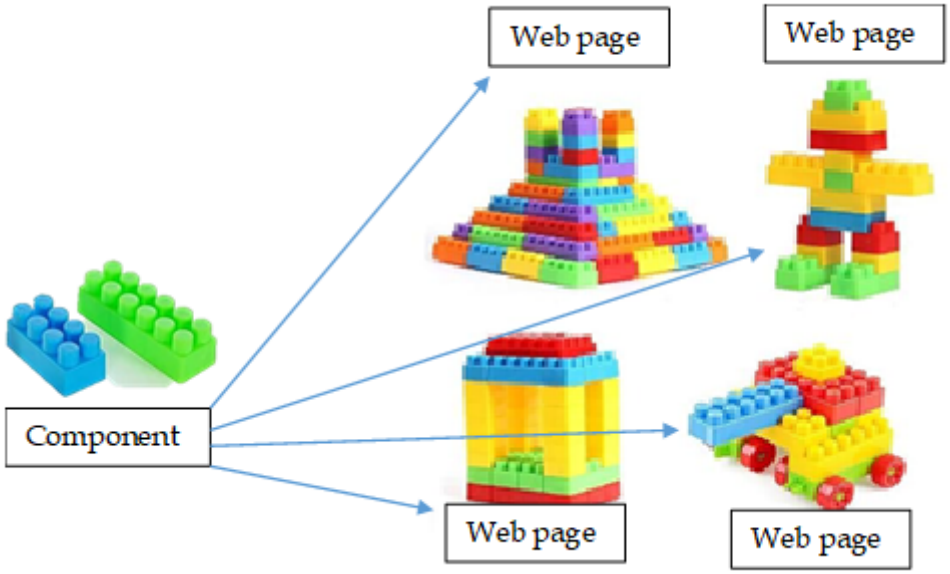
Program Name: BCA Semester V
Course Name & Code: Full-stack Development-II & BCA57116
Class: BCA2023
Academic Session: 2025-26

Study Material

(React CSS with Bootstrap package and router concepts)

Table of Contents

2.1 Components in React



The provided image effectively illustrates the concept of components in web development. On the left, you see a single "Component" represented by two individual building blocks (like Lego bricks). On the right, these same individual "Components" are used to construct various distinct "Web pages" or larger UI elements.

This visual perfectly reinforces the explanation:

- **Reusability:** The single "Component" (the building block) is used multiple times to create different structures (the various "Web pages"). This shows that you define a component once and then reuse it wherever needed, rather than writing the same code repeatedly.
- **Independence:** Each building block is a distinct unit, just as a component is a self-contained piece of code. It has its own properties and behavior.
- **Building Blocks:** The image clearly demonstrates how smaller, independent "Components" combine to form larger, more complex "Web pages." This highlights that a web application is essentially an assembly of these individual components.
- **Reduced Redundancy:** If you needed to change the color or shape of that specific building block, you'd only change the design of the "Component" itself, and all instances of it in the "Web pages" would automatically update. This directly translates to reducing code redundancy in software development.

In React, **Components** are the fundamental building blocks of your user interface. They are self-contained, independent, and reusable pieces of code that encapsulate both UI (what it looks like) and behavior (how it acts). This modular approach significantly reduces code redundancy and improves maintainability.

React primarily offers two types of components:

2.1.1 Functional Components

2.1.2 Class Components

2.1.1 Functional Components

A **React Functional Component** is a JavaScript function that takes an object of props (properties) as its argument and returns React elements, which are essentially a description of what should appear on the screen (often written using JSX).

It's the modern and most common way to write components in React due to its simplicity, readability, and the powerful capabilities brought by React Hooks.

Core Characteristics:

- **Just a JavaScript Function:** At its heart, a functional component is a plain, regular JavaScript function. It doesn't use classes or this keyword contexts in the same way class components do.
- **Receives props as an Argument:** Any data passed down from a parent component to a functional component is received as an object in its first (and usually only) argument, typically named props.
- **Returns JSX:** A functional component's primary job is to return JSX. JSX is a syntax extension for JavaScript that looks a lot like HTML, but it allows you to write UI structures directly within your JavaScript code. React then understands this JSX and uses it to construct the actual UI elements.
- **Simplicity and Readability:** Because they are simple functions, functional components are often more concise and easier to read and understand than their class-based counterparts.
- **State and Lifecycle with Hooks:** Historically, functional components were "stateless" and couldn't manage their own internal data or perform side effects (like data fetching) without being converted to class components. However, with the introduction of **React Hooks** (e.g., useState, useEffect, useContext), functional components gained the full power to manage state, handle lifecycle events, and interact with the React context system. This made them equally, if not more, capable than class components.
- **No this Keyword:** The absence of the this keyword within functional components removes a common source of confusion and bugs for developers, especially those new to JavaScript or React.

