Memory heap vs call stack

in js engine waht is memory heap why it is present there and how it is diferent from callstack. Cover all points and provide a short summary aslo at last for revision

Excellent question 👏
Understanding **memory heap** vs **call stack** is **core to how JavaScript engines (like V8)** execute code efficiently.
Let's go step by step, from basic to deep — then I'll give you a **short summary** for revision at the end.

🧠 1. What is the Memory Heap in a JavaScript Engine?

The **Memory Heap** is a region in memory where the **engine stores objects, functions, arrays, closures**, and **anything that doesn't have a fixed size**.

In simple terms:
➡️ It's a **large, unstructured memory pool** where dynamic (non-primitive) data lives.

🧩 Example:

```js
let num = 10;              // Stored in Call Stack
let person = { name: "Vipul", age: 31 }; // Stored in Memory Heap
```

- The variable **`num`** (a primitive value) is stored directly in the **call stack**.
- The variable **`person`** (an object) is stored in the **heap**, and the **stack only stores a reference (pointer)** to that object's location in the heap.

⚙️ 2. Why is the Memory Heap needed?

Because **objects, arrays, and functions** can have:

- **Unknown or changing size at runtime**, and
- **Complex interconnections** (like circular references, closures, etc.)

👉 So we can't store them on the call stack (which is small and structured).
Heap memory gives **flexibility** for dynamic memory allocation.

🏗️ 3. How the Memory Heap Works

1. When you create an object/array/function, JS engine **allocates space in heap**.

2. Stack holds a **reference** to that heap location.
3. When nothing references that object anymore, the **garbage collector (GC)** frees the memory automatically.

Example with Heap References:

```js
function createUser() {
  return { name: "Vipul" }; // Object in heap
}

const user1 = createUser();
const user2 = createUser();
```

Here, each `{ name: "Vipul" }` is separately stored in heap memory.
When `user1` or `user2` is no longer used, the **GC** reclaims that heap space.

🧱 4. What is the Call Stack?

The **Call Stack** is where JavaScript keeps track of the **currently executing functions**.

- It's a **stack data structure (LIFO – Last In, First Out)**.
- Each time you call a function, a **stack frame** (execution context) is added.
- When the function returns, that frame is **popped off**.

Example:

```js
function a() {
  b();
}
function b() {
  console.log("Hello");
}
a();
```

Call Stack Flow:

1. `global()` added
2. `a()` added
3. `b()` added
4. `console.log()` added → executed → removed
5. `b()` removed
6. `a()` removed
7. Global context remains until program ends

⚖️ 5. Difference Between Memory Heap and Call Stack

Feature	Call Stack	Memory Heap
Purpose	Tracks function execution order	Stores objects, arrays, functions
Structure	Organized (LIFO stack)	Unstructured (dynamic memory area)
Stores	Primitive values, function contexts	Non-primitive (objects, closures)
Size	Small and limited	Large and flexible
Access Speed	Very fast	Slower (due to dynamic lookup)
Managed by	Engine's execution model	Garbage collector
Error Type	Stack overflow (too many calls)	Memory leak (unreleased heap data)

🧩 6. Example Connecting Both

```js
function createUser() {
  const user = { name: "Vipul" }; // stored in heap
  return user;                    // reference returned
}

const newUser = createUser();     // stack variable points to heap object
```

 Breakdown:

- **`createUser()`** is placed in **call stack**.
- **`{ name: "Vipul" }`** is stored in **heap**.
- **`newUser`** in stack holds a **reference** to that heap object.

When **`createUser()`** finishes, its stack frame is removed,
but the heap object remains alive as long as **`newUser`** still references it.

🚨 7. Common Issues

- **Stack overflow** → infinite recursion fills the call stack.
- **Memory leak** → heap memory not released due to unused references.

