# Building Dynamic UIs: The Core of React

A Deep Dive into JSX, Components, Props, and State

# Why Do We Need a New Syntax?

Before React, creating UI elements in JavaScript was verbose. JSX makes it declarative and intuitive.

## JavaScript (document.createElement)

```
const heading = document.createElement('h1');
heading.className = 'greeting';
heading.textContent = 'Hello, World!';

const container = document.getElementById('root');
container.appendChild(heading);
```

## React (JSX)

```
const element = <h1 className="greeting">Hello,
  World!</h1>;

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

**JSX lets you describe *what* the UI should look like, not the step-by-step process of creating it.**

# JSX: Writing UI Inside JavaScript

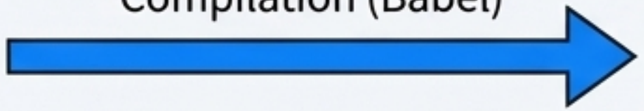
JSX stands for JavaScript XML. It's a syntax extension for JavaScript that allows you to write HTML-like code directly in your JS files.

## How it Works

**Your JSX Code**

```
const el = <h1>Hi</h1>;
```

Compilation (Babel)



**Plain JavaScript**

```
React.createElement('h1', null, 'Hi');
```

### Readability

UI structure and logic live in the same place.

### Declarative Syntax

Code clearly describes the final output.

### Tooling Support

Excellent support for autocompletion, linting, and type-checking.

# The Essential Rules of JSX

## Rule: One Root Element

Your JSX must return a single parent element. Wrap siblings in a `<div>` or a React Fragment (`<>...</>`).

```
// Correct
return (
  <>
    <h2>Title</h2>
    <p>Paragraph</p>
  </>
);
```

## Rule: Attribute Naming

JSX uses camelCase for most attributes and special names for HTML keywords.

- `class` becomes `className`
- `for` becomes `htmlFor`
- Event handlers like `onclick` become `onClick`

## Rule: Embedding JavaScript

Use curly braces `{}` to embed any valid JavaScript expression—a variable, a function call, or a calculation—directly into your markup.

```
<h1>User: {user.name}</h1>
```

## Rule: Self-Closing Tags

Any element that doesn't have children must be explicitly closed with `/>`.

```
<img src="..." /> or <br />
```

# Bringing Your UI to Life with JavaScript Expressions

## Conditional Rendering

Show or hide elements based on logic.

### Pattern 1: Ternary Operator

```
<div>
  {isLoggedIn ? <UserProfile /> : <LoginForm />}
</div>
```

### Pattern 2: Logical &&

```
<div>
  {unreadMessages.length > 0 &&
    <h2>You have {unreadMessages.length} unread
messages.</h2>
  }
</div>
```

## Rendering Lists

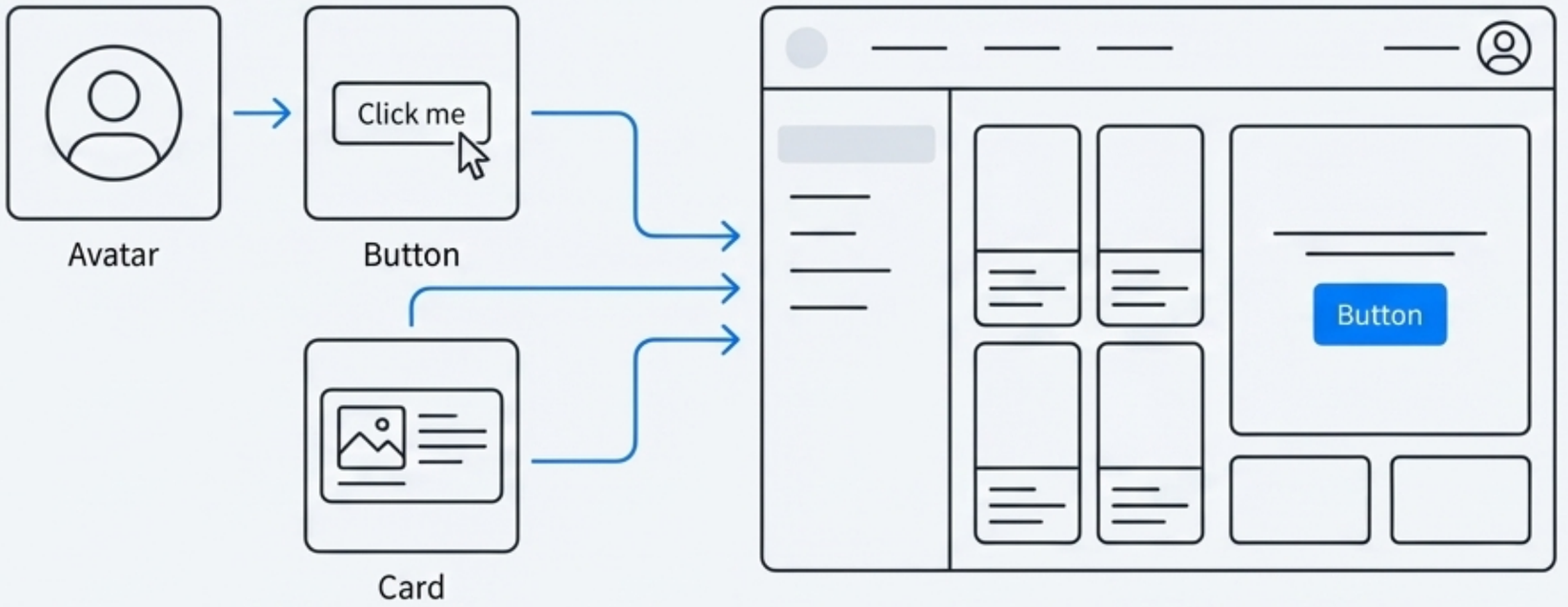
Transform arrays of data into UI elements using `.map()`.

