Contents

[**part 01 – Template Variables and ng-template** 2](#_Toc141940876)

[**part 02 – Template Reference to Other DOM Element** 4](#_Toc141940877)

[**part 03 – @ViewChild** 6](#_Toc141940878)

[**part 04 – Content Projection with ng-content** 9](#_Toc141940879)

[**part 05 – Custom Directive and Renderer** 12](#_Toc141940880)

[**part 06 – Structural Directives Limitations** 14](#_Toc141940881)

[**part 07 – View Encapsulation** 16](#_Toc141940882)

[**Appendix A – Improving Part 06** 20](#_Toc141940883)

[**Appendix B – TypeScript Introduction** 22](#_Toc141940884)

[**Appendix C – Angular Life-Cycle Hooks** 24](#_Toc141940885)

Day02 Working with the DOM

# part 01 – Template Variables and ng-template

Template variables can target various parts of an HTML template such as DOM elements, directives or components. It provides the .ts file with a handle on to the element it targets. This feature is mainly used to respond to user input. In this section we talk about template variables and ng-template. On the first day in Part 07 I hinted at local references and ng-template. Template variables or local references can only be used inside the template. It can of course be passed to the TS code, but only the reference to the HTML element object is passed.

1. For this part, make sure the second-child component is being displayed from the parent app, so in app.component.html, add this line:

|  |
| --- |
| **</div>**  **<app-second-child></app-second-child>**  **</div>** |

1. In the second-child component class, add a Boolean property:

|  |
| --- |
| **export class SecondChildComponent implements OnInit {**  **approved : boolean = true;**  **constructor() { }** |

1. In the template of the same component, add a <span> tag to be displayed based on the value of the variable from #1:

|  |
| --- |
| **<p>second-child works!</p>**  **<span \*ngIf="approved">You are approved!</span>** |

Toggle this variable from the TS code and you will see the content appear/disappear on the browser.

1. To construct an *else* part, we need the angular directive known as ng-template. The ng-template directive usually wraps a piece of HTML code that you want to display or not, based on some condition. Add the following <span> tag wrapped in the ng-template and give it a name of pending:

|  |
| --- |
| **<span \*ngIf="approved">You are approved!</span> <ng-template #pending>**  **<span>Approval Pending...</span>**  **</ng-template>** |

Now we can target this <span> tag from another part of the DOM.

1. That other part of the DOM is the <span> tag just above ng-template, add the *else* part now:

|  |
| --- |
| **<span \*ngIf="approved; else pending">You are approved!</span>**  **<ng-template #pending>** |

Notice the semi-colon. Now when you toggle the value of approved from within the TS code you see a different message each time.

1. When local reference is used, it is possible to access the object the reference points to. Let’s look at local references or template variables. Add a new input element, give it a reference, a value and a class, but the class is not necessary:

|  |
| --- |
| **</ng-template>**  **<div>**  **<label for="approval">Approved**  **<input**  **type="checkbox"**  **#cbApproved**  **value="approved"**  **class="checkbox-lg"**  **>**  **</label>**  **</div>** |

1. We will hook into the click event and pass to a function in the TS code, the entire HTML element as an object:

|  |
| --- |
| **</ng-template>**  **<div>**  **<label for="approval">Approved**  **<input**  **type="checkbox"**  **#cbApproved**  **value="approved"**  **(click)="getApproval(cbApproved)"**  **class="checkbox-lg"**  **>**  **</label>**  **</div>** |

We will write the getApproval() function next. Notice that when we pass cbApproved as a parameter, we are passing the entire input form control as an object.

1. In the logic .ts file, add the getApproval() function to handle this checkbox click event:

|  |
| --- |
| **ngOnInit(): void {**  **}**  **getApproval(){**  **console.log();**  **}** |

1. In order to connect the two, we need to pass a reference to the checkbox into the function:

|  |
| --- |
| **getApproval(cb : HTMLInputElement){**  **console.log(cb.value);**  **}** |

Since we know that the checkbox (cb) is an HTMLInputElement, we specify that in the parenthesis. Once we have access to this type of element, we know that it has a *value* property. This value property has a string value of course.