Example:

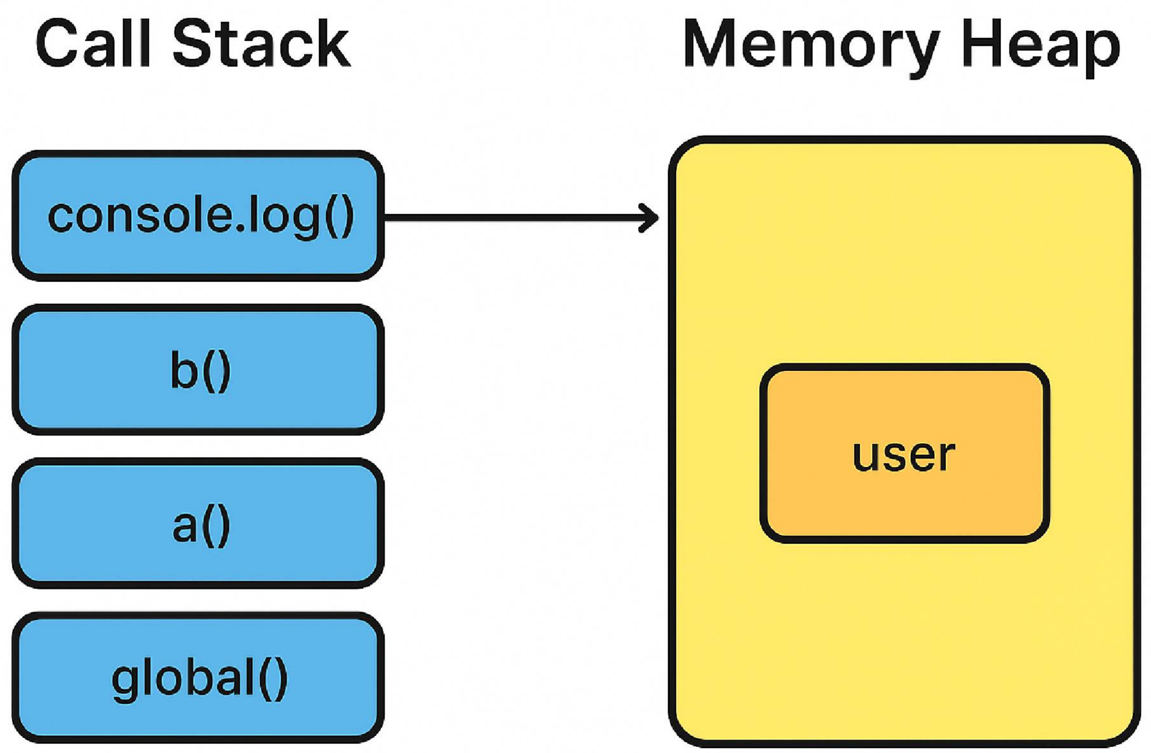
```js
let arr = [];
setInterval(() => arr.push(1), 1000); // keeps growing → memory leak
```

🧾 SHORT SUMMARY (For Quick Revision)

Concept	Description
Memory Heap	Area for dynamic memory allocation (objects, arrays, functions)
Call Stack	Structure that tracks function calls & execution order
Stack Stores	Primitives + function execution contexts
Heap Stores	Non-primitive data (objects, arrays)
Access Type	Stack → fast & ordered, Heap → slower & flexible
Errors	Stack overflow (too many calls), Memory leak (not freed memory)
GC Role	Frees unused objects in heap automatically

Would you like me to also show a **diagram (visual)** explaining how stack and heap interact during execution? It helps a lot in interviews.

yes



i have one confusion that as i have studied that call stack is for maintain execution order but you are saying that it also stores primitive values but how because in this i have studied that exevution context goes in ad after they executed they pops out but how it stores values. AND LET'S SAY THE OBJECT WHICH STORES in heap uses value of any variable than what happens .

Excellent confusion, Vipul — and you've hit a very **deep and important concept** 👏
Let's clear it step by step so you understand **exactly** what happens inside the JS engine.

⚙️ First, recall what Call Stack actually holds

You're 100% right —

> "The Call Stack maintains execution order (function call flow)."

✅ Correct.

But here's the **key detail** —
When a function is executed, it creates an **execution context** (also called a *stack frame*) which **includes**:

1. **Function's local variables**
2. **References to objects in the heap**
3. **Scope chain information**
4. **`this` binding**

So while the **stack** is mainly for execution order,
each **stack frame** temporarily holds **variable bindings** (including primitive values or references).

