**

React offers two common ways to define default props:

**1. Using the defaultProps Property (Class or Functional Components)**

javascript

CopyEdit

function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

Greeting.defaultProps = {

name: "Guest", // Default value for 'name'

};

export default Greeting;

* If the parent component does not pass the name prop, the Greeting component will use the default value "Guest".

**Usage in Parent Component:**

javascript

CopyEdit

export default function App() {

return (

<div>

<Greeting /> {/\* Will display: "Hello, Guest!" \*/}

<Greeting name="John" /> {/\* Will display: "Hello, John!" \*/}

</div>

);

}

**2. Using Default Parameter Syntax in Function Arguments**

You can set default values directly in the function argument list using ES6 syntax:

javascript

CopyEdit

function Greeting({ name = "Guest" }) {

return <h1>Hello, {name}!</h1>;

}

export default Greeting;

* This approach is simpler and works well for functional components without requiring the defaultProps property.

**Using Default Props in Class Components**

For class components, you define default props using the defaultProps property:

javascript

CopyEdit

class Greeting extends React.Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

Greeting.defaultProps = {

name: "Guest",

};

export default Greeting;

**Default Props with Destructuring**

You can combine default props with object destructuring for a cleaner implementation:

javascript

CopyEdit

function UserCard({ username = "Anonymous", age = "N/A" }) {

return (

<div>

<p>Username: {username}</p>

<p>Age: {age}</p>

</div>

);

}

export default UserCard;

**Dynamic Example with Default Props**

javascript

CopyEdit

function Button({ label, onClick }) {

return <button onClick={onClick}>{label}</button>;

}

Button.defaultProps = {

label: "Click Me",

onClick: () => alert("Button clicked!"),

};

export default function App() {

return (

<div>

<Button /> {/\* Uses default label and onClick \*/}

<Button label="Submit" /> {/\* Custom label, default onClick \*/}

</div>

);

}

**Best Practices for Using Default Props**

1. **Use Defaults for Optional Props**: Define default values only for props that are optional.
2. **Avoid Overloading Defaults**: Use defaults for sensible fallback values. Avoid complex or large default objects.
3. **Combine with Prop Types**: Use PropTypes to enforce data types while providing defaults.
4. **Prefer Simplicity**: For functional components, prefer setting defaults in the argument list.

**Combining Default Props and PropTypes**

Using PropTypes together with defaultProps ensures type safety and default handling:

javascript

CopyEdit

import PropTypes from "prop-types";

function Card({ title, description }) {

return (

<div>

<h1>{title}</h1>

<p>{description}</p>

</div>

);

}

Card.defaultProps = {

title: "Default Title",

description: "Default Description",

};

Card.propTypes = {

title: PropTypes.string,

description: PropTypes.string,

};

export default Card;

**What are Props in React?**

**Props** (short for properties) are a mechanism for passing data from a parent component to a child component in React. They allow components to be reusable by providing dynamic, configurable values that can be customized.

**Key Features of Props**

1. **Immutable**: Props are read-only. A child component cannot modify its props—it can only use or display them.
2. **Unidirectional Data Flow**: Data flows in one direction—from the parent component to the child component.
3. **Customizable Components**: Props make components dynamic by allowing different values to be passed to them.
4. **Access via Arguments**: Props are accessed in a child component via the props object.

**Example of Using Props**

**Parent Component:**

javascript

CopyEdit

export default function App() {

return (

<div>

<Greeting name="John" />

<Greeting name="Jane" />

</div>

);

}

**Child Component:**

javascript

CopyEdit

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

**Explanation**:

* The App component passes the name prop ("John" and "Jane") to the Greeting component.
* The Greeting component receives the props object and accesses props.name to display the personalized message.

