Episode-08 | Let’s Get Classy

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-08** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) How do you `create Nested Routes react-router-dom configuration` ?

> In React applications using react-router-dom, we can create nested routes by nesting our `<Route>` components inside each other within the route configuration. This allows you to define routes and components hierarchically, making it easier to manage the routing structure of your application.

💡 Here's a step-by-step guide on how to create nested routes using react-router-dom:

1 `Install react-router-dom if we haven't already`:

```
npm install react-router-dom
```

2 `Import the necessary components from react-router-dom in your application file`:

```
import { BrowserRouter as Router, Route, Switch } from 'react
-router-dom';
```

1. Defining our route hierarchy by nesting components within each other. Typically, this is done within a component that acts as a layout or container for the nested routes. For example, if we have a layout component called Layout:

```
import React from 'react';
import { Route } from 'react-router-dom';

// Import your nested route components
import Home from './Home';
import About from './About';

function Layout() {
  return (
    <div>
      <h1>My App</h1>
```

```
        <Route path="/home" component={Home} />
        <Route path="/about" component={About} />
      </div>
    );
}

export default Layout;
```

1. In our main application file, wrap our entire application with the Router component, and use the component to render only the first matching route:

```
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react
-router-dom';

// Import your Layout component that defines nested routes
import Layout from './Layout';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/" component={Layout} />
      </Switch>
    </Router>
  );
}

export default App;
```

Now we have a simple example of nested routes. In this case, the Layout component defines the `/home` and `/about` routes, and these nested routes can have their own components and nested routes as well. We can continue to nest routes further by adding more `<Route>` components inside the Home and About components to create a more complex routing structure. Remember that this is just a basic example, and we can customize our routing structure based on the requirements of our application. We can also use the exact prop on routes to ensure that only the exact path is matched if needed.

Q) Read about `createHashRouter` , `createMemoryRouter` from React Router docs.

1. `createHashRouter` - createHashRouter is part of the React Router library and provides routing capabilities for single-page applications (SPAs). It's commonly used for building client-side navigation within applications. Unlike traditional server-side routing, it uses the fragment identifier (hash) in the URL to manage and handle routes on the client side. This means that changes in the URL after the # symbol do not trigger a full page reload, making it suitable for SPAs.

To use createHashRouter, we typically import it from the React Router library and define our routes using Route components. Here's a basic example of how you might use it:

```
import { createHashRouter, Route } from 'react-router-dom';

const App = () => (
  <createHashRouter>
    <Route path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
  </createHashRouter>
);
```

1. `createMemoryRouter` - createMemoryRouter is another routing component provided by React Router. Unlike createHashRouter or BrowserRouter, createMemoryRouter is not associated with the browser's URL. Instead, it allows you to create an in-memory router for testing or other scenarios where you don't want to interact with the actual browser's URL.

💡 Here's a simple example of how to use createMemoryRouter:

```
import { createMemoryRouter, Route } from 'react-router-dom';

const App = () => (
  <createMemoryRouter>
    <Route path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
  </createMemoryRouter>
);
```

In both cases, we define our application's routes within the router component and specify the components to render for each route. The choice between `createHashRouter` and `createMemoryRouter` depends on our specific use case, such as whether we're building an SPA that interacts with the browser's URL or a scenario where we need an in-memory router for testing.

Q) What is the `order of life cycle method calls in Class Based Components` ?

`Constructor` - The constructor method is the first to be called when a component is created. It's where we typically initialize the component's state and bind event handlers.

`Render` - The render method is responsible for rendering the component's UI. It must return a React element (typically JSX) representing the component's structure.

`ComponentDidMount` - This method is called immediately after the component is inserted into the DOM. It's often used for making AJAX requests, setting up subscriptions, or other one-time initializations.

`ComponentDidUpdate` - This method is called after the component has been updated (re-rendered) due to changes in state or props. It's often used for side effects, like updating the DOM in response to state or prop changes.

`ComponentWillUnmount` - This method is called just before the component is removed from the DOM. It's used to clean up resources or perform any necessary cleanup.

For more reference React-Lifecycle-methods

Q) Why do we use `componentDidMount` ?

The `componentDidMount` lifecycle method in React class-based components is used for a specific purpose: it is called immediately after a component is inserted into the DOM (Document Object Model). This makes it a crucial point in the component's lifecycle and provides a valuable opportunity to perform various tasks that require interaction with the DOM or external data sources. Here are some common use cases for componentDidMount:

`Fetching Data` - It's often used to make asynchronous requests to fetch data from APIs or external sources. This is a common scenario for components that need to display dynamic content.

`DOM Manipulation` - When we need to interact with the DOM directly, such as selecting elements, setting attributes, or applying third-party libraries that require DOM elements to be present, we can safely do so in componentDidMount. This is because the component is guaranteed to be in the DOM at this point.

```
class MyComponent extends React.Component {
  componentDidMount() {
    // Fetch data from an API
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => {
        // Update the component's state with the fetched data
        this.setState({ data });
      })
      .catch(error => {
        // Handle any errors
        console.error(error);
      });
  }

  render() {
    // Render component based on state
    return (
      <div>{/* Display data from this.state.data */}</div>
    );
  }
}
```

By using `componentDidMount`, we can ensure that the data fetching or other side effects happen after the initial render and that our component interacts with the DOM or external data sources at the right time in the component's lifecycle.

