EGOTECHWORLD.COM

BEGINNER TUTORIAL

<JSX/> From Zero to Confident

A guided walk through JSX — the special syntax React uses to describe what your UI should look like. Step by step, no surprises.

WHAT YOU'LL LEARN

1. What JSX is & why it exists
2. Your first JSX expression
3. Embedding values with curly braces
4. Attributes & the className quirk
5. Children, nesting & fragments
6. Conditional rendering
7. Rendering lists with .map()
8. Styling JSX (3 ways)
9. JSX gotchas & quick reference

Published by
egotechworld.com

Edition 2026
~25 minutes · beginner level

CHAPTER 1

What is JSX, and why does it exist?

JSX looks like HTML, but it isn't. Understanding what it really is makes everything else click.

When you start with React, the very first thing that looks weird is this:

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

HTML inside JavaScript? Without quotes? That's **JSX** — short for *JavaScript XML*. It's a special syntax extension that lets you write what looks like HTML directly in your JavaScript files.

The browser doesn't actually understand JSX

This is the most important thing to know up front: **JSX is not HTML, and the browser cannot read it**. Behind the scenes, a tool (Vite, Babel, or similar) converts your JSX into regular JavaScript function calls before the browser ever sees it.

What you write (JSX)

```
<h1 className="title">
  Hello, world!
</h1>
```

What the browser actually runs

```
React.createElement(
  'h1',
  { className: 'title' },
  'Hello, world!'
)
```

The JSX form on the left is just a friendlier way to write the JavaScript on the right. Both produce the exact same output.

Why bother with JSX?

You could write React without JSX — but nobody does. Here's why:

- **It's familiar.** If you know HTML, JSX feels like home.
- **It's visual.** You can see the structure of your UI at a glance.
- **You get the full power of JavaScript.** Variables, loops, conditions — all available right inside your markup.
- **Errors are clearer.** Mistyped a tag? Forgot a closing bracket? Your editor and the build tool will tell you instantly.

> **Mental model:** Think of JSX as "HTML with JavaScript superpowers". Whenever you're inside JSX, you can drop into JavaScript with curly braces `{ }`, and whenever you're inside JavaScript, you can drop into JSX with angle brackets `< >`.

CHAPTER 2

Your first JSX expression

Let's actually write some JSX, store it in a variable, and use it in a component.

JSX is a JavaScript expression

This means JSX behaves just like any other value in JavaScript. You can:

```
// Store it in a variable
const heading = <h1>Welcome!</h1>

// Return it from a function (this is what components do)
function Greeting() {
  return <p>Hello there!</p>
}

// Pass it as a value
const elements = [
  <h2>Title</h2>,
  <p>Paragraph</p>
]
```

The simplest possible component

A React component is just a JavaScript function that returns JSX. The function name **must start with a capital letter** — that's how React knows it's a component, not an HTML tag.

```
function Hello() {
  return <h1>Hello from egotechworld.com!</h1>
}
```

To actually see this on the screen, you use it inside another component, just like an HTML tag:

```
function App() {
  return <Hello />
}
```

RENDERS AS

Hello from egotechworld.com!

Returning multi-line JSX — wrap in parentheses

If your JSX is more than one line, wrap it in `( )` right after `return`. Without the parentheses, JavaScript's automatic semicolon insertion will break things.

❌ **Breaks silently**

```
function Card() {
  return
    <div>
      <h2>Title</h2>
    </div>
}
```

✅ **Works correctly**

```
function Card() {
  return (
    <div>
      <h2>Title</h2>
    </div>
  )
}
```

🛠 **TRY IT YOURSELF**

Open `src/App.jsx` in your project and replace the contents with a simple component that returns <h1>, your name, and a <p> with a short bio. Save the file and watch your browser update instantly.

CHAPTER 3

Embedding values with curly braces

Curly braces are how you escape from JSX into regular JavaScript. This is where the magic happens.

