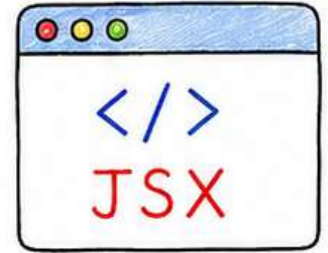


Chapter 3: JSX

3.1 Introduction to JSX

- * JSX stands for **JavaScript XML**.
- * It is a syntax extension for JavaScript used with **React**.
- * JSX allows us to write **HTML-like code** inside JavaScript.
- * It makes our code more **readable** and **easy to write**.
- * JSX is not a requirement to use **React**, but it is widely used.



Key Points:

- ✓ JSX makes React code more readable and easy to maintain.
- ✓ It allows us to write HTML elements in JavaScript.
- ✓ Browsers cannot understand JSX directly. It needs to be converted to regular JavaScript using Babel.



3.2 JSX Syntax

- * JSX looks similar to **HTML**, but it is written inside **JavaScript**.
- * JSX elements must have a **single parent element**.
- * JSX is case-sensitive.
- * Every JSX element must be closed.

Example 1: Simple JSX Element

```
const element = <h1>Hello, World!</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

This will display
Hello, World!
on the page.

Example 2: JSX with Multiple Elements

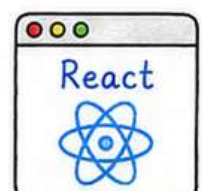
```
const element = (
  <div>
    <h1>Welcome to React</h1>
    <p>JSX makes UI development easy.</p>
  </div>
);
ReactDOM.render(element, document.getElementById('root'));
```

JSX elements
must return
a single parent
element.

Key Takeaway:



JSX allows us to write HTML-like code in JavaScript. It makes React development **easier** and more **efficient**.

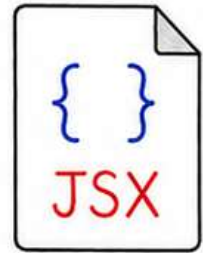


3.3 JSX Expressions

- * JSX expressions allow us to embed JavaScript expressions inside JSX using curly braces `{ }`.
- * The expression inside `{ }` will be **evaluated** and the result will be rendered.
- * We can use **variables**, **numbers**, **strings**, **functions**, or any valid JavaScript expression.

Example:

```
const name = "React Developer";
const age = 22;
const element = (
  <div>
    <h2>Hello, {name}!</h2>
    <p>I am {age} years old.</p>
  </div>
);
ReactDOM.render(element, document.getElementById('root'));
```



This will display the values of **name** and **age** on the page.

Key Points:

- ☒ Use `{ }` to embed JavaScript expressions in JSX.
- ☒ Expressions are evaluated and the result is rendered.
- ☒ We can use variables, numbers, strings, functions, and more.

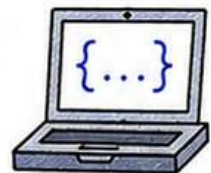


3.4 JavaScript Inside JSX

- * We can write any valid JavaScript code inside JSX using `{ }`.
- * This includes **variables**, **functions**, **conditional statements**, **loops** and more.
- * It helps make our components **dynamic and interactive**.

Example:

```
const isLoggedIn = true;
const message = isLoggedIn ? "Welcome Back!" : "Please Login";
const element = (
  <div>
    <h2>{message}</h2>
    <p>Today is {new Date().toLocaleDateString()}</p>
  </div>
);
ReactDOM.render(element, document.getElementById('root'));
```

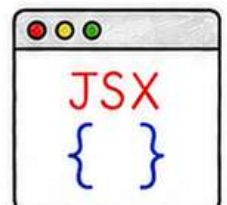


This will display a **dynamic** message and today's date.

Key Takeaway:



JSX allows us to use JavaScript code inside `{ }`. This makes our React components **dynamic** and **powerful**.



3.5 JSX Attributes

- * Attributes provide additional information about JSX elements.
- * They are written inside the opening tag.
- * Attributes work **similar to HTML attributes**, but are written in **camelCase**.
- * We can pass strings, numbers, expressions or functions as values.

