JS

▼ **Kỹ thuật debounce, throttling và cách code thực tế**

▼ **Function programming**

- Functional Programming là phương pháp lập trình lấy function làm đơn vị thao tác cơ bản.
- Các nguyên tắc trong FP:
  - **Immutability**: tính bất biến. Cái nào đã khai báo một lần thì mãi mãi như vậy, không bao giờ thay đổi nữa.
  - **Purity**: tính thuần khiết, thuần tuý không bị pha tạp.
    - Tất cả các hàm đều phải là pure function, không có hiệu ứng phụ (side effect), không được tác động lên bất cứ giá trị nào bên ngoài nó, cũng nói không với chỉnh sửa tham số input.
    - Đặc điểm quan trọng nữa của pure function là với mỗi tập giá trị đầu vào nhất định, luôn có 1 và chỉ 1 kết quả trả về tương ứng. Đây là tính chất của hàm số toán học.
    - Pure function trong Functional Programming thường ngắn gọn, đơn giản và chỉ xử lý duy nhất 1 vấn đề logic.

```
// not pure function
const getDuration = (timestamp) => {
  return Date.now() - timestamp;
};

// pure function
const add = (a, b) => {
  return a + b;
};
```

- **Higher order function**: là một khái niệm đến từ Toán học. Bất cứ hàm nào tiếp nhận 1 function như tham số, hoặc trả về 1 function như kết quả, thì đều được coi là higher-order function.

- **Function Composition:** Đây là khái niệm Toán học mà tiếng Việt ta gọi là "hàm hợp", hay "hàm phức hợp". Mọi thứ trong Functional Programming đều có nguồn gốc Toán học.
    - Function Composition là sự phối hợp, liên kết nhiều hàm lại với nhau, thành một hàm lớn, nhiều chức năng hơn.
    - Hoặc dễ hiểu đây là cách chúng ta kết hợp nhiều function lại với nhau. Kết quả của function này sẽ được sử dụng cho function tiếp theo. Cứ như vậy nó sẽ tạo thành một chuỗi các function thực hiện các nhiệm vụ theo thứ tự.
    - Có 2 kỹ thuật căn bản trong Function Composition là `compose` và `pipe`.
        - **compose**: trong toán học cơ bản với $y = f(g(x))$ ta tính g(x) trước rồi truyền cho f() là ra kết quả, từ phải sang trái. Ý tưởng của `compose` là xếp cuốn chiếu các hàm lại với nhau, theo thứ tự từ trái sang phải để tạo ra một hàm mới, mà khi được thực thi, nó sẽ lần lượt gọi các hàm đã truyền vào trước đó theo thứ tự ngược lại, từ phải sang trái (`Right to left`)
        - **Pipe:** Một biến thể của `compose` là `pipe`, vận hành theo chiều ngược lại. Ta có thể implement bằng cách đảo vị trí `f` và `g` (`Left to right`)
- **Currying function:** là một HOF, biến đổi 1 function n tham số thành n hàm nhận vào duy nhất 1 tham số

▼ Từ khóa "this" trong JS

- Từ khoá "this" trong javascript tham chiếu đến một đối tượng có thuộc tính là một hàm.
- Giá trị của "this" phụ thuộc vào đối tượng đang gọi hàm.

▼ Giải thích cách hoạt động của `this` trong JavaScript

- Không có lời giải thích đơn giản nào cho `this`; nó là một trong những khái niệm khó hiểu nhất trong JavaScript. Giá trị của `this` phụ thuộc vào cách hàm được gọi. Các quy tắc sau được áp dụng:
    - Nếu từ khóa `new` được sử dụng khi gọi hàm, thì bên trong hàm này là một đối tượng hoàn toàn mới.

    - Nếu `apply, call` hoặc `bind` được sử dụng để gọi / tạo một hàm, thì bên trong hàm này là đối tượng được truyền vào dưới dạng đối số.
    - Nếu một hàm được gọi là một phương thức, chẳng hạn như `obj.method()` – thì `this` là đối tượng mà hàm là thuộc tính của nó.
    - Nếu một hàm được gọi dưới dạng một lệnh gọi hàm miễn phí, nghĩa là nó được gọi mà không có bất kỳ điều kiện nào ở trên, thì đây là đối tượng toàn cục. Trong trình duyệt, nó là đối tượng `window`. Nếu ở chế độ nghiêm ngặt (`'use strict'`), `this` sẽ là `undefined` thay vì đối tượng toàn cục.
    - Nếu áp dụng nhiều quy tắc trên, quy tắc nào cao hơn sẽ được ưu tiên và sẽ đặt giá trị này.
    - Nếu hàm là một arrow function ES2015, nó sẽ bỏ qua tất cả các quy tắc ở trên và nhận `this` của phạm vi xung quanh tại thời điểm nó được tạo.

