REACT State and Hooks Review

Working with Events in React

- **Synthetic Event System**: This is React's way of handling events. It serves as a wrapper around the native events like the click, keydown, and submit events. Event handlers in React use the camel casing naming convention. (Ex. onClick, onSubmit, etc)

Here is an example of using the onClick attribute for a button element in React:

Example Code

```
function handleClick() {
  console.log("Button clicked!");
}


<button onClick={handleClick}>Click Me</button>;
```

In React, event handler functions usually start with the prefix handle to indicate they are responsible for handling events, like handleClick or handleSubmit.

When a user action triggers an event, React passes a Synthetic Event object to your handler. This object behaves much like the native Event object in vanilla JavaScript, providing properties like type, target, and currentTarget.

To prevent default behaviors like browser refresh during an onSubmit event, for example, you can call the preventDefault() method:

Example Code

```
function handleSubmit(event) {
  event.preventDefault();
  console.log("Form submitted!");
}


<form onSubmit={handleSubmit}>
```

```
  <input type="text" />
  <button>Submit</button>
</form>;
```

You can also wrap a handler function in an arrow function like this:

Example Code

```
function handleDelete(id) {
  console.log("Deleting item:", id);
}


<button onClick={() => handleDelete(1)}>Delete Item</button>;
```

Working with State and the useState Hook

- **Definition for state**: In React, state is a object that contains data for a component. When the state updates the component will re-render. React treats state as immutable, meaning you should not modify it directly.
- **useState() Hook**: The useState hook is a function that lets you declare state variables in functional components. Here is an basic syntax:

Example Code

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

In the state variable you have the following:

- stateVariable holds the current state value
- setStateFunction (the setter function) updates the state variable
- initialValue sets the initial state

Here is a complete example for a Counter component:

Example Code

```
import { useState } from "react";


function Counter() {
 const [count, setCount] = useState(0);


 return (
  <div>
   <h2>{count}</h2>


   <button onClick={() => setCount(count - 1)}>Decrement</button>
   <button onClick={() => setCount(count + 1)}>Increment</button>
  </div>
 );
}


export default Counter;
```

Rendering and React Components

- **Definition**: In React, rendering is the process by which components appear in the user interface (UI), usually the browser. The rendering process consists of three stages: trigger, render, and commit.

The trigger stage occurs when React detects that something has changed and the user interface (UI) might need to be updated. This change is often due to an update in state or props.

Once the trigger happens, React enters the render stage. Here, React re-evaluates your components and figures out what to display. To do this, React uses a lightweight copy of the "real" DOM called the virtual DOM. With the virtual DOM, React can quickly check what needs to change in the component.

The commit stage is where React takes the prepared changes from the virtual DOM and applies them to the real DOM. In other words, this is the stage where you see the final result on the screen.

Updating Objects and Arrays in State

- **Updating Objects in State**: If you need to update an object in state, then you should make a new object or copy an existing object first, then set the state for that new object. Any object put into state should be considered as read-only. Here is an example of setting a user's name, age and city. The handleChange function is used to handle updates to the user's information:

Example Code

```
import { useState } from "react";


function Profile() {
 const [user, setUser] = useState({ name: "John Doe", age: 31, city: "LA" });


 const handleChange = (e) => {
  const { name, value } = e.target;


  setUser((prevUser) => ({...prevUser, [name]: value}));
 };


 return (
  <div>
   <h1>User Profile</h1>
   <p>Name: {user.name}</p>
   <p>Age: {user.age}</p>
   <p>City: {user.city}</p>


   <h2>Update User Age </h2>
   <input type="number" name="age" value={user.age} onChange={handleChange} />
```

```
    <h2>Update User Name </h2>

    <input type="text" name="name" value={user.name} onChange={handleChange} />


    <h2>Update User City </h2>

    <input type="text" name="city" value={user.city} onChange={handleChange} />

   </div>

  );

}


export default Profile;
```

- **Updating Arrays in State**: When updating arrays in state, it is important not to directly modify the array using methods like push() or pop(). Instead you should create a new array when updating state:

Example Code

```
const addItem = () => {

  const newItem = {

   id: items.length + 1,

   name: `Item ${items.length + 1}`,

  };


  // Creates a new array

  setItems((prevItems) => [...prevItems, newItem]);

};
```

If you want to remove items from an array, you should use the filter() method, which returns a new array after filtering out whatever you want to remove:

Example Code

```
const removeItem = (id) => {
 setItems((prevItems) => prevItems.filter((item) => item.id !== id));
};
```

Referencing Values Using Refs

- **ref Attribute**: You can access a DOM node in React by using the ref attribute. Here is an example to showcase a ref to focus an input element. The current property is used to access the current value of that ref:

Example Code

```
import { useRef } from "react";


const Focus = () => {
 const inputRef = useRef(null);


 const handleFocus = () => {
  if (inputRef.current) {
   inputRef.current.focus();
  }
 };


 return (
  <div>
   <input ref={inputRef} type="text" placeholder="Enter text" />
   <button onClick={handleFocus}>Focus Input</button>
  </div>
 );
};


export default Focus;
```

Working with the useEffect Hook

- **useEffect() Hook**: In React, an effect is anything that happens outside the component rendering process. That is, anything React does not handle directly as part of rendering the UI. Common examples include fetching data, updating the browser tab's title, reading from or writing to the browser's local storage, getting the user's location, and much more. These operations interact with the outside world and are known as side effects. React provides the useEffect hook to let you handle those side effects. useEffect lets you run a function after the component renders or updates.

Example Code

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


useEffect(() => {
 // Your side effect logic (usually a function) goes here
}, [dependencies]);
```

The effect function runs after the component renders, while the optional dependencies argument controls when the effect runs.

Note that dependencies can be an array of "reactive values" (state, props, functions, variables, and so on), an empty array, or omitted entirely. Here's how all of those options control how useEffect works:

- If dependencies is an array that includes one or more reactive values, the effect will run whenever they change.
- If dependencies is an empty array, useEffect runs only once when the component first renders.
- If you omit dependencies, the effect runs every time the component renders or updates.

How to Create Custom Hooks

- **Custom Hooks**: A custom hook allows you to extract reusable logic from components, such as data fetching, state management, toggling, and side effects like tracking online status. In React, all built-in hooks start with the word use, so your custom hook should follow the same convention.

Here is an example of creating a useDebounce hook:

Example Code

```
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

export { useDebounce };
```