Here's the basic syntax for a React functional component:

```
import React from 'react';// Optional for React 17+, but good practice
// Define the component as a JavaScript function
// It typically takes 'props' as its only argument
function MyFunctionalComponent(props) {
 // Component logic can go here (e.g., using Hooks like useState)
 // Returns JSX (JavaScript XML) which describes the UI
 // Must return a single root element (or a Fragment <></>)
 return (
   <div>
     <h1>Hello from a Functional Component!</h1>
     {/* You can access props like props.someValue */}
     {/* You can embed JavaScript expressions using curly braces {} */}
   </div>
 );
}
// Export the component so it can be used in other files
export default MyFunctionalComponent;
```

Key Points:

- It's a standard JavaScript function.
- The component name (MyFunctionalComponent) **must start with an uppercase letter**.
- It takes props (properties) as an argument.
- It returns **JSX**, which looks like HTML but is JavaScript.
- The JSX must have a **single root element**.
- It's exported so other files can import it.

The provided example perfectly illustrates the core of a React Functional Component:

```
import React from 'react'; // Imports the React library

function App() { // Defines a functional component named 'App'
  const greeting = 'Hello Function Component!'; // Declares a JavaScript variable

  return <h1>{greeting}</h1>; // Returns JSX, embedding the JS variable with curly braces
}
export default App; // Exports the component for use in other parts of the application
```

This App component demonstrates the essential characteristics:

- **A Plain JavaScript Function:** function App() { ... } defines the component.
- **Returns JSX:** The return <h1>{greeting}</h1>; statement specifies the UI structure.
- **JavaScript within JSX:** {greeting} showcases how JavaScript variables (or any valid JavaScript expression) are seamlessly embedded within the HTML-like JSX using curly braces. Here, the greeting variable's value dynamically populates the h1 tag.

Hello Function Component!

When you want to render a React Component inside another (typically a parent) Functional Component, you simply define the child component and then use it as a React element within the parent's JSX.

Example for clarity:

```
import React from 'react';
// Child Component: Headline
// This is a simple functional component that renders a greeting.
function Headline() {
  const greeting = 'Hello Function Component!';
  return <h1>{greeting}</h1>;
}

// Parent Component: App
// This functional component renders the 'Headline' component as a React element.
```

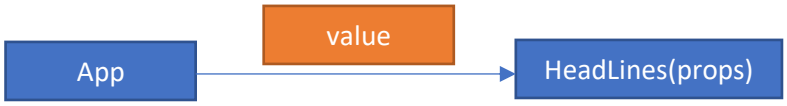
```
function App() {
  // By using <Headline />, we are instructing React to render the Headline component's output here.
  return <Headline />;
}

export default App;
```

It's the core React practice of building complex UIs by combining smaller, independent **child components** (like Headline) within larger **parent components** (like App). Embed children by rendering them as React elements (<Headline />) in the parent's JSX. This approach champions **modularity, reusability, and simplified development**, akin to building with Lego bricks.

React Function Component: Props

In React, **props** (short for properties) are how you pass information from a parent component down to a child component. You can think of props as the "parameters" or "arguments" for your React Function Components, allowing them to remain generic while receiving specific data or configurations from their parent.



Key Principle: Props are always passed **down** the component tree, ensuring a clear and unidirectional data flow.

Let's look at an example:

```
import React from 'react';

// Parent Component
function App() {
  const greeting = 'Hello Function Component!';

  // Here, 'value' is a prop being passed to the Headline component.
  // The attribute 'value' in JSX becomes a property on the 'props' object.
  return <Headline value={greeting} />;
}

// Child Component
function Headline(props) { // 'props' is the first argument, an object containing all passed properties
  return <h1>{props.value}</h1>; // Accessing the 'value' property from the 'props' object
}

export default App;
```

When rendering a component (e.g., Headline inside App), you pass props as HTML-like attributes (value={greeting}). Inside the receiving Function Component (Headline), the entire set of passed props is available as an object, typically named props, as the first argument in the function signature.

Props are Read-Only: It's crucial to remember that props should be treated as read-only within the component that receives them. A component should never modify its own props; instead, if data needs to change, it should be managed as state within the component itself or passed down from a parent that manages the state.

Using JavaScript Object Destructuring for Props

Since props are always an object, and you often need to extract specific pieces of information from it, JavaScript's object destructuring comes in very handy. You can directly destructure the props object in the function signature for a cleaner and more concise way to access the values:

```
import React from 'react';

function App() {
  const greeting = 'Hello Function Component!';

  return <Headline value={greeting} />;
}

// Destructuring 'value' directly from the 'props' object in the function signature
function Headline({ value }) {
  return <h1>{value}</h1>; // Now you can use 'value' directly
}

export default App;
```

Important Note on Destructuring Syntax: If you try to access props by defining multiple arguments like function Headline(value1, value2) { ... }, you will likely encounter "props undefined" messages or unexpected behavior. This is because props is always received as a single object (the first argument). You must destructure from that object: function Headline({ value1, value2 }) { ... }.

