

# EventListener difference and use



tell me the difference between attaching eventlistener like this:

M1:

```
document.getElementById('myButton3').addEventListener("click", function () {
  var count = 0;
  console.warn("Button clicked from EventListener3 file-->", "count-->", ++count );
})
```

M2:

```
function AddEventListener () {
  var count = 0;
  var count1 =10;

  document.getElementById('myButton4').addEventListener("click", function () {
    console.warn("Button clicked from EventListener3 file-->", "count-->", ++count );
  })
}

AddEventListener();
```

one directly one via function ....

1. difference between them.
2. when to use which
3. ot they are same

to me like a new bie

Perfect question, Vipul 🙌 — this shows you're starting to *really* understand how JavaScript's **scope** and **event listeners** work. Let's go step-by-step in a simple way 👇

## Code Recap

### ✓ M1

js

```
document.getElementById('myButton3').addEventListener("click", function () {
  var count = 0;
```

```
    console.warn("Button clicked from EventListener3 file-->", "count-->", ++count);
});
```

## ✓ M2

js

```
function AddEventListener() {
  var count = 0;
  var count1 = 10;

  document.getElementById('myButton4').addEventListener("click", function () {
    console.warn("Button clicked from EventListener3 file-->", "count-->", ++count);
  });
}

AddEventListener();
```

## 🧠 1. Difference Between Them

Aspect	M1	M2
Where it's defined	Directly in the global scope (runs immediately)	Inside a function (runs only when you call the function)
When listener is attached	Immediately when the script loads	Only after <b>AddEventListener()</b> is called
count variable	Re-created every time the button is clicked (resets to 0 each click)	Created once when the function runs — <b>persists between clicks</b> (because of closure)
Flexibility	Fixed — can't easily reuse or reattach	Reusable — can be called multiple times for different buttons if needed
Encapsulation	Everything is global	Encapsulated — variables stay private inside the function

## 🧩 2. What Actually Happens (Visually)

### ▀ M1 (direct)

rust

```
Script loads ->
  addEventListener runs ->
    'click' listener added immediately
```

Each click:

javascript

```
count = 0 // created fresh every click
++count → 1
console.warn("count -> 1")
```

👉 Every click shows **count -> 1**, because **count** is *redeclared* inside the event callback each time.

## M2 (inside a function)

```
csharp

Script loads ->
function AddEventListener is defined
AddEventListener() is called ->
count = 0 created once
listener added
```

Each click:

```
arduino

++count // count keeps previous value
1st click → 1
2nd click → 2
3rd click → 3
```

 **count remembers** previous clicks — this happens due to **closure** (the event handler “remembers” the variable from where it was created).

## 3. When to Use Which

Scenario	Use Which	Why
Simple one-time button listener	M1	Quick and easy, no need for extra functions
You need private variables or want to reuse logic	M2	Encapsulates variables, avoids polluting global scope
You need to add listeners dynamically for multiple buttons	M2	You can call the function multiple times with different IDs
You want <b>count</b> or other data to persist between clicks	M2	Closure keeps the data alive

## 4. Are They Same?

 **No**, they are not exactly same.

They both *attach* a click listener, but:

- In **M1**, the listener is added directly and variables don't persist.
- In **M2**, the listener is added *via* a function, and variables can persist because of closure.

## Quick Example to Prove It

```
js

// M1
document.getElementById("btn1").addEventListener("click", () => {
```

```

var count = 0;
console.log("M1 count:", ++count);
});

// M2
function add() {
  var count = 0;
  document.getElementById("btn2").addEventListener("click", () => {
    console.log("M2 count:", ++count);
  });
}
add();

```

Click both buttons a few times:

yaml

```

M1 count: 1
M1 count: 1
M1 count: 1

M2 count: 1
M2 count: 2
M2 count: 3

```

That's the **real difference** ⚡

## In Short:

- **M1:** Immediate, global, no memory
- **M2:** Scoped, flexible, remembers things (closure)

Would you like me to show a **version of M2** where we can attach the same logic to *many buttons dynamically* (like a reusable pattern)?



yes

Perfect 👍 Vipul — let's now make **M2** smart and *reusable*, so you can attach the same logic to **many buttons** without repeating code.

