useLayoutEffect Hook

- useLayoutEffect is a React hook used to perform side effects that need to occur before the browser paints the screen.
- It is similar to useEffect, but it runs synchronously after the DOM updates and before the browser renders the screen.
- useLayoutEffect is a React hook that allows developers to perform side effects synchronously after the DOM has been updated but before the browser has painted the screen.
- It provides a way to ensure DOM-related changes or measurements occur before the user perceives them.

Key Characteristics

- Runs synchronously after DOM mutations.
- Blocks the browser's painting process until its callback completes.
- Useful for measuring DOM properties or manipulating the DOM before the user sees it.

Why useLayoutEffect introduced?

- **Timing of Updates:** While useEffect is sufficient for most scenarios, it operates asynchronously and runs after the screen is painted. This can lead to situations where DOM measurements or updates cause layout shifts or flickers.
- **Precise DOM Interactions:** For use cases that demand measuring DOM elements, applying layout-critical styles, or interacting with third-party libraries that manipulate the DOM, useLayoutEffect ensures the UI is finalized before being displayed to the user.

How does useLayoutEffect work?

- **Execution Flow**
  - The component renders and updates the DOM.
  - React calls the useLayoutEffect callback before the browser paints the screen.
  - Once the callback completes, the browser continues with the painting process.
- **Synchronous Nature:** Unlike useEffect, which is asynchronous and doesn't block the paint process, useLayoutEffect runs synchronously. This means it delays the browser's rendering until its callback finishes.

Example: Changing Background Color Before Render

```
import React, { useLayoutEffect, useState } from "react";
function App() {
  const [color, setColor] = useState("lightblue");
  useLayoutEffect(() => {
    // Synchronously update the background color before the browser paints
    setColor("lightgreen");
  }, []);
  return (
    <div
      style={{
        width: "300px",
        height: "100px",
        backgroundColor: color,
        textAlign: "center",
        lineHeight: "100px",
        color: "white",     }}     >
      Background Color: {color}
    </div>
  );
}
export default App;
```

Differences Between useEffect and useLayoutEffect

Aspect	useEffect	useLayoutEffect
Execution Timing	Runs asynchronously *after* the browser has painted the screen.	Runs synchronously *before* the browser paints the screen.
Blocking Nature	Non-blocking; does not delay rendering.	Blocking; delays rendering until the callback finishes.
UseCase	For side effects that don't affect the visual appearance of the UI (e.g., API calls, logging).	For layout-critical side effects (e.g., DOM measurements, layout adjustments).
Performance Impact	Minimal, as it doesn't block rendering.	Can slow rendering if used excessively or for heavy computations.
Visible UI Flicker	Changes may cause a visible flicker if they affect the DOM.	Prevents visible flickers by applying changes synchronously before render.

Example: Styling Based on Measurements

Code Using useEffect (May Cause Flicker)

```
import React, { useEffect, useRef, useState } from "react";
function App() {
  const divRef = useRef();
  const [bgColor, setBgColor] = useState("lightblue");
  useEffect(() => {
    if (divRef.current && divRef.current.offsetWidth > 100) {
      setBgColor("lightgreen");    }
  }, []);
  return (
    <div       ref={divRef}
      style={{
        width: "150px",       height: "100px",       background: bgColor,      }}    >
      Styled with Effect
    </div>
  );}
export default App;
```

Code Using useLayoutEffect (Prevents Flicker)

```
import React, { useLayoutEffect, useRef, useState } from "react";
function App() {
  const divRef = useRef();
  const [bgColor, setBgColor] = useState("lightblue");
  useLayoutEffect(() => {
    if (divRef.current && divRef.current.offsetWidth > 100) {
      setBgColor("lightgreen");    }
  }, []);
  return (
    <div       ref={divRef}
      style={{  width: "150px",    height: "100px",   background: bgColor,      }}    >
      Styled with LayoutEffect
    </div>
  );}
export default App;
```

