Week 9.1

Custom Hooks

In this lecture, Harkirat presents a comprehensive guide to `Custom Hooks in React` The discussion begins by contrasting `class-based` and `functional component` approaches. It then delves into the rationale behind the advent of custom hooks, highlighting their role in maintaining cleaner code. The session concludes with `hands-on exploration` of diverse examples, providing practical insights into effectively `implementing custom hooks`.

> Note: Today's lecture notes are comprehensive, surpassing the depth of previous lectures. Consider them an all-inclusive resource for a thorough understanding and effective utilization of Custom Hooks in your React applications.

A Few Concepts Before We Begin

1] Ternary Operator

The ternary operator, also known as the conditional operator, is a concise way to write an `if-else` statement in a single line. It has the following syntax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

Here's how it works:

- The `condition` is evaluated. If it is `true`, the expression before the `:` (colon) is executed; otherwise, the expression after the `:` is executed.

Let's look at a simple example:

```
const isRaining = true;

const weatherMessage = isRaining ? "Bring an umbrella" : "E
njoy the sunshine";

console.log(weatherMessage);
```

In this example, if `isRaining` is `true`, the `weatherMessage` will be set to "Bring an umbrella"; otherwise, it will be set to "Enjoy the sunshine."

> The ternary operator is often used for simple conditional assignments, making the code more concise. However, it's important to use it judiciously, as overly complex ternary expressions can reduce code readability.

2] Lifecycle Events

Lifecycle events in React represent various phases a component goes through from its birth to its removal from the DOM. These events are associated with class components and provide developers with the ability to execute code at specific points during a component's existence.

> It is completely okay if you find this section challenging at the moment, as we delve into a more in-depth discussion in the latter part of today's notes. For now, aim to grasp an overview of the concepts introduced and the associated terminology.

The key lifecycle events in a class-based React component are:

1. `componentDidMount`: This method is called after a component has been rendered to the DOM. It is commonly used to perform initial setup, data fetching, or subscriptions.

```
componentDidMount() {
  // Perform setup or data fetching here
}
```

2. `componentDidUpdate`: This method is invoked immediately after an update occurs. It's useful for reacting to prop or state changes and performing additional actions.

```
componentDidUpdate(prevProps, prevState) {
  // Perform actions based on prop or state changes
}
```

3. `componentWillUnmount`: This method is called just before a component is removed from the DOM. It's suitable for cleanup tasks, such as removing event listeners or canceling subscriptions.

```
componentWillUnmount() {
  // Clean up (e.g., remove event listeners or cancel subsc
riptions)
}
```

With the introduction of React Hooks, functional components also gained lifecycle-like behavior through the `useEffect` hook. The equivalent hooks are:

1. `useEffect` **with an empty dependency array:** Equivalent to `componentDidMount`. Runs after the initial render.

```
useEffect(() => {
  // Code to run after initial render
}, []);
```

2. `useEffect` **with dependencies:** Equivalent to `componentDidUpdate`. Runs whenever the specified dependencies change.

```
useEffect(() => {
  // Code to run when dependencies change
}, [dependency1, dependency2]);
```

3. `useEffect` **with a cleanup function:** Equivalent to `componentWillUnmount`. Runs before the component is unmounted.

```
useEffect(() => {
  // Code to run on component mount

  return () => {
    // Cleanup code (similar to componentWillUnmount)
  };
}, []);
```

These lifecycle events are crucial for managing side effects, updating UI in response to changes, and maintaining clean-up procedures for optimal application performance.

> Well as emphasized before it is completely okay if you found the above section challenging at the moment, as we delve into a more in-depth discussion in the latter part of today's notes. For now, aim to grasp an overview of the concepts introduced and the associated terminology.

3] Understanding Debouncing

Debouncing is a programming practice used to ensure that time-consuming tasks do not fire so often, making them more efficient. In the context of `onInput` events, debouncing is often applied to delay the execution of certain actions (e.g., sending requests) until after a user has stopped typing for a specific duration.

