**React JS**

**What is React?**

React is a front-end JavaScript library ,it is used for building UI components.

React was developed by the Facebook Software Engineer Jordan Walke. Latest version of React.JS is 19.0.0 (December 2024).

**How does React Work?**

React creates a VIRTUAL DOM in memory.

Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulations, before making the changes in the browser DOM.React only changes what needs to be changed.

**What is JSX?**

**JSX** stands for **JavaScript XML**.  
It is a **syntax extension** for JavaScript, used with **React** to describe what the UI should look like.

JSX allows you to write **HTML code inside JavaScript**. This makes it easier to build and visualize UI components.

**Rules of JSX**

**1. Return Only One Parent Element**  
JSX must have **one root element**. Use a <div> or <> </> (Fragment) to wrap multiple elements

return (

<div>

<h1>Hello</h1>

<p>Welcome</p>

</div> );

**2. Use className Instead of class**  
Since class is a reserved word in JavaScript, use className for CSS classes.

<div className="container">Content</div>

**3. Use camelCase for Attributes**  
HTML attributes like onclick, for become onClick, htmlFor in JSX.

<button onClick={handleClick}>Click Me</button>

<label htmlFor="name">Name:</label>

**4. Expressions in Curly Braces {}**  
You can use JavaScript expressions inside {} in JSX.

const name = "Shruthi";

<h1>Hello, {name}</h1>

**5. Self-Closing Tags Must End with /**  
Tags like <img>, <input> must be closed like <img />

<img src="logo.png" alt="Logo" />

**6. JSX Must Be Inside a JavaScript File**  
JSX is typically written inside .js or .jsx files.

**7. Comments in JSX**  
Use {/\* comment \*/} for comments inside JSX.

return (

<div>

{/\* This is a comment \*/}

<p>Hello</p>

</div>

);

**React Components**

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Components come in two types, Class components and Function components

**Class Component**

A class component must include the extends React.Component statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.

The component also requires a render() method, this method returns JSX.

Class components can hold and manage **state** and **lifecycle methods** like componentDidMount, componentDidUpdate, etc.

**Example**

class Car extends React.Component {

render() {

return <h2>Hi, I am a Car!</h2>;

}

}

**Functional Component**

A Functional Component is a simple JavaScript function that returns JSX. Earlier, it was stateless, but now with React Hooks, it can also manage state and side effects.

**Example**

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

**Difference b/w Class and Functional component**

| **Feature** | **Class Component** | **Functional Component** |
| --- | --- | --- |
| Syntax | Uses ES6 class | Uses function or arrow function |
| State Management | Uses this.state | Uses useState() hook |
| Lifecycle Methods | Supported (e.g., componentDidMount) | Uses useEffect() hook |
| this keyword | Required (e.g., this.props) | Not required |
| Simplicity | More complex | Simpler and cleaner |

**Props**

Props stands for "properties". They are read-only data passed from one component to another in React. Props are used to send data from a parent component to a child component, making components dynamic and reusable.

**Why Props Are Used**

Props are used to pass dynamic data between components. They help in making reusable components and allow communication between components.

**How Props Work**

A parent component passes data using custom attributes in JSX.  
The child component receives props as a parameter (in functional components) or through this.props (in class components).

Example (Functional Component)

function Greet(props) {

return <h1>Hello, {props.name}</h1>;

}

function App() {

return <Greet name="Shruthi" />;

}

**Output:**  
Hello, Shruthi

**Example (Class Component)**

class Greet extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

function App() {

return <Greet name="Shruthi" />;

}

**Key Features of Props**

**1. Props are Read-Only**  
Child components cannot modify the props they receive.

props.name = "New Name"; // Not allowed

**2. Props Can Be of Any Data Type**  
You can pass strings, numbers, arrays, objects, functions, even components.

<Greet name="Shruthi" age={21} hobbies={["Coding", "Reading"]} />

**3. Props are Immutable in the Child Component**  
They can be used, but not changed by the component receiving them.

**4. Props Can Be Passed to Child Components**  
You can pass props to any level of components in the tree.

**5. Destructuring Props (Functional Components)**

Instead of using props.name, you can directly extract the value using destructuring.

import React from 'react';

function Greet(props) {

const { name, age } = props;

return (

<div>

<h1>Hello, {name}</h1>

<p>You are {age} years old</p>

</div>

);

}

function App() {

return <Greet name="Shruthi" age={21} />;

}

export default App;

**6. Default Props**

You can define **default values** for props using defaultProps.

function Greet({ name }) {

return <h1>Hello, {name}</h1>;

}

Greet.defaultProps = {

name: "Guest"

};

**Lifecycle of Components**

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

**1. Mounting Phase (Component Creation)**

This is when the component is **being created and inserted into the DOM**.

**Lifecycle Methods:**

**1. constructor(props)**

* The constructor is used to initialize the component.
* It is called when the component is first created, before it is mounted.
* You can set up the initial state and bind methods inside the constructor.
* Called only once when the component is created.