React Arrow Function Component

With the introduction of **JavaScript ES6 (ECMAScript 2015)**, new syntax and coding concepts became available in JavaScript, which naturally extended to React development. One significant addition was the **arrow function** (often called a lambda function or fat arrow function).

Because of this, a React Function Component is very commonly defined using arrow function syntax, leading to terms like **Arrow Function Component** or **Arrow Function Expression Component**.

Let's look at our previous example refactored to use arrow function expressions:

```
import React from 'react';

// App component defined using an arrow function with a block body
const App = () => {
  const greeting = 'Hello Function Component!'; // Internal variable declaration

  return <Headline value={greeting} />; // Returns JSX
};

// Headline component defined using an arrow function with a block body
const Headline = ({ value }) => { // Props are destructured here
  return <h1>{value}</h1>; // Returns JSX
};

export default App;
```

In the example above, both App and Headline components use a function block body (enclosed in curly braces {}). However, for components that **only return JSX without any other logic** (like Headline in this case), arrow functions allow for an even more concise syntax: the **concise body (or implicit return)**. When you omit the curly braces {} and the return keyword, the expression immediately following the arrow (=>) is implicitly returned.

```
import React from 'react';

// App component (still with a block body because of the 'greeting' variable)
const App = () => {
  const greeting = 'Hello Function Component!';
  return <Headline value={greeting} />;
};

// Headline component using a concise body (implicit return)
// It only returns the JSX directly
const Headline = ({ value }) =>
  <h1>{value}</h1>; // No curly braces or 'return' keyword needed

export default App;
```

When using arrow functions for React components, the way props are handled remains exactly the same. They are still accessible as the function's arguments (e.g., as props or destructured directly as { value }). The choice between a traditional function declaration and an arrow function for defining a React component is primarily a matter of stylistic preference and conciseness, as both create a React Functional Component

- **"Stateless" Functional Component (Purely Presentational)**

Before Hooks, if a functional component didn't need to manage any internal state and simply rendered UI based on the props it received, it was referred to as "stateless" or a "purely presentational component." They primarily focused on *how* things look.

Characteristics (Historical Context):

Historically, React's **Functional Components** were characterized by their simplicity and limitations. They were designed **without internal state**, meaning they could not manage this.state or use setState(). Consequently, they also lacked access to **lifecycle methods** like componentDidMount or componentDidUpdate, making them unable to perform side effects tied to a component's mounting, updating, or unmounting. Instead, they **relied entirely on data passed down from parent components via props**. This made them **pure functions**; given the same props, they would consistently render the same output without any side effects. Their straightforward nature made them generally **simpler to write and test**.

Due to these characteristics, functional components were historically used primarily for **displaying data, simple formatting, or wrapping other components without requiring complex logic or internal state management**. They were often referred to as "presentational" or "stateless" functional components.

Example (Pre-Hooks "Stateless" Functional Component):

```
import React from 'react';
```

```
// This component is purely presentational.
// It just takes a 'title' prop and displays it.
function DisplayTitle(props) {
  return (
    <h2>{props.title}</h2>
  );
}

// Example usage in a parent component
function App() {
  return (
    <div>
      <DisplayTitle title="My Static Heading" />
      <DisplayTitle title="Another Title Here" />
    </div>
  );
}

export default App;
```

- **"Stateful" Functional Component (with Hooks)**

With the introduction of **React Hooks** (starting from React 16.8), functional components gained the ability to manage state and perform side effects. This effectively made the "stateless" vs. "stateful" distinction less about the *type* of component (functional vs. class) and more about whether the component *uses* state.

A functional component that utilizes Hooks like useState to manage internal data that changes over time is now a "stateful" functional component. They can manage *what* happens and *how* things change.

Characteristics (Modern Context):

Modern Functional Components leverage Hooks like useState to manage internal state and useEffect for handling side effects (like data fetching), essentially gaining capabilities once exclusive to class components. They remain pure JavaScript functions, offering superior readability and conciseness. This makes them the **preferred method** for almost all React development scenarios, from simple displays to complex forms, data fetching, and highly interactive elements.