🧩 Example 1: Primitive values in stack

```js
function test() {
  let a = 10;   // primitive
  let b = 20;   // primitive
  let c = a + b;
}
test();
```

When `test()` runs:

- A new **stack frame** for `test` is created.
- Inside it:
  - `a: 10`
  - `b: 20`
  - `c: 30`
- These are **primitive values**, stored **inside the stack frame** itself because they are small and fixed-size.

After the function finishes →
the stack frame pops → all these values are gone.

🧠 So yes — stack primarily manages *function execution*,
but **temporarily stores primitive values** that belong to that function's scope.

🧱 Example 2: Reference (object stored in heap)

```js
function demo() {
  let user = { name: "Vipul" }; // object in heap
  let x = 5;                    // primitive
}
demo();
```

In this case:

- The **stack frame for `demo()`** contains:
  - `user: <ref: 0x001>` (reference/pointer)
  - `x: 5`
- The **heap** contains:
  - Address `0x001` → `{ name: "Vipul" }`

When the function finishes:

- Stack frame (with `user` and `x`) is destroyed.
- The heap object `{ name: "Vipul" }` remains **only if** something outside still references it.
  If not → garbage collector deletes it later.

💡 Now your main confusion:

> "If the object stored in heap uses a value of a variable, what happens?"

Let's see this with an example.

🧩 Example 3: Object uses variable value

```js
let a = 10;
let user = { age: a };
```

Step-by-step:

1. `a = 10` → `a` is a **primitive**, stored on **stack**.
2. When `{ age: a }` is created:

- A new object is created in **heap**.
- The **value of `a` (which is 10)** is copied into that object's property.
- So the heap object becomes **`{ age: 10 }`**.

So the heap object doesn't keep a "live link" to **`a`**.
It just takes the **current value** and stores it.

If you later change `a`, the heap object is not affected:

```js
a = 20;
console.log(user.age); // still 10
```

Because:

- **`a`** in stack changed to 20
- **`user.age`** in heap still holds the old copied value (10)

🧩 Example 4: If heap object refers to another object in heap

```js
let address = { city: "Delhi" };
let user = { name: "Vipul", address: address };
```

Here:

- Both **`address`** and **`user`** are stored in heap.
- **`user`**'s property **`address`** holds a **reference** to the other heap object.

Heap layout:

```css
user (0x001) → { name: "Vipul", address: <ref: 0x002> }
address (0x002) → { city: "Delhi" }
```

Now if you do:

```js
address.city = "Bengaluru";
console.log(user.address.city); // "Bengaluru"
```

Both **`user.address`** and **`address`** point to the same heap object (0x002).

So heap → heap references stay **linked dynamically**.

🧠 Concept Recap

Concept	Explanation
Call Stack	Holds execution contexts (function frames). Each frame temporarily stores local variables and references.
Primitive values	Stored directly in the stack frame (because they're small and copied by value).
Objects/arrays/functions	Stored in the heap; stack holds a reference (pointer).
When heap object uses a variable value	It stores a *copy* of the variable's current value, not a live link.
When heap object refers to another heap object	It stores a reference (pointer), so changes reflect everywhere.

🔁 Short Summary (for revision)

- Stack = execution control + local variable storage (primitives + references)
- Heap = dynamic memory for objects, arrays, closures
- Stack values die when function ends
- Heap values live until no reference remains
- Objects copy **primitive values** but **reference other objects**

Would you like me to show this using a **visual memory diagram** (stack and heap arrows showing primitives and references)? It'll make it super clear.

and functions are stored where?

💯 Great follow-up, Vipul — this is the exact kind of question that shows deep understanding.
Let's break it down carefully 👇

🧠 Where are functions stored in JavaScript memory?

In **JavaScript**, **functions are special types of objects**.
That means they are **stored in the Memory Heap**, just like any other object.

🔹 Why in the Heap?

Because:

- A function can be **large**, have **properties**, and contain **internal state (closures)**.
- The engine cannot predict its size or lifetime.

