**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.

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.

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).

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 syntax options for conditions:

**a. if/else**

* Most straightforward way to handle conditions.
* Example:

function Item({ name, isPacked }) {

if (isPacked) {

return <li>{name} ✅</li>;

} else {

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

}

}

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

* Useful for inline conditional rendering.
* Example:

function Item({ name, isPacked }) {

return (

<li>{name} {isPacked ? '✅' : ''}</li>

);

}

**c. Logical && Operator**

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

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.

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:

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)**

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:**

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:

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:

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:

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**

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:

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:**

export default function App() {

return (

<div>

<Greeting name="John" />

<Greeting name="Jane" />

</div>

);

}

**Child Component:**

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:

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:**

export default function App() {

return (

<div>

<Counter title="Counter 1" />

<Counter title="Counter 2" />

</div>

);

}

**Child Component with Props and State:**

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**:

import { useState } from 'react';

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

let index = 0;

With:

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:

function handleClick() {

setIndex(index + 1);

}

**Example: Basic State Usage**

Here’s the corrected example:

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:

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

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

* **Example Component**:

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:

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 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 (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**:

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**:

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

<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 .
  + 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 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**: functions that take props as input and return JSX elements.
* **Syntax**:

jsx

function Greeting(props) {

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

}

OR

jsx

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

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

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

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

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

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

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** | 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**:

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.

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 (...).

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**:

<Card>

<Avatar />

</Card>

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

function Card({ children }) {

return (

<div className="card">

{children}

</div>

);

}

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

<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:

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 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 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:

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:

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

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

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

**Full Example:**

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

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:

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:

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:

const chemists = people.filter(person =>

person.profession === 'chemist'

);

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

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:

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

**Full Example:**

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:

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():

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:

const listItems = people.map(person =>

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

);

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

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.

**HOOKS:-**

**What is useMemo?**

* useMemo is a **React Hook** used to **optimize performance** by memoizing the result of an expensive computation.
* It **only recalculates** the memoized value when its dependencies change.
* It helps **prevent unnecessary calculations** on every render.

**Syntax**

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

1. useMemo **returns a memoized (cached) value**.
2. It **accepts two arguments**:
   * A function that **returns the computed value**.
   * A **dependency array** [a, b], which determines when the function should be re-executed.
3. If the dependencies do **not** change, the function **is not executed again**, and the previous result is returned.

**Why Use useMemo?**

* Avoids **unnecessary recalculations** of expensive functions.
* Improves **performance** in large applications.
* Prevents **unnecessary re-renders** of child components.

**When to Use useMemo?**

Use useMemo in situations where:

1. **Expensive Calculations**
   * When a function does a lot of computations (e.g., filtering, sorting, complex math).
2. **Preventing Re-renders**
   * When a computed value is passed as a prop to a child component that does **not need to re-render**.
3. **Reference Equality Issues**
   * When using objects or arrays as dependencies in useEffect or useCallback, useMemo helps **maintain reference equality**.

**Example 1: Basic Usage of useMemo**

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

function ExpensiveCalculation({ num }) {

const computeFactorial = (n) => {

console.log('Calculating...');

return n <= 1 ? 1 : n \* computeFactorial(n - 1);

};

const factorial = useMemo(() => computeFactorial(num), [num]);

return <div>Factorial of {num}: {factorial}</div>;

}

function App() {

const [count, setCount] = useState(5);

return (

<div>

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

<ExpensiveCalculation num={count} />

</div>

);

}

export default App;

**Explanation:**

* computeFactorial(num) is an expensive recursive function.
* useMemo caches the computed factorial **until num changes**.
* Clicking the button only recalculates the factorial when num updates.

