REACT TUTORIAL

📘 Chapter 1: Introduction to React and Setup

🔹 *What is React?*

React is a JavaScript library for building **user interfaces**—especially single-page applications. It lets you create reusable UI components.

🔹 *Why React?*

- Reusable components
- Fast rendering with Virtual DOM
- Huge community support

🔧 Setting Up React (Using Create React App)

1. Install Node.js

Download from: https://nodejs.org

2. Create your first React app:

Open your terminal/command prompt:

```
npx create-react-app my-first-app
cd my-first-app
npm start
```

This will start the development server and open your app in the browser.

🔍 What You See in the App

Inside `my-first-app`, the folder structure looks like:

```
my-first-app/
├── public/
├── src/
│   ├── App.js     # Your main component
│   ├── index.js   # Entry point
├── package.json
```

When the app runs, it displays content from `App.js`.

🩹 Try Editing Your App

In `src/App.js`, replace everything with:

```
function App() {
  return (
    <div>
      <h1>Hello, React Beginner!</h1>
      <p>This is your first React component.</p>
    </div>
  );
}

export default App;
```

Save the file — the browser updates automatically!

✅ **Your Task Before Chapter 2**:

- Make sure you can run a React app
- Replace the content of `App.js` with your own text

📘 Chapter 2: JSX and Components (Detailed)

🔷 JSX (JavaScript XML)

🧠 *What is JSX?*

JSX lets you write **HTML-like code inside JavaScript**. React uses it to **describe what the UI should look like**.

Instead of:

```
const element = React.createElement("h1", null, "Hello React");
```

You can simply write:

```
const element = <h1>Hello React</h1>;
```

This is easier to read and write!

📌 JSX Basic Rules

Rule	Example	Notes
Must return a single parent	✅ `<div><h1>Hello</h1><p>Welcome</p></div>`	❌ Multiple sibling elements without a wrapper will throw an error
Use `className` instead of `class`	`<div className="box">`	`class` is a reserved JS keyword
Use `{}` for expressions	`{2 + 2}`, `{name}`	You can write any valid JS inside `{}`

🩹 *Try This:*

```
function App() {
  const name = "React Beginner";
  const age = 20;

  return (
    <div>
      <h1>Welcome, {name}!</h1>
      <p>You are {age} years old.</p>
      <p>2 + 3 = {2 + 3}</p>
    </div>
  );
}
```

This will render:

```
Welcome, React Beginner!
You are 20 years old.
2 + 3 = 5
```

◇ What is a Component?

📦 *Think of components like:*

- Lego blocks 🧱: Each block is reusable.
- HTML templates 🏭: You can pass different data to each.

🧠 *Components help:*

- Organize your code
- Reuse logic/UI
- Keep code readable and manageable

🔧 Two Types of Components

1. Functional Component (Recommended)

```
function Greeting() {
  return <h2>Hello from Greeting Component!</h2>;
}
```

2. Class Component (Less common now)

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

class Greeting extends Component {
  render() {
    return <h2>Hello from Class Component!</h2>;
  }
}
```

💡 For beginners, stick to **functional components** using `function`.

🔨 Create and Use a Custom Component

Step-by-Step:

📄 Create a new file: `Message.js`

```
function Message() {
  return <p>This is a message from another component.</p>;
}

export default Message;
```

📂 Your folder structure:

```
src/
├── App.js
├── Message.js
```

🔗 Now, import it in App.js

```
import Message from "./Message";

function App() {
  return (
    <div>
      <h1>Hello React!</h1>
      <Message />
    </div>
  );
}
```

🖥️ Output:

```
Hello React!
This is a message from another component.
```

📦 You Can Create Many Components

```
// Header.js
function Header() {
  return <h2>This is the Header</h2>;
}
export default Header;

// Footer.js
function Footer() {
  return <h4>Footer Section</h4>;
}
```

```
export default Footer;
```

Then in `App.js`:

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

function App() {
  return (
    <div>
      <Header />
      <p>Main content goes here</p>
      <Footer />
    </div>
  );
}
```

☑ Summary

Term	Meaning
JSX	Syntax to write HTML inside JS
Component	A function that returns JSX
Functional Component	Simpler way to write components using `function`
Reusability	Components can be reused anywhere

🏆 Your Mini Challenge

1. Create a new component called `Bio.js`
2. It should return your name, age, and favorite language
3. Import and use it in `App.js`

Example:

```
// Bio.js
function Bio() {
  return (
    <div>
      <h2>Name: Alex</h2>
      <p>Age: 21</p>
      <p>Fav Language: JavaScript</p>
    </div>
  );
}
export default Bio;
```

📓 Chapter 3: Props – Passing Data to Components

◇ What are Props?