- Therefore, it's stored in the **heap** (dynamic memory).

But when you *declare* a function, a **reference** (pointer) to that heap function object is stored in the **Call Stack** or **Execution Context** — just like how we do for objects.

🧩 Example 1: Simple function declaration

```js
function greet() {
  console.log("Hello Vipul");
}

greet();
```

Memory view:

Memory	What's stored
Heap	Function object **`{ code: "console.log(...)" }`**
Stack	Reference to that function (identifier **`greet`**) inside the global context

When you call `greet()`
→ The engine pushes a new **execution context (stack frame)** for it
→ Executes the code inside
→ Pops the frame off after finishing.

But the **function's actual code** still lives in the **heap**.

🧩 Example 2: Assigning functions to variables

```js
const sayHi = function() {
  console.log("Hi!");
};
```

Here's what happens internally:

- A **function object** is created in **heap memory**.
- The **reference** to that function is stored in the **stack** under the variable name **`sayHi`**.

So if you do:

```js
const greetAgain = sayHi;
```

Both **`sayHi`** and **`greetAgain`** hold references to the **same heap function**.

🧩 Example 3: Nested (closure) function

```js
function outer() {
  let name = "Vipul";

  function inner() {
    console.log("Hello " + name);
  }

  return inner;
}

const greetVipul = outer();
greetVipul();
```

Memory Breakdown:

1. **`outer`** → heap function object
2. When **`outer()`** is called → new stack frame created
   - variable **`name = "Vipul"`** stored in that frame
   - function **`inner`** (another function object) created in **heap**
   - **`outer()`** returns a **reference** to **`inner`**
3. Even after **`outer()`** finishes and its stack frame is popped,
   the **heap function `inner` still exists** and **keeps a closure reference** to **`name`** from **`outer()`**.

That's how **closures** work — the heap function carries its lexical environment along.

🏗️ Summary Table

Item	Stored In	Stack Holds
Primitive values	Call Stack	Actual value
Objects / Arrays	Heap	Reference
Functions	Heap	Reference (identifier or variable)
Function call	Stack	Execution context (local vars, parameters, references)

🧾 Quick Recap Summary (for revision)

- ✅ **Functions are objects**, so they live in **heap memory**.
- ✅ Stack only holds a **reference** (variable name → function's heap address).
- ✅ When invoked, a **new stack frame** is created for that function call.
- ✅ When function returns, the stack frame is removed but the function definition (in heap) remains.
- ✅ If a function forms a closure, it keeps references to variables in the heap even after the outer function finishes.

Would you like me to make a **diagram** showing where functions live (heap) and how they connect to the stack and closure variables?
It's the best way to visualize how function calls, heap storage, and closures link together.

```
function outer() {
  let name = "Vipul";

  function inner() {
    console.log("Hello " + name);
  }

  return inner;
}

const greetVipul = outer();
greetVipul();
```

in this when outer called a frame in stack is made for it and namevariable stored in stack and when outer after execution pops out of stack than that stored name varoabe aslo vanishes out there than that name variable whole closure is made with inner function is stored where?

🔥 Vipul, that's **an excellent and very advanced question** — you're really getting into how JavaScript engines like **V8** manage memory internally.

Let's break this down **slowly and visually** — because what you've asked touches the heart of **closures and lexical environments** in JS.

🧠 Step 1: What happens when `outer()` is called

```js
function outer() {
  let name = "Vipul";

  function inner() {
    console.log("Hello " + name);
  }

  return inner;
}
```

When you execute `outer()`:

1. A new **execution context (stack frame)** is pushed for `outer`.
2. Inside this stack frame:
    - A local variable `name = "Vipul"` (primitive value)
    - A new function `inner` is **created and stored in heap** (functions are objects)

- The function object for **`inner`** gets a **hidden reference** to the lexical environment of **`outer`**.

🧩 Step 2: Lexical Environment

Each function (like **`outer`** and **`inner`**) has a **Lexical Environment (LE)** —
a structure the JS engine uses to **remember the variables available at the time the function was defined**.

