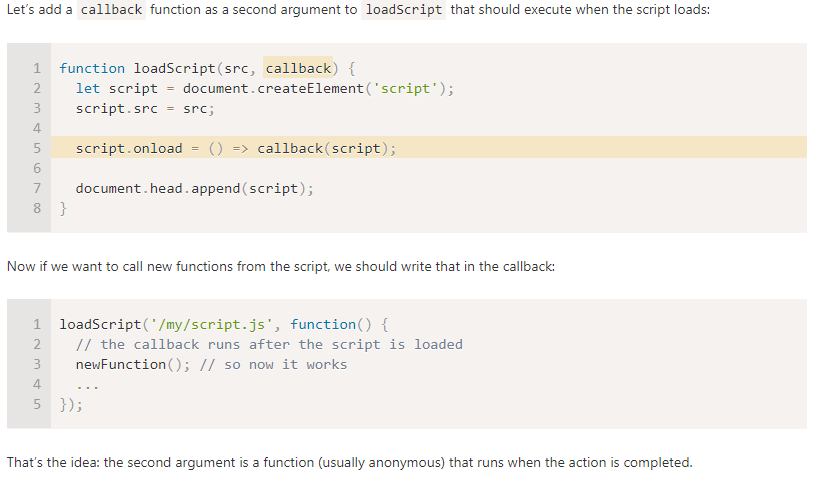
**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Callbacks\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* VD khi load 1 script, nếu muốn dùng 1 hàm trong script đó ta cần phải đợi cho script đó load xong, js ko có cơ chế cho biết khi nào sẽ load xong. Nên để khắc phục ta sẽ dùng callback.

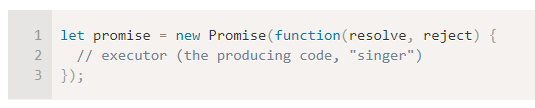


* Lúc này, hàm mà ta muốn thực hiện sẽ được gọi ngay sau khi script load.
* Nếu có error ta có thể làm như sau :



**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PROMISE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* Producing code : là code thực thi việc gì đó nhưng sẽ cần thời gian (xem như ca sĩ)
* Consuming code : là code muốn kết quả của producing code khi nó thực hiện xong, như các hàm cần có kết quả từ hàm khác , (xem như fans)
* 1 Promise là 1 object JS đặc biệt, sẽ kết nối producing code và consuming code lại với nhau. Giống như là danh sách đăng ký, producing code thực thi và trả về kết quả promise, và promise sẽ làm cho kết quả đó chuyển đến với các consuming code.
* Constructor syntax cho promise object như sau :



* Function được truyền vào **new Promise**  được gọi là **executor.**
* Khi **new Promise** được khởi tạo, executor sẽ tự động thực thi.
* Executor chứa producing code nghĩa là cuối cùng nó sẽ tạo ra kết quả.
* resolve và reject là các callback được cung cấp bới chính JS, Code của chúng ta sẽ chỉ nằm bên trong excutor.
* Khi mà executor thu được kết quả, sớm hay muộn thì 1 trong 2 callbacks trên sẽ được gọi.

+ resolve(value) : nếu mọi việc hoàn thanh tốt đẹp, với kết quả là value.

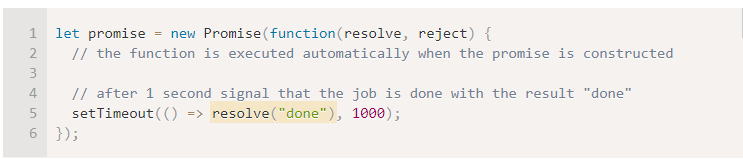
+ reject(error) : Nếu có lỗi xảy ra, error sẽ là error object

* **promise object** được trả về bởi **new Promise constructor**  sẽ có các internal properties như sau :

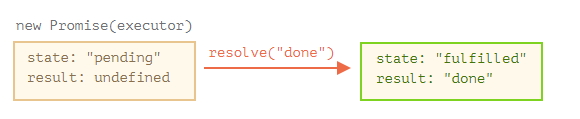
+ **state** : **“pending”** (là giá trị ban đầu), sẽ thay đổi thành **“fulfilled”** **khi resolve được gọi**, hoặc **“rejected” khi reject được gọi.**

+ **result : undefined** (là giá trị ban đầu), sẽ thay đổi thành **value** **khi resolve(value) được gọi** hoặc **error khi reject(error) được gọi**.

