Callback Functions

(Execution Control + Async Foundation)

Subtitle: Understand how JavaScript uses functions as callbacks to control timing, sequence, and real-world event-driven behavior.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

🧠 Introduction

Modern JavaScript does not run in a straight line only.

Real applications are:

- event-based (click, input, scroll)
- time-based (delays, intervals)
- async-based (API calls, database fetch)

To handle this, JavaScript uses **callbacks**.

A callback is not just a function — it is a **control mechanism for when code should run**.

⚙️ Step 1: What is a Callback Function?

A callback function is:

A function passed into another function to be executed later.

Simple Example:

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

function processUser(callback) {
  callback("Ali");
}
```

processUser(greet);

Key Idea:

We are not calling the function directly — we are giving control to another function.

🔄 Step 2: Why Callbacks Exist

Without callbacks:

- execution is rigid
- no control over timing
- no event handling

With callbacks:

- code becomes flexible
- execution becomes dynamic
- behavior can change at runtime

Mental Model:

Function A → passes control → Function B → executes later

⚙️ Step 3: Synchronous Callback

Callback executes immediately.

```
function run(callback) {
  callback();
}


run(function () {
  console.log("Executed immediately");
});
```

⏳ Step 4: Asynchronous Callback (Real Power)

JavaScript becomes powerful when callbacks run later.

Example: setTimeout

```
console.log("Start");


setTimeout(function () {
  console.log("Delayed execution");
}, 2000);


console.log("End");
```

Output:

Start

End

Delayed execution

Why this happens:

- JS does not wait
- callback runs after delay
- execution continues

🌍 Step 5: Real-World Callback Uses

1. Event Handling

```
button.addEventListener("click", function () {
  console.log("Button clicked");
});
```

2. API Calls

```
function fetchData(callback) {
  setTimeout(() => {
    callback("Data received");
  }, 1000);
```

}

3. File Processing / Backend Systems

- read file
- process data
- return result via callback

🧠 Step 6: Callback Flow Model

Function called → Callback stored → Event/Delay happens → Callback executed

⚠️ Step 7: Callback Hell (Major Problem)

When callbacks are nested too deeply:

```
doA(function () {
  doB(function () {
    doC(function () {
      doD(function () {
        console.log("Done");
      });
    });
  });
});
```

Problem:

- unreadable code
- hard debugging
- messy structure

This leads to modern solutions:

- Promises
- async/await

⚙️ Step 8: Callback vs Normal Function

Type	Behavior
Normal function	executed directly
Callback	executed by another function

🧠 Step 9: Why Callbacks Matter in Real Systems

Callbacks are everywhere:

- UI interactions
- server responses
- timers
- background tasks

👉 Without callbacks, JavaScript would be static and blocking.

⚠️ Step 10: Common Mistakes

❌ 1. Thinking callback runs immediately always

Not true — depends on context

❌ 2. Over-nesting callbacks

Leads to callback hell

❌ 3. Confusing callback with return

Callback = delayed execution
Return = immediate output

❌ 4. Not understanding async behavior

Most bugs come from this

🧪 Step 11: Mini Exercises

Exercise 1

Create a function that accepts a callback and executes it after 1 second.

Exercise 2

Create a custom function that simulates setTimeout.

Exercise 3

Build a simple login flow using callbacks.

🧠 Step 12: Mini Quiz

1. What is a callback function?
2. Why are callbacks used in JavaScript?
3. What is callback hell?
4. Difference between sync and async callback?

🚀 Step 13: Thinking Upgrade

If you understand callbacks:

- you understand event-driven programming
- you understand async execution
- you are ready for APIs, Promises, and async/await

👉 This is the bridge from basic JS to real-world backend/frontend systems.

📌 Step 14: Summary

- Callback = function passed to another function
- Used for delayed or controlled execution
- Core of async JavaScript
- Enables event-driven programming
- Can lead to callback hell if misused