▼ **Phân biệt call, apply, bind**

- cả 3 đều dùng để thay đổi context của từ khóa this
- thiết đặt và thay đổi giá trị "**this**" theo ý muốn
- cả 3 hàm đều là các prototype của function. Nên chỉ có Function mới gọi được 3 hàm này.
- Về cơ bản là giống nhau về cách sử dụng nhưng khác nhau về cách gọi và cách truyền tham số
    - **call**: gọi trực tiếp hàm và cho phép truyền vào 1 object và ds những tham số ngăn cách bởi dấu phẩy
    - **apply**: giống như call nhưng nó truyền tham số là 1 array
    - **bind**: không gọi trực tiếp mà nó sẽ trả về 1 hàm mới và truyền danh sách tham số ngăn cách bởi dấy phẩy

▼ **Giải thích cơ chế hoisting trong JS**

- Hoisting là một hành vi mặc định của Javascript di chuyển việc khai báo lên đầu trong scope của nó.

▼ **Giải thích thêm về cách bộ máy JS hoạt động:**

- Khi bộ máy JS xử lý đoạn code của bạn , nó tạo ra 1 cái gọi là bối cảnh thực thi (execution context). Có 2 quá trình liên quan đến việc tạo cái bối cảnh thực thi này:
    - **GD1** (creation) : Trong giai đoạn này các biến và function được thêm vào bộ nhớ. Bộ máy JS sẽ đi qua một lượt đoạn code của bạn và thêm tất cả các biến vào bộ nhớ máy tính. **Nhưng nó chưa gán giá trị cho các biến này.** Trong khi đó các function thì lại được **thêm toàn bộ vào bộ nhớ bao gồm cả tên và khối code bên trong nó.**
    - **GD2** (execution) : Trong giai đoạn này giá trị sẽ được gán cho các biến và function sẽ được gọi. Nên kể cả bạn khởi tạo 1 biến với giá trị ban đầu thì ở giai đoạn 2 này biến mới được gán giá trị. Ở giai đoạn 1 giá trị không được gán cho biến , nó được thêm vào bộ nhớ với giá trị khởi tạo là undefined.
- Trước khi function được thực thi , nó đã được thêm vào bộ nhớ trong giai đoạn 1 (creation) nên bộ máy Javascript biết function này nằm ở đâu. **Nó không chuyển cái function này lên trên đầu.**
- Đối với biến quá trình cũng giống như vậy nhưng có 1 chút khác biệt. Các biến cũng được thêm vào bộ nhớ trong giai đoạn 1 nhưng không có giá trị nào được gán cho chúng. Trong JS khi một biến được khai báo mà không có giá trị nào bộ máy JS tự động thêm giá trị **undefined**

▼ **Temporal dead zone là gì ?**

- Từ es6 trở lên giới thiệu 2 từ khóa **let** và **const** để khai báo biến.
- Điểm khác biệt so với var là **ở giai đoạn 1 (creation) chúng không được khởi tạo với giá trị undefined như với var.**
- Thay vào đó: chúng được set 1 chế độ đặc biệt gọi là **Temporal Dead Zone**. Đây là 1 khoảng thời gian giữa việc khai báo và khởi tạo biến.
- Trong giai đoạn này bạn sẽ không thể truy cập vào biến đó được.
- Điều này có nghĩa là chúng vẫn tồn tại ở đó nhưng bạn sẽ không thể truy cập được cho đến khi bạn khởi tạo giá trị cho biến sẽ được thực hiện ở giai đoạn 2.

▼ **Ép kiểu ngầm trong JS là gì**

- là sự chuyển đổi tự động của giá trị từ kiểu dữ liệu này sang kiểu dữ liệu khác

- Ví dụ:
  - số + chuỗi = chuỗi
  - số - chuỗi = số
  - boolean sử dụng toán tử logic
    - Falsy: null, undefined, 0, 0n, -0, NaN, “”
    - Truethy: là những giá trị khác falsy

▼ **JS là static hay dynamic**

- là dynamic, kiểu dữ liệu của biến được kiểm tra trong khi đang chạy chương trình
- Đối với static thì nó sẽ kiểm tra sau khi biên dịch xong

▼ **Higher order function là gì (HOC)**

- là 1 hàm nhận vào tham số là hàm khác hoặc return về 1 hàm, hoặc cả 2 vừa nhận vừa return về 1 hàm

▼ **Scope trong JS**

- là khái niệm nhắm xác định phạm vi hoạt động của biến
- Các loại scope
  - **Global scope:**
    - biến được định nghĩa ở bên ngoài function
    - sẽ được sử dụng và thay thế ở bất cứ đâu
    - nếu ta định nghĩa 1 biến mà không dùng từ khóa nào thì mặc định nó thuộc global scope
  - **Local scope:**
    - biến được định nghĩa bên trong 1 function
    - mỗi function khi gọi sẽ tạo ra 1 scope mới
    - mỗi function sẽ tạo ra 1 local scope khác nhau, cho nên ta có thể khai báo các biến trùng tên ở các scope local
  - **Lexical scope:**