**Using Props with Object Destructuring**

You can simplify the code by destructuring the props object:

javascript

CopyEdit

function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

**Difference Between Props and State**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Props** | **State** |
| **Definition** | Props are inputs passed to components. | State is a local memory for a component. |
| **Mutability** | Immutable (read-only). | Mutable (can be updated with setState). |
| **Scope** | Passed from parent to child. | Managed within a single component. |
| **Usage** | Used to configure and customize components. | Used to store and manage dynamic data. |
| **Responsibility** | Controlled by the parent component. | Controlled by the component itself. |
| **Updates** | Cannot trigger re-renders. | Triggers re-renders when updated. |

**Example: Props vs. State**

**Parent Component:**

javascript

CopyEdit

export default function App() {

return (

<div>

<Counter title="Counter 1" />

<Counter title="Counter 2" />

</div>

);

}

**Child Component with Props and State:**

javascript

CopyEdit

import { useState } from 'react';

function Counter({ title }) {

const [count, setCount] = useState(0); // State

function increment() {

setCount(count + 1);

}

return (

<div>

<h2>{title}</h2> {/\* Using props \*/}

<p>Count: {count}</p> {/\* Using state \*/}

<button onClick={increment}>Increment</button>

</div>

);

}

**Explanation**:

1. **Props**: title is passed from the App component to each Counter component.
2. **State**: count is a local variable in the Counter component that is updated when the button is clicked.

**Key Differences Simplified**

|  |  |
| --- | --- |
| **Props** | **State** |
| External data (passed) | Internal data (managed) |
| Read-only | Can be changed (via setter functions) |
| Passed down to children | Exists only in the component |
| Static for a component | Dynamic and interactive |

**Props and State Together**

React components often use both **props** and **state** together to build complex, dynamic applications. By combining them:

* **Props** provide configuration and customization from parents.
* **State** manages dynamic data and user interactions within the component.

**What is State?**

* **Definition**: State is a component-specific memory in React used to store and manage data that changes over time.
* **Purpose**: Helps components "remember" information between renders, such as:
  + The current input value in a form.
  + The current image in a carousel.
  + Items in a shopping cart.
* **Managed by React**:
  + React keeps track of the state and re-renders the component when state updates.

**Key Concepts You Will Learn**

1. How to add a state variable using the useState Hook.
2. How the useState Hook provides two values (state variable and setter function).
3. Adding multiple state variables in one component.
4. Why state is called **local**.
5. When a regular variable isn’t enough.

**Why a Regular Variable Isn’t Enough**

* **Issue #1**: **Doesn’t persist between renders**:
  + Local variables are re-initialized every time the component re-renders.
* **Issue #2**: **Doesn’t trigger re-renders**:
  + Changes to local variables don’t notify React to update the UI.

**Solution: The useState Hook**

The useState Hook provides:

1. A **state variable** to retain data between renders.
2. A **state setter function** to update the state and trigger a re-render.

**Adding a State Variable**

1. **Import useState from React**:

javascript

CopyEdit

import { useState } from 'react';

1. **Replace a Local Variable with a State Variable**: Replace:

javascript

CopyEdit

let index = 0;

With:

javascript

CopyEdit

const [index, setIndex] = useState(0);

* + index: State variable.
  + setIndex: Setter function to update the state.

1. **Use the State Setter Function**: Update the handleClick function:

javascript

CopyEdit

function handleClick() {

setIndex(index + 1);

}

**Example: Basic State Usage**

Here’s the corrected example:

javascript

CopyEdit

import { useState } from 'react';

import { sculptureList } from './data.js';

export default function Gallery() {

const [index, setIndex] = useState(0); // Add state variable

function handleClick() {

setIndex(index + 1); // Update state

}

const sculpture = sculptureList[index];

return (

<>

<button onClick={handleClick}>Next</button>

<h2>

<i>{sculpture.name}</i> by {sculpture.artist}

</h2>

<h3>

({index + 1} of {sculptureList.length})

</h3>

<img src={sculpture.url} alt={sculpture.alt} />

<p>{sculpture.description}</p>

</>

);

}

**How useState Works**

1. **Initial Value**:
   * The first value passed to useState is the **initial value** of the state variable.
   * Example: useState(0) initializes index to 0.
2. **Array Destructuring**:
   * useState returns an array with two items:
     + State variable (e.g., index).
     + State setter function (e.g., setIndex).
3. **Updating State**:
   * Call the setter function with a new value.
   * This triggers React to:
     + Update the state value.
     + Re-render the component.

