useRef Hook

- useRef is a React Hook that allows you to create a mutable reference to store data or interact directly with the DOM.
- Unlike useState, changing a useRef value does not cause a re-render of the component.
- This makes it highly efficient for scenarios where you need to persist values across renders without triggering updates to the UI.
- useRef is a React hook that provides a way to create a mutable reference that persists across renders.
- useRef hook is often used to:
    - Access DOM elements directly.
    - Store a mutable value that does not trigger re-renders when updated.
- The useRef hook returns an object with a .current property that holds the mutable value.

How useRef Works

- **Initialization:** When you call useRef(initialValue), it returns an object with a single property, .current, which stores the value.

```
const myRef = useRef(initialValue);
console.log(myRef.current);                    // Logs the initial value
```

- **Persistence Across Renders:** The .current value persists across re-renders, meaning it maintains its value even after the component re-renders. However, it does not reset unless explicitly changed.

- **Mutable Value:** You can directly modify the .current property, and it will update immediately. However, since useRef does not cause re-renders, you won't see the updated value in the UI unless you explicitly trigger a re-render through useState or other means.

- **Direct DOM Interaction:** useRef is commonly used to directly interact with DOM elements, replacing the legacy React.createRef() in functional components.

Why Use useRef?

- **Access DOM Elements:** Directly interact with native DOM elements like <input>, <button>, etc.
  Example: Setting focus, selecting text, or triggering a scroll action.
- **Store Persistent Values:** Keep mutable values that persist across renders without causing re-renders (e.g., previous values, timers, counters).
- **Avoid Unnecessary State Updates:** Using useRef for mutable values avoids the overhead of re-rendering caused by useState.
- **Performance Optimization:** Reduces unnecessary renders, improving performance for components with frequent updates.

Example: Accessing a DOM Element

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

function App() {
  const inputRef = useRef(null);

  useEffect(() => {
    // Focus the input element on component mount
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} placeholder="Type something..." />;
}

export default App;
```

In above program:

- useRef(null) initializes the reference.
- ref={inputRef} links the input element to the reference.
- inputRef.current allows direct interaction with the DOM element.

Problems Solved by useRef

- **Avoiding Re-renders**
  - When managing data that should not trigger re-renders, such as timers, counters, or intermediate values.
  - Using useState for such tasks would trigger unnecessary re-renders, which can lead to performance issues.
- **Accessing DOM Elements**
  - Direct interaction with DOM elements (like setting focus or modifying styles) is often necessary in React.
  - useRef provides a clean way to achieve this without using document.querySelector or getElementById.
- **Storing Mutable Values**
  - For data that changes but does not affect the UI (e.g., storing the previous state or keeping track of interval IDs).

Example: Avoiding Unnecessary Re-renders before useRef (using useState)

```
import React, { useState } from "react";
function App() {
 const [timer, setTimer] = useState(null);
 const startTimer = () => {
  setTimer(setInterval(() => console.log("Timer running..."), 1000));
 };
 const stopTimer = () => {
  clearInterval(timer); // Accessing timer through state
  setTimer(null);
 };
 console.log("Component re-rendered");
 return (
  <div>
   <button onClick={startTimer}>Start Timer</button>
   <button onClick={stopTimer}>Stop Timer</button>
  </div>
 );}
export default App;
```

Problem: Every time setTimer is called, it triggers a re-render even though the timer value does not affect the UI.

Example: Avoiding Unnecessary Re-renders with useRef

```
import React, { useRef } from "react";
function App() {
  const timerRef = useRef(null);
  const startTimer = () => {
    timerRef.current = setInterval(() => console.log("Timer running..."), 1000);
  };
  const stopTimer = () => {
    clearInterval(timerRef.current);
    timerRef.current = null;
  };
  console.log("Component rendered once");
  return (
    <div>
      <button onClick={startTimer}>Start Timer</button>
      <button onClick={stopTimer}>Stop Timer</button>
    </div>
  );
}
export default App;
```

Solution: Using useRef avoids re-renders, improving performance.

Example: Accessing DOM Elements before useRef (using document.querySelector)

```
import React, { useEffect } from "react";
function App() {
  useEffect(() => {
    const input = document.querySelector("#my-input");
    input.focus();       // Focuses the input element on mount
  }, []);
  return <input id="my-input" placeholder="Type something..." />;
}
export default App;
```

Problem: Using document.querySelector breaks React's declarative pattern and tightly couples the component logic to the DOM.

Example: Accessing DOM Elements with useRef

```
import React, { useRef, useEffect } from "react";
function App() {
  const inputRef = useRef();
  useEffect(() => {
    inputRef.current.focus();          // Focuses the input element
  }, []);
  return <input ref={inputRef} placeholder="Type something again..." />;
}
export default App;
```

Solution: useRef provides a declarative and cleaner approach to accessing DOM elements.

Example: Storing Mutable Values Across Renders before useRef (using local variables)

