Memory management and garbage collection

Let's explore **Memory Management and Garbage Collection** in JavaScript — crucial for writing efficient and performant applications.

🧠 What Is Memory Management?

Memory management is the process of:

1. **Allocating memory** when variables, objects, or functions are created.
2. **Using** that memory while the program runs.
3. **Freeing** (releasing) memory when it's no longer needed.

In JavaScript, this is done **automatically** through **garbage collection**, but understanding how it works is key to avoiding memory leaks and performance issues.

📦 Types of Memory

1. Stack Memory

- Used for **primitive values**: numbers, strings, booleans, `null`, `undefined`, `symbol`, and `bigint`.
- Fast and managed automatically.

```
let x = 42; // stored in stack
```

2. Heap Memory

- Used for **reference types**: objects, arrays, functions.
- More complex to manage.

```
let obj = { name: "Abhi" }; // stored in heap
```

♻️ What Is Garbage Collection (GC)?

Garbage collection is the process of **reclaiming memory** occupied by objects **no longer reachable** in the program.

JavaScript engines (like **V8**, used in Chrome and Node.js) **automatically** identify and clean up unused memory.

🔍 Reachability

A value is **reachable** if it can be accessed in some way (directly or indirectly) from the **root**.

🌳 Roots Include:

- Global variables
- Local variables in currently executing functions
- Closures
- DOM references in the current page

If no references to a value exist (i.e., it's **unreachable**), it's marked for garbage collection.

Example:

```
let user = {
  name: "Abhi"
};

user = null;
// The object { name: "Abhi" } is now unreachable → garbage collected
```

🧹 How GC Works: Algorithms

1. Mark-and-Sweep Algorithm (used in most JS engines)

Steps:

- **Mark:** Traverse the object graph from the root and mark reachable objects.
- **Sweep:** Remove all unmarked (unreachable) objects from memory.

2. Reference Counting (used in older engines)

- An object is deleted when its **reference count** drops to 0.
- Problem: **Cyclic references** can't be detected properly.

```
let a = {};
let b = { a: a };
a.b = b;
// Reference cycle → old engines can't collect them
```

Modern engines use **mark-and-sweep** to avoid this issue.

🧠 Common Memory Leaks

1. **Global variables** that should be local.
2. **Closures** holding references to large variables.
3. **Detached DOM nodes** still referenced in JS.
4. **Timers** (`setInterval`) not cleared.
5. **Uncleared event listeners**.

✅ Best Practices

- Use `let` and `const` for block-scoped variables.
- Avoid unnecessary global variables.
- Remove event listeners when not needed.

- Clear intervals/timeouts when no longer needed.
- Use tools like Chrome DevTools' **Memory tab** to spot leaks.

🛠️ Debugging Tools

- **Chrome DevTools → Performance & Memory tabs**
- Take **heap snapshots** to inspect memory usage.
- Use **performance profiles** to spot memory leaks.

🔚 Summary

Feature	Description
Memory types	Stack (primitives), Heap (objects/functions)
Garbage collection	Automatic memory cleanup of unreachable values
Main algorithm	Mark-and-sweep
Reachability	Value accessible from global/local/closure scope
Memory leak prevention	Clear intervals, remove unused listeners, avoid globals

---

Would you like a cheat sheet on memory-safe JavaScript patterns next?