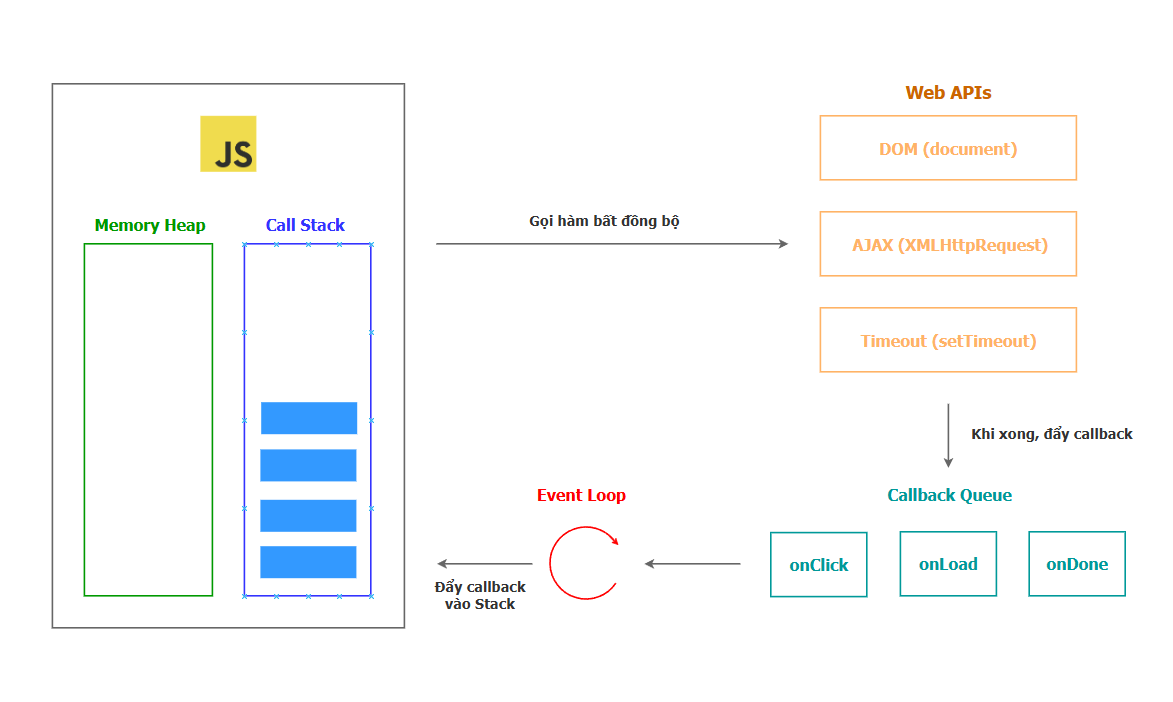
Sơ đồ bất đồng bộ trong JavaScript



* Memory Heap: Nơi lưu trữ biến, object.
* Call Stack: Nơi thực thi mã JS đồng bộ, thực thi từng hàm theo LIFO.
* Web APIs: Các API bất đồng bộ do trình duyệt cung cấp (DOM, AJAX, Timer...)
* Callback Queue = Task Queues: Hàng đợi callback từ Web APIs sau khi hoàn thành.
* Event Loop: Vòng lặp kiểm tra xem Call Stack trống chưa; nếu trống, lấy công việc từ Callback Queue đẩy vào chạy.

Diễn giải luồng hoạt động:

* Hàm bất đồng bộ (setTimeout, AJAX, sự kiện DOM,...) được gọi từ Call Stack và chuyển sang Web APIs xử lý.
* Khi Web API hoàn thành, callback được đẩy vào Callback Queue.
* Event Loop liên tục kiểm tra nếu Call Stack trống sẽ lấy callback từ Queue và đẩy vào Stack để thực thi.

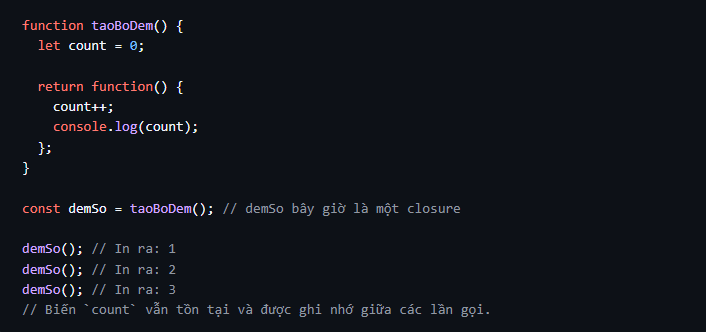
Quy trình lặp lại cho đến khi không còn tác vụ nào.

