**Module 1**

**Topic 1: Course Introduction**

**Lecture 1: Introduction to the course: React Basics**

✅ **Introduction to React Basics**

* React is a JavaScript library used to build user interfaces for websites.
* This course introduces the basic structure and usage of React.js.

✅ **Understanding Components**

* User interfaces are made up of small, independent parts called components.
* Components can be reused and combined to create more complex structures.
* React helps organize and manage these components efficiently.

✅ **Why React is Popular**

* React makes it easy to build and manage UI components.
* It performs well and does not use too many browser resources.

✅ **Working with State**

* State refers to all the values that the app is using at a specific time.
* Managing state allows apps to update and respond to user actions.

✅ **Styling in React**

* You will learn how to add styles to your app and reuse them across components.

✅ **Handling Events**

* The app can respond to events like clicks or data entered by the user.

✅ **Navigation and Assets**

* You will learn how to create navigation between pages.
* You will also use assets like images and icons in your app.

✅ **Course Outcome: Portfolio Projects**

* You will complete projects to apply what you’ve learned.
* These projects will help you build a personal portfolio.

**Lecture 2: How is React used in the real world?**

✅ **React at Meta (Facebook.com)**

* Meta uses many experimental and new React features.
* Facebook.com served as a testing ground for unreleased features from the React team.
* Rewriting Facebook.com involved building on a modern and faster tech stack using React.

✅ **Role of the React Apps Team**

* The React Apps team at Meta builds new features for Facebook.com.
* The team experiments with advanced and upcoming features of React.

✅ **Popular Apps Built with React**

* React is used by many companies including Facebook, Instagram, Netflix, Airbnb, and The New York Times.
* Interactive websites often use React for building dynamic UIs.

✅ **Why Rewrite Facebook.com in React**

* The older site was outdated and had performance limitations.
* React offered better speed, responsiveness, and ease of development.
* The redesign aimed to create a modern look and improved user experience.

✅ **Scale and Collaboration**

* Around 40 engineers worked together on the Facebook.com rewrite.
* Collaboration with various product teams helped transition products to the new stack.
* It was a large, high-risk effort requiring strong proof of performance and reliability.

✅ **Proving the Value of React**

* Teams needed to be convinced that the new version was not only modern-looking but also faster and more efficient.
* The goal was to support product teams in delivering better web experiences.

✅ **React as an Open Source Project**

* React is open source and welcomes contributions from developers worldwide.
* Engineers inside and outside of Meta contribute to its development.
* This fosters a strong, active community that continuously maintains and improves React.

✅ **Community and Contribution**

* Developers can ask questions, suggest features, or contribute code.
* There's active engagement through documentation, forums, and updates.
* A React conference is held annually to share updates and connect developers.

✅ **Getting Involved with React**

* The React community offers many ways to participate and learn.
* Beginners are encouraged to explore official documentation.
* Connecting with the community enhances the development experience and opens contribution opportunities.

**Topic 2: React Components and Where They Live**

**Lecture 1:** **Why React?**

✅ **Introduction to React Usage in Tech Careers**

* React is widely used across the tech industry.
* Learning React is valuable for long-term career growth.

✅ **React vs Traditional Programming Concepts**

* most programming was object-oriented using inheritance.
* React uses **composition**, a different approach that may feel unfamiliar at first.
* With strong documentation and community support, the transition becomes easier.

✅ **Learning and Community Support**

* Numerous resources available: documentation, YouTube, and open-source projects.
* React has a large, active community that helps beginners and experienced developers alike.

✅ **Why Developers Choose React**

* Easy to learn and pick up.
* Offers high flexibility for creating custom UIs.
* Ideal for building modern, interactive web applications.

✅ **Integration with Other Libraries**

* React is just a **front-end library**, not a full-stack solution.
* It integrates well with tools like Redux and other third-party libraries.
* Perfect for developers who need flexibility in their tech stack.

✅ **Handling Complex UIs**

* React excels in building applications with many custom UI elements.
* Supports **component reusability** across different parts of an application.

✅ **Code Reusability and Efficiency at Meta**

* Meta uses a shared library of core UI components across Facebook, Instagram, and Messenger.
* Fixing a bug in one component improves the entire system.
* Promotes efficiency and high code quality.

✅ **React vs Angular**

* **Angular**: Full-fledged framework with built-in routing, HTTP, etc.
* **React**: Flexible and lightweight, but needs third-party tools for full app development.
* React is better for custom UI needs; Angular is easier for creating single-page apps quickly.

✅ **Advice for Learning React**

* Start with simple projects—don’t aim for complexity at the beginning.
* Always follow best practices and consult the documentation.
* Focus on learning key concepts like **hooks** properly.

✅ **Long-Term Value of React**

* React will likely appear in many roles throughout a developer's career.
* Its large support network and ecosystem make it a strong skill to master.

**Lecture 2: React.js overview**

✅ **Understanding Single-Page Applications (SPAs)**

* SPAs load content dynamically without full page reloads.
* Unlike traditional websites, they don't reload common elements like headers and footers.
* React is a popular tool for building SPAs, providing better speed and user experience.
* SPAs are ideal for businesses needing fast, scalable, and interactive web apps.

✅ **React and Its Growing Popularity**

* React is developed by Meta and powers many top websites.
* Often, React apps update content without changing the URL, a common SPA trait.
* React offers flexibility, speed, and rich user interfaces.

✅ **React Basics: Component-Based Architecture**

* React apps are built using reusable **components**.
* Each component handles its own functionality and UI structure.
* Components are modular, allowing easy integration and maintenance.

✅ **Benefits of Using Components in React**

* Components can be reused and moved around the app as needed.
* Multiple developers can work on different components independently.
* Promotes clean, maintainable, and scalable code.

✅ **Components in UI Design**

* All user interfaces are created using a mix of **simple and complex components**.
* Example: An e-commerce checkout page may have
  + **Header** (logo, nav, cart)
  + **Payment Section** (form, validation, submit button)
  + **Sidebar** (order summary)
* Each section is a self-contained component with its own HTML, CSS, and JavaScript.

✅ **Not Just React — Component Use is Common**

* Component-based UI design exists outside React too.
* React simplifies and streamlines this design approach.

✅ **Rendering Components in React**

* Components are rendered efficiently to the **DOM (Document Object Model)**.
* React minimizes performance impact when updating components.

✅ **Traditional DOM Manipulation Challenges**

* Before React, developers had to manually manipulate the DOM.
* This made code messy and hard to maintain — known as **spaghetti code**.