# part 02 – Template Reference to Other DOM Element

In Part 01 #6 we used a template reference to reference back to the same object in which the reference was an *attribute*. In most cases, however, you will be referencing a totally different HTML element.

1. Still in the second-child.component.html file, change the HTML checkbox element to be a text box instead, or just add a new input form control of type text:

|  |
| --- |
| **</ng-template>**  **<div>**  **<label for="fName">First Name:**  **<input**  **type="text"**  **#fName**  **>**  **</label>**  **</div>** |

1. Add a button and a pair of <span> tags. The idea here is that the user will enter a name, the button is clicked and the name is show between the <span> tags:

|  |
| --- |
| **>**  **</label>**  **</div>**  **<div>**  **<button>Get Name</button>**  **</div>**  **<span></span>** |

1. When the button activates, we tap into it’s click event and pass the textbox over to the TS code, as an HTML element object:

|  |
| --- |
| **<div>**  **<button (click) = "getApproval(fName)">Get Name</button>**  **</div>** |

I am using the same method we created in Part01 (getApproval**)**, but you can change this. Also we did something similar in Part05 of Day01.

1. We will create an fName property in the class, but for now add string interpolation to the HTML file:

|  |
| --- |
| **<div>**  **<button (click) = "getApproval(fName)">Get Name</button>**  **</div>**  **<span>{{fName}}</span>** |

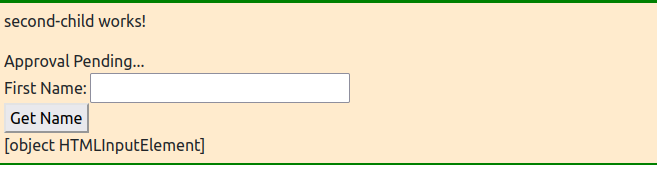
1. In the TS file, add a property to store the value that the user will type into this fName box:

|  |
| --- |
| **export class SecondChildComponent implements OnInit {**  **approved : boolean = false;**  **fName : string = "";**  **constructor() { }** |

1. At the same time, change the getApproval() function to assign the user’s value to the property from #5

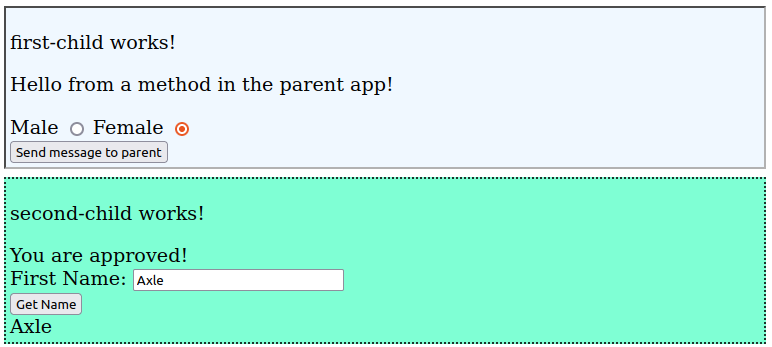
|  |
| --- |
| **getApproval(cb : HTMLInputElement){**  **this.fName = cb.value;**  **}** |

1. When the page refreshes it shows the HTMLInputElement as an object:



1. The reason for this is that on the template, the fName we added between the <span> tags, refers to an object. If we select the value property of that object, we get a string:

|  |
| --- |
| **</div>**  **<span>{{fName.value}}</span>** |



# part 03 – @ViewChild

The reference variable we used so far works but we can only access the referenced HTML element after the page is rendered in the browser. The feature @ViewChild gives us access to elements even before the DOM element is loaded. A related topic to this is Life Cycle Hooks, but we cover some of that later in the day.