**Multiple State Variables**

* You can manage multiple independent state variables in a component.
* Example:

javascript

CopyEdit

const [index, setIndex] = useState(0);

const [showMore, setShowMore] = useState(false);

* **Example Component**:

javascript

CopyEdit

import { useState } from 'react';

import { sculptureList } from './data.js';

export default function Gallery() {

const [index, setIndex] = useState(0);

const [showMore, setShowMore] = useState(false);

function handleNextClick() {

setIndex(index + 1);

}

function handleMoreClick() {

setShowMore(!showMore); // Toggle state

}

const sculpture = sculptureList[index];

return (

<>

<button onClick={handleNextClick}>Next</button>

<h2>

<i>{sculpture.name}</i> by {sculpture.artist}

</h2>

<h3>

({index + 1} of {sculptureList.length})

</h3>

<button onClick={handleMoreClick}>

{showMore ? 'Hide' : 'Show'} details

</button>

{showMore && <p>{sculpture.description}</p>}

<img src={sculpture.url} alt={sculpture.alt} />

</>

);

}

**Why State is Called Local**

* **Scope**:
  + State variables belong only to the component in which they are declared.
  + They are not shared with other components.
* **Component-Specific Memory**:
  + Each component can have its own unique state, independent of others.

**Pitfalls of Using Hooks**

1. **Hooks Must Be Called at the Top Level**:
   * Cannot call Hooks inside loops, conditions, or nested functions.
   * Example:

javascript

CopyEdit

if (true) {

const [index, setIndex] = useState(0); // ❌ Not allowed

}

1. **React-Managed Scope**:
   * Hooks work only during rendering, as React controls when and how they are called.

**Anatomy of useState**

1. **Initial Render**:
   * useState(0) initializes index to 0.
2. **State Update**:
   * Calling setIndex(1) updates index to 1.
3. **Re-render**:
   * React re-renders the component with the updated state.

**When to Use Multiple State Variables**

* Use separate state variables if they represent unrelated pieces of information.
* **Example**:
  + index: Tracks the current item.
  + showMore: Toggles additional details.
* **When to Combine**:
  + If two variables are updated together frequently, combine them into a single state object.
  + Example: A form with multiple fields.

**Real DOM, Virtual DOM, and Shadow DOM – What's the Difference?**

**1. What is the DOM?**

* The **DOM (Document Object Model)** is a programming interface for web documents.
* It represents the **HTML structure of a webpage as a tree of objects**.
* Each element of the HTML is a node in this tree, allowing JavaScript to interact with and manipulate these elements.

**2. Real DOM**

* **Definition**: The actual DOM tree structure created by the browser based on the HTML content of the page.
* **Characteristics**:
  + Accessible via the document object.
  + **Manipulation**: Can be accessed and updated using JavaScript (e.g., document.body.style.background = 'red' changes the background color).
  + **Drawback**: Updates are **expensive and slow** because modifying the DOM re-renders the entire tree.
* **Example**:

javascript

CopyEdit

document.body.style.background = 'blue';

* **Problem**:
  + Even small updates trigger a re-render of the **entire DOM tree**, which can negatively impact performance, especially for complex web applications.

**3. Virtual DOM**

* **Definition**: A lightweight, in-memory representation of the real DOM.
* **How it works**:
  1. React creates a **virtual DOM** that mirrors the real DOM.
  2. Any changes to the page are first applied to the **virtual DOM** (not the real DOM directly).
  3. React creates a copy of the virtual DOM (let's call it **virtual DOM 2**) and compares it with the original virtual DOM.
  4. It uses a **diffing algorithm** to detect changes between the original and updated virtual DOM.
  5. Only the **changed parts** are updated in the real DOM.
* **Benefits**:
  1. Faster updates since React minimizes the number of real DOM manipulations.
  2. Improves performance for dynamic web applications.
* **Example**:

javascript

CopyEdit

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

// Virtual DOM only updates the affected node instead of the entire DOM

return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;

* **Visual Representation**:
  1. Virtual DOM → Copy Virtual DOM → Compare → Update only the necessary nodes in the Real DOM.

