**Notes - React Bootcamp**

**Day 1: Introduction to React.js and Component Basics**

**Module 1: What is React.js?**

* **Introduction to React.js**
* Overview of React.js as a JavaScript library for building user interfaces

React.js is a **powerful JavaScript library** developed by **Facebook (Meta)** for building **fast, dynamic, and interactive user interfaces (UIs)**. It is widely used for **modern web applications**.

**1. What is React.js? 📌**

✔ **Component-Based** – UI is divided into **reusable** components.

✔ **Declarative** – Describe what the UI should look like; React handles the updates.

✔ **Virtual DOM** – Efficient updates for better performance.

✔ **One-Way Data Binding** – Controlled and predictable state management.

✔ **Used for Frontend Development** – Works with APIs, state management, and UI rendering.

**2. Why Use React.js? 🤔**

✅ **Faster Performance** – Uses a **Virtual DOM** to minimize updates.

✅ **Reusable Components** – Saves development time with modular design.

✅ **Strong Ecosystem** – Works with libraries like **Redux, React Router, and Next.js**.

✅ **SEO-Friendly** – Server-side rendering (SSR) with **Next.js**.

✅ **Large Community & Support** – Maintained by **Meta (Facebook)** and thousands of developers.

**3. How React Works? 🔄**

React follows a **component-based architecture**:

✔ **Components** – Small, reusable building blocks of UI.

✔ **Props (Properties)** – Pass data from parent to child components.

✔ **State** – Manages data **inside a component**.

**4. React.js Basic Syntax & Example 🔥**

**🔹 Installing React (Using create-react-app)**

npx create-react-app my-app

cd my-app

npm start

**🔹 Example: Basic React Component (App.js)**

import React from "react";

function App() {

return (

<div>

<h1>Hello, React! 🚀</h1>

<p>React makes building UIs easy and efficient.</p>

</div>

);

}

export default App;

📌 **Explanation:**

* + **JSX (<div>...</div>)** – Combines JavaScript + HTML.
  + **Component (function App())** – Returns UI structure.
  + **Export (export default App)** – Makes the component available for use.

**5. React’s Core Features 📌**

| **Feature** | **Description** |
| --- | --- |
| **Components** | Reusable building blocks of UI |
| **JSX (JavaScript XML)** | Allows writing HTML inside JavaScript |
| **State Management** | Maintains dynamic data using useState |
| **Props (Properties)** | Passes data from parent to child components |
| **Hooks** | Allows functional components to manage state (useState, useEffect) |
| **Virtual DOM** | Optimizes updates and improves performance |
| **React Router** | Handles navigation between pages |

**6. Summary 🚀**

✔ **React.js is a JavaScript library** for building fast, reusable, and dynamic UIs.

✔ Uses **component-based architecture** for better organization.

✔ Leverages **JSX, Virtual DOM, state, and props** for efficiency.

✔ Works well with tools like **Redux, Next.js, and React Router** for full web development.

📌 **React is the foundation of modern frontend development and powers apps like Facebook, Instagram, and Netflix!** 🎯

* Real-life Example: Websites like Facebook use React to create dynamic and interactive UIs.

**Module 2: Setting Up a React Project**

* **Creating a React App with Create React App (CRA)**
  + Install create-react-app and set up your first React project:

create-react-app is a **command-line tool** that helps developers quickly set up a **React.js project** with a pre-configured environment.

✅ No need to manually configure **Webpack, Babel, or ESLint**.

✅ Comes with **Hot Reloading**, so changes update automatically.

✅ Ideal for **beginners and production-ready** applications.

**1. Prerequisites 🛠️**

Before setting up a React project, make sure you have:

✔ **Node.js (v14 or higher)** installed → [Download Node.js](https://nodejs.org/)

✔ **npm (Node Package Manager)** installed (comes with Node.js)

👉 **Check if Node.js and npm are installed:**

node -v # Check Node.js version

npm -v # Check npm version

**2. Install create-react-app and Set Up Your First Project**

**Step 1: Create a New React App**

Open your **terminal/command prompt** and run:

npx create-react-app my-first-react-app

👉 This command:

* + - **Creates a new folder** named my-first-react-app.
    - **Installs all dependencies** automatically.

If npx is not working, install it globally:

npm install -g create-react-app

create-react-app my-first-react-app

**Step 2: Navigate to the Project Folder**

cd my-first-react-app

**Step 3: Start the Development Server**

npm start

👉 This will:

* + - Launch the React app in **localhost:3000**.
    - Automatically refresh when you make changes.

**3. Understanding the Project Structure**

my-first-react-app/

│── node\_modules/ # Installed dependencies

│── public/ # Static files (index.html, favicon, etc.)

│── src/ # Main React code (App.js, components)

│── .gitignore # Ignore unnecessary files in Git

│── package.json # Project dependencies & scripts

│── README.md # Project documentation

**4. Editing Your First React Component**

**Modify App.js to Display "Hello, React!"**

* + - Open src/App.js and replace the content with:

import React from "react";

function App() {

return (

<div>

<h1>Hello, React! 🎉</h1>

<p>My first React app is running!</p>

</div>

);

}

export default App;

* + - Save the file and check **localhost:3000**. 🎉

**5. Summary 🚀**

✔ Installed **React using create-react-app**.

