JSX

What JSX is, how it differs from HTML, embedding JavaScript with curly braces { }, attributes and className, JSX rules, and a quick look at conditional rendering.

Beginner 12 min

Last updated: October 5, 2026
Read online: https://www.learnforgex.com/en/topics/jsx

JSX

In Creating a React Application, we saw how to set up a React project. Now we get into actual React code. The syntax you use to write UI in React is called **JSX** -- this lesson covers what JSX is and its basic rules, with examples kept as simple as possible.

What Is JSX?

JSX is a syntax that looks a lot like HTML, but is actually an extension of JavaScript. Most React components use JSX to describe what the UI should look like:

```
// JSX looks like HTML, but it's actually an extension of JavaScript.
// The browser doesn't understand it directly -- a compiler (Babel/Vite)
// converts it into a regular JavaScript function call behind the scenes.
const element = <h1>Hello, React!</h1>;

// The line above means the same thing as the following (if it were
// written without JSX):
// const element = React.createElement("h1", null, "Hello, React!");

console.log(element);
```

This code looks like HTML, but it's really assigning a value to a JavaScript variable (`element`). The browser can't understand JSX directly -- the project's build tooling (Vite, in this course) automatically transforms this code into plain JavaScript function calls the browser understands. You never need to do this step by hand.

HTML vs. JSX

JSX looks like HTML, but it isn't the same thing. One of the most important differences: JSX is, in the end, a JavaScript value -- you can assign it to a variable, return it from a function, or put it in a list. Plain HTML can't do any of that.

There's also a naming difference (we'll see this in "Attributes and className" below). But overall, the look is largely the same as HTML -- tags like `<div>`, `<h1>`, and `<button>` will look familiar.

JavaScript Inside JSX: Curly Braces { }

The most powerful part of JSX is that you can embed real JavaScript inside it. You do this with a single curly brace `{ }`:

```
// Inside curly braces { } you can write ANY JavaScript expression that
// produces a result: a variable, an addition operation, a function call...
const name = "Emma";
const greeting = <h1>Hello, {name}!</h1>;

const a = 5;
const b = 3;
const total = <p>Total: {a + b}</p>;

function shout(text) {
  return text.toUpperCase();
}
const loud = <p>{shout("i am learning react")}</p>;

console.log(greeting, total, loud);
```

Anything that **produces a value** can go inside the curly braces: a variable (`{name}`), a calculation (`{a + b}`), or a function call (`{shout(...)}`). Things that don't produce a value (like an `if` block) can't go here -- we'll see why in "A Quick Look at Conditional Rendering" below.

Attributes and className

You use HTML attributes (`src` , `alt` , `class` , etc.) in JSX too, but with two important differences:

```
// Attributes in JSX are very similar to HTML, but there are a few differences:
// 1) "className" is used instead of "class" (class has a separate meaning in JavaScript).
// 2) Attribute values can also be a JavaScript variable using { }.
const userName = "ayse";

const avatar = (
  <img className="avatar" src={`/images/${userName}.png`} alt="User avatar" />
);

// If the tag itself doesn't close (e.g. <img>, <input>, <br>), it MUST be
// "self-closing" in JSX -- a / is added at the end.
const divider = <hr />;

console.log(avatar, divider);
```

First: instead of HTML's `class`, JSX uses `className` (because `class` already means something else in JavaScript). Second: just like with text content, you can bind an attribute's value to a JavaScript expression using curly braces `{ }`.

JSX Rules

JSX has a few simple rules you need to follow:

```
// JSX has a few simple but strict rules:

// 1) A JSX block must have a SINGLE root element.
// This does NOT WORK: <h1>Title</h1><p>Text</p>  (two sibling elements, no single root)
// Instead, wrap it with a <div> (or with <> </> as below):
const withDiv = (
  <div>
    <h1>Title</h1>
    <p>Text</p>
  </div>
);

// <> </> ("Fragment") does the same job without adding an extra <div> --
// a wrapper invisible to the DOM, existing only to satisfy the JSX rule.
const withFragment = (
  <>
    <h1>Title</h1>
    <p>Text</p>
  </>
);

// 2) Attribute names are written in camelCase: "onClick" not "onclick",
// "tabIndex" not "tabindex".

// 3) "className" is used because "class" is a keyword in JavaScript.

console.log(withDiv, withFragment);
```

The most important one: a JSX block must have a **single root element** -- you can't return two sibling elements side by side. You can wrap them in a `<div>`, or use `<> </>` (a

"Fragment") if you don't want to add an extra HTML element. The other rules: attribute names are written in camelCase (`onClick` , `tabIndex`), and self-closing tags (`<img>` , `<input>`) must always be closed with a `/` in JSX (`<img />`).

A Quick Look at Conditional Rendering

We saw that curly braces `{ }` can only hold something that "produces a value" -- that's why you can't write `if` directly inside JSX (`if` doesn't produce a value, it just decides whether to run a block of code). Instead, you use something that does produce a value, like a ternary (`? :`):

```
// You can't write "if" inside JSX (if is a statement, not an
// expression) -- but since you can put an "expression" inside { }, tools
// based on expressions like ternary (? :) or && come in handy here. We'll
// go into this topic in much more detail in the "Conditional Rendering"
// section (in the State & Events category) -- here we're just seeing why
// JSX does it this way.
const isLoggedIn = true;

const message = isLoggedIn ? <p>Welcome!</p> : <p>Please log in.</p>;

console.log(message);
```

This is just a quick preview -- we'll cover showing different UI based on a condition ("conditional rendering") in much more depth in the "Conditional Rendering" lesson, in the State & Events category.

Summary and Glossary

JSX is a syntax that looks like HTML but is actually JavaScript. You can embed JavaScript expressions inside it with curly braces `{ }` . It uses `className` instead of `class` , every JSX block must have a single root element, and self-closing tags must be closed with `/` .

Glossary

JSX — A syntax that looks like HTML but is actually an extension of JavaScript.

Expression — A piece of code that produces a value (e.g. `a + b`); only this kind of code can go inside `{ }` in JSX.

className — JSX's equivalent of HTML's `class` attribute.

Fragment (`<> </>`) — A JSX tool for grouping multiple elements under a single root without adding an extra HTML element.

Root Element — The single, outermost element in a JSX block -- every JSX block must have exactly one.

Practical Project

There's a real, runnable example project that brings together the concepts from this category (What Is React?, Creating a React Application, JSX): **React Fundamentals Demo**.

It shows JSX, JavaScript expressions with curly braces `{ }`, attributes, and a short conditional-rendering preview with a ternary, all working together. You can download it and run it yourself, and read through the code line by line:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/react-fundamentals
npm run dev
```

The `react-course-projects` repo uses **npm workspaces** -- `npm install` only needs to run once, at the repo root, and every project folder shares the same dependencies (no separate `node_modules` per folder). If you've already run `npm install` at the root, you can just `cd react-course-projects/projects/react-fundamentals` and run `npm run dev`.