**`useContext`:**

- Passes data directly to components deep in the tree without passing it through intermediate components, avoiding props drilling.

- Manage shared state data in a single centralized location, making it easier to update and maintain.

- Makes your components more concise and focused by sharing data.

**`useState`** relates to component lifecycles:

- **Initialization:** When a component first renders, `useState` initializes the state variable with the value you provide. This is similar to how state is initialized in the constructor of a class component.

- **Re-renders:** When the state variable is updated using the state updater function (e.g., `setData` in the `useFetch` example), React re-renders the component. This is crucial for keeping the UI in sync with the latest state.

- **Persistence:** Unlike regular variables in a function that are re-created on every render, state variables persist across re-renders. This allows components to "remember" their previous state

**`useEffect`** `is` closely related to lifecycle methods of ***class components (only class components have lifecycle methods)***. They provide a way to manage side effects and handle logic that was previously managed by methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

Here's how `useEffect` relates to component lifecycles:

- **`componentDidMount`:** Code within `useEffect` runs after the component first renders, similar to `componentDidMount`. This is useful for setting up subscriptions, fetching data, or manipulating the DOM.

- **`componentDidUpdate`:** By specifying dependencies in the `useEffect` dependency array, you can control when the effect runs. This allows you to perform actions when specific props or state variables change, similar to `componentDidUpdate`.

- **`componentWillUnmount`:** You can return a cleanup function from `useEffect`. This function will be executed when the component unmounts, allowing you to unsubscribe from events, clear timers, or perform other cleanup tasks, much like `componentWillUnmount`.

**Differences:**

- **Functional components:** Hooks are designed to be used in functional components, while lifecycle methods are specific to class components.
- **More flexible:** `useEffect` can be used multiple times within a single component to handle different side effects, whereas you were limited to a single instance of each lifecycle method in a class component.
- **Dependency array:** The dependency array in `useEffect` gives you fine-grained control over when the effect runs, making it more efficient and preventing unnecessary re-renders.

**CODE FROM THE CUSTOM HOOKS LAB EXERCISE:**

**`useSearch` custom hook:**

**Note: the search code currently searches all data for each row. To only search by name:**

```
  useEffect(() => {
    if (!query) {
      setFilteredData(data);
    } else {
      const lowercasedQuery = query.toLowerCase();
      const filtered = data.filter(item =>
        item.name.toLowerCase().includes(lowercasedQuery)
// Only check the name property
      );
      setFilteredData(filtered);
    }
  }, [data, query]);

  return filteredData;
};
```

### `useState` in the Search functionality:

### Manages and Updates Search Results by:

- **Storing the filtered data:** `useState` provides a way to store the search results in the `filteredData` state variable. This allows the component to keep track of the filtered data and display it to the user.
- **Triggering re-renders:** When the `query` or `data` changes, the `useEffect` hook updates the `filteredData` state using `setFilteredData`. This triggers a re-render of the component, ensuring that the displayed results always reflect the latest search query and data.

### Improves Performance:

- **Efficient updates:** By using `useState` and `useEffect`, you ensure that the component only re-renders when necessary (i.e., when `query` or `data` changes). This helps to optimize performance, especially when dealing with large datasets.

### Code Organization and Readability:

- **Encapsulation:** `useState` helps to encapsulate the search logic within the `useSearch` hook, making the code more organized and easier to understand.
- **Separation of concerns:** The use of `useState` and `useEffect` separates the concerns of data fetching, filtering, and state management, making the code more modular and maintainable.

`useState` in this `useSearch` hook provides a clean and efficient way to manage and update the filtered data based on the search query, while also improving performance and code organization.

**`useEffect` in the search:**

**Managing Side Effects:**

- **Performing actions after render:** `useEffect` allows you to perform side effects (actions that are not directly related to rendering UI) after the component renders. In this case, the side effect is filtering the `data` based on the `query`.

- **Synchronization with state/prop changes:** The dependency array `[data, query]` ensures that the filtering logic within `useEffect` runs whenever `data` or `query` changes. This keeps the `filteredData` synchronized with the latest inputs.

**2. Optimizing Performance:**

- **Preventing unnecessary filtering:** By using the dependency array, `useEffect` prevents the filtering logic from running unnecessarily on every render. It only executes when `data` or `query` actually change.
- **Avoiding infinite loops:** If the filtering logic was placed directly in the component body, it could lead to an infinite loop of re-renders as `filteredData` updates would trigger further re-renders. `useEffect` prevents this by controlling when the filtering happens.

**3. Improving Code Structure:**

- **Separating concerns:** `useEffect` separates the filtering logic from the main component logic, making the code cleaner and easier to understand.
- **Readability:** It clearly indicates that the filtering is a side effect that depends on `data` and `query`.

`useEffect` in this hook ensures that the filtering logic is executed efficiently and at the right times, responding to changes in `data` and `query` while preventing performance issues and improving code organization.

**Hooks in the `useFetch` custom hook:**

**1. `useState`**

- **`data`:**

    - **Storing fetched data:** `useState([])` initializes a state variable `data` to hold the fetched data from the API. This allows the component using this hook to access and display the data.
    - **Triggering re-renders:** When the fetched data is available, `setData(result)` updates the `data` state, causing the component to re-render and display the new data.

- **`loading`:**

    - **Managing loading state:** `useState(true)` initializes a state variable `loading` to `true`, indicating that the data is being fetched. This allows you to display a loading indicator while waiting for the data.
    - **Improving user experience:** By updating `setLoading(false)` after the fetch operation (whether successful or not), you provide feedback to the user that the loading process is complete.

- **`error`:**

    - **Handling errors:** `useState(null)` initializes a state variable `error` to store any errors that occur during the fetch process.
    - **Providing error feedback:** If an error occurs, `setError(error)` updates the `error` state, allowing you to display an error message to the user.

**2. `useEffect`**

- **Fetching data after render:** `useEffect` ensures that the `fetchData` function is called after the component renders, initiating the data fetching process.
- **Dependency array `[url]`:** The dependency array `[url]` ensures that the effect runs only when the `url` prop changes. This prevents unnecessary fetches if the component re-renders for other reasons.
- **Cleaning up:** If the `url` changes before the previous fetch completes, `useEffect` automatically aborts the previous fetch, preventing potential race conditions and memory leaks. (Although this example doesn't explicitly show an abort mechanism, `useEffect` provides the mechanism to implement one if needed, like using an AbortController).

Using `useState` with `useEffect` in this `useFetch` hook makes it efficient to:

- Fetch data from an API.
- Manage loading and error states.
- Update the component with the fetched data.

- Optimize performance by only fetching when necessary.
- Handle potential race conditions and cleanup.