Agenda:

- Lifting States up in react
- Component's Composition in react
- Component's Inheritance in react
- Composition vs Inheritance

Lifting The State Up in React Js

Introduction:

In React, each component has its state. **Lifting the state up** is a valuable concept for React developers since we often have a state that accommodated within a single component but needs to be shared with its siblings. We lift the state up to make the parent state a single shared state ar a sole "source of truth" and pass the parent's data to its children.

This concept is called lifting state up. It is of great use to maintain data consistency in our react applications.

When to lift the state up ?

So when to do the process of lifting the state up?. Many times, multiple components must reflect the same changing data. And if the data is not in syr between the "parent and children components" or "cousin components", it is recommended to lift the shared state up to the closest common ancestor.

Let's understand this concept of lifting state up with an example and see how this will work in actual practice:

In this example, we'll make a simple speed calculator to see if we're within the speed limit.

And on the output page, it will display three sections:

1. Accepting the speed input in kilometers per hour (kmph)
2. Receiving speed input in miles per hour
3. The line will tell you whether you are within the speed limit.

In addition, we want our speeds from both inputs to be synced. The change in one should be reflected by the other too.

Let's see this example step by step:

First, we'll create a component called SpeedMonitor that accepts the 'kmph' speed in kilometers per hour as a prop and displays whether it's within th limit as shown below:

```
function SpeedMonitor(props) {
  if (props.kmph <= 80) {
  return <p>You are within the speed limit.</p>;
}

 return <p>You are exceeding the speed limit.</p>;
}
```

After that, we'll make a component called Calculator. It creates an `<input>` element that allows you to enter the speed and saves its value this.state.speed.

In addition, the SpeedMonitor is rendered for the current input value.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {speed: ''};
}

  handleChange(e) {
  this.setState({speed: e.target.value});
}

  render() {
  const speed = this.state.speed;