```
<div  
  id="main"  
  title="Hello"  
  hidden />
```

Example:

```
const element = (  
    
);  
ReactDOM.render(element, document.getElementById('root'));
```

This will display an image with given **attributes**.

Key Points:

- ✓ Attributes provide extra information about elements.
- ✓ Use **camelCase** for attribute names.
- ✓ We can pass strings, numbers, expressions and functions.



3.6 className

- * In JSX, we use **className** instead of **class**.
- * **class** is a **reserved keyword** in JavaScript.
- * **className** is used to apply CSS classes to elements.
- * It works exactly like the **class** attribute in HTML.

```
.box  
{  
  color: blue;  
  padding: 10px;  
}
```

Example:

```
const element = (  
  <div className="box">  
    <h2 className="title">Welcome to React</h2>  
    <p className="description">JSX makes UI easy!</p>  
  </div>  
);  
ReactDOM.render(element, document.getElementById('root'));
```

This will apply CSS classes using **className**.

Key Takeaway:



- ✓ Use **className** instead of **class** in JSX.
- ✓ It helps us apply CSS styles to our elements.



3.7 Inline Styling

- * We can apply **CSS styles** directly inside JSX.
- * Inline styles are written using the **style** attribute.
- * The value of **style** is a JavaScript **object**.
- * Property names are written in **camelCase**.



```
style = {  
  color: "blue",  
  fontSize: "20px"  
}
```

Example:

```
const element = (  
  <h2 style={{ color: "blue", fontSize: "24px",  
    textAlign: "center" }}>  
    Hello, Inline Styling!  
  </h2>  
>);  
ReactDOM.render(element, document.getElementById('root'));
```

→ This will display
a **blue** heading
of size 24px
and **centered**.

Key Points:

- ✓ Use the **style** attribute for inline styling.
- ✓ Styles are written as a JavaScript object.
- ✓ Property names are written in **camelCase**.



3.8 Conditional JSX

- * We can render **different JSX** based on **conditions**.
- * This is useful to **show** or **hide** elements.
- * We can use **if** statements, **ternary operators** (**? :**), **&&** operator and more.



Example 1: Using if Statement

```
const isLoggedIn = true;  
let message;  
if (isLoggedIn) {  
  message = <p>Welcome Back!</p>;  
} else {  
  message = <p>Please Login.</p>;  
}  
ReactDOM.render(message, document.getElementById('root'));
```

Example 2: Using Ternary Operator

```
const isAdmin = false;  
const message = isAdmin ?  
  <p>Admin Panel</p> :  
  <p>Access Denied</p>;  
ReactDOM.render(message,  
  document.getElementById('root'));
```

Example 3: Using && Operator

```
const isVisible = true;  
const element = (  
  <div>  
    <h3>Dashboard</h3>  
    {isVisible && <p>This content is visible.</p>}  
  </div>  
>);  
ReactDOM.render(element, document.getElementById('root'));
```

→ Content inside
&& is rendered
only if the
condition is true.

Key Takeaway:



- ✓ Conditional rendering helps to show different UI based on conditions.
- ✓ We can use **if...else**, **ternary operator**, **&&** and more.
- ✓ It makes our React applications **dynamic** and **user-friendly**.



3.9 Rendering Arrays

- * We can render lists of elements using **arrays**.
- * We use the **map()** function to loop through the array.
- * Each item in the list must have a unique **key** prop.
- * Keys help React identify which items changed, added or removed.

