State and Props

In React, **state** and **props** are essential concepts for managing and passing data within components. They allow you to build dynamic and interactive user interfaces by controlling the flow of data and ensuring that components update when their data changes.

1. Understanding State

State is a built-in object in React components that allows you to store property values that belong to that component. When the state of a component changes, the component re-renders, reflecting the updated state in the UI.

a. Using State in Functional Components (with Hooks)

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

Example: Counter Component Using `useState`

jsx

```
import React, { useState } from 'react';

function Counter() {
  // Declare a state variable called "count" initialized to 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      {/* Update state on button click */}
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Counter;
```

- **`useState(0)`:** Initializes the `count` state variable to `0`.
- **`setCount`:** A function that updates the `count` state. Calling `setCount` with a new value causes the component to re-render with the updated state.

b. Using State in Class Components

Before hooks, state was managed in class components using the `this.state` object and the `this.setState()` method.

Example: Counter Component Using Class Component

jsx

```
import React, { Component } from 'react';

class Counter extends Component {
  constructor(props) {
    super(props);
    // Initialize state
    this.state = { count: 0 };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        {/* Update state on button click */}
        <button onClick={() => this.setState({ count:
this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}

export default Counter;
```

- **`this.state = { count: 0 }`**: Initializes the `count` state variable.
- **`this.setState({ count: this.state.count + 1 })`**: Updates the `count` state and re-renders the component.

2. Understanding Props

Props (short for "properties") are read-only attributes passed from a parent component to a child component. Props are used to pass data and event handlers to child components, allowing components to be more dynamic and reusable.

a. Passing Data with Props

Props are passed to a component similarly to how HTML attributes are passed to an element.

Example: Passing Props

jsx

```
import React from 'react';

function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
}

export default App;
```

- **`props.name`:** The `Greeting` component receives the `name` prop from its parent component (`App`).
- **Reusability:** The `Greeting` component is reusable because it can display a different name based on the passed prop.

b. Default Props

You can define default values for props in case they are not passed by the parent component.

Example: Default Props

jsx

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

// Set default props
Greeting.defaultProps = {
  name: 'Stranger'
};

function App() {
  return (
    <div>
      <Greeting />         {/* Uses default prop */}
      <Greeting name="Bob" />  {/* Uses passed prop */}
    </div>
  );
}

export default App;
```

- **`Greeting.defaultProps`**: If the `name` prop is not provided, it defaults to "Stranger".

3. State vs. Props

- **State:**
  - Managed within the component.
  - Mutable (can change over time).
  - Used to handle data that can change (e.g., user input, API responses).
- **Props:**
  - Passed from a parent component.
  - Immutable (cannot be changed by the child component).
  - Used to pass data and event handlers to child components.

4. Lifting State Up

When multiple components need to share and synchronize state, the state should be lifted up to their nearest common ancestor. The common ancestor can manage the state and pass it down to child components via props.

Example: Lifting State Up

jsx

```
import React, { useState } from 'react';

function TemperatureInput(props) {
  return (
    <div>
      <label>{props.scale === 'c' ? 'Celsius' :
'Fahrenheit'}:</label>
      <input value={props.temperature} onChange={e =>
props.onTemperatureChange(e.target.value)} />
    </div>
  );
}

function Calculator() {
  const [temperature, setTemperature] = useState('');
  const [scale, setScale] = useState('c');

  const handleCelsiusChange = (temp) => {
    setTemperature(temp);
    setScale('c');
  };

  const handleFahrenheitChange = (temp) => {
    setTemperature(temp);
    setScale('f');
  };

  return (
    <div>
      <TemperatureInput
        scale="c"
        temperature={scale === 'f' ? (parseFloat(temperature) -
32) * 5 / 9 : temperature}
        onTemperatureChange={handleCelsiusChange}
      />
      <TemperatureInput
        scale="f"
        temperature={scale === 'c' ? (parseFloat(temperature) *
9 / 5) + 32 : temperature}
        onTemperatureChange={handleFahrenheitChange}
      />
    </div>
  );
}

export default Calculator;
```

- **`Calculator`:** The parent component manages the shared state (`temperature` and `scale`).

- **`TemperatureInput`:** The child components receive the temperature and scale via props and notify the parent of changes via callback props (`onTemperatureChange`).

5. PropTypes for Prop Validation

React provides `PropTypes` to validate the props passed to a component, ensuring that they are of the correct type.

Example: Using PropTypes

jsx

```
import React from 'react';
import PropTypes from 'prop-types';

function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

// Prop validation
Greeting.propTypes = {
  name: PropTypes.string
};

function App() {
  return <Greeting name={123} />;  // Will produce a warning
because `name` is not a string
}

export default App;
```

- **`PropTypes`:** A set of validators that can be used to ensure the props received by a component match the expected types.

Summary

- **State**: Used to manage data within a component that can change over time. Functional components use the `useState` hook, while class components use `this.state` and `this.setState()`.
- **Props**: Used to pass data and event handlers from parent to child components. Props are immutable and cannot be changed by the receiving component.
- **Lifting State Up**: When multiple components need to share state, lift the state to the nearest common ancestor and pass it down via props.
- **PropTypes**: Used to validate the types of props passed to a component, helping to catch potential bugs.