✅ **React’s Virtual DOM Advantage**

* React uses a **virtual DOM** — a lightweight in-memory copy of the real DOM.
* Updates are first made in the virtual DOM, then selectively applied to the real DOM.
* This results in **faster rendering**, improved performance, and fewer browser resource issues.

**Lecture 3:** **Introduction to functional component**

✅ **Functional Components in React**

* React components are similar to JavaScript functions.
* They take inputs, perform actions, and return JSX (UI).
* Two types of components in React:
  + Functional Components (focus of this lesson)
  + Class Components (covered later)

✅ **Root Component and Rendering**

* Every React app must have one **Root Component**.
* Default root is the App component inside index.js.
* Components are rendered like self-closing HTML tags: <App />.

✅ **Component Composition**

* Root component can contain other custom components.
* Components represent different parts of the UI (e.g., header, sidebar).
* The App component is rendered inside a <div id="root"> in the HTML.

✅ **JSX – JavaScript XML**

* JSX is a **syntax extension** to JavaScript.
* Looks similar to HTML but can include JavaScript code.
* JSX enables dynamic content in components.

✅ **Creating a Functional Component (Example)**

* Create a new file named Heading.js.
* Use a capital letter for the component name: function Heading().
* Inside, declare a variable: const title = "This is some heading text";
* Return JSX:

return <h1>{title}</h1>;

* Curly braces {} are used to inject JavaScript variables inside JSX.

✅ **Capitalization Rules in React**

* Component names **must start with a capital letter**.
* Lowercase names are treated as built-in HTML tags by React.

✅ **Transpiling JSX**

* JSX is not valid JavaScript – it needs to be converted (transpiled).
* Transpiling means converting JSX into actual JavaScript + HTML.
* Tools like Babel handle this process in the background.

**Lecture 4:** **Creating React components**

✅ **Creating a New React App in VS Code**

* Open VS Code terminal in the desired folder.
* Run command: npx create-react-app .
  + The . means create the app in the current folder.
* Wait for the app to install (may take a few minutes).
* Start the development server with: npm start.
* App opens at: http://localhost:3000.

✅ **Understanding the Project Structure**

* Key folders/files created:
  + node\_modules: contains project dependencies.
  + public: static assets.
  + src: where development happens (focus here).
  + package.json: project configuration and dependencies.

✅ **Cleaning the Default App Component**

* Go to App.js.
* Remove the existing content inside the App function.
* Save the file – the page shows blank (expected).

✅ **Creating a New Functional Component**

* Inside App.js, define a new function:

function Header() {

return <h1>Hello World</h1>;

}

* This is a simple **functional component** returning JSX.

✅ **Rendering the Header Component**

* Inside the App function, render the Header component:

function App() {

return <Header />;

}

* Use self-closing tag syntax like HTML.
* Save changes (Ctrl + S / Cmd + S).
* "Hello World" appears in the browser.

✅ **Component Relationships**

* App is the **root component**.
* It renders the Header component.
* Both components currently exist in the same file.

✅ **Making Components Reusable**

* Next step: move Header to its own file.
* Allows reusing Header across the app.
* You'll learn to import/export components in upcoming lessons.

**Lecture 5:** **Transpiling JSX (Readings)**

✅ **What is JSX and Why It Needs Transpiling**

* JSX is a syntax extension for JavaScript used in React.
* Browsers **cannot understand JSX directly**.
* JSX must be **transpiled** (converted) into plain JavaScript before a browser can run it.

✅ **What is a Transpiler?**

* A **transpiler** converts code from one version or type to another.
* Example:
  + **ES6 Code**: const PI = 3.14 (modern JS)
  + **ES5 Equivalent**: var pi = 3.14 (older browser compatible)
* JSX → JavaScript (ES5/ES6) is the core task here

✅ **Babel – The Popular JSX Transpiler**

* **Babel** is a tool used to transpile JSX into JavaScript.
* Website: Babel Online REPL
* Choose “**Classic Runtime**” in settings to see React-style output.

✅ **Example 1: Transpiling a JSX Component**

JSX Code:

function Heading(props) {

return <h1>{props.title}</h1>;

}

Transpiled Output:

function Heading(props) {

return React.createElement("h1", null, props.title);

}

* React.createElement("h1", null, props.title)
  + "h1" → tag name
  + null → no attributes
  + props.title → content inside the tag

✅ **Example 2: Transpiling JSX Render Syntax**

JSX Render:

<Heading title="This is the heading text!" />

Transpiled Output:

React.createElement(Heading, {

title: "This is the heading text!"

});

* Heading → React component
* { title: "..." } → props passed to component

✅ **Why Transpiling Matters**

* Ensures **browser compatibility** across old and new browsers.
* Makes it possible to use **modern JavaScript features** today.
* Allows JSX to be used freely in React projects.

✅ **Minimum Code to Show Something on Screen**

To display content using a React component:

function App() {

return <h1>Hello</h1>;

}

Transpiles to:

function App() {

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

}

This is the **simplest working component** that renders something in React.

**Lecture 6: The React project structure**

✅**Default Folder Structure in a React App**

**1. Main Folders Created by create-react-app**

* **node\_modules/**:
  + Stores all installed npm packages.
  + Automatically generated when you run npm install.
* **public/**:
  + Holds static assets (e.g. favicon.ico, logo.png, manifest.json, robots.txt).
  + Contains the important index.html file – where the React app is injected into.
* **src/ (Source folder)**:
  + Main development area where all your components and logic live.
  + Key files:
    - index.js: Entry point – renders the root component.
    - App.js: Root component of the app.
    - App.css, index.css: CSS files for styling.
    - App.test.js, setupTests.js, reportWebVitals.js: Testing and performance tools.
    - logo.svg: Used in the default homepage.

✅ **Root-Level Files**

* **.gitignore**:
  + Lists files/folders to exclude from version control (e.g., node\_modules/).
* **README.md**:
  + Documentation for project – useful for GitHub.
* **package.json**:
  + Holds project metadata and dependency list.
  + Defines npm scripts (e.g., start, build).
* **package-lock.json**:
  + Stores exact versions of installed dependencies for reproducibility.