✔ Set up a **new React project**.

✔ Started the **development server** (npm start).

✔ Modified the **App.js component** to display custom content.

* + Code Example:
* npx create-react-app my-first-react-app
* cd my-first-react-app
* npm start
  + Real-life Example: Creating a simple app setup to work like a workspace or kitchen for building your app.

**Module 3: React Components**

* **What are Components?**
  + React components are reusable pieces of UI.

**React Components: Reusable Pieces of UI 🔄🎨**

In **React.js**, components are the **building blocks** of the user interface. They allow developers to **break down the UI into smaller, reusable pieces**, making development **efficient and scalable**.

**1. What are React Components? 📌**

✔ A **component** is a **self-contained, reusable piece of UI**.

✔ Components **receive data (props)** and **manage state**.

✔ Components can be **nested**, meaning one component can contain another.

✔ React has **two types of components**:

* + - **Functional Components** (Recommended ✅)
    - **Class Components** (Older Approach)

**2. Functional Components (Modern & Recommended) 🚀**

A **functional component** is a JavaScript function that **returns JSX (UI structure)**.

**🔹 Example: Simple Functional Component**

import React from "react";

function Greeting() {

return <h1>Hello, React! 🎉</h1>;

}

export default Greeting;

📌 **Key Features:** ✔ Uses a simple **JavaScript function**.

✔ Returns **JSX (HTML-like syntax in JavaScript)**.

✔ Can be **exported and reused** in different parts of the app.

**🔹 Using a Functional Component in App.js**

import React from "react";

import Greeting from "./Greeting"; // Importing component

function App() {

return (

<div>

<Greeting /> {/\* Using the Greeting component \*/}

</div>

);

}

export default App;

📌 **Output:** Displays **"Hello, React! 🎉"** in the browser.

**3. Props: Passing Data to Components 🔄**

Props (**properties**) allow data to be passed from **parent to child components**.

**🔹 Example: Component with Props**

function Welcome(props) {

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

}

**🔹 Using the Component with Different Props**

function App() {

return (

<div>

<Welcome name="Alice" />

<Welcome name="Bob" />

</div>

);

}

📌 **Output:**

Welcome, Alice! 🎉

Welcome, Bob! 🎉

✅ **Props make components reusable with different data.**

**4. State: Managing Component Data 🛠️**

A component’s **state** is used to **store and update data dynamically**.

**🔹 Example: Component with useState Hook**

import React, { useState } from "react";

function Counter() {

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

return (

<div>

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

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

</div>

);

}

📌 **Explanation:** ✔ useState(0) initializes the state (count = 0).

✔ setCount(count + 1) updates the state.

✔ Clicking the button **increments** the count dynamically.

**5. Class Components (Older Approach) ⏳**

React originally used **class-based components**, but **functional components with hooks** (useState, useEffect) are now preferred.

import React, { Component } from "react";

class Welcome extends Component {

render() {

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

}

}

export default Welcome;

📌 **Note:**

* + - Requires this.props instead of just props.
    - Uses a render() method, making it **more complex** than functional components.

**6. Summary Table 📌**

| **Feature** | **Functional Component (✅ Recommended)** | **Class Component (⏳ Older)** |
| --- | --- | --- |
| **Syntax** | function Component() {} | class Component extends React.Component {} |
| **State Handling** | useState() (Hooks) | this.state (More complex) |
| **Lifecycle Methods** | useEffect() | componentDidMount(), componentDidUpdate() |
| **Code Complexity** | ✅ Simpler & Cleaner | ❌ More Boilerplate Code |
| **Performance** | ✅ Better Optimization | ❌ Slightly Slower |

**7. Best Practices for Using Components ✅**

✔ **Use functional components** for simplicity and better performance.

✔ **Break down UI into smaller components** to improve reusability.

✔ **Use props** to pass data between components.

✔ **Use hooks (useState, useEffect)** instead of class-based state management.

* + Code Example:
  + function Welcome() {
  + return <h1>Hello, World!</h1>;
  + }
  + export default Welcome;
  + **Real-life Example:** Components are like building blocks in LEGO, small reusable parts that come together to form a structure.

**Module 4: JSX – The Syntax of React**

* **Introduction to JSX**
  + JSX is a mix of HTML and JavaScript:

**JSX (JavaScript XML)** is a **syntax extension** for JavaScript that allows writing **HTML-like code inside JavaScript**. It is used in **React.js** to define UI components.

**1. What is JSX? 📌**

✔ **Looks like HTML**, but is actually JavaScript.

✔ **Makes UI code more readable** by keeping JavaScript and UI together.

✔ **Gets compiled to JavaScript** using Babel before running in the browser.

✔ **Allows embedding JavaScript inside {} within JSX.**

**🔹 Example: JSX vs. Traditional JavaScript**

**✅ With JSX (Recommended)**

function Greeting() {

return <h1>Hello, JSX! 🎉</h1>;

}

**❌ Without JSX (Using React.createElement)**

function Greeting() {

return React.createElement("h1", null, "Hello, JSX! 🎉");

}

📌 **JSX is much cleaner and easier to read!**

**2. Embedding JavaScript Inside JSX 🔄**

You can use **JavaScript expressions inside {}** in JSX.

