**Question 1: What are props in React.js? How are props different from state?**

**Props in React.js**

Props (short for "properties") are used to pass data from a parent component to a child component in React. They are read-only and allow components to be reusable by dynamically receiving different data.

**Key Characteristics of Props:**

* Immutable: Props cannot be modified inside the component that receives them.
* Passed from parent to child: Props help components communicate with each other.
* Used to make components dynamic: They allow components to receive different values and render accordingly.

**Example of Props:**

function Greeting(props) {

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

}

function App() {

return <Greeting name="Dharmik" />;

}

**Difference Between Props and State**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Props** | **State** |
| **Mutability** | Immutable (cannot be changed by the component) | Mutable (can be updated within the component) |
| **Where it is used** | Passed from parent to child | Managed within the component itself |
| **Who updates it?** | Parent component | Component itself using useState (for functional components) |
| **Re-renders** | Causes re-render when new props are received | Causes re-render when state changes |
| **Usage** | Used for communication between components | Used to manage component behaviour and data |

**Question 2: Explain the concept of state in React and how it is used to manage component data.**

**State in React**

State is a built-in object in React that allows components to store and manage dynamic data. When the state of a component changes, react automatically re-renders the component to reflect the updated data.

**Key Characteristics of State**

1. **Mutable**: Unlike props, state can be updated within the component.
2. **Private to the Component**: Each component manages its own state, which is not accessible to other components unless explicitly passed.
3. **Triggers Re-render**: Updating state causes React to re-render the component to reflect changes in the UI.
4. **Used with Hooks (Functional Components)**: Managed using the useState hook in functional components.
5. **Managed with this.state in Class Components** (Older approach).

**Using State in Functional Components (with useState)**

React introduced hooks in version 16.8, allowing functional components to use state via the useState hook.

**Example: Counter with State**

import { useState } from "react";

function Counter() {

// Declare a state variable 'count' with an initial value of 0

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

return (

<div>

<p>Count: {count}</p>

{/\* Update state when button is clicked \*/}

<button onClick={() => setCount(count + 1)}>Increase</button>

</div>

)}:

export default Counter;

**Using State in Class Components (Older Approach)**

Before hooks, state was managed in class components using this.state and updated with this.setState().

**Example: Counter with State in Class Component**

import React, { Component } from "react";

class Counter extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

increaseCount = () => {

this.setState({ count: this.state.count + 1 });

};

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.increaseCount}>Increase</button>

</div>

);

}

}

export default Counter;

**Key Differences (Functional vs. Class Components):**

* Functional components use **hooks** (useState).
* Class components use this.state and this.setState().

**Question 3: Why is this.setState() used in class components, and how does it work?**

**Why is this.setState() Used in Class Components?**

In class components, state is managed using the this.state object. However, state should never be modified directly. Instead, React provides the this.setState() method to update state safely and efficiently.

**How this.setState() Works**

1. Merges State Updates: It updates the component’s state and triggers a re-render.
2. Asynchronous Behaviour: this.setState() does not update state immediately; instead, React batches multiple state updates for better performance.
3. Ensures Proper Re-rendering: Directly modifying this.state does not trigger a re-render, but this.setState() ensures the component updates correctly.

**Example: Counter Component Using this.setState()**

**import React, { Component } from "react";**

**class Counter extends Component {**

**constructor(props) {**

**super(props);**

**this.state = { count: 0 };**

**}**

**increaseCount = () => {**

**this.setState({ count: this.state.count + 1 });**

**};**

**render() {**

**return (**

**<div><p>Count: {this.state.count}</p>**

**<button onClick={this.increaseCount}>Increase</button>**

**</div>**

**);**

**}**

**}**

**export default Counter;**

**Why Not Modify this.state Directly?**

this.state.count = this.state.count + 1; **// ❌ Wrong! Won't trigger re-render**

** Issue:** Direct updates to this.state do not notify React to re-render the component.

** Solution:** Always use this.setState() to ensure state changes are recognized.

**Handling State Updates Based on Previous State**

Since this.setState() is **asynchronous**, relying on this.state directly inside setState() can lead to incorrect updates. Instead, use a **callback function**:

**Example: Correct Way to Update State Based on Previous State**

this.setState((prevState) => ({

count: prevState.count + 1

}));

**Why?**

* **Ensures correct state updates**, especially when multiple updates happen at the same time.
* **Avoids stale state issues**.

**Example: Counter with Multiple Clicks**

increaseCount = () => {

this.setState((prevState) => ({

count: prevState.count + 1

}));

};