The rule: `{ ... }` means "JavaScript expression"

Anything you put between curly braces inside JSX is treated as JavaScript. The result is then displayed as text.

```
function Profile() {
  const name = 'Suranjith'
  const age = 35
  const isOnline = true

  return (
    <div>
      <h2>Hello, {name}!</h2>
      <p>You are {age} years old.</p>
      <p>Status: {isOnline ? 'Online' : 'Offline'}</p>
    </div>
  )
}
```

RENDERS AS

Hello, Suranjith!

You are 35 years old.

Status: Online

You can put any JS expression inside the braces

```
const firstName = 'Su'
const lastName = 'Ranjith'
const price = 1500

<p>Full name: {firstName + ' ' + lastName}</p>
<p>Total: Rs. {price * 1.15}</p>
<p>Today: {new Date().toLocaleDateString()}</p>
<p>Length: {firstName.length} chars</p>
```

What you CAN'T put in curly braces

Curly braces accept *expressions* (things that produce a value). They do **not** accept *statements* like `if`, `for`, or variable declarations.

❌ **Statement — won't work**

```
<p>
  {if (age > 18) {
    return 'Adult'
  }}
</p>
```

✅ **Expression — works**

```
<p>
  {age > 18 ? 'Adult' : 'Minor'}
</p>
```

If you really need an `if` statement, do it before the `return`:

```
function Status({ age }) {
  let label = 'Minor'
  if (age >= 18) label = 'Adult'
  return <p>{label}</p>
}
```

Watch out: The values `null`, `undefined`, `true`, and `false` render as **nothing**. The number `0`, however, renders as the character "0". This trips up many beginners — see Chapter 6.

CHAPTER 4

Attributes & the camelCase rules

JSX attributes look like HTML attributes — but with a few important differences that catch every beginner.

Static attributes use quotes; dynamic ones use braces

```
// Static value (string) → use quotes
<img src="/logo.png" alt="Logo" />

// Dynamic value from a variable → use curly braces
const imageUrl = '/users/suranjith.jpg'
<img src={imageUrl} alt={user.name} />

// You can mix both with a template literal
<img src={`/users/${user.id}.jpg`} />
```

Don't mix them: writing `src="{imageUrl}"` sets the literal string `"{imageUrl}"` — the curly braces become text. Either quotes or braces, never both.

HTML attribute names that are different in JSX

Because JSX is JavaScript, some HTML attribute names had to be renamed (mostly because they clash with JavaScript reserved words, or because JSX uses camelCase).

HTML	JSX	Why
`class`	`className`	`class` is a reserved JavaScript keyword
`for`	`htmlFor`	`for` is a reserved keyword (loops)
`onclick`	`onClick`	JSX uses camelCase for events
`onchange`	`onChange`	camelCase
`tabindex`	`tabIndex`	camelCase
`maxlength`	`maxLength`	camelCase
`readonly`	`readOnly`	camelCase

Good news: `data-*` and `aria-*` attributes keep their original names with hyphens. Only the standard HTML attributes were renamed.

Boolean attributes

For boolean attributes like `disabled`, `checked`, or `readOnly`, just including the attribute means "true". To set them dynamically, use a JS expression.

```
// Always disabled
<button disabled>Save</button>

// Conditionally disabled
<button disabled={isLoading}>Save</button>

// Pre-checked checkbox if user is admin
<input type="checkbox" checked={user.isAdmin} />
```

CHAPTER 5

Children, nesting & fragments

Every JSX expression must return one parent element. Here's how to handle that without wrapping everything in extra divs.

The "one parent element" rule

This is one of the strictest JSX rules. A component can only return **one** root element. Multiple siblings without a wrapper will throw an error.

❌ **Two roots — error!**

```
return (
  <h1>Title</h1>
  <p>Description</p>
)
```

✅ **Wrapped in a div**

```
return (
  <div>
    <h1>Title</h1>
    <p>Description</p>
  </div>
)
```

Fragments — wrappers without extra divs

Sometimes wrapping in a `<div>` isn't ideal — for example, in a table row where an extra div breaks the layout. A **Fragment** wraps your elements without adding any extra DOM node.

