

# Garbage collection explanation



in this function heap memory is created and call stack created tha context destroyed and garbage colected :

```
let a = 10;
function check(a) {
  console.log(a);
  let user = { age: a };
}
check(a);
```

but i want to know when and how garbage collection takes place bcause i have read that gc runs at interval how and whe that interval decides

Excellent question, Vipul 🙌 — you're diving into a deep but super-important part of how JavaScript engines like **V8 (used in Chrome and Node.js)** manage memory. Let's break this down **step-by-step** in a simple, beginner-friendly but technically accurate way.

## Recap of Your Example

```
js

let a = 10;

function check(a) {
  console.log(a);
  let user = { age: a };
}

check(a);
```

## What Happens Step-by-Step

1. `a = 10` → A number **10** is stored in stack memory.
2. `function check(a)` → Function definition (object) is stored in **heap memory**, with a reference to it stored in the **stack**.
3. When you call `check(a)`:
  - A **new execution context** is created and pushed onto the **call stack**.
  - Inside that context:
    - The parameter `a` (value **10**) is stored on the **stack**.
    - A new object `{ age: a }` is created in the **heap**, and a reference to it is stored in the **local variable user** on the **stack**.
4. The function executes `console.log(a)` → logs **10**.
5. The function finishes → execution context is **popped off the call stack**.

6. Now, `user` (a local variable) goes **out of scope**.

Since **no other references** to `{ age: 10 }` exist, it becomes **garbage (unreachable)**.

## 💡 So When Does Garbage Collection (GC) Happen?

Here's the key part — **not immediately** after the function ends.

The **GC (Garbage Collector)** runs **automatically** but **not at fixed time intervals**.

It runs **based on internal heuristics** decided by the JavaScript engine (like Chrome's V8).

Let's explain how that works 🤔

## ⚙️ How the GC Decides When to Run

The GC doesn't use a timer.

Instead, it uses **memory pressure + heuristics**.

Think of it like this:

| Event                           | What GC Thinks                        |
|---------------------------------|---------------------------------------|
| Heap memory usage is increasing | "I should clean soon."                |
| System memory is low            | "Better free up some space."          |
| CPU is idle                     | "Now's a good time to clean quietly." |
| New allocations fail            | "Urgent cleanup now!"                 |

So it's **adaptive**, not periodic.

There's **no fixed interval** like "every 5 seconds."

## 🧩 How the GC Works (Simplified Overview)

Modern JS engines like **V8** use a **Generational Garbage Collector**, mainly divided into two spaces:

### 1. New Space (Young Generation)

- For **short-lived objects** (like your `{ age: 10 }`).
- Cleaned frequently and quickly using a "**scavenge**" algorithm (copying live objects to a new space and clearing the old).
- Since your object is created and discarded quickly, it'll be collected here almost immediately after GC runs.

### 2. Old Space (Old Generation)

- For **long-lived objects** that survive several GCs.
- Cleaned less often using **mark-and-sweep** or **mark-compact** algorithms.
- Used for things like large data structures or global objects that persist.

## ⌚ When Exactly Will `{ age: 10 }` Be Collected?

Let's apply that here:

1. After `check()` finishes, `{ age: 10 }` is **unreachable**.
2. GC **does not run immediately**, but the object is **marked as collectible**.
3. The **next time GC runs** (usually within milliseconds or seconds, depending on system state), it will **detect** that no reference points to `{ age: 10 }` and **remove it from the heap**.

You **cannot predict** the exact moment — it's entirely up to the **engine's optimization**.

## If You Want to See It in Action

You can open Chrome DevTools → **Memory tab** → Take heap snapshots before and after a function call to **observe** memory allocation and release.

Or, you can even **manually trigger** GC for debugging (in Node.js):

```
bash
node --expose-gc
```

Then in JS:

```
js
global.gc();
```

(This is only for debugging — never use it in production.)