**🔹 Example: Using Variables in JSX**

const name = "Alice";

function Greeting() {

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

}

📌 **Inside {} you can use:** ✔ **Variables ({name})**

✔ **Functions ({getGreeting()})**

✔ **Math expressions ({5 + 5})**

**3. JSX Rules & Best Practices 📏**

**🔹 1. JSX Must Have One Parent Element**

❌ **Invalid JSX (Multiple root elements)**

return (

<h1>Hello</h1>

<p>Welcome!</p>

);

✅ **Fix: Wrap everything in a <div> or <> (Fragment)**

return (

<>

<h1>Hello</h1>

<p>Welcome!</p>

</>

);

**🔹 2. Use className Instead of class (JSX Uses JavaScript Syntax)**

❌ **Incorrect (HTML style)**

<p class="text-bold">Hello</p>

✅ **Correct (JSX style)**

<p className="text-bold">Hello</p>

**🔹 3. Self-Closing Tags Must End with /**

❌ **Incorrect**

<img src="logo.png">

✅ **Correct**

<img src="logo.png" />

**🔹 4. Inline Styles Use Objects**

❌ **Incorrect (HTML style)**

<p style="color: red;">Hello</p>

✅ **Correct (JSX style)**

<p style={{ color: "red" }}>Hello</p>

**4. Conditional Rendering in JSX ⚡**

You can use **ternary operators (? :)** or **logical operators (&&)** inside JSX.

**🔹 Example: Conditional Rendering**

const isLoggedIn = true;

function Welcome() {

return (

<div>

{isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in</h1>}

</div>

);

}

📌 **Shorter Alternative (&& Operator)**

{isLoggedIn && <h1>Welcome back!</h1>}

✅ **Displays <h1> only if isLoggedIn is true.**

**5. Summary 🚀**

| **Feature** | **JSX Usage** | **Notes** |
| --- | --- | --- |
| **Embedding JavaScript** | {expression} | Use {} to insert JavaScript inside JSX |
| **Multiple Elements** | <></> | Wrap JSX in a single parent (div or <>) |
| **Class Attribute** | className="..." | Use className instead of class |
| **Inline Styles** | style={{ color: "red" }} | Use double curly braces ({}) |
| **Conditional Rendering** | {isLoggedIn ? <h1>Hi</h1> : <h1>Bye</h1>} | Use ternary or && |

* + Code Example:
* const name = "John";
* const element = <h1>Hello, {name}!</h1>;
  + Real-life Example: Like writing a recipe card where you mix ingredients (HTML) and instructions (JavaScript) in one place.

**Day 2: State, Props, and Event Handling**

**Module 1: Understanding Props**

* **What are Props?**
  + Props are arguments passed into components.

**Props (short for "properties")** are **arguments passed to React components**. They allow components to be **dynamic, reusable, and configurable**.

**1. What Are Props? 📌**

✔ **Props pass data from a parent component to a child component.**

✔ **Props are read-only** (cannot be modified inside the child component).

✔ **Passed as attributes** inside the component tag.

**2. Passing Props to Functional Components 🎯**

**🔹 Example: Passing a Name Prop**

function Welcome(props) {

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

}

function App() {

return (

<div>

<Welcome name="Alice" />

<Welcome name="Bob" />

</div>

);

}

export default App;

📌 **How It Works:**

* + - <Welcome name="Alice" /> → **Passes "Alice" as a prop** to the Welcome component.
    - Inside Welcome(), **props.name** receives "Alice".
    - Output:
    - Welcome, Alice! 🎉
    - Welcome, Bob! 🎉

**3. Using Props with Multiple Values 🔄**

**🔹 Example: Passing Multiple Props**

function UserCard(props) {

return (

<div>

<h2>Name: {props.name}</h2>

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

</div>

);

}

function App() {

return (

<div>

<UserCard name="Alice" age={25} />

<UserCard name="Bob" age={30} />

</div>

);

}

export default App;

📌 **Props can be strings ("Alice"), numbers (25), or even objects and functions!**

**4. Destructuring Props for Cleaner Code 🧹**

Instead of using props.name, you can **destructure props**.

function UserCard({ name, age }) {

return (

<div>

<h2>Name: {name}</h2>

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

</div>

);

}

📌 **Benefits of Destructuring:**

✅ **Cleaner syntax**.

✅ No need to write props. multiple times.

**5. Default Props: Providing Default Values ✅**

If a prop is missing, you can set a **default value**.

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

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

}

<Welcome /> // Output: "Welcome, Guest!"

📌 **This prevents undefined values when a prop is missing.**

**6. Props in Class Components (Older Approach) ⏳**

Before React Hooks, props were used in **class components** like this:

class Welcome extends React.Component {

render() {

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

}

}

📌 **Modern React uses functional components (useState, useEffect) instead of class components.**

**7. Passing Functions as Props (Event Handling) 🎯**

Props can **pass functions** to child components to handle events.