Example (Modern "Stateful" Functional Component using useState):

```
import React, { useState } from 'react'; // Import the useState Hook

// This component is stateful because it manages its own 'count' state.
function Counter() {
  // 'count' is the state variable, initialized to 0.
  // 'setCount' is the function used to update 'count'.
  const [count, setCount] = useState(0);
```

```
 return (
   <div>
     <h2>Current Count: {count}</h2>
     <button onClick={() => setCount(count + 1)}>Increment</button>
     <button onClick={() => setCount(0)}>Reset</button>
   </div>
 );
}

// Example usage in a parent component
function App() {
 return (
   <div>
     <h1>My Interactive App</h1>
     <Counter /> {/* This instance of Counter manages its own state */}
     <Counter /> {/* This is another independent instance, also managing its own state */}
   </div>
 );
}

export default App;
```

React Function Component: Event Handlers

In React, **event handlers** are functions that allow your components to respond to user interactions, such as typing, clicking, or hovering. For instance, the onChange event handler is perfect for input fields, triggering a function every time the field's value changes. React provides a consistent set of event handlers that map to common HTML events, including onClick (for buttons, etc.), onMouseDown, onBlur, and many more.

While you can define event handlers inline using arrow functions directly within JSX, it's often cleaner and more maintainable to **extract them into named functions** within your component's body. This improves readability, especially for more complex logic.

Let's illustrate this with an example:

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

// Parent component (remains unchanged)
const App = () => {
 return <Headline />;
};

// Child component demonstrating an extracted event handler
const Headline = () => {
 // State to manage the greeting text
 const [greeting, setGreeting] = useState('Hello Function Component!');
```

```
  // Extracted event handler function.
  // It receives the synthetic event object from React.
  const handleChange = event => {
    // Update the 'greeting' state with the current value of the input field
    setGreeting(event.target.value);
  };

  return (
    <div>
      <h1>{greeting}</h1>
      <input
        type="text"
        value={greeting}      // The input's value is controlled by our 'greeting' state
        onChange={handleChange} // Attach the named function as the event handler
      />
    </div>
  );
};

export default App;
```

Key Takeaways:

Defining functions directly inside a Functional Component acts as the "methods" for that component, granting full access to its props and state. This practice significantly boosts organization and reusability by encapsulating logic. Furthermore, it enables the controlled component pattern, where form input values are explicitly managed by React state, giving you full control over their behavior.

React Function Component: Callback Functions (Passing Functions as Props)

In React, props are not just for passing data like strings or numbers; you can also pass **functions as props**. This is a fundamental pattern for enabling child components to communicate back to their parent components, or for sharing logic across different parts of your application.

Let's illustrate how a function defined in a parent component can be passed down as a prop and executed by a child:

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

const App = () => {
  const [greeting, setGreeting] = useState('Hello Function Component!');

  // This function is defined in the App (parent) component
  const handleChange = event => setGreeting(event.target.value);

  return (
    // The handleChange function is passed as a prop named 'onChangeHeadline'
    <Headline headline={greeting} onChangeHeadline={handleChange} />
  );
```

```
};

const Headline = ({ headline, onChangeHeadline }) => (
  <div>
    <h1>{headline}</h1>
    {/* The function received via props is executed when the input changes */}
    <input type="text" value={headline} onChange={onChangeHeadline} />
  </div>
);

export default App;
```

In this setup, the Headline component doesn't manage its own greeting state. Instead, it receives the headline value and the onChangeHeadline function from its parent, App. When the input field within Headline changes, it calls the onChangeHeadline prop, which in turn executes handleChange in the App component, updating the greeting state. This demonstrates how a child can trigger a state change in its parent.

You can also execute additional logic within the child component's event handler before calling the prop function (e.g., trimming the value or performing validation).

Sharing Functions Between Sibling Components

This pattern becomes even more powerful when you need to share a function or state management logic between **sibling components**. The common approach is to lift the shared state and the function that modifies it up to their closest common parent. The parent then passes the relevant state and the function down as props to the respective children.

Consider this example with Headline and Input as sibling components:

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

const App = () => {
  const [greeting, setGreeting] = useState('Hello Function Component!');

  // This function is defined in the App (parent) component
  const handleChange = event => setGreeting(event.target.value);

  return (
    <div>
      {/* Headline component receives the current greeting value */}
      <Headline headline={greeting} />

      {/* Input component receives the current greeting value AND the function to change it */}
      <Input value={greeting} onChangeInput={handleChange}>
        Set Greeting:
      </Input>
    </div>
  );
```

```
};

// Child component: Headline (displays the value)
const Headline = ({ headline }) => <h1>{headline}</h1>;