Implementation:
The following example demonstrates debouncing in the `onInput` event to delay the execution of a function that sends a request based on user input.

```
<html>
  <body>
    <!-- Input field with onInput event and debouncing -->
    <input id="textInput" type="text" onInput="debounce(han
dleInput, 500)" placeholder="Type something...">

    <!-- Display area for the debounced input value -->
    <p id="displayText"></p>

    <script>
      // Debounce function to delay the execution of a func
tion
      function debounce(func, delay) {
        let timeoutId;

        return function() {
```

```
            // Clear the previous timeout
            clearTimeout(timeoutId);

            // Set a new timeout
            timeoutId = setTimeout(() => {
              func.apply(this, arguments);
            }, delay);
          };
        }

        // Function to handle the debounced onInput event
        function handleInput() {
          // Get the input field's value
          const inputValue = document.getElementById("textInp
ut").value;

          // Display the input value in the paragraph
          document.getElementById("displayText").innerText =
"You typed: " + inputValue;

          // Simulate sending a request (replace with actual
AJAX call)
          console.log("Request sent:", inputValue);
        }
      </script>
    </body>
</html>
```

Explanation:

- The `debounce` function is a generic debounce implementation that takes a function (`func`) and a delay time (`delay`).
- Inside the `debounce` function, a timeout is set to delay the execution of the provided function (`func`) by the specified delay time (`delay`).
- The `handleInput` function is the actual function to be executed when the `onInput` event occurs. It simulates sending a request (e.g., an AJAX call) based on user input.

How it works:

- When a user types in the input field, the `onInput` event triggers the `debounce` function.
- The `debounce` function sets a timeout, and if the user continues typing within the specified delay time, the previous timeout is cleared, and a new one is set.
- After the user stops typing for the specified delay, the `handleInput` function is executed.

> This ensures that the function associated with the `onInput` event is not called on every keystroke but rather after the user has stopped typing for a brief moment, reducing unnecessary and potentially resource-intensive calls, such as sending requests.

Class Components vs Functional Components

State Management

Let us take a look at the implementation of a simple counter component using both functional and class-based approaches in React, emphasizing state management.

Using Class Component

```
import React from 'react';

class MyComponent extends React.Component {
  // Constructor to initialize state
  constructor(props) {
```

```
    super(props);
    this.state = { count: 0 };
  }

  // Method to increment the count when the button is click
ed
  incrementCount = () => {
    // Updating the count state using this.setState
    this.setState({ count: this.state.count + 1 });
  }

  // Render method to display the current count and the "In
crement" button
  render() {
    return (
      <div>
        <p>{this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</bu
tton>
      </div>
    );
  }
}
```

1. In the class-based component, state is initialized in the constructor using `this.state`.
2. `this.state = { count: 0 };` initializes the `count` state variable with an initial value of 0.
3. The `incrementCount` method increases the `count` state by 1 when the "Increment" button is clicked.
4. The `render` method displays the current count and a button that triggers the `incrementCount` method.

Using Functional Component

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

function MyComponent() {
  // Using the useState hook to manage state in a functiona
l component
  const [count, setCount] = useState(0);

  // Function to increment the count when the button is cli
cked
  const incrementCount = () => {
    // Updating the count state using the setCount function
    setCount(count + 1);
  };

  // Rendering the component with the current count and an
"Increment" button
  return (
    <div>
      <p>{count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

1. The `useState` hook is used to declare state variables within functional components.
2. `const [count, setCount] = useState(0);` initializes the `count` state variable with an initial value of 0, and `setCount` is a function used to update the `count` state.
3. The `incrementCount` function increases the `count` state by 1 each time the "Increment" button is clicked.

4. The JSX returned by the component displays the current count and a button that triggers the `incrementCount` function.

> Both implementations achieve the same result, showcasing different approaches to managing state in React components. The functional component uses the `useState` hook for a more concise and modern syntax, while the class-based component follows the traditional class syntax for state management. The choice between them often depends on personal preference and the specific requirements of the application.