Props (short for "properties") are used to **pass data from one component to another**, usually from a **parent to a child**.

🎯 *Why Use Props?*

- To make components **reusable** and **dynamic**
- Instead of hardcoding values, pass them as variables

🧱 Real-Life Analogy

Think of a component as a **function**, and **props are like arguments** passed to that function.

```
function greet(name) {
  return "Hello, " + name;
}
greet("Alice"); // Hello, Alice
```

Now with React:

```
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}
<Greeting name="Alice" />
```

🔨 How to Use Props

Let's go step-by-step 👇

1. Create a file: `Greeting.js`

```
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}

export default Greeting;
```

2. Use the component in `App.js`

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

function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
      <Greeting name="Charlie" />
    </div>
```

```
  );
}
```

✅ Output:

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

📦 Multiple Props

```
// Info.js
function Info(props) {
  return (
    <div>
      <h3>{props.title}</h3>
      <p>{props.description}</p>
    </div>
  );
}
export default Info;
// App.js
import Info from "./Info";

function App() {
  return (
    <div>
      <Info title="React" description="A JavaScript library for
building UI" />
      <Info title="Vue" description="Another JavaScript framework" />
    </div>
  );
}
```

✅ Output:

```
React
A JavaScript library for building UI

Vue
Another JavaScript framework
```

🧠 Destructuring Props (Cleaner Syntax)

Instead of writing `props.name`, `props.age` every time:

```
function Profile({ name, age }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>Age: {age}</p>
    </div>
  );
}
```

This works the same as:

```
function Profile(props) {
  return (
    <div>
      <h3>{props.name}</h3>
      <p>Age: {props.age}</p>
    </div>
  );
}
```

📌 Summary Table

Concept	Explanation

Props	Inputs passed to components like function args
Parent -> Child	Data flow is always one way (downward)
Reusability	Pass different props to use the same component multiple times

🏆 Your Mini Challenge

1. Create a component named `Book.js`
2. It should accept `title` and `author` as props
3. Use it in `App.js` to show 3 different books

Example usage:

```
<Book title="1984" author="George Orwell" />
<Book title="The Alchemist" author="Paulo Coelho" />
<Book title="Harry Potter" author="J.K. Rowling" />
```

📓 Chapter 4: State – Making Components Interactive

🔸 What is State in React?

- **State** is a **special variable** in a component that stores **dynamic data**.
- It allows components to **remember things** and **react to user input** or changes.

🧠 Think of **state** as the **brain** of a component that changes over time.

🧱 Real-Life Analogy

Imagine a **toggle switch**:

- OFF = false
- ON = true

If someone flips it, it changes from OFF to ON.

In React, you'd store the ON/OFF value in **state**.

🔧 How to Use State

In **functional components**, we use the `useState` hook to add state.

```
import { useState } from "react";
```

📌 Syntax of useState

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

- `stateVariable` is the current value
- `setStateFunction` is used to update the value

🧪 Simple Counter Example

```
import { useState } from "react";

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

  function increase() {
    setCount(count + 1);
  }

  return (
    <div>
      <h2>Count: {count}</h2>
```

```
      <button onClick={increase}>Increase</button>
    </div>
  );
}

export default Counter;
```

Use it in `App.js`:

```
import Counter from "./Counter";

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

🖥️ Output:

- Shows `Count: 0`
- Clicking the button increases the number

⚠️ Why Can't You Use Normal Variables?

If you write:

```
let count = 0;
```

React will not **re-render** the UI when `count` changes.

Only **state changes trigger re-renders**.

📦 Multiple States Example

```
function InfoBox() {
  const [name, setName] = useState("Alex");
  const [age, setAge] = useState(21);

  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      <button onClick={() => setAge(age + 1)}>Increase Age</button>
    </div>
  );
}
```

✅ Clicking the button will update age, and React will automatically update the UI.

💡 Updating State with Previous Value

```
setCount(prevCount => prevCount + 1);
```

This is helpful when updates depend on the **previous state** (especially in complex apps).

🧠 Summary

Concept	Meaning
useState	Hook to add state to functional components
setState	Function to update the state
Re-render	React updates UI when state changes
Initial Value	The first value passed to `useState()`

🏆 Mini Challenge

1. Create a component `Toggle.js`
2. Add a `useState` with value `false`
3. Create a button that toggles between:
    a. "The light is ON"
    b. "The light is OFF"

Hint:

```
setLightOn(!lightOn);
```

📘 Chapter 5: Conditional Rendering – Showing Things Based on Conditions

🔷 What is Conditional Rendering?

