**USE**

To modify the structure or properties of HTML elements. Can only be used inside Component Templates.

**We can use only ONE structural directive inside an element!**

To fix this, we can add an extra element, called *ng-container.* This only exists for the directive and is not rendered as an extra element in HTML.

**\*ngFor**

a) Basic use: inside an element, place the directive and reference an array

\*ngFor="let item of items"

b) Alias syntax: the directive has some hidden properties that can be accessed with this syntax. See more here:

https://angular.io/api/common/NgForOf

\*ngFor="let item of item; let i = index;"

**\*ngIf**

Adds or removes HTML elements. Apply the directive inside the element and it will act depending on the condition, e.g. is hidden or shown

\*ngIf="condition"

**[ngClass]**

1. It’s a property/attribute directive. It can apply classes to the element, depending on the logic entered. Example:

[ngClass]="{ active: i===currentPage}"

The *currentPage* is a reference to a variable in the app component. The directive creates an object which is applied then to the element.

1. Can be used with functions

*In the element*

[ngClass]="{ getClass()}"

*In the app component*

getClass(){

    if(something) return 'thisThing';

    if(somethingElse) return 'thatThing';

    if(somethingYetElse) return 'thatOtherThing';

  }

1. Can be used with functions that apply an array of strings to the element

*In the element as above*

*In the app component*

getClass(){

    const classes = [];

    if(something) classes.push('thisThing');

    if(somethingElse) classes.push('thatThing');

    if(somethingYetElse) classes.push('thatOtherThing');

    return classes;

}

**[ngSwitch]**

Works as a switch statement.

1. Put the [ngSwitch] in an element and the condition that will be evaluated.

<div [ngSwitch]="condition">

1. Put the \*ngSwitchCase in child elements

<div \*ngSwitchCase="value">This is displayed</div>

1. Add the default case

<div \*ngSwitchDefault>Default case</div>

**Create directives**

**Attribute Directives**

1. ng generate directive <name>
2. The selector displays what we need to input in the element in order to use the directive
3. Import *ElementRef*  from ‘@angular/core’. This allows us to access the element in which the directive was used. Put the element as a parameter inside the constructor. In order to access the actual element, use the *nativeElement* property.

constructor(private *element*:ElementRef) {}

1. To pass arguments to the directive:
   1. Import the *Input* decorator in the directive

import { Input } from '@angular/core';

* 1. Declare an *@Input()* property and pass the directive’s name as an argument. Therefore, the directive can be used with the property binding syntax directly.
  2. Declare a function that *sets* the property, which makes the code on execution check for any property that is defined while the class instance is constructed. This function takes as a parameter an object which is set in the element.
  3. Iterate through the keys of the object and add or remove any class names that are passed in the element.

@Input(directivesName) set classNames(*classObj*:any){

    for (const key in *classObj*) {

      if (*classObj*[key]) {

        this.element.nativeElement.classList.add(key);

      }

      else{

        this.element.nativeElement.classList.remove(key);

      }

    }

  }

* 1. (the above is what is standard practice, but below it’s simpler code to understand)

Declare some *set* methods that define *@Input* properties and set their values as assigned from the element. In order to use the directive directly with the property binding syntax, pass the directive’s name in the @Input as an argument. Therefore, the program is “redirected” to set the property we define, but externally we just use the directive

@Input(directivesName) set property(*elementValue*:type){

this.element.nativeElement.someProperty = *elementValue*;

}

We cannot set properties inside the constructor right away, because it fires first and finishes. After that, any set of any property will be read, which is too late.

* 1. Use the property binding syntax in the element to pass the values to the *@Input* parameters and the conditions on which they are applied

<div [directivesName] = "'someClass: someCondition'">

**Structural Directives**

1. ng generate directive <name>
2. Import the *ViewContainerRef, TemplateRef and Input* from ‘@angular/core

import { ViewContainerRef, TemplateRef, Input } from '@angular/core';

1. Pass into the constructor 2 parameters:
   1. A reference to the view container, which is the parent element, on which the directive is applied
   2. A reference to the child elements that will be manipulated depending on the directive

constructor(

    private *viewContainer*: ViewContainerRef,

    private *templateRef*: TemplateRef<any>

  ) { }

1. To pass arguments to the directive:
   1. Declare an *@Input()* property and pass the directive’s name as an argument. Therefore, the directive can be used with the property binding syntax directly.
   2. Declare a function that *sets* the property, which makes the code on execution check for any property that is defined while the class instance is constructed. This function takes as a parameter a number which is how many times the child element will be created.
   3. Clear the View Container
   4. Define how the child elements will be created, a loop for example
   5. In the *CreateEmbeddedView* function, the reference to the child element is passed as an argument, together with a context object, which is needed to expose any properties that should be used for aliasing.

@Input('appTimes') set render(*times*:number){

    this.viewContainer.clear();

    for(let i = 0; i<*times*; i++){

      this.viewContainer.createEmbeddedView(this.templateRef,{

        index:i

      });

    }

  }