**Example 2: Preventing Unnecessary Re-renders**

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

function ChildComponent({ data }) {

console.log("Child Rendered");

return <p>Data: {data.value}</p>;

}

function ParentComponent() {

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

const memoizedData = useMemo(() => ({ value: "Hello, World!" }), []);

return (

<div>

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

<ChildComponent data={memoizedData} />

</div>

);

}

export default ParentComponent;

**Explanation:**

* useMemo ensures memoizedData remains **the same object reference** across renders.
* The ChildComponent does **not** re-render unnecessarily because the reference **remains stable**.

**Example 3: Optimizing Array Filtering**

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

function FilterList({ items, filter }) {

const filteredItems = useMemo(() => {

console.log('Filtering...');

return items.filter((item) => item.includes(filter));

}, [items, filter]);

return (

<ul>

{filteredItems.map((item, index) => (

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

))}

</ul>

);

}

function App() {

const [search, setSearch] = useState('');

const items = ['Apple', 'Banana', 'Orange', 'Mango'];

return (

<div>

<input value={search} onChange={(e) => setSearch(e.target.value)} placeholder="Search..." />

<FilterList items={items} filter={search} />

</div>

);

}

export default App;

**Explanation:**

* useMemo **memoizes the filtered array** so that filtering **only occurs when the search value changes**.
* Without useMemo, filtering would run **on every re-render**, affecting performance.

**Rules of useMemo**

1. **Use only when necessary**:
   * Applying useMemo to simple calculations **adds unnecessary complexity**.
2. **Avoid using it with primitive values**:
   * If the calculation is trivial, **no need for useMemo**.
3. **Dependencies must be correct**:
   * Incorrect dependency arrays may cause:
     + **Stale values** (missing dependencies).
     + **Unnecessary recalculations** (extra dependencies).
4. **Works only inside functional components**:
   * useMemo cannot be used in class components.

**Common Mistakes with useMemo**

|  |  |
| --- | --- |
| **Mistake** | **Correct Approach** |
| Using useMemo for **every** state update | Use it **only** for expensive operations |
| Forgetting dependencies | Always provide correct dependencies |
| Memoizing a function instead of using useCallback | Use useCallback for memoizing functions |

**Difference Between useMemo and useCallback**

|  |  |  |
| --- | --- | --- |
| **Feature** | **useMemo** | **useCallback** |
| Purpose | Memoizes **computed values** | Memoizes **functions** |
| Return Value | Returns a value | Returns a function |
| When to Use | When calculations are **expensive** | When passing functions as **props** |

**Example:**

* **useMemo**

const memoizedValue = useMemo(() => expensiveCalculation(num), [num]);

* **useCallback**

const memoizedFunction = useCallback(() => handleClick(), []);

**Key Takeaways**

✔ useMemo optimizes performance by **memoizing computed values**.  
✔ It is useful for **expensive calculations**, **preventing re-renders**, and **maintaining reference stability**.  
✔ Only re-runs when **dependencies change**.  
✔ Do **not** overuse it; use only when needed.  
✔ Use useCallback instead if you need to **memoize a function**.

**Final Thoughts**

* useMemo is a **powerful performance optimization tool** in React.
* It is best used for **expensive calculations** and **reducing unnecessary re-renders**.
* However, **overusing useMemo can make the code harder to read and debug**.

Would you like a **real-world example** on how useMemo improves performance in a **React project**?

**useCallback :-**

**How to Explain useCallback to a Beginner in React?**

If you're teaching useCallback to a **beginner React developer**, keep the explanation simple, practical, and engaging. Focus on **why it's needed**, **how it works**, and **when to use it**.

**1. Start with a Simple Analogy 🧠**

Think of useCallback like **saving a favorite contact in your phone** 📱.

🔹 Instead of typing the full number every time you call, you **save it in contacts**.  
🔹 Whenever you want to call, you just **reuse the saved contact**.  
🔹 This avoids unnecessary effort and prevents mistakes.

Similarly, useCallback **remembers a function** and **prevents it from being recreated unnecessarily** on every render.