**Lưu ý:**  
- JavaScript chỉ có 1 Call Stack duy nhất (single-thread), nhưng nhờ cơ chế này có thể xử lý nhiều tác vụ bất đồng bộ mà không bị "đơ".  
- **Callback Queue** ở đây tương đương với **Task Queue** (macro-task) trong các tài liệu mới.  
- Microtask Queue (Promise...) có ưu tiên cao hơn, nhưng không được thể hiện trong sơ đồ này.

Advanced JS

# **I. Closure**

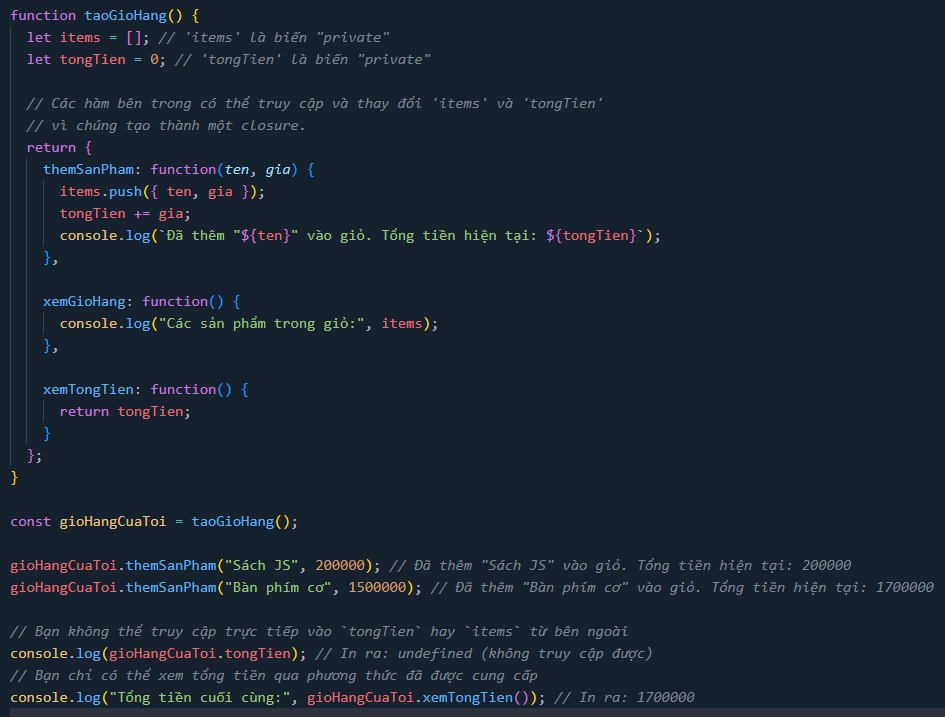
A closure is a function that remembers the scope where it was created. This means it can access the parent function’s variables, even after the parent function has executed.

Example:  


## **1. Applications**

**a. Data Privacy / Encapsulation**

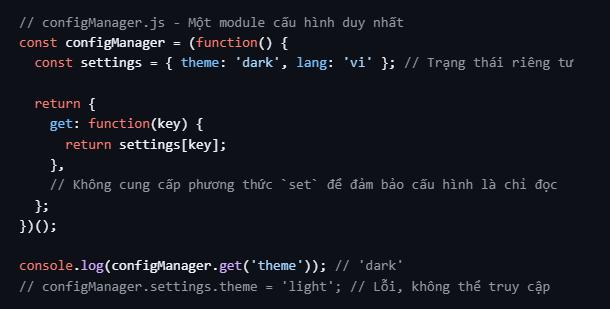
In JavaScript, before **class** with **#private fields** appeared, there was no way to create a truly private variable. Closures allow to simulate this behavior by creating variables that can only be accessed through the methods provided.

Example: Create an order management module that prevents direct modification of the total price.  


With **the introduction of class syntax in ES6**, and especially private class fields (#), there is now **a more official and clear method**for achieving the encapsulation that we previously had to rely on closures for.