```
import React, { useState } from "react";
function App() {
  let renderCount = 0;
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);
    renderCount++;          // This resets on each render
    console.log("Render Count:", renderCount);
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
export default App;
```

Problem: The renderCount variable resets on every render because it does not persist across renders.

Example: Storing Mutable Values Across Renders with useRef

```
import React, { useRef, useState } from "react";
function App() {
  const renderCount = useRef(0);
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);
    renderCount.current++; // Updates persistently without re-renders
    console.log("Render Count:", renderCount.current);
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );}
export default App;
```

Solution: useRef persists the renderCount value across renders without resetting.

Example: Tracking Component Mount/Unmount Status before useRef (Using State)

```
import React, { useState, useEffect } from "react";
function App() {
  const [isMounted, setIsMounted] = useState(false);
  useEffect(() => {
    setIsMounted(true); // Set mounted state to true
    return () => {
      setIsMounted(false); // Reset to false on unmount
    };
  }, []);
  console.log("Is component mounted:", isMounted);
  return <div>Check console for mount status.</div>;
}
export default App;
```

Problem: Using useState to track the mount status triggers unnecessary re-renders.

Example: Tracking Component Mount/Unmount Status with useRef

```
import React, { useRef, useEffect } from "react";
function App() {
  const isMounted = useRef(false);
  useEffect(() => {
    isMounted.current = true; // Set to true on mount
    return () => {
      isMounted.current = false; // Reset to false on unmount
    };
  }, []);
  console.log("Is component mounted:", isMounted.current);
  return <div>Check console for mount status.</div>;
}
export default App;
```

Solution: Using useRef avoids re-renders while efficiently tracking the component's mount/unmount status.

Point to Remember

- **Before useRef**
  - Relying on useState for mutable values leads to unnecessary re-renders.
  - Accessing DOM elements with selectors breaks React's declarative principles.
  - Mutable variables reset on every render.

- **With useRef**
  - Avoids re-renders while persisting mutable values.
  - Provides a declarative, React-friendly way to interact with the DOM.
  - Enables efficient management of side effects like timers or counters.

Difference Between useRef and useState

useRef and useState are both React hooks, but they serve different purposes and work differently. Below is a detailed comparison:

Aspect	useState	useRef
Purpose	To manage stateful data that triggers re-renders.	To store mutable data or references that persist across renders.
Re-renders	Updating state triggers a re-render.	Updating the .current property does not trigger a re-render.
Primary Use Case	Storing dynamic UI-related data (e.g., form inputs).	Accessing DOM elements, managing timers, or storing mutable values.
Data Mutability	Immutable by default; state must be replaced to update.	Mutable; .current can be directly modified.
Value Reset on Re-renders	State persists across renders.	Ref value persists across renders but is not reactive.
Interaction with DOM	Not designed for direct DOM manipulation.	Can directly reference and manipulate DOM elements.
Performance Impact	May cause performance issues if updated frequently.	Efficient for frequent updates as it avoids re-renders.
Examples	Storing form values, counters, or UI states.	Managing timers, storing scroll positions, or DOM references.

Use Cases for Each Hook

When to Use useState

- When you want React to re-render the component after the state updates.
- When managing UI-related states (e.g., form values, toggles).

When to Use useRef

- When you need to store a mutable value that doesn't affect the UI.
- When accessing DOM elements or managing timers/intervals.

Example: Counter Example using useState

```
import React, { useState } from "react";
function App() {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1); // Triggers a re-render
  };
  console.log("Component re-rendered");
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
export default App;
```

Behavior: Each time the button is clicked, the state (count) updates, and the component re-renders to reflect the new state.

Example: Counter Example using useRef

```
import React, { useRef } from "react";
function App() {
  const countRef = useRef(0);
  const increment = () => {
    countRef.current++; // Updates without re-rendering
    console.log("Current Count:", countRef.current);
  };
  console.log("Component rendered once");
  return <button onClick={increment}>Increment</button>;
}
export default App;
```

Behavior: The count value is updated in the console without causing re-renders.

Example: Storing Previous Value using useState

```
import React, { useState, useEffect } from "react";
function App() {
  const [count, setCount] = useState(0);
  const [prevCount, setPrevCount] = useState(null);
  useEffect(() => {
    setPrevCount(count); // Updates the previous value state
  }, [count]);
  return (
    <div>
      <p>Current Count: {count}</p>
      <p>Previous Count: {prevCount}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );}
export default App;
```

Problem: Using useState for the previous value causes unnecessary re-renders.

Example: Storing Previous Value using using useRef

