**1. JSX (JavaScript XML)**

**Definition:**  
JSX is a syntax extension for JavaScript. It looks like HTML inside JavaScript. React uses JSX to describe the **UI structure** in a declarative way.

Example:

const element = <h1>Hello, React!</h1>;

**Why JSX?**

1. **Declarative Syntax:** Easier to read and write UI code.
2. **Compile-time Optimization:** JSX is compiled to React.createElement() calls.
3. **Dynamic UI:** You can embed JavaScript expressions inside JSX.

**JSX Basics**

1. **Embedding JavaScript in JSX**

Use curly braces {}:

const name = "Veda"; const element = <h1>Hello, {name}!</h1>;

1. **JSX is an Expression**

You can use JSX in variables, functions, conditionals:

const isLoggedIn = true; const element = <h1>{isLoggedIn ? "Welcome!" : "Please login"}</h1>;

1. **JSX Tags must have one parent**

You cannot return multiple sibling elements without a wrapper:

*// ❌ Invalid* return ( <h1>Hello</h1> <p>World</p> ); *// ✅ Valid* return ( <div> <h1>Hello</h1> <p>World</p> </div> ); *// ✅ Also valid using Fragment* return ( <> <h1>Hello</h1> <p>World</p> </> );

1. **Attributes in JSX**

* JSX attributes are camelCase:

<img src="logo.png" alt="Logo" /> <button onClick={handleClick}>Click Me</button>

1. **JSX is not HTML**

* class → className
* for → htmlFor
* Inline styles: object syntax

<div style={{ color: "red", fontSize: "20px" }}>Hello</div>

1. **JSX Compiles to React.createElement()**

const element = <h1>Hello</h1>;

is compiled to:

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

**Advanced JSX Features**

1. **Conditional Rendering**

{isLoggedIn && <h1>Welcome Back!</h1>}

1. **Rendering Lists**

const numbers = [1, 2, 3]; const listItems = numbers.map((n) => <li key={n}>{n}</li>); return <ul>{listItems}</ul>;

1. **JSX Spread Attributes**

const props = { id: "myDiv", className: "container" }; <div {...props}>Hello</div>;

**2. React Components**

React **Components** are **reusable UI blocks**. Everything in React is a component.

**Types of Components**

**A. Functional Components (Modern)**

* Simple JS functions that return JSX.
* Can use hooks like useState, useEffect.

function Greeting(props) { return <h1>Hello, {props.name}</h1>; } *// Using arrow function* const Greeting = ({ name }) => <h1>Hello, {name}</h1>;

**Advantages:**

* Easier to read and test.
* Supports Hooks.
* No this keyword.

**B. Class Components (Old/Legacy)**

* ES6 classes extending React.Component.
* Uses render() method.
* Can have state using this.state.

class Greeting extends React.Component { constructor(props) { super(props); this.state = { count: 0 }; } render() { return <h1>Hello, {this.props.name}</h1>; } }

**Advantages:**

* Full React features before Hooks.
* Can handle lifecycle methods (componentDidMount, componentDidUpdate, etc.)

**Disadvantages:**

* More verbose.
* this can be confusing.

**3. Component Props**

Props are **inputs to a component**.

function Greeting({ name }) { return <h1>Hello, {name}</h1>; } <Greeting name="Veda" />

* Props are **read-only**.
* Used for passing **data from parent → child**.

**Interview note:**

* Functional component props are just function parameters.
* Class component props are accessed via this.props.

**4. Component State**

State is **internal data** that can change over time.

**Functional component example:**

import { useState } from "react"; function Counter() { const [count, setCount] = useState(0); return ( <div> <h1>{count}</h1> <button onClick={() => setCount(count + 1)}>Increment</button> </div> ); }

* useState returns [state, setState].
* Changing state **re-renders** the component.

**Class component example:**

class Counter extends React.Component { constructor() { super(); this.state = { count: 0 }; } increment = () => { this.setState({ count: this.state.count + 1 }); }; render() { return ( <div> <h1>{this.state.count}</h1> <button onClick={this.increment}>Increment</button> </div> ); } }

**5. Component Lifecycle (Important for interviews)**

1. **Mounting** – Component is added to DOM
   * constructor(), render(), componentDidMount() (Class)
   * useEffect(() => {}, []) (Functional)
2. **Updating** – Component re-renders due to state/props change
   * shouldComponentUpdate(), componentDidUpdate() (Class)
   * useEffect(() => {}, [deps]) (Functional)
3. **Unmounting** – Component removed from DOM
   * componentWillUnmount() (Class)
   * useEffect(() => { return cleanup }, []) (Functional)

**6. Child to Parent Communication**

* Parent passes **callback function** as a prop.
* Child calls it to send data **up**.

function Child({ onSend }) { return <button onClick={() => onSend("Hello from Child")}>Send</button>; } function Parent() { const handleMessage = (msg) => { console.log(msg); }; return <Child onSend={handleMessage} />; }

**7. Key Points for Interviews**

1. **JSX:**
   * Looks like HTML, actually JavaScript.
   * Must have a single parent.
   * Can embed JS expressions.
   * Attributes use camelCase.
   * Compiles to React.createElement.
2. **Components:**
   * Two types: Functional (modern), Class (legacy).
   * Can have props (read-only) and state (mutable).
   * Reusable, composable blocks of UI.
   * Functional + Hooks are preferred today.
3. **Props vs State:**

| **Feature** | **Props** | **State** |
| --- | --- | --- |
| Mutable? | No | Yes |
| Passed from | Parent | Internal to component |
| Re-render? | Yes, if parent changes | Yes, when updated |
| Use Case | Pass data & callbacks | Manage local component data |

1. **Best Practices:**
   * Use functional components + hooks.
   * Keep components small and reusable.
   * Lift state up if needed.
   * Use Fragments to avoid extra DOM nodes.