Example: Styling Based on Measurements

1. Code Using useEffect (May Cause Flicker)

```
import React, { useEffect, useRef, useState } from "react";
function App() {
  const divRef = useRef();
  const [bgColor, setBgColor] = useState("lightblue");
  useEffect(() => {
    if (divRef.current && divRef.current.offsetWidth > 100) {
      setBgColor("lightgreen");    }
  }, []);
  return (
    <div       ref={divRef}
      style={{     width: "150px",    height: "100px",    background: bgColor,      }}    >
      Styled with Effect
    </div>
  ); }
export default App;
```

2. Code Using useLayoutEffect (Prevents Flicker)

```
import React, { useLayoutEffect, useRef, useState } from "react";
function App() {
  const divRef = useRef();
  const [bgColor, setBgColor] = useState("lightblue");
  useLayoutEffect(() => {
    if (divRef.current && divRef.current.offsetWidth > 100) {
      setBgColor("lightgreen");    }
  }, []);
  return (
    <div      ref={divRef}
      style={{   width: "150px",   height: "100px",   background: bgColor,      }}    >
      Styled with LayoutEffect
    </div>
  );}

export default App;
```

Example: Create a modal that dynamically adjusts its position and dimensions based on its content or parent element.

1. Code Using useEffect (May Cause Flicker)

```
import React, { useState, useEffect, useRef } from "react";
function Modal({ isOpen }) {
  const modalRef = useRef();
  const [style, setStyle] = useState({});
  useEffect(() => {
    if (isOpen && modalRef.current) {
      const { offsetWidth, offsetHeight } = modalRef.current;
      setStyle({
        position: "absolute",
        top: `calc(50% - ${offsetHeight / 2}px)`,
        left: `calc(50% - ${offsetWidth / 2}px)`,
        backgroundColor: "white",
        padding: "20px",
        border: "1px solid gray",
      });    }
  }, [isOpen]);
  if (!isOpen) return null;
  return (
    <div ref={modalRef} style={style}>
      <p>Modal with useEffect Again</p>
    </div>
  );}

function App() {
  const [isOpen, setIsOpen] = useState(false);
  return (
    <div>
      <button onClick={() => setIsOpen(!isOpen)}>
        {isOpen ? "Close Modal" : "Open Modal"}
      </button>
      <Modal isOpen={isOpen} />
    </div>
  );}
export default App;
```

2. Code Using useLayoutEffect (Prevents Flicker)

```
import React, { useState, useLayoutEffect, useRef } from "react";
function Modal({ isOpen }) {
  const modalRef = useRef();
  const [style, setStyle] = useState({});
  useLayoutEffect(() => {
    if (isOpen && modalRef.current) {
      const { offsetWidth, offsetHeight } = modalRef.current;
      setStyle({
        position: "absolute",
        top: `calc(50% - ${offsetHeight / 2}px)`,
        left: `calc(50% - ${offsetWidth / 2}px)`,
        backgroundColor: "white",
        padding: "20px",
        border: "1px solid gray",
      });
    }
  }, [isOpen]);
  if (!isOpen) return null;
  return (
    <div ref={modalRef} style={style}>
      <p>Modal with useLayoutEffect</p>
    </div>
  );}

function App() {
  const [isOpen, setIsOpen] = useState(false);
  return (
    <div>
      <button onClick={() => setIsOpen(!isOpen)}>
        {isOpen ? "Close Modal" : "Open Modal"}
      </button>
      <Modal isOpen={isOpen} />
    </div>
  );
}
export default App;
```

Pitfalls of useLayoutEffect

- **Blocking the Browser Render Process:** useLayoutEffect runs synchronously and blocks the browser from painting the screen until the callback completes. If it performs heavy computations or has asynchronous code, it can cause noticeable delays and impact performance.
  Example

```
useLayoutEffect(() => {
  // Long computation
  for (let i = 0; i < 1e8; i++) {}
}, []);
```

  Impact: The page will "freeze" momentarily before rendering.
  Solution: Move non-layout-critical tasks to useEffect.