Lifecycle Events

Now, let's explore the implementation of how lifecycle events are handled in both class-based and functional components in React.

Using Class Component

```
import React from 'react';

class MyComponent extends React.Component {
  // componentDidMount: Invoked after the component is firs
t mounted to the DOM
  componentDidMount() {
    // Perform setup or data fetching here
    console.log('Component is mounted to the DOM');
  }

  // componentWillUnmount: Invoked just before the componen
```

```
t is unmounted and destroyed
  componentWillUnmount() {
    // Clean up (e.g., remove event listeners or cancel sub
scriptions)
    console.log('Component is about to be unmounted');
  }

  // render: Renders the UI of the component
  render() {
    return (
      // Render UI
      <div>
        Component Lifecycle Events (Class-Based)
      </div>
    );
  }
}
```

1. `componentDidMount`: Invoked after the component is first mounted to the DOM. It's a suitable place for initial setup or data fetching operations.
2. `componentWillUnmount`: Invoked just before the component is unmounted and destroyed. It's used for cleanup tasks, such as removing event listeners or canceling subscriptions.
3. `render`: The method responsible for rendering the UI of the component.

Using Functional Component

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

function MyComponent() {
  // useEffect: Invoked after the component is mounted and
reinvoked if dependencies change
  useEffect(() => {
    // Perform setup or data fetching here
```

```
    // Cleanup function (similar to componentWillUnmount)
    return () => {
      console.log('Component is about to be unmounted (clea
nup)');
      // Cleanup code goes here
    };
  }, []); // Empty dependency array ensures useEffect runs
only on mount and unmount

  // Render UI
  return (
    <div>
      Component Lifecycle Events (Functional)
    </div>
  );
}
```

1. `useEffect`: Invoked after the component is mounted and reinvoked if dependencies change. It can be used for both setup and cleanup.
2. The empty dependency array (`[]`) ensures that the `useEffect` runs only on mount and unmount, simulating the behavior of `componentDidMount` and `componentWillUnmount`.
3. The cleanup function within `useEffect` is invoked just before the component is unmounted.

> In functional components, the `useEffect` hook is a versatile replacement for various lifecycle events in class-based components. It handles both mounting and unmounting scenarios, offering a more concise and expressive way to manage side effects.

Significance of Returning a Component from useEffect

In the provided code snippet, the we utilize the `useEffect` hook along with the `setInterval` function to toggle the state of the `render` variable every 5 seconds. This, in turn, controls the rendering of the `MyComponent` or an empty `div` based on the value of `render`. Let's break down the significance of returning a component from `useEffect`:

```
import React, { useEffect, useState } from 'react';
import './App.css';

function App() {
  const [render, setRender] = useState(true);

  useEffect(() => {
    // Toggle the state every 5 seconds
    const intervalId = setInterval(() => {
      setRender(r => !r);
    }, 5000);

    // Cleanup function: Clear the interval when the compon
ent is unmounted
    return () => {
      clearInterval(intervalId);
    };
  }, []);

  return (
    <>
      {render ? <MyComponent /> : <div></div>}
    </>
  );
}
```

```
function MyComponent() {
  useEffect(() => {
    console.error("Component mounted");

    // Cleanup function: Log when the component is unmounte
d
    return () => {
      console.log("Component unmounted");
    };
  }, []);