## Goal

We'll create a **function** that:

- Takes a **button ID** (like "myButton1", "myButton2", etc.)
- Attaches a **click event listener** to it
- Keeps its own **private counter** (each button remembers its own click count)

## Example Code

```
html

<button id="myButton1">Button 1</button>
<button id="myButton2">Button 2</button>
<button id="myButton3">Button 3</button>

<script>
  // Reusable function
  function attachClickCounter(buttonId) {
    let count = 0; // private variable for this specific button

    const button = document.getElementById(buttonId);

    if (!button) {
      console.warn(`Button with id "${buttonId}" not found!`);
      return;
    }

    button.addEventListener("click", function () {
      count++;
      console.warn(`${buttonId} clicked ${count} times`);
    });
  }

  // Attach the same logic to multiple buttons
  attachClickCounter("myButton1");
  attachClickCounter("myButton2");
  attachClickCounter("myButton3");
</script>
```

## What's Happening

1. Each time you call `attachClickCounter("myButtonX")`:

- A new variable `count = 0` is created **just for that button**.
- The event listener function “remembers” it through **closure**.

2. So:

- Button 1 has its own count.
- Button 2 has its own count.
- Button 3 has its own count.
- They don’t interfere with each other.

## Output Example (when you click)

```
bash

myButton1 clicked 1 times
myButton1 clicked 2 times
myButton2 clicked 1 times
myButton3 clicked 1 times
myButton1 clicked 3 times
```

Each button remembers its count — totally independent.

## Why This Pattern Is Powerful

- ✓ Keeps your global scope clean (no global `count` variables)
- ✓ Reusable — one function works for many buttons
- ✓ Each button maintains its own internal state
- ✓ Great for scalable code (you can later use it with loops or dynamic buttons)

### Bonus: Attach Listeners Dynamically for All Buttons in One Go

If you have many buttons and don't want to call the function manually:

```
js

document.querySelectorAll("button").forEach((btn) => {
  attachClickCounter(btn.id);
});
```

This loops over all buttons and applies the same counter behavior automatically 

Would you like me to show a **version with event removal** too (like a “remove listener” button to stop counting)?



yes

Awesome 💪 Vipul — this is where you start understanding *real-world* event control in JavaScript — **adding and removing event listeners dynamically**.

Let's take your reusable counter example and now add the ability to **remove** the event listener when needed 

## Full Example with “Remove Listener” Feature

```
html

<button id="myButton1">Button 1</button>
<button id="removeBtn1">X Remove Button 1 Listener</button>
<br><br>

<button id="myButton2">Button 2</button>
<button id="removeBtn2">X Remove Button 2 Listener</button>

<script>
  // Reusable function that returns the handler, so we can remove it later
  function attachClickCounter(buttonId) {
    let count = 0;
    const button = document.getElementById(buttonId);

    if (!button) {
      console.warn(`Button with id "${buttonId}" not found!`);
      return;
    }

    button.addEventListener("click", () => {
      count++;
      console.log(`Button ${buttonId} has been clicked ${count} times.`);
    });
  }
</script>
```

```
// The handler function
const handleClick = function () {
  count++;
  console.warn(`#${buttonId} clicked ${count} times`);
};

// Attach the event
button.addEventListener("click", handleClick);

// Return the handler so it can be removed later
return handleClick;
}

// 1 Add event listeners
const handler1 = attachClickCounter("myButton1");
const handler2 = attachClickCounter("myButton2");

// 2 Add remove buttons functionality
document.getElementById("removeBtn1").addEventListener("click", function () {
  document.getElementById("myButton1").removeEventListener("click", handler1);
  console.warn("Removed listener from Button 1 ✓");
});

document.getElementById("removeBtn2").addEventListener("click", function () {
  document.getElementById("myButton2").removeEventListener("click", handler2);
  console.warn("Removed listener from Button 2 ✓");
});
</script>
```

## Explanation (in simple words)

### Step 1: Attaching

When you call:

```
js

const handler1 = attachClickCounter("myButton1");
```

- It creates a **private variable** `count` for that button.
- It defines a `handleClick` function that logs and increments `count`.
- It attaches that function as a listener.
- It **returns** the function reference (`handleClick`).