Conditional Rendering means **showing or hiding parts of the UI** based on a **condition** (just like `if`, `else` in regular JavaScript).

You use it when you want your app to **dynamically display different content** depending on:

- user actions
- state values
- props

📘 Example Scenarios

- If a user is logged in, show the dashboard. If not, show the login screen.
- If `isDarkMode === true`, apply dark mode styling.

- Show "Loading..." while data is being fetched.

🔧 Syntax Options

☑ *1. `if` Statement (used inside the function)*

```
function Greet(props) {
  if (props.isLoggedIn) {
    return <h2>Welcome back!</h2>;
  } else {
    return <h2>Please log in.</h2>;
  }
}
```

☑ *2. Ternary Operator (used inside JSX)*

```
function Greet({ isLoggedIn }) {
  return (
    <h2>{isLoggedIn ? "Welcome back!" : "Please log in."}</h2>
  );
}
```

☑ *3. && Operator (only show if true)*

```
function Notification({ hasMessage }) {
  return (
    <div>
      <h2>Inbox</h2>
      {hasMessage && <p>You have new messages!</p>}
    </div>
  );
}
```

🩹 Full Example

LoginControl.js

```
import { useState } from "react";

function LoginControl() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  function toggleLogin() {
    setIsLoggedIn(!isLoggedIn);
  }

  return (
    <div>
      <h2>{isLoggedIn ? "Welcome User!" : "Please Log In"}</h2>
      <button onClick={toggleLogin}>
        {isLoggedIn ? "Logout" : "Login"}
      </button>
    </div>
  );
}

export default LoginControl;
```

Use in App.js:

```
import LoginControl from "./LoginControl";

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

🖥️ Output:

- Initially shows: `Please Log In` + Login button
- After clicking: `Welcome User!` + Logout button

🧠 Summary Table

Syntax Used	Works Like
`if-else`	Traditional logic
Ternary (`? :`)	One-line if-else inside JSX
Logical AND (&&)	Show something only if true

🏆 Mini Challenge

Create a component called `Weather.js`:

- Take a prop called `isRaining`
- If `isRaining === true`, show "Take an umbrella!"
- Otherwise, show "It's sunny!"

Use a ternary inside JSX.

Example usage:

```
<Weather isRaining={true} />
<Weather isRaining={false} />
```

📗 Chapter 6: Handling Events – React's Way of Dealing with User Actions

🔹 What are Events?

Events are user interactions like:

- Clicking a button 🖱️
- Typing into a textbox ⌨️
- Hovering over an element 🖐️

Just like in JavaScript, you can **respond to these actions** in React using event handlers.

🔧 Syntax for Events in React

React events are **camelCase** and use **curly braces {}** to pass a function.

```
<button onClick={myFunction}>Click Me</button>
```

✅ `onClick` (not `onclick`)

✅ `{myFunction}` (not `"myFunction()"`)

🧪 Example 1: Click Event

```
function Clicker() {
  function handleClick() {
    alert("Button was clicked!");
  }

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

🩹 Example 2: Typing in an Input Box

```
import { useState } from "react";

function InputBox() {
  const [text, setText] = useState("");

  function handleChange(event) {
    setText(event.target.value);
  }

  return (
    <div>
      <input type="text" onChange={handleChange} />
      <p>You typed: {text}</p>
    </div>
  );
}
```

🧠 Event Object (event)

React passes a special object to your handler function with useful info like:

- What was clicked or typed
- What the value is
- Mouse position (in advanced cases)

```
function handleClick(event) {
  console.log(event.target); // logs the DOM element clicked
}
```

🧪 Example 3: Toggle with Button

```
import { useState } from "react";

function ToggleButton() {
  const [isOn, setIsOn] = useState(false);

  function toggle() {
    setIsOn(!isOn);
  }

  return (
    <button onClick={toggle}>
      {isOn ? "ON" : "OFF"}
    </button>
  );
}
```

✅ Clicking the button switches between ON and OFF.

⚠️ Common Mistakes

Mistake	Correct Version
`onClick="handleClick()"`	`onClick={handleClick}`
`onclick`	`onClick` (React is camelCase)
Directly calling a function	Only pass the reference

💡 Tip: Inline Functions

You can define event logic **inline**:

```
<button onClick={() => alert("Hi!")}>Say Hi</button>
```

Use inline functions **for small logic**. Otherwise, create a separate function.