```
<ul>
  {products.map(product => (
    <li key={product.id}>
      {product.name}
    </li>
  ))}
</ul>
```

**The `key` prop is crucial.** It must be a unique, stable identifier that helps React efficiently update the list.

# The Blueprint of React: Components

Think of Components as custom LEGO bricks for your UI. You define their look and logic once, then reuse and combine them to build anything.




## Reusable

Use the same component in multiple places.

## Composable

Build complex components by nesting simpler ones.

## Isolated

Each component can manage its own logic and data, making your app easier to maintain.

# Two Ways to Build: Functional vs. Class Components

## Functional Components (The Modern Standard)

Simple JavaScript functions that accept `props` and return JSX. They use **Hooks** (like `useState`) to handle state and logic.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

## Class Components (The Classic Blueprint)

ES6 classes that extend `React.Component`. They use a `render()` method to return JSX and manage state with `this.state`.

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

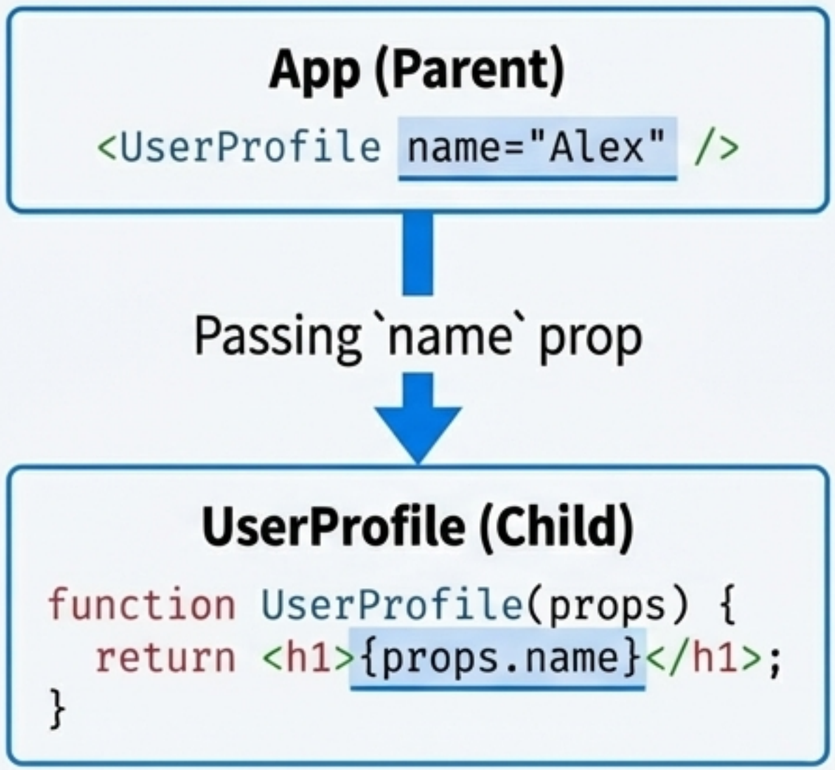
While you'll see Class Components in older codebases, **Functional Components with Hooks are the recommended standard for all new React development.**

# Head-to-Head: The Defining Differences

| Feature | Functional Component | Class Component |
| --- | --- | --- |
| **Syntax** | Plain function or arrow function | ES6 class extending `React.Component` |
| **Props Access** | `props.name` | `this.props.name` |
| **State** | `useState()` Hook | `this.state` and `this.setState()` |
| **Lifecycle** | `useEffect()` Hook | Lifecycle Methods (e.g., `componentDidMount`) |
| **Conciseness** | More concise and readable | More boilerplate and verbose |
| **Recommendation** | **Current standard.** Use for new code. ✅ | **Legacy.** Understand for maintenance. 🕓 |

# Props: Passing Data From the Outside In

Props (short for “properties”) are how you pass data from a **parent component** down to a **child** component. They make your components configurable and reusable.




**Props are Read-Only.** A child component can *never* change the props it receives. They are immutable.

# State: Giving Components an Internal Memory

State is data that a component **owns and controls internally**. When this data changes, React automatically re-renders the component to reflect the new state.

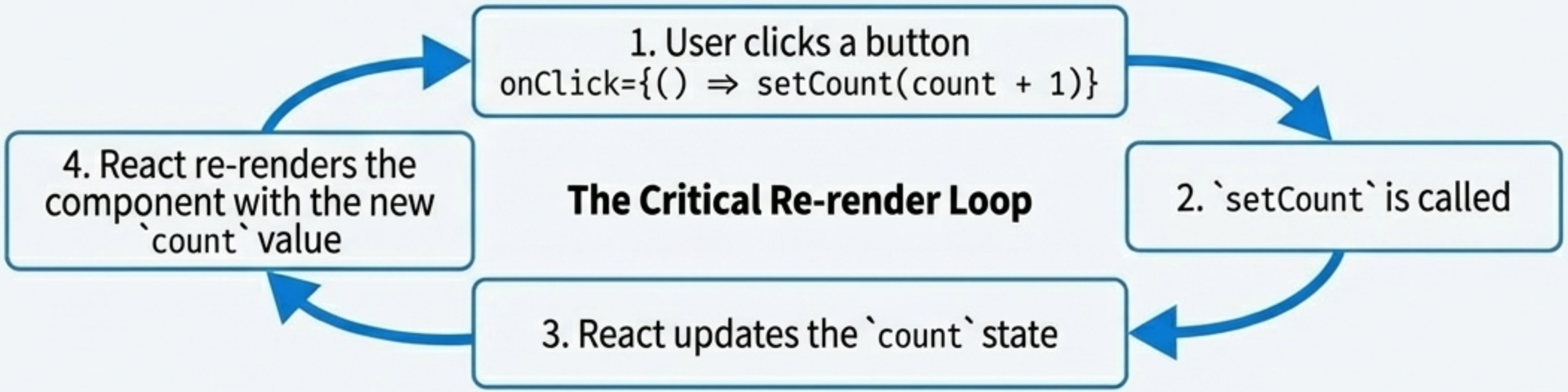
## Introducing the `useState` Hook

### How it Works

2. It returns a pair: the current state value...