        - Ví dụ ta có fn A nằm trong fn B, mà trong A có các biến tham chiếu đến fn B thì ta nói fn A có lexical scope và lexical scope của A chính là scope của nó + với scope của fn B
    - Ngoài ra ta còn có block statements (if/else, switch case, for loop)
        - Không giống như function nó không tạo ra local scope mới mà nó giữ nguyên scope của nó
        - Các biến khai báo bằng var nó sẽ không tạo ra local scope trong block statements
        - let và const giúp tạo local scope bên trong 1 block
        - Biên được khai báo trong block {} thì chỉ truy cập được trong block đó thôi, bên ngoài không truy cập được

▼ **scope chain trong JS?**

- JS sử dụng scope để tìm kiếm biến
- Nếu không tìm thấy biến bên trong function scope thì nó sẽ tiếp tục tìm biến đó ra phạm vi bên ngoài
- Nếu không có sẽ tiếp tục tìm ở global scope

→ **Nếu không có nữa sẽ có lỗi tham chiếu được đưa ra**

▼ **Closures trong JS là gì ?**

- là 1 hàm mà có khả năng ghi nhớ nơi nó được tạo ra và truy cập được các biến ở bên ngoài phạm vi của nó

→ Vì lexical scope là khái niệm đi với function nên có thể coi closure là tập hợp gồm function và lexical scope của function đó

- Giúp ta viết code ngắn gọn hơn, đồng thời ứng dụng được tính **private** trong OOP

▼ **Currying Function là gì ?**

- Biến đổi 1 hàm nhiều tham sô thành n hàm nhận vào 1 tham số
- Về bản chất ta không thay đổi chức năng của hàm mà chỉ thay đổi cách gọi hàm

▼ **var, let, const khác nhau gì ?**

- về mặt scope
  - let, const giúp tạo local scope bên trong block statements
  - var rất khó xác định scope cụ thể nên bây giờ cũng tránh sử dụng var, ngoài ra các biến khai báo bằng var đều bị hoisting
- Về mặt giá trị
  - const là bất biến ta không thể gán lại giá trị cho biến được khai báo với const
  - var, let thì có thể gán lại giá trị cho biến

▼ **null ≠ undefined như thế nào ?**

- Khi ta khai báo nhưng chưa gán giá trị cho biến đó thì nó là **undefined**
- Khi ta lấy giá trị của một biến mà chưa khai báo thì nó sẽ gây ra lỗi **is not define**
- **Null** có nghĩa là giá trị rỗng hoặc giá trị không tồn tại, nó có thể được sử dụng để gán cho một biến như là một đại diện không có giá trị.
- **Null** typeof là **object** còn **undefined** typeof là **undefined**
- null == undefined = true
- null === undefined = false

▼ **Prototype trong JS là gì ?**

- Prototype là khái niệm cốt lõi trong JavaScript và là cơ chế quan trọng trong việc thực thi mô hình OOP trong JavaScript.
- Tất cả các object trong javascript đều có một prototype, và các object này kế thừa các thuộc tính (properties) cũng như phương thức (methods) từ prototype của mình.
- Trong JavaScript, trừ *undefined*, toàn bộ các kiểu còn lại đều là object.
- Các kiểu string, số, boolean lần lượt là object dạng *String*, *Number, Boolean*. Mảng là object dạng *Array*, hàm là object dạng *Function*

- Trước kia (ES5) trong JavaScript **không có khái niệm class**, do vậy, để **kế thừa các trường/hàm** của một object, ta phải sử dụng prototype.
- Khi ta gọi property hoặc function của một object, JavaScript sẽ tìm trong chính object đó, nếu không có thì tìm lên cha của nó. Do đó, ta có thể gọi các hàm *toUpperCase*, *trim* trong String là do các hàm đó đã tồn tại trong *String.prototype*
- Khi ta thêm function cho prototype, toàn bộ những thằng con của nó cũng học được function tương tự.
- **Giới hạn của prototype trong JavaScript:**
  - Không được phép kế thừa prototype vòng tròn.
  - Giá trị của `__proto__` có thể là `null` hoặc là một object, nhưng các kiểu dữ liệu khác đều bị bỏ qua.
  - Prototype không hỗ trợ thay đổi giá trị thuộc tính

▼ **Tóm lại:**

- Trong JavaScript, tất cả các object đều có thuộc tính ẩn `[[Prototype]]` với giá trị là `null` hoặc kiểu object.
- Bạn có thể sử dụng `obj.__proto__` như là một **getter/setter** để truy cập vào `[[Prototype]]`.
- Object ứng với `[[Prototype]]` được gọi là một prototype.
- Khi truy cập một thuộc tính hay phương thức trong object mà nó không tồn tại thì JavaScript sẽ tự động tìm kiếm trong prototype.
- Prototype chỉ hỗ trợ việc đọc, không hỗ trợ ghi/xóa thuộc tính trực tiếp trên prototype.
- Khi bạn gọi `obj.method()` và `method()` được lấy từ prototype, giá trị của `this` vẫn tham chiếu đến `obj` chứ không phải prototype.
- Vòng lặp `for...in` duyệt tất cả các thuộc tính trong object và thuộc tính của prototype thông qua kế thừa.