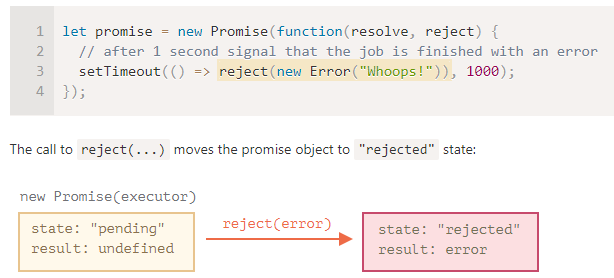
* VD :



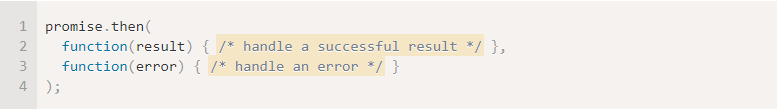
* Ở đây ta giả sử thành công
* Executor được gọi ngay khi bởi new Promise.
* Executor nhận 2 đối số là resolve và reject, đây là 2 hàm được tạo bới JS. Ta sẽ chỉ cần gọi 1 trong 2 khi mọi thứ sẵn sàng.
* Sau 1s, executor gọi resolve(“done”) để tạo ra kết quả. Khi đó object sẽ thay đổi :



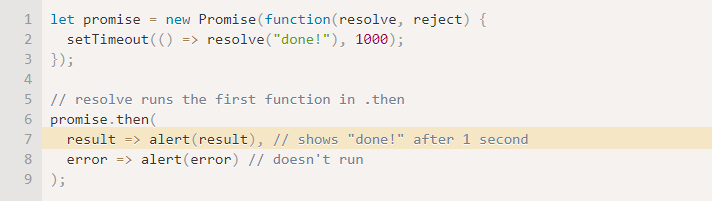
* Ở trường hợp có lỗi :



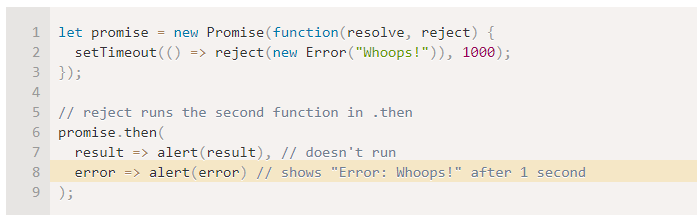
* Lúc này ta thấy cần phải gọi reject.
* Tóm lại, Executor sẽ thực hiện 1 tác vụ và sẽ gọi 1 trong 2 resolve hoặc reject và thay đổi promise object tương ứng.
* 1 promise sau khi resolve hoặc reject được gọi là “settled”.
* NOTE : Exector sẽ chỉ gọi 1 resolve hoặc 1 reject. Việc gọi các resolve hoặc reject sau đó đều sẽ bị bỏ qua. Và reject và resolve cũng chỉ có 1 argument.
* NOTE : Ta chỉ nên dùng Error object khi gọi reject() dù ta có thể truyền vào bất cứ kiểu nào.
* NOTE : resolve và reject vẫn có thể được gọi ngay lập tức mà không phải thực hiện bất cứ tác vụ gì trong executor.
* NOTE : 2 properties là state và result là internal, ta ko thể truy cập trực tiếp mà phải dùng methods .then/.catch/.finally.
* **Consumer Code : then, catch, finally :**
* 1 Promise object là cầu nối giữa executor và consuming function là hàm sẽ nhận value hoặc error. Thông qua các method : .then, .catch, .finally.
* **THEN : SYNTAX**



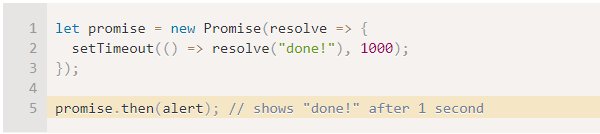
* Argument 1 của then là function sẽ chạy khi promise resolve và nhận kết quả
* Argument 2 của then là function sẽ chạy khi promise là rejected và nhận error



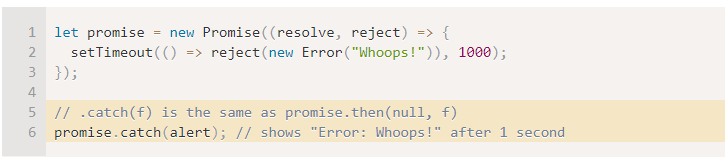
* function đầu tiên được thực thi.



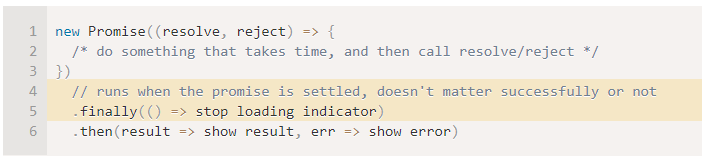
* **function thứ hai được thực thi**
* **Nếu ta chỉ muốn hàm đầu tức chỉ mún chạy khi thành công ta có thê chỉ dùng hàm 1**



* **CATCH : SYNTAX**
* Nếu ta chỉ quan tâm đến error, ta có thể dùng null ở argument đầu tiên trong .then như .then(null, errorHandlingFunction). Hoặc ta có thể dùng .catch(errorHandlingFunction) sẽ thực hiện tương tự :



