# Google JavaScript Technical Interview (Callbacks, Promises, Await/Async)

GP Lee
May 19 · 7 min read ★

## Web Developers Tomorrow

Enjoyed this article? If so, get more similar content by **subscribing to our publication!** OR publish your articles by sending your **Medium name**, and a **draft link** to **developers.tomorrow@gmail.com**

## Takeaways from reading this article

*learn following things:*

### 1. Callbacks

- What is a Callback in JavaScript
- Callbacks in real Life

### 2. Callbacks vs. ES6 Promises & Await / Async

- Why Promises
- What are Promises
- How Promises resolves Callback Hell

## Sample problems (one of them is an actual Callback problem I had from the Interview)

**Q1.** Print the letters A, B, C in that order in callback , Promise , Async / Await

**Q2.** Print the array of list [A, B, C] in the order

**Q3.** Print the array of alphabets [A … Z] in the order

**Q4.** Make Callback API Request for a given number of times

## What is a Callback?

## Why do we even need Callback in JavaScript?

- Web Browser is a playground for JavaScript. JavaScript enables interactive web pages along with best friend HTML & CSS
- They are many events in the web browser and JavaScript's role to **response** to the events and show something in action. Events could be user-generated like **mouse click** or **typing**
- The fundamental **reason for a callback** is to run code in response to an event. To register a callback function for an event, you need to be able to pass it to another function, which is responsible for binding the event and callback together

We use **callback** everyday. For example, the example below print `Hello` in response to clicking event to the body of a webpage

## Simple Callback Example

*Even EventListener event function uses a callback*

## Explanation

- In this case, we are passing a function, **"callback"** (this name can be any name; (i.e., clickEventFunct) it is just a function name)to another function **"addEventListener"**
- When **addEventListener** runs, it registers **callback** with the click event

## There are 2 different ways JavaScript is being run here:

**1. The script runs once when the page loads**

- The **body** constant is declared, and a value is assigned to it

- The function **callback** is declared, but **NOT** executed
- The method **addEventListener** is executed (**callback** is passed in to **addEventListener**)
- one of the **addEventListener**'s job is to tell the browser to run **callback** when the click event occurs on the body

**2. The other time JavaScript is run is when an event occurs (in this case click event)**

- This time **callback** is finally executed
- **callback** executes when the body is clicked, and "Hello" is logged to the console
- This can happen more than once, namely whenever the body is clicked

> We now get a sense of why callback is important in JavaScript. Now Let's move onto How we use callback in real life

## Callbacks in real Life

### Let's make a function called loadScript which loads scripts and modules asynchronously

- It appends to the document the new, dynamically created, tag `<script src="...">` with given `src`

- The browser automatically starts loading it and executes when complete

## PROBLEM

- The script is executed "asynchronously", as it starts loading now, but runs laters, when the function has already finished
- If there is any code below `loadScript(...)`, it does not wait until the script loading finish and would **NOT work**

## SOLUTION

- Let's add a **callback** function as a second argument to **loadScript** that should execute when the script loads

- That's the idea : the second argument is a function (usually anonymous) that runs when the action is completed

### Callback in Callback

*Let's go over one more callback example before we move on to ES6 Promise & Async / Await*

- How can we load two scripts sequentially : the first one, and then the second one after it?
- The natural solution would be to put the second `loadScript` call inside the callback

> We now get a sense of how to use callback. Now Let's move onto ES6 Promise

## 2. Callback vs. ES6 Promise

### Why Promise?

- **Pyramid of Doom** or **Callback Hell**
- Callback has native difficulty ; As calls becomes more nested, the code becomes deeper and increasingly more difficult to manage

## What is a Promise?

- **Promise** is introduced in ES6 to resolve the **callback hell** issue and handle asynchronous operations

### Our loadScript function using Promises

- Promise allow us to do things in the natural order; run `loadScript(script)` and `.then` we write what to do with the result
- We can call `.then` on a Promise as many times as we want

## How Promise resolves Callback Hell

## 3. Sample problems

### (Q1 — *Callback) Print the letters A, B, C in that order

### printString function to print string along with callback function

First, make a **printString** function that print (console.log) the string and **callback** parameterized function with interval (**setTimeout**) of 1 seconds (1000 ms)

## Let's try to print the letter A, B , C in the order

It's 3 nested callback (already callback hell?? I do not know , you tell me 🤞)

- IMPORTANT* If you look at **line 4,** You need to **placeholder** empty function to prevent the error. **printString** function is still looking for parameterized **callback** function

### (Q1 — *Promise) Print the letters A, B, C in that order

### printString function to print string in Promise way

The settled state of Promise is either **resolve** or **reject.** In this example, If the settled state is **resolve (**successful**)** then move onto the next **(.then)** (thenable) and if **reject** (not successful) do something `//not implemented in this case`

## Let's try to print the letter A, B , C in the order

In Promise we use **then** to move onto the next operation **In this case, call another printString()**

## (Q1 — *Await Async) Print the letters A, B, C in that order

### printString function to print string in Await Async way

**Await** is basically syntact sugar for Promise. You can use `then` (thenable) for **Await Async** as well. You can use the same `then` code above to call the function

**USAGE**

**Async Await function**

## (Q2) Print the array of list [A, B, C] in the order

*In the interview, once you solve the problem, the question is always slight revised. Interviewers are often asking*

> "You did great. Now, Let's think about WHAT IF … "

***Q2 is the revised / upgraded version of Q1***

> Instead of hard coded alphabet A B C, your inputs are changed to array of alphabets [A , B , C]. If you solve the Q1 easily, I recommend you to solve by yourself before looking at the solution below

**Instead of string A , B , C , the input value is changed to array**

**Callback way**

**Usage**

## (Q3) Print the array of alphabets [A … Z] in the order

*In the interview, once you solve the problem, the question is always slight revised. Interviewers are often asking*

> "You did great. Now, Let's think about WHAT IF … "

***Q3 is the revised / upgraded version of Q2***

> NOW THIS IS TRICKY PART OF Q1 , Q2

Still, we want to callback multiple times (instead of making callback hell)

## Let's make it optimized

In this example, we will print all alphabets (26 letters A … Z). Instead of making 26 callback hell; we will implement it using **callback within callback (call itself)**

> printAll

**function**

- First we define `arr` which has all alphabets ["A", "B", "C", … "Z"]
- we define `index` , which we will increment by 1 each time after **printString** call `array[index++]`
- In **printString** function we pass 3 parameters (an item from `arr` , index , callback)
- *Most Importantly* we need to call the function `cbOfcb` (line 10)

### printString

**function**

- It works as a mediator between **printAll** function & **callback** function (define below)
- If index is 27, we send **cb** `err`
- else, we pass **cb** `null` & `string`

### callback

**function**

- If err is passed (instead of null), print `done` && return
- If not, print `string (str)`

## (Q4) Make Callback API Request for a given number of times

*In JavaScript, we often API request to the server asynchronously. Sometimes we need to request multiple times (rather than having a false or null from a single request) until we get the data*

## Q4 is a practical version (API Request) of Q3

### request

**function**

- axios post request

**If response is successful**

- **`response.data`** has something ; In this case (response.data == 1). Call **callback**

**If not (else)**

- try **request** again if **retries** is greater than 0
- If no **retries** remaining, call **callback** (Similar to **callback** function above, we check the parameters in this version of **callback** function)

## callback

**function**

- Similar to **callback** function in Q3 we check whether the values in parameters , in this case `data` and `error`

**If error**

- return error

**Else**

- print data

## Thank you!

Google JavaScript Programming Web Development Technology