**🔹 Example: Parent-Child Communication**

function Button({ handleClick }) {

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

}

function App() {

const showAlert = () => alert("Button clicked!");

return <Button handleClick={showAlert} />;

}

📌 **Clicking the button triggers the showAlert function in the parent component.**

**8. Summary Table 📌**

| **Feature** | **Description** | **Example** |
| --- | --- | --- |
| **Passing Props** | Send data from parent to child | <Child name="Alice" /> |
| **Using Props** | Access inside child component | props.name or { name } |
| **Destructuring** | Cleaner prop handling | function Child({ name }) |
| **Default Props** | Set fallback values | name = "Guest" |
| **Function as Prop** | Pass function from parent to child | handleClick={showAlert} |

* + Code Example:
  + function Greeting(props) {
  + return <h1>Hello, {props.name}!</h1>;
  + }
  + function App() {
  + return <Greeting name="Alice" />;
  + }
  + export default App;
  + Real-life Example: Props are like ingredients being passed to different stations in a kitchen.

**Module 2: Introduction to State**

* **What is State?**
  + State allows React components to change and re-render based on data.

In **React**, **state** is a built-in feature that allows components to store **dynamic data** and re-render when the data changes.

**1. What is State? 📌**

✔ **State stores component-specific data** that can change over time.

✔ **Changes in state trigger a re-render**, updating the UI dynamically.

✔ **State is managed within the component**, unlike props (which are passed from parent to child).

✔ **Controlled using the useState hook** in functional components.

**2. Using State in Functional Components (useState) 🎯**

The useState hook is used to **define and update state** in functional components.

**🔹 Example: Simple Counter with useState**

import React, { useState } from "react";

function Counter() {

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

return (

<div>

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

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

</div>

);

}

export default Counter;

📌 **Explanation:**

* + - useState(0) initializes the state variable count with 0.
    - setCount(count + 1) updates the state when the button is clicked.
    - React **automatically re-renders** the component when state changes.

**3. Understanding useState Syntax 📜**

const [stateVariable, setStateFunction] = useState(initialValue);

✔ stateVariable → Holds the current value of the state.

✔ setStateFunction → Updates the state and triggers a re-render.

✔ initialValue → The starting value of the state.

**4. State in Class Components (Older Approach) ⏳**

Before React Hooks, state was managed using this.state in **class components**.

**🔹 Example: Counter Using Class Component**

import React, { Component } from "react";

class Counter extends Component {

constructor() {

super();

this.state = { count: 0 };

}

increment = () => {

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

};

render() {

return (

<div>

<h2>Count: {this.state.count}</h2>

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

</div>

);

}

}

export default Counter;

📌 **Why Functional Components (useState) Are Better?**

✅ **Less boilerplate code**

✅ **Easier to read and maintain**

✅ **Encouraged in modern React development**

**5. Updating State Correctly ⚠️**

**🔹 Using Functional Updates (Recommended for Previous State)**

setCount(prevCount => prevCount + 1);

📌 **Why?**

* + - Ensures **correct updates** when state changes rapidly (e.g., multiple clicks).

**🔹 Example: Correct vs. Incorrect Updates**

<button onClick={() => setCount(count + 1)}>Increment</button> // ❌ Might cause stale updates

<button onClick={() => setCount(prev => prev + 1)}>Increment</button> // ✅ Always correct

**6. Managing Multiple State Variables**

const [user, setUser] = useState({ name: "Alice", age: 25 });

setUser({ ...user, age: 26 }); // ✅ Update age while keeping name

📌 **Use the spread operator (...user)** to keep existing state values.

**7. Summary 🚀**

| **Feature** | **Functional Component (useState)** | **Class Component (this.state)** |
| --- | --- | --- |
| **Syntax** | const [state, setState] = useState(initialValue) | this.state = { key: value } |
| **Updating State** | setState(newValue) | this.setState({ key: value }) |
| **Re-Renders Component?** | ✅ Yes | ✅ Yes |
| **Best Practice?** | ✅ Recommended | ❌ Older method |

* + Code Example using useState:
  + import React, { useState } from "react";
  + function Counter() {
  + const [count, setCount] = useState(0);
  + return (
  + <div>
  + <p>You clicked {count} times</p>
  + <button onClick={() => setCount(count + 1)}>Click me</button>
  + </div>
  + );
  + }
  + export default Counter;
  + Real-life Example: State is like a shopping cart that changes as you add items.

**Module 3: Handling Events in React**

* **Event Handling**
  + Handling user interactions like button clicks.

React provides a simple way to handle **user interactions** like **button clicks, form submissions, and input changes** using **event handlers**.

**1. Handling Click Events 📌**

In React, event handlers are defined using **camelCase** and passed as functions.

**🔹 Example: Handling a Button Click**

import React from "react";

function App() {

function handleClick() {

alert("Button Clicked!");

}

return (

<div>

<button onClick={handleClick}>Click Me</button>

</div>

);

}

export default App;

📌 **How It Works:**

* + - The handleClick function **runs when the button is clicked**.
    - The event listener is written as **onClick={handleClick}** instead of HTML’s onclick.

**2. Handling Click Events with useState 🔄**

We can use **state (useState)** to update the UI when a button is clicked.

**🔹 Example: Counter Button**

import React, { useState } from "react";

function Counter() {

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

return (

<div>

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

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

</div>

);

}

export default Counter;

📌 **How It Works:**

* + - useState(0) initializes count = 0.
    - Clicking the button **increments count**, and React **re-renders the UI**.