```
[  
  "React",  
  "JavaScript",  
  "HTML"  
]
```

Example:

```
const items = ["React", "JavaScript", "HTML"];  
const element = (  
  <ul>  
    {items.map((item, index) => (  
      <li key={index}>{item}</li>  
    ))}  
  </ul>  
);  
ReactDOM.render(element, document.getElementById('root'));
```



→ This will display an unordered list of items from the **array**.

Key Points:

- ✓ Use **map()** to loop through arrays in JSX.
- ✓ Always provide a unique **key** for each item.
- ✓ Keys should be stable and unique among siblings.



3.10 Fragments

- * A component can return **multiple elements**.
- * Fragments allow us to group multiple elements **without** adding an **extra DOM node**.
- * We can use **<> </>** (short syntax) or **<React.Fragment> </React.Fragment>**.
- * Fragments are useful to keep the DOM clean.



Example 1: Using Short Syntax

```
const element = (  
  <>  
    <h2>Hello, React!</h2>  
    <p>Fragments make JSX cleaner.</p>  
  </>  
);  
ReactDOM.render(element, document.getElementById('root'));
```

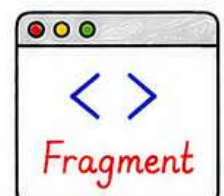
Example 2: Using <React.Fragment>

```
const element = (  
  <React.Fragment>  
    <h2>Hello, React!</h2>  
    <p>Fragments make JSX cleaner.</p>  
  </React.Fragment>  
);  
ReactDOM.render(element, document.getElementById('root'));
```

Key Takeaway:



- ✓ Use **fragments** to return multiple elements without an extra wrapper.
- ✓ Use **<> </>** for short syntax and **<React.Fragment> </React.Fragment>** for explicit syntax.
- ✓ Fragments help keep the DOM structure **clean** and **lightweight**.



3.11 JSX Rules

- * JSX has some **important rules** that we must follow.
- * These rules help us write **clean** and **error-free** code.
- * Following the rules ensures our code **works correctly**.

Rules:

- 1 JSX elements must have a **single parent** element.
- 2 Every JSX element must be **closed**.
Use **self-closing** tags for elements without content.
- 3 Use **camelCase** for all attributes and property names.
Example: `className`, `htmlFor`, `onClick`
- 4 Use **className** instead of **class**.
- 5 Use **curly braces { }** to embed JavaScript expressions.
Example: `<p>{name}</p>`
- 6 JSX is **case-sensitive**.
`<div>` is not equal to `<Div>`.
- 7 JSX must return a **single element** from a component.
Use **fragments** if returning multiple elements.

Key Takeaway:

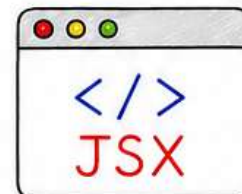
- ✓ JSX must have a **single parent** element.
- ✓ Always **close** all JSX elements.
- ✓ Use **camelCase** for attributes and property names.
- ✓ Use **className** instead of **class**.
- ✓ Use **{ }** to embed JavaScript expressions.
- ✓ JSX is **case-sensitive** and must return a **single element**.



Example:

```
const element = (  
  <div className="box">  
    <h2>Hello JSX Rules!</h2>  
    <p>Follow the rules to  
      write better React code.</p>  
  </div>  
)
```

Following these rules
helps us avoid errors and
write **correct JSX code**.



3.12 JSX Best Practices

- * Best practices help us write **clean**, **efficient** and **maintainable** code.
- * Following them improves code **quality** and **performance**.
- * These practices are useful for every **React developer**.

Best Practices:

- 1 Keep components **small** and **focused** on a single task.
- 2 Use **meaningful** and **consistent** names for components and variables.
- 3 Use **fragments** instead of extra `<div>` wrappers.
- 4 Keep JSX code **clean** and properly **formatted**.
- 5 Use **inline styling** only when necessary.
- 6 Avoid **complex logic** inside JSX; move it to **functions**.
- 7 Use **keys** when rendering lists to improve **performance**.

Key Takeaway:



- ✓ Write **small**, **focused** and **reusable** components.
- ✓ Use **meaningful** names and keep JSX **clean**.
- ✓ Use **fragments**, avoid unnecessary wrappers and **inline styles**.
- ✓ Use **keys**, avoid **complex logic** in JSX and follow best practices for **better code**.



Example:

```
const ItemList = ({ items }) => {  
  return (  
    <ul>  
      {items.map((item) => (  
        <li key={item.id}>{item.name}</li>  
      ))}  
    </ul>  
  )  
};
```

Using **keys** while rendering lists
helps React update the UI **efficiently**.