But closure still is useful for some scenario:

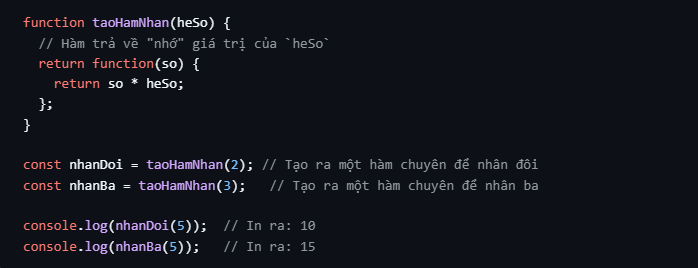
* When follow Functional Programming method
* When creating a simple, independent module, if you only want to create a single private object (like a **Singleton**) and don't intend to create many instances of it, using a closure to create the module is still very clear and productive. For example:



**b. Function Factories**

Closures allow us to create ‘pre-configured’ functions with an initial parameter.

Example: Creating a function that produces other multiplication functions.



Pros: Reusing logic and creating specialized functions flexibly without having to rewrite code.

**c. Currying**

Currying is a functional programming technique that turns a function with many parameters into a sequence of functions, each accepting only one parameter. Closures are the core mechanism that allows currying to work.

Example:   

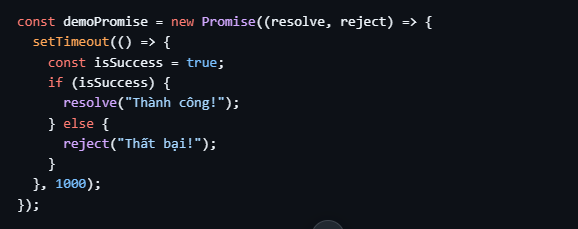

# **II. Promises và Async/Await**

## **1. Promises**

Promises are objects that represent a value that may exist (or an error) in the future, and are often used to handle asynchronous actions. Promises help us write more readable asynchronous code and avoid callback hell.

### a. Promises have 3 main states:

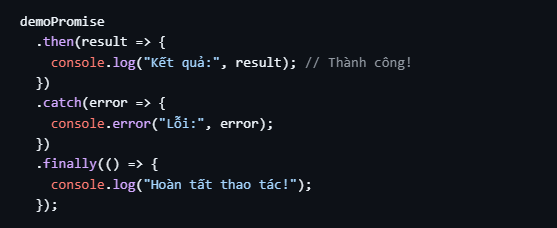
* **Pending**: When the Promise has just been created and has not yet completed or failed.
* **Fulfilled**: When the action succeeds and returns a value.
* **Rejected**: When the action fails and returns a reason (error).

Example:  


### b. Handle Promise’s result

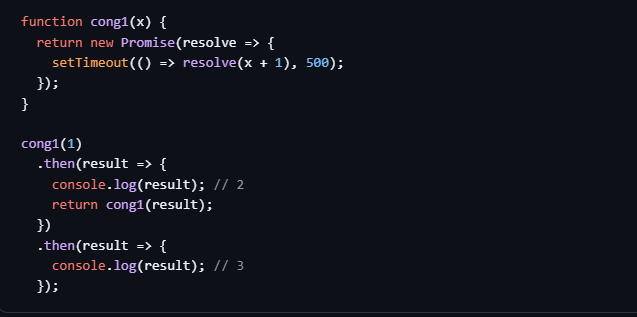
Promise provides the following main methods:

* .then(onFulfilled, onRejected): Xử lý khi thành công (fulfilled) hoặc thất bại (rejected).
* .catch(onRejected): Xử lý khi thất bại (giống như then(null, onRejected)).
* .finally(onFinally): Thực thi khi Promise hoàn thành, dù thành công hay thất bại.



### c. Promise chaining

We can chain multiple asynchronous actions by returning a Promise in a .then().



## **d. Handle parallel and synthesize many Promises**

Static method on the Promise class:

* Promise.all([p1, p2, ...]): Waits for all Promise to complete and returns an array of result. If any Promise is rejected, the entire Promise is rejected.
* Promise.race([p1 p2, ...]):  Returns the result of the first Promise that settles (either fulfilled or rejected).
* Promise.allSettled([p1, p2, ...]): Returns the status and value of all Promises, whether they are fulfilled or rejected.
* Promise.any([p1, p2, ...]): Returns the value of the first fulfilled Promise. If all Promises are rejected, it returns an AggregateError.