Welcome to Lecture 24

Agenda

Session Objectives

- Introduction to Components
- Understanding JSX
- Interaction between components
    - Defining relationships
    - Exchanging data - props
- Prop Drilling
- Quiz

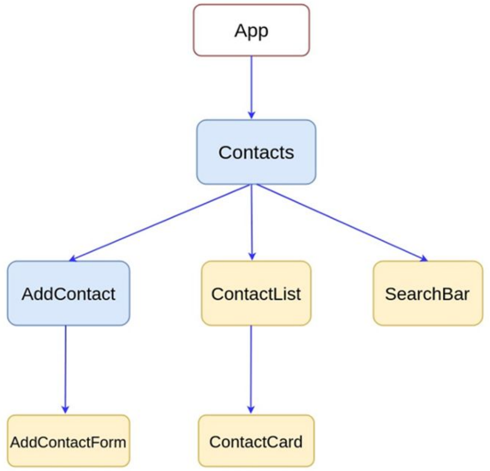
The “Why” and “What” of Components

Why Do We Need Components? A Glance Back

- In early web development, we built entire pages. A change in one place (like a header) often meant updating many different HTML files. This was slow and error-prone
- With the rise of server-side frameworks, interfaces could be generated dynamically. The browser still received a full HTML page, though
  - Interactivity in such scenarios was added on top with libraries like jQuery, which manipulated the DOM directly
- **The Problem**
  - Manual DOM queries and imperative updates for every data change create repetitive, fragile glue code
  - In a large, dynamic UI (e.g. an Instagram-style feed), those scattered, ad-hoc update calls are hard to maintain

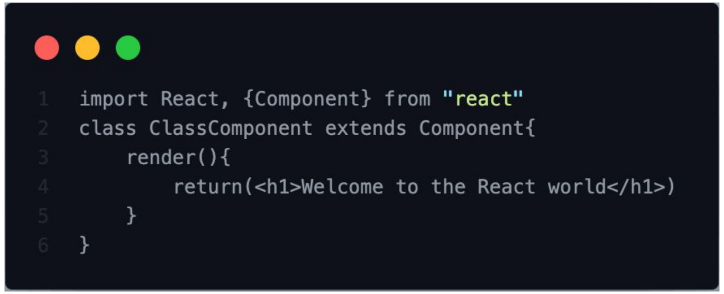
The Solution: Building with Reusable Blocks

- **The Core Idea:** Break the UI down into small, independent, and reusable pieces called "components"
- Each component manages its own HTML, CSS, and JavaScript logic. It's a self-contained unit
- **The Improvement**: Instead of thinking about "pages", we think about a "tree of components"
  - Eg: You build a <Header>, a <ProductCard>, a <LikeButton>, and compose them together to create an application
  - If you update the <ProductCard> component, and every place it's used updates automatically

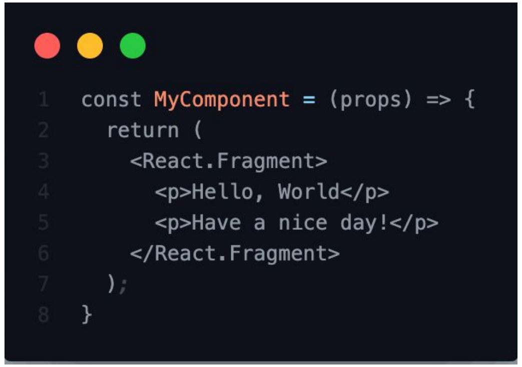


Two Flavors of Components: Functional & Class

- Class Components
  - Built using JavaScript ES6 classes
    - ES6 is the JavaScript version from 2015 that introduced the class syntax, making this component style possible
  - They are "stateful" by nature, extending the React.Component class, which gives them access to state and special lifecycle methods
  - Relied heavily on the this keyword to access props, state, and methods
- Functional Components
  - Initially just simple JavaScript functions used for UI that didn't need state ("stateless" or "presentational" components)
  - With React 16.8, the introduction of Hooks (useState, useEffect) allowed functional components to do everything class components could, but with less code and a simpler syntax



```
import React, {Component} from "react"
class ClassComponent extends Component{
    render(){
        return(<h1>Welcome to the React world</h1>)
    }
}
```



```
const MyComponent = (props) => {
  return (
    <React.Fragment>
      <p>Hello, World</p>
      <p>Have a nice day!</p>
    </React.Fragment>
  );
}
```

JavaScript Prerequisite: What is this?

- Before we can compare component types, we must understand a fundamental keyword in Javascript: the this keyword
- this is not a variable. It's a keyword whose value is determined by the execution context, i.e, how a function is called
    - Think of it as a reference to the object that is making the call
- In a Method: When a function is called as a method of an object, this refers to the object itself
- In a Standalone Function: When a function is called by itself (not as part of an object), **this** is undefined. In older, non-strict mode, it would refer to the global window object. This difference is a major source of bugs
- With that knowledge, let's return to class vs functional components...