**Example:**

constructor(props) {

super(props);

this.state = { count: 0 }; // Initialize state

}

**2. static getDerivedStateFromProps(props, state)**

* This method is called **before every render**, both when the component mounts and when the props or state change.
* It allows you to **update the state** based on changes in **props**.
* It returns an **object** to update the state, or **null** if no state update is needed.

**Example**

static getDerivedStateFromProps(nextProps, nextState) {

if (nextProps.count !== nextState.count) {

return { count: nextProps.count }; // Update state if props have changed

}

return null; // No update needed

}

**3. render()**

* Returns the **JSX** (HTML-like code) that tells React what to show on the screen.
* It is the **only required method** in a class component.
* It runs During mounting and updating phases.

**Example:**

render() {

return <h1>Welcome, {this.props.name}</h1>;

}

**4. componentDidMount()**

* Runs **after the component appears** on the screen (right after the component is mounted to the DOM).
* Commonly used to **fetch data**, **set timers**, or **add event listeners**.

**Example:**

componentDidMount() {

console.log("Component has mounted!");

// Example: fetch data from API

}

**Updating**

This phase occurs when the component’s state or props change, causing it to re-render.

**1. static getDerivedStateFromProps(props, state)**

**2. shouldComponentUpdate(nextProps, nextState)**

* Helps to improve performance by letting you **control** whether the component should re-render.
* If you return false, the component **won’t re-render**.
* It will run Before re-rendering the component after a change in props or state.

**Example:**

shouldComponentUpdate(nextProps, nextState) {

return nextProps.count !== this.props.count; }

**3. render()**

* It’s called again to update the UI when needed.

**4. getSnapshotBeforeUpdate(prevProps, prevState)**

* Lets you **capture information** (like scroll position) from the DOM **before** React updates it.
* The value you return from this method is passed to componentDidUpdate.
* It runs right before React updates the DOM after a render.

**Example**

getSnapshotBeforeUpdate(prevProps, prevState) {

return document.documentElement.scrollTop;

}

**5. componentDidUpdate(prevProps, prevState, snapshot)**

* Runs after the component’s **DOM is updated**.
* Often used to **work with updated DOM**, make **API calls**, or update based on previous props/state.
* After the component updates and the DOM is re-rendered.

**Example:**

componentDidUpdate(prevProps, prevState, snapshot) {

console.log("Component updated");

}

**Unmounting Phase**

This phase occurs **right before** the component is removed from the DOM.

**1. componentWillUnmount()**

* Clean up anything the component was using (e.g. **timers**, **API calls**, **event listeners**).
* Prevents memory leaks and bugs.
* it runs Just before the component is unmounted and removed.

**Example:**

componentWillUnmount() {

clearInterval(this.timer); // example cleanup

}

**React Events**

Just like HTML DOM events, React can perform actions based on user events.

React has the same events as HTML: click, change, mouseover etc.

**Adding Events**

* React events are written in camelCase syntax: onClick instead of onclick.
* React event handlers are written inside curly braces: onClick={shoot}  instead of onclick="shoot()".

Example:

<button onClick={shoot}>Take the Shot!</button>

**Passing Arguments to event handler**

To pass an argument to an event handler, use an arrow function.

function Football() {

const shoot = (a) => {

alert(a);

}

return (

<button onClick={() => shoot("Goal!")}>Take the shot!</button>

);

}

**React Conditional Rendering**

In React, you can conditionally render components. There are several ways to do this.

**1) if Statement**

We can use the if JavaScript operator to decide which component to render.

function MissedGoal() {

return <h1>MISSED!</h1>;

}

function MadeGoal() {

return <h1>GOAL!</h1>;

}

function Goal(props) {

const isGoal = props.isGoal;

if (isGoal) {

return <MadeGoal/>;

}

return <MissedGoal/>;

}

**2. Logical && Operator**

Another way to conditionally render a React component is by using the && operator.

**Example:**

function Garage(props) {

const cars = props.cars;

return (

<>

<h1>Garage</h1>

{cars.length > 0 &&

<h2>

You have {cars.length} cars in your garage.

</h2>

}

</>

);

}

const cars = ['Ford', 'BMW', 'Audi'];

**3. Ternary Operator**

Another way to conditionally render elements is by using a ternary operator.

condition ? true : false

import React from 'react'

function TernaryOperator() {

    const Goal=false

  return (

    <div>

      {Goal? <h1>Goal</h1>:<h1>Missed</h1>}

    </div>

  )

}

export default TernaryOperator

**React Lists**

React Lists allow you to render a list of items dynamically using .map().

**Keys**

Keys are unique identifiers given to each element in the list to help React identify which items changed, added, or removed. if an item is updated or removed, only that item will be re-rendered instead of the entire list.

function UserList() {

    const users=[

        {id:1 , name:"Alice"},

        {id:2,name:"Bob"},

        {id:3, name:"john"},

        {id:4,name:"Shruthi"} ]

  return (

    <div>

      {users.map(user=>(

        <li key={user.id}>{user.name}</li>

      ))}

    </div>

  )

}