// Child component: Input (modifies the value by calling the parent's function)
const Input = ({ value, onChangeInput, children }) => (
  <label>
    {children} {/* Renders "Set Greeting:" */}
    <input type="text" value={value} onChange={onChangeInput} />
  </label>
);

export default App;
```

This example perfectly illustrates the core principle:

- The App (Parent Component) centrally manages the greeting state and the handleChange logic.
- Headline (Sibling Component) receives greeting as a prop to *display* it.
- Input (Sibling Component) receives greeting (as value) and the handleChange function (as onChangeInput) to *modify* it.

2.1.2 Class Components in React

Class components in React are fundamental building blocks for creating **reusable, stateful, and interactive user interfaces**. Unlike functional components, class components can manage their own state and lifecycle methods, making them a preferred choice for more complex applications before the introduction of React Hooks.

- It manages its own state using this.state.
- Support lifecycle methods like componentDidMount for added control.
- Access props and state using this keyword.

- **Basic Class Components of React**

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <div>
        <h1>Welcome to React!</h1>
        <p>This is a simple class component.</p>
      </div>
    );
  }
}
export default App;
```

In this Example

- **Class Component**: The App component is created as a class that extends React.Component, allowing it to use React's features like rendering UI.
- **render Method**: The render() method defines what the component displays, returning JSX (HTML-like syntax).
- **Exporting the Component:** The export default App; makes this component usable in other parts of the application.

2.2 Understanding createElement in React

2.2.1 What is createElement?

React.createElement is a core React function used to create React elements, which are the fundamental building blocks of a UI structure. While developers commonly use JSX for a more declarative syntax, JSX code is ultimately compiled into calls to React.createElement. This function allows for the programmatic construction of React elements, defining components, props, and children without relying on JSX syntax.

Syntax of React.createElement

The createElement function takes three primary arguments:

```
React.createElement(
    type, // The type of element (string for HTML tags, function/class for components)
    props, // An object containing element attributes or properties
    children // The child elements or text content inside the element
);
```

These three arguments are crucial in defining any React element object.

Example: Creating a Simple React Element

```
const heading = React.createElement(
    "h1",
    { className: "title" },
    "Hello, React!"
);
```

This function call generates an object representing a React element, which later gets rendered into the actual DOM.

2.2.2 Creating Multiple and Nested Elements

Creating a Parent-Child Structure

To generate nested elements, pass multiple child elements inside createElement.

```
const container = React.createElement(
    "div",
    { className: "container" },
    React.createElement("h1", null, "Welcome"),
    React.createElement("p", null, "This is a paragraph.")
);
```

Handling Many Nested Elements

When working with many nested elements, using React.createElement helps maintain a structured UI without JSX.

```
const nestedStructure = React.createElement(
        "div",
        null,
        React.createElement("header", null, React.createElement("h1", null, "Header")),
        React.createElement("main", null, React.createElement("p", null, "Content")),
        React.createElement("footer", null, React.createElement("small", null, "Footer"))
);
```

2.3 Managing Props and Children

Passing Props

Props define an element's behavior and appearance. The props object contains various attributes assigned to an element.

```
const button = React.createElement(
        "button",
        { onClick: () => alert("Clicked!") },
        "Click Me"
);
```

Using Null for Empty Props

You can pass null when no attributes are required.

```
const simpleText = React.createElement("p", null, "This is a paragraph.");
```

Working with Multiple Children

To include multiple elements inside a parent, list them as additional third arguments.

```
const list = React.createElement(
        "ul",
        null,
        React.createElement("li", null, "Item 1"),
        React.createElement("li", null, "Item 2"),
        React.createElement("li", null, "Item 3")
);
```

React DOM: React DOM contains the arguments that are necessary to render react elements in the browser.

```
ReactDOM.render(element,containerElement);
```

Parameters: ReactDOM.render() takes two arguments:

- **element:** The element that needs to be rendered in the DOM.
- **containerElement:** Where to render in the dom.

Example: This code in Index.js shows the use of react.createEement method and render that component using ReactDOM.render method.

// Filename - index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
let demo = React.createElement(
   "h1", { style: { color: "green" } }, "Welcome to GeeksforGeeks"
)
ReactDOM.render(
   demo,
   document.getElementById('root')
);
```

2.4 JSX in React - An Overview

React's JSX (JavaScript XML) is a syntax extension that enables programmers to construct JavaScript code resembling HTML. It simplifies user interface creation by offering a more declarative and intuitive approach. Developers leverage JSX to define reusable React components by blending JavaScript with HTML-like syntax. Before presentation to the DOM, JSX code is converted into standard JavaScript by transpilers like Babel, enabling dynamic content and efficient updates. This fusion of JavaScript's power with the familiar HTML structure significantly accelerates UI development in React applications.

2.4.1 What is JSX?

JSX is a special way to write JavaScript code that looks a lot like HTML. It makes building parts of your website (called UI components) in React much simpler and clearer. With JSX, you can easily mix HTML designs with the smart actions of JavaScript. This lets you put changing information, like names or numbers, directly into your web page's design.

Syntax:

```
const element = <h1>Hello, world!</h1>;
```

<h1>Hello, world!</h1> is a JSX element, similar to HTML, that represents a heading tag.

JSX is converted into JavaScript behind the scenes, where React uses React.createElement() to turn the JSX code into actual HTML elements that the browser can understand

Understanding JSX Syntax

A syntax that is very similar to HTML can be used to write HTML components in JSX. For example, we can write <h1>Hello, World!</h1>instead of using React.createElement('h1', {}, 'Hello, World!').

2.4.2 What is Syntactic Sugar?

In programming, **syntactic sugar** refers to syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use, even though it doesn't add any new functionality or change what the code *does* under the hood. It's simply a more convenient or aesthetically pleasing way to write something that could be expressed in a more verbose or complex way.

2.4.3 JSX as Syntactic Sugar for React.createElement

This is precisely what JSX does for React:

- **Direct Translation:** Every single piece of JSX code you write is not directly understood by the browser. Instead, during the build process (typically by a transpiler like Babel), JSX is **transformed or compiled** into plain JavaScript. Specifically, it's converted into calls to React.createElement().

Example JSX:

```
<MyComponent prop1="value">
  Hello, world!
</MyComponent>
```

Transpiled JavaScript:

```
React.createElement(
  MyComponent,
  { prop1: "value" },
  "Hello, world!"
);
```

- **No New Functionality:** JSX doesn't introduce any new capabilities to React that you couldn't achieve by manually writing React.createElement calls. You could technically build an entire React application without writing a single line of JSX, just using React.createElement.

2.4.4 Why is it Beneficial as Syntactic Sugar?

Despite not adding new features, JSX provides immense benefits:

- **Readability and Maintainability:** Manually writing nested React.createElement calls, especially for complex UIs, quickly becomes cumbersome, hard to read, and prone to errors (e.g., mismatched parentheses or misplaced arguments). JSX's HTML-like structure is much more intuitive and readable, making it easier to understand the UI's hierarchy at a glance.
- **Familiarity for Web Developers:** For developers coming from an HTML background, JSX is instantly more familiar and less intimidating than a series of JavaScript function calls to define markup.
- **Declarative Nature:** It reinforces React's declarative programming paradigm by allowing developers to describe the desired UI state directly, rather than imperatively constructing it.
- **Developer Experience (DX):** It significantly improves the overall development experience by making component creation faster, less error-prone, and more enjoyable.
- **Tooling Integration:** While not inherent to "syntactic sugar," the widespread adoption of JSX has led to robust tooling support (linters, formatters, IDE integrations) that further enhance its benefits.

2.4.5 The Benefits of JSX in React

JSX is a fundamental part of React because it brings several significant advantages that streamline UI development:

- **Co-location of Markup and Logic:** JSX allows developers to seamlessly combine HTML-like markup directly within JavaScript logic inside the same component file. This co-location simplifies the

codebase by keeping related UI structure and behavior together, making components easier to understand, manage, and maintain.

- **Robust TypeScript Integration:** JSX integrates flawlessly with TypeScript, providing strong static type-checking for your UI components. This helps catch errors during development (rather than at runtime), leading to more reliable applications and a significantly improved developer experience through better autocompletion and error detection.

- **Built-in Security:** JSX automatically **escapes** any dynamic content embedded within it before rendering. This crucial feature inherently protects your applications from common injection vulnerabilities, such as Cross-Site Scripting (XSS) attacks, by converting potentially malicious characters into harmless string representations.

2.4.6 How Does JSX Work?

- **You Write JSX:** You create your user interface (UI) using JSX, which looks a lot like regular HTML mixed with some JavaScript. It's like a special shorthand for describing what your UI should look like.
- **Babel Translates:** Before your web browser can understand JSX, a tool called Babel steps in. Babel takes your JSX code and changes it into regular JavaScript that browsers can understand.
- **JSX Builds Components:** You use JSX to build individual, reusable pieces of your UI, called components. Think of them like building blocks.
- **Add Dynamic Content:** If you want to include dynamic information (like a user's name or a calculated value) within your JSX, you just put your JavaScript code inside curly braces {}.
- **JSX Becomes React.createElement():** Babel doesn't just turn JSX into *any* JavaScript. It specifically converts each JSX element into a call to React.createElement(). This function is what actually tells React to create a UI element.
- **React Renders It:** Finally, React takes these React.createElement() calls and uses them to build and update what you see on your screen (the DOM). When your data changes, React efficiently updates only the necessary parts of the UI.

2.4.7 Advantages of JSX

- **Easier to Read and Manage:** JSX lets you write HTML-like code directly inside your JavaScript. This makes your code much easier to read and understand, especially when you're looking at how your user interface is put together.

- **Reusable Building Blocks:** With JSX, you can create pieces of your UI that you can use over and over again. This makes your code more organized and easier to manage, just like using LEGO bricks to build different things.

- **Faster Performance:** JSX helps React make your website or app load and update quicker than some older ways of building web interfaces.

- **Dynamic Content Made Easy:** You can easily drop JavaScript code right into your HTML-like JSX to display changing information, like a user's name or a calculated total.

2.4.8 Disadvantages of JSX

1. **Extra Step Needed:** Before your web browser can understand JSX, it needs to be converted into regular JavaScript. This means you need an extra tool (like Babel) to do this conversion.

2. **Can Be Confusing for Newcomers:** If you're not used to working with HTML and CSS, learning JSX might feel a bit overwhelming at first because it blends JavaScript with HTML.
3. **Debugging Can Be Tricky:** Sometimes, because JavaScript and HTML-like code are mixed together in JSX, it can be a little harder to find and fix errors in your code.
4. **Not for Every Project:** For some projects where you absolutely need to keep HTML, CSS, and JavaScript completely separate, JSX might not be the best fit.

2.4.9 JSX Expressions

In JSX, an "expression" refers to a piece of JavaScript code that produces a value. You can embed these JavaScript expressions directly within your HTML-like JSX markup by wrapping them in curly braces {}. This is incredibly powerful because it allows you to make your UI dynamic and responsive to data.

As content within JSX tags: You can display dynamic text, numbers, or the result of a function call directly inside a JSX element.

