Closures in JavaScript

The Power of Saved Scope

JavaScript Master Notes

Definition

1️⃣ What is a Closure?

Closure = Function + Lexical Scope (Saved)

It means an **Inner Function remembers** and can access the variables of its **Outer Function**.

> This happens even after the outer function has finished executing.

Think of it as: "The function carries its creation environment with it."

Concepts

2️⃣ Core Concepts

📚 **Lexical Scope:** A function can always access variables from the scope where it was physically written (its surrounding scope).

🔒 **Closure:** When the inner function is returned, it keeps a reference to the outer function's variables alive.

🗑️ **Garbage Collection:** Variables needed by a closure are **preserved** in memory (Heap) and are not automatically cleaned up.

Example 1

Basic Closure Example: The Counter

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count);
  };
}

const counter = outer();
counter(); // 1
counter(); // 2
counter(); // 3
```

💡 Explanation

The `outer()` function finishes, but the `count` variable is still alive.

The `counter` function (which is the returned `inner` function) is a closure, and it continues to update the same `count`.

3️⃣ Why Use Closures?

🛡️ Data Privacy (Encapsulation)

To hide variables from the global scope and only allow access through controlled methods.

🏭 Function Factories

To create specialized functions that share common configuration or state (e.g., creating a `makeMultiplier(x)` function).

🕒 Asynchronous Tasks

Crucial for maintaining state in `setTimeout`, `setInterval`, and **Event Handlers**.

🧩 Module Pattern

Used historically to create clean public interfaces while keeping implementation details private.

Corner Case

4️⃣ Loop Problem: Using var

```
// 1 second later...
for (var i = 1; i ⩽ 3; i++) {
  setTimeout(() ⇒
    console.log(i), 1000);
}

// Output: 4, 4, 4 😱
```

❌ Why?

var is **Function Scoped**. It ignores the loop block.

All three `setTimeout` functions share the **same single reference** to `i`.

By the time they run, the loop has already finished, and `i` is globally set to 4.

Solution

✅ Loop Fix: Using let

```
// 1 second later...
for (let i = 1; i ⩽ 3; i++) {
  setTimeout(() ⇒
    console.log(i), 1000);
}

// Output: 1, 2, 3 🎉
```

✅ Why?

`let` is **Block Scoped**.

The `for` loop creates a **new block scope** (and thus a new closure) for `i` in **each iteration**.

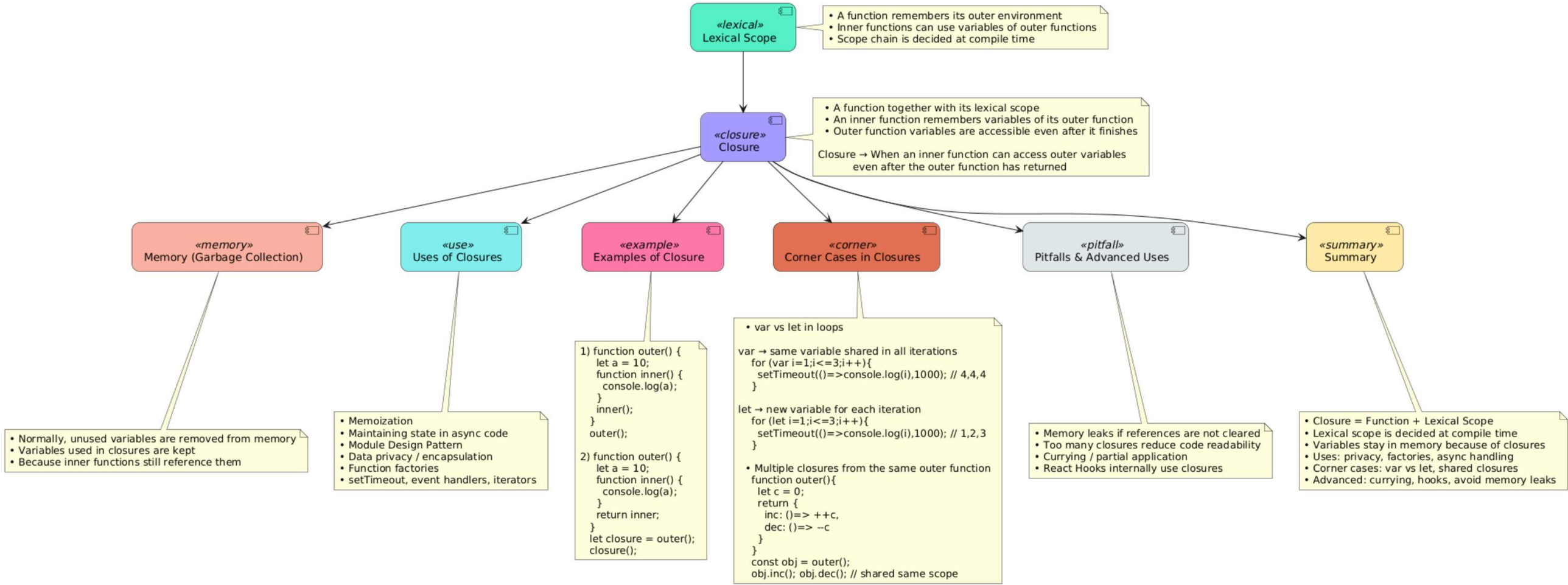
Each closure remembers its specific value of `i` (1, 2, or 3).

Summary

📝 Easy Recap (Interview Ready)

- A closure is a function bundled with its **lexical environment**.
- Outer variables stay **alive** because the inner function still needs them.
- **Memory** is preserved in the Heap (not cleaned up by GC).
- **Key Uses:** Data privacy, creating factory functions, and managing state in async code.

JavaScript: Closures, Lexical Scope, Examples, Corner Cases & Uses



Thank You!

Any Further Questions on Scope?

🔔 Follow for More JavaScript Deep Dives!

By Ali Hassan