```
// 1. Call useState with an initial value.
const [count, setCount] = useState(0);
```

...and a function to update it.




**Golden Rule:** Never modify state directly (e.g., `count = 1`). Always use the setter function (`setCount(1)`).

# The Full Picture: An Interactive Component

```jsx
// 1. A Functional Component that accepts `props`
function LikeButton({ initialLikes }) {

  // 2. Initializing internal `state` with useState
  const [likes, setLikes] = useState(initialLikes);

  const handleLike = () => {
    // 3. Using the setter to update state on click
    setLikes(likes + 1);
  };

  // 4. Returning JSX to define the UI
  return (
    <button onClick={handleLike}>
      👍 Like | {likes} // 5. Displaying the state variable
    </button>
  );
}
```

**1. Component & Props**
A reusable function that receives `initialLikes` from its parent.

**2. State**
The component's internal memory, starting with the value from props.

**3. Event Handling**
A function that changes the state when the user interacts.

**4. JSX**
Declarative markup describing what the button looks like.

**5. Dynamic Rendering**
The UI displays the current value from state, and automatically updates when it changes.

# React Core Concepts: Quick Reference

## JSX

- JavaScript syntax extension for UI.
- Rules: Single root, `className`, `{}` for expressions.
- Compiles to `React.createElement()`.

## Components

- Reusable, isolated UI building blocks.
- Functional components are the modern standard.
- Receive data via props, manage their own data with state.

## Props

- Data passed from parent to child.
- Read-only (immutable).
- Used to configure and customize components.
- Accessed via `props` object in functional components.

## State (`useState`)

- Internal memory of a component.
- Managed with the `useState` Hook.
- `const [value, setValue] = useState(initial);`
- Updating state with `setValue` triggers a re-render.

# Frequently Asked Questions

**Q: Why `className` instead of `class`?**

A: Because `class` is a reserved keyword in JavaScript for creating classes. JSX is JavaScript, so it avoids this conflict.

**Q: Why must JSX return a single root element?**

A: Under the hood, a component's JSX is converted into a single function call (`React.createElement()`) that must return a single object. Wrappers like `<>` or `<div>` ensure this.

**Q: What's the simplest difference between Props and State?**

A: **Props** are passed *into* a component (like function arguments). **State** is managed *within* the component (like variables declared inside a function).

**Q: When will I actually encounter a Class Component?**

A: Primarily when working on older React projects, or with certain advanced libraries that haven't fully adopted Hooks. It's important to recognize them, but not necessary to write them for new projects.

# Your First Challenge: Build an Interactive Counter

Objective: Apply your knowledge of components, JSX, and `useState` to build a simple counter.

**Requirements:**

- ✅ Create a new `Counter` functional component.
- ✅ Use the `useState` Hook to initialize a `score` variable to 0.
- ✅ Render the current `score` on the screen.
- ✅ Add a button that, when clicked, increases the score by 1.