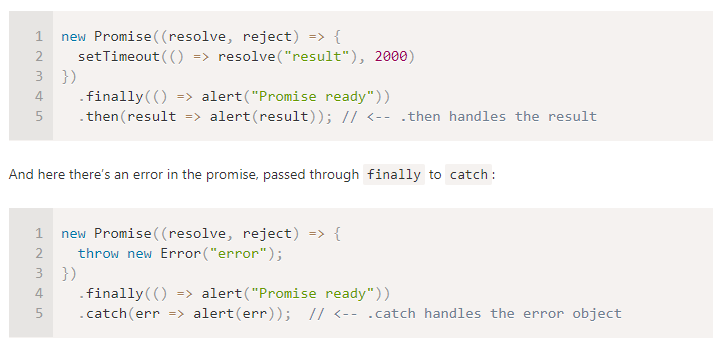
* **FINALLY : SYNTAX**
* Việc gọi .finally(f) tương tụ như .then(f,f) mang ý nghĩa là f sẽ luôn được thực thi khi promise được settled (resolve hoặc reject).
* Finally là method tốt cho thực hiện việc làm sạch. VD như dừng việc thực thi trường hợp nó ko còn cần thiết.



* Finally sẽ không giống hoàn toàn với then :

+ finally sẽ ko có arguments, Với finally ta ko biết liệu promise có thành công hay ko.

+ finally sẽ passes results và errors cho lần handler tiếp theo. Vd :



* **NOTE : khi promise ở state pending thì .then/catch/finally sẽ ở trạng thái đợi. Lúc này, ta có thể dùng handler ở bất cứ đâu và chỉ chờ đổi state.**
* **Về loadScript :**



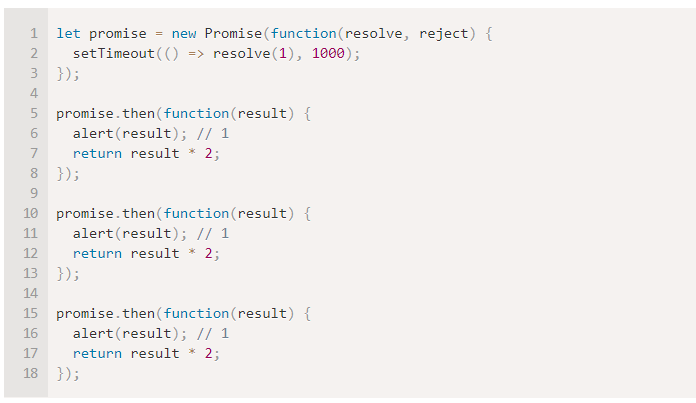


**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PROMISE CHAINING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

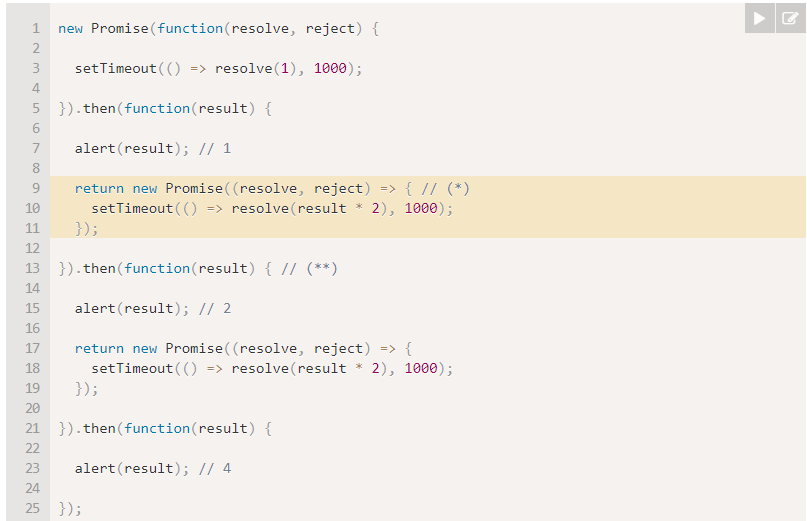
* Khi ta có 1 chuỗi các tác vụ bất đồng bộ cần thực hiện trình tự sau khi tác vụ trước hoàn thành. Promises có thể giải quyết.



* **Ví dụ trên hoạt động là vì : việc gọi promise.then sẽ trả về 1 promise do đó ta có thể gọi .then với nó. Khi 1 handler trả về 1 value nó sẽ trở thành result của promise đó.**
* **NOTE : Về kỹ thuật ta có thể gọi nhiều .then cho cùng 1 promise nhưng đó không phải là chaining. Khác với gọi then liên tục. Vd :**



* Kết quả không trả về cho các lần gọi then sau đó.
* Trong thực tế, ta ít khi dùng nhiều method trên cùng 1 promise mà sẽ hay dùng chaining hơn.
* **Returning promises :**
* 1 handler dùng trong .then(handler) có thể tạo và trả về 1 promise.
* Khi đó các handlers sau sẽ đợi đến khi promise được settles và nhận result.