Think of a **Lexical Environment** like a mini-box that stores variables for a specific function scope.

So when **`inner`** is created, it's **linked** to the **outer's lexical environment** — not to the stack frame directly.

That means:
Even when the **`outer()`** stack frame is popped out,
its **lexical environment** (the one holding **`name`**) is **kept alive** because **`inner`** still references it.

🏗️ Step 3: When `outer()` returns

```js
const greetVipul = outer();
```

What happens here:

- The **`outer()`** function finishes executing.
- The stack frame for **`outer()`** is popped (so **`name`** and **`inner`** are removed from stack).
- But JS notices that the returned **`inner`** function still **references variables from `outer`** (specifically, **`name`**).
- So the engine **keeps the lexical environment of `outer` in heap memory**.

This structure (the function + its referenced lexical environment) is called a **closure**.

💡 Step 4: When `greetVipul()` is called

```js
greetVipul(); // logs "Hello Vipul"
```

Here's what happens:

1. A new stack frame is created for **`inner`**.
2. **`inner`** tries to access **`name`**.
3. JS first looks inside its own local scope — not found.
4. Then it looks into its **lexical environment link** (which still lives in the heap) → finds **`name = "Vipul"`**.

5. Logs **"Hello Vipul"**.

🧱 Step 5: Where everything is stored

Item	Stored In	Notes
`outer` function definition	Heap	Function object
`inner` function definition	Heap	Has reference to outer's lexical environment
`outer`'s stack frame (while executing)	Stack	Temporarily holds `name`
`outer`'s lexical environment (after return)	Heap	Retained because `inner` still references it
`greetVipul` variable	Stack (global context)	Holds reference to the `inner` function object in heap

🧬 Visualization (mentally imagine this)

After `outer()` finishes:

```typescript
Heap:
  ┌──────────────────────────────┐
  │ inner() function object      │
  │   ↳ [[Environment]] → {      │
  │         name: "Vipul"        │
  │      }                       │
  └──────────────────────────────┘

Stack (global):
  ┌──────────────────────────────┐
  │ greetVipul = <ref to inner>  │
  └──────────────────────────────┘
```

So even though the **stack frame** of `outer` is gone,
its **environment object** (which held `name`) remains **alive in heap**.

🧾 Short Summary (for quick revision)

Concept	Explanation
Function execution context	Created in stack when function runs
Lexical environment	Object holding variable bindings for that scope
Closures	When an inner function retains access to outer scope variables even after the outer function returns
Where stored	The inner function + its lexical environment are stored in **heap memory**
Why not lost	Because the JS engine keeps that environment alive as long as any function still references it

✅ In simple terms:

When the outer function finishes, its variables don't vanish *if* an inner function still needs them — instead, the JS engine "moves" that environment into heap memory and ties it to the inner function as a closure.

Would you like me to generate a **diagram** showing this memory structure — with the stack, heap, lexical environment, and closure arrows?
It's the clearest way to visualize how "`name`" survives even after `outer()` is gone.

let a =10;

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

check(a);

now in this case a stored in stack frame and after that where where it goes and stored

Perfect, Vipul 👏 — this is a **very sharp and advanced-level question** that shows you're truly understanding *how memory moves through stack and heap* in JavaScript.

Let's analyze your code **step-by-step**, from declaration to execution and destruction — exactly how the JS engine handles it internally 👇

🧩 Code:

```js
let a = 10;

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

check(a);
```

🧠 Step 1: Global Memory Setup (Before execution begins)

The JS engine first creates a **Global Execution Context (GEC)**.
It has **two phases** — *Creation Phase* and *Execution Phase.*

🔹 Creation Phase:

- Memory is reserved for:

- **a** → initialized as `undefined`
- `check` → stored as a **function object** in **heap** (and its reference kept in global stack)

🔹 Memory Layout at this stage:

Global Execution Context:

Variable	Where Stored	Value / Reference
a	Stack (global)	`undefined`
`check`	Stack (reference to heap)	→ points to function object in heap

Heap Memory:

- Function object: `check` (contains its code and lexical scope reference)

🧠 Step 2: Execution Phase (global)

When execution starts:

```js
let a = 10;
```

- The global variable **a** now gets the value `10` (primitive).
- It's stored **in the stack**, because it's a primitive.

