Day 03: React Core Concepts (Part 1)

Building Blocks of Modern Web Development

Journey into the heart of React, exploring the fundamental concepts that power dynamic user interfaces.

Chapter 1

JSX and React Elements

JSX (JavaScript XML) allows you to write HTML-like code directly within your JavaScript files. It might look like HTML, but it's not a string; it's a syntax extension that gets **transpiled** into standard JavaScript by tools like Babel.

This transformation turns your readable JSX into `React.createElement()` calls, which React then uses to build your UI. The main benefit? It makes your code much easier to read and write, blending UI structure and logic seamlessly.

Key Points

- **Single Parent Element:** JSX must always return one root element.
- **Embedded JavaScript:** Use `{}` to embed JavaScript expressions.
- **DOM Attributes:** `class` becomes `className`, and inline styles use objects (e.g., `{{ color: 'red' }}`).

JSX Example:

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

Transpiled by Babel:

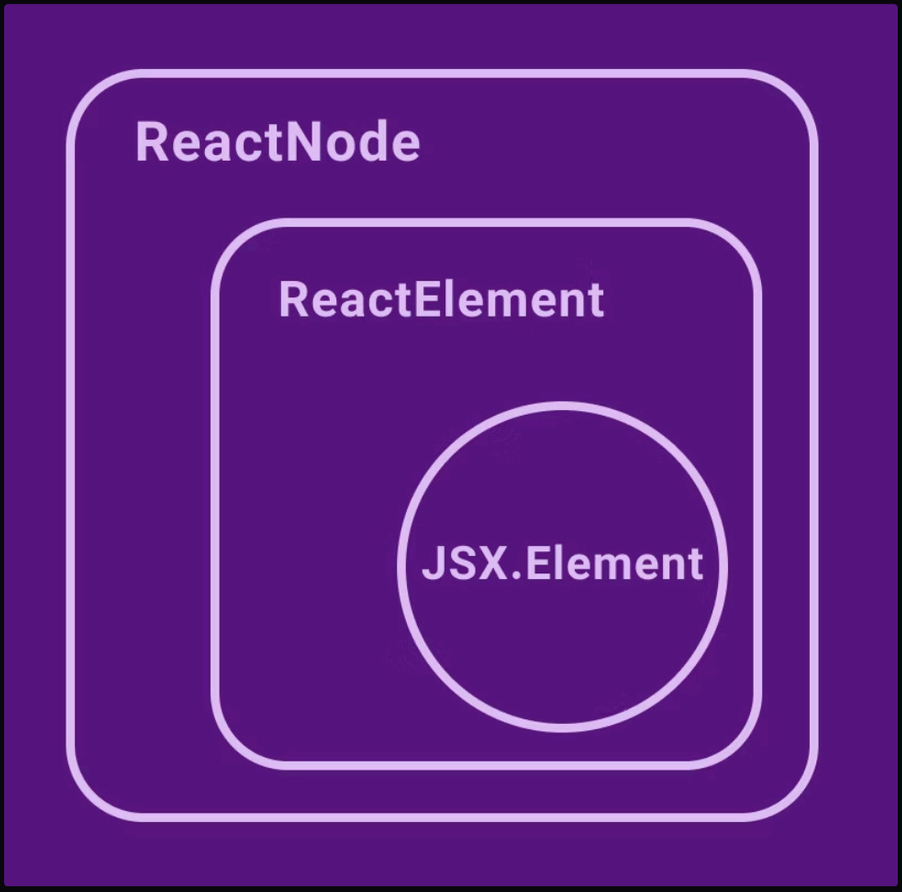
```
const element = React.createElement("h1", null, "Hello, world!");
```

Embedding JS:

```
const name = "React";
const element = <p>Hello, {name.toUpperCase()}!</p>;
```

Chapter 2

React Elements: The UI Blueprint



A React element isn't an actual piece of your webpage (a DOM node) yet. Instead, think of it as a **lightweight blueprint** or a description of what you want to appear on the screen.

When you log a JSX element to the console, you'll see it's just a plain JavaScript object. This object holds information about the element's type (like `"h1"`) and its properties (`props`), such as its children.

```
console.log(<h1>Hello</h1>);
```

👉 Logs a plain JS object like:

```
{
  type: "h1",
  props: { children: "Hello" }
}
```

React then takes these objects and **reconciles** them with the actual browser DOM, efficiently updating only what's changed on the screen.

Chapter 3

Rendering Your Application

The Root Element

Your React application needs a single "root" DOM element where all your React components will be rendered. This is usually a `<div>` with an `id="root"` in your `index.html`.

ReactDOM.createRoot

For React 18 and above, `ReactDOM.createRoot()` is the modern way to tell React where to start managing your UI. It takes your root DOM element as an argument.

Mounting the App

Finally, you call `.render()` on the created root, passing in your main application component (commonly `<App />`). This is when React draws your UI to the screen.

```
import ReactDOM from "react-dom/client";
import App from "./App";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App/>);
```

Chapter 4

Components: Building Blocks of UI

Components are independent, reusable pieces of UI. They let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Function Components (Preferred)