  return (
      <fieldset>
```

```
        <legend>Enter speed in kilometer per hour:</legend>
        <input
        value={speed}
        onChange={this.handleChange} />
      <SpeedMonitor
        kmph={parseFloat(speed)} />
    </fieldset>
    );
  }
}
```

Output:

Now, taking a second input:

Now our next aim is to take a second input in miles per hour in addition to our previous input in kilometers per hour, and we even want them to be sync.

To begin, we will extract a SpeedInput component from the Calculator. We will add a new scale prop to it that can be either "km" or "mi" as shown below

Code:

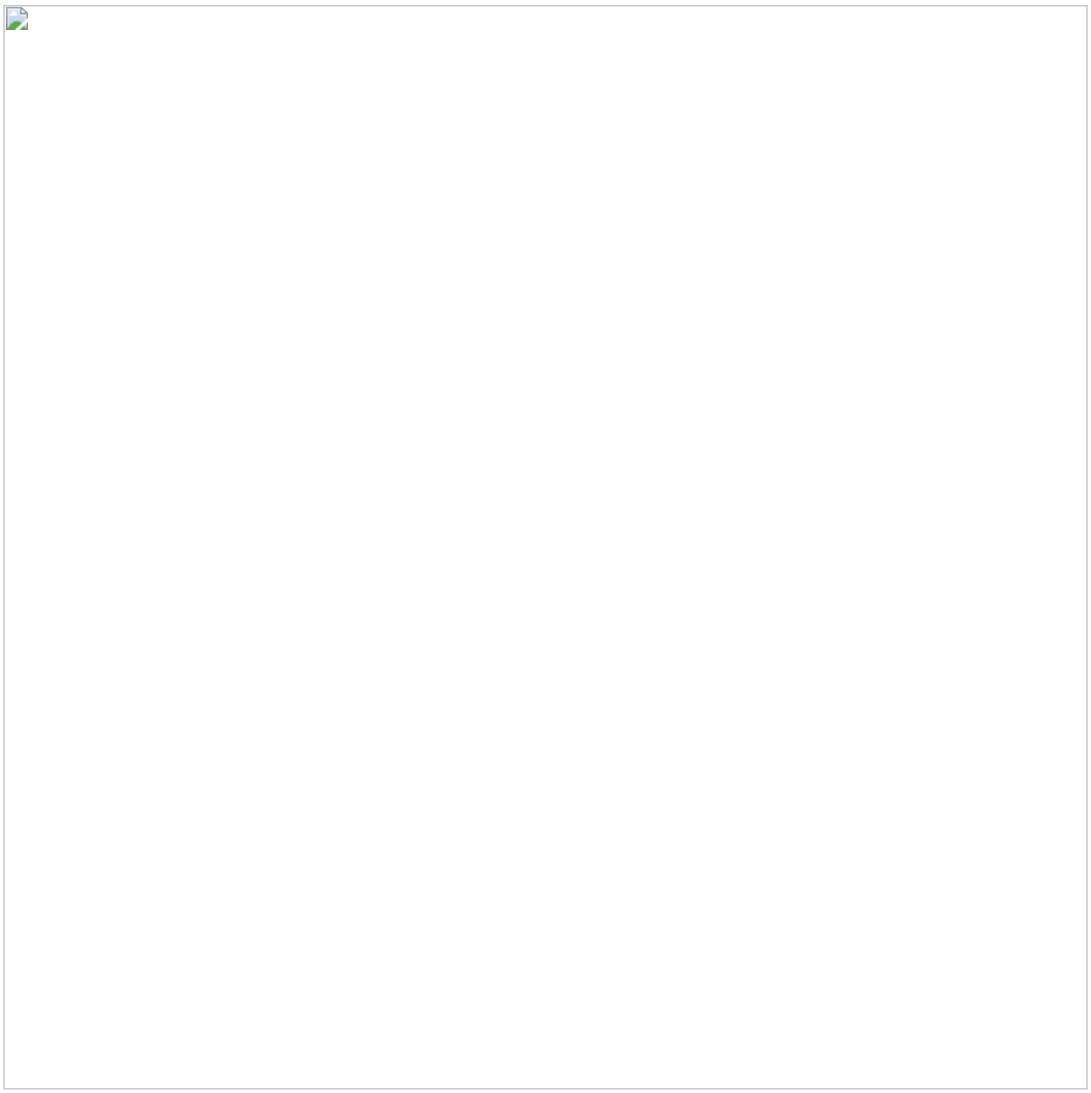
```
const scaleNames = {
  km: 'km per hour ',
  mi: 'mi per hour.'
};

class SpeedInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {speed: ''};
  }

  handleChange(e) {
    this.setState({speed: e.target.value});
  }

  render() {
    const speed = this.state.speed;
    const scale = this.props.scale;

    return (
      <fieldset>
        <legend>Enter speed in {scaleNames[scale]}:</legend>
        <input value={speed}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Now, we can use the Calculator to display two different speed inputs as shown below:

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <SpeedInput scale="km" />
        <SpeedInput scale="mi" />
      </div>
    );
```

```
    }
  }
```

Output:



Now we have two inputs, but when you change the speed in one of them, the other does not change. At this stage, they are not in sync.

The Calculator's SpeedMonitor is also unavailable for display. This is because the current speed is hidden inside the SpeedInput, and the Calculat doesn't know it.

Now, Writing conversions:

Let's write two functions to convert KMPH to MPH and vice versa as shown below:

```
function toKmph(miph) {
  return (miph / 0.6213);
}

function toMiph(kmph) {
  return (kmph * 0.6213);
}
```

Both of these functions convert numbers. We'll create a new function that takes two arguments: a string speed and a converter function, and returns string. We'll use it to determine the value of one input based on the value of the other.

It returns an empty string when the speed is invalid, and the output is rounded to the third decimal place as shown below:

```
function tryConvert(speed, convert) {
  const input = parseFloat(speed);

  if (Number.isNaN(input)) {
    return '';
```

```
  }

  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;

  return rounded.toString();
}
```

Now, Lifting the state up :

At the current state, both the SpeedInput components keep their values independently. But we want them to be in sync with each other. The change KMPH should reflect the converted change in MPH and vice versa.

Sharing state in React is done by pushing it up to the nearest common ancestor of the components that require it. This is referred to as "raisir state up." The local state will be removed from the SpeedInput and moved to the Calculator instead.

Since the Calculator owns the shared state, it becomes the "source of truth" for the current speed in both inputs. It can instruct the child components have values that are in sync with one another. The two inputs will always be in sync because the props of both SpeedInput components come from th same parent Calculator component.