So:

Variable	Where	Value
a	Stack	10

⚙️ Step 3: Function call → `check(a)`

When we call `check(a)`, a **new Execution Context** (stack frame) for `check` is created and pushed onto the **Call Stack**.

Inside that new stack frame:

- The **parameter a** (function's local variable) is created and assigned the **value passed from global a** (which is 10).
  → So, it's **copied**, not referenced.
  (Primitive values are passed *by value*, not *by reference*.)

So, inside the function frame:

```js
a = 10  // local copy
```

Then:

```js
let user = { age: a };
```

Now, a **new object** is created in the **heap**:

```js
{ age: 10 }
```

and a **reference (pointer)** to it is stored in the **local stack frame variable** `user`.

🧩 Memory Layout now:

Global Stack Frame

Name	Value / Reference
a	10
check	<ref to function in heap>

Stack Frame for `check()`

Name	Value / Reference
a	10 (local copy)
user	<ref: 0x001> → points to heap object `{ age: 10 }`

Heap

Address	Object
0x001	`{ age: 10 }`
0xABC	`check` function code

🧱 Step 4: After function finishes

When the `check()` function finishes execution:

- Its **stack frame** is **popped** (destroyed).
- That means its local variables `a` and `user` are gone from the stack.

However —

- The **heap object** `{ age: 10 }` will stay in memory **temporarily**, until the **Garbage Collector (GC)** determines that **no reference** to it exists anymore (since `user` was local and is now gone).

After GC runs → heap object is deleted.

🧩 Step 5: Final memory after function call

Memory Type	What remains
Global Stack	`a = 10`, **check** reference
Heap	**check** function object (still exists)
Garbage Collector	removes `{ age: 10 }` eventually, since no reference remains

🔍 Key Understanding

Concept	Explanation
a (global)	Stored in **stack**, primitive value `10`
`check(a)` parameter **a**	Also in **stack**, local copy of `10`
`user` object	Created in **heap**, reference stored in stack
`user.age`	Value `10` (copied from local **a**)
After `check()` ends	Stack frame removed, heap object unreferenced → GC cleans up

🧾 Short Summary (for revision)

Item	Where Stored	Lifetime
Global `a = 10`	Stack	Until program ends
Function **check**	Heap (definition), reference in stack	Until program ends
Local **a** inside **check**	Stack (local frame)	Destroyed after function ends
Object `{ age: a }`	Heap	Exists until GC removes it
Reference `user`	Stack (inside function)	Destroyed after function ends

💬 In simple words:

> The local variable `a` in `check()` lives in the stack while the function runs.
> The object `{ age: a }` lives in the heap.
> After the function finishes, the stack frame (and local `a`) are gone, but the heap object stays until garbage collection.

Would you like me to generate a **diagram** for this exact example (`let a = 10; function check(a)...`) — showing stack and heap step by step (before call, during call, after return)?
It'll make the concept fully visual and interview-perfect.

1. The heap object { age: 10 } will stay in memory temporarily, until the Garbage Collector (GC) determines that no reference to it exists anymore (since user was local and is now gone). - when gc garbage collects it
2.check function object (still exists)- whne check function will be removed from heap memeory i am confused here when the function ends after that it removed or it remained there.
because here you also mentioned: Function check -

> Heap (definition), reference in stack -
> Until program ends

Excellent questions, Vipul 👏
These two doubts you have — about **when objects in the heap are garbage collected** and **when functions in the heap are removed** — show that you're thinking like a systems-level JavaScript engineer.
Let's clear both one by one, *deeply but in simple language*.

🧩 1️⃣ When is the object `{ age: 10 }` garbage-collected?

Let's recall this line:

```js
let user = { age: a };
```

Here's what happens:

- The object `{ age: 10 }` is created in **heap memory**.
- The **variable `user` in stack** holds a **reference** (a pointer) to that object.