Q) Why do we use `componentWillUnmount`? Show with example.

> The `componentWillUnmount` lifecycle method in React class-based components is used to perform cleanup and teardown tasks just before a component is removed from the DOM. It's a crucial part of managing resources and subscriptions to prevent memory leaks and ensure that the component's behavior is properly cleaned up. Here's why and when we should use componentWillUnmount:

1 `Cleanup Resources` - If your component has allocated any resources, such as event listeners, subscriptions, timers, or manual DOM manipulations, it's essential to release these resources to prevent memory leaks. componentWillUnmount is the appropriate place to do this.

2 `Cancel Pending Requests` - If your component has initiated any asynchronous requests, such as AJAX calls or timers, you should cancel or clean them up to avoid unexpected behavior after the component is unmounted.

Here's an example of using componentWillUnmount to remove an event listener when a component is unmounted:

```
class MyComponent extends React.Component {
  constructor() {
    super();
    this.handleResize = this.handleResize.bind(this);
  }

  componentDidMount() {
    // Add a window resize event listener when the component
is mounted
    window.addEventListener('resize', this.handleResize);
  }

  componentWillUnmount() {
    // Remove the window resize event listener when the compo
nent is unmounted
    window.removeEventListener('resize', this.handleResize);
  }

  handleResize(event) {
    // Handle the resize event
    console.log('Window resized:', event);
  }

  render() {
    return <div>My Component</div>;
  }
}
```

In this example, the component adds a resize event listener to the window when it's mounted, and it removes that listener in the componentWillUnmount method.

This ensures that the event listener is properly cleaned up when the component is unmounted, preventing memory leaks or unexpected behavior.

By using componentWillUnmount, we can ensure that any cleanup tasks are executed reliably when the component is no longer needed, helping to maintain the integrity of our application and avoiding potential issues.

Q) (Research) Why do we use `super(props)` in constructor?

> In JavaScript, when you define a class that extends another class (inherits from a parent class), we often use the super() method with props as an argument in the constructor of the child class. This is commonly seen in React when you create class-based components. The super(props) call is used for the following reasons:

`Access to Parent Class's Constructor` - When a child class extends a parent class, the child class can have its constructor. However, if the child class has a constructor, it must call super(props) as the first statement in its constructor. This is because super(props) is used to invoke the constructor of the parent class, ensuring that the parent class's initialization is performed before the child class's constructor code is executed. It is essential to maintain the inheritance chain correctly.

`Passing Props to the Parent Constructor` - By passing props to super(props), we ensure that the props object is correctly passed to the parent class's constructor. This is important because the parent class may need to set up its properties or handle the props somehow. By calling super(props), we make the props available for the parent class's constructor to work with.

💡 Here's an example of how super(props) is used in a React component:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props); // Call the constructor of the parent class
(React.Component)
    // Initialize your component's state or perform other set
up
  }

  render() {
    // Render the component based on its state and props
    return <div>{this.props.someProp}</div>;
  }
}
```

In this example, the super(props) call ensures that the `React.Component` class's constructor is called, which is necessary for React to set up the component correctly. This is especially important because React uses the props object to pass data from parent components to child components. By calling super(props), we make sure that the props are properly handled in the parent class's constructor, and we can access them in our child component.

In modern JavaScript and React, it's also common to define a constructor without explicitly calling super(props), and it will be automatically called for us. However, if we define a constructor in a child class, and the parent class has its constructor, it's a good practice to include super(props) to ensure that the parent class's constructor is invoked correctly.

Q) (Research) Why can't we have the `callback function` of `useEffect async`?

A: In React, the `useEffect` hook is designed to handle side effects in functional components. It's a powerful and flexible tool for managing asynchronous operations, such as data fetching, API calls, and more. However, useEffect itself cannot directly accept an async callback function. This is because useEffect expects its callback function to return either nothing (i.e., undefined) or a cleanup function, and it doesn't work well with Promises returned from async functions. There are a few reasons for this:

`Return Value Expectation` - The primary purpose of the useEffect callback function is to handle side effects and perform cleanup. React expects us to either return nothing (i.e., undefined) from the callback or return a cleanup function. An async function returns a Promise, and it doesn't fit well with this expected behavior.

`Execution Order and Timing` - With async functions, we might not have fine-grained control over the execution order of the asynchronous code and the cleanup code. React relies on the returned cleanup function to handle cleanup when the component is unmounted or when the dependencies specified in the useEffect dependency array change. If you return a Promise, React doesn't know when or how to handle cleanup.

To work with async operations within a useEffect, we can use the following pattern:

```
useEffect(() => {
  const fetchData = async () => {
    try {
      // Perform asynchronous operations
      const result = await someAsyncOperation();
      // Update the state with the result
      setState(result);
    } catch (error) {
      // Handle errors
      console.error(error);
    }
  };

  fetchData(); // Call the async function

  return () => {
    // Cleanup code, if necessary
    // This function will be called when the component unmoun
ts or when dependencies change
```

```
  };
}, [/* dependency array */]);
```

In this pattern, we define an async function within the useEffect callback, perform our asynchronous operations, and then call that function. Additionally, we return a cleanup function from the useEffect to handle any necessary cleanup tasks when the component unmounts or when specified dependencies change.

By using this approach, we can effectively manage asynchronous operations with useEffect while adhering to React's expectations for the callback function's return value.