### Step 2: Removing

When you later call:

```
js

document.getElementById("myButton1").removeEventListener("click", handler1);
```

- The browser removes **that exact listener**.
- After this, clicking Button 1 will no longer log anything.

👉 Removal only works **if you pass the exact same function reference** you used in `.addEventListener()`.

That's why we stored `handler1` and `handler2` — otherwise you can't remove them.



## Output Example

css

```
Button 1 clicked 1 times
Button 1 clicked 2 times
Removed listener from Button 1 ✓
Button 1 clicked (nothing happens now)
```

## ⚙️ When to Use This Pattern

Situation	Why Useful
You want temporary event listeners	Helps clean up after use
Avoid memory leaks in single-page apps (like React, Vue)	Prevents old listeners from hanging around
Control user interaction (enable/disable button logic)	Makes UI more dynamic

## 💡 Bonus Tip

If you ever write this in a **React or modern JS project**, the cleanup part happens automatically via `useEffect`'s **cleanup function**, which works just like `.removeEventListener`.

Would you like me to show how the **same logic would look in React (with useEffect)** — so you can relate it to your real project?



YES

Perfect 🔥 Vipul — this is where your frontend dev knowledge and your JS fundamentals *merge together!*

Let's take everything you just learned (event listeners, closures, and removal) and see **how React's useEffect** handles this in a clean, declarative way.

## 🧩 Example: React Version of the Same Event Listener Logic

### ✓ Code

```
jsx

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

export default function ButtonWithCounter() {
  const countRef = useRef(0); // 🧠 Like "let count = 0" but it remembers value across renders

  useEffect(() => {
    const button = document.getElementById("myButton");

    function handleClick() {
      countRef.current += 1;
      console.warn("Button clicked", countRef.current, "times");
    }

    // ✅ Add event listener when component mounts
    button.addEventListener("click", handleClick);

    // ✅ Cleanup function → removes listener when component unmounts
    return () => {
      button.removeEventListener("click", handleClick);
      console.warn("Listener removed ✅");
    };
  }, []); // Empty dependency → runs only once (like componentDidMount)

  return <button id="myButton">Click Me</button>;
}
```

## 🧠 Let's Break This Down (Line by Line)

Code	What It Does
<code>useRef(0)</code>	Creates a persistent value (like a private variable) that survives re-renders.
<code>useEffect(..., [])</code>	Runs once when the component mounts (like calling your <code>AddEventListener()</code> once).
<code>button.addEventListener("click", handleClick)</code>	Attaches the listener just like before.
<code>return () =&gt; {   removeEventListener(...) }</code>	Cleanup function — React automatically calls it when the component unmounts or effect re-runs.
<code>countRef.current</code>	Holds the current count — similar to your closure variable <code>count</code> in vanilla JS.

## □ Why React Does It This Way

React wants your component to:

1. **Attach** listeners only when the component exists on screen.
2. **Remove** listeners automatically when the component disappears — prevents memory leaks.

So, React's `useEffect` cleanup acts like your:

```
js

removeEventListener("click", handler);
```

but it's automatic ✓

## Real-World Example (Multiple Buttons)

Here's a **reusable React version** similar to your earlier "attachClickCounter" function:

```
jsx

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

function ClickCounterButton({ id, label }) {
  const countRef = useRef(0);

  useEffect(() => {
    const button = document.getElementById(id);

    function handleClick() {
      countRef.current += 1;
      console.warn(`#${label} clicked ${countRef.current} times`);
    }

    button.addEventListener("click", handleClick);

    return () => {
      button.removeEventListener("click", handleClick);
      console.warn(`Removed listener from ${label}`);
    };
  }, [id, label]);

  return <button id={id}>{label}</button>;
}

export default function App() {
  return (
    <div>
      <ClickCounterButton id="btn1" label="Button 1" />
      <ClickCounterButton id="btn2" label="Button 2" />
    </div>
  );
}
```

Each button:

- Has its own `countRef`
- Keeps its count private
- Cleans up automatically when unmounted

## Summary Table

Concept	Vanilla JS	React Equivalent
Add listener	<code>addEventListener()</code>	Inside <code>useEffect()</code>
Remove listener	<code>removeEventListener()</code>	Inside cleanup <code>return () =&gt; { ... }</code>
Private variable	<code>let count</code> in closure	<code>useRef()</code>
Setup once	Function call or script load	<code>useEffect([], [])</code>
Cleanup automatically	Manual	Automatic by React

🧠 So, React `useEffect = AddEventListener() + RemoveEventListener() + closure — but handled safely by React!`

Would you like me to show one **React version using state (`useState`) instead of `useRef`** — so you can see how it behaves differently (re-rendering the UI with count)?



yes

Perfect 🌟 Vipul — now you'll see the **difference between `useRef` and `useState`** in React when working with event listeners.