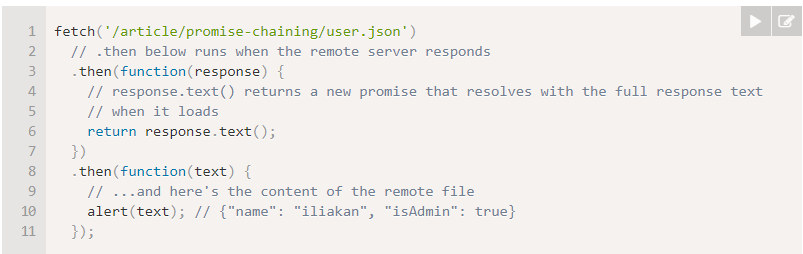
* Giống ví dụ trên nhưng sẽ đợi 1 giây.
* **Ex : loadScript() : Khi load nhiều script như sau :**



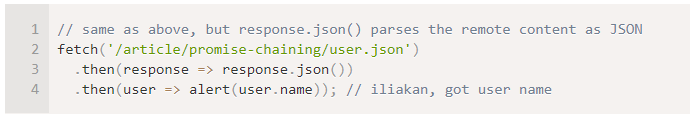
* Mỗi lần gọi loadScript sẽ trả vê 1 promise và lần gọi then tiếp theo sẽ thực thi khi promise resolve. Do đó scripts loaded one after one.
* Lúc này, các nested function có thể access các outer scope.
* **NOTE : 1 cách chính xác, 1 handler có thể không return hoàn toàn là 1 promise mà được gọi là “thenable”. Là 1 object trừu tượng có method then. Được thực hiện như then của 1 promise.**
* **“thenable” thực chất mang ý nghĩa là nếu khi trả về 1 object từ then, nếu nó có 1 method là then thì nó sẽ gọi method cung cấp native function resolve, reject như 1 arguments.**
* **Bigger Example : fetch :**
* fetch() method được dùng để load thông tin từ server, syntax :



* lúc này sẽ gọi 1 network request đến url và return promise. Promise resolve với 1 **response** object khi server phản hồi 1 header nhưng trước khi full response được downloaded.
* Để đọc full response ta sẽ gọi method **response.text()** : **nó sẽ trả về 1 promise mà sẽ resolve khi full text được downloaded từ server.** Với text đó là result.



* Response từ fetch cũng có method response.json() đọc data và parse như JSON.

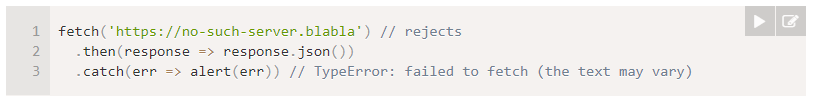


* Ví dụ request lên github :



**\*\*\*\*\*\*ERROR HANDLING WITH PROMISES\*\*\*\*\*\*\*\***

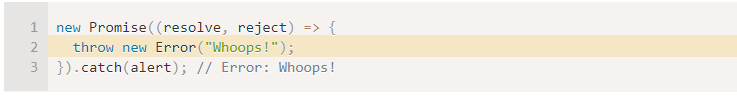
* Khi promise reject, control flow sẽ nhẩy đến rejection handler gần nhất. rất tiện



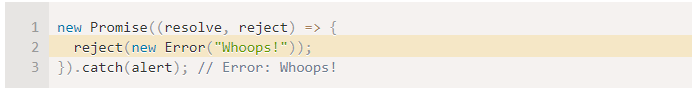
* Ta thấy catch không cần phải viết ngay sau fetch mà có thể cách vài then.
* Cách tốt nhất để bắt các lỗi tối ưu thì catch ta sẽ để ở cuối. Catch sẽ ko bắt được tất cả nhưng nếu có lỗi sẽ bắt được



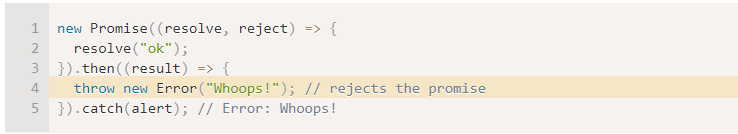
* **Try…catch ngầm :**
* Code của 1 promise executor và promise handlers có 1 try…catch ẩn xung quanh nó. Nếu có 1 trường hợp ngoài mong muốn xảy ra thì nó sẽ được bắt và xem như là 1 rejection.