**4. Shadow DOM**

* **Definition**: A hidden DOM tree that is attached to an element in the real DOM.
* **Purpose**:
  + Encapsulation: Provides a **scope for styles and behavior**, preventing interference from styles or scripts in the global real DOM.
* **Key Features**:
  + Completely **separate** from the real DOM.
  + Changes in the shadow DOM **do not affect** the real DOM and vice versa.
  + Often used in **custom elements** (Web Components) like <video> players or custom widgets.
* **Example Use Case**:
  + A video player’s internal buttons (e.g., play, pause) are controlled by the shadow DOM and are not visible in the real DOM.
* **How to Inspect Shadow DOM**:
  + In Chrome DevTools:
    1. Go to **Settings** → **Preferences** → **Elements**.
    2. Enable **"Show user agent shadow DOM"**.
* **Code Example**:

html

CopyEdit

<div id="shadow-host"></div>

<script>

const shadowHost = document.getElementById("shadow-host");

const shadowRoot = shadowHost.attachShadow({ mode: "open" });

shadowRoot.innerHTML = "<p>This is shadow DOM content</p>";

</script>

**5. Comparison: Real DOM vs Virtual DOM vs Shadow DOM**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Real DOM** | **Virtual DOM** | **Shadow DOM** |
| **Definition** | Actual DOM rendered by the browser. | Lightweight, in-memory copy of the DOM. | Hidden DOM attached to real DOM elements. |
| **Performance** | Updates are slow and expensive. | Updates are faster using diffing. | No direct relation to performance. |
| **Purpose** | Represents the actual HTML structure. | Enhances performance for dynamic updates. | Encapsulation for custom elements. |
| **Updates** | Entire DOM re-renders for any change. | Only affected nodes are updated. | Independent of real DOM updates. |
| **Usage** | Standard for all web pages. | React, Vue.js, and other frameworks. | Web Components like <video>. |
| **Style Scope** | Global; styles can bleed. | Global, but scoped by framework. | Encapsulated and scoped locally. |

**6. Examples of Use Cases**

* **Real DOM**:
  + Simple static pages where performance isn't a concern.
* **Virtual DOM**:
  + React applications that require frequent updates and dynamic rendering.
* **Shadow DOM**:
  + Custom widgets or reusable components (e.g., <video> controls, custom date pickers).

**7. Key Differences**

* **Real DOM**:
  + Directly manipulated by JavaScript.
  + Updates are slow and expensive.
* **Virtual DOM**:
  + Acts as a middle layer to optimize updates to the real DOM.
  + Faster than real DOM operations due to selective updates.
* **Shadow DOM**:
  + Provides encapsulation and isolation for custom elements.
  + Not tied to the real DOM or virtual DOM updates.

**8. Conclusion**

* **Real DOM**: The traditional structure representing the HTML document.
* **Virtual DOM**: Optimizes the update process, enhancing performance for modern frameworks like React.
* **Shadow DOM**: Isolates styles and functionality, enabling reusable, self-contained components.  
  Understanding these differences helps in choosing the right technology for specific web application needs.

**Function Components vs Class Components in React (Point-by-Point Explanation)**

**1. What are React Components?**

* React components are **reusable** building blocks of a user interface.
* They encapsulate their **logic** and **UI structure**.
* Components accept **props** (inputs) and return **React elements** to describe what appears on the screen.
* **Two main types** of React components:
  1. **Function Components**: Defined as JavaScript functions, suitable for stateless or lightweight components.
  2. **Class Components**: Defined as ES6 classes, used for stateful components and those requiring lifecycle methods.

**2. Function Components**

* **Definition**: JavaScript functions that take props as input and return JSX elements.
* **Syntax**:

jsx

CopyEdit

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

OR

jsx

CopyEdit

const Greeting = (props) => {

return <h1>Hello, {props.name}!</h1>;

};

* **Key Features**:
  + Simpler and easier to read.
  + Stateless in earlier React versions, but now **stateful** with **React Hooks**.
  + Preferred in modern React development.

**3. Class Components**

* **Definition**: ES6 classes that extend React.Component and include a render() method.
* **Syntax**:

jsx

CopyEdit

class Greeting extends React.Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

* **Key Features**:
  + Can manage state using this.state.
  + Use lifecycle methods like componentDidMount and componentDidUpdate.
  + Verbose and requires this context, making them slightly harder to read.

