Garbage Collection (GC) and Memory Leaks in JavaScript

Garbage Collection is responsible for freeing up memory occupied by objects that are no longer accessible.
Modern JavaScript engines like V8 use mark-and-sweep algorithms to identify unused objects.

Mark-and-Sweep Algorithm

- **Mark phase**: The GC starts from a root (e.g., window or global scope) and "marks" all objects that can still be accessed.
- **Sweep phase**: It deallocates memory for objects that are not marked as accessible.

```
let x  = {"name": "Shaik"};
x = null; // will be garbage collected as x is no longer referencing the object
```

Memory Leaks

A memory leak occurs when a JavaScript program holds onto references to objects that are no longer needed, preventing the garbage collector from freeing up memory.

Common Causes of Memory Leaks:

1. **Global variables (unintentionally created)**
    - Unintentionally creating variables in the global scope keeps them alive for the lifetime of the application.

    ```
    function createLeak() {
        leak = "This is a global variable"; // No 'let', 'const', or 'var' makes it glol
    }
    ```

2. **Uncleared timers (setInterval/setTimeout)**
    - References to closures or objects in setInterval or setTimeout can prevent memory from being freed.

    ```
    let obj = { name: "leak" };
    setInterval(() => console.log(obj.name), 1000); // obj is never cleared
    ```

3. **Unremoved event listeners**

```
const btn = document.getElementById("btn");
function handleClick() { console.log("Clicked"); }
btn.addEventListener("click", handleClick);
// the listener still exists in memory
```

Global Variables and Memory Leaks

A global variable is one that is declared outside of any function or simply without the `var` , `let` , or `const` keywords, making it accessible from any part of the code, regardless of the scope. It is attached to the global object, which is `window` in a web browser or `global` in Node.js.

Issues with Global Variables:

- Too many global variables pollute the Global Execution Context, leading to memory leaks and performance degradation.
- Variable collisions can occur, making debugging difficult.

Preventing Memory Leaks

- **Use `let` or `const` to avoid accidental global variables**.
- **Clear timers** with `clearInterval` or `clearTimeout` .
- **Remove event listeners** using `removeEventListener` when they are no longer needed.
- **Use weak references** (`WeakMap` and `WeakSet`) where appropriate to prevent unintended memory retention.

By managing memory efficiently and avoiding common pitfalls, developers can ensure optimal performance and prevent applications from becoming slow or crashing due to excessive memory consumption.