**2. What Problem Does useCallback Solve?**

**❌ Without useCallback: Unnecessary Function Recreation**

Every time a component **re-renders**, functions inside it **get recreated**. This causes:

1. **Unnecessary memory usage** 🛑
2. **Unwanted child component re-renders** 🔄

**✅ With useCallback: Optimized Function Reuse**

useCallback **remembers the function** and **only recreates it when necessary**, making React more efficient.

**3. Simple Example Without useCallback 🚀**

Let’s say we have a component that passes a function to its child.

jsx

CopyEdit

import React, { useState } from "react";

import ChildComponent from "./ChildComponent";

const ParentComponent = () => {

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

const handleClick = () => {

console.log("Button clicked!");

};

return (

<div>

<h2>Count: {count}</h2>

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

<ChildComponent onClick={handleClick} />

</div>

);

};

export default ParentComponent;

Now, let’s check ChildComponent.js:

jsx

CopyEdit

import React from "react";

const ChildComponent = ({ onClick }) => {

console.log("Child re-rendered!");

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

};

export default React.memo(ChildComponent);

**Problem Here:**

* Every time ParentComponent **re-renders** (when count changes), handleClick is **recreated**.
* This causes ChildComponent to **re-render unnecessarily**, even though nothing changed.

**4. Optimizing with useCallback 🏎️**

Now, let’s **fix** this issue using useCallback:

jsx

CopyEdit

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

import ChildComponent from "./ChildComponent";

const ParentComponent = () => {

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

const handleClick = useCallback(() => {

console.log("Button clicked!");

}, []); // ✅ Function stays the same unless dependencies change

return (

<div>

<h2>Count: {count}</h2>

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

<ChildComponent onClick={handleClick} />

</div>

);

};

export default ParentComponent;

**What Changed?**

✔️ handleClick is **now memoized** with useCallback().  
✔️ ChildComponent **won’t re-render** unnecessarily.  
✔️ App performance improves. 🚀

**5. Explaining useCallback in Simple Steps**

1️⃣ **useCallback remembers functions**  
2️⃣ **It only recreates them when dependencies change**  
3️⃣ **Prevents unnecessary re-renders in child components**

**6. Key Takeaways for a Beginner 🎯**

✔️ **Use useCallback when:**

* A function is passed **to a child component** (especially with React.memo)
* You want to **avoid unnecessary re-renders**
* The function **doesn’t need to change often**

❌ **Avoid useCallback when:**

* The function is **not passed to a child**
* Performance issues **don’t exist**

**7. Difference Between useCallback and useMemo**

|  |  |  |
| --- | --- | --- |
| **Feature** | **useCallback** | **useMemo** |
| **What it Memoizes?** | A **function** | A **computed value** |
| **Use Case** | When passing functions to children | When storing expensive calculations |
| **Returns** | A **memoized function** | A **memoized value** |

**8. Final One-Liner Summary**

💡 "useCallback helps React **remember functions**, so they don’t get recreated unnecessarily, preventing extra re-renders and improving performance!" 🚀

**useContext:-**

**1. Start with a Simple Analogy 🎭**

Imagine you are in a **huge house** 🏡 with multiple rooms.

* There's a **WiFi router** in the living room providing internet to everyone. 📶
* Instead of **running cables** to every room, each person simply **connects to the WiFi** to access the internet.

**How does this relate to React?**

* The **WiFi router** 🛜 = **Context Provider** (Stores shared data)
* The **people connecting to WiFi** 👨‍👩‍👦‍👦 = **Context Consumers** (useContext Hook)
* No need to **pass cables (props) to every room** → **Direct access!**

Similarly, useContext **lets components access shared data without passing props manually**.