✅ **Key Concepts and Best Practices**

* **JSX and Babel**:
  + JSX must be transpiled (via Babel) into plain JavaScript using React.createElement.
* **Index.html**:
  + React components are injected into the <div id="root"> inside this file.
* **Customizing Folder Structure**:
  + Add components/, assets/, utils/ directories inside src/ as needed.
* **Planning Structure**:
  + Helps maintainability, scalability, and team collaboration.
* **Deleting Unused Files**:
  + You can safely delete some default files (like logo.svg, test files) if not needed, but remove all references in code too.

**Lecture 7:** **Customizing the project**

✅ **Project Structure & Planning**

* **Goal**: Build a typography-focused layout for a coding blog (no images).
* **Sections**: Nav, Promo, Blog post intros (Intro1, Intro2, Intro3), Footer.
* **Setup Command**:

npm init react-app customizing-example

* **Organize Code**: Add a components folder in src/.

**🧩 Component Breakdown**

* **Nav.js**: Main navigation menu with Home, Articles, About, Contact.
* **Promo.js**: Promotional section with main heading and subheading.
* **Intro1.js**: First blog post preview.
* **Intro2.js**: Second blog post preview.
* **Intro3.js**: Third blog post preview.
* **Footer.js**: Footer with simple copyright.

✅ **Final Folder Structure**

src/

components/

Nav.js

Promo.js

Intro1.js

Intro2.js

Intro3.js

Footer.js

App.js

index.js

index.css

...

**💬 Important JSX Notes**

* **className**: Used instead of class in JSX due to JavaScript syntax rules.
* **Component Duplication**: Used Intro1/2/3.js for simplicity; reusable components come later.
* **Props Not Used Yet**: Will be introduced in upcoming lessons.
* **No <a> tags**: Internal links will later be handled using React Router (not raw <a> tags).

**Lecture 8: Importing components**

✅ **Component-Based Architecture in React**

* React apps are built using **self-contained, reusable components**.
* Each component is like a building block used to create powerful UIs.
* Splitting apps into components makes code **organized and manageable**.

✅ **Understanding Modules in JavaScript**

* A **module** is a standalone unit of code that can be reused.
* JavaScript files act as modules.
* Modules allow you to **add, remove, or replace** parts of your code easily.

✅ **Components vs Modules**

* A **component** is usually a small UI piece (e.g., a button).
* A **module** can contain one or more components.
* Components focus on **UI**, while modules focus on **functionality grouping**.

✅ **Exporting and Importing in JavaScript**

* Use the **export** keyword to make code available to other files.
* Use the **import** keyword to use exported code in another file.

**Types of Exports:**

* Default Export: Used when function name matches file name.
* Named Export: Used when function name differs from file name.

✅ **Importing React Components**

* Use the import keyword followed by component name.
* Use from to specify the file location.
* Syntax:

import Header from './Header';

✅ **Structuring React Components**

* Create a folder named \*\*components\*\* to store all component files.
* Helps in grouping and managing similar files together.
* Example:
  + Header.js for top section
  + Main.js for payment section
  + Sidebar.js for sidebar section

✅ **Using Components in Root File (App.js)**

* Each component is **imported and returned** in App.js.
* This forms the main structure of your application page.

**Topic 3:** **Component Use and Styling**

**Lecture 1:** **Principles of components: Props**

✅ **Introduction to Props in React**

* Functional components in React are like JavaScript functions.
* React uses **props** (short for *properties*) to **pass data between components**.
* Props make components **dynamic and reusable**.

✅ **JavaScript Object Refresher**

* A JavaScript object stores related data in **key-value pairs**.
* You access object values using **dot notation**.
* React's **props** work similarly—props are essentially objects.

✅ **What Are Props?**

* Props are **arguments passed to components** using JSX syntax.
* Used like **HTML attributes** in the parent component.
* Accessed inside the component using the props object and dot notation.
* Syntax:

<App title="Welcome!" /> // Parent Component

function App(props) {

return <h1>{props.title}</h1>; // Child Component

}

✅ **Props Accept Different Data Types**

* Props can hold:
  + Strings and numbers
  + Arrays and objects
  + Functions
* This allows **high flexibility and reusability**.

✅ **Component Hierarchy and Data Flow**

* React follows a **parent-child structure**.
* **Parent component** sends props → **Child component** receives.
* One parent can send the same data to **multiple child components**.
* Data flow is **one-directional** (from parent to child only).

✅ **Limitations of Props**

* **Cannot send data from child to parent** using props.
* Props should be **read-only** inside the component.
* React components should behave like **pure functions**:
  + Same inputs (props) should always produce the same output.
  + Components should **never modify their own props**.

**Lecture 2: Using props in components**

✅ **Understanding Props in React**

* Props in React are like **function parameters** in JavaScript.
* They allow **data to be passed** from one component to another.

✅ **JSX is Just Syntactic Sugar**

* JSX is a **nicer way to write** complex JavaScript.
* Browsers don’t understand JSX directly — it must be **transpiled** to plain JavaScript.
* Use the online tool [**babeljs.io**](https://babeljs.io/) to see the **converted JavaScript code**.

✅ **Transpiling JSX to JavaScript**

* Example JSX:

function App() {

return <h1>Hello there</h1>

}

* Transpiled output:

function App() {

return React.createElement("h1", null, "Hello there");

}

✅ **How React.createElement Works**

* React.createElement takes **three arguments**:
  1. **Type of element** (e.g., "h1", "div")
  2. **Props object** (e.g., {} or null)
  3. **Children** (content inside the element)

✅ **Nested JSX Translates to Nested createElement**

* JSX:

return (

<div>

<h1>Hello there</h1>

</div>

)

* Transpiled JavaScript:

return React.createElement(

"div",

null,

React.createElement("h1", null, "Hello there")

);

* **JSX nesting = Nested createElement calls**

✅ **The Second Argument: Props Object**

* null can be replaced with {} to show an empty props object.
* Example:

React.createElement("h1", {}, "Hello there")

* This props object is how you send data from a parent to child component.

✅ **React.createElement Syntax Overview**

React.createElement(

type, // Element type (e.g., "div", MyComponent)

[props], // Optional props object

[...children] // Inner content or nested elements

)

✅ **The Third Argument: Children**

* This is the **inner content** of the component or element.
* Enables **nesting of elements** just like HTML.
* Makes JSX structure **readable and organized**.

**Lecture 3:** **Introducing JSX**

✅ **What Makes JSX Special?**

* JSX is **expressive** – lets you write HTML directly inside JavaScript.
* It resembles **HTML/XML** but works inside JS code.

✅ **Mixing JavaScript, HTML, and CSS in JSX**

* JSX allows **mixing static HTML and dynamic values** using curly braces {}.
* You can pass data dynamically via **props**.
* Curly braces = place for **any JavaScript expression**.