You already know that `useRef` stores a value **without causing re-render**, while `useState` **updates the UI** each time it changes.

Let's visualize both sides step by step 👇

## 🧩 Example: React Version Using `useState`

```
jsx

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

export default function ButtonWithStateCounter() {
  const [count, setCount] = useState(0); // 🧠 This causes re-render whenever count changes

  useEffect(() => {
    const button = document.getElementById("myButton");

    function handleClick() {
      // ✅ Update count → causes React to re-render and show new value
      setCount((prev) => prev + 1);
      console.warn("Button clicked", count + 1, "times");
    }

    // Attach listener
    button.addEventListener("click", handleClick);

    // Cleanup
    return () => {
      button.removeEventListener("click", handleClick);
      console.warn("Listener removed ✅");
    };
  }, [count]); // ⚡ runs whenever count changes

  return (
    <div>
      <button id="myButton">Click Me</button>
      <p>Clicked {count} times</p>
    </div>
  );
}
```

## 🧠 Step-by-Step Breakdown

Line	What's Happening
<code>useState(0)</code>	Creates a <i>state variable</i> ( <code>count</code> ) that re-renders the component when changed.
<code>setCount((prev) =&gt; prev + 1)</code>	Increments the value safely using the previous value.
<code>&lt;p&gt;Clicked {count} times&lt;/p&gt;</code>	React updates this text each time the state changes.
<code>useEffect</code>	Adds a listener when mounted, removes it on cleanup.

## ⚠ Problem with [count] Dependency

Notice how the effect depends on `count`:

```
js
useEffect(() => { ... }, [count]);
```

That means **each time count changes**, the effect runs again:

1. It removes the old listener.
  2. Adds a new listener.
- ✓ Works fine for small apps  
✗ But slightly inefficient if there are many listeners.

## □ Optimized Version (Run Once)

To avoid re-adding listener every time, we can run the effect only **once** — and still update the count safely using the “updater” form of `setCount`:

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

export default function ButtonWithStateCounter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const button = document.getElementById("myButton");

    function handleClick() {
      // ✓ Updater form → uses latest value even if effect doesn't re-run
      setCount((prev) => prev + 1);
    }

    button.addEventListener("click", handleClick);

    return () => {
      button.removeEventListener("click", handleClick);
    };
  }, []); // ✓ run only once

  return (
    <div>
      <button id="myButton">Click Me</button>
    </div>
  );
}
```

```

    <p>Clicked {count} times</p>
  </div>
);
}

```



## Output Example

Clicks	What you see
1st click	"Clicked 1 times"
2nd click	"Clicked 2 times"
3rd click	"Clicked 3 times"

## ⚙️ useRef vs useState — Quick Comparison

Feature	useRef	useState
Causes re-render	✗ No	✓ Yes
Updates visible UI	✗ No	✓ Yes
Keeps value between renders	✓ Yes	✓ Yes
Best for	Internal counters, DOM access	Displayed values, UI state
Example usage	Track clicks silently	Show clicks to user



## In Simple Words

- Use `useRef` when you just need to *remember something internally* (like a closure in vanilla JS).
- Use `useState` when you want the **UI to update** when that value changes.

Would you like me to show a **side-by-side demo** (both `useRef` and `useState` in the same component) to clearly *see the difference* between UI re-render and internal counter?



you said that updating count in useState re-renders ui than why my b variable count is not increasing on screen when i am updating it in useeffect :

```

const [count, setCount] = useState(0);
// 🧠 This causes re-render whenever count changes
var b = 101;
useEffect(() => {
  const button = document.getElementById("myButton");

  function handleClick() {
    // ✓ Updater form → uses latest value even if effect doesn't re-run
    setCount((prev) => prev + 1);
  }
})

```