**2. What Problem Does useContext Solve?**

**❌ Without useContext: Prop Drilling Hell 😵**

If a **parent component** needs to pass data **to a deep child component**, we have to pass props **through multiple components**:

jsx

CopyEdit

const App = () => {

const user = "John Doe";

return <Parent user={user} />;

};

const Parent = ({ user }) => {

return <Child user={user} />;

};

const Child = ({ user }) => {

return <GrandChild user={user} />;

};

const GrandChild = ({ user }) => {

return <h2>Hello, {user}!</h2>;

};

**Problems Here:**

1️⃣ We have to **pass user through every component**, even if they **don’t need it**.  
2️⃣ **More components = More complexity** and **harder maintenance**.

**Syntax of useContext**

javascript

CopyEdit

const value = useContext(MyContext);

**Parameters:**

* **MyContext** – The context object created using React.createContext().

**Returns:**

* **The current value of the context.**

**When to Use useContext?**

✔ When **data needs to be accessed by multiple components** (e.g., user authentication, themes, language settings).  
✔ When **prop drilling** (passing props down multiple levels) becomes problematic.  
✔ When a **global state is required**, but Redux or other state management libraries are too complex.

**Example: Using useContext for Theme Switching**

**Step 1: Create Context**

javascript

CopyEdit

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

// Create a context

const ThemeContext = createContext();

export function ThemeProvider({ children }) {

const [theme, setTheme] = useState('light');

const toggleTheme = () => {

setTheme(prevTheme => (prevTheme === 'light' ? 'dark' : 'light'));

};

return (

<ThemeContext.Provider value={{ theme, toggleTheme }}>

{children}

</ThemeContext.Provider>

);

}

export default ThemeContext;

* ThemeContext is created using createContext().
* The ThemeProvider component manages the theme state.
* ThemeContext.Provider wraps the application and **provides the state and function** to its children.

**Step 2: Consume Context in Components**

javascript

CopyEdit

import React, { useContext } from 'react';

import ThemeContext from './ThemeContext';

function ThemeToggleButton() {

const { theme, toggleTheme } = useContext(ThemeContext);

return (

<button onClick={toggleTheme}>

Current Theme: {theme} (Click to Toggle)

</button>

);

}

export default ThemeToggleButton;

* useContext(ThemeContext) is used to **access theme and toggleTheme** from ThemeProvider.
* The **button updates the theme** when clicked.

**Step 3: Wrap the Application with ThemeProvider**

javascript

CopyEdit

import React from 'react';

import { ThemeProvider } from './ThemeContext';

import ThemeToggleButton from './ThemeToggleButton';

function App() {

return (

<ThemeProvider>

<ThemeToggleButton />

</ThemeProvider>

);

}

export default App;

* ThemeProvider wraps the entire app, making the theme available to all components.

**How Context Works Internally**

1. **Create a Context** – React.createContext(defaultValue).
2. **Provide the Context Value** – Use Context.Provider to supply a value.
3. **Consume the Context Value** – Use useContext(Context) to access it in components.

**Avoiding useContext Mistakes**

❌ **Using useContext without a Provider**

javascript

CopyEdit

const theme = useContext(ThemeContext); // theme will be undefined if no Provider is used.

✔ **Always wrap components with the Context Provider.**

❌ **Updating Context Directly**

javascript

CopyEdit

const theme = useContext(ThemeContext);

theme = 'dark'; // ❌ This will not update the state.

✔ **Use the provided setter function (setTheme in this case).**

**useContext vs Prop Drilling**

|  |  |  |
| --- | --- | --- |
| **Feature** | **useContext** | **Prop Drilling** |
| Code Cleanliness | ✅ Cleaner code | ❌ Messy with deeply nested props |
| Performance | ✅ Optimized | ❌ Can cause unnecessary re-renders |
| Readability | ✅ More readable | ❌ Difficult to manage |
| State Sharing | ✅ Global access | ❌ Limited to passed props |

**Best Practices for useContext**

✔ **Use it for globally shared state** (e.g., authentication, themes, language preferences).  
✔ **Avoid unnecessary re-renders** by **structuring context properly**.  
✔ **Combine with useReducer** for more complex state management.  
✔ **Keep separate context files** for modularity.