```
const name = "Alice";
const age = 30;
const calculateSum = (a, b) => a + b;

function MyComponent() {
 return (
  <div>
   <h1>Hello, {name}!</h1> {/* Displays "Hello, Alice!" */}
   <p>You are {age} years old.</p> {/* Displays "You are 30 years old." */}
   <p>The sum is: {calculateSum(5, 7)}</p> {/* Displays "The sum is: 12" */}
  </div>
 );
}
```

As values for JSX attributes: Instead of a static string, you can provide a JavaScript expression as the value for an attribute.

```
const imageUrl = "https://example.com/profile.jpg";
const isActive = true;

function UserAvatar() {
 return (
  <img
   src={imageUrl} // Dynamic image source
   alt="User Profile"
   className={isActive ? "active-user" : "inactive-user"} // Dynamic class name based on a condition
  />
 );
}
```

The Core Rule: { JavaScript Expression }

Anything you can write as a JavaScript expression can go inside these curly braces.

Let's look at various types of expressions you can use:

- **Variables (Strings, Numbers, Booleans):**

```
import React from 'react';
const userName = "Jane Doe";
const productCount = 5;
const isAvailable = true;

function MyComponent() {
  return (
    <div>
      {/* String variable */}
      <p>Welcome, {userName}!</p>

      {/* Number variable */}
      <p>You have {productCount} items in your cart.</p>

      {/* Boolean variable (true/false don't render visually, but are useful for conditions) */}
      {isAvailable && <p>This product is in stock.</p>}
      {!isAvailable && <p>Currently out of stock.</p>}
    </div>
  );
}
```

- **Arithmetic Operations / Simple Calculations:**

```
function PriceDisplay({ unitPrice, quantity }) {
  return (
    <div>
      <p>Unit Price: ${unitPrice.toFixed(2)}</p>
      <p>Quantity: {quantity}</p>
      {/* Direct arithmetic calculation */}
      <p>Total: ${(unitPrice * quantity).toFixed(2)}</p>
    </div>
  );
}
```

- **Function Calls:**

The function must return something that React can render (like a string, number, null, or another JSX element).

```
function formatFullName(firstName, lastName) {
  return `${firstName} ${lastName}`;
}

function UserGreeting({ user }) {
```

```
  return (
    <div>
      {/* Calling a function to get a string */}
      <h2>Hello, {formatFullName(user.firstName, user.lastName)}!</h2>
    </div>
  );
}
```

- **Ternary Operator (Conditional Rendering):**

This is the most common way to do if/else logic *inside* JSX.

```
function AuthButton({ isLoggedIn }) {
  return (
    <div>
      {/* If isLoggedIn is true, render "Logout", otherwise render "Login" */}
      {isLoggedIn ? (
        <button>Logout</button>
      ) : (
        <button>Login</button>
      )}
    </div>
  );
}
```

- **Logical && (Short-Circuiting for Conditional Rendering):**

If the condition on the left of && is true, the expression on the right (often a JSX element) is rendered. If the condition is false (or null, undefined, 0, ""), nothing is rendered.

```
function Notification({ message, show }) {
  return (
    <div>
      {/* Only render the div if 'show' is true and 'message' exists */}
      {show && message && (
        <div style={{ border: '1px solid blue', padding: '10px' }}>
          {message}
        </div>
      )}
    </div>
  );
}
```

- **Array map() for Lists:**

This is how you render a collection of items. Remember the key prop!