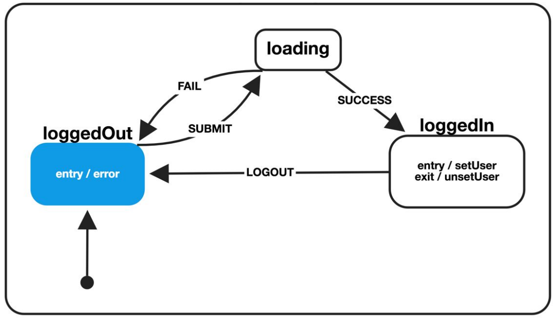
Deep Dive: The Core Differences

Feature	Class Component	Functional Component
Syntax	Requires ES6 class, extends React.Component, and needs a render() method	A plain JavaScript function that returns JSX
Props	Accessed via this.props	Passed directly as an argument to the function, e.g., function MyComponent(props)
State	Needs a constructor to initialize this.state. Updated with this.setState(), which merges the new state with the old	Uses the useState() hook. Returns the state variable and a setter function [count, setCount]. The setter replaces the old state
this Keyword	Required everywhere. This is a major source of confusion. Methods often need to be manually .bind(this) in the constructor to preserve their context	Not used at all. This eliminates an entire category of common bugs and simplifies the code
Lifecycle	Uses specific lifecycle methods: componentDidMount, componentDidUpdate, componentWillUnmount	Uses the useEffect() hook, which can handle all the same scenarios (mounting, updating, unmounting) in a single API

Demo!

What is State? A Component's Memory

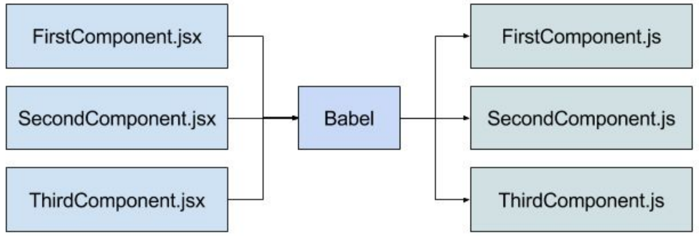
- Many components need to change what's on the screen in response to user actions (like a click)
- State is the data that a component "remembers" and manages internally. It's a component's private memory
- When a component's state changes, React automatically re-renders the component to reflect those changes
- In Class Components, state is managed with this.state and this.setState()
- In Functional Components, we use the useState() Hook
    - We'll cover hooks in detail later



JSX - The Language of React

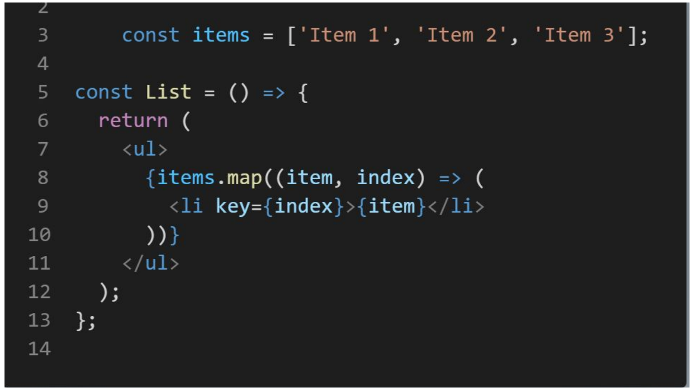
What is JSX? (And What It's Not)

- JSX stands for JavaScript XML
- Permits developers to write HTML-like syntax directly in JavaScript files
- JSX is not HTML. It is not a string. It's a declarative approach to describe the UI
- Crucially, browsers don't natively understand JSX.
  - It needs to be converted to Javascript before being run in the browser
  - A tool called a **compiler** (like Babel, which is included in Vite) reads your JSX
  - The compiler converts the JSX into a standard JavaScript function call: React.createElement()
  - The browser runs React.createElement() function, which tells React to create an <h1> element on the page
- Example
  - JSX: <h1 className="greeting">Hello, world!</h1>
  - Babel compiles it to: React.createElement(h1, { className: 'greeting' }, 'Hello, world!');



The Power of {} in JSX

- The curly braces, {}, are your "escape hatch" back into JavaScript
- Inside {} you can put any valid JavaScript expression
  - A variable. Eg: {userName}
  - Math operations. Eg: {10 * 5}
  - A function call that returns a value. Eg: {formatName(user)}
  - Ternary operators for conditional rendering. Eg: {isLoggedIn ? <p>Welcome</p> : <p>Please log in</p>}
- Code blocks like if/else cannot be placed inside the {}
  - Expressions used inside the {} must resolves to a single value



```
    const items = ['Item 1', 'Item 2', 'Item 3'];

const List = () => {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
};

```

JSX Gotchas

- Rule 1: Return a single root element. Your component can't return two adjacent elements. Wrap them in a <div>.
- Rule 2: HTML attributes are camelCased. onclick becomes onClick. tabindex becomes tabIndex
- The "Why": Since JSX is transformed into JavaScript, attribute names can't conflict with JavaScript's reserved keywords and code style
  - class is a reserved word in JavaScript for creating classes. Therefore, JSX uses className
  - for is a reserved word for loops. Therefore, JSX uses htmlFor for labels

