

Memory leaks and how to avoid them

Memory Leaks in JavaScript — Explained in Detail

A **memory leak** occurs when memory that is no longer needed **isn't released**, causing your application to consume more memory over time. This leads to slow performance, crashes, or browser tab freezes.

How JavaScript Memory Works

JavaScript uses **automatic garbage collection**:

- It allocates memory when variables/objects are created.
- It automatically reclaims memory when there are **no references** to an object.

A memory leak happens when **objects remain referenced** even though they're no longer needed.

Common Causes of Memory Leaks

1. Global Variables

```
function createLeak() {  
  leak = "I am a global leak"; // `leak` is implicitly global  
}  
createLeak();
```

✅ **Fix:** Always use `let`, `const`, or `var`.

2. Forgotten Timers or Intervals

```
setInterval(() => {  
  console.log("Still running...");
```

```
}, 1000); // runs forever unless cleared
```

✅ **Fix:** Clear intervals when no longer needed.

```
const id = setInterval(...);  
clearInterval(id);
```

3. Closures Holding References

```
function outer() {  
  const bigObject = { data: new Array(1000000).fill("leak") };  
  return function inner() {  
    console.log(bigObject);  
  };  
}  
  
const leaky = outer(); // bigObject is retained by closure
```

✅ **Fix:** Ensure closures don't hold unnecessary references.

4. Detached DOM Elements

```
const div = document.getElementById("myDiv");  
div.remove(); // DOM removed, but...  
someArray.push(div); // still referenced
```

✅ **Fix:** Nullify references after DOM removal.

```
someArray = [];
```

5. Event Listeners Not Removed

```
function setup() {  
  const el = document.getElementById("btn");
```

```
el.addEventListener("click", () => console.log("clicked"));  
}
```

If the element is removed but the listener isn't, memory leaks.

✅ **Fix:** Remove listeners when done:

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

✅ Best Practices to Avoid Memory Leaks

Practice	Why It Helps
Declare variables with <code>let/const</code>	Avoid unintended globals
Clear intervals/timeouts	Prevent infinite memory usage
Dereference objects	Helps garbage collector reclaim memory
Clean up event listeners	Avoid memory retention by DOM elements
Use WeakMap/WeakSet	Automatically removed when object is GC'd
Avoid large in-memory caches	Only cache what's necessary

🔍 Detecting Memory Leaks

1. Chrome DevTools

- Open DevTools → **Memory tab**
- Take snapshots → Look for increasing object counts

2. Performance Monitoring Tools

- Lighthouse
- Heap snapshot diff tools

🔧 Example: Fixing a Leak

❌ **Bad**

```
const items = [];

function create() {
  const el = document.createElement("div");
  document.body.appendChild(el);
  items.push(el); // never removed
}
```

✓ Good

```
function create() {
  const el = document.createElement("div");
  document.body.appendChild(el);

  // Clean up after use
  setTimeout(() => {
    el.remove();
    // Dereference
    el = null;
  }, 1000);
}
```

Summary

Cause	Fix
Global variables	Always declare with <code>let</code> , <code>const</code> , <code>var</code>
Forgotten timers/intervals	Clear them when done
Closures holding refs	Avoid keeping large objects in closures
Detached DOM elements	Remove both DOM and JS references
Event listeners	Remove when elements are no longer in the DOM

Want a blog-style version of this with diagrams or real-world case studies? Let me know!