▼ **Class trong JS?**

- sử dung oop mà không cần dùng prototype

- không bị hoistring như function, cần khai báo trước khi sử dụng
- tuân thủ use strict
- sử dụng extends để kế thừa

▼ **Constructor trong JS**

- Dùng để tạo ra các thuộc tính mặc định của đối tượng trong JS
- khi chúng ta muốn tạo ra 1 đối tượng với những thuộc tính và phường thức giống nhau nhưng khác giá trị

▼ **Arrow Function là gì**

- Arrow function được giới thiệu từ phiên bản ES6 của javascript.
- Nó cung cấp một cú pháp mới và ngắn hơn cho khai báo hàm.
- Hàm arrow có thể sử dụng như là một biểu thức hàm. Ta sẽ so sánh khai báo hàm thông thường với hàm arrow.
- Sự khác biệt lớn nhất giữa nhất giữa hàm truyền thống với arrow, là ở từ khoá **this**.
    - Như định nghĩa, từ khoá this tham chiếu đến đối tượng chứa hàm được gọi.
    - Còn ở hàm arrow, không có ràng buộc nào của từ khóa this.
    - Từ khoá this trong hàm arrow, không tham chiếu đến đối tượng đang gọi nó. Nó kế thừa giá trị của nó từ phạm vi cha là `window object` trong trường hợp này.

▼ **Destructuring, spread operator và rest parameter ?**

- **Destructuring:** Là một cú pháp cho phép bạn gán các thuộc tính của một Object hoặc một Array. Điều này có thể làm giảm đáng kể các dòng mã cần thiết để thao tác dữ liệu trong các cấu trúc này
- **Spread operator:** Là ba dấu chấm **(...)**, có thể chuyển đổi một mảng thành một chuỗi các tham số được phân tách bằng dấu phẩy. Spread có thể tao ra một cấu trúc dữ liệu shallow copy để tăng tính thao tác dữ liệu.
- **Rest parameter:** Là một cú pháp tạo ra một array từ một số lượng giá trị không xác định Giúp chúng ta có thể định nghĩa một hàm với số lượng tham số có thể

thay đổi tuỳ ý. Hay nói theo cách khác khi chúng ta không biết chắc chắn số lượng tham số cần có của một hàm chúng ta có thể sử dụng rest parameter

▼ **Generator trong JS**

- es6
- là function có khả năng hoãn lại quá trình thực thi mà vẫn giữ nguyên context
- Nó có khả năng tạm dừng trước khi hàm kết thúc và có khả năng thực hiện lại tại 1 thời điểm khác, khác với những function bình thường là thực thi hết tất cả câu lệnh trong function

▼ **ES6 có gì nổi bật**

- let, const
- arrow function
- generator function
- Destructuring, spread operator và rest parameter
- class
- async/await
- module
- map, set
- template literal

▼ **const ≠ Object.freeze() gì ?**

- const là 1 key dùng để ràng buộc các biến không thể thay đổi giá trị
- object.freeze() hoạt động với giá trị đối tượng. Làm đối tượng trở nên bất biến. Chỉ read không được write các thuộc tính trong đối tượng

▼ **Proxy trong JS là gì**

- **Proxy** là một class được giới thiệu từ ES6, cho phép bạn can thiệp và thay đổi hành vi của một đối tượng (object).
- Các hành vi này bao gồm: truy xuất/thiết lập (getter/setter) thuộc tính của một đối tượng, thay đổi prototype, gọi hàm, khởi tạo đối tượng bằng từ khóa `new`

- Chúng ta có thể áp dụng Proxy cho bất cứ object nào trong JavaScript, kể cả mảng, hàm hay một proxy khác.

▼ **Memoization là gì?**

- là một kỹ thuật tối ưu, nhằm tăng tốc chương trình bằng cách lưu trữ kết quả của các câu gọi function và trả về các kết quả này khi function được gọi với cùng input đã gọi.