**Example: Combining useContext with useReducer**

javascript

CopyEdit

import React, { createContext, useReducer } from 'react';

const CounterContext = createContext();

const counterReducer = (state, action) => {

switch (action.type) {

case 'increment': return state + 1;

case 'decrement': return state - 1;

default: return state;

}

};

export function CounterProvider({ children }) {

const [count, dispatch] = useReducer(counterReducer, 0);

return (

<CounterContext.Provider value={{ count, dispatch }}>

{children}

</CounterContext.Provider>

);

}

export default CounterContext;

* **Combining useContext and useReducer** allows for **better state management**.
* This pattern is similar to **Redux but built-in** to React.

**Key Takeaways**

✔ useContext allows **direct access to context values** without prop drilling.  
✔ It **works with createContext()** and a **Provider component**.  
✔ Used for **global state management** (e.g., themes, authentication, language settings).  
✔ **Avoid unnecessary re-renders** by **structuring context efficiently**.  
✔ Can be **combined with useReducer** for advanced state management.

**Final Thoughts**

* useContext **simplifies state management** in React.
* **Perfect for avoiding prop drilling** and **sharing data across multiple components**.
* If you need more control, **combine it with useReducer or Redux**.

🚀 **Need more examples? Let me know!**

**useEffect() vs. Component Lifecycle Methods in React :-**

**Introduction**

* Before React Hooks, **class components** used **lifecycle methods** to handle side effects.
* With Hooks, **useEffect() replaces lifecycle methods** for functional components.
* Both approaches help manage **component mounting, updating, and unmounting**.

**1️⃣ What is useEffect() in React?**

* useEffect() is a **React Hook** that lets you perform side effects in functional components.
* It **runs after the component renders** and can handle **data fetching, subscriptions, DOM updates, etc.**
* It can **mimic** class component lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount).

**Syntax of useEffect()**

javascript

CopyEdit

import { useEffect } from 'react';

useEffect(() => {

// Side effect logic here

return () => {

// Cleanup function (like componentWillUnmount)

};

}, [dependencies]); // Dependencies array

🔹 Runs **after every render** (if no dependencies are provided).  
🔹 Runs **only once** if the **dependencies array is empty** ([]).  
🔹 Runs **when dependencies change** (if dependencies are listed).  
🔹 Cleanup function **executes when the component unmounts**.

**2️⃣ What are Component Lifecycle Methods?**

* **Class components** use lifecycle methods to manage component behavior.
* These methods are grouped into **Mounting, Updating, and Unmounting** phases.

**React Lifecycle Methods**

|  |  |  |
| --- | --- | --- |
| **Phase** | **Lifecycle Method** | **Equivalent useEffect() Usage** |
| **Mounting** | componentDidMount | useEffect(() => { ... }, []) |
| **Updating** | componentDidUpdate | useEffect(() => { ... }, [dependencies]) |
| **Unmounting** | componentWillUnmount | useEffect(() => { return () => { ... }; }, []) |

**3️⃣ useEffect() vs. Lifecycle Methods: Comparison**

**🔹 ComponentDidMount (Runs Once on Mount)**

| **Lifecycle Method** | **useEffect Equivalent** |
| --- | --- |
| componentDidMount() | useEffect(() => { /\* Side effect \*/ }, []); |

✔ Used for **API calls, subscriptions, event listeners, logging, etc.**  
✔ Runs **only once** after the component mounts.