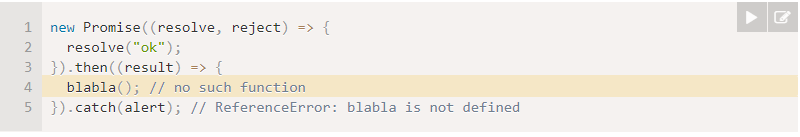
* Tương tự như :



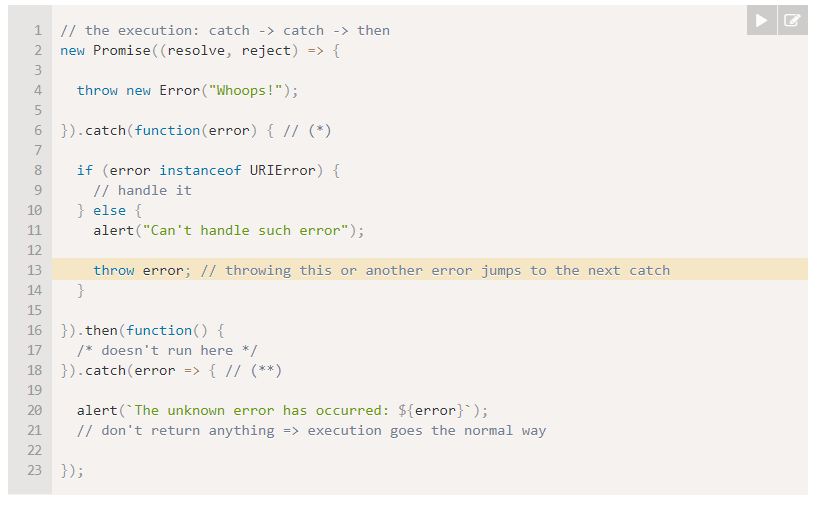
* Try…catch ẩn xung quang executor sẽ tự động bắt error và chuyển nó thành rejected promise.
* Nó cũng hoạt động với handlers. Nếu ta throw bên trong .then handler điều đó có nghĩa là 1 rejected promise sau đó sẽ nhảy đến error handler gần nhất.



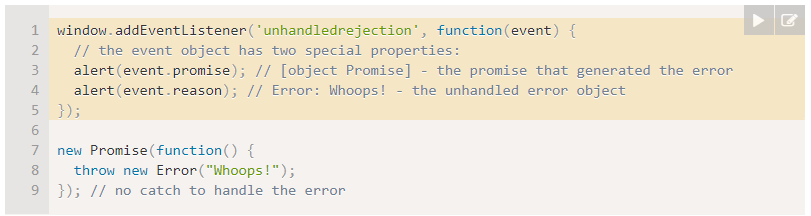
* Và việc bắt lỗi này xảy ra với mọi lỗi không chỉ riêng throw.



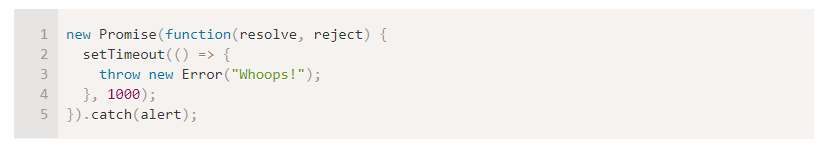
* **RETHROWNING :**
* Ta thấy có thể có nhiều .then và dùng 1 .catch ở cuối để bắt lỗi tất cả của chúng nếu có.
* Nếu ta throw bên trong catch sau đó sẽ chuyển đến error handler gần nhất tiếp theo. Nếu ta bắt lỗi thành công thì sẽ tiếp tục .then kế tiếp



* Ở ví dụ trên, .then sẽ ko thực thi do không bắt được lỗi ở trên
* **UNHANDLED REJECTION.**
* Giả sử nếu ta quên hay không thể bắt 1 lỗi gì đó thì phải làm thế nào ?
* Trong browser có thể bắt lỗi bằng event bằng sự kiện “unhandledrejection” như sau :



* Event này là 1 phần của HTML standard.
* **NOTE : try..catch ẩn sẽ bắt các lỗi 1 cách đồng bộ, nếu ko đồng bộ sẽ ko bắt được. vd :**



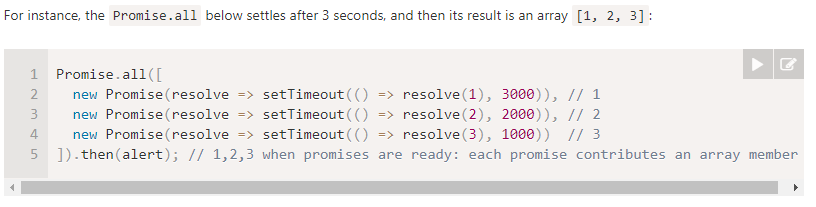
* **Ở đây việc throw là không đồng bộ với executor nên sẽ ko bắt được lỗi.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* PROMISE API\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