▼ **Có thể dùng những vòng lặp nào trong JS (for-loop) ?**

- **map**: để thực hiện một chức năng trên mỗi phàn tử của một mảng. Sử dụng **map** nếu chúng ta muốn thực hiện cùng một thao tác hoặc chuyển đổi trên từng phần tử của mảng và lấy lại một array mới có cùng **length** với các **value** được chuyển đổi.
- **filter**: khi chúng ta muốn xoá các mục không thoã mãn điều kiện. Mỗi phần tử của mãng được truyền cho hàm callback. Trên mỗi lần lặp nếu callback trả về true, thì phần tử đó sẽ được thêm vào mảng mới và ngược lại
- **reduce**: sử dụng để lặp qua các phần tử của mảng sau đó tính toán và trả về 1 kết quả cuối cùng. Thường được sử dụng để giải quyết các bài toán như lặp qua một mảng → xử lý gì đó → trả về một kết quả cuối cùng
- **for...in**: dùng để lặp lại trong một object. Số lượng vòng lặp sẽ tương ứng với số lượng thuộc tính trong object. Array cũng là object nên ta có thể sử dụng cho cả array, nhưng key sẽ là giá trị index của phần tử
- **for...of**: dùng để lặp các đối tượng từ Array, String, Map, WeakMap, Set, ... Đặc biệt có thể dùng bất đồng bộ trong vòng lặp
- for, do while, forEach, while

▼ **CORS**

- (Cross-Origin Resource Sharing) là một kĩ thuật được sinh ra để làm cho việc tương tác giữa client và server được dễ dàng hơn, nó cho phép JavaScript ở một trang web có thể tạo request lên một REST API được host ở một domain khác.
- Hiểu sâu hơn đó chính là chia sẻ tài nguyên có nhiều nguồn gốc khác nhau. Chính sách nguồn gốc giống nhau của trình duyệt là một cơ chế bảo mật quan trọng. Khách hàng từ các nguồn khác nhau không thể truy cập tài nguyên của

nhau nếu không được phép. Định nghĩa của tương đồng là protocol, domain và port của liên kết truy cập là giống nhau.

▼ **Xử lý bất đồng bộ là gì ?**

Giả sử bạn có một nhiệm vụ bao gồm 2 công việc tốn thời gian, tạm gọi là A và B.

- **Xử lý đồng bộ:**
  - bạn sẽ thực hiện công việc A
  - đợi A hoàn thành xong thì sẽ thực hiện B
  - rồi lại đợi B hoàn thành thì nhiệm vụ cuối cùng mới coi như xong.
  - Nghĩa là thời gian để hoàn thành nhiệm vụ là tổng của thời gian hoàn thành A và B. Hơn nữa, trong khoảng thời gian này bạn sẽ không thể thực hiện thêm 1 hành động nào khác
  - Điều này rõ ràng làm giảm hiệu năng và trải nghiệm người dùng đối với chương trình.
- **Xử lý đa luồng:**
  - Để khắc phục tình trạng này, các ngôn ngữ lập trình như C/C++, Java,... sẽ sử dụng **cơ chế đa luồng (multi-thread)**.
  - Nghĩa là mỗi công việc tốn thời gian sẽ được thực hiện trên một thread riêng biệt mà không can thiệp vào thread chính.
  - Bạn vẫn có thể thực hiện các công việc tốn thời gian mà vẫn có thể bắt các sự kiện ở thread chính.
  - Với ví dụ trên, thời gian để hoàn thành nhiệm vụ sẽ chỉ bằng thời gian hoàn thành của A hoặc B. Cái nào thực hiện xong trước sẽ đợi cái còn lại hoàn thành thì nhiệm vụ sẽ kết thúc.
- **Xử lý bất đồng bộ:**
  - Tuy nhiên, JavaScript lại là một câu chuyện khác. Hai nền tảng quan trọng với JavaScript đều là **single-thread.**
  - Chính vì vậy, bạn không thể xử lý đa luồng với JavaScript được mà phải sử dụng cơ chế **xử lý bất đồng bộ**

- Với cách xử lý bất đồng bộ, khi A bắt đầu thực hiện, chương trình tiếp tục thực hiện B mà không đợi A kết thúc.
- Việc mà bạn cần làm ở đây là cung cấp một phương thức để chương trình thực hiện khi A hoặc B kết thúc.

▼ **Callback function là gì, Promise là gì, async await là gì? so sánh?**

▼ **callback function**

- Là một hàm sẽ được thực hiện sau khi một khác đã thực hiện xong
- Trong JS, hàm là các Object dạng function. Do đó các hàm có thể lấy các hàm làm đối số và có thể được trả về bởi các hàm khác. Các hàm thực hiện điều này gọi là **HOC**.
- Bất kỳ hàm nào được truyền dưới dạng đối số thì đều được gọi là hàm **callback**

▼ **Promise**

- Là một đối tượng sẽ trả về một giá trị trong tương lai.
- Rõ hơn thì một object **Promise** đại diện cho một giá trị ở thời điểm hiện tại có thể chưa tồn tại, nhưng sẽ được xử lý và có giá trị vào một thời gian nào đó trong tương lai.
- Ta thường hay gặp các trường hợp sử dụng quá nhiều **callback** dãn đến tình trạng gọi **callback hell**, nên **promise** là cách để giải quyết vấn đề trên
- Việc sử dụng promise sẽ giúp cho việc cử lý không đồng bộ sẽ trở nên đồng bộ và gọn gàng hơn.
- **Promise có 3 trạng thái**
  - Pending (đang xử lý)
  - Fulfilled (đã hoàn thành)
  - Rejected (đã bị từ chối)

▼ **Async await**

- Trước đây, để làm việc với code bất đồng bộ, chúng ta sử dụng callback và promise, **Async/Await** là một cách mới để viết code bất đồng bộ.