✅ **Creating Components with JSX**

* Components are **JavaScript functions** that return JSX.
* Example: A Nav component can return:

<nav>

<ul>

<li>{dynamicItem}</li>

</ul>

</nav>

✅ **JSX Return Statement Rules**

* JSX must be wrapped in a **single parent element** (e.g., <div>).
* Use <> </> (React **Fragments**) to avoid extra DOM elements.
* For multi-line returns, **wrap JSX in parentheses ()**.

✅ **Styling JSX Elements**

* Use className instead of class (since class is a **JavaScript keyword**).
* className follows **camelCase** syntax.  
  Example:

<div className="container mainContent">Hello</div>

✅ **Why JSX is Easy to Work With**

* JSX looks like HTML but supports **JavaScript flexibility**.
* Helps you create **interactive, styled components** with less effort.

✅ **Conclusion**

* JSX = HTML + JavaScript + styling in one place.
* You now understand how JSX allows you to:
  + Write **HTML structure**
  + Insert **dynamic data**
  + Apply **CSS styles**
  + Follow specific **syntax rules**

**Lecture 4:** **Props and children (Reading)**

✅ **Props and Children in React**  
Understanding how JSX components can receive and render nested elements using props.children.

✅ **What is props.children?**

* It is a special prop automatically passed to every React component.
* It represents whatever is placed between the opening and closing tags of a component.
* Makes components flexible and reusable for various content.

✅ **Real-life Analogy**

* A bag can carry apples or pears; it doesn’t change based on the content.
* Similarly, a component can wrap different child components without changing itself.
* The bag doesn’t care what’s inside—just like props.children.

✅ **Component Example Overview**

Apples Component:

<Apples color="yellow" number="5" />

Pears Component:

<Pears friend="Peter" />

Bag Component using props.children:

function Bag(props) {

return <div style={...}>{props.children}</div>;

}

✅ **Using props.children in JSX**

* You can pass children by placing them between component tags:

<Bag>

<Apples color="yellow" number="5" />

</Bag>

* You can also pass it as a named prop (less common):

<Bag children={<Apples color="yellow" number="5" />} />

✅ **Nesting JSX Elements**

* JSX allows nesting multiple elements:

<Trunk>

<Bag>

<Apples />

<Pears />

</Bag>

</Trunk>

* The Bag component uses props.children to render both Apples and Pears.

✅ **Benefits of Using props.children**

* Makes components more flexible and easier to reuse.
* Enables components to wrap any kind of content, like text or JSX elements.
* Helps in creating layout wrappers, like cards, sections, or containers.

✅ **Important Rules and Considerations**

* Use props.children when a component needs to wrap other components.
* It allows consistent layout or styling while keeping the content dynamic.

✅ **Component Modularity**

* Don’t over-split your layout into tiny components—it becomes harder to manage.
* Don’t keep everything in one big component either—it becomes messy.
* Find a balance by creating components for meaningful sections like header, footer, etc.

✅ **Conclusion**

* Wrapping one component inside another is made possible by props.children.
* It helps you write cleaner and more organized React components.
* This concept supports reusable, modular design in React applications.

**Lecture 5:** **Styling JSX Elements (Reading)**

✅ **Styling JSX Elements**

* JSX allows combining JavaScript, HTML, and CSS.
* You can style JSX in multiple ways for both visual and functional results.

✅ **1. External CSS Styling with className**

* Add a <link> tag in index.html to include a CSS file.
* Use className (not class) in JSX to refer to CSS classes.

**Example:**

function Promo(props) {

return (

<div className="promo-section">

<h1>{props.heading}</h1>

<h2>{props.promoSubHeading}</h2>

</div>

);

}

**CSS:**

.promo-section {

font-weight: bold;

line-height: 20px;

}

✅ **2. Inline Styling in JSX**

* Styles are defined directly in JSX using a JavaScript object.
* Syntax: style={{ property: "value" }}
* CSS properties are camelCased, and values are strings.

**Example:**

<h1 style={{ color: "tomato", fontSize: "40px", fontWeight: "bold" }}>

{props.heading}

</h1>

✅ **3. Saving Inline Styles in a Variable**

* Define a styles object and reuse it in JSX.
* This keeps styles close to the component but may affect maintainability.

**Example:**

function Promo(props) {

const styles = {

color: "tomato",

fontSize: "40px"

};

return (

<div className="promo-section">

<h1 style={styles}>{props.heading}</h1>

<h2>{props.promoSubHeading}</h2>

</div>

);

}

✅ **Important Notes**

* Use inline styles for quick, component-scoped styling.
* Use external CSS or CSS modules for reusable, maintainable styles.
* Avoid mixing too many styling methods in one component.

**Lecture 6:** **Practical styling**

✅ **1. CSS Inclusion Techniques**

* There are **three main ways** to include CSS in HTML or JSX:
  + **Inline CSS**: Using the style attribute directly in an HTML element.
  + **Internal CSS**: Writing CSS inside <style> tags within the <head> of the document.
  + **External CSS**: Linking an external .css file using the <link> element.

✅ **2. Converting External CSS to Internal Styling in JSX**

* CSS styles from an external file (like index.css) can be moved into a component.
* These styles must be converted into a **JavaScript object** to be used inside a React component.

✅ **3. Steps to Apply Internal Styling in a Component**

* **Select relevant CSS code** (e.g., sidebar styles) from index.css.
* **Cut and paste** the code into the respective component file (e.g., Sidebar.js).
* Declare a const style object above the return statement.

✅ **4. Convert CSS to JavaScript Object**

* Replace **semi-colons (;)** with **commas (,)**.
* Convert **hyphenated property names** to **camelCase**.
  + Example: background-color → backgroundColor
* Wrap all **property values** in **double quotes**.

✅ **5. Apply Style to JSX Element**

* Use the style attribute in the JSX tag.
  + Example: <aside style={asideStyle}>
* Save all files to view the updated styles.

**Lecture 7:** **JSX syntax and the arrow function (Reading)**

✅ **1. Component Syntax Options in React**

* React components can be defined in different ways:
  + **Function Declarations**
  + **Function Expressions**
  + **Arrow Functions**

✅ **2. Function Declaration Example**

function Nav(props) {

return (

<ul>

<li>{props.first}</li>

</ul>

)

}

* Traditional ES5 syntax.
* Accepts props and returns JSX.