🧩 Summary

Event	Usage
`onClick`	Clicks
`onChange`	Typing/input
`onSubmit`	Forms (later chapter)
`onMouseOver`	Mouse hover (bonus)

🏆 Mini Challenge

Create a component called `Counter.js`:

- Show a number starting at 0
- Two buttons: **Increase** and **Reset**
- Clicking **Increase** adds +1
- Clicking **Reset** sets it back to 0

📘 Chapter 7: Lists & Keys – Rendering Multiple Items Dynamically

🔷 Why Use Lists?

In React, we often want to **display multiple items**, such as:

- a list of names
- search results

- menu items
- comments, messages, etc.

To do this, we use JavaScript's `.map()` function inside JSX.

📄 Basic Example: Rendering a List

```
function FruitsList() {
  const fruits = ["Apple", "Banana", "Orange"];

  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
      ))}
    </ul>
  );
}
```

- `fruits.map(...)` loops through each item
- `key={index}` helps React identify each `<li>` uniquely
- `key` is **required** for better performance and no warnings

🔐 What is a Key?

A **key** is a **unique identifier** for each element in a list.

Why is it important?

- Helps React update only the changed item efficiently.
- Prevents bugs and improves performance.

👉 Always use a **unique value** (like an `id` if available), not just `index`, when possible.

🧪 Example with Objects

```
function StudentsList() {
  const students = [
    { id: 101, name: "Asha" },
    { id: 102, name: "Ben" },
    { id: 103, name: "Catherine" },
  ];

  return (
    <ul>
      {students.map(student => (
        <li key={student.id}>{student.name}</li>
      ))}
    </ul>
  );
}
```

💡 .map() vs .forEach()

Feature	.map()	.forEach()
Returns a value	✅ Yes	❌ No
Used in JSX	✅ Common	❌ Not suitable

🧠 You Can Use Any JSX in .map()

```
function Cards() {
  const items = ["HTML", "CSS", "JavaScript"];

  return (
    <div>
```

```
      {items.map((item, index) => (
        <div key={index} style={{ border: "1px solid gray", margin:
"5px", padding: "5px" }}>
          <h3>{item}</h3>
          <p>This is a {item} card.</p>
        </div>
      ))}
    </div>
  );
}
```

🔥 Bonus: Conditional Rendering in List

```
const tasks = [
  { id: 1, name: "Read", done: true },
  { id: 2, name: "Code", done: false },
];

function TaskList() {
  return (
    <ul>
      {tasks.map(task => (
        <li key={task.id}>
          {task.name} {task.done && "✅"}
        </li>
      ))}
    </ul>
  );
}
```

🧩 Summary

Concept	Description
`map()`	Loop through and render JSX for each item

`key`	Unique ID that helps React optimize rendering
`.forEach()`	❌ Not used for JSX returns
Arrays of Objects	Common pattern in real apps

🏆 Mini Challenge

Create a component `BooksList.js`:

- An array of objects like:

```
[
  { id: 1, title: "Harry Potter" },
  { id: 2, title: "Sherlock Holmes" },
  { id: 3, title: "The Alchemist" }
]
```

- Render each book title inside `<li>` with a proper `key`.

📓 Chapter 8: Forms in React – Handling User Input Properly

🔸 Why Are Forms Important?

Forms allow users to:

- enter their name, email, or any data
- submit feedback
- search for something
- log in or sign up

In React, we use **controlled components** to manage form inputs with `useState`.

🔧 What is a Controlled Component?

A **controlled component** is an input field where **React controls the value**.

You need:

1. A `useState` variable to **store the value**
2. An `onChange` function to **update the state**
3. A `value` attribute on the input to **bind state to input**

🩹 Example 1: Simple Text Input

```
import { useState } from "react";

function NameForm() {
  const [name, setName] = useState("");

  function handleChange(e) {
    setName(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault(); // prevent form refresh
    alert(`Hello, ${name}!`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name: </label>
      <input type="text" value={name} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

✅ Output:

- Typing updates the state
- Clicking **Submit** shows an alert with the name

🔧 Managing Multiple Inputs

```
function LoginForm() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    console.log(email, password);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button type="submit">Login</button>
    </form>
  );
}
```

🔧 Example 3: Radio, Checkbox, and Dropdown

```
function SurveyForm() {
  const [language, setLanguage] = useState("JavaScript");
  const [isChecked, setIsChecked] = useState(false);

  return (
    <form>
      <label>Choose a language:</label>
      <select value={language} onChange={(e) =>
setLanguage(e.target.value)}>
        <option>JavaScript</option>
        <option>Python</option>
        <option>Java</option>
      </select>