**Example: Fetching API Data (Class Component)**

javascript

CopyEdit

class FetchData extends React.Component {

componentDidMount() {

fetch("https://api.example.com/data")

.then(response => response.json())

.then(data => console.log(data));

}

render() {

return <h1>Fetching Data...</h1>;

}

}

**Example: Fetching API Data (useEffect() in Functional Component)**

javascript

CopyEdit

import { useEffect } from 'react';

function FetchData() {

useEffect(() => {

fetch("https://api.example.com/data")

.then(response => response.json())

.then(data => console.log(data));

}, []);

return <h1>Fetching Data...</h1>;

}

**🔹 ComponentDidUpdate (Runs on Updates)**

| **Lifecycle Method** | **useEffect Equivalent** |
| --- | --- |
| componentDidUpdate() | useEffect(() => { /\* Side effect \*/ }, [dependency]); |

✔ Runs **after state or props change**.  
✔ Used for **updating DOM, sending API requests, responding to prop changes**.

**Example: Updating the Title When a Counter Changes (Class Component)**

javascript

CopyEdit

class Counter extends React.Component {

state = { count: 0 };

componentDidUpdate(prevProps, prevState) {

if (prevState.count !== this.state.count) {

document.title = `Count: ${this.state.count}`;

}

}

render() {

return (

<button onClick={() => this.setState({ count: this.state.count + 1 })}>

Click Me {this.state.count}

</button>

);

}

}

**Example: Updating the Title When a Counter Changes (useEffect() in Functional Component)**

javascript

CopyEdit

import { useState, useEffect } from 'react';

function Counter() {

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

useEffect(() => {

document.title = `Count: ${count}`;

}, [count]); // Runs only when count changes

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

}

**🔹 ComponentWillUnmount (Cleanup Before Unmounting)**

|  |  |
| --- | --- |
| **Lifecycle Method** | **useEffect Equivalent** |
| componentWillUnmount() | useEffect(() => { return () => { /\* Cleanup \*/ }; }, []); |

✔ Runs **before a component is removed from the DOM**.  
✔ Used for **cleaning up timers, event listeners, or subscriptions**.

**Example: Cleaning Up Event Listeners (Class Component)**

javascript

CopyEdit

class MouseTracker extends React.Component {

componentDidMount() {

window.addEventListener('mousemove', this.trackMouse);

}

componentWillUnmount() {

window.removeEventListener('mousemove', this.trackMouse);

}

trackMouse = (event) => {

console.log(`Mouse at: ${event.clientX}, ${event.clientY}`);

};

render() {

return <h1>Move your mouse!</h1>;

}

}

**Example: Cleaning Up Event Listeners (useEffect() in Functional Component)**

javascript

CopyEdit

import { useEffect } from 'react';

function MouseTracker() {

useEffect(() => {

const trackMouse = (event) => {

console.log(`Mouse at: ${event.clientX}, ${event.clientY}`);

};

window.addEventListener('mousemove', trackMouse);

return () => {

window.removeEventListener('mousemove', trackMouse);

};

}, []);

return <h1>Move your mouse!</h1>;

}

**4️⃣ Key Differences Between useEffect() and Lifecycle Methods**

|  |  |  |
| --- | --- | --- |
| **Feature** | **useEffect() (Functional)** | **Lifecycle Methods (Class)** |
| **Component Type** | Functional Component | Class Component |
| **Code Structure** | Concise & modular | More boilerplate |
| **Multiple Effects** | Can use multiple useEffect() calls | Single lifecycle methods per phase |
| **State Handling** | Uses useState() | Uses this.state |
| **Dependency Control** | Runs on specific dependencies | Runs on state/prop change |
| **Cleanup Handling** | Uses cleanup function in useEffect() | Uses componentWillUnmount() |
| **Performance** | More optimized | Less optimized |

**5️⃣ When to Use useEffect() vs Lifecycle Methods?**

✔ **Use useEffect()** if you’re working with **functional components** (modern approach).  
✔ **Use Lifecycle Methods** if you’re maintaining **class components** in legacy code.  
✔ **Convert class components to functional components** when possible for better maintainability.