## In Summary

| Concept                                 | Explanation   |
|---|---|
| GC runs at interval?                    |  No fixed interval.  |
| GC trigger                              |  Based on memory usage, idle CPU, or low memory.               |
| Immediate cleanup after function?       |  No. Object is just <i>eligible</i> for GC.                    |
| How <code>{ age: 10 }</code> is cleaned |  Collected during the next GC cycle when detected unreachable. |
| Who controls GC                         |  The JS engine (e.g., V8).                                     |

Would you like me to show you a **visual memory timeline** (stack/heap + GC cleanup animation style diagram) for this exact example? It'll make the process even clearer visually.



how to take snap shot which you are saying i want to take and see there please guide

Perfect, Vipul  — that's exactly how you *truly understand* what's happening behind the scenes. Let's walk step-by-step through **how to take and read heap snapshots** in Chrome DevTools so you can

see when your `{ age: 10 }` object gets garbage collected.

## Step-by-Step: Taking a Heap Snapshot in Chrome

### What You'll Need

- Use **Google Chrome** browser.
- You can test this directly in **Console** or inside a small **HTML file**.

### Step 1: Open DevTools

1. Open any webpage (or a blank tab like `about:blank`).

2. Press

**Windows/Linux:** `Ctrl + Shift + I`

**Mac:** `Cmd + Option + I`

3. Go to the **Memory** tab on the top bar of DevTools.

(If you don't see it, click the "»" icon and choose *More tools → Performance & Memory*.)

### Step 2: Write Your Test Function

Go to the **Console** tab and type this:

```
js

let a = 10;

function check(a) {
  console.log(a);
  let user = { age: a };
}

check(a);
```

Now the object `{ age: 10 }` exists briefly in heap memory.

### Step 3: Take the First Snapshot (Before GC)

1. Go back to the **Memory** tab.
2. In the dropdown, choose "**Heap snapshot**" (default option).
3. Click "**Take snapshot**."

 You'll now see all currently alive objects in memory.

### Step 4: Run the Function Again and Take Another Snapshot

Now go to **Console** again, run the function one more time:

```
js
```

```
check(a);
```

Then immediately take **another snapshot** (click the “Take snapshot” button again).

## ✓ Step 5: Force Garbage Collection and Take Third Snapshot

Now in the **Console**, run:

```
js
```

```
window.gc && window.gc();
```

⚠ If that doesn’t work, launch Chrome with GC exposed (for accurate manual GC):

- Close Chrome completely.
- Run from terminal:

```
bash
```

```
chrome --js-flags="--expose-gc"
```

- Then open your test page again.

Now rerun:

```
js
```

```
window.gc();
```

After this, take **a third snapshot**.

## ✓ Step 6: Compare Snapshots

In the **Memory tab**, you can:

- Compare **Snapshot 1 → Snapshot 2 → Snapshot 3**.
- Click “Comparison” view.
- Look for the object type like **object** or something named like your variable (**user** if it appears).
- You’ll notice that after GC, the object `{ age: 10 }` will **disappear** (no longer referenced).

This proves it was garbage collected.

## 📊 Optional: Watch It Live with Allocation Sampling

If you want to *watch live memory allocation*:

1. In the **Memory tab**, select “**Allocation sampling**” instead of “Heap snapshot.”
  2. Run your function multiple times.
  3. You’ll see memory allocations and automatic cleanup (drops in memory usage).
- 

### Pro Tip

You can also use the **Performance tab** → **Record** → **Memory** and then execute your function while recording.

You’ll see memory spikes (object created) and drops (garbage collected).

---

### Summary

| Step | Action  |
|------|---|
| 1    | Open DevTools → Memory tab                              |
| 2    | Run your JS function                                    |
| 3    | Take Snapshot 1   |
| 4    | Run again → Take Snapshot 2                             |
| 5    | Force GC ( <code>window.gc()</code> ) → Take Snapshot 3 |
| 6    | Compare snapshots → See object gone                     |

---

Would you like me to give you a **ready-made HTML page** you can open in Chrome, which creates that object and lets you click buttons like “Run Function,” “Force GC,” and “Take Snapshot” for easy visual learning?