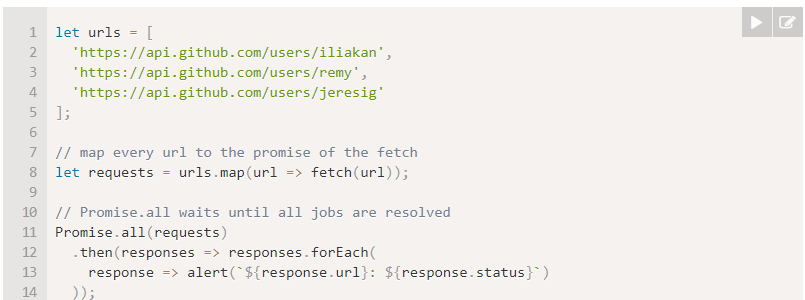
* **Ta có 5 static method trong Promise class.**
* **PROMISE.ALL :**
* Syntax :



* Nếu ta muốn promises thực thi 1 cách song song hay đa thực thi và chờ cho tất cả ready ta sẽ dùng Promise.all
* Promise.all sẽ nhận 1 array của các promises và trả về 1 promise.
* New promise này sẽ resolve khi tất cả các promises trong list đươc settled và array của các kết qủa trở thành result.



* Thứ tự kết quả sẽ giống như thứ tự của promises bất chấp thời gian thực thi.
* **Nếu 1 trong các promises rejected thì promise trả về từ Promise.all sẽ là rejects với error. Vầ bỏ qua các promises khác nhưng các promises khác vẫn sẽ thực thi nhưng sẽ bị bỏ qua.**
* Có 1 trick là dùng map để chuyển array data thành array các promises và wrap lại bằng Promise.all. vd :



* **PROMISE.allSettled :**
* Promise.allSettled sẽ chờ cho tất cả các promises settle. Bất kể result. Array kết quả sẽ là :

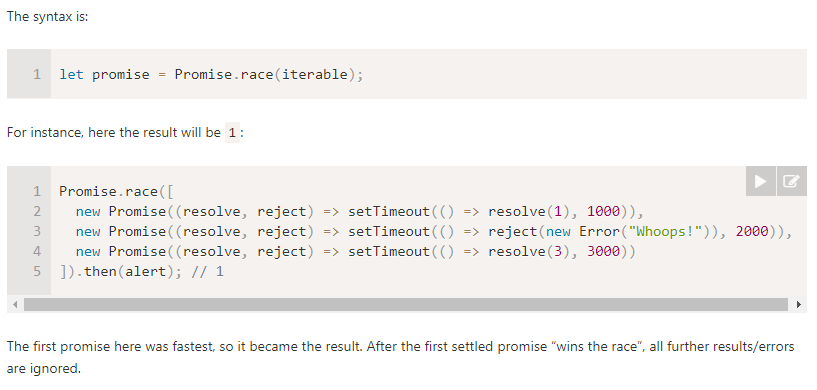
**+ {status: “fulfilled”, value:result} : cho 1 response thành công.**

**+ { status: “rejected”,reason:error } : cho error.**

* **Ví dụ :**



* **Promise.race** : giống như Promise.all nhưng sẽ chỉ đợi đến khi nhận được promise settled đầu tiên và các promises khác sẽ được bỏ qua.



* **Promise.resolve/reject : Ít khi dùng thay vào đó dùng await/async**

**+ Promise.resolve(value) : tạo 1 resolved promise với result là value.**

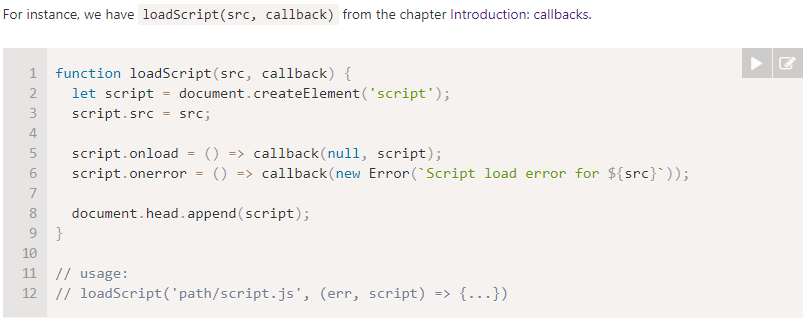
**Let promise = new Promise(resolve => resolve(value));**

**+ Thường được dùng khi function cần return 1 promise.**

**+ Promise.reject(error) : tạo 1 rejected promise với error.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PROMISIFICATION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