✅ **3. Function Expression Example**

const Nav = function(props) {

return (

<ul>

<li>{props.first}</li>

</ul>

)

}

* Uses an **anonymous function** assigned to a const.
* Same behavior as a function declaration.

✅ **4. Arrow Function Syntax**

const Nav = (props) => {

return (

<ul>

<li>{props.first}</li>

</ul>

)

}

* Uses => instead of the function keyword.
* Offers a **shorter, modern syntax**.

✅ **5. Arrow Function Syntax Rules**

* **Single Parameter:** Parentheses are optional:

const Nav = props => { ... }

* **No Parameters:** Use empty parentheses:

const Nav = () => { ... }

✅ **6. Implicit Return in Arrow Functions**

* When written in **one line**, return is **automatic**:

const Nav = () => <ul><li>Home</li></ul>

* No need for {} or return.

✅ **7. Using Arrow Functions Elsewhere**

* Arrow functions are also useful in JavaScript array methods:

[10, 20, 30].forEach(item => item \* 10)

* Equivalent ES5 version:

[10, 20, 30].forEach(function(item) {

return item \* 10

})

✅ **8. Summary: Why Use Arrow Functions**

* **Cleaner** and **shorter syntax**
* Useful for **one-liners** and **callbacks**
* Widely used in **JSX and React development**

**Lecture 8: Embedded JSX expressions**

✅ **1. Introduction to JSX**

* JSX is a **syntax extension** for JavaScript used in **React**.
* Allows writing **HTML directly in JavaScript code**.
* Enhances flexibility and readability when building user interfaces.

✅ **2. JSX Example – Outputting Text**

* Example:

const result = <p>Hello World</p>

* result stores a **React element** that can be rendered on the webpage.
* JSX **automatically builds React elements** from HTML-like code.

✅ **3. Embedded Expressions in JSX**

* JSX allows **inserting JavaScript variables or expressions** inside HTML using { }.

✅ **3.1 Using Variables**

* You can insert variable values directly:

jsx

CopyEdit

const name = "John";

const greeting = <h1>Hello, {name}</h1>;

✅ **3.2 Using Functions**

* Functions can also be called inside JSX:

function formatName(user) {

return user.firstName + " " + user.lastName;

}

const user = { firstName: "John", lastName: "Doe" };

const element = <h1>Hello, {formatName(user)}</h1>;

✅ **4. Embedded Expressions in Attributes**

* JSX supports expressions in **HTML attributes**, such as src, alt, etc.

const url = "https://example.com/profile.jpg";

const image = <img src={url} />

* **Double quotes** around values are **not needed** in JSX – they are handled automatically.

✅ **5. JSX Benefits**

* Mixes JavaScript logic and HTML easily.
* Makes UI building more dynamic and efficient.
* Supports variables, functions, and expressions inside HTML elements.

✅ **6. Summary**

* JSX simplifies combining HTML and JavaScript.
* Supports embedded expressions for dynamic content.
* Makes it easy to create **interactive and data-driven UIs** in React.

**Lecture 9:** **Ternary operators and functions in JSX (Reading)**

✅ **1. Understanding Ternary Operators in JavaScript**

✅ **1.1 What is a Ternary Operator?**

* A shorthand version of if...else statements.
* Syntax: condition ? valueIfTrue : valueIfFalse
* Example:

let name = 'Bob';

name == 'Bob' ? console.log('Hello, Bob') : console.log('Hello, Friend');

✅ **1.2 Purpose of Ternary Operators**

* Used to simplify conditions.
* Helps keep JSX concise and readable.

✅ **2. Using Ternary Expressions in JSX**

✅ **2.1 Embedding Conditions in JSX**

* JSX supports JavaScript inside {}.
* Ternary can be used directly within JSX to return dynamic content.
* Example:

function Example() {

return (

<div className="heading">

<h1>{Math.random() >= 0.5 ? "Over 0.5" : "Under 0.5"}</h1>

</div>

);

}

✅ **2.2 How It Works**

* Math.random() returns a number between 0 and 1.
* If it's >= 0.5, display "Over 0.5".
* Else, display "Under 0.5".

✅ **3. Using Functions Inside JSX**

✅ **3.1 Function Invocation in JSX**

* Functions can be called directly inside JSX using {}.
* JSX treats function calls as expressions, which return values.

✅ **3.2 Example – Inline Function Call**

function Example2() {

return (

<div className="heading">

<h1>Here's a random number from 0 to 10: { Math.floor(Math.random() \* 10) + 1 }</h1>

</div>

);

}

✅ **3.3 Example – Extracted Function**

function Example3() {

const getRandomNum = () => Math.floor(Math.random() \* 10) + 1;

return (

<div className="heading">

<h1>Here's a random number from 1 to 10: { getRandomNum() }</h1>

</div>

);

}

✅ **3.4 Alternative Function Forms**

* **Function Expression**

const getRandomNum = function() {

return Math.floor(Math.random() \* 10) + 1;

};

* **Function Declaration**

function getRandomNum() {

return Math.floor(Math.random() \* 10) + 1;

}

✅ **4. Summary**

* Ternary operators offer a clean alternative to if...else inside JSX.
* Functions can be directly called within JSX expressions.
* These features help write dynamic and readable React components efficiently.

**Lecture 10:** **Expressions as props (Reading)**

**✅ Expressions as Props (14px)**

* You can pass **any kind of expression** as a prop in React — not just values.
* Expressions include:
  + Ternary operators
  + Function calls
  + Arithmetic operations
  + Boolean logic

**✅ Using Boolean Expressions as Props (14px)**

* Example:  
  !bool is passed, which evaluates to true because !false is true.
* To **render a boolean** inside JSX, use the .toString() method to convert it to text:

{props.toggleBoolean.toString()}

**✅ Passing Multiple Expressions as Props (14px)**

* You can pass and render multiple props using various expression types.
* Example Props:
  + toggleBoolean – A boolean value.
  + math – An arithmetic expression.
  + str – A string made by **concatenating literals and variables**.

**✅ Arithmetic Expressions in JSX (14px)**

* You can directly write operations like 10 \* 10 as props.
* JSX will evaluate and render the result:

<p>{props.math}</p>

**✅ String Concatenation in JSX Props (14px)**

* Strings can be combined using + operator:

str1 + " another " + "string"

**✅ Summary ()**

* JSX allows you to use **any valid JavaScript expression** as a prop value.
* This includes booleans, strings, numbers, functions, and operations.
* React evaluates the expression and renders the result inside the component.