**3. Passing Arguments to Event Handlers 🎯**

If an event handler needs **arguments**, use an arrow function inside onClick.

**🔹 Example: Passing a Parameter to handleClick**

function App() {

function handleClick(name) {

alert(`Hello, ${name}!`);

}

return (

<button onClick={() => handleClick("Alice")}>Greet</button>

);

}

📌 **Why Use an Arrow Function?**

* + - onClick={handleClick("Alice")} would **execute immediately** instead of waiting for a click.
    - onClick={() => handleClick("Alice")} **defers execution** until clicked.

**4. Event Object (event) in React ⚡**

React provides an **event object (event)** to access details about the interaction.

**🔹 Example: Logging Button Clicks**

function App() {

function handleClick(event) {

console.log("Event Type:", event.type);

console.log("Button Text:", event.target.innerText);

}

return (

<button onClick={handleClick}>Click Me</button>

);

}

📌 **The event object provides details like:**

✔ event.type → "click"

✔ event.target.innerText → "Click Me"

**5. Handling Multiple Buttons in a List 🔄**

When handling multiple buttons, use **map()** to generate buttons dynamically.

**🔹 Example: Handling Clicks for Multiple Items**

function App() {

const items = ["Apple", "Banana", "Cherry"];

function handleClick(item) {

alert(`You selected: ${item}`);

}

return (

<div>

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

<button key={index} onClick={() => handleClick(item)}>

{item}

</button>

))}

</div>

);

}

📌 **Now each button will show an alert with its corresponding fruit name.**

**6. Disabling a Button After Click 🛑**

import React, { useState } from "react";

function App() {

const [disabled, setDisabled] = useState(false);

return (

<button onClick={() => setDisabled(true)} disabled={disabled}>

{disabled ? "Clicked!" : "Click Me"}

</button>

);

}

📌 **Once clicked, the button becomes disabled.**

**7. Summary Table 📌**

| **Feature** | **Example** | **Description** |
| --- | --- | --- |
| **Basic Click Event** | onClick={handleClick} | Calls a function when clicked |
| **Updating State** | setCount(count + 1) | Changes UI dynamically |
| **Passing Arguments** | onClick={() => handleClick("Alice")} | Passes data on click |
| **Using Event Object** | onClick={(event) => console.log(event.target)} | Access event details |
| **Multiple Buttons** | items.map((item) => <button onClick={() => handleClick(item)}> | Handles clicks dynamically |
| **Disable Button** | <button disabled={disabled}> | Prevents further clicks |

🚀 **React makes handling user interactions easy, dynamic, and efficient!**

* + Code Example:
  + function Button() {
  + const handleClick = () => {
  + alert("Button was clicked!");
  + };
  + return <button onClick={handleClick}>Click Me!</button>;
  + }
  + export default Button;
  + Real-life Example: A button that adds an item to a shopping cart.

**Module 4: Conditional Rendering**

* **Conditional Rendering in JSX**
  + Rendering components based on a condition.

In React, we often need to **conditionally render** components based on **state, props, or logic**. There are multiple ways to handle this.

**1. Using if Statements (Inside a Function) ✅**

The traditional way to conditionally render components is by using an if statement inside a function.

**🔹 Example: Showing a Login Button Based on Authentication**

function AuthButton({ isLoggedIn }) {

if (isLoggedIn) {

return <button>Logout</button>;

} else {

return <button>Login</button>;

}

}

function App() {

return <AuthButton isLoggedIn={true} />;

}

export default App;

📌 **Explanation:**

* + - If isLoggedIn is true, it renders the "Logout" button.
    - Otherwise, it renders the "Login" button.

**2. Using the Ternary Operator (? :) ⚡**

A more compact way to conditionally render components inside JSX.

**🔹 Example: Conditional Text**

function Greeting({ isLoggedIn }) {

return (

<h1>{isLoggedIn ? "Welcome back!" : "Please sign in."}</h1>

);

}

function App() {

return <Greeting isLoggedIn={false} />;

}

📌 **Explanation:**

* + - If isLoggedIn is true, it displays "Welcome back!".
    - Otherwise, it shows "Please sign in.".

✅ **Best for short conditional logic inside JSX.**

**3. Using the && Operator (Short-Circuit Rendering) 🚀**

Use **logical &&** when you only need to render something **if a condition is true**.

**🔹 Example: Show a Notification If There’s a Message**

function Notification({ hasMessage }) {

return (

<div>

<h1>Dashboard</h1>

{hasMessage && <p>You have new messages!</p>}

</div>

);

}

function App() {

return <Notification hasMessage={true} />;

}

📌 **Explanation:**

* + - If hasMessage is true, the <p> message is displayed.
    - If false, nothing renders.

✅ **Best for simple one-condition rendering.**

**4. Using return null (Hiding Components) 🎭**

If you want to **conditionally hide a component**, return null.

**🔹 Example: Hiding a Warning Component**

function Warning({ showWarning }) {

if (!showWarning) {

return null;

}

return <p>⚠️ Warning: Something went wrong!</p>;

}

function App() {

return <Warning showWarning={false} />;

}

📌 **Why Use null?**

✅ **Prevents unnecessary DOM elements.**

✅ **Improves performance.**

**5. Conditional Rendering Based on State (useState) 🔄**

Using **state (useState)** to dynamically change UI.