      <div>
        <label>
          <input
            type="checkbox"
            checked={isChecked}
            onChange={(e) => setIsChecked(e.target.checked)}
          />
          I agree to terms
        </label>
      </div>

      <p>Selected: {language}</p>
      <p>{isChecked ? "Agreed" : "Not Agreed"}</p>
    </form>
  );
}
```

▩ Summary

Input Type	Handler Example

Text Input	`onChange={e => setValue(e.target.value)}`
Checkbox	`checked={value}`
Select/Dropdown	`value={value}`
Submit	`onSubmit={handleForm}`

- All inputs use **state** to **store and reflect** values.
- Forms should use `onSubmit` to handle submit logic.

🏆 Mini Challenge

Create a component called `FeedbackForm.js`:

- One input for "Your Feedback"
- A textarea input
- A submit button
- On submit, show an alert with the text entered

📓 Chapter 9: `useEffect` – Side Effects in React

◇ What Is a "Side Effect"?

A **side effect** is anything that happens *outside the component*:

- fetching data
- changing the DOM directly
- setting up event listeners
- timers, intervals, etc.

React provides a **hook** called `useEffect()` to manage side effects.

🔧 Basic Syntax of useEffect

```
import { useEffect } from "react";

useEffect(() => {
  // Side effect code here (runs after component renders)
}, []);
```

- The **function** runs **after the component is rendered**
- The **empty array `[]`** means "run once" (like `componentDidMount`)

🩹 Example 1: Run Once After Render

```
import { useEffect } from "react";

function Welcome() {
  useEffect(() => {
    console.log("Component mounted!");
  }, []);

  return <h1>Welcome!</h1>;
}
```

📝 Output in console: `Component mounted!`

✅ This will only run **once**, just after the first render.

🔄 Example 2: Run When a Value Changes

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

function Counter() {
  const [count, setCount] = useState(0);
```

```
  useEffect(() => {
    console.log("Count changed to:", count);
  }, [count]);

  return (
    <>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </>
  );
}
```

📌 This `useEffect` will run **every time `count` changes**

🕓 Example 3: Run a Timer

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setSeconds(prev => prev + 1);
    }, 1000);

    return () => clearInterval(timer); // cleanup
  }, []);

  return <h2>Timer: {seconds} sec</h2>;
}
```

⚠️ Always return a cleanup function to clear timers, listeners, etc.

📡 Example 4: Fetch API Data

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

function Users() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => res.json())
      .then(data => setUsers(data));
  }, []);

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

📥 This fetches data *only once* when the component is mounted.

⌨️ Summary

`useEffect` Timing	`Dependency Array` Example	When It Runs
Once	`[]`	On first mount
On value change	`[count]`	When `count` changes
On every render	*none*	Every render (⚠️ avoid usually)

- You **must use `useEffect`** for things like:
  - Fetching data
  - Setting up timers
  - Subscribing to events
- Always return a **cleanup** if needed (like for intervals or listeners)

🏆 Mini Challenge

Create a `QuoteFetcher.js`:

- On mount, fetch a quote from: https://api.quotable.io/random
- Display the quote content and author
- Button to refresh a new quote

📘 Chapter 10: Conditional Rendering in React

🔸 What Is Conditional Rendering?

Conditional rendering means showing or hiding parts of the UI depending on some condition, like:

- Is the user logged in?
- Is data loaded?
- Did an error occur?

In React, you use JavaScript logic inside JSX to decide what to render.

🔧 Basic If-Else with Ternary Operator

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h1>Welcome Back!</h1> : <h1>Please Log In</h1>}
    </div>
  );
}
```

🔧 Using Logical AND (&&) for Conditional Display

```
function Warning({ showWarning }) {
  return (
    <div>
      {showWarning && <p style={{ color: "red" }}>Warning: Check this
out!</p>}
    </div>
  );
}
```

- If `showWarning` is `true`, the message is shown
- If `false`, nothing is rendered

🔧 If-Else Using a Function

Sometimes using a helper function helps:

```
function LoginControl({ isLoggedIn }) {
  function renderMessage() {
    if (isLoggedIn) {
      return <h1>Welcome back!</h1>;
    }
    return <h1>Please sign up.</h1>;
  }

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

🔧 Conditional Rendering with Components

You can also conditionally render different components:

```
function UserStatus({ isLoggedIn }) {
  if (isLoggedIn) {
    return <LogoutButton />;
  } else {
    return <LoginButton />;
  }
}

function LoginButton() {
  return <button>Login</button>;
}
function LogoutButton() {
  return <button>Logout</button>;
}
```

🔧 Handling Loading and Error States

