1. Higher Order Component (HOC):

A Higher Order Component (HOC) is a pattern in React that allows you to reuse component logic by wrapping components with additional functionality. HOCs are not part of the React API, but rather a pattern that emerges from the compositional nature of React components.

A HOC is a function that takes a component as input and returns a new component with enhanced functionality. It can inject props, modify component behavior, and provide additional data or methods to the wrapped component. HOCs are commonly used for cross-cutting concerns like authentication, authorization, logging, or data fetching.

Here's an example of a HOC that adds a "loading" prop to a component while it fetches some data:

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

const withDataFetching = (WrappedComponent, dataSource) => {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = {
        data: [],
        isLoading: false,
        error: null
      };
    }

    componentDidMount() {
      this.setState({ isLoading: true });
      fetch(dataSource)
        .then(response => response.json())
        .then(data => this.setState({ data, isLoading: false }))
        .catch(error => this.setState({ error, isLoading: false }));
    }

    render() {
```

```
      const { data, isLoading, error } = this.state;
      return (
        <WrappedComponent
          data={data}
          isLoading={isLoading}
          error={error}
          {...this.props}
        />
      );
    }
  };
};

const MyComponent = ({ data, isLoading, error }) => {
  if (isLoading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <ul>
      {data.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
};

const MyComponentWithFetching = withDataFetching(MyComponent,
'https://api.example.com/data');

export default MyComponentWithFetching;
```

In this example, the `withDataFetching` function is a HOC that takes a component (`WrappedComponent`) and a data source (`dataSource`). It returns a new component that handles the data fetching logic and passes the fetched data, loading state, and error to

the wrapped component (`MyComponent`). The HOC encapsulates the common data fetching behavior, allowing the wrapped component to focus on rendering the data.

Pros:

- **Reusability**: HOCs enable you to reuse component logic across multiple components.

- **Separation of Concerns**: HOCs allow you to separate cross-cutting concerns from the core components.

- **Code Organization**: HOCs help in organizing and abstracting complex logic, making it easier to understand and maintain.

Cons:

- **Prop Clutter**: HOCs can sometimes introduce prop clutter, as they add additional props to the wrapped component.

- **Component Identity**: HOCs create a new component instance each time they are used, which can affect certain optimizations or lead to unnecessary re-renders if not implemented carefully.

Dos and Don'ts:

- Do use HOCs for reusable and cross-cutting concerns.

- Do keep HOCs focused on a single responsibility to maintain code readability.

- Don't overuse HOCs when a simpler solution like render props or React hooks can suffice.

- Don't mutate the wrapped component directly within the HOC; instead, use composition to modify behavior.

2. Props:

Props (short for properties) are a fundamental concept in React that allow you to pass data and configuration to components. Props are passed from parent components to child components and are immutable within the

child component. They provide a way to customize the behavior and appearance of components dynamically.

In React, props are specified as attributes on JSX elements and are accessible within the component via the `props` object.

Here's an example that demonstrates the usage of props:

```
import React from 'react';

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

const App = () => {
  return <Greeting name="John" />;
};

export default App;
```

In this example, the `Greeting` component receives the `name` prop and renders a personalized greeting. The `App` component passes the prop `name="John"` to the `Greeting` component.

Dos:

- Do use props to pass data and configuration to child components.

- Do pass props in a declarative manner to make components more reusable.

- Do provide default values for props using the `defaultProps` property or by checking for undefined values within the component.

Don'ts:

- Don't modify props within the child component, as they are intended to be immutable.

- Don't rely on props for complex state management; instead, use React's `state` or other state management libraries.

3. Prop Drilling:

Prop drilling refers to the process of passing props through multiple intermediate components in order to reach a deeply nested child component that needs access to those props. It occurs when there is a chain of parent-child components where the intermediate components don't require the props but need to forward them down to the desired component.

While prop drilling is a common pattern in React, it can lead to code clutter and make it harder to maintain or refactor components. To mitigate this issue, alternative patterns like context or state management libraries can be used to avoid excessive prop drilling.

Example of prop drilling:

```
import React from 'react';

const GrandChild = ({ value }) => {
  return <p>{value}</p>;
};

const Child = ({ value }) => {
  return <GrandChild value={value} />;
};

const Parent = ({ value }) => {
  return <Child value={value} />;
};

const App = () => {
  const value = 'Hello, Prop Drilling!';
  return <Parent value={value} />;
};

export default App;
```

In this example, the `App` component passes the `value` prop to the `Parent` component, which passes it to the `Child` component, and finally, it reaches the `GrandChild` component where it is rendered.

Prop drilling can become cumbersome when multiple levels of nesting are involved or when components that don't need the prop are required to pass it down. In such cases, using context or state management libraries like Redux can be more efficient and maintainable.

Dos and Don'ts:

- Do use prop drilling when the intermediate components genuinely need access to the prop.

- Do consider alternative patterns like context or state management libraries when prop drilling becomes excessive or hard to manage.

- Don't use prop drilling as the only solution for passing data between deeply nested components if a better alternative is available.

4. Overview of Component Lifecycle:

In earlier versions of React, components had several lifecycle methods that provided hooks to perform actions at different stages of a component's lifecycle. With the introduction of React 16.3, the legacy lifecycle methods have been partially deprecated in favor of newer lifecycle methods and hooks.

The component lifecycle can be divided into three main phases: mounting, updating, and unmounting.

Mounting:

1. `constructor()`: The constructor is called when the component is created, and it's used for initializing state and binding methods.

2. `render()`: The render method is responsible for rendering the component's UI based on its props and state. It must be a pure function without any side effects.

3. `componentDidMount()`: This method is invoked immediately after the component is mounted to the DOM. It's commonly used for side effects like fetching data from APIs or subscribing to events.

Updating:

1. `render()`: The render method is called whenever the component's props or state change.

2. `componentDidUpdate(prevProps, prevState)`: This method is triggered after the component updates and is commonly used for side effects that depend on the previous props or state.

Unmounting:

1. `componentWillUnmount()`: This method is called right before the component is unmounted from the DOM. It's used for cleaning up any resources or event listeners created in `componentDidMount`.

Example of lifecycle methods:

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

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

  componentDidMount() {
    console.log('Component mounted');
  }

  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log('Count updated');
```

```
    }
  }

  componentWillUnmount() {
    console.log('Component unmounted');
  }

  handleClick() {
    this.setState(prevState => ({ count: prevState.count + 1 }));
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.handleClick()}>Increment</button>
      </div>
    );
  }
}

export default ExampleComponent;
```

In this example, the `ExampleComponent` class defines the lifecycle methods `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`. The component initializes the state in the constructor, updates it in the `handleClick` method, and renders the UI based on the state.

Note that with the introduction of React hooks, such as `useEffect`, many use cases for lifecycle methods can be handled in functional components as well.

Pros and Cons:

- Pros:

- Lifecycle methods provide hooks to perform actions at specific stages of a component's lifecycle.

- They allow for managing side effects and asynchronous operations.

- Cons:

- The legacy lifecycle methods (`componentWillMount`, `componentWillUpdate`, etc.) have been deprecated and may not be available in future versions of React.

- Lifecycle methods can make components more complex and harder to understand. The introduction of hooks provides a simpler alternative in many cases.

Dos and Don'ts:

- Do use lifecycle methods when you need to perform actions at specific stages of a component's lifecycle, such as data fetching or subscriptions.

- Do be aware of the deprecation of certain lifecycle methods and prefer newer alternatives like hooks.

- Don't rely solely on lifecycle methods for managing complex state or side effects. Consider using React hooks or external state management libraries for more advanced scenarios.

Prepared By :
Divyansh Singh (LinkedIn : rgndunes)