**🔹 Example: Toggle Visibility**

import React, { useState } from "react";

function ToggleText() {

const [isVisible, setIsVisible] = useState(true);

return (

<div>

<button onClick={() => setIsVisible(!isVisible)}>

{isVisible ? "Hide" : "Show"}

</button>

{isVisible && <p>This text is toggled!</p>}

</div>

);

}

export default ToggleText;

📌 **Explanation:**

* + - Clicking the button **toggles isVisible between true and false**.
    - The <p> **only renders when isVisible is true**.

✅ **Great for interactive UI elements like modals and dropdowns.**

**6. Summary Table 📌**

| **Method** | **Best For** | **Example** |
| --- | --- | --- |
| **if statement** | Complex conditions | {if (isLoggedIn) { return <Logout /> } else { return <Login /> }} |
| **Ternary ? :** | Simple conditions | {isLoggedIn ? <Logout /> : <Login />} |
| **Logical &&** | Rendering if true | {hasMessage && <p>You have new messages!</p>} |
| **return null** | Hiding a component | {!showWarning && <Warning />} |
| **State (useState)** | Dynamic changes | setIsVisible(!isVisible) |

* + Code Example:
  + function UserGreeting(props) {
  + return <h1>Welcome back!</h1>;
  + }
  + function GuestGreeting(props) {
  + return <h1>Please sign up.</h1>;
  + }
  + function Greeting(props) {
  + const isLoggedIn = props.isLoggedIn;
  + if (isLoggedIn) {
  + return <UserGreeting />;
  + }
  + return <GuestGreeting />;
  + }
  + function App() {
  + return <Greeting isLoggedIn={false} />;
  + }
  + export default App;
  + Real-life Example: Displaying a "Logged In" or "Guest" message based on the user’s login status.

**Day 3: Lists, Forms, and Lifecycle Methods**

**Module 1: Rendering Lists**

* **Rendering Lists in React**
  + Using .map() to render a list of items.

The .map() method in JavaScript allows React to **dynamically render a list of items** by iterating over an array and returning a React element for each item.

**1. Why Use .map() in React? 📌**

✔ **Dynamically generates components** from an array.

✔ **Prevents repetitive code** (no need to manually write each <li>).

✔ **Improves performance** by efficiently updating the UI.

**2. Basic Example: Rendering a List of Items 📝**

function ItemList() {

const items = ["Apple", "Banana", "Cherry"];

return (

<ul>

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

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

))}

</ul>

);

}

export default ItemList;

📌 **Explanation:**

* + - .map() iterates over the items array.
    - Each item is wrapped in <li> and displayed inside <ul>.
    - The **key prop** helps React track items efficiently.

**3. Using .map() to Render an Array of Objects 📋**

If the list contains **objects**, extract properties inside .map().

function UsersList() {

const users = [

{ id: 1, name: "Alice", age: 25 },

{ id: 2, name: "Bob", age: 30 },

{ id: 3, name: "Charlie", age: 22 }

];

return (

<ul>

{users.map(user => (

<li key={user.id}>

{user.name} - {user.age} years old

</li>

))}

</ul>

);

}

export default UsersList;

📌 **Explanation:**

* + - **Each object has an id, name, and age**.
    - We pass user.id as the **unique key prop** to improve performance.
    - This method is **better than using the index as a key**.

**4. Rendering Components Dynamically with .map() 🎨**

Instead of returning simple <li> elements, we can return **custom components**.

**🔹 Example: Rendering a List of Users with a UserCard Component**

function UserCard({ name, age }) {

return (

<div className="user-card">

<h3>{name}</h3>

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

</div>

);

}

function UsersList() {

const users = [

{ id: 1, name: "Alice", age: 25 },

{ id: 2, name: "Bob", age: 30 }

];

return (

<div>

{users.map(user => (

<UserCard key={user.id} name={user.name} age={user.age} />

))}

</div>

);

}

export default UsersList;

📌 **How It Works:**

* + - The UserCard component displays **each user's name and age**.
    - .map() dynamically renders a UserCard for **each user** in the array.
    - The **unique key={user.id}** helps React optimize re-renders.

**5. Best Practices for Using .map() ✅**

✔ **Always provide a unique key** when mapping over lists.

✔ **Avoid using index as a key** unless the list never changes.

✔ **Break large components into smaller reusable components**.

✔ **Ensure all list items return valid JSX elements**.

**6. Summary Table 📌**

| **Use Case** | **Example** |
| --- | --- |
| **Basic List Rendering** | {items.map((item, index) => <li key={index}>{item}</li>)} |
| **Mapping Objects** | {users.map(user => <li key={user.id}>{user.name}</li>)} |
| **Using Components in .map()** | {users.map(user => <UserCard key={user.id} name={user.name} />)} |
| **Keys in Lists** | <li key={user.id}>...</li> (✅ Best) vs. <li key={index}>...</li> (❌ Avoid) |

* + Code Example:
  + function ItemList() {
  + const items = ['Apple', 'Banana', 'Orange'];
  + return (
  + <ul>
  + {items.map((item, index) => (
  + <li key={index}>{item}</li>
  + ))}
  + </ul>
  + );
  + }
  + export default ItemList;
  + Real-life Example: Displaying a list of items in a grocery app.