```
class SpeedInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onSpeedChange(e.target.value);
  }

  render() {
    const speed = this.props.speed;
    const scale = this.props.scale;

    return (
      <fieldset>
        <legend>Enter speed in {scaleNames[scale]}:</legend>
        <input value={speed}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Let's see this happening step by step.

In the SpeedInput component, we'll first change this.state.speed with this.props.speed. Let's suppose this.props.speed already exists for the time bein even though we'll need to pass it from the Calculator later as shown below:

```
render() {
    // Before: const speed = this.state.speed;

  const speed = this.props.speed;
  // ...
```

As we know, props are read-only. The SpeedInput could call this.setState when the speed was in the local state. On the other hand, the SpeedInput ha no control over the speed because it comes from the parent as a prop.

This is commonly fixed in React by making a component **"controlled."** The custom SpeedInput can accept both speed and onSpeedChange props fro its parent Calculator.

Now SpeedInput calls this.props.onSpeedChange: when it needs to update its speed.

```
handleChange(e) {
    // Before: this.setState({speed: e.target.value});
```

```
    this.props.onSpeedChange(e.target.value);
    // ...
```

The parent Calculator component will provide the onSpeedChange prop with the speed prop.

It will respond to the change by changing its local state, causing both inputs to be re-rendered with the new values.

Let's recollect the changes we have made to the SpeedInput so far. We removed the local state from it, and now we are reading this.props.speed instea of this.state.speed. We now call this.props.onSpeedChange(), instead of calling this.setState(), which will be provided by the Calculator whenever w want to make a change.

```
class SpeedInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
```

Let's move on to the Calculator component.

We'll save the speed and scale of the current input in its local state. This is the "source of truth" for both of them, as it is the state we "lifted up" from th inputs. It's the simplest representation of all the information we need to render both inputs.

We could have saved the values of both inputs, but that would have been wasteful. It is sufficient to save the value of the most recently updated inp and the scale it reflects. The value of the other input can then be inferred solely based on the current speed and scale.

Now the inputs stay in sync because their value is calculated from derived from the same state.

Now, no matter which input you edit, this.state.speed and this.state.scale in the Calculator get updated. The value of one input will be recalculated whe the value of the other is changed.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleKmphChange = this.handleKmphChange.bind(this);
    this.handleMiphChange = this.handleMiphChange.bind(this);
    this.state = {speed: '', scale: 'km'};
  }

  handleKmphChang(speed) {
    this.setState({scale: 'km', speed});
  }

  handleMiphChange(speed) {
    this.setState({scale: 'mi', speed});
  }

  render() {
    const scale = this.state.scale;
    const speed = this.state.speed;
    const kmph = scale === 'mi' ? tryConvert(speed, toKmph) : speed;
    const miph = scale === 'km' ? tryConvert(speed, toMiph) : speed;

    return (
      <div>
        <SpeedInput
          scale="km"
          speed={kmph}
          onSpeedChange={this.handleKmphChange} />
        <SpeedInput
          scale="mi"
          speed={miph}
          onSpeedChange={this.handleKmphChange} />
        <SpeedMonitor
          kmph={parseFloat(kmph)} />
      </div>
    );
```

```
    }
  }
```

Output:

Component's Composition in React

In React, we can make components more generic by accepting **props**, which are to React components what parameters are to functions.

Component composition is the name for **passing components as props to other components**, thus creating new components with other components

Consider the example shown below :

```
const Button = ({ onClick, children }) => (
 <button onClick={onClick}>{children}</button>
);

const App = () => {
  const onClick = () => alert('Hey');

  return (
    <Button onClick={onClick}>Click me!</Button>
  );
};
```

Note: `children` is nothing more than a prop to the `Button` component.

In the example above, we have created a Button component and we are calling it in our App component while passing down the props f the `Button` component. In the above code, we passing the `onClick` and `children` props from our App component to the Button component.

Instead of passing down a string to `Button` , we may want to **add an icon** to the text as well as shown below:

```
const App = () => {
  const onClick = () => alert('Hey');

  return (
    <Button onClick={onClick}>
      <img src="/logos/logo.svg" />
      Click me!
    </Button>
  );
};
```

But we're not limited to the `children` prop. We can create more specific props that can accept components as well as shown below:

```
const Button = ({ onClick, icon, children }) => (
 <button onClick={onClick}>{icon}{children}</button>
);

const App = () => {
  const onClick = () => alert('Hey');

  return (
    <Button
      onClick={onClick}
      icon={<img src="/logos/logo.svg" />}
    >
      Click me!
    </Button>
```

```
  );
};
```

And that is the essence of component composition: a simple yet incredibly powerful pattern that makes React components **highly reusable**.

When working on React projects, you'll continuously find yourself refactoring components to be more generic through component composition so th you can use them in multiple places.

Component's Inheritance in React

Inheritance is a way to achieve code reusability when some objects have the same number of properties that can be shared across the app. Inheritanc allows the app to do the coupling between the parent-child component and reuse properties such as state values and function in its child components.

React does not use inheritance except in the initial component class, which extends from the `react` package.

Implementing Inheritance in React

Inheritance uses the keyword `extends` to allow any component to use the properties and methods of another component connected with the parer Using the `extends` keyword, you can allow the current component to access all the component's properties, including the function, and trigger it fro the child component.

Consider the example shown below:

```
import React from "react";

class ParentClass extends React.Component {
  constructor(props) {
    super(props);
    this.callMe = this.callMe.bind(this);
  }

  // ParentClass function
  callMe() {
    console.log("This is a method from parent class");
  }

  render() {
    return false;
  }
}
```

In the above code, we have created a `ParentClass` component, which contains a `callMe()` function.

Here, `ParentClass` extends the component from React as `React.component` , which means the newly created component itself is using th inheritance.

Now, after creating a ParentClass component, let's create a Child component, called `Example`

as shown below:

```
export default class Example extends ParentClass {
  constructor() {
    super();
  }
  render() {
    this.callMe();
    return false;
  }
}
```

The `Example` class extends `ParentClass` so the child class will access all the properties and methods created inside the parent component.

```
render() {
  this.callMe();
  return false;
}
```

Here in the child class, the `this.callMe()` function is called the part of parent class implementation, so the parent component's properties and i methods can be accessed by implementing inheritance in the child component.

Composition vs Inheritance

The techniques for using several components together are done by composition and inheritance in React. This facilitates code reuse. React recommend using composition instead of inheritance as far as feasible, and inheritance should only be utilized in particular instances.

The ‘is-a relationship’ mechanism was used in **inheritance**. Derived components had to inherit the properties of the base component, which mad changing the behaviour of any component quite difficult. The **composition** aspires to be better. Why not inherit only behaviour and add it to the desire component instead of inheriting properties from other components?

Only the behaviour is passed down from composition without the inheritance of properties. Why is this a plus point? It was challenging to add ne behaviour via inheritance as the derived component inherited all of the parent class's properties, making it impossible to add new behaviour. More us cases had to be included. However, we only inherit behaviour in composition, and adding new behaviour is relatively easy.

React proposes utilizing composition instead of inheritance to reuse code between components because React has an advanced composition mode Between Composition and Inheritance in React, we can distinguish the following points:

- We can overuse ‘inheritance’.
- ‘Behavior’ composition can be made simpler and easier.
- Composition is preferred over deep inheritance in React.
- Inheritance inherits the properties of other components, whereas composition merely inherits the behaviour of other components.
- It was difficult to add new behaviour via inheritance since the derived component inherits all of the parent class's properties, making it impossible to add new behaviour.

Conclusion:

In this session, we have studied about :

- lifting state up in react and when we need to lift the state up.
- react’s component’s composition
- react’s component’s inheritance
- and which is better to use and when to use composition or inheritance.

Now, in the next session, we will learn about how to manage states in a react project.

Thank You !