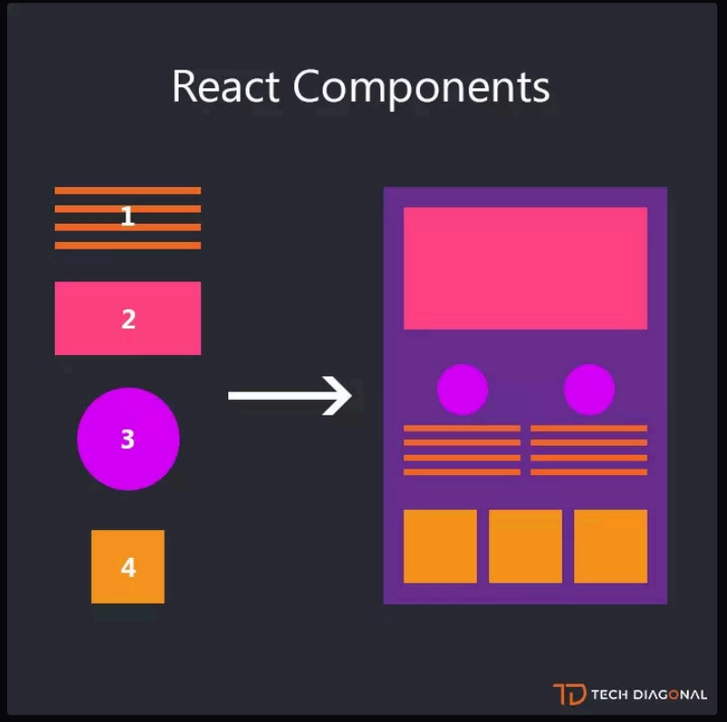
Function components are JavaScript functions that accept "props" (properties) as an argument and return React elements. They are the modern standard in React development, especially when paired with Hooks for state and lifecycle management.

```
function Welcome(props) {
  return <h2>Hello, {props.name}</h2>;
}
```

Class Components (Legacy)

Older React applications might use class components, which are ES6 classes that extend `React.Component` and require a{" "} `render()` method. While still supported, they are generally not used for new development.

```
class Welcome extends React.Component {
  render() {
    return (<h2>Hello, {this.props.name}</h2>);
  }
}
```



Chapter 5

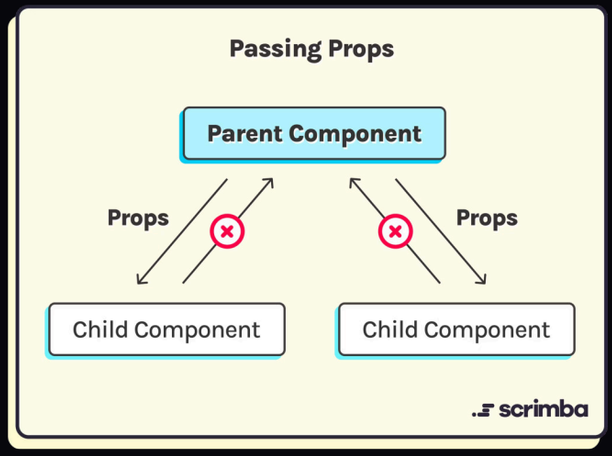
Props: Passing Data Down

Props (short for properties) are how you pass data from a parent component to a child component. Think of them like arguments you pass to a JavaScript function.

Inside the receiving component, props are **immutable** – you should never directly modify props within a component. This ensures a predictable data flow and makes your components easier to reason about.

```
function Button({ label }) {
  return <button>{label}</button>;
}

function App() {
  return <Button label="Click Me" />;
}
```



Prop Validation (Optional, but Recommended for Robust Apps)

For larger applications, you can use libraries like `prop-types{" "}` to define the expected types and requirements for your component's props, catching potential errors early.

```
npm install prop-types
```

```
import PropTypes from "prop-types";

Button.propTypes = {
  label: PropTypes.string.isRequired
};
```

Hands-on: Simple Counter App

Let's combine JSX, rendering, props, and function components to build a dynamic counter!

Step 1: CounterButton Component

This component will display the current count and handle clicks.

```
function CounterButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>;
}
```

Step 2: App Component

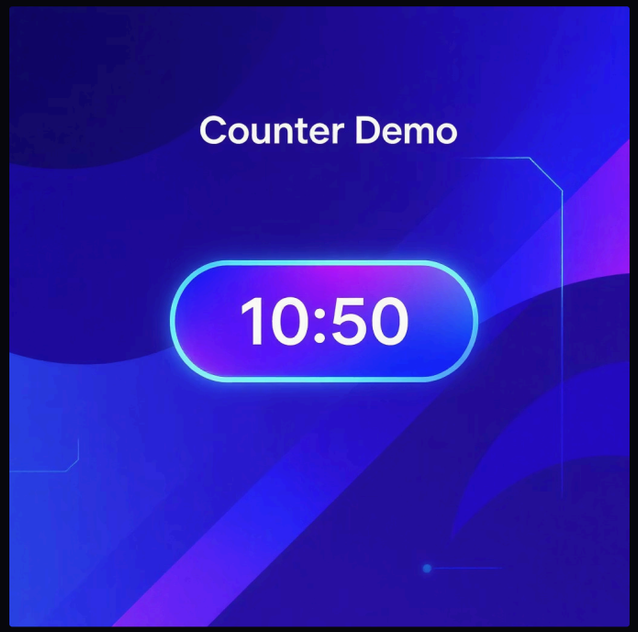
The main application component will manage the counter's state and pass it down.

```
import { useState } from "react";
import CounterButton from "./CounterButton";

function App() {
const [count, setCount] = useState(0);

return (
  <div>
    <h1>Counter Demo</h1>
      <CounterButton
        count={count}
        onClick={() => setCount(count + 1)}
      />
  </div>
);
}

export default App;
```



Step 3: Run & Observe

- Every click on the button triggers the `onClick` function.
- This updates the component's internal `count` state.
- React automatically detects the state change and efficiently re-renders the `CounterButton` with the new count.