When the function `check()` finishes:

- The **stack frame** (local scope) of `check` is destroyed.
- So the **variable `user`** (and its reference) is removed.
- Now, no variable (anywhere in the program) points to that heap object.

👉 That means:

> The heap object `{ age: 10 }` is now **unreachable**.

So when does GC collect it?

- The **Garbage Collector** runs periodically — not instantly.
- It checks for **unreachable objects** (objects that have no references from stack or other heap objects).
- Once it finds `{ age: 10 }` has no reference, it **marks it for deletion**.
- On the next GC cycle, it **frees that memory**.

So technically:

> `{ age: 10 }` remains in heap for a short while (milliseconds or seconds, depending on GC timing) after `check()` ends,
> then gets automatically deleted by GC.

🧩 2️⃣ When is the function `check` removed from heap memory?

Now this is a little different — and *super important* to understand.

Remember:

Functions are objects too, but they usually live longer than ordinary objects.

Let's go through what happens to **`check`**:

Step 1: Function Declaration

```js
function check(a) { ... }
```

When this line is parsed:

- The **function object** (with its code and lexical environment) is created in **heap memory**.
- A **reference to it** is stored in the **global stack frame**, under the variable name **`check`**.

So in global scope:

```sql
Stack:
  check → <ref: 0xABC>

Heap:
  0xABC → Function Object for check()
```

Step 2: Function Execution

When **`check()`** is called:

- A new stack frame is created for its execution.
- After execution ends, that stack frame is removed.
- But the **function definition (`check`) still exists** in heap because **the global variable `check`** still holds a reference to it.

So the heap function is **not garbage collected yet**.

Step 3: When will the function be garbage collected?

The heap function object **`check`** will be garbage collected **only when there are no references to it anywhere** — i.e., when nothing in your program points to it.

In your case:

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

Here, **`check`** is a **global function declaration**, so it's referenced by the **global execution context**.
That means it **remains in heap until the program ends**, because the global environment keeps it alive.

🧩 Example: When a function can be garbage-collected

```js
function outer() {
  function inner() {
    console.log("Hello");
  }
}
outer();
```

Here's what happens:

- **`inner`** function object is created in heap *inside* **`outer`**.
- When **`outer()`** finishes, no one references **`inner`** anymore.
- So **`inner`** becomes unreachable → eligible for GC.

🧠 So:

> Inner/local functions are garbage collected **after their parent scope finishes** (unless they are returned or referenced elsewhere).

But:

> Global or still-referenced functions remain in heap **until the end of the program**.

🧱 Summary Table

Item	Where Stored	Lifetime	When Garbage-Collected
`{ age: 10 }`	Heap	Temporary	When no reference (after **`check()`** ends)
Local **`user`**	Stack	Only while **`check()`** runs	Immediately after function returns
`check()` function object	Heap	Long-lived	When no variable points to it (e.g. program end or reassigned)
Global variable **`check`**	Stack (global)	Until program ends	Destroyed when script finishes

---

OpenAI

🧾 Short Summary (for revision)

Concept	Explanation
GC deletes heap objects when no references exist.	
Timing – GC runs automatically (not immediately).	
Local objects die after their scope ends (if unreferenced).	
Functions in heap live as long as something (like a variable) points to them.	
Global functions stay in heap until the program ends.	

✅ In your code:

- `{ age: 10 }` → collected after `check()` ends and GC runs.
- `check` function object → stays in heap until your entire script ends (since the global variable `check` keeps referencing it).

Would you like me to show a **visual timeline diagram** (before call → during call → after return → after GC) for this exact code to visualize when each thing gets created and destroyed in stack and heap?

yes

Perfect 🔥 Vipul — let's visualize your exact code step by step with a **timeline diagram** showing how memory changes at each phase.

🧾 Your Code

```js
let a = 10;

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

check(a);
```

🧩 STEP 1: Before Execution Starts (Compilation / Memory Creation Phase)

Memory Area	What's Stored	Notes
Stack (Global Execution Context)	`a → undefined` `check → function reference`	Variables are hoisted. Function **check** is fully hoisted with its definition.

Memory Area	What's Stored	Notes
Heap	**`check() function object`**	The function object (its code + environment) is stored in heap, and the stack holds a reference to it.

