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

Answer:

Props (Properties) are a mechanism for passing data from a parent component down to a child component. They are read-only and immutable, meaning a component cannot change its own props. They allow for a unidirectional (one-way) data flow, which makes applications easier to debug and understand.

**Key Differences Between Props and State:**

| Aspect | Props | State |
| --- | --- | --- |
| Purpose | To pass data down from a parent to a child component. | To manage data within a component itself. |
| Mutability | Immutable (read-only). The child component cannot change its props. | Mutable. The component can and does change its own state using setState (class) or the setter function (functional). |
| Ownership | Owned and set by the parent component. | Owned and managed within the component itself. |
| Access | In functional components: props.propName In class components: this.props.propName | In functional components: stateVariable (via useState Hook) In class components: this.state.stateName |
| Analogy | Function parameters. You pass arguments to a function. | A component's internal variables. The function uses them for its own logic. |

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

Answer:

State is a built-in React object that contains data or information that can change over time and influences what a component renders. It is managed within the component itself and is its "memory."

**How it is used to manage component data:**

Dynamic Data: State allows components to create and manage their own dynamic data. For example, a Counter component would use state to keep track of the current count value.

Re-rendering the UI: When a component's state changes, React automatically re-renders the component and its children. This is the key to creating interactive UIs. The UI always reflects the current state of the data.

Single Source of Truth: For a given component, the state is the single source of truth for any data that changes. You don't need to manually manipulate the DOM; you change the state, and React updates the DOM to match.

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

Answer:

**Why it's used**:  
In class components, this.setState() is the only correct way to update the component's state. You should never modify the state directly by writing this.state.count = 1 because:

It won't trigger a re-render: The component will not update on the screen to reflect the new data.

It breaks React's consistency: Direct mutation can lead to bugs and make the application's behavior unpredictable, especially with features that rely on the state's immutability.

**How it works**:

**Asynchronous** Update: setState() enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state. It is asynchronous for performance reasons. React may batch multiple setState() calls into a single update.

**Accepts an Object or a Function**: You can pass either an object or an updater function to setState().

**Object: Merges the provided object into the current state.**

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

**Function**: Used when the new state depends on the previous state. This is the safer method to avoid state update inconsistencies due to batching.

this.setState((prevState, props) => {

return { count: prevState.count + 1 };

});

**Merging, Not Replacing**: When you call setState(), React performs a shallow merge of the object you provide into the current state. For example, if your state has { count: 0, name: 'John' } and you call this.setState({ count: 1 }), the resulting state will be { count: 1, name: 'John' }. The name property is left intact.

**Triggers Re-rendering**: After updating the state, React schedules a re-render of the component and its children, ensuring the UI is always in sync with the data.