- **Accessing Null References:** If ref.current is accessed before the DOM node is fully created or when the component unmounts, it may result in errors.
  Example

```
useLayoutEffect(() => {
  console.log(divRef.current.offsetWidth); // Throws an error if divRef is null
}, []);
```

  Impact: Causes runtime errors if divRef.current is null.
  Solution: Always check ref.current before accessing it:

```
if (divRef.current) {
  console.log(divRef.current.offsetWidth);
}
```

- **Overuse of useLayoutEffect:** Using useLayoutEffect for tasks that don't involve layout calculations or DOM updates can make your application unnecessarily complex and slow.
  Example

```
useLayoutEffect(() => {
  console.log("Component mounted"); // No layout-critical operation here
}, []);
```

  Impact: Unnecessary blocking of rendering.
  Solution: Use useEffect for non-visual side effects like logging or API calls.

- **Infinite Loops:** Accidentally causing infinite re-renders by updating state inside useLayoutEffect without properly managing dependencies.
  Example

```
useLayoutEffect(() => {
  setState(s => s + 1); // Causes infinite re-renders
});
```

  Impact: Infinite loop crashes the app.
  Solution: Carefully manage dependency arrays and avoid unnecessary state updates.

Debugging Techniques of useLayoutEffect

- **Inspect Render Timing:** Use React DevTools Profiler to analyze when useLayoutEffect runs and measure its impact on rendering.
- **Log ref Values:** Add console.log to check the state of ref.current:

```
useLayoutEffect(() => {
  console.log("ref.current:", ref.current);
}, []);
```

- **Split Effects:** Break large useLayoutEffect logic into smaller effects for better debugging and clarity.

Identifying When to Switch to useEffect

Use useEffect When:

- **Non-Visual Side Effects:** Tasks like logging, fetching data, or triggering analytics don't affect the layout and should not block rendering.
  Example

```
useEffect(() => {
  console.log("Component mounted");
  fetch("/api/data");
}, []);
```

- **Post-Paint Tasks:** If the task can wait until the browser has painted the screen, use useEffect to avoid blocking the render.
  Example

```
useEffect(() => {
  document.title = "Page Loaded";
}, []);
```

- **Avoiding Performance Bottlenecks:** Use useEffect for operations that might take time, such as state updates or asynchronous tasks.
  Example

```
useEffect(() => {
  const fetchData = async () => {
    const data = await fetch("/api/data");
    setData(await data.json());
  };
  fetchData();
}, []);
```

Use useLayoutEffect When:

- **Layout or DOM Measurement:** If you need precise measurements or changes before the user sees the screen.
  Example

```
useLayoutEffect(() => {
  const height = divRef.current.offsetHeight;
  console.log("Height:", height);
}, []);
```

- **Preventing Flickers or Layout Shifts:** Use useLayoutEffect to ensure DOM updates happen before rendering.
  Example

```
useLayoutEffect(() => {
  if (divRef.current) {
    divRef.current.style.backgroundColor = "lightgreen";
  }
}, []);
```

useDebugValue Hook

- The useDebugValue hook in React is designed specifically for debugging custom hooks.
- It allows you to display a label in React Developer Tools to provide more insight into what a custom hook is doing.
- It enables you to add labels or data to custom hooks that can be displayed in React Developer Tools, offering insight into what the hook is doing internally without modifying the UI.
- Unlike other hooks, useDebugValue has no effect on runtime functionality. It serves only to aid debugging and is ignored in production builds for performance.