**4. State Management**

* **Function Components**:
  + Use useState hook for state management.
  + Example:

jsx

CopyEdit

import React, { useState } from 'react';

const Counter = () => {

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

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

};

export default Counter;

* **Class Components**:
  + State is defined in the constructor and updated using this.setState.
  + Example:

jsx

CopyEdit

class Counter extends React.Component {

constructor(props) {

super(props);

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}>Increment</button>

</div>

);

}

}

**5. Lifecycle Methods**

* **Class Components**:
  + Use predefined lifecycle methods:
    - componentDidMount: Runs after the component is mounted.
    - componentDidUpdate: Runs after the component updates.
    - componentWillUnmount: Runs before the component is removed from the DOM.
  + Example:

jsx

CopyEdit

class Timer extends React.Component {

componentDidMount() {

console.log('Component mounted');

}

componentWillUnmount() {

console.log('Component will unmount');

}

render() {

return <p>Timer running...</p>;

}

}

* **Function Components**:
  + Use the useEffect hook to mimic lifecycle behavior:

jsx

CopyEdit

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

const Timer = () => {

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

useEffect(() => {

console.log('Component did mount or update');

return () => console.log('Component will unmount');

});

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

};

export default Timer;

* + useEffect runs after every render, similar to componentDidMount and componentDidUpdate. Cleanup mimics componentWillUnmount.

**6. Readability**

* **Function Components**:
  + Concise and straightforward.
  + Less boilerplate code.
* **Class Components**:
  + Verbose due to this bindings and lifecycle methods.

**7. Performance**

* **Function Components**:
  + Lightweight, as they don’t require instances.
  + Optimized for modern React with hooks and React.memo.
* **Class Components**:
  + Heavier due to component instances and this context.
  + May require performance optimizations like shouldComponentUpdate.

**8. React Hooks (Only in Function Components)**

* Hooks enable function components to:
  + Manage state (useState).
  + Use lifecycle methods (useEffect).
  + Access context (useContext).
* Example:

jsx

CopyEdit

import React, { useContext } from 'react';

const ThemeContext = React.createContext('light');

const ThemedComponent = () => {

const theme = useContext(ThemeContext);

return <div>The current theme is {theme}.</div>;

};

**9. React’s Recommendation**

* React recommends using **function components** with hooks for modern development.
* Function components are easier to use, test, and maintain.

**10. When to Use Which?**

* **Function Components**:
  + Most use cases in modern React.
  + Simplifies development with hooks.
* **Class Components**:
  + Working with legacy codebases.
  + Using libraries or tools that depend on class components.

**11. Comparison Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Function Components** | **Class Components** |
| **Definition** | JavaScript function | ES6 class extending React.Component |
| **State Management** | useState, useReducer | this.state, this.setState |
| **Lifecycle Methods** | useEffect | componentDidMount, etc. |
| **Syntax** | Concise and simple | Verbose and requires this context |
| **Performance** | Lightweight | Heavier due to instances |
| **Hooks Support** | Yes | No |

**Prop**

**1. Specifying a Default Value for a Prop**

* **Default Values**: You can set a default value for a prop in case it is not provided by the parent component.
* **How to Set Default**:

javascript

CopyEdit

function Avatar({ person, size = 100 }) {

// If size is not passed, it defaults to 100.

}

* **Behavior**:
  + The default value is used when:
    - The prop is **missing**.
    - The prop is explicitly set to **undefined**.
  + The default value is **not used** when:
    - The prop is set to **null**.
    - The prop is set to other falsy values, like **0**.

**2. Forwarding Props with the JSX Spread Syntax**

* **Repetitive Prop Forwarding**: When passing all props individually, the code may become repetitive.

javascript

CopyEdit

function Profile({ person, size, isSepia, thickBorder }) {

return (

<Avatar

person={person}

size={size}

isSepia={isSepia}

thickBorder={thickBorder}

/>

);

}

* **Using the Spread Syntax**: Instead of listing each prop, you can forward all props using the spread operator (...).

javascript

CopyEdit

function Profile(props) {

return (

<Avatar {...props} />

);

}

* **When to Use**: Use the spread syntax **sparingly**. If overused, it may make the code less readable and harder to debug.
* **Best Practice**: Avoid using the spread syntax if your component uses most or all of the props directly.

