* Agular là 1 framework dùng để build single page đều có index.html.
* File main.ts là file chạy đầu tiên khi chạy dự án angular.
* AppModule bản thân là 1 TypeScript module là 1 file riêng lẻ có thể import những thứ bên ngoài vào để sử dụng và cũng có thể export những thứ của nó ra bên ngoài sử dụng
* @NgModule (TypeScipt Decorator) là cú pháp liên quan đến type script. Angular sử dụng TypeScipt Decorator để cung cấp Metadata
* Component là 1 thành phần của web. 1 web bao gồm nhiều component. Các component có thể tái sử dụng.
* AppComponent là component root là component đầu tiên được render
* Khi tạo 1 component cần phải khai báo nó trong file app.module.ts. 1 component chỉ được khai báo cho 1 cái module nhất định
* String Interpolation: {{expression}}
* Data Binding:

+ property binding: [property] = “tên biến”: hiển thị dữ liệu từ trong app.component.ts ra template

* Event Binding: Nhận sự kiện từ template và xử lí ở trong app.component.ts
* To way data binding:

<input type = “text” [(ngModel)]=”name” />

Tương đương với:

<input type = “text” [ngModel]=”name” (ngModelChange)=”name = $event” />

* Class binding:

+ [class.ten-class]=”Biến true hoặc false”

+ [class] = “array tên class”

+ [class] = “object” {[ten-clas: string]: boolean}

+[class] = “variable”

