Deep dive into Data binding:

* As we already learned, we can bind data and events to components in a couple of ways. These bindings are local to the component itself. However, in real world applications, we often have components communicate with each other and have interlinked states. How do we achieve this?
* There are a few ways we can achieve this based on the direction of communication:

1. *Parent-to-child* - If a parent component wants to pass down a property to a child component, we use the @Input() decorator, which is imported from @angular/core. Notice that this directive has parenthesis at the end, which indicates its an internal function call. This directive is used on a property that belongs to the child element. The format is @Input <PROPERTY\_NAME>. In type script, you may have to mention the type of data to expect from this input property. The need to use this decorator is because all the properties of a component is accessible only within the component by default, which is a good thing. We use this decorator to explicitly mention which property can be bound from outside the component. An additional option for this decorator is using an alias for a property that is accessed from outside. This can be done by adding a parameter the decorator like so - @Input(<Alias>).
2. *Child-to-parent* - We can emit events to parent components too. Custom properties and events can be bound both ways for directives and components. EventEmitter is a generic type that is available in @angular/core (generic types usually mention what type of data it can hold) which emits events that are listenable. A property is made listenable by using the @Output() decorator imported from @angular/core. We add a listenable property in the child component, which also has a function that actually emits the event by calling emit(<PARAMETERS>) function of the listenable property, and we add the listener functions at the parent component. An event emitter can be created using new EventEmitter<TYPE>(). The custom events that emitted are bound at parent component HTML. Here also an alias can be assigned using @Output(<Alias>).

View Encapsulation

* When we apply CSS styles to a component, we expect the styles to cascade to all elements in the DOM for the specified selector because that’s how CSS actually works. But in Angular, the styles are only local to the component it is written for.
* Angular adds additional attributes to the component tags that makes it uniquely identifiable. In a way, it emulates a shadow DOM to which these extra properties are added. This is how Angular achieves *View Encapsulation*.
* This behaviour is only in Angular and can be modified. In the @Component decorator, add a property named encapsulation. The value of this property is derived from ViewEncapsulation module imported from @angular/core. It has 3 values - Native, None and Emulated (Default). If we select None, the styles applied in the component will be applied globally.
* This feature is not used very often.

Local References

* This is neat feature that comes with angular that allows us to refer to an element within the component HTML. This reference cannot be used directly in the .*TS* file but can be used anywhere in the HTML file, even as a parameter to a function call.
* A local reference is created by adding the #<REFERENCE\_NAME> attribute to the tag to be referred to. Only the name is used when we refer to that element else where in the HTML.
* Now, if we want to select a tag or a component directly in the .*TS* file, we can use another decorator called @ViewChild() which is imported from @angular/core. This decorator takes the local reference or the component (not as a string) as a parameter (in Angular 8+, we must also pass {static: true/false} based on whether we are going to use this in the ngOnInit function or not. In Angular 9+, the {static: false} parameter is not needed).
* The property to which we assign this decorator is assigned returns an ElementRef type output which is imported from @angular/core.
* Using this decorator, we can get direct access to the underlying DOM elements via the nativeElement property. An important thing to keep in mind is that we should NOT manipulate the DOM elements from here.

Rendering Children from another Component

* Often we may need to render the children of component (anything that is to be contained inside the child component) from a parent component that contains it. The main reason why we may need this is if we want to reuse a particular component for multiple scenarios.
* Consider we have a ‘Tab’ component. Each tab is supposed to have different contents within it and also needs to be used as part of other complex components. Hence, children need to be added dynamically.
* In Angular, we can achieve this by using a directive named ng-content. This directive is added in the component that is to be reused. It acts as hook that is used by Angular to dynamically add the contents or the children specified in the parent component to the current component at the marked location.
* This works similar to an ng-template directive, with the difference being that we specify the actual child contents in the parent component and within the component itself.
* The @ViewChild decorator only works if the element referred to is already a part of the template. If we want to get the reference of element that is rendered using ng-content, we have to use the @ContentChild decorator. It works exactly the same way as @ViewChild, but is used only in this case.

Life-cycle Methods:

* There are a few life-cycle methods that Angular provides that hook into the various phases of the components life-cycle:

1. *ngOnChanges* - This is executed when a component is created and *after* a change in any of the @Input properties.
2. *ngOnInit* - This is executed *once* the component is initialized by Angular (in other words, the object was created). This doesn’t mean that the component is displayed in the screen.
3. *ngDoCheck* - Called during *every* change detection run (whether you clicked a button, a timer fired, etc.). It responds to all the events to detect if there needs to be a change in the template. This may seem inefficient but Angular does it in quite an efficient way.
4. *ngAfterContentInit* - Similar to ngOnInit but with respect to the ng-content directive. Called *after* content gets projected in view. Again, doesn’t mean that the content is displayed. Just initialized by Angular.
5. *ngAfterContentChecked* - Called *every* time when the change detection mechanism has checked the ng-content contents.
6. *ngAfterViewInit* - Called *after* the view renders.
7. *ngAfterViewChecked* - Called *every* time the view and the children have been checked.
8. *ngOnDestroy* - Called *when* a component is about to be destroyed (Removed from the DOM).

* When adding a life-cycle method to a component, it is a good practice to implement the correct interfaces to the component class. For example, for ngOnChanges, implement OnChanges interface.
* The @ViewChild decorator works only after the ngAfterViewInit call. Similarly, the @ConentChild decorator works only after the ngContentInit call.