```
import React, { useState, useRef, useEffect } from "react";
function App() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count; // Updates the previous value
  });
  return (
    <div>
      <p>Current Count: {count}</p>
      <p>Previous Count: {prevCountRef.current}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );}
export default App;
```

Solution: useRef efficiently stores the previous value without triggering re-renders.

useRef Pitfalls

When using useRef, it's important to understand its limitations and avoid common pitfalls to ensure your code remains clean and bug-free.

1. **Misunderstanding useRef Updates, Do Not Trigger Re-renders**

   Bug: Assuming updating a useRef value will cause the component to re-render.

   Example

```
import React, { useRef } from "react";
function App() {
  const countRef = useRef(0);
  const increment = () => {
    countRef.current++;
    console.log("Count:", countRef.current);
  };
  return (    <div>
      <p>Count: {countRef.current}</p> { /* This will not update */ }
      <button onClick={increment}>Increment</button>
    </div>  );}
export default App;
```

   Pitfall: The <p> element will not update because useRef does not trigger re-renders.

   Solution: Use useState for reactive updates if the UI depends on the value:

   Example

```
import React, { useRef, useState } from "react";
function App() {
  const countRef = useRef(0);
  const [count, setCount] = useState(0);
  const increment = () => {
    countRef.current++;
    setCount(countRef.current); // Trigger re-render
  };
  return (    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>  );}
export default App;
```

2. Forgetting to Initialize useRef Properly

Bug: Trying to access useRef.current before it has been initialized.

Example

```
import React, { useRef } from "react";
function App() {
  const inputRef = useRef(); // Default is undefined
  const handleFocus = () => {
    inputRef.current.focus();  // Error: Cannot read property 'focus' of undefined
  };
  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );
}
export default App;
```

Pitfall: Accessing current without ensuring the ref has been assigned.

Solution: Ensure the ref is attached to a DOM element before accessing it.

Example

```
import React, { useRef } from "react";
function App() {
  const inputRef = useRef(null); // Explicitly set initial value
  const handleFocus = () => {
    if (inputRef.current) {
      inputRef.current.focus();
    }
  };
  return (
    <div>
      <input ref={inputRef} />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );}
export default App;
```

3. Mutating useRef Incorrectly in useEffect

Bug: Improperly mutating a useRef value inside useEffect can lead to stale or unexpected values.

Example

```
import React, { useRef, useEffect } from "react";

function App() {
  const refValue = useRef(0);
  useEffect(() => {
    refValue.current = refValue.current + 1; // Unintended updates
    console.log("Current Ref Value:", refValue.current);
  });

return <div>Ref Value: {refValue.current}</div>;
}

export default App;
```

Pitfall: This pattern can result in unpredictable updates and is hard to debug.

Solution: Update useRef values only when necessary and understand its behavior in useEffect.

Example

```
import React, { useRef, useEffect } from "react";

function App() {
  const refValue = useRef(0);
  useEffect(() => {
    refValue.current = refValue.current + 1; // Controlled and intentional
    console.log("Updated Ref Value:", refValue.current);
  }, []); // Only runs on mount

return <div>Ref Value (Updated on Mount): {refValue.current}
</div>;
}

export default App;
```

4. **Overusing useRef for Values Better Suited to useState**

Bug: Using useRef for reactive values that should trigger UI updates.

Example

```
import React, { useRef } from "react";
function App() {
  const counter = useRef(0);
  const increment = () => {
    counter.current++; // Value updates, but UI doesn't
    console.log("Counter:", counter.current);
  };
  return (
    <div>
      <p>Counter: {counter.current}</p> { /* Will not update */ }
      <button onClick={increment}>Increment</button>
    </div>
  );
}
export default App;
```

Pitfall: This creates a disconnect between the value and the UI, leading to confusing behavior.

Solution: Use useState for reactive data.

Example

```
import React, { useState } from "react";
function App() {
  const [counter, setCounter] = useState(0);
  const increment = () => {
    setCounter(counter + 1); // Updates state and triggers re-render
  };
  return (
    <div>
      <p>Counter: {counter}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
export default App;
```

5. **Forgetting Cleanup for DOM References**

Bug: Leaving event listeners or intervals tied to useRef values causes memory leaks.

Example

```
import React, { useRef, useEffect } from "react";
function App() {
  const intervalRef = useRef();
  useEffect(() => {
    intervalRef.current = setInterval(() => {
      console.log("Interval running...");
    }, 1000);
    // Missing cleanup
  }, []);
  return <div>Check console for interval logs.</div>;
}
export default App;
```

Pitfall: Not cleaning up intervals or event listeners can lead to performance issues or unexpected behavior.

Solution: Always clean up side effects.

Example

```
import React, { useRef, useEffect } from "react";
function App() {
  const intervalRef = useRef();
  useEffect(() => {
    intervalRef.current = setInterval(() => {
      console.log("Interval running...");
    }, 1000);
    return () => {
      clearInterval(intervalRef.current); // Cleanup
      console.log("Interval cleared");
    };
  }, []);
  return <div>Check console for interval logs.</div>;
}
export default App;
```