Long form

```
import { Fragment } from 'react'

return (
  <Fragment>
    <h1>Title</h1>
    <p>Body</p>
  </Fragment>
)
```

Short form (much more common)

```
// No import needed

return (
  <>
    <h1>Title</h1>
    <p>Body</p>
  </>
)
```

The empty tags `<>` `</>` are the fragment shorthand.

Self-closing tags

In JSX, every tag must be closed. Tags that don't have content (in HTML these are called void elements) must self-close with `/>`:

```
// ❌ Wrong in JSX (works in HTML, fails in JSX)
<img src="logo.png">
<br>
<input type="text">

// ✅ Correct in JSX
<img src="logo.png" />
<br />
<input type="text" />
```

Nesting components

Components can contain other components, just like HTML elements can contain HTML elements:

```
function Header() { return <h1>My Site</h1> }
function Footer() { return <p>© egotechworld.com</p> }

function App() {
  return <><Header/><main>Content here</main><Footer/></>
}
```

CHAPTER 6

Conditional rendering

Showing or hiding elements based on conditions. There are three patterns; you'll see all of them constantly.

Pattern 1: Ternary — when there are two options

Use a ternary `condition ? a : b` when you need to show one of two things.

```
function Greeting({ isLoggedIn, user }) {
  return (
    <div>
      {isLoggedIn
        ? <p>Welcome back, {user.name}!</p>
        : <p>Please log in.</p>
      }
    </div>
  )
}
```

Pattern 2: && — show only when true

When you want to show something **only if** a condition is true (and nothing otherwise), use &&.

```
function Notifications({ messages }) {
  return (
    <div>
      <h2>Inbox</h2>
      {messages.length > 0 && (
        <p>You have {messages.length} unread messages.</p>
      )}
    </div>
  )
}
```

The 0 trap: if you write `{messages.length && ...}` and `messages.length` is `0`, React will render the literal "0" on the screen. Always compare explicitly: `{messages.length > 0 && ...}` or convert with `{!!messages.length && ...}`.

Pattern 3: Early return — when half the page changes

If the component renders something completely different in one case, return early. This keeps the main path clean.

```
function UserProfile({ user, loading, error }) {
  if (loading) return <p>Loading…</p>
  if (error)   return <p>Something went wrong.</p>
  if (!user)   return null   // render nothing

  // The main happy path
  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.bio}</p>
    </div>
  )
}
```

Tip: `return null` from a component is perfectly valid — it tells React to render nothing.

CHAPTER 7

Rendering lists with `.map()`

This is the single most common pattern in real React apps. You'll write it dozens of times a day.

The pattern: array → array of JSX

To render a list, you take an array of data and use `.map()` to turn it into an array of JSX elements. React then renders each one in order.

```
function Skills() {
  const skills = ['PHP', 'Python', 'React', 'MySQL']

  return (
    <ul>
      {skills.map((skill, index) => (
        <li key={index}>{skill}</li>
      ))}
    </ul>
  )
}
```

RENDERS AS

- PHP
- Python
- React
- MySQL

Working with arrays of objects (the realistic case)

```
function CourseList() {
  const courses = [
    { id: 1, title: 'PHP Basics', price: 3500 },
    { id: 2, title: 'Python for Web', price: 3500 },
    { id: 3, title: 'React Beginner', price: 5000 }
  ]

  return (
    <div>
      {courses.map(course => (
        <div key={course.id} className="card">
          <h3>{course.title}</h3>
          <p>Rs. {course.price}</p>
        </div>
      ))}
    </div>
  )
}
```

The key prop — please don't skip this

React uses the `key` prop to identify each item across re-renders. Without unique keys, React can't tell which items moved, were added, or removed — leading to bugs and slow performance.

Rules for keys:

- **Unique** among siblings (not globally — siblings only).
- **Stable** — same item should keep the same key across renders.
- **Use a database id** when you have one — almost always the right answer.
- **Avoid the array index** if items can be reordered, added, or removed.

```
// ✅ Best — using a stable, unique id
{users.map(u => <li key={u.id}>{u.name}</li>)}
```

CHAPTER 8

Styling JSX (three ways)

There are three common ways to style elements in JSX. Each has its place — let's see them in order from simplest to most powerful.

