1. React Custom Hook (`useCustomHook`)

What is a Custom Hook?

A **custom hook** in React is a JavaScript function that starts with `use` and allows you to reuse stateful logic between components. It helps in keeping components clean and organized.

Why Use Custom Hooks?

- Reuse code logic across components.
- Make code cleaner and more readable.
- Share stateful logic without repeating code.

Example: Creating a Custom Hook (`useCounter`)
This custom hook will provide a simple counter with `increment`, `decrement`, and `reset` functions.

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

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

  const increment = () => setCount((prev) => prev + 1);
  const decrement = () => setCount((prev) => prev - 1);
  const reset = () => setCount(initialValue);

  return { count, increment, decrement, reset };
}

export default useCounter;
```

—---------------------------------------

```
import React from 'react';
import useCounter from './useCounter';

function CounterComponent() {
  const { count, increment, decrement, reset } = useCounter(0);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={increment}>Increment ➕</button>
      <button onClick={decrement}>Decrement ➖</button>
      <button onClick={reset}>Reset 🔄</button>
    </div>
  );
}

export default CounterComponent;
```

Key Points:

- The hook `useCounter` can now be reused in multiple components.
- Keeps the logic of counting separate and reusable.
- Can pass different initial values when using the hook.

2. `useTransition` Hook

What is `useTransition`?

The **`useTransition`** hook is used to manage **UI transitions**. It lets you mark state updates as non-urgent, preventing blocking UI rendering during slow updates.

Why Use `useTransition`?

- For smoother UI during heavy state updates.
- Prevents the UI from "freezing" during expensive operations.
- Improves user experience by prioritizing immediate interactions.

Basic Syntax:

```
const [isPending, startTransition] = useTransition();
```

`isPending`: A boolean that shows if the transition is ongoing.

`startTransition(callback)`: Wraps the state updates that should be treated as low-priority.

Example: Search Filter with `useTransition`

This example shows how `useTransition` can prevent UI from lagging during filtering.

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

function SearchFilter() {
  const [input, setInput] = useState('');
  const [list, setList] = useState([]);
  const [isPending, startTransition] = useTransition();

  const ITEMS = Array.from({ length: 10000 }, (_, i) => `Item ${i + 1}`);

  const handleChange = (e) => {
    const value = e.target.value;
    setInput(value);

    // Low-priority update handled by useTransition
    startTransition(() => {
      const filteredItems = ITEMS.filter((item) =>
        item.toLowerCase().includes(value.toLowerCase())
      );
      setList(filteredItems);
    });
  };

  return (
    <div>
      <input type="text" value={input} onChange={handleChange} placeholder="Search Items..."
/>
      {isPending && <p>Loading filtered results...</p>}
      <ul>
        {list.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default SearchFilter;
```

Key Points of `useTransition`:

- Prevents the input field from lagging while filtering a large dataset.
- The **loading text** (`Loading filtered results...`) appears if filtering takes time.

- Immediate UI updates (like typing in input) happen first, while heavy tasks (like filtering) run in the background.