HTML

```html
<div>
  <label for="email-input" class="form-label">Email</label>
  <input type="text" id="email-input" onclick="handleInputClick()">
</div>
```

JSX

```jsx
<div>
  <label htmlFor="email-input" className="form-label">Email</label>
  <input type="text" id="email-input" onClick={handleInputClick} />
</div>
```

Props - Passing Data

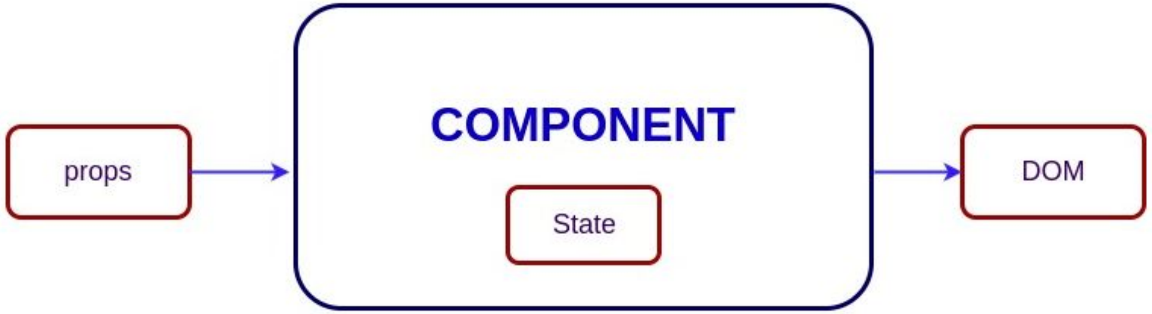
The Problem: Components Are Isolated Islands

- Our components so far have been self-contained. The MovieCard knows about its own movie data, but nothing outside of it
- This is not reusable. If we want to show a list of 10 different movies, do we create 10 different components?
- We need a way to build a generic "template" component and pass in different data from an external source (a parent component)

Source

Props: A Component's "Arguments"

- **Props** (short for properties) are the way we pass data from a parent component down to a child component
- **Analogy**: They are to React components what arguments are to JavaScript functions. They let you configure the component from the outside
- Data flows in a one-way, top-down street: from parent to child
- **Golden Rule**: A component must never modify its own props. They are read-only. This makes our application's data flow predictable



rce

Demo!

That’s it for today.
Questions?

Quiz!

Question 1

- **You write a Class Component and find that the** handleClick **function fails, throwing an error that** *this* **is** undefined**. What is the most common reason for this classic React bug?**
  - A) You forgot to pass props to the constructor
  - B) The render method cannot access functions from the class
  - C) this is a reserved keyword and cannot be used in Class Components
  - D) The handleClick method was passed as an event handler, which changed its execution context, so this was no longer bound to the component instance
  -

Correct Answer: D

This is the classic binding problem in React class components. When you pass a method like this.handleClick to an event listener (e.g., onClick), the context of this is lost by the time the event fires. The solution is to manually bind this in the constructor (e.g., this.handleClick = this.handleClick.bind(this)), which ensures this always refers to the component instance. We'll discover more about this in the future.

Question 2

- What does it mean when we say "browsers do not understand JSX"?
  - A) JSX is an older version of HTML that modern browsers have deprecated.
  - B) You must have the React browser extension installed for JSX to work
  - C) JSX syntax is converted into React.createElement() function calls by a compiler like Babel before the code is sent to the browser
  - D) JSX can only be run on the server and not in a user's browser

Correct Answer: C

Question 3

- Which of the following statements about "props" is TRUE?
  - A) Props are a component's internal memory and can be changed by the component itself using this.setState.
  - B) A component should treat its props as read-only; it should never attempt to modify them.
  - C) Props allow data to flow bi-directionally, from parent-to-child and from child-to-parent.
  - D) Props can only be simple data types like strings and numbers.

Correct Answer: B

Question 4

- You see the following line of code inside a component's render method: <p>{this.props.username}</p>. What can you determine for certain about this component?
  - A) It is a Functional Component
  - B) It is a Class Component
  - C) It is using a Hook to manage username
  - D) The code will cause an error because this is not defined

Correct Answer: B

Question 5

- Why must we use className instead of class when adding a CSS class to an element in JSX?
    - A) className has more features and allows for multiple classes, whereas class only allows one.
    - B) It's a new standard in HTML5 that React has adopted for better performance.
    - C) class is a reserved keyword in JavaScript used to define classes, and since JSX is converted to JavaScript, using class as a prop name would cause a syntax conflict.
    - D) className is just a naming suggestion; class will still work correctly in most browsers

Correct Answer: C

This is a direct consequence of JSX being JavaScript under the hood. To avoid conflicts with JavaScript's own syntax and reserved words

Have a
good one!