  return <div>
    From inside MyComponent
  </div>;
}

export default App;
```

Understanding the Code

- The `useEffect` hook is used to create a side effect (in this case, toggling the `render` state at intervals) when the component mounts.
- A cleanup function is returned within the `useEffect`, which will be executed when the component is unmounted. In this example, it clears the interval previously set by `setInterval`.
- By toggling the `render` state, the component (`MyComponent` or an empty `div`) is conditionally rendered or unrendered, demonstrating the dynamic nature of component rendering.
- The `return` statement within the `useEffect` of `MyComponent` is used to specify what should be rendered when the component is active, in this case, a simple `div` with the text "From inside MyComponent."

> In summary, the ability to return a cleanup function from `useEffect` is crucial for managing resources, subscriptions, or

> intervals created during the component's lifecycle. It helps ensure proper cleanup when the component is no longer in use, preventing memory leaks or unintended behavior.

React Hooks

React Hooks are functions that allow functional components in React to have state and lifecycle features that were previously available only in class components. Hooks were introduced in React 16.8 to enable developers to use state and other React features without writing a class.

> Using these hooks, developers can manage state, handle side effects, optimize performance, and create more reusable and readable functional components in React applications. Each hook serves a specific purpose, contributing to a more modular and maintainable codebase.

In previous lectures, specifically Week 6 —we have already covered in depth the most commonly used hooks provided to us by React: `useEffect`, `useMemo`, `useCallback`, `useRef`, `useReducer`, `useContext`, `useLayoutEffect`

Custom Hooks

What?

Custom Hooks in React are user-defined functions that encapsulate reusable logic and stateful behavior. They allow developers to extract and share common functionality across multiple components, promoting code reusability and maintaining cleaner and more modular code.

Why?

The need for custom hooks arises from the desire to avoid code duplication and create a clean separation of concerns. By encapsulating specific logic in a custom hook, you can isolate and organize the functionality related to a particular concern or feature. This not only makes your codebase more maintainable but also facilitates easier testing and debugging.

How?

Custom hooks solve the problem of sharing logic between components without the need for higher-order components or render props. They provide a mechanism to encapsulate complex behavior, making it easier to reason about and reuse across different parts of your application. Ultimately, custom hooks contribute to a more efficient and scalable React codebase.

Examples

1. Data fetching hooks
2. Browser functionality related hooks - **useOnlineStatus , useWindowSize, useMousePosition**
3. Performance/Timer based - **useInterval, useDebounce**

Use Cases of Custom Hooks

1] Data Fetching Hooks

Data fetching hooks can be used to encapsulate all the logic to fetch the data from your backend

Now below is how our code looks before using custom hooks

```
import { useEffect, useState } from 'react'
import axios from 'axios'

function App() {
  const [todos, setTodos] = useState([])
```

```
  useEffect(() => {
    axios.get("https://sum-server.100xdevs.com/todos")
      .then(res => {
        setTodos(res.data.todos);
      })
  }, [])

  return (
    <>
      {todos.map(todo => <Track todo={todo} />)}
    </>
  )
}

function Track({ todo }) {
  return <div>
    {todo.title}
    <br />
    {todo.description}
  </div>
}

export default App
```

We will now see a step by step guide on how you can use custom hook for the above use case.

Step 1 - Converting the `data fetching` bit to a custom hook

```
import { useEffect, useState } from 'react';
import axios from 'axios';

// Custom hook for fetching todos
function useTodos() {
  const [todos, setTodos] = useState([]);
```

```
  useEffect(() => {
    // Fetching todos using Axios
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
      })
  }, []);

  // Return the todos state
  return todos;
}

// Main App component
function App() {
  // Using the custom hook to fetch todos
  const todos = useTodos();

  return (
    <>
      {/* Rendering Track component for each todo */}
      {todos.map(todo => <Track key={todo.id} todo={todo} /
>)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;
```

Explanation:

1. **Custom Hook (`useTodos`):**
   - It encapsulates the data fetching logic using `axios` within a custom hook.
   - Utilizes the `useState` and `useEffect` hooks to manage state and perform side effects respectively.
   - Returns the `todos` state, making it accessible in the component that uses this custom hook.
2. **Main App Component:**
   - Imports and uses the `useTodos` custom hook, fetching todos and storing them in the `todos` variable.
   - Maps over the `todos` array, rendering the `Track` component for each todo.
3. **Track Component:**
   - Receives an individual `todo` as a prop and renders its `title` and `description`.

> By creating a custom hook (`useTodos`), the data fetching logic is abstracted and can be easily reused across different components. This promotes a cleaner and more modular code structure.

Step 2 - Cleaning the hook to include a `loading` parameter

What if you want to show a loader when the data is not yet fetched from the backend?