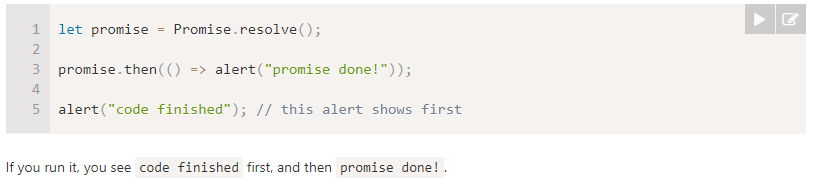
* **Là 1 cách chuyển đổi 1 function chấp nhận 1 callback mà return 1 promise.**
* **Do trong thực tế các hàm và thư viện là các callback-based. Nhưng promise lại tiện lợi hơn.**
* **VD :**





**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MICROTASKS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* **Promise handler .then/.catch/.finally luôn luôn bất đồng bộ**
* **Cho dù Promise ngay lập tức resolve thì code bên dưới .then/.catch/.finally vẫn sẽ execute trước các handlers này.**



* **Tại sao ??**
* **MICROTASKS QUEUE :**
* Asynchronous tasks cần có 1 sự quản lý riêng.
* **ECMA standard** định nghĩa 1 internal queue là **PromiseJobs** hay liên quan đến **microtask queue trong ES8 term.**
* Theo như specification :

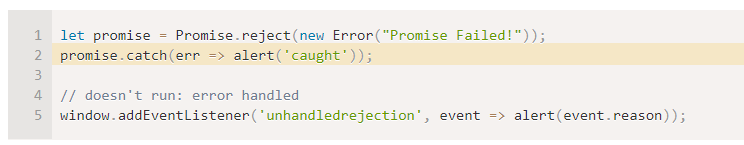
**+** Queue là first-in-first-out : tasks đợi trước sẽ ra trước.

**+** Execution của 1 task sẽ khởi tạo khi không có gì đang chạy.

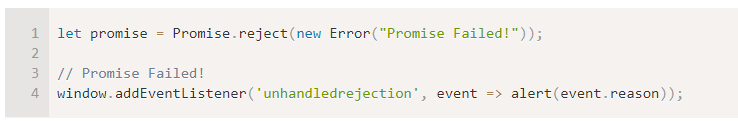
* Nói 1 cách đơn giản , khi 1 promise sẵn sàng thì các .then/.catch/.finally của nó sẽ được đẩy vào trong queue và sẽ chưa execute ngay. Khi JS lấy lại từ do từ current code. Nó sẽ nhận task từ queue và thực thi.
* Đó là lý do code first hiện lên trước ở ví dụ trên.



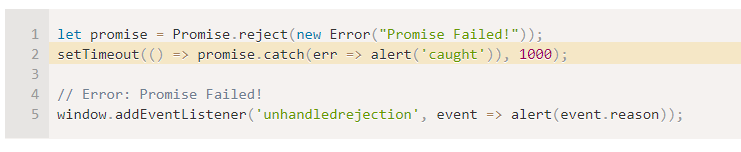
* Promise handlers sẽ luôn đi qua internal queue này.
* Nếu có nhiều handlers liên tiếp(chain) thì mỗi handler sẽ execute bất đồng bộ, nó sẽ được đưa vào hàng đợi và khi current code và handler trước đó hoàn thành thì nó sẽ chạy
* Nếu muốn thực hiện theo thứ tự ta chỉ việc đưa code vào .then
* **UNHANDLED REJECTION :**
* Ở trên ta có sử dụng event unhandledrejection ta sẽ đi chi tiết cách nó bắt lỗi.
* **1 “unhandled rejection” xảy ra khi 1 promise error không được handle tại cuối của microtask queue.**
* Thông thường khi muốn bắt 1 lỗi, ta sẽ đặt .catch vào promise chain.



* Nếu ta quên bỏ catch vào thì lúc sau microtask queue sẽ empty. Và engine trigger event.



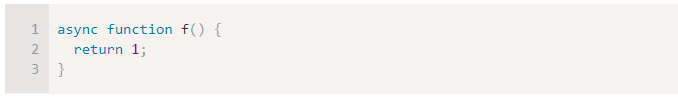
* **Nhưng nếu ta handle error lúc sau thì sẽ ntn :**



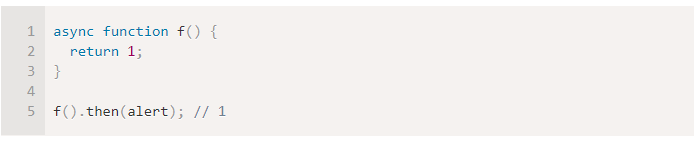
* Khi chạy ta thấy “Promise Failed” trước sau đó là “caught”.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ASYNC/AWAIT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

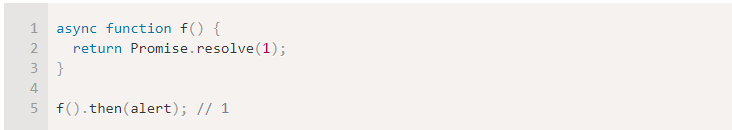
* **ASYNC FUNCTIONS :**
* Ta sẽ sử dụng async keyword, nó được đặt trước function.