**3. Passing JSX as children**

* **Definition**: The children prop allows you to pass JSX elements inside a component as if they were "nested."
* **Example with Nesting**:

javascript

CopyEdit

<Card>

<Avatar />

</Card>

* **Accessing children**: The parent component can receive the nested JSX in the children prop.

javascript

CopyEdit

function Card({ children }) {

return (

<div className="card">

{children}

</div>

);

}

* **Flexibility**:
  + children can hold any JSX content.
  + For example:

javascript

CopyEdit

<Card>

<h1>Hello!</h1>

</Card>

* + The Card component doesn’t need to know what is inside; it simply renders its children.
* **Use Case**: The children prop is often used for creating reusable visual wrappers, like panels, cards, or grids.

**4. How Props Change Over Time**

* **Dynamic Props**:
  + Props are **not always static**; they can change over time based on user interaction or data updates.
  + Example:

javascript

CopyEdit

export default function Clock({ color, time }) {

return (

<h1 style={{ color: color }}>

{time}

</h1>

);

}

* + The time prop updates every second, and the color prop changes when a user selects a different color.
* **Immutability of Props**:
  + Props are **immutable**—once passed, they cannot be modified by the child component.
  + If a child component needs updated data, the parent must pass **new props**.
  + Example:
    - If the Clock component needs a new time, the parent must update the time prop.
* **React Memory Management**:
  + When props change:
    - The old props are replaced with new ones.
    - The JavaScript engine automatically cleans up the memory of the old props.

**5. Don’t Modify Props**

* **Key Rule**: Never attempt to modify props inside a child component.
  + If you need to respond to user actions or dynamic data, use **state** (a concept explained separately).
* **Correct Approach**:
  + If a prop needs to change, the child component must ask the parent to pass new props.

**6. Summary**

* **Default Values**: Use = value in the destructuring to specify defaults for missing props.
* **Forwarding Props**: Use the spread syntax (...props) to forward all props, but apply it judiciously.
* **Using children**: The children prop lets you create flexible components that can wrap any nested JSX.
* **Changing Props**: Props can update over time, but they remain immutable. Updates must come from the parent.

**2. Lists and Keys:-**

**1. Why Render Lists in React?**

* Many applications involve rendering lists of similar components, like:
  + Comments sections
  + Photo galleries
  + Lists of items (e.g., users, tasks, products)
* React's ability to work with JavaScript methods like map() and filter() allows us to dynamically create and manipulate lists of components.

**2. Using map() to Render Lists**

**Steps:**

1. **Store Data in an Array**: Example:

javascript

CopyEdit

const people = [

'Creola Katherine Johnson: mathematician',

'Mario José Molina-Pasquel Henríquez: chemist',

'Mohammad Abdus Salam: physicist',

'Percy Lavon Julian: chemist',

'Subrahmanyan Chandrasekhar: astrophysicist',

];

1. **Transform Array into Components**: Use map() to iterate over the array and transform each item into a JSX element:

javascript

CopyEdit

const listItems = people.map(person => <li>{person}</li>);

1. **Render the List**: Wrap the listItems in a parent element, such as <ul>:

javascript

CopyEdit

return <ul>{listItems}</ul>;

**Full Example:**

javascript

CopyEdit

const people = [

'Creola Katherine Johnson: mathematician',

'Mario José Molina-Pasquel Henríquez: chemist',

'Mohammad Abdus Salam: physicist',

'Percy Lavon Julian: chemist',

'Subrahmanyan Chandrasekhar: astrophysicist',

];

export default function List() {

const listItems = people.map(person =>

<li>{person}</li>

);

return <ul>{listItems}</ul>;

}

**3. React Key Warning**

**Problem:**

If you run the above code, you will see this console warning:

sql

CopyEdit

Warning: Each child in a list should have a unique "key" prop.