Example

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

function useCustomHook(value) {          //Custom Hook
 useDebugValue(value); // This will show the value in React Developer Tools
 return value * 2;
}
```

Purpose of useDebugValue

- Provide insights about the internal state of a custom hook in React DevTools.
- Improve readability and debugging efficiency for hooks used in large-scale applications.
- Avoid cluttering the console with console.log-style debugging statements.

Use Cases of useDebugValue

- **Multiple Hooks Sharing Similar Behavior:** Distinguishing between hooks in React DevTools.
  Example: Hooks managing different API calls.
- **Complex Custom Hooks:** Debugging hooks with derived state or multi-step logic.
  Example: A hook that computes filtered and paginated lists.
- **Condition-Based Hooks:** Showing only relevant debug values when a condition is met.

Why it is specifically used for debugging Custom Hooks?

Due to Focus on Abstraction, because Custom hooks abstract logic and don't directly expose internal details to components. useDebugValue helps surface these details in a developer-friendly way.

- **Without useDebugValue:** Custom hooks appear as black boxes in React DevTools, making it hard to understand what's happening.
- **With useDebugValue:** Developers can inspect the state or derived computations within the hook, aiding debugging and testing without introducing additional runtime overhead.

Example: Use of useDebugValue hook with custom hook

```
import React, { useState, useEffect, useDebugValue } from "react";

function useFetchData(url) {        // custom hook
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((response) => response.json())
      .then((json) => setData(json));
  }, [url]);

  // UseDebugValue helps surface the current data state
  useDebugValue(data ? "Data Loaded" : "Loading...");

  return data;
}

function App() {
  const [url, setUrl] = useState("https://jsonplaceholder.typicode.com/posts");
  const data = useFetchData(url);

  return (
    <div style={{ padding: "20px", fontFamily: "Arial" }}>
```

```
      <h1>Data Fetching with useDebugValue</h1>

      <div style={{ marginBottom: "20px" }}>
        <label>
          API URL:
          <input
            type="text"
            value={url}
            onChange={(e) => setUrl(e.target.value)}
            style={{ marginLeft: "10px", width: "300px" }}
          />
        </label>
      </div>

      <div>
        <h2>Fetched Data:</h2>
        {data ? (
          <ul>
            {data.map((item) => (
              <li key={item.id}>
                <strong>{item.title}</strong>: {item.body}
              </li>
            ))}
          </ul>
        ) : (
          <p>Loading...</p>
        )}
      </div>
    </div>
  );
}

export default App;
```

DevTools Output: When debugging, this helps distinguish between the hook states
useFetchData: "Data Loaded"

Key Features of useDebugValue

- **Debug Information in DevTools:** Shows formatted and readable output for custom hooks.
- **Formatted Output:** Supports transforming values into human-readable formats using a formatter function.
- **No Production Impact:** It's stripped out in production builds, ensuring no impact on performance.
- **Conditional Computation:** Computations for debug values are executed only when React DevTools is active.

Example: Debugging multiple custom hooks

```
import React, { useState, useEffect, useDebugValue } from "react";

function useAuth(user) {                    // custom hook
  const [isAuthenticated, setAuthenticated] = useState(false);
  useEffect(() => {
    setAuthenticated(!!user);
  }, [user]);
  // Debugging information
  useDebugValue(isAuthenticated ? "Authenticated" : "Not Authenticated");
  return isAuthenticated;
}

function useProfile(userId) {          // custom hook
  const [profile, setProfile] = useState(null);
  useEffect(() => {
    if (!userId) return;
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
      .then((res) => res.json())
      .then((data) => setProfile(data))
      .catch(() => setProfile(null));
  }, [userId]);
  // Debugging information
  useDebugValue(profile?.name || "Fetching Profile");
  return profile;
}
```

```
function App() {
  const [user, setUser] = useState(null);
  const isAuthenticated = useAuth(user);
  const profile = useProfile(user?.id);

  return (
    <div style={{ padding: "20px", fontFamily: "Arial" }}>
      <h1>User Authentication and Profile</h1>
      <div style={{ marginBottom: "20px" }}>
        <label>
          Enter User ID:
          <input            type="number"   placeholder="User ID"
            onChange={(e) => setUser({ id: e.target.value })}
            style={{ marginLeft: "10px", width: "200px" }}           />
        </label>
      </div>
      <div>
        <h2>Authentication Status:</h2>
              <p>{isAuthenticated   ?   "User   is   Authenticated"   :   "User   is   Not
Authenticated"}</p>