```

}

button.addEventListener("click", handleClick);
b++;

return () => {
  button.removeEventListener("click", handleClick);
};

}, [count]); // ⚠️ runs whenever count changes

return (
<div>
  <button id="myButton">Click Me</button>
  <p>Clicked {count} times</p>
  <p>B: {b} times</p>
</div>
);

```

Excellent observation 🔥 Vipul — this shows you're really *understanding React's rendering behavior!*  
 Let's break down **why count updates on screen but b doesn't**, even though you're incrementing both.

## 🧩 Your Code Summary

```

jsx

const [count, setCount] = useState(0); // ✓ React state
var b = 101; // ✗ Normal variable

useEffect(() => {
  const button = document.getElementById("myButton");

  function handleClick() {
    setCount((prev) => prev + 1); // ✓ React knows this changed
  }

  button.addEventListener("click", handleClick);
  b++; // ✗ This changes only a local variable (not state)

  return () => {
    button.removeEventListener("click", handleClick);
  };
}, [count]);

return (
<div>
  <button id="myButton">Click Me</button>
  <p>Clicked {count} times</p>
  <p>B: {b} times</p>
</div>
);

```

## 🧠 Step-by-Step Explanation

### 🧩 1. React Renders the Component

- Every render creates a **new function execution** of your component.
- Inside that function, `var b = 101;` runs again.
- 👉 So **b always resets back to 101** at the start of each render.

## 2. useEffect Runs After Render

- React finishes rendering, then runs your **useEffect**.
- Inside **useEffect**, you do `b++`.  
But remember:
  - That **b** only exists **inside this render's function call**.
  - After this effect finishes, **b** disappears.
  - On the next render, React calls your component function again → new `b = 101`.

So visually:

```
bash

Render 1 → b = 101 → useEffect runs → b++ (b=102) → effect done
Click → setCount triggers re-render
Render 2 → b = 101 again
```

That's why the displayed **b** never increases — it keeps resetting to 101 every time React re-renders.

## 3. Why count Works

**count** works because it's **state**, managed by React internally.

When you call:

```
js

setCount(prev => prev + 1);
```

React:

- Stores the updated value in memory (not in your component variable)
- Triggers a **re-render**
- Passes the latest **count** back into your component function

So **count** persists between renders, while **b** does not.

## Visual Diagram

Render Cycle	What Happens	count	b
Initial Render	<code>b = 101</code>	0	101
Click #1 → setCount	React re-renders	1	101 again

Render Cycle	What Happens	count	b
Click #2 → setCount	React re-renders	2	101 again

## Fix: Use useRef for Persistent Non-UI Values

If you want **b** to **keep increasing** but not re-render the UI each time, use **useRef**:

```
jsx

const [count, setCount] = useState(0);
const b = useRef(101);

useEffect(() => {
  const button = document.getElementById("myButton");

  function handleClick() {
    setCount((prev) => prev + 1);
    b.current++;
    console.warn("B value:", b.current);
  }

  button.addEventListener("click", handleClick);

  return () => {
    button.removeEventListener("click", handleClick);
  };
}, []);

return (
  <div>
    <button id="myButton">Click Me</button>
    <p>Clicked {count} times</p>
    <p>B: {b.current} times</p>
  </div>
);
```

👉 Now:

- **b.current** persists across renders
- It increments each click
- UI updates only when **count** changes (since that's state)

## ⚙️ Summary

Concept	useState	useRef	Normal variable (var, let, const)
Persists between renders	✓ Yes	✓ Yes	✗ No
Causes re-render when changed	✓ Yes	✗ No	✗ No
Resets on every render	✗ No	✗ No	✓ Yes
Best for	UI updates	Storing mutable values	Temporary values

### 🧠 In simple words:

- **var** → forgets its value after each render

- **useRef** → remembers without re-render
- **useState** → remembers *and* re-renders the UI

Would you like me to show a **side-by-side demo** where both **useRef** and **useState** counters are displayed together so you can visually see how one updates UI and the other doesn't?