**Module 2: Building Forms in React**

* **Handling Form Input**

Forms are essential for user interactions, such as collecting input, authentication, and searching. In React, **controlled components** ensure that form elements are managed through **state**.

**1. Controlled Components (Recommended) ✅**

A **controlled component** is a form element whose value is controlled by **React state**.

**🔹 Example: Handling Text Input**

import React, { useState } from "react";

function TextInputForm() {

const [text, setText] = useState(""); // State for input

function handleChange(event) {

setText(event.target.value); // Update state

}

function handleSubmit(event) {

event.preventDefault(); // Prevent page refresh

alert(`Submitted: ${text}`);

}

return (

<form onSubmit={handleSubmit}>

<input type="text" value={text} onChange={handleChange} />

<button type="submit">Submit</button>

</form>

);

}

export default TextInputForm;

📌 **Explanation:**

✔ **value={text}** → Controlled by state.

✔ **onChange={handleChange}** → Updates state on input.

✔ **handleSubmit prevents page reload (event.preventDefault()).**

**2. Handling Multiple Inputs 📋**

When a form has multiple fields, use a **single state object**.

**🔹 Example: Managing Multiple Fields**

import React, { useState } from "react";

function MultiInputForm() {

const [formData, setFormData] = useState({ name: "", email: "" });

function handleChange(event) {

setFormData({

...formData,

[event.target.name]: event.target.value

});

}

function handleSubmit(event) {

event.preventDefault();

alert(`Name: ${formData.name}, Email: ${formData.email}`);

}

return (

<form onSubmit={handleSubmit}>

<input name="name" type="text" value={formData.name} onChange={handleChange} placeholder="Name" />

<input name="email" type="email" value={formData.email} onChange={handleChange} placeholder="Email" />

<button type="submit">Submit</button>

</form>

);

}

export default MultiInputForm;

📌 **How It Works:**

* + **Use a single state (formData)** for multiple fields.
  + **Dynamically update** state based on event.target.name.
  + Prevents **creating separate states for each field**.

**3. Handling Checkboxes & Radio Buttons ✅**

**🔹 Example: Checkbox Input**

function CheckboxForm() {

const [isChecked, setIsChecked] = useState(false);

function handleChange(event) {

setIsChecked(event.target.checked); // Boolean value

}

return (

<label>

<input type="checkbox" checked={isChecked} onChange={handleChange} />

Accept Terms

</label>

);

}

📌 **checked={isChecked}** ensures React controls the checkbox.

**🔹 Example: Radio Button Input**

function RadioForm() {

const [gender, setGender] = useState("");

return (

<div>

<label>

<input type="radio" value="Male" checked={gender === "Male"} onChange={(e) => setGender(e.target.value)} />

Male

</label>

<label>

<input type="radio" value="Female" checked={gender === "Female"} onChange={(e) => setGender(e.target.value)} />

Female

</label>

</div>

);

}

📌 **Radio buttons require checked={value === selectedValue} to work properly.**

**4. Handling Select Dropdowns ⬇️**

function SelectForm() {

const [selected, setSelected] = useState("apple");

return (

<select value={selected} onChange={(e) => setSelected(e.target.value)}>

<option value="apple">Apple</option>

<option value="banana">Banana</option>

<option value="cherry">Cherry</option>

</select>

);

}

📌 **value={selected} ensures controlled behavior.**

**5. Uncontrolled Components (Using useRef()) ⏳**

Instead of managing form state with useState(), we can use **useRef()** for **direct DOM access**.

**🔹 Example: Using useRef()**

import React, { useRef } from "react";

function UncontrolledForm() {

const inputRef = useRef(null);

function handleSubmit(event) {

event.preventDefault();

alert(`Entered: ${inputRef.current.value}`);

}

return (

<form onSubmit={handleSubmit}>

<input ref={inputRef} type="text" />

<button type="submit">Submit</button>

</form>

);

}

📌 **Use useRef() when you don't need state updates but want to access the input field directly.**

**6. Summary Table 📌**

| **Feature** | **Example** |
| --- | --- |
| **Controlled Input (Text)** | <input type="text" value={state} onChange={handleChange} /> |
| **Multiple Fields** | {...formData, [event.target.name]: event.target.value} |
| **Checkbox** | <input type="checkbox" checked={isChecked} onChange={handleChange} /> |
| **Radio Button** | <input type="radio" value="Male" checked={gender === "Male"} onChange={handleChange} /> |
| **Select Dropdown** | <select value={selected} onChange={handleChange}>...</select> |
| **Uncontrolled Input** | <input ref={inputRef} type="text" /> |

* Code Example:
* Real-life Example: A registration form for signing up for an event.

**Module 3: Introduction to React Lifecycle**

* **React Component Lifecycle**
  + Introduction to useEffect hook.

The **useEffect hook** in React **performs side effects** in functional components, such as **fetching data, updating the DOM, setting up event listeners, and handling timers**.

**1. What is useEffect? 📌**

✔ **Allows functional components to handle side effects** (like lifecycle methods in class components).