Way 1: External CSS file with `className`

Just like regular HTML and CSS — except the attribute is called `className`, not `class`.

```
// styles.css
.card { border: 1px solid #ccc; padding: 16px; border-radius: 8px; }
.primary { background: #0ea5e9; color: white; }

// App.jsx — import the CSS once at the top of the app
import './styles.css'

function App() {
  return <div className="card primary">Hello</div>
}
```

This is what you'll use 90% of the time — including with frameworks like Bootstrap or Tailwind. It separates concerns and keeps your JSX clean.

Way 2: Dynamic className with template literals

Often the class needs to change based on state or props. Template literals make this easy.

```
function Button({ variant, isActive }) {
  return (
    <button className={`btn btn-${variant} ${isActive ? 'active' : ''}`}>
      Click me
    </button>
  )
}

// <Button variant="primary" isActive={true} />
// renders → <button class="btn btn-primary active">
```

Way 3: Inline `style` — the double curly braces

For one-off styles or values that come from JavaScript, use the `style` attribute. It takes a JavaScript **object**, not a string — which is why you see two sets of curly braces.

```
function Banner({ color, isHighlighted }) {
  return (
    <div style={{
      color: color,
      fontSize: '18px',
      backgroundColor: isHighlighted ? 'yellow' : 'white',
      padding: '10px'
    }}>
      Hello!
    </div>
  )
}
```

The double-brace explanation:

The **outer** `{ }` means "JS expression" (Ch3). The **inner** `{ }` is a JS object literal. So `style={{ color: 'red' }}` = "this attribute is an object with a color property".

CSS keys are camelCased: `backgroundColor` not `background-color`. Values with units must be strings: `'18px'`, not `18px`.

CHAPTER 9

JSX gotchas & quick reference

A field guide to the small things that trip up every beginner. Bookmark this page.

Comments inside JSX

Inside JSX, regular `// comments` don't work between tags. Use `{/* comment */}`.

```
<div>
  {/* This is a JSX comment */}
  <h1>Hello</h1>
</div>
```

Capitalization decides everything

JSX uses the first letter of a tag name to decide what it is:

- **Lowercase** like `<div>` or `<header>` → treated as a normal HTML tag
- **Capitalized** like `<Header>` or `<UserCard>` → treated as a React component

This means a component named `welcome` (lowercase) won't render — React thinks it's an unknown HTML element.

The full beginner cheat-row

Rule	Example
Component names start uppercase	`function MyButton() { ... }`
Return one parent or use `<></>`	`return ( <> ... </> )`
Close every tag	`<img />   <input />`
Use `className` not `class`	`<div className="card">`
Use `htmlFor` not `for`	`<label htmlFor="email">`
Events use camelCase	`onClick`, `onChange`, `onSubmit`
JS values inside `{ }`	`<p>Hi, {name}!</p>`
Inline style is a JS object	`style={{ color: 'red' }}`
CSS keys are camelCase	`backgroundColor`, `fontSize`
List items need a unique `key`	`map(u => <li key={u.id}>{u.name}</li>)`
JSX comments use `{/* */}`	`{/* this is a comment */}`
`null`, `false`, `undefined` → render nothing	Useful for hiding elements
`0` renders as "0"	Use `{n > 0 && ...}`, not `{n && ...}`

🎉 You've got JSX!

JSX is one of those things that feels foreign for the first hour and natural forever after. The patterns in this tutorial cover ~95% of the JSX you'll ever write.

What to practice next

- Build a profile card component using props, conditional rendering, and a `.map()` over skills
- Make a list of products with prices, filtered by category using a ternary
- Build a comment list with `{comments.length > 0 && ...}` and an empty state
- Style a button with both `className` and dynamic inline `style`
- Recreate this PDF's "renders as" boxes as a real React component

The egotechworld learning path

- **1.** JavaScript for React Cheatsheet — the JS features you need first
- **2.** JSX for Beginners (this tutorial) — describing your UI
- **3.** The React Cheatsheet — components, props, state, hooks
- **4.** Build a real project — to-do app, weather app, or portfolio