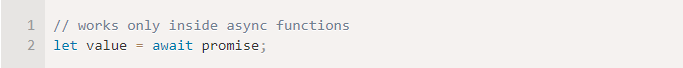
* **Keyword “async” ở trước có nghĩa là 1 function sẽ luôn return về 1 promise. Value được wrapped trong 1 resolved promise 1 cách tự động.**



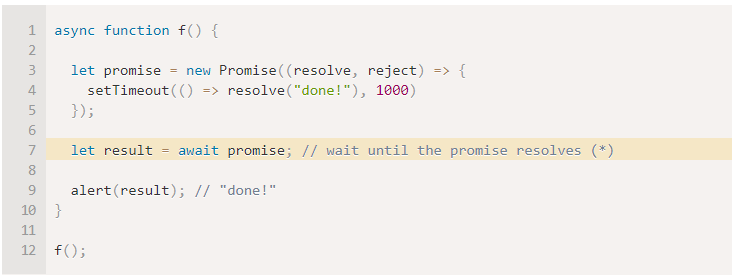
* **Tương tự như sau**



* Tóm lại, async đảm bảo function sẽ trả về 1 promise và wrap non-promise vào nó.
* **AWAIT :**
* **Keywword await sẽ chỉ hoạt động bên trong async function.**



* **Await** khiến JS đợi đến khi promise settle và return result của nó.



* **Function execution sẽ bị tạm dừng tại await và tiếp tục khi promise settle, với result trowe thành result của nó.**
* Await khiến JS đợi đến khi promise settle và tiếp tục khi có result. Việc này không làm mất CPU resources vì engine có thể làm việc khác vào lúc đó : thực thi script khác, handle event.
* Giống như 1 cách viết khác của promise.then
* Ta không thể dùng await ở 1 hàm thông thường.
* Ví dụ :

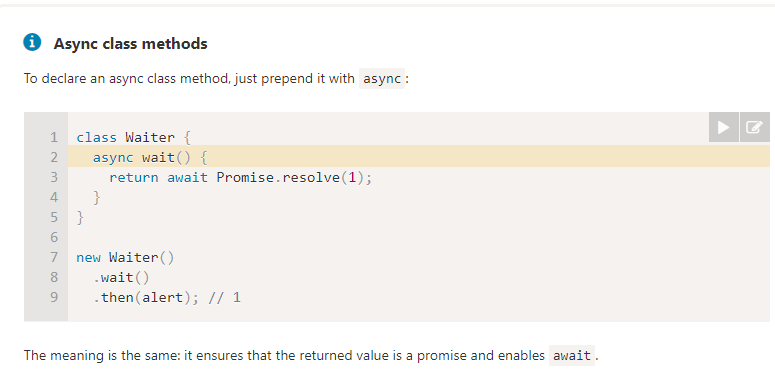


* **NOTE : ta ko thể dùng await ở top-level của code vd :**



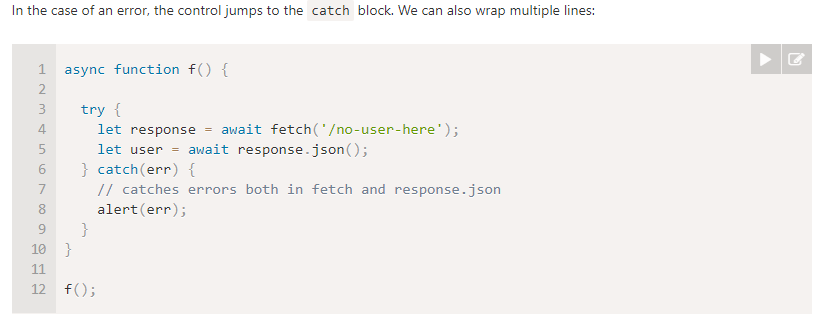
* **Nhưng ta có thể làm như sau :**



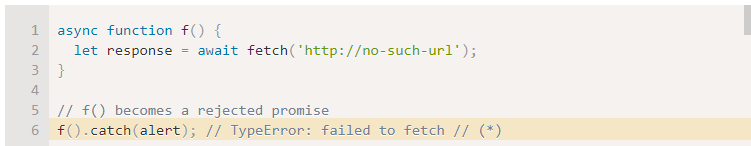


* **ERROR HANDLING :**
* Khi gặp lỗi, await sẽ phải đợi khá lâu.





* Nếu ta không có try…catch thì promise được gọi bởi async sẽ được trả về là rejected. Ta có thể dùng catch để bắt nó.



* **Thường ta sẽ dùng await/async thì sẽ ít dùng then thay vào đó ta sẽ dùng try..catch**