✔ Runs **after the component renders**.

✔ Can be used for **fetching API data, subscriptions, event listeners, timers, etc.**

✔ **Replaces lifecycle methods** like componentDidMount, componentDidUpdate, and componentWillUnmount.

**2. Basic Syntax of useEffect 🎯**

import React, { useEffect } from "react";

function MyComponent() {

useEffect(() => {

console.log("Component Mounted!");

});

return <h1>Hello, useEffect!</h1>;

}

export default MyComponent;

📌 **How It Works:**

* + - **Runs after every render**.
    - Prints "Component Mounted!" to the console.
    - **No dependencies**, so it runs after every re-render.

**3. Controlling When useEffect Runs ✅**

**🔹 1. useEffect with No Dependency Array (Runs on Every Render)**

useEffect(() => {

console.log("Runs after every render");

});

📌 **This runs after every render and re-render.**

**🔹 2. useEffect with an Empty Dependency Array (Runs Only Once)**

useEffect(() => {

console.log("Runs only on mount");

}, []);

📌 **Runs only once when the component mounts, similar to componentDidMount().**

**🔹 3. useEffect with Dependencies (Runs When Dependencies Change)**

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

useEffect(() => {

console.log(`Count changed to ${count}`);

}, [count]);

📌 **Runs only when count changes, similar to componentDidUpdate().**

**4. Using useEffect for API Calls 🌐**

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

function DataFetcher() {

const [data, setData] = useState([]);

useEffect(() => {

fetch("<https://jsonplaceholder.typicode.com/posts>")

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

.then(json => setData(json));

}, []); // Runs once on mount

return (

<ul>

{data.slice(0, 5).map(item => (

<li key={item.id}>{item.title}</li>

))}

</ul>

);

}

export default DataFetcher;

📌 **Why Use useEffect for API Calls?**

✔ Ensures data is fetched **after component mounts**.

✔ Prevents **unnecessary re-fetching**.

✔ Keeps UI **updated with new data**.

**5. Cleaning Up Side Effects (componentWillUnmount) 🧹**

If useEffect creates a **subscription, timer, or event listener**, it should be cleaned up to **prevent memory leaks**.

**🔹 Example: Cleanup on Component Unmount**

useEffect(() => {

const interval = setInterval(() => {

console.log("Running...");

}, 1000);

return () => {

clearInterval(interval); // Cleanup function

};

}, []);

📌 **Why Cleanup?**

✔ Prevents **memory leaks**.

✔ Ensures **no unnecessary background tasks** remain.

**6. Summary Table 📌**

| **Use Case** | **useEffect Code** | **Runs When?** |
| --- | --- | --- |
| **On Every Render** | useEffect(() => { ... }); | Every time component renders |
| **On Mount Only** | useEffect(() => { ... }, []); | Once when the component mounts |
| **On State Change** | useEffect(() => { ... }, [state]); | When state changes |
| **Cleanup on Unmount** | useEffect(() => { return () => { cleanup(); } }, []); | Before component unmounts |

* + Code Example:
  + import React, { useEffect, useState } from "react";
  + function DataFetcher() {
  + const [data, setData] = useState(null);
  + useEffect(() => {
  + fetch("<https://api.example.com/data>")
  + .then((response) => response.json())
  + .then((data) => setData(data));
  + }, []);
  + return (
  + <div>
  + <h1>Fetched Data:</h1>
  + <pre>{JSON.stringify(data, null, 2)}</pre>
  + </div>
  + );
  + }
  + export default DataFetcher;
  + Real-life Example: Fetching and displaying data when the component loads.

**Module 4: Mini Project To-Do List App**

In this mini project, students will build a basic to-do list app to reinforce the concepts learned over the three days. This project includes components, state, props, event handling, and conditional rendering.

**Project Features:**

* **Add New Task:** User can input a task and add it to the list.
* **Mark Task as Completed:** Each task has a button to mark it as complete.
* **Delete Task:** User can delete a task from the list.

**Code Example for To-Do List App:**

import React, { useState } from "react";

function TodoApp() {

const [tasks, setTasks] = useState([]);

const [input, setInput] = useState("");

const addTask = () => {

if (input) {

setTasks([...tasks, { text: input, completed: false }]);

setInput("");

}

};

const toggleComplete = (index) => {

const newTasks = [...tasks];

newTasks[index].completed = !newTasks[index].completed;

setTasks(newTasks);

};

const deleteTask = (index) => {

const newTasks = tasks.filter((task, i) => i !== index);

setTasks(newTasks);

};

return (

<div>

<h1>To-Do List</h1>

<input

type="text"

value={input}

onChange={(e) => setInput(e.target.value)}

placeholder="Add new task"

/>

<button onClick={addTask}>Add Task</button>

<ul>

{tasks.map((task, index) => (

<li key={index}>

<span

style={{ textDecoration: task.completed ? "line-through" : "none" }}

>

{task.text}

</span>

<button onClick={() => toggleComplete(index)}>

{task.completed ? "Undo" : "Complete"}

</button>

<button onClick={() => deleteTask(index)}>Delete</button>

</li>

))}

</ul>

</div>

);

}

export default TodoApp;