**Module 2**

**Topic 1: Dynamic events and how to handle them**

**Lecture 1: Types of events**

✅ **Understanding JavaScript Events**

* Events allow JavaScript to interact with HTML.
* Triggered by user or browser actions (e.g., clicks, keypresses).
* Enable enhanced interactivity on web pages.

✅ **How Events Work**

* Events "listen" in the background for interactions.
* HTML elements have associated events accessed via **event listeners**.
* Common example: clicking a button to trigger a function.

✅ **Event Triggering and Handling**

* Triggering: user interaction activates event handler.
* Example: clicking “Add to Cart” updates the cart count dynamically.
* Handlers execute specific code in response to actions.

✅ **Events in the DOM**

* Events are part of the **Document Object Model (DOM)**.
* Understanding events is essential for modern frontend development.

✅ **Introduction to React Events**

* Events in React are handled differently from plain JavaScript.
* Use **JSX event attributes** similar to HTML but with camelCase syntax.

✅ **JSX Event Attributes**

* HTML: onclick
* React JSX: onClick (camelCase format)
* CamelCase means starting with a lowercase letter and capitalizing subsequent words.

✅ **Types of Events in React**

* Events are grouped into categories such as:
  + **Mouse Events**: onClick, onDoubleClick, onContextMenu
  + **Clipboard Events**: onCopy, onCut, onPaste
  + **Keyboard Events** and many others.

✅ **Scope of Event Knowledge**

* Not all events need to be memorized.
* Focus on **commonly used events** relevant to your application needs.
* Many events are tied to specific use cases (e.g., drag-and-drop).

**Lecture 2: Common event handling**

**✅ Creating and Using a New React Component (Btn)**

* A new component named Btn was added to the src folder.
* The App.js file was simplified by:
  + Removing the logo import.
  + Importing the Btn component.
  + Clearing the return content and rendering <Btn />.

**✅ Rendering a Button Element**

* Inside Btn.js, a button element was added with the label: **Click Me**.
* Saved the file using Ctrl + S or Cmd + S, and the button rendered successfully on the screen.

**✅ Handling Click Events with JSX**

* onClick attribute was added to the button with a function reference: {clickHandler}.
* The return statement was reformatted over several lines for better readability.

**✅ Defining the Event Handler Function**

* A clickHandler was created using a **const arrow function**:

const clickHandler = () => {

console.log('clicked');

};

* On button click, the message **"clicked"** appears in the browser console.

**✅ Testing the Click Event**

* Developer tools were opened and the **Console** tab was used.
* Clicking the button printed **"clicked"** to the console each time.

**Lecture 3:** **Syntax for handlers**

**✅ What Are Events in the Browser?**

* Every user interaction like **clicking**, **scrolling**, or **closing a popup** produces an event.
* Events require **event handlers** to execute corresponding actions.
* Example: Click (event) → onClick (handler) → open menu (action).

**✅ HTML vs React Event Handling: Overview**

* HTML and React differ in **event handler syntax** and **implementation approach**.
* React promotes **declarative** coding; HTML often requires direct DOM manipulation.

**✅ HTML Event Handling Syntax**

* Example HTML button:

html

CopyEdit

<button id="js-btn" onclick="clickHandler()">Click Me</button>

* Uses onclick attribute directly within HTML with a function invocation.

**✅ JavaScript DOM-Based Event Handling**

* HTML is typically controlled using JavaScript via the **DOM**.
* Requires two steps:
  1. Select the element using:

const jsBtn = document.getElementById('js-btn');

* 1. Add event listener using:

jsBtn.addEventListener('click', clickHandler);

**✅ Why Prefer JavaScript Over Inline HTML Events**

* JavaScript provides **separation of concerns**, **cleaner logic**, and **flexibility**.
* Inline HTML event handlers are less maintainable and discouraged in modern apps.

**✅ React Event Handling: Key Differences**

* React discourages direct DOM manipulation.
* Uses **JSX event attributes** to handle events declaratively.
* React and HTML event names differ slightly:
  + HTML: onclick
  + React: onClick

**✅ JSX Syntax for Event Handling**

* Example in React:

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

* Use **camelCase** for event names.
* Enclose function name in **curly braces {}**.
* Do **not invoke** the function directly; just **pass the reference**.

**✅ HTML vs React: Event Syntax Comparison**

|  |  |  |
| --- | --- | --- |
| Feature | HTML | React (JSX) |
| Event Name | onclick | onClick |
| Function Call | "clickHandler()" | {clickHandler} |
| Invocation | Function is called in quotes | Function reference only |

**✅ Passing Event Handlers as Props in React**

* You can pass event handler functions to child components using **props**.
* Example:

<Counter onClick={clickHandler} />

* Enables **component-level control** and **reusability**.

**✅ Key Takeaways**

* React and HTML have **similar concepts** but **different syntaxes** for events.
* React emphasizes:
  + Declarative syntax
  + JSX expressions
  + Avoiding direct DOM manipulation

**Lecture 4:** **Event handling and embedded expressions (Reading)**

**✅ Different Ways to Handle Events in React**

* You can handle events in React using **different types of functions**.
* All approaches produce the same result (e.g., logging to console), but differ in **syntax and readability**.

**✅ 1. Inline Anonymous ES5 Function**

* **Syntax:**

<button onClick={function() { console.log('first example') }}>

An inline anonymous ES5 function event handler

</button>

* ✅ Rarely used in React.
* ❌ Verbose and not idiomatic for modern React code.

**✅ 2. Inline Anonymous ES6 Arrow Function**

* **Syntax:**

<button onClick={() => console.log('second example')}>

An inline anonymous ES6 function event handler

</button>

* ✅ Very common and concise.
* ✅ Good for short, simple logic.
* ❌ Can reduce readability if logic gets too long.

**✅ 3. Separate Function Declaration**

* **Syntax:**

function App() {

function thirdExample() {

console.log('third example');

}

return (

<button onClick={thirdExample}>

using a separate function declaration

</button>

);

}

* ✅ Best for **complex logic** or **multi-line functions**.
* ✅ Improves **readability** and **maintainability**.
* ✅ Suitable when following a **clean coding style**.

**✅ 4. Separate Function Expression (Arrow Function Assigned to Const)**

* **Syntax:**

function App() {

const fourthExample = () => console.log('fourth example');

return (

<button onClick={fourthExample}>

using a separate function expression

</button>

);

}

* ✅ Most **modern** and **preferred** pattern in many React apps.
* ✅ Combines arrow function syntax with cleaner modular structure.
* ✅ Ideal for functions that grow in complexity.