* Input binding: khi ta tạo ra 1 component ta chỉ thao tác với nó qua template của nó, bây giờ ta muốn tương tác với nó thông qua thằng cha ta sẽ thêm @input() vào trước thuộc tính. Và trong template của thằng cha sử dụng component này, ta sẽ truyền các thuộc tính này vào. @input báo rằng thuộc tính này sẽ truyền vào từ component cha (thằng con lấy từ thằng cha)
* Ouput Binding: từ thằng con bắn event ra cho thằng cha xử lí. Trong thằng con sẽ khai báo @Output() select = new EventEmitter<Author>(). Khi nào muốn bắn sự kiện ra cho thằng cha thì ta gọi hàm emit(): this.select.emit(this.author). Thằng cha hứng và gọi hàm xử lí event nhận được (select)="onSelect($event)
* Two way data binding: giúp 2 component có thể tương tác với nhau, thằng cha truyền vào cho thằng con, thằng con thay đổi thằng cha cũng thay đổi và ngược lại
* **Vòng đời của 1 component**
* Khi khởi tạo component, constructor sẽ được gọi lúc này html css đã được load đã có trên view ở thời điểm đó Angular chưa có bind input những thuộc tính @input chưa nhận được giá trị từ bên ngoài truyền vào (ngày 7)
* Trong trường hợp có input data thì ngOnchanges chạy tiếp theo để bind giá trị vào input. Nếu như không có input truyền vào ngOnchanges không chạy. ngOnchanges sẽ được chạy mỗi khi có thay đổi input truyền vào.
* ngOnInit được gọi sau khi input được binding. ngOnInit chỉ chạy 1 lần duy nhất
* Tại sao khi có Contructor rồi vẫn phải có onInit để làm gì ? Vì Contructor được gọi đầu tiên, lúc Angular chưa có binding input. Còn onInit được gọi 1 lần duy nhất sau khi angular binding input -> giả sử muốn lưu 1 giá trị trước lúc nó được binding data ta sẽ xử lí trong hàm constructor(), muốn lưu giá trị sau khi binding Input lần đầu tiên ta sẽ lưu trong hàm onInit().
* Structural Directive là 1 đoạn Directive có thể thay đổi cấu trúc của dom
* ? là trước khi get nó sẽ kiểm tra nếu khác rỗng thì nó get, bằng rỗng thì nó ko get
* Mỗi khi giá trị biến trong file component.ts thay đổi Angular tự động thay đổi hiển thị trên trang html (không phải load lại cả trang web)
* **@ViewChild():** ta muốn tham chiếu đến 1 thằng con trên cái view của component. Ta truyền vào string là cái biến (template variable) trong view hoặc tên của component. Sau khi view được khởi tạo xong thì ta sẽ tham chiếu đến được @ViewChild() ở trong hàm ngAfterViewInit() {}. Ở thời điểm onInit Angular chưa tham chiếu đến được @ViewChild() -> trong trường hợp chúng ta muốn tham chiếu được trong onInit() thì @ViewChild() cung cấp cho ta 1 cái option có 1 property là static ta truyền vào true @ViewChild(ToggleComponent, {static: true}) mặc định là false và mình chỉ có thể sử dụng static là true khi mà cái mà ta tham chiếu không nằm trong Structural Directive nào, có 1 property nữa là {read: ElementRef} mặc định là ElementRef, Angular sẽ cố gắng lấy về cho ta 1 elementRef ta có thể truyền vào các option khác ViewContainerRef
* **ViewChildren**: dùng để tham chiếu tới nhiều children trên view
* job?: string gọi là Optional property, job là 1 property không bắt buộc, nghĩa là khi tạo đối tượng ta có thể có thể thêm thuộc tính này, có thể bỏ qua cũng ko bị lỗi
* **<ng-content></ng-content>** : trong quá trình phát triển ứng dụng angular chúng ta sẽ gặp các một số dạng component giống nhau về phần layout, điểm khác biệt chỉ là 1 số lable, content trong đó. Lúc này rất dễ để các bạn có thể tạo 1 component có nhận vào các input và render dựa vào các input đó. Có 1 cách khác nữa là dùng ng-content. Chúng ta có thể đặt ng-content vào bất kì đâu trong template của component và ta sẽ truyền 1 thẻ html, 1 component vào trong cặp thẻ của component ở nơi sử dụng nó. Trong cặp thẻ ng-content có 1 thuộc tính là select giống như CSS selector truyền vào tag-selector, CSS Class selector, attribute selector dùng để xác định thẻ nào sẽ được truyền vào vào ng-content
* **ngProjectAs** dùng để đặt tên cho thẻ hoặc component truyền vào ng-content. Giúp ng-content có thể select được
* **ng-template**: là 1 cái thẻ trên html. Khi code html phần code được bao quanh bởi ng-template sẽ không đươc render. Thường được sử dụng với \*ngIf và trường hợp khi 1 số UI element trong 1 component bị lặp lại trong chính component đó nhưng phần code đó quá nhỏ để tách ra làm 1 component riêng. Ta sử dụng ng-template để khi có sửa đổi ta chỉ cần sửa đổi 1 chỗ duy nhất
* Dependency Injection trong Angular: trường hợp thường dùng là khi ta inject các service vào trong component. Trường hợp tiếp theo là khi ta muốn inject thằng cha vào trong thằng con để khi ta gọi hàm gì ở trong thằng con mình sẽ trực tiếp dùng thằng cha để quản lí những thằng con (bình thường muốn làm như thế ta phải tạo 1 @Output để emit cho thằng cha để nó xử lí)
* RxJS: là 1 thư viện giúp bạn làm việc với bất đồng bộ
* Promise sinh ra để giúp chúng ta xử lí những thao tác bất đồng bộ.
* Promise: Bình thường khi ta cần xử lí những công việc có sự phụ thuộc lẫn nhau(ví dụ việc này xong thì mới làm việc kia chẳng hạn) lúc này ta sẽ truyền vào 1 callback function để handle sau khi công việc hoàn thành -> việc sử dụng callback function này rất dễ dẫn đến callback hell (code khó coi, khó debug…) => lúc này ta có thể dùng promise để cho code gọn gàng hơn, promise đơn giản là 1 lời hứa thành công thì làm gì, thất bại thì làm gì. Khởi tạo Promise bằng toán tử new nhận vào 1 function, funciton này sẽ được gọi ngay khi toán tử new được gọi trước cả khi promise được tạo ra. Function này nhận vào 2 tham số cũng là 2 function, tham số thứ nhất là resolve, thứ 2 là reject. Phần thân function là logic của chúng ta muốn xử lí (lời hứa), nếu logic xử lí là thành công thì ta gọi hàm resolve(), thất bại gọi reject(). Nhu vậy ta đã tạo được 1 đối tượng promise, đối tượng này có các phương thức sau then(function), catch(function), finally(function) . Phương thức then() được gọi khi lời hứa thành công tức là hàm resolve() được gọi, Phương thức catch() được gọi khi lời hứa thất bại tức là hàm reject() được gọi, Khi mà 1 trong 2 phương thức then() và catch() được gọi thì phương thức finally() sẽ được gọi
* Trạng thái Promise:

+ Pending: trạng thái đang chờ (khi resolve(), reject() chưa được gọi) trạng thái này đang bị rò rỉ bộ nhớ.

+ Fulfilled: trạng thái thành công

+ Rejected: trạng thái thất bại

* Tính chất của promise: Giá trị trả về của thằng then phía trước lại là giá trị đầu vào của thằng then phía sau. Trong function của then() nếu không return ra 1 promise thì nó chạy ngay thằng then liền kề với nó. Nếu nó trả về 1 promise thì phải chờ promise này chạy xong
* Promise chỉ xử lí được 1 tác vụ bất đồng bộ mà thôi và nó chỉ trả về 1 giá trị nếu muốn áp dụng vào mô hình trả về nhiều giá trị => không thể sử dụng promise được, muốn sử dụng được chúng ta phải tạo ra nhiều object promise => observable ra đời giúp chúng ta xử lí được các tác vụ bất đồng bộ trả về nhiều giá giá trị cùng 1 lúc
* Observable: là 1 object đại diện cho 1 tập hợp nhiều giá trị, hoặc là những event được gửi đến trong tương lai. Nó có thể dùng để xử lí những tác vụ bất đồng bộ và đồng bộ
* Producer: là nơi sinh ra dữ liệu truyền sang cho Consumer, khi có bất kì dữ liệu nào mới nó sẽ được gửi về cho consumer
* Consumer: là nơi nhận về dữ liệu
* Observable sinh ra để chuẩn hóa mô hình Producer – Consumer với cùng 1 API giống nhau và cùng 1 cơ chế xử lí
* Trạng thái Observable: success (thành công) có thể nhiều lần, error, complete( khi nào mà chúng ta không muốn nhận dữ liệu nữa hoặc dữ liệu đã hết)
* Subcrice: là 1 function, là cơ chế để mình đăng kí để nhận dữ liệu từ cái mô hình Producer-Consumer. Bình thường khi sử dụng promise khi ta khởi tạo promise thì function truyền vào sẽ được gọi. Còn observable phải khi ta subcribe
* Next() là function được gọi khi consumer có tín hiệu từ producer
* Quy tắc đặt tên tất các obj nào của observable thì có dấu $ đằng sau
* Khi 1 cái Observable Subcribe nó sẽ trả về cho ta 1 cái subcription, mình có thể unSubcribe 1 cái tác vụ của thằng Observable mà chúng ta không muốn nhận dữ liệu nữa
* Observer: bản chất cũng là 1 object, chứa tập hợp các callback, các callback này sẽ giúp chúng ta lắng nghe các giá trị được gửi tới từ producer {next, complete, error}
* Router outlet là nơi mà router nó out cái html ra. Mỗi component như 1 class được khai báo trong 1 module (tương tự như 1 như 1 package trong java). 1 module nếu có các tác vụ cần chuyển hướng sẽ có 1 file routing để cấu hình chuyển hướng đến các trang (các component). Trong ứng dụng của chúng ta, chúng ta chỉ out ra 1 chỗ duy nhất là app-component