- Nó được xây dựng trên Promise và tương thích với tất cả Promise dựa trên API
- **Async** được dùng để khai báo một hàm bất đồng bộ
  - 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** được sử dụng để chờ một Promise. Nó chỉ có thể được sử dụng bên trong một khối Async.
  - 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.
- **Dùng Async Await vì**:
  - Ngắn gọn, sạch sẽ: viết code tuần tự, làm cho số lượng code viết giảm đáng kể.
  - Xử lý lỗi: giúp ta xử lý cả error đồng bộ lẫn bất đồng bộ theo cùng 1 cấu trúc
  - Câu lệnh điều kiện: giúp ta xử lý những câu điều kiện dựa vào kết quả trả về một cách dễ hiểu và nhanh hơn
  - Debug: việc debug trở nên dễ dàng hơn
- **Async / Await làm cho promise lỗi thời ?**
  - Khi làm việc với Async / Await, thật ra chúng ta vẫn đang sử dụng ngầm Promises.
  - Vì thế, kể cả khi đang sử dụng Async / Await cần một sự hiểu biết tốt về Promises sẽ rất tốt cho chúng ta.
  - Ngoài ra, có những trường hợp mà Async / Await không sử dụng được và chúng ta phải sử dụng Promises.

    - Ví dụ khi ta sử dụng để gọi song song các tác vụ bất đồng bộ và không phụ thuộc vào nhau thì async await không giải quyết được, nhưng **Promise.all()** có thể làm được

▼ **Shallow copy và deep copy trong js**

- **shallow copy**: nhiệm vụ của nó chỉ copy những giá trị nông, nghĩa là nó sao chép những giá trị đối tượng bình thường nhưng các giá trị lồng nhau vẫn sử dụng để tham chiếu (reference) đến đối tượng ban đầu.
- **deep copy**: đơn giản cũng giống với shallow copy nhưng các giá trị của reference trong object gốc không thay đổi trong object đã clone

▼ **localStorage & sessionStorage & cookies**

▼ **sessionStorage:** Dữ liệu chỉ tồn tại cho đến khi người dùng đóng tab hoặc đóng trình duyệt, còn lại mọi thứ đều giống localStorage

▼ **cookie:** Thường có khoảng thời gian sống nhất định và tuỳ thuộc vào mục đích sử dụng sẽ có khoảng thời gian sử dụng khác nhau, ngoài hạn sử dụng cookie còn có thời gian sống (max-age)

- Khả năng lưu trữ thông thường khoảng 4 KB
- Thông tin được gửi lên server
- Có thể đọc ở phía máy chủ khác với Local/Session Storage chỉ đọc được ở phía máy khách.

▼ **localStorage:** Dữ liệu được lưu trữ vô thời hạn, dữ liệu sẽ không mất đi trừ khi sử dụng chức năng xoá của trình duyệt hoặc dùng localStorage API để xoá.

- Khả năng lưu trữ khoảng 5MB.
- Dữ liệu không được gửi đi đến server thông qua các request header.
- Được lưu trữ trên trình duyệt của người dùng nên việc sử dụng sẽ không ảnh hưởng đến hiệu suất của trang web nhưng sẽ làm nặng trình duyệt (rất nhỏ gần như không đáng kể).

▼ **Map, Set, WeakMap, WeakSet là gì?**

▼ Map trong JavaScript là một **cấu trúc dữ liệu** cho phép lưu trữ dữ liệu theo kiểu **key-value**, tương tự như object. Tuy nhiên, chúng khác nhau ở chỗ là:

- **Object** chỉ cho phép sử dụng String hay Symbol làm key
- **Map** cho phép mọi kiểu dữ liệu (String, Number, Boolean, NaN, Object,...) có thể làm key và Map có thuộc tính **size** và một số phương thức đặc trưng
- Một số phương thức và thuộc tính của Map:
  - `new Map([iterable])`: khởi tạo Map với tham số là một iterable object (không bắt buộc) với mỗi phần tử có dạng `[key, value]`.
  - `map.set(key, value)`: lưu `value` bởi `key` và trả về `map`.
  - `map.get(key)`: trả về `value` bởi `key`, nếu `key` không tồn tại thì trả về `undefined`.
  - `map.has(key)`: trả về `true` nếu `key` tồn tại, ngược lại thì trả về `false`.
  - `map.delete(key)`: xóa giá trị ứng với `key` và trả về `true` nếu `key` tồn tại, ngược lại thì trả về `false`.
  - `map.clear()`: xóa tất cả các phần tử trong `map`.
  - `map.size`: trả về số phần tử hiện tại có trong `map`.

▼ **Set trong Javascript** là một loại object cho phép bạn lưu trữ dữ liệu một cách duy nhất, không trùng lặp.

- Set là một loại object
- Dữ liệu trong set là duy nhất và không trùng lặp. Không trùng lặp ở đây được hiểu là các phần tử không được giống nhau.
- Có thể lưu `NaN` và `undefined` vào Set trong JavaScript.
- Các phương thức của Set là:
  - `new Set(iterable)`: khởi tạo Set bằng cách truyền vào một iterable object (không bắt buộc), trường hợp không truyền vào tham số nào thì Set sẽ rỗng.
  - `set.add(value)`: thêm phần tử `value` vào Set và trả về chính Set đó.
  - `set.delete(value)`: xóa một phần tử trong Set và trả về `true` nếu giá trị `value` tồn tại trong Set, ngược lại trả về `false`.

- `set.has(value)`: trả về `true` nếu giá trị `value` tồn tại trong Set, ngược lại thì trả về `false`.
- `set.clear()`: xóa tất cả các phần tử trong Set.
- `set.size`: trả về số lượng phần tử trong Set.

https://completejavascript.com/ban-biet-gi-ve-set-trong-javascript/

▼ WeakMap

- WeakMap trong JavaScript tương tự như Map, cho phép lưu trữ dữ liệu theo kiểu `key-value`. Tuy nhiên, WeakMap chỉ chấp nhận **object** làm **key** còn map thì cho phép tất cả mọi kiểu dữ liệu
- Khi object bị hủy, object tương ứng trong WeakMap sẽ bị giải phóng vì không còn cách nào để truy cập vào object đó nữa.

```
let alex = { name: "Alex" };

// khai báo map và sử dụng biến alex làm key cho map
let map = new Map();
map.set(alex, "1");

// ghi đè giá trị của biến alex
alex = null;

// mặc dù biến alex bị gán bằng null, nhưng object alex vẫn tồn tại trong map
console.log(map.size); // 1
for (let item of map) {
  console.log(item);
  /**
   * [{name: 'Alex'}, '1']
   */
}
// Để hủy ta phải dùng method delete trong map
```

```
let alex = { name: "Alex" };

// khai báo map và sử dụng biến alex làm key cho weakMap
let weakMap = new WeakMap();
weakMap.set(alex, "1");

// ghi đè giá trị của biến alex
alex = null;

// Sau khi biến alex bị gán bằng null,
```

```
// không còn cách nào có thể truy cập vào phần tử với key là alex trước đó.
// Vì vậy, object với alex sẽ bị hủy.
```

- WeakMap không phải iterable object nên không có cách nào để duyệt hết các phần tử trong WeakMap (như cách dùng `for...of` với Map).
- Ứng dụng của weakMap:
  - Lưu dữ liệu ví dụ lưu lại số lần truy cập của một đối tượng user
  - Caching dữ liệu
- Các phương thức của WeakMap là:
  - `weakMap.set(key, value)`: lưu giá trị `value` vào thuộc tính `key` và trả về chính WeakMap.
  - `weakMap.get(key)`: trả về giá trị của thuộc tính `key`, nếu `key` không tồn tại thì trả về `undefined`.
  - `weakMap.delete(key)`: xóa thuộc tính `key` trong WeakMap, nếu `key` tồn tại thì trả về `true`, ngược lại thì trả về `false`.
  - `weakMap.has(key)`: trả về `true` nếu `key` tồn tại trong `weakMap`, ngược lại thì trả về `false`.

https://completejavascript.com/weakmap-trong-javascript/

▼ WeakSet

- WeakSet trong JavaScript tương tự như Set, cho phép **lưu trữ dữ liệu một cách duy nhất**, không trùng lặp. Tuy nhiên, WeakSet chỉ chấp nhận phần tử kiểu object.
- Khi object bị hủy, object tương ứng trong WeakSet sẽ bị giải phóng vì không còn cách nào để truy cập vào object đó nữa.
- WeakSet không phải iterable object nên không có cách nào để duyệt hết các phần tử trong WeakSet (như cách dùng `for...of` với Set).
- Các phương thức của WeakSet là:
  - `weakSet.add(value)`: lưu giá trị `value` vào WeakSet và trả về chính WeakSet.

    - `weakSet.delete(value)`: xóa phần tử `value` trong WeakSet, nếu `value` tồn tại thì trả về `true`, ngược lại thì trả về `false`.
    - `weakSet.has(value)`: trả về `true` nếu `value` tồn tại trong `weakSet`, ngược lại thì trả về `false`.

▼ **Sự khác biệt giữa `Set`, `WeakSet`, `Map` và `WeakMap` trong JavaScript là gì?**

- `WeakSet` và `Set` đều là tập hợp các giá trị duy nhất. Sự khác biệt chính là `WeakSet` chỉ lưu trữ đối tượng và không thể chứa các giá trị tùy ý thuộc bất kỳ loại nào, nhưng các `Set` thì có thể.
- Sets hữu ích khi bạn cần nối từng dữ liệu một vào cấu trúc dữ liệu nhưng cũng muốn loại bỏ các phần trùng lặp. Các hoạt động tập hợp có giá trị trung bình là `O(1)`, điều này làm cho chúng tiết kiệm thời gian.
- `WeakMap` và `Map` là tập hợp các cặp khóa / giá trị. Sự khác biệt chính là trong `WeakMap`, các khóa phải là các đối tượng. Trong `Map`, các khóa có thể thuộc bất kỳ loại nào.
- Cũng cần biết rằng các giá trị `WeakMap` không thể được lặp lại và chúng giữ một tham chiếu yếu (weak reference) đến khóa. Ví dụ: nếu bạn xóa thủ công một khóa được tham chiếu trong `WeakMap`, khóa đó sẽ được thu gom.

▼ **setTimeout, setInterval**

- Là những hàm cho phép bạn thực hiện một đoạn mã Javascript tại một thời điểm nào đó trong tương lai. Nó được gọi là "lập lịch một cuộc gọi" (scheduling a call)
    - **setTimeout**: sử dụng để thực thi một hàm hoặc đoạn mã được chỉ định chỉ một lần sau một khoảng thời gian nhất định.
    - **setInterval**: sử dụng để thực thi một hàm hoặc đoạn mã được chỉ định lặp đi lặp lại vào những khoảng thời gian cố định.
- Dừng thực thi bộ đếm thời gian sử dụng: **clearTimeout()** và **clearInterval()**

▼ **Event loop trong JS**

- **Event Loop** là cơ chế giúp Javascript có thể thực hiện nhiều thao tác cùng một lúc (concurrent model)
- Cuối cùng thì event loop là gì ?

- Event loop trong JS

- Do JS đơn luồng nên nó chỉ xử lý được duy nhất một tác vụ trong cùng một thời điểm, điều này rất khác với các ngôn ngữ đa luồng (multi-thread) khác như C#, Java, PHP với mỗi tác vụ thì nó sẽ chia ra một luồng để xử lý.
- Nếu như thế thì JS rất dễ rơi vào tình trạng **Blocking** nếu gặp phải các tác vụ mất nhiều thời gian như xử lý nhiều request, call APIs, …
- Event Loop ra đời để giải quyết vấn đề này, khiến JS chỉ đơn luồng như cân tất nhiều tác vụ cùng lúc
- Về Event Loop thì cần biết thêm 2 khái niệm đó chính là:
    - **Web APIs**: bản chất Runtime của Javascript chỉ có 1 luồng và không thể chạy multi-thread, vì thế browser đã viết thêm một Web APIs để bọc runtime này lại (tương tự dưới NodeJS sẽ dùng C++ để bọc V8 lại). Web APIs này sẽ giúp cho JS có thể hoạt động một cách bất đồng bộ như multi-thread.
    - **Callback Queue**: Như tên của nó là hàng đợi các callback do thằng Web APIs ở trên trả về.
- **Cách hoạt động:**
    - Khi hàm main được chạy thì các đoạn code trong main sẽ được thực thi. Nó sẽ lần lượt đẩy các hàm vào bên trong call stack theo nguyên tắc LIFO
    - Các hàm hay tác vụ liên quan đến Events (click, change, listener, …), AJAX (Call APIs), Timing (setTimeout, setInterval) sẽ được đẩy từ call stack sang Web APIs. Còn lại thì sẽ được thực thi trong call stack đến khi nào xong thì pop nó ra cho hàm bên dưới được thực thi.
    - Ở Web APIs sẽ tận dụng các nhân của thiết bị để xử lý riêng biệt các tác vụ này. Sau khi hoàn tất thì Web APIs sẽ trả về một callback và đẩy nó vào trong Callback Queue.
    - Callack Queue hoạt động theo nguyên tắc của queue là FIFO (vào trước ra trước) không như stack.
    - Event loop hiểu nôm na là một vòng lặp vô tận, nó luôn trực chờ ở đó để quan sát Callback Queue và bé Call stack.

- Bất kể khi nào mà call stack trống (tất cả các hàm được pop ra) thì nó sẽ nắm các thằng callback ở trong Callback Queue và ném vào trong Call Stack để tiếp tục thực thi.