**✅ Quick Tip: Declaration vs Expression**

* **Declaration**: Starts with function keyword.
* **Expression**: Assigned to a variable (const, let), often an arrow function.

**✅ Key Takeaways**

* **Use inline arrow functions** for short actions.
* **Use separate functions** for reusable or multi-line logic.
* **Understand all styles** to read and adapt code across different teams or companies.
* Pick the **most appropriate style** based on readability, complexity, and code guidelines.

**Topic 2: Data and Events**

**Lecture 1:** **Parent-child data flow**

✅ **Understanding One-Way Data Flow in React**

* React enforces **unidirectional (top-to-bottom)** data flow.
* Data moves **from parent to child** through the component hierarchy.
* Ensures predictable behavior and controlled data transmission.

✅ **Why One-Way Data Flow Matters**

* Prevents unexpected side effects and makes debugging easier.
* Centralizes data management in parent components.
* Supports a clean, scalable component structure.

✅ **Props as a Data Carrier (Stateless)**

* Props are **read-only** data passed from **parent ➝ child**.
* Child components **cannot modify** props — only consume and render them.
* Analogy: *Props = money given to you by employer (parent); you can use, not change the source*.

✅ **State for Internal Control (Stateful)**

* **State** is internal, **mutable** data managed by the component itself.
* Unlike props, components can **change their own state** using setState.
* Helps manage dynamic, interactive behavior in apps.

✅ **Difference Between Props and State**

|  |  |
| --- | --- |
| Props | State |
| Passed from parent | Managed internally |
| Read-only | Can be updated |
| Used to render content | Used to render and change |

✅ **Code Example Summary**

* **App.js**: A **class component** with internal **state** holding current date/time.
* **Child.js**: A **function component** that receives message as **prop** and displays it in <h1>.
* App's state ➝ passed as props ➝ consumed in child ➝ displayed in UI.

✅ **Stateless vs. Stateful Components in Practice**

* Stateless: Only **render data** passed to them (Child.js).
* Stateful: **Manage and update data** (App.js) and pass to children.

✅ **React Data Flow Recap**

* **Props**: Passed from **parent ➝ child**, immutable in child.
* **State**: Exists within a component, mutable by the same component.
* Together, props and state support **structured and interactive** UIs.

**Lecture 2:** **What Are Hooks?**

✅ **Introduction to React Hooks**

* Hooks add **interactivity** and **state management** to functional components.
* Introduced in **React 16.8**.
* Help manage **complex component logic** and reduce **code duplication**.

✅ **What Are Hooks?**

* Hooks are **functions** that let you "hook into" React features like state and lifecycle methods.
* Can only be used in **functional components** (not class components).
* Eliminate the need for HOCs and render props for logic reuse.

✅ **Using the useState Hook**

* useState is the **most common** hook.
* Allows you to **add state** to functional components.
* Must be **imported** from 'react':

import { useState } from 'react';

✅ **Declaring State with useState**

* Syntax (with array destructuring):

const [showMenu, setShowMenu] = useState(false);

* showMenu: state variable
* setShowMenu: function to update the state
* false: initial value

✅ **Why Array Destructuring?**

* useState returns an array: [currentState, setStateFunction]
* Destructuring improves **code readability** and **simplicity**
* Without it, you'd need to access items via index (e.g. state[0]), which is less clear.

✅ **Types of State You Can Track**

* Works with **Booleans, Strings, Numbers, Arrays, Objects**, etc.
* Example: Count button clicks with useState(0).

✅ **Best Practices with useState**

* Call at the **top level** of the component.
* **Don't call** hooks inside loops, conditions, or nested functions.
* You can use **multiple useState** calls in a single component.

✅ **Custom Hooks**

* You can create **your own hooks** to reuse logic across components.
* Improves **code modularity and reusability**.

✅ **Benefits of Hooks**

* Enhance **readability** and **simplicity**.
* Reduce need for **boilerplate code**.
* Enable better **logic organization** in functional components.

**Lecture 3:** **Using Hooks**

**✅** **Using Hooks in React**

* Hooks let you manage **state and side effects** in functional components.
* useState is used to **store and update local state** (like text input).
* State is **local to the component** – not accessible from outside.

**✅ Example: InputComponent with useState**

* Uses useState to track input value:

const [inputText, setText] = useState('hello');

* **handleChange function** updates state on user input:

function handleChange(e) {

setText(e.target.value);

}

* A **reset button** sets the state back to 'hello'.

**✅ State is Local**

* The variable inputText exists **only within** InputComponent.
* State in React is **component-scoped**, not global.

**✅ Rules of Using Hooks**

1. ✅ Call hooks **at the top level** of your component or custom hook.
2. ❌ Don't call hooks inside **loops, conditions, or nested functions**.
3. ✅ Hooks can only be called from **React function components** or **custom hooks**.

**✅ Multiple Fields: RegisterForm Example**

* Use **one useState call** for multiple form fields:

const [form, setForm] = useState({

firstName: 'Luke',

lastName: 'Jones',

email: 'lukeJones@sculpture.com'

});

* **Better readability** and easier updates than having multiple states.

**✅ Custom Hooks**

* Extract **reusable logic** into custom hooks.
* Examples: handling forms, timers, API calls, etc.

**✅ useRef Hook**

* Used to **access DOM elements** directly.
* Returns a **ref object** with a current property.
* Example: Focus an input field:

const inputEl = useRef(null);

inputEl.current.focus();

**Lecture 4:** **What Is State?**

**✅ React State vs Props**

* **Props**: Passed **from parent to child** to customize UI.
* **State**: Internal **data of a component**, can change over time.

**✅ Alarm Clock Analogy**

* Modes like **Alarm On**, **Alarm Off**, **Snooze** = internal behavior.
* In React, these modes can be handled using **State**, not just props.

**✅ What Is State?**

* **Internal data** in a component that controls **behavior & rendering**.
* Useful for dynamic content (e.g., toggles, counters, form inputs).
* State changes trigger **re-render** of component.

**✅ Why Use State?**

* Allows components to **track data** over time.
* Keeps **UI in sync** with data.
* Components **update automatically** when state changes.
* Enables **parent-to-child updates** via props.

**✅ Stateless vs Stateful Components**

* **Stateless**: Just render static content, no internal state.
* **Stateful**: Use useState() to manage dynamic behavior.