        <h2>User Profile:</h2>
        {profile ? (
          <div>
            <p><strong>Name:</strong> {profile.name}</p>
            <p><strong>Email:</strong> {profile.email}</p>
            <p><strong>Phone:</strong> {profile.phone}</p>
          </div>
        ) : (
          <p>Loading profile...</p>
        )}
      </div>
    </div>
  );
}

export default App;
```

useDebugValue with Conditional Formatting

- The useDebugValue hook can be used with a formatting function to display the debug value in a more human-readable or context-aware format.
- This formatting function is executed only when React Developer Tools are open, preventing unnecessary performance overhead in production or during normal app usage.

Syntax

useDebugValue(value, formatFunction);

where,

- value: The value you want to debug (e.g., state, derived data, etc.).
- formatFunction: A function that takes value as input and returns a formatted string or object for display in React Developer Tools.

Example: Debugging a Custom Hook for Theme Management

```
import { useState, useDebugValue } from "react";
function useTheme() {
  const [theme, setTheme] = useState("light"); // Default theme is light.
  // Toggle between light and dark themes
  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
  };
  // Using useDebugValue with conditional formatting
  useDebugValue(theme, (theme) => (theme === "light" ? "Light Theme" : "Dark Theme"));
  return [theme, toggleTheme];
}
function App() {
  const [theme, toggleTheme] = useTheme(); // Using the hook in a component
  return (
    <div style={{ background: theme === "light" ? "#fff" : "#333", color: theme === "light" ? "#000" : "#fff" }}>
      <h1>{theme === "light" ? "Light Mode" : "Dark Mode"}</h1>
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}
export default App;
```

Example: Custom Hook for Online/Offline Status

```
import { useState, useEffect, useDebugValue } from "react";
function useOnlineStatus() {
  const [isOnline, setIsOnline] = useState(false);
  useEffect(() => {
    function handleOnline() {
      setIsOnline(true);    }
    function handleOffline() {
      setIsOnline(false);    }
    // Adding event listeners for online and offline events
    window.addEventListener("online", handleOnline);
    window.addEventListener("offline", handleOffline);
    // Cleanup function to remove event listeners
    return () => {
      window.removeEventListener("online", handleOnline);
      window.removeEventListener("offline", handleOffline);
    };
  }, []);
  // Debug value for Developer Tools
  useDebugValue(isOnline, (isOnline) => (isOnline ? "User is Online" : "User is
Offline"));
  return [isOnline, setIsOnline];
}
function App() {
  const [isOnline, setIsOnline] = useOnlineStatus();
  const toggleOnlineStatus = () => {
    setIsOnline((prev) => !prev); // Toggle the online status
  };
  return (
    <div style={{ textAlign: "center", marginTop: "20px" }}>
      <h1>{isOnline ? "You are Online" : "You are Offline"}</h1>
      <button onClick={toggleOnlineStatus} style={{ padding: "10px 20px", fontSize:
"16px" }}>        Toggle Status      </button>
    </div>
  );
}
export default App;
```

When to Use useDebugValue

useDebugValue should be used when:

- You are creating custom hooks and want to make their state or behavior more transparent during debugging in React Developer Tools.
- You need to provide context-aware information about the internal state or computations of the hook without cluttering the UI or the console.

Avoiding Unnecessary Debug Value Computation

Why This Matters

If the value passed to useDebugValue involves expensive computations (e.g., large data processing), it can negatively impact performance, even if the value is not being inspected in React Developer Tools.

Solution: Use Conditional Formatting

The second parameter of useDebugValue allows you to define a formatter function. This function is executed only when React Developer Tools is open.

Impact on Application Performance

Using useDebugValue improperly (e.g., with heavy computations) can:

- Introduce unnecessary overhead during normal renders.
- Lead to performance degradation, especially in applications with complex hooks or large data processing.