BÁO CÁO

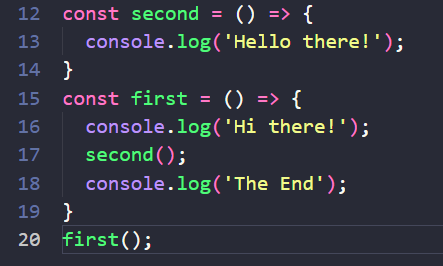
Manipulate: vận dụng, thao tác,

**Synchronous JavaScript:**

### Execution Context: được hiểu là một môi trường nơi js code được đánh giá và thực hiện. Bất cứ code nào được thực thi trong js, nó chạy trong một execution context. Code cục bộ và toàn cục sẽ được thực thi với EC riêng.

### Call Stack: là một ngăn xếp LIFO được sử dụng để lưu trữ tất cả các Excution context được tạo ra trong quá trình thực thi code.

* JavaScript có một ngăn xếp duy nhất vì nó là một ngôn ngữ lập trình đơn luồng. Call Stack có cấu trúc LIFO nghĩa là các mục chỉ có thể được thêm hoặc xóa khỏi đầu ngăn xếp.





1. Khi đoạn code này được thực thi, một execution context toàn cục được tạo ra (biểu diễn bằng hàm main ()) và được đẩy vào đầu ngăn xếp. Khi hàm first () được gọi, nó được đẩy vào đầu ngăn xếp.
2. Tiếp theo, console.log ('Hi there!') được đẩy vào đầu ngăn xếp, khi kết thúc, nó được lấy ra từ ngăn xếp. Sau đó hàm second () được gọi và nó được đẩy vào đầu của ngăn xếp.
3. console.log ('Hello there!') được đẩy vào đầu ngăn xếp là lấy ra khi nó kết thúc. Hàm second kết thúc, do đó nó được lấy ra khỏi ngăn xếp.
4. console.log (‘The End’) được đẩy vào đầu của ngăn xếp và xóa đi khi nó kết thúc. Sau đó, hàm first kết thúc và nó được lấy ra khỏi ngăn xếp.
5. Chương trình thực hiện xong ở đây và execution context toàn cục (main ()) được lấy ra khỏi ngăn xếp.

Blocking:

* Ex: ta có 3 funtion () được khai báo lần lượt với thời gian chạy khác nhau (7s,8s,3s). Sau khi hàm 1th thực hiện rồi đến thứ 2th và 3th điều này có nghĩa các hàm này đang ngăn chặn call stack. Do đó chúng ta không thể thực hiện khác khi code trên đang thực hiện. nếu nhiều hàm chạy quá lâu thì tèo trình duyệt.

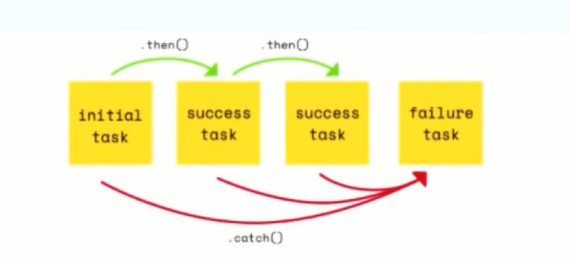
Ta có thể giải quyết vấn đề trên bằng Call back: Funtion a có thể được truyền vào funtion b dưới dạng tham số, funtion a sẽ được funtion b thực thi sau này. Ta có thể xử lý việc này bằng call back nhưng tại sao ta không nên dùng như vậy vì như vậy nó sẽ tạo ra call back hell



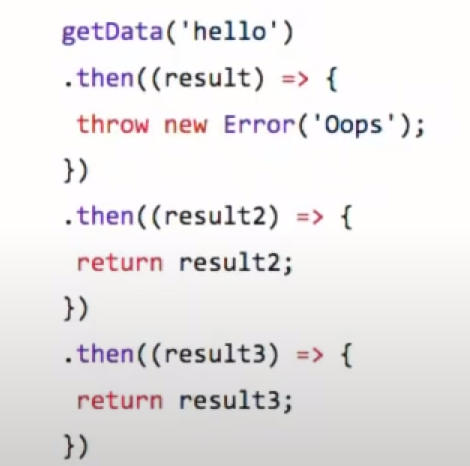
Như thế sẽ rất khó để debug hoặc bảo trì thay vào đó ta sử dụng promise.

Promise: là một kết quả của một hoạt động bất đồng bộ, 1 promise có 3 trạng thái:

1. Pending: Đang chạy.
2. Fulfiled: Chạy xong.
3. Reject: lỗi.



Khi task ban đầu xử lý xong trả ra kết quả 🡺 ta dùng then. Để xử lý task tiếp theo để tránh lồng các call back với nhau, và ta không cần throw err mỗi call back mà chỉ cần dùng reject.



Async/await:

* **Async / Await** là một tính năng của JavaScript giúp chúng ta làm việc với các hàm bất đồng bộ theo cách thú vị hơn và dễ hiểu hơn. Nó được xây dựng trên Promises và tương thích với tất cả các Promise dựa trên API. Trong đó:
* **Async** - khai báo một hàm bất đồng bộ (async function someName (){...}).
  + Tự động biến đổi một hàm thông thường thành một Promise.
  + Khi gọi tới hàm async nó sẽ xử lý mọi thứ và được trả về kết quả trong hàm của nó.
  + Async cho phép sử dụng Await.
* **Await** - tạm dừng việc thực hiện các hàm async. (Var result = await someAsyncCall ().
  + Khi được đặt trước một Promise, nó sẽ đợi cho đến khi Promise kết thúc và trả về kết quả.
  + Await chỉ làm việc với Promises, nó không hoạt động với callbacks.
  + Await chỉ có thể được sử dụng bên trong các function async.

Ta có thể dùng await để lấy kết quả từ một promise nhờ đó ta viết funtion bất đồng bộ mà nhìn như đồng bộ ☺.