```
function ItemList({ items }) {
  return (
    <ul>
      {items.map(item => (
```

```
      // The `key` prop is crucial for list rendering performance and stability
      <li key={item.id}>{item.name}</li>
    ))}
  </ul>
 );
}
```

- **Object Property Access:**

```
const user = {
 id: 1,
 firstName: "Ali",
 lastName: "Khan",
 email: "ali.khan@example.com"
};

function UserProfile() {
 return (
  <div>
   <h3>User Details:</h3>
   <p>ID: {user.id}</p>
   <p>Email: {user.email}</p>
  </div>
 );
}
```

- **Inline Styles (Object Literals):**

Remember, it's a JavaScript object, and CSS properties are camelCased.

```
function ColoredText({ text, color }) {
 return (
  // Outer {} for the JSX expression, inner {} for the JavaScript object literal
  <p style={{ color: color, fontSize: '18px', fontWeight: 'bold' }}>
   {text}
  </p>
 );
}
```

What You CANNOT Put Inside {} (and how to work around it)

You cannot put JavaScript **statements** directly inside JSX curly braces.

if/else statements:

```
// BAD
// { if (isAdmin) { <p>Admin</p> } else { <p>User</p> } }

// GOOD: Use Ternary Operator
{isAdmin ? <p>Admin</p> : <p>User</p>}
```

```
// GOOD: Move logic outside JSX
function MyComponent({ isAdmin }) {
  let content;
  if (isAdmin) {
    content = <p>Admin</p>;
  } else {
    content = <p>User</p>;
  }
  return <div>{content}</div>;
}
```

for loops:

```
// BAD
// { for (let i = 0; i < items.length; i++) { <p>{items[i].name}</p> } }

// GOOD: Use Array.prototype.map()
{items.map(item => <p key={item.id}>{item.name}</p>)}
```

Variable Declarations (const, let, var):

```
// BAD
// { const greeting = "Hello"; <p>{greeting}</p> }

// GOOD: Declare variables before the return statement
function MyComponent() {
  const greeting = "Hello";
  return <p>{greeting}</p>;
}
```

2.4.10 Comments in JSX

In React, JSX allows you to embed JavaScript expressions directly into your markup using curly braces {}. This same mechanism is used for comments within JSX. Unlike standard JavaScript comments (// or /* ... */), which are ignored by the parser, JSX comments are treated as JavaScript expressions that evaluate to nothing.

Guidelines for Using Comments in JSX:

- **Enclose Comments in Curly Braces {}:** Every comment within JSX must be wrapped in curly braces, just like any other JavaScript expression. Inside these braces, you use standard JavaScript multiline comment syntax /* ... */.

Example:

```
function MyComponent() {
  return (
    <div>
      {/* This is a single-line comment in JSX */}
      <p>Hello, React!</p>
```

```
    {/*
      This is a
      multi-line comment
      in JSX.
    */}
    <button>Click Me</button>
  </div>
 );
}
```

- **No Nesting of Comments:** You cannot nest JSX comments directly within each other. This is because the JSX parser treats the entire content of {} as a single JavaScript expression. If you try to nest /* */ comments, it will result in a syntax error.

Invalid Example (will cause an error):

```
JavaScript
// This will NOT work:
// <div>
//   {/* Outer comment { /* Inner comment */ } */}
// </div>
```

Workaround: If you need to "comment out" a section that *already* contains comments, you'll often temporarily enclose the entire JSX block in a larger JavaScript multiline comment *outside* the JSX, or use conditional rendering that evaluates to null.

- **Comments Can Be Used Anywhere a JSX Expression is Valid:** This means comments can be placed:

As Children (between tags): This is the most common use case.

```
<div>
 {/* This comment is a direct child of the div */}
 <h1>My Application</h1>
 <p>Some content.</p>
</div>
```

Within Props (attributes): This is useful for documenting why a prop is set in a certain way.

```
<img
 src={user.avatarUrl}
 alt="User Avatar"
 // Comment within an attribute definition
 className={isActive ? 'active' : 'inactive'}
/>
```

Within an Existing JavaScript Expression: If you have an expression already wrapped in {} (e.g., for inline styles or complex calculations), you can include standard JavaScript comments *inside* that expression.

```
<p
 style={{
   color: 'blue', // Standard JS comment within the style object expression
   fontSize: '16px' /* Another standard JS comment */
```

```
  }}
>
  Styled Text
</p>
```

Before/After JSX Elements (outside {}): Standard JavaScript comments (// or /* ... */) can be used anywhere *outside* JSX tags, in regular JavaScript code.

```
// This is a standard JavaScript comment
const MyComponent = () => {
  /*
   * This is also a standard JavaScript comment.
   * It applies to the entire component definition.
   */
  return (
    <div>
      {/* JSX comment inside JSX */}
      <p>Content</p>
    </div>
  );
};
```