**6️⃣ Summary & Takeaways**

✔ useEffect() **combines multiple lifecycle methods** (componentDidMount, componentDidUpdate, componentWillUnmount).  
✔ **Easier and cleaner syntax** than class lifecycle methods.  
✔ **Better separation of concerns** by using multiple useEffect() calls.  
✔ **Encourages functional programming** and is the **preferred way** in modern React.

🚀 **Now you understand how useEffect() replaces lifecycle methods! Start using it in your functional components.** 🎯

**Routing in React :-**

**Routing in React.js**

**1️⃣ Introduction to React Routing**

* **Routing** in React allows navigation between different views/pages in a single-page application (**SPA**).
* React does **not have built-in routing**, so we use a library called **React Router**.
* **React Router** helps in managing **URLs**, **dynamic paths**, and **navigation** without reloading the page.
* It provides **client-side routing**, improving user experience by **avoiding full-page reloads**.

**2️⃣ Installing React Router**

Before using React Router, install it using **npm** or **yarn**:

bash

CopyEdit

npm install react-router-dom

or

bash

CopyEdit

yarn add react-router-dom

**3️⃣ Basic Concepts in React Router**

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **BrowserRouter** | Enables routing using HTML5 history API (Used for web apps). |
| **Routes** | Defines different route paths inside the application. |
| **Route** | Renders a component when a specific URL matches. |
| **Link** | Replaces <a> tags for navigation without reloading. |
| **NavLink** | Similar to <Link>, but adds active styling when the link is active. |
| **useNavigate** | A hook for programmatic navigation. |
| **useParams** | A hook for accessing dynamic parameters in URLs. |
| **useLocation** | Provides information about the current URL. |
| **useHistory (Deprecated)** | Previously used for navigation; replaced with useNavigate. |

**4️⃣ Setting Up Routing in React**

**🔹 Basic Routing Example**

jsx

CopyEdit

import React from "react";

import { BrowserRouter, Routes, Route } from "react-router-dom";

import Home from "./Home";

import About from "./About";

import Contact from "./Contact";

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

<Route path="/contact" element={<Contact />} />

</Routes>

</BrowserRouter>

);

}

export default App;

✅ **Explanation:**  
✔ BrowserRouter wraps the application to enable routing.  
✔ Routes is used to define multiple Route components.  
✔ Each Route has a **path** ("/", "/about", "/contact") and an **element** to render.

**5️⃣ Navigation Using <Link>**

* Instead of using <a href="...">, React Router provides <Link> to prevent full-page reloads.

**🔹 Example: Using <Link> for Navigation**

jsx

CopyEdit

import { Link } from "react-router-dom";

function Navbar() {

return (

<nav>

<Link to="/">Home</Link> |

<Link to="/about">About</Link> |

<Link to="/contact">Contact</Link>

</nav>

);

}

export default Navbar;

✅ **Explanation:**  
✔ <Link> is used for navigation, replacing <a> to prevent reloading.  
✔ **Faster transitions** and **better user experience** compared to <a>.

**6️⃣ Active Navigation Using <NavLink>**

* NavLink is like Link but adds an **active class** when the link is selected.

**🔹 Example: Using <NavLink>**

jsx

CopyEdit

import { NavLink } from "react-router-dom";

function Navbar() {

return (

<nav>

<NavLink to="/" activeClassName="active">Home</NavLink> |

<NavLink to="/about" activeClassName="active">About</NavLink> |

<NavLink to="/contact" activeClassName="active">Contact</NavLink>

</nav>

);

}

export default Navbar;

✅ **Explanation:**  
✔ NavLink adds an **active class** automatically when the link is active.  
✔ Useful for highlighting the current page in the navigation menu.

**7️⃣ Dynamic Routing with URL Parameters**

* Use **:param** in routes to create **dynamic URLs**.