```
import { useEffect, useState } from 'react';
import axios from 'axios';
```

```
// Custom hook for fetching todos with loading indicator
function useTodos() {
  const [loading, setLoading] = useState(true);
  const [todos, setTodos] = useState([]);

  useEffect(() => {
    // Fetching todos using Axios
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching todos:", error);
        setLoading(false);
      });
  }, []);

  // Return todos and loading state
  return {
    todos: todos,
    loading: loading
  };
}

// Main App component
function App() {
  // Using the custom hook to fetch todos
  const { todos, loading } = useTodos();

  // Rendering loading message if data is still loading
  if (loading) {
    return <div>Loading...</div>;
  }

  // Rendering Track component for each todo
  return (
    <>
```

```
      {todos.map(todo => <Track key={todo.id} todo={todo} />)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;
```

Explanation:

1. **Custom Hook (`useTodos`):**
   - Introduces a `loading` state to track whether the data is still being fetched.
   - The `setLoading(false)` is placed in both the successful and error scenarios to handle loading completion.
   - Returns an object with both `todos` and `loading` states.
2. **Main App Component:**
   - Destructures the result from the custom hook, including `todos` and `loading`.
   - If `loading` is `true`, it renders a loading message. Otherwise, it maps over the `todos` array and renders the `Track` component.
3. **Track Component:**
   - Remains the same, rendering individual todos.

> By including a loading parameter in the custom hook, you can provide better user experience by displaying a loading message while the data is being fetched.

Step 3 - Auto refreshing hook

What if you want to keep polling the backend every n seconds? `n` needs to be passed in as an input to the hook

```
import { useEffect, useState } from 'react';
import axios from 'axios';

// Custom hook for fetching todos with auto-refresh
function useTodos(n) {
  const [loading, setLoading] = useState(true);
  const [todos, setTodos] = useState([]);

  // Function to fetch data from the backend
  function getData() {
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching todos:", error);
        setLoading(false);
      });
  }

  useEffect(() => {
    // Initial data fetch
    getData();

    // Set up interval to fetch data every n seconds
    const intervalId = setInterval(() => {
      getData();
```

```
    }, n * 1000);

    // Clean up the interval on component unmount or when n
changes
    return () => clearInterval(intervalId);
  }, [n]);

  // Return todos and loading state
  return {
    todos: todos,
    loading: loading
  };
}

// Main App component
function App() {
  // Using the custom hook to fetch todos with auto-refresh
every 5 seconds
  const { todos, loading } = useTodos(5);

  // Rendering loading message if data is still loading
  if (loading) {
    return <div>Loading...</div>;
  }

  // Rendering Track component for each todo
  return (
    <>
      {todos.map(todo => <Track key={todo.id} todo={todo} /
>)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
```

```
        {todo.title}
        <br />
        {todo.description}
      </div>
    );
}

export default App;
```

Explanation:

1. **Custom Hook (`useTodos`):**
   - Accepts an input `n` representing the interval in seconds for auto-refresh.
   - Utilizes the `getData` function to fetch data from the backend.
   - In the `useEffect`, sets up an interval to call `getData` every `n` seconds.
   - Cleans up the interval when the component unmounts or when `n` changes.
2. **Main App Component:**
   - Utilizes the custom hook with an interval of 5 seconds (`useTodos(5)`).
   - Renders a loading message if data is still loading and maps over `todos` otherwise.
3. **Track Component:**
   - Remains the same, rendering individual todos.

> This step enhances the hook by adding an auto-refresh feature, ensuring the data is periodically fetched from the backend.

Step 4 - We Clear the Interval

```
import { useEffect, useState } from 'react';
import axios from 'axios';

// Custom hook for fetching todos with auto-refresh
function useTodos(n) {
  const [todos, setTodos] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Set up interval to fetch data every n seconds
    const intervalId = setInterval(() => {
      axios.get("<https://sum-server.100xdevs.com/todos>")
        .then(res => {
          setTodos(res.data.todos);
          setLoading(false);
        })
        .catch(error => {
          console.error("Error fetching todos:", error);
          setLoading(false);
        });
    }, n * 1000);