**Solution:**

* React requires a unique key prop for each item in a list.
* **Why?** React uses key to identify which items have changed, been added, or removed, optimizing rendering performance.
* Update the map() function to include a key:

javascript

CopyEdit

const listItems = people.map((person, index) =>

<li key={index}>{person}</li>

);

**4. Structuring Data**

**Why Structure Data?**

* Storing complex data in an array of objects improves organization and flexibility.
* Example:

javascript

CopyEdit

const people = [

{ id: 0, name: 'Creola Katherine Johnson', profession: 'mathematician' },

{ id: 1, name: 'Mario José Molina-Pasquel Henríquez', profession: 'chemist' },

{ id: 2, name: 'Mohammad Abdus Salam', profession: 'physicist' },

{ id: 3, name: 'Percy Lavon Julian', profession: 'chemist' },

{ id: 4, name: 'Subrahmanyan Chandrasekhar', profession: 'astrophysicist' },

];

**5. Using filter() to Render Specific Items**

**Steps:**

1. **Filter the Data**: Use the filter() method to return only items that meet a specific condition:

javascript

CopyEdit

const chemists = people.filter(person =>

person.profession === 'chemist'

);

1. **Map Over Filtered Data**: Use map() to create components from the filtered array:

javascript

CopyEdit

const listItems = chemists.map(person =>

<li key={person.id}>

<b>{person.name}:</b> {person.profession}

</li>

);

1. **Render the Filtered List**: Wrap the listItems in a <ul> and return it:

javascript

CopyEdit

return <ul>{listItems}</ul>;

**Full Example:**

javascript

CopyEdit

const people = [

{ id: 0, name: 'Creola Katherine Johnson', profession: 'mathematician' },

{ id: 1, name: 'Mario José Molina-Pasquel Henríquez', profession: 'chemist' },

{ id: 2, name: 'Mohammad Abdus Salam', profession: 'physicist' },

{ id: 3, name: 'Percy Lavon Julian', profession: 'chemist' },

{ id: 4, name: 'Subrahmanyan Chandrasekhar', profession: 'astrophysicist' },

];

export default function List() {

const chemists = people.filter(person =>

person.profession === 'chemist'

);

const listItems = chemists.map(person =>

<li key={person.id}>

<b>{person.name}:</b> {person.profession}

</li>

);

return <ul>{listItems}</ul>;

}

**6. Rendering Lists with Additional Data**

**Enhanced Example:**

Suppose each person has an additional property, accomplishment:

javascript

CopyEdit

const people = [

{ id: 0, name: 'Creola Katherine Johnson', profession: 'mathematician', accomplishment: 'Space calculations' },

{ id: 1, name: 'Mario José Molina-Pasquel Henríquez', profession: 'chemist', accomplishment: 'Ozone research' },

{ id: 2, name: 'Mohammad Abdus Salam', profession: 'physicist', accomplishment: 'Electroweak theory' },

{ id: 3, name: 'Percy Lavon Julian', profession: 'chemist', accomplishment: 'Steroid synthesis' },

{ id: 4, name: 'Subrahmanyan Chandrasekhar', profession: 'astrophysicist', accomplishment: 'Stellar evolution' },

];

Use filter() and map():

javascript

CopyEdit

export default function List() {

const chemists = people.filter(person =>

person.profession === 'chemist'

);

const listItems = chemists.map(person =>

<li key={person.id}>

<img

src={getImageUrl(person)} // Assume getImageUrl is defined elsewhere

alt={person.name}

/>

<p>

<b>{person.name}:</b> {person.profession} known for {person.accomplishment}

</p>

</li>

);

return <ul>{listItems}</ul>;

}

**7. Pitfalls to Avoid**

**Implicit vs. Explicit Return in Arrow Functions**

* **Implicit Return**: No curly braces; directly returns the expression:

javascript

CopyEdit

const listItems = people.map(person =>

<li>{person.name}</li> // Implicit return

);

* **Explicit Return**: Uses curly braces; must include return:

javascript

CopyEdit

const listItems = people.map(person => {

return <li>{person.name}</li>; // Explicit return

});

* **Tip**: Use explicit returns when you need more lines of code inside the arrow function.

**8. Benefits of This Approach**

1. **Dynamic Rendering**: Automatically updates when the array changes.
2. **Reusable Logic**: filter() and map() make the code modular and reusable.
3. **Optimized Performance**: Using keys ensures React can efficiently update the DOM.
4. **Clear Structure**: Separating data (people) from rendering logic (filter and map) improves readability.