1. In second-child.component.ts add the decorator @ViewChild in front of fName:

|  |
| --- |
| **approved : boolean = false;**  **@ViewChild() fName : string = "";**  **constructor() { }** |

1. Next, pass as a string, the name of the template reference variable (from #1 above):

|  |
| --- |
| **approved : boolean = false;**  **@ViewChild('fName') fName : string = "";**  **constructor() { }** |

1. Now once this change is made, fName is no longer a string, it is of type ElemenRef. Lets make this change:

|  |
| --- |
| **approved : boolean = false;**  **@ViewChild('fName') fName : ElementRef;**  **constructor() { }** |

1. At this point of declaration, the IDE will complain since it does not know what fName is referring to exactly. We can fix this is two different ways:

|  |
| --- |
| **@ViewChild('fName') fName : ElementRef | undefined;**  **Or @ViewChild('fName') fName! : ElementRef;** |

The second option is using the non-null assertion operator

1. If you did use @ViewChild, then we will need to get to the value the user typed into the text box, via the nativeElement property of the ElementRef object. Then that property has the attached *value* property and hence the string value:

|  |
| --- |
| **getApproval(cb : HTMLInputElement){**  **console.log(this.fName.nativeElement.value);**  **}** |

1. This decorator can be used with other Directives and Components. When used with Components, the parent component can have access to child components and can even call methods on that child component. Let’s start by adding a method in the secondChild component to just log something:

|  |
| --- |
| **getApproval(elName : HTMLInputElement){**  **console.log(this.fName.nativeElement.value);**  **}**  **calledFromParent(){**  **console.log("Called from Parent");**  **}** |

1. In app.component.ts file, the parent, import the child component at the top, then create a property to reference this child component:

|  |
| --- |
| **@import { Component, ViewChild } from '@angular/core';**  **import { SecondChildComponent } from "./second-child/second-child.component";**  **@Component({**  **…**  **})**  **export class AppComponent {**  **title : string = 'skills2';**  **compStatus : string = "App Component - Top Component";**  **secondChild**  **messageToChildren(){** |

1. Add the @ViewChild() decorator in front of the property added in #7. Also, pass the component name into the parenthesis of @ViewChild() and then point everything to the component itself.

|  |
| --- |
| **title : string = 'skills2';**  **compStatus : string = "App Component - Top Component";**  **@ViewChild(SecondChildComponent) secondChild!:SecondChildComponent;**  **messageToChildren(){** |

In this way, secondChild points to the SecondChildComponent. You will need to import ViewChild from @angular/core.

1. All we have to do now is call the calledFromParent() method on the child component from the parent component. It is recommended that we use a special lifecycle hook method for this, it is called ngAfterViewInit():

|  |
| --- |
| **ngAfterViewInit(){**  **this.secondChild.calledFromParent();**  **}** |



So what happened here. Well there is a function called

calledFromParent() that was coded in the child component, secondChild to be specific. That function was called from the parent using @ViewChild() construction.

1. (Optional) You could add a template reference to the child at the time it is being rendered in app.component.html file:

|  |  |
| --- | --- |
| **</app-first-child> <app-second-child class = "secondChild" #child2></app-second-child>** | **compStatus : string = "App Component - Top Component";**  **@ViewChild('child2') secondChild!:SecondChildComponent; messageToChildren(){** |

Then in app.component.ts file, pass that reference into the @ViewChild parenthesis. In this way the code may be shorter to write and read.

# part 04 – Content Projection with ng-content

An alternative to property binding is Content Projection. It is achieved with the help of a directive called ng-content. With this feature we can pass HTML content directly into the child component using the content part of the component selector. So, wherever we place selectors for a child component, between those tags we can pass additional HTML content. This is done with the help of the ng-content directive.

1. Add a new grand child component like this:

|  |
| --- |
| **ng g c first-child-child --skip-tests** |

This will be the child of the first child component, so a grand child of app component.

1. Add some content to the new component’s template. I added a CSS class as well, in order to make this component stand out a bit:

|  |
| --- |
| **<p>first-child-child (grand Child) works!</p>** |

If you wish to use the CSS class, it should be in the files you downloaded

1. Add a pair of <div> tags around the <p> tag, so that we can apply style to the <div> itself:

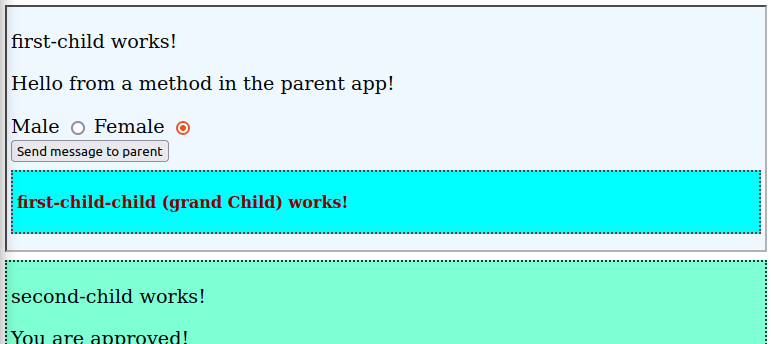
|  |
| --- |
| **<div class="grandChild">**  **<p>first-child-child (grand Child) works!</p>**  **</div>** |

This CSS class will be in the code you downloaded

1. We will call this grand child from first child, so between the selector tags of the grand child, add some HTML content (do this in first-child):

|  |
| --- |
| **Send message to parent**  **</button>**  **<app-first-child-child>**  **<h3>Hello grand child :0</h3>**  **</app-first-child-child>**  **</div>** |

The outer custom HTML elements will be what Angular provide to us. The <h3> in the middle is the HTML we want to project into the child component.



1. So far, the child component did not show the <h3> content. However if we add a pair of directives (in the grand child), it works:

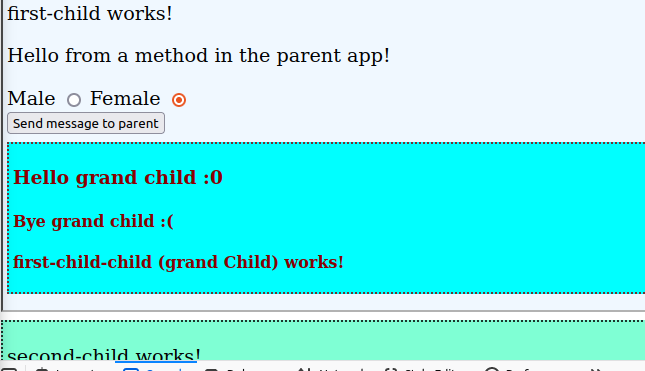
|  |
| --- |
| **<div class="grandChild">**  **<ng-content></ng-content>**  **<p>first-child-child (grand Child) works!</p>**  **</div>** |

Since we placed the ng-content tags above the <p> tag, the *hello* message appears above the original child content. (ng-content can be placed in several places, here I placed in inside of the upper <div>).

1. What if we wanted to add a second (or more) HTML tag and wanted it projected into the child component:

|  |
| --- |
| **<app-first-child-child>**  **<h3>Hello grand child :0</h3>**  **<h4>Bye grand child :(</h4>**  **</app-first-child-child>** |

Since we placed the ng-content tags above the <p> tag, the *hello* message appears above the original child content. Do this in first child.

1. Well, both pieces of content just appear one on top of the other, but maybe we wanted the goodbye message to be below the original child content in brown:  
     
   
2. We could add another directive below the original content, in the grand child. This would now push both messages below the original child content:

|  |
| --- |
| **<ng-content></ng-content>**  **<p class="grandChild">first-child-child (grand Child) works!</p>**  **<ng-content></ng-content>** |

1. One way to distinguish between the two is to use what we already have. The hello message is stuck between a pair of <h3> tags while the goodbye message is between <h4> tags. Add a new attribute to the ng-content custom tag that will identify each message:

|  |
| --- |
| **<ng-content select="h3"></ng-content>**  **<p class="grandChild">first-child-child (grand Child) works!</p>**  **<ng-content select="h4"></ng-content>** |

1. Another way to distinguish between the two is to add different CSS classes to the individual ng-content directives. Since each class has a different name, we can identify each element by using their class name. Note, you DO NOT have to have an actual CSS class for this to work:

|  |
| --- |
| **.helloMessage {**  **color:blue;**  **margin-top: 20px;**  **font-family: Verdana, Geneva, Tahoma, sans-serif;**  **}**  **.byeMessage {**  **color:green;**  **margin-top: 20px;**  **font-family: 'Franklin Gothic Medium', 'Arial Narrow', Arial, sans-serif;**  **}** |

Add these two classes inside of first-child-child.component.css file.