    // Initial data fetch
    axios.get("<https://sum-server.100xdevs.com/todos>")
      .then(res => {
        setTodos(res.data.todos);
        setLoading(false);
      })
      .catch(error => {
        console.error("Error fetching todos:", error);
        setLoading(false);
      });

    // Clean up the interval on component unmount or when n
changes
    return () => clearInterval(intervalId);
  }, [n]);
```

```
  // Return todos and loading state
  return { todos, loading };
}

// Main App component
function App() {
  // Using the custom hook to fetch todos with auto-refresh
every 10 seconds
  const { todos, loading } = useTodos(10);

  // Rendering loading message if data is still loading
  if (loading) {
    return <div>Loading...</div>;
  }

  // Rendering Track component for each todo
  return (
    <>
      {todos.map(todo => <Track key={todo.id} todo={todo} /
>)}
    </>
  );
}

// Track component for rendering individual todo
function Track({ todo }) {
  return (
    <div>
      {todo.title}
      <br />
      {todo.description}
    </div>
  );
}

export default App;
```

Explanation:

1. **Custom Hook (`useTodos`):**
    - Continues to have the auto-refresh functionality with a 10-second interval (`useTodos(10)`).
    - Clears the interval using `clearInterval` in the cleanup function returned by `useEffect`. This ensures that the interval is cleared when the component unmounts or when `n` changes.
2. **Main App Component:**
    - Remains the same, utilizing the custom hook and rendering todos.
3. **Track Component:**
    - Unchanged, rendering individual todos.

> This step enhances the hook by adding a cleanup mechanism to clear the interval when it's no longer needed, preventing potential memory leaks.

SWR Library

The `swr` library is a powerful tool for data fetching in React applications. It simplifies the process of handling data fetching, caching, and re-fetching when needed. Here's an explanation of the provided code snippet:

```
// Import the useSWR hook from the 'swr' library
import useSWR from 'swr';

// Define a fetcher function to handle data fetching
const fetcher = async function(url) {
  // Fetch data from the specified URL
  const data = await fetch(url);

  // Parse the response as JSON
  const json = await data.json();
```

```
  // Return the parsed JSON data
  return json;
};

// Example component using the useSWR hook
function Profile() {
  // Use the useSWR hook to fetch data from the specified U
RL
  const { data, error, isLoading } = useSWR('<https://sum-s
erver.100xdevs.com/todos>', fetcher);

  // Handle different states: loading, error, and successfu
l data fetch
  if (error) return <div>Failed to load</div>;
  if (isLoading) return <div>Loading...</div>;

  // Render the component with the fetched data
  return <div>Hello, you have {data.todos.length} todos!</d
iv>;
}
```

Explanation:

1. **Importing `useSWR`:**
    - The `useSWR` hook is imported from the 'swr' library. This hook simplifies data fetching by providing caching and re-fetching capabilities.
2. **Fetcher Function:**
    - A `fetcher` function is defined to handle data fetching. It uses the `fetch` API to retrieve data from the specified URL, parses the response as JSON, and returns the parsed data.
3. **Usage in `Profile` Component:**
    - The `useSWR` hook is used in the `Profile` component to fetch data from the specified URL (`https://sum-server.100xdevs.com/todos`). The `fetcher` function is provided as the second argument to `useSWR`.
4. **Handling Different States:**

- The component checks for different states: `error`, `isLoading`, and successful data fetch. Depending on the state, it renders appropriate content (error message, loading indicator, or the fetched data).

5. **Rendering Component:**
   - If the data is successfully fetched, the component renders a message indicating the number of todos.

> Using `swr` can significantly simplify data fetching in React applications, providing a clean and efficient way to manage remote data. https://swr.vercel.app/

2] Browser Functionality Related Hooks

useOnlineStatus

The Custom React Hook — `useIsOnline` determines whether the user is currently online or offline. It utilizes the `window.navigator.onLine` property and event listeners to keep track of the online status. Here's a detailed explanation:

1. `useIsOnline` Hook:

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

function useIsOnline() {
  // Initialize state with the current online status
  const [isOnline, setIsOnline] = useState(window.navigato
r.onLine);

  useEffect(() => {
    // Add event listeners to track online/offline changes
    const handleOnline = () => setIsOnline(true);
    const handleOffline = () => setIsOnline(false);
```

```
    // Attach event listeners to the 'online' and 'offline'
events
    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);

    // Cleanup: Remove event listeners on component unmount
    return () => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
    };
  }, []);

  // Return the current online status
  return isOnline;
}
```