**✅ useState Hook Explained**

* Syntax:

js

CopyEdit

const [greet, setGreet] = useState("Hello");

* greet: current state value.
* setGreet: function to **update the state**.
* Changing state: setGreet("Hi!") triggers re-render.

**✅ Behind the Syntax – Array Destructuring**

* useState("Hello") returns an array:

["Hello", function to update]

* ES6 destructuring:

const [greet, setGreet] = useState("Hello");

**✅ Best Practices**

* Match state variable & updater function names:  
  Example: count & setCount, user & setUser
* Place useState **at the top** of the function.
* Avoid placing hooks **inside loops or conditions**.

**Lecture 5:** **Management Matters**

**✅ Why State Management Matters**

* As apps grow, managing state **across components** becomes more complex.
* Some components may need access to the **same piece of state**, even if they’re not directly related.

**✅ Example Scenario**

* App that tracks a **daily meal plan**:
  + **App.js** (Root)
  + **MealsList** (Lists today's meals)
  + **Counter** (Shows meals left to eat)
* Problem: Counter needs data from MealsList, but they’re **sibling components**.

**✅ The Problem**

* **Siblings** can’t directly share state.
* State in MealsList cannot be accessed directly by Counter.

**✅ Solution: Lifting State Up**

* Move state **up to the common ancestor**, in this case, App.js.
* Pass state down via **props**:
  + App → MealsList
  + App → Counter
* This enables both child components to share the same state.

**✅ Prop Drilling**

* When props are passed down through **multiple nested components**.
* Example: App → MealsList → MealItem.
* Problems with prop drilling:
  + Becomes **hard to manage** as component tree grows.
  + Components may receive props they **don’t directly use**.
  + Small change can lead to **many re-renders**.

**✅ Global State Problem**

* Some state (like meals) is needed in **many parts of the app**.
* If state is kept in App.js, it becomes **bloated**.
* Not all state belongs in the top-level component.

**✅ Elegant Solution: React Context API**

* Provides a way to **share state globally** without prop drilling.
* Components can **consume context directly**, no middleman.
* Steps:
  1. **Create a Context**
  2. **Wrap** your app with a **Context Provider**
  3. **Use useContext()** in any component to access the state

**✅ Benefits of Context API**

* **No prop drilling**.
* Centralized **global state**.
* Cleaner and more **scalable** code.

**Lecture 6: React Context API**

✅ **Challenges of State Management in Complex React Apps**

* Managing state between parent and child components becomes harder as component levels increase.
* Prop drilling (passing props down multiple levels) becomes inefficient and messy.

✅ **Introducing React Context API**

* Context API provides a cleaner way to **share state across multiple levels** without prop drilling.
* It works like "teleporting" state directly to components that need it, skipping intermediate levels.

✅ **Setting Up the Context API**

* Create context using React.createContext().
* Define a **Provider component** to wrap parts of the app that need access to the state.
* The Provider holds the state and shares it via the value prop.
* Components that consume this state use useContext() to access it.

✅ **Using Context API: Example Overview**

* App.js imports:
  + MealsProvider (Context Provider)
  + MealsList and Counter (components that use the context)
* MealsProvider:
  + Defines the context and initial state (todayMeals array).
  + Wraps children in MealsContext.Provider and passes the meals value.
  + Also defines a custom hook useMealsListContext to simplify context access.

✅ **Accessing Context in Components**

* In MealsList, use the useMealsListContext hook to get meals.
* Destructure the array and use .map() to render meals as <h2> elements.
* Counter accesses the same context using the same hook — **no prop drilling needed**.

✅ **Benefits of Using Context API**

* Centralized state management.
* No need for lifting state up or prop drilling.
* Cleaner, more maintainable code structure.

✅ **Enhancing State Logic with useReducer Hook**

* UseReducer is like a **supercharged version of useState**.
* Accepts:
  + An initial state.
  + A **reducer function** to define how state should change.
* dispatch() is used to send actions that the reducer processes.
* Great for **complex state transitions**, such as a rideshare wallet:
  + Add money on customer pickup.
  + Subtract money on refueling.

✅ **Key Takeaways**

* Use Context API to share state **globally** across components.
* Use useContext() for **simple access** to shared state.
* Use useReducer() for **complex state logic**.
* These tools eliminate the need for prop drilling and simplify managing state in large apps.

**Example:**

**✅ 1. MealsContext.js (Context + Provider + Reducer)**

// MealsContext.js

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

// Initial state

const initialState = {

meals: ['Salad', 'Chicken', 'Rice'],

consumed: 0,

};

// Reducer function

function mealsReducer(state, action) {

switch (action.type) {

case 'CONSUME\_MEAL':

return {

...state,

consumed: state.consumed + 1,

};

default:

return state;

}

}

// Create Context

const MealsContext = createContext();

// Provider component

export function MealsProvider({ children }) {

const [state, dispatch] = useReducer(mealsReducer, initialState);

return (

<MealsContext.Provider value={{ state, dispatch }}>

{children}

</MealsContext.Provider>

);

}

// Custom hook for consuming context

export function useMealsContext() {

return useContext(MealsContext);

}

**✅ 2. MealsList.js (List meals and consume them)**

// MealsList.js

import React from 'react';

import { useMealsContext } from './MealsContext';

function MealsList() {

const { state, dispatch } = useMealsContext();

const { meals } = state;

return (

<div>

<h2>Today's Meals</h2>

{meals.map((meal, index) => (

<div key={index}>

<h3>{meal}</h3>

<button onClick={() => dispatch({ type: 'CONSUME\_MEAL' })}>

Mark as Consumed

</button>

</div>

))}

</div>

);

}

export default MealsList;

**✅ 3. Counter.js (Show number of meals consumed)**

// Counter.js

import React from 'react';

import { useMealsContext } from './MealsContext';

function Counter() {

const { state } = useMealsContext();

const total = state.meals.length;

const remaining = total - state.consumed;

return (

<div>

<h2>Meals Left: {remaining} / {total}</h2>

</div>

);

}

export default Counter;

**✅ 4. App.js (Wrap everything with the Provider)**

// App.js

import React from 'react';

import { MealsProvider } from './MealsContext';

import MealsList from './MealsList';

import Counter from './Counter';

function App() {

return (

<MealsProvider>

<div className="App">

<h1>Healthy Meal Tracker</h1>

<MealsList />

<Counter />

</div>

</MealsProvider>

);

}

export default App;