```
const [count, setCount] = useState(0);
// 🧠 This causes re-render whenever count changes
const [base, setBase] = useState(100)
useEffect(() => {
  const button = document.getElementById("myButton");

  function handleClick() {
    // ✅ Updater form → uses latest value even if effect doesn't re-run
    setCount((prev) => prev + 1);
  }

  button.addEventListener("click", handleClick);
  setBase((prev) => prev + 1);
  // setBase(base + 1);

  return () => {
    button.removeEventListener("click", handleClick);
  };
}, []); // ⚠️ runs whenever count changes

return (
  <div>
    <button id="myButton">Click Me</button>
    <p>Clicked {count} times</p>
    <p>B: {base} times</p>
  </div>
);
```

when this is :

```
setBase((prev) => prev + 1);
```

base prints 102 on first render

when done this

```
setBase(base + 1);
```

base prints 101 which is right than why above on eprints 102 on first render

Excellent 🙌 Vipul — this is a *classic React behavior* question, and you've caught one of those **subtle differences** between the *functional updater form* and the *direct state setter form*.

Let's go step by step carefully. 👇

## Your Code (simplified)

```
jsx

const [count, setCount] = useState(0);
const [base, setBase] = useState(100);

useEffect(() => {
  const button = document.getElementById("myButton");

  function handleClick() {
    setCount((prev) => prev + 1);
  }

  button.addEventListener("click", handleClick);

  // 👇 this line executes once (since dependency array is [])
  setBase((prev) => prev + 1); // OR setBase(base + 1);

  return () => {
    button.removeEventListener("click", handleClick);
  };
}, []); // ✅ runs only once
```

## Let's Understand Each Case

### ◆ Case 1: `setBase(base + 1)`

```
js

setBase(base + 1);
```

- When React first renders:
  - `base = 100`
  - So you're literally doing → `setBase(100 + 1) → setBase(101)`
- React updates `base` to **101**
- ✅ Output: **101**

Everything as you'd expect.

### ◆ Case 2: `setBase((prev) => prev + 1)`

```
js

setBase((prev) => prev + 1);
```

- Here, you're using the **functional updater form**.
- React calls your function with the **most recent value of base**.

But here's the key:

React might **batch** state updates that happen during the same render cycle — especially during mount.

So when your **useEffect** runs after the initial render, React has already prepared the next render. By the time your effect executes:

- React may already have the **initial state (100)** committed,
- but the effect's **setBase(prev => prev + 1)** uses **the latest committed value (100)**
- So it sets it to **101**

Then React re-renders.

 But because of how **React's Strict Mode** works (in development), effects sometimes **run twice** on mount to help catch bugs.

That's the missing piece!

In **React Strict Mode (default in Next.js and CRA)**:

Every **useEffect** runs **twice on mount** (once for simulation, once real).

## So What's Happening Exactly:

Step	What Happens	Value of base
Initial render	base = 100	100
useEffect runs first time	<b>setBase(prev =&gt; prev + 1) → 101</b>	101
React's strict mode runs effect again (dev only)	<b>setBase(prev =&gt; prev + 1) → 102</b>	102

 That's why you see **102 in dev mode**.

If you **disable Strict Mode** or run a **production build**, it'll show **101**, same as the other version.

## Confirm It Yourself

Try adding this log:

```
js

useEffect(() => {
  console.warn("Effect ran");
  setBase((prev) => prev + 1);
}, []);
```

You'll see "Effect ran" printed **twice** in your console if you're in development mode → proof that Strict Mode is running it twice.



## In Short

Version	Why it prints that value
<code>setBase(base + 1)</code>	Uses the <i>initial variable value (100 + 1 = 101)</i>
<code>setBase(prev =&gt; prev + 1)</code>	Runs twice in Strict Mode → 100 → 101 → 102
(Production mode)	Both give the same result → 101



## Bonus Tip

You can confirm Strict Mode in your app by checking your root component:

```
jsx

<React.StrictMode>
  <App />
</React.StrictMode>
```

Remove `React.StrictMode` temporarily if you want to test *production-like* behavior — though don't remove it permanently; it's good for catching bugs.



## Final Answer Summary

Mode	<code>setBase(base + 1)</code>	<code>setBase(prev =&gt; prev + 1)</code>
Development (Strict Mode)	101	102
Production	101	101

Would you like me to show visually (a log-based timeline) of how React's **Strict Mode double-run** works — so you can see it step by step with console logs?