Deep dive into Directives:

* So far, we already know that there are two types of directives - Attribute and Structural. We also learned about examples of each of the types, such as \*ngFor, \*ngIf, ngStyle, ngClasss, etc.
* Now we are going to create our own directives. For this, we use a new decorator named @Directive which is imported from @angular/core. Like a component, this also requires a selector. However, these are named in camel-case instead of kebab-case.
* Like components, we can create a directive from the the CLI using ng g d <DIRECTIVE\_NAME>.We also need to add the custom directive to the set of declarations in the app module.
* In the constructor of our custom attribute directive, we can pass a few arguments. One of them is the reference of the element on which this directive sits on. We can use this reference to make the changes we desire.
* We learned that direct manipulation elements is not a good practice. The reason for this is, Angular is not limited to the browser. It can also be used in service workers and these are environments where the DOM is not accessible. In such cases using the native element might throw errors.
* Hence, we have to use an alternative. We can use the Renderer2, which imported from @angular/core, to do this. This , along with the element reference, is used another argument in the constructor.
* This is a class provided by Angular to make customizations to the rendering process. It has quite a few useful functions that allow us to make changes in the underlying element.
* One such function is the setStyle function. It takes 4 parameters - *the native element*, *the style property*, *the style value* and optional *flags* (like !important).
* It is best practice to make changes to the element in the ngOnInit function and also to make constructor arguments private.
* If you want to react on some event that occurs at the referenced element, we can use the @HostListener(EVENT\_NAME) decorator, imported from @angular/core. It takes the event as a string parameter.
* We can also access the properties of the referenced element via @HostBinding(PROPERTY\_NAME) decorator, imported from @angular/core. This way, we can avoid using the Renderer2 class (Its not mandatory though, just an easier way for manipulation).

---------------------------------------------------------------------------------------------------------------------------------

* When it comes to structural directives, we see a \* beside it. This is used by angular to differentiate between the directives. Under the hood, it converts the underlying into ng-template and the displays it based on satisfaction of the conditions.
* We can also create our own structural directives using the same code mentioned above. The constructor here takes in 2 arguments imported from @angular/core - the TemplateRef (the template to be rendered) and the ViewContainerRef (the location in the view where the template is to be rendered).
* One special thing about these directives though, is the way the input is defined. We use the @Input decorator and the we write a setter function for the input like so - set <DIRECTIVE\_NAME> (condition: boolean) {}. Inside the function, we can use functions provided by the ViewContainerRef the display (createEmbeddedView(<TEMPLATE\_REF>)) and hide the template (clear()).
* Finally just use the directive in the HTML like the other structural directives - \*<DIRECTIVE\_NAME>.

ngSwitch:

* This structural directive is used just like a normal switch case. We use it in the property binding form - [ngSwitch]=”<VALUE\_TO\_SWITCH>”.
* Just like a switch in JavaScript, we have cases here - \*ngSwitchCase=”<VALUE>”