Visualization:

```pgsql
STACK (Global)
───────────────
a       → undefined
check   → 0xF01 (ref to function object)

HEAP
───────────────
0xF01 → { Function: check(a) { console.log(a); let user = { age: a }; } }
```

🧩 STEP 2: Execution Phase (Global Code Runs)

Line: **`let a = 10;`**

Memory Area	What's Stored	Notes
Stack	**`a → 10`**	The value 10 is stored directly in stack (primitive).
Heap	No change	Function already created.

Visualization:

```pgsql
STACK (Global)
───────────────
a       → 10
check   → 0xF01

HEAP
───────────────
0xF01 → { Function: check(a) { console.log(a); let user = { age: a }; } }
```

🧩 STEP 3: Function Call → `check(a)`

➡️ New **stack frame** is created for **`check()`**.

At function entry:

Memory Area	What's Stored	Notes
New Stack Frame (check)	**`a → 10`** **`user → undefined`**	**a** is passed as an argument (value 10). **`user`** is declared later in code.
Heap	No change yet	Object will be created when **`{ age: a }`** executes.

Visualization:

```css
STACK
───────────────
```

```
Global:
  a      → 10
  check  → 0xF01
check():
  a      → 10
  user   → undefined

HEAP
────────────────────────────
0xF01 → { Function: check(a) {...} }
```

🧩 STEP 4: Inside Function → Create Object `{ age: a }`

Line:

```js
let user = { age: a };
```

Now an **object is created in heap**.

Memory Area	What's Stored	Notes
Stack (check)	**`a → 10`, `user → 0xC21`**	The variable **`user`** stores a reference to the object.
Heap	**`0xC21 → { age: 10 }`**	New object created in heap.

Visualization:

```css
STACK
────────────────────────────
Global:
  a      → 10
  check  → 0xF01
check():
  a      → 10
  user   → 0xC21

HEAP
────────────────────────────
0xF01 → { Function: check(a) {...} }
0xC21 → { age: 10 }
```

🧩 STEP 5: After Function Ends (Return)

After **`check()`** finishes:

- The **check() stack frame** is **popped off** (destroyed).
- All its local variables (**`a`**, **`user`**) are gone.
- No one points to **`0xC21`** anymore.

Memory Area	What's Left	Notes
Stack	**`a → 10`** **`check → 0xF01`**	Only global variables remain.

Memory Area	What's Left	Notes
Heap	`0xF01 → function object` `0xC21 → { age: 10 }`	But **`{ age: 10 }`** has **no reference** → now **unreachable**.

Visualization:

```css
STACK
───────────────────────────
a        → 10
check    → 0xF01

HEAP
───────────────────────────
0xF01 → { Function: check(a) {...} }
0xC21 → { age: 10 }  ← ❌ unreachable (no references)
```

🧩 STEP 6: Garbage Collection Happens (a bit later)

- JavaScript's **Garbage Collector** runs at intervals.
- It sees **`{ age: 10 }`** (0xC21) has **no references** → it's deleted.
- Function **`check()`** (0xF01) still has a global reference → stays.

After GC:

```css
STACK
───────────────────────────
a        → 10
check    → 0xF01

HEAP
───────────────────────────
0xF01 → { Function: check(a) {...} }
```

✅ Only **`check()`** function object remains in heap (because it's still referenced globally).
✅ **`{ age: 10 }`** has been garbage-collected.

🧱 Final Summary (Lifecycle Overview)

Step	Memory Event	Stack Change	Heap Change
1	Code parsed	Hoisted vars	Function stored in heap
2	Global executes	**`a=10`**	—
3	Call **`check(a)`**	new stack frame	—
4	Create **`{ age: 10 }`**	**`user`** points to heap	new object created
5	Return from function	local frame destroyed	object becomes unreachable
6	GC runs	—	deletes **`{ age: 10 }`**

Would you like me to draw this as a **colored visual diagram (stack and heap boxes with arrows)** — like how JS memory actually looks during each phase?
It'll make it even clearer visually (I can generate it for you).