Explanation:

- **Initialization:** The `isOnline` state variable is initialized with the current value of `window.navigator.onLine`. This represents the initial online status.
- **Effect Hook:** The `useEffect` hook is used to add event listeners for the 'online' and 'offline' events when the component mounts. These listeners update the `isOnline` state accordingly.
- **Event Listeners:** Two event listeners, `handleOnline` and `handleOffline`, are defined to update the `isOnline` state based on the user's online or offline status.
- **Cleanup:** The `useEffect` hook also returns a cleanup function. This function removes the event listeners when the component is unmounted, preventing memory leaks.

2. `App` Component:

```
function App() {
  // Use the custom hook to get the current online status
  const isOnline = useIsOnline();
```

```
  // Render different messages based on the online status
  return (
    <>
      {isOnline ? "You are online, yay!" : "You are not onl
ine"}
    </>
  );
}

export default App;
```

Explanation:

- The `App` component uses the `useIsOnline` hook to determine the current online status.
- Based on the online status, it renders different messages to inform the user whether they are online or offline.

> This custom hook provides a reusable way to track the user's online status throughout the lifecycle of a React component.

useMousePosition

The Custom React hook — `useMousePointer` allows tracking the current position of the mouse pointer. It utilizes the `window.addEventListener` method with the 'mousemove' event to update the mouse position. Here's a detailed explanation:

1. `useMousePointer` Hook:

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

const useMousePointer = () => {
  // Initialize state with the initial mouse position (0,
```

```
0)
  const [position, setPosition] = useState({ x: 0, y: 0 });

  // Event handler to update the mouse position on mouse mo
vement
  const handleMouseMove = (e) => {
    setPosition({ x: e.clientX, y: e.clientY });
  };

  useEffect(() => {
    // Add event listener for 'mousemove' event when the co
mponent mounts
    window.addEventListener('mousemove', handleMouseMove);

    // Cleanup: Remove event listener on component unmount
    return () => {
      window.removeEventListener('mousemove', handleMouseMo
ve);
    };
  }, []);

  // Return the current mouse position
  return position;
};
```

Explanation:

- **Initialization:** The `position` state variable is initialized with the initial mouse position (`{ x: 0, y: 0 }`).
- **Event Handler:** The `handleMouseMove` function is defined to update the `position` state with the current mouse coordinates (`e.clientX` and `e.clientY`) when the mouse is moved.
- **Effect Hook:** The `useEffect` hook is used to add an event listener for the 'mousemove' event when the component mounts. This listener triggers the `handleMouseMove` function on mouse movement.

- **Cleanup:** The `useEffect` hook returns a cleanup function that removes the 'mousemove' event listener when the component is unmounted, preventing memory leaks.

2. `App` Component:

```
function App() {
  // Use the custom hook to get the current mouse position
  const mousePointer = useMousePointer();

  // Render a message displaying the current mouse position
  return (
    <>
      Your mouse position is {mousePointer.x} {mousePointe
r.y}
    </>
  );
}

export default App;
```

Explanation:

- The `App` component utilizes the `useMousePointer` hook to obtain the current mouse position.
- It renders a message displaying the x and y coordinates of the mouse position in real-time.

> This custom hook provides an easy way to track and utilize the mouse pointer position within a React component.

3] Performance/Timer Based

useInterval

The Custom React Hook — `useInterval` facilitates running a callback function at specified intervals. This hook is then utilized in the `App` component to increment a timer every second. Here's an in-depth explanation:

1. `useInterval` Hook:

```
import { useEffect } from 'react';

const useInterval = (callback, delay) => {
  useEffect(() => {
    // Set up an interval and store the interval ID
    const intervalId = setInterval(callback, delay);

    // Cleanup: Clear the interval when the component is un
mounted
    return () => clearInterval(intervalId);
  }, [callback, delay]);
};
```

Explanation:

- **Function Signature:** The `useInterval` hook takes two parameters - `callback` (the function to be executed) and `delay` (the interval in milliseconds).
- **Effect Hook:** Inside the `useEffect` hook, `setInterval` is used to repeatedly call the `callback` function at the specified `delay`.
- **Cleanup:** The returned cleanup function ensures that the interval is cleared when the component using this hook is unmounted, preventing memory leaks.

2. `App` Component:

```
import { useState } from 'react';
import useInterval from './useInterval';  // Import the cus
tom useInterval hook

function App() {
  // State to store the count value
  const [count, setCount] = useState(0);

  // Utilize the useInterval hook to increment the count ev
ery second
  useInterval(() => {
    setCount(c => c + 1);
  }, 1000);

  // Render the current count value
  return (
    <>
      Timer is at {count}
    </>
  );
}

export default App;
```

Explanation:

- The `App` component utilizes the `useInterval` hook to increment the `count` state value every second.
- The rendered output displays the current value of the timer, which increases every second.

> This custom hook simplifies the implementation of interval-based functionality in React components, providing a reusable and clean solution.

useDebounce

The Custom React Hook — `useDebounce` is utilized in a `SearchBar` component to debounce the user input, making it ideal for scenarios such as live search functionality. Below is a detailed explanation:

1. `useDebounce` Hook:

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

const useDebounce = (value, delay) => {
  // State to store the debounced value
  const [debouncedValue, setDebouncedValue] = useState(valu
e);

  useEffect(() => {
    // Set up a timer to update the debounced value after t
he specified delay
    const timerId = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    // Clean up the timer if the value changes before the d
elay has passed
    return () => clearTimeout(timerId);
  }, [value, delay]);

  return debouncedValue;
};
```

Explanation:

- **Function Signature:** The `useDebounce` hook takes two parameters - `value` (the input value to be debounced) and `delay` (the debounce delay in milliseconds).

- **State:** The `debouncedValue` state holds the debounced value.
- **Effect Hook:** Inside the `useEffect` hook, a timer is set using `setTimeout`. This timer updates the `debouncedValue` with the current input value after the specified delay.
- **Cleanup:** The `clearTimeout` function is used for cleanup to ensure that the timer is cleared if the input value changes before the delay has passed.
- **Dependencies:** The effect hook depends on the `value` and `delay` parameters, ensuring the effect is re-run when they change.

2. `SearchBar` Component:

```
import React, { useState } from 'react';
import useDebounce from './useDebounce';

const SearchBar = () => {
  // State to manage the user input
  const [inputValue, setInputValue] = useState('');

  // Use the useDebounce hook to get the debounced value
  const debouncedValue = useDebounce(inputValue, 500); // 5
00 milliseconds debounce delay

  // Integrate the debouncedValue in your component logic
(e.g., trigger a search API call via a useEffect)

  return (
    <input
      type="text"
      value={inputValue}
      onChange={(e) => setInputValue(e.target.value)}
      placeholder="Search..."
    />
  );
};

export default SearchBar;
```

Explanation:

- The `SearchBar` component uses the `useDebounce` hook to get the debounced value of the user input (`inputValue`).
- The `onChange` handler updates the `inputValue` state as the user types.
- The debounced value (`debouncedValue`) can be further integrated into the component logic, such as triggering a search API call via a `useEffect`.

> This custom hook allows for efficient handling of debounced values, reducing the number of API calls or other expensive operations triggered by frequent user input changes.