**🔹 Example: Using useParams for Dynamic Routing**

jsx

CopyEdit

import { useParams } from "react-router-dom";

function UserProfile() {

const { username } = useParams();

return <h1>Welcome, {username}!</h1>;

}

**🔹 Route Setup**

jsx

CopyEdit

<Routes>

<Route path="/user/:username" element={<UserProfile />} />

</Routes>

✅ **Explanation:**  
✔ If the user visits **/user/John**, it will display "Welcome, John!".  
✔ useParams() extracts username from the URL dynamically.

**8️⃣ Programmatic Navigation with useNavigate**

* useNavigate is used to navigate **programmatically** inside event handlers.

**🔹 Example: Redirecting on Button Click**

jsx

CopyEdit

import { useNavigate } from "react-router-dom";

function Home() {

const navigate = useNavigate();

function goToAbout() {

navigate("/about");

}

return <button onClick={goToAbout}>Go to About Page</button>;

}

export default Home;

✅ **Explanation:**  
✔ Clicking the button redirects the user to /about.  
✔ navigate("/about") works like <Link>, but inside a function.

**9️⃣ Handling 404 Pages (Not Found)**

* A 404 Page is shown when no routes match the given URL.

**🔹 Example: Handling 404 Page**

jsx

CopyEdit

function NotFound() {

return <h1>404 - Page Not Found</h1>;

}

<Routes>

<Route path="\*" element={<NotFound />} />

</Routes>

✅ **Explanation:**  
✔ The "\*" path matches **all undefined routes**.  
✔ Displays a custom 404 page when a user enters an unknown URL.

**🔟 Nested Routing**

* **Nested routes** allow components to be rendered inside parent components.

**🔹 Example: Nested Routes**

jsx

CopyEdit

<Routes>

<Route path="/dashboard" element={<Dashboard />}>

<Route path="settings" element={<Settings />} />

<Route path="profile" element={<Profile />} />

</Route>

</Routes>

✅ **Explanation:**  
✔ Visiting /dashboard/settings renders both **Dashboard** and **Settings** inside it.  
✔ Useful for **layouts with sidebars or dashboards**.

**🔟 Private Routes (Authentication)**

* Private routes require authentication before accessing them.

**🔹 Example: Creating a Private Route**

jsx

CopyEdit

import { Navigate } from "react-router-dom";

function PrivateRoute({ children }) {

const isAuthenticated = false; // Change to `true` if user is logged in

return isAuthenticated ? children : <Navigate to="/login" />;

}

**🔹 Usage of Private Route**

jsx

CopyEdit

<Routes>

<Route path="/dashboard" element={<PrivateRoute><Dashboard /></PrivateRoute>} />

</Routes>

✅ **Explanation:**  
✔ Redirects unauthorized users to the /login page.  
✔ Only logged-in users can access /dashboard.

**🔟 Handling Query Parameters**

* **Query parameters** store additional information in URLs (?key=value).

**🔹 Example: Using useLocation to Get Query Parameters**

jsx

CopyEdit

import { useLocation } from "react-router-dom";

function SearchPage() {

const location = useLocation();

const params = new URLSearchParams(location.search);

const searchTerm = params.get("q");

return <h1>Search Results for: {searchTerm}</h1>;

}

✅ **Explanation:**  
✔ Extracts ?q=value from the URL using useLocation().  
✔ /search?q=React will display "Search Results for: React".

**📝 Summary: Key Takeaways**

✔ **React Router enables client-side routing for SPAs**.  
✔ Use **<BrowserRouter>** to enable routing.  
✔ Use **<Routes>** and **<Route>** for defining paths.  
✔ Use **<Link>** and **<NavLink>** instead of <a> for navigation.  
✔ Use **useParams()** for dynamic URLs.  
✔ Use **useNavigate()** for programmatic navigation.  
✔ Use **useLocation()** for query parameters.  
✔ Use **Private Routes** to protect authenticated pages.  
✔ Implement **404 Pages** using "\*" routes.  
✔ Use **Nested Routes** for hierarchical navigation.

🚀 **Master React Router to create dynamic, fast, and user-friendly web applications!**