```
function DataFetcher({ isLoading, error, data }) {
  if (isLoading) {
    return <p>Loading...</p>;
  }

  if (error) {
    return <p>Error: {error}</p>;
  }

  return <div>Data: {data}</div>;
}
```

🟩 Summary

Technique	Example	When to Use
Ternary Operator	`{cond ? A : B}`	Simple true/false conditions
Logical AND (&&)	`{cond && A}`	Show something only if true

If-Else in Function	`function render() { ... }`	Complex logic before rendering
Early Returns	`if (...) return ...`	Different components or states

🏆 Mini Challenge

Create a `UserProfile.js`:

- Accepts a prop `isLoggedIn`
- If logged in, show "Welcome, User!"
- If not logged in, show a login button
- Bonus: Add a loading state that shows "Loading user info..."

📘 Chapter 11: React Router – Navigating Between Pages

🔹 What is React Router?

React Router helps you build **single-page applications (SPA)** with **multiple views/pages**.

Instead of loading a new page from the server, React Router **changes the UI dynamically** based on the URL.

🔧 How to Install React Router

```
npm install react-router-dom
```

🔧 Basic Setup

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-
router-dom";

function Home() {
  return <h1>Home Page</h1>;
}

function About() {
  return <h1>About Page</h1>;
}

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link> |{" "}
        <Link to="/about">About</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
}

export default App;
```

🔍 How This Works

- `<Router>` wraps your app to enable routing.
- `<Link>` is like `<a>`, but it prevents page reload.
- `<Routes>` contains `<Route>` elements, each mapping a URL path to a component.

- `element={<Component />}` tells React what to show for that route.

🩹 Example: Navigating Pages

- Go to `/` → shows **Home Page**
- Go to `/about` → shows **About Page**
- Clicking links changes URL **without full reload**

🔧 Dynamic Route Parameters

You can pass dynamic segments like user IDs:

```
function UserProfile({ id }) {
  return <h2>User Profile: {id}</h2>;
}

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/user/:id" element={<User />} />
      </Routes>
    </Router>
  );
}

function User() {
  const { id } = useParams();
  return <UserProfile id={id} />;
}
```

Visiting `/user/123` will show `User Profile: 123`.

🧩 Summary

Component	Purpose
`<Router>`	Wraps your app to enable routing
`<Routes>`	Container for route definitions
`<Route>`	Defines a route and linked component
`<Link>`	Navigation links without reload
`useParams()`	Access dynamic route parameters

🏆 Mini Challenge

Build a small app with 3 pages:

- Home (`/`)
- About (`/about`)
- Profile (`/profile/:username`) showing username from URL

📘 Chapter 12: React Context API – Sharing Data Globally

🔷 What Is Context API?

React's **Context API** lets you share data across many components **without passing props manually at every level** (also called **prop drilling**).

Why Use Context?

Imagine you have:

```
<App>
  <Header>
    <UserProfile />
  </Header>
</App>
```

If App has user info you want to show in `UserProfile`, normally you'd pass props like:

```
<App user={user}>
  <Header user={user}>
    <UserProfile user={user} />
  </Header>
</App>
```

With **Context**, you provide the data once, and any nested component can access it directly.

🔧 How to Use Context: Step-by-Step

1. Create a Context

```
import React from "react";

const UserContext = React.createContext(null);
```

2. Provide Context Value

Wrap part of your app with `UserContext.Provider` and pass a value:

```
function App() {
  const user = { name: "Alice", loggedIn: true };
```

```
  return (
    <UserContext.Provider value={user}>
      <Header />
    </UserContext.Provider>
  );
}
```

3. Consume Context in Child Components

You can access the context value with `useContext`:

```
import { useContext } from "react";

function UserProfile() {
  const user = useContext(UserContext);

  return <p>User: {user.name}</p>;
}
```

🧪 Full Example

```
import React, { createContext, useContext } from "react";

const UserContext = createContext(null);

function App() {
  const user = { name: "Alice", loggedIn: true };

  return (
    <UserContext.Provider value={user}>
      <Header />
    </UserContext.Provider>
  );
}
```

```
function Header() {
  return (
    <header>
      <UserProfile />
    </header>
  );
}

function UserProfile() {
  const user = useContext(UserContext);
  return <p>Hello, {user.name}!</p>;
}

export default App;
```

◇ Why Use Context API?

- Avoids passing props through many levels
- Good for theme, user info, settings, localization data
- Helps keep components clean and simple

⚠ Important Notes

- Avoid overusing Context for everything, it's mainly for **global/shared state**
- If you just need to pass data a few levels down, props are fine
- Updating context causes re-render of all consuming components

🏆 Mini Challenge

- Create a `ThemeContext` with values like `{color: "blue", background: "lightgray"}`

- Use the context to style a button inside deeply nested components

📘 Chapter 13: Custom Hooks – Reusable Logic in React

🔹 What Are Custom Hooks?

- **Custom Hooks** are JavaScript functions that let you **reuse stateful logic** between components.
- They start with the word **`use`** (like `useState`, `useEffect`).
- They help keep your code clean by **extracting common logic** into one place.

🔧 Why Use Custom Hooks?

Imagine you have several components that:

- Fetch data from an API
- Manage form input states
- Handle timers or intervals

Instead of duplicating code, you put that logic in a **custom hook** and reuse it.

🔧 How to Create a Custom Hook?

1. Write a function starting with `use`
2. Use React hooks (`useState`, `useEffect`, etc.) inside it
3. Return the data or functions you want to share

🔧 Example: A Custom Hook to Track Window Width

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

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    function handleResize() {
      setWidth(window.innerWidth);
    }

    window.addEventListener("resize", handleResize);

    // Cleanup the event listener on unmount
    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return width;
}
```

🔧 Using the Custom Hook in a Component

```
function WidthDisplay() {
  const width = useWindowWidth();

  return <p>Window width: {width}px</p>;
}
```

◇ Explanation

- `useWindowWidth` tracks window size and updates the state on resize.
- Any component can use `useWindowWidth()` to get current width.

- Logic is separated, making components clean and reusable.

🏆 Mini Challenge

- Create a custom hook called `useToggle` that manages a boolean state (true/false).
- Use it in a component with a button to toggle between "ON" and "OFF".

📓 Chapter 14: Advanced Data Fetching & Side Effects with `useEffect`

🔹 Recap: What is `useEffect`?

- `useEffect` lets you perform **side effects** in React components:
  - Fetching data
  - Subscribing to events
  - Manually updating the DOM
- Runs after render, and can run again when dependencies change.

🔧 Typical Data Fetching Pattern

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

function DataFetcher() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts/1")
      .then((response) => {
        if (!response.ok) throw new Error("Network error");
        return response.json();
```

```
      })
      .then((json) => {
        setData(json);
        setLoading(false);
      })
      .catch((err) => {
        setError(err.message);
        setLoading(false);
      });
  }, []); // Empty dependency array: runs once after mount

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;

  return (
    <div>
      <h2>{data.title}</h2>
      <p>{data.body}</p>
    </div>
  );
}
```

◇ Important Details

- The empty array [ ] as the second argument means **run only once** after component mounts.
- If you include variables in the array, the effect runs again when they change.
- Always **clean up side effects** if needed (e.g., event listeners, subscriptions).

🔧 Cleaning up Effects

Example: subscribing to window resize

```
useEffect(() => {
  function onResize() {
```

```
    console.log("Resized");
  }

  window.addEventListener("resize", onResize);

  return () => {
    // Cleanup to avoid memory leaks
    window.removeEventListener("resize", onResize);
  };
}, []);
```

🔧 Avoiding Stale Closures & Race Conditions

If you fetch data in an effect, cancel outdated requests or ignore their results to prevent bugs.

🏆 Mini Challenge

Build a component that fetches a random joke from https://official-joke-api.appspot.com/random_joke every time you click a button. Handle loading and error states.

🟫 Chapter 15: Performance Optimization — `React.memo`, `useCallback`, and `useMemo`

🔸 Why Optimize Performance?

React re-renders components when props or state change, but sometimes re-renders are unnecessary and slow down your app.

Optimization tools help **avoid unnecessary re-renders** and improve responsiveness.

1 `React.memo` — Memoize Components

- Wrap a component with `React.memo` to **prevent re-rendering if props don't change**.
- Works only with **function components**.

```
const MyComponent = React.memo(function({ value }) {
  console.log("Rendered!");
  return <div>{value}</div>;
});
```

- If parent re-renders but `value` stays the same, `MyComponent` won't re-render.

2 `useCallback` — Memoize Functions

- React recreates functions on every render, which may cause child components to re-render.
- `useCallback` returns a memoized version of a function that only changes if dependencies change.

```
const memoizedCallback = useCallback(() => {
  console.log("Clicked!");
}, []);  // Runs only once
```

3 `useMemo` — Memoize Expensive Calculations

- Use to memoize **expensive calculations** so they don't run on every render.

```
const expensiveValue = useMemo(() => {
  // some expensive calculation
```

```
  return computeHeavyStuff();
}, [dependencies]);
```

🩹 Example Putting It All Together

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

const List = React.memo(({ items, onClick }) => {
  console.log("List rendered");
  return (
    <ul>
      {items.map((item) => (
        <li key={item} onClick={() => onClick(item)}>{item}</li>
      ))}
    </ul>
  );
});

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

  const items = useMemo(() => {
    console.log("Calculating items");
    return ["Apple", "Banana", "Cherry"];
  }, []);

  const handleClick = useCallback((item) => {
    alert(`You clicked ${item}`);
  }, []);

  return (
    <div>
      <button onClick={() => setCount(c => c + 1)}>Count:
{count}</button>
      <List items={items} onClick={handleClick} />
    </div>
```

```
  );
}

export default App;
```

◇ What's Happening?

- `List` is memoized, so it won't re-render if props stay the same.
- `items` is memoized with `useMemo` (expensive to create?).
- `handleClick` is memoized with `useCallback` (prevents unnecessary re-creation).

🏆 Mini Challenge

Create a component that renders a list of numbers and a button that increases a counter. Use `React.memo` and `useCallback` to prevent unnecessary list re-renders when counter updates.

📓 Chapter 16: Testing React Components

◇ Why Test React Components?

- Ensure your UI works as expected
- Catch bugs early
- Make your code easier to maintain and refactor

🔧 Common Testing Tools

- **Jest**: JavaScript testing framework (comes by default with Create React App)

- **React Testing Library**: Helps test React components in a way similar to how users interact with them

🔧 Basic Setup (Create React App)

If you used Create React App, Jest and React Testing Library are pre-configured!

🔧 Writing Your First Test

Example: Testing a simple `Greeting` component

```
// Greeting.js
export default function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

```
// Greeting.test.js
import { render, screen } from "@testing-library/react";
import Greeting from "./Greeting";

test("renders greeting message", () => {
  render(<Greeting name="Alice" />);
  const greetingElement = screen.getByText(/hello, alice!/i);
  expect(greetingElement).toBeInTheDocument();
});
```

◇ Explanation

- `render()` renders the component in a virtual DOM for testing
- `screen.getByText()` queries for elements like a user would see
- `expect(...).toBeInTheDocument()` asserts the element is present

🔧 Testing User Events

You can simulate user actions like clicks, typing:

```
import { render, screen, fireEvent } from "@testing-library/react";
import Counter from "./Counter"; // component with button to increase
count

test("increments counter on click", () => {
  render(<Counter />);
  const button = screen.getByRole("button");
  fireEvent.click(button);
  expect(screen.getByText("Count: 1")).toBeInTheDocument();
});
```

🏆 Mini Challenge

- Create a `Toggle` component with a button that toggles between "ON" and "OFF".
- Write tests to check if the button text changes correctly on click.

Want to continue to **Deploying React Apps** or explore more testing techniques? Just say **"Next Chapter"**!

📓 Chapter 17: Deploying React Apps

🔸 Why Deployment?

- Deployment means **making your app live** on the internet so others can use it.
- After development, deployment is the final step to share your React app.

🔧 Common Deployment Options for React Apps

1. **GitHub Pages**
2. **Netlify**
3. **Vercel**
4. **Firebase Hosting**
5. **Custom server (e.g., AWS, DigitalOcean)**

🔧 Easy Deployment with GitHub Pages

Step 1: Build your app

```
npm run build
```

- This creates a `build` folder with static files.

Step 2: Install `gh-pages` package

```
npm install --save-dev gh-pages
```

Step 3: Add these scripts to `package.json`

```
"scripts": {
  "predeploy": "npm run build",
  "deploy": "gh-pages -d build"
}
```

Step 4: Add `homepage` field in `package.json`

```
"homepage": "https://your-username.github.io/your-repo-name"
```

Replace with your GitHub username and repo.

Step 5: Deploy

```
npm run deploy
```

Your app will be live at the URL you set!

🔧 Deploy with Netlify (simpler for many projects)

- Go to netlify.com and sign up.
- Connect your GitHub repo.
- Set build command: `npm run build`
- Set publish directory: `build`
- Netlify will build and deploy automatically!

◇ Tips for Deployment

- Use relative paths or set `homepage` in `package.json` correctly.
- Test your app locally with `npm run build` and a local server like `serve`.
- Configure environment variables if needed (e.g., API keys).

🏆 Mini Challenge

- Deploy your current React project to GitHub Pages or Netlify.
- Share the live URL to test your deployment skills!