1. Now add those classes to the selector tags on the parent:

|  |
| --- |
| **<app-first-child-child>**  **<h3 class="helloMessage">Hello grand child :0</h3>**  **<h4 class="byeMessage">Bye grand child</h4>**  **</app-first-child-child>** |

1. Then in the child component:

|  |
| --- |
| **<ng-content select=".helloMessage"></ng-content>**  **<p class="grandChild">first-child-child (grand Child) works!</p>**  **<ng-content select=".byeMessage"></ng-content>** |

Don’t forget the period in this component

# part 05 – Custom Directive and Renderer

With Angular we can develop our own custom directives. Available are attribute and structure directives. Attribute directives affect how HTML elements are rendered in the browser. Structural directives affect how the DOM builds itself by controlling the addition or removal of elements.

Appendix C has more information on Directives.

As a reminder, directives can be used just like an HTML tag, a class or as a directive on an existing element. This part focusses on the latter.

In this example we will create a directive that will change the text of an element when the element is hovered over by the mouse.

A note of caution. It is possible to manipulate the DOM directly using ElementRef and the nativeElement property of that element. In this method, a life cycle hook is completely bypassed. Although this works, it is not recommended. This example shows the recommended method, at this point in time.

1. Use the CLI to generate a new directive

|  |
| --- |
| **ng generate directive boldNBlue –-skip-tests** |

This command will create the class that will hold the logic for this hover action and will also amend the app.module.ts file so that we can use this directive in any component in the application.

1. In the new file, bold-nblue.directive.ts, we need to import a few classes from @angular/core:

|  |
| --- |
| **import { Directive, ElementRef, OnInit, Renderer2} from '@angular/core';**  **@Directive({** |

Enable custom UI changes with Renderer2

1. With the CLI version of Directive classes, the OnInit class is NOT implemented by default. Add it in the export statement:

|  |
| --- |
| **@Directive({**  **selector: '[appBoldNBlue]'**  **})**  **export class CHoverDirective implements OnInit {** |

When we create components, this is done automatically. Also notice that appBoldNBlue has square brackets around it. This makes this directive applicable by just adding it as an attribute on the DOM.

1. Add the two main objects into the constructor. We need access the DOM element that this directive is going to be attached to and also we need the supporting class called Renderer2 from @Angular/Core:

|  |
| --- |
| **export class CHoverDirective implements OnInit {**  **constructor(private elRef: ElementRef, private renderer:Renderer2) { }**  **ngOnInit(){** |

1. Inside of the ngOnInit() method, we can call the setStyle method of Renderer2 and pass it the three parameters it requires:

|  |
| --- |
| **constructor(private elRef: ElementRef, private renderer:Renderer2) { }**  **ngOnInit(){**  **this.renderer.setStyle(this.elRef.nativeElement, 'color', 'blue');**  **this.renderer.setStyle(this.elRef.nativeElement, 'font-weight', 'bold');**  **}** |

This method, setStyle() requires the element that is the target of this directive, the property we want to change and the value of that property.

**The next modification is optional, time permitting.**

1. It is possible to pass a value directly into the child class and therefore make the code more configurable. In this way, we can pass in the colour of choice into the directive.

Add the Input class to the bond-nblue.directive.ts file and create a class property to hold the value being passed in:

|  |
| --- |
| **import { Directive, ElementRef, Input, OnInit, Renderer2 } from '@angular/core';**  **@Directive({**  **selector: '[appBoldNBlue]'**  **})**  **export class BoldNBlueDirective implements OnInit {**  **@Input() appBoldNBlue : string = "";** |

This is called *custom property binding*. Refer to Day01 Part08.

1. We can now wrap the original two lines into the listen() method of renderer. This method takes the object to listen on and the event to listen to. The third parameter is a function that will execute once the event occurs on that object:

|  |
| --- |
| **ngOnInit(){**  **this.renderer.listen(this.elRef.nativeElement, "mouseenter", () => {**  **this.renderer.setStyle(this.elRef.nativeElement, 'color', this.appBoldNBlue);**  **this.renderer.setStyle(this.elRef.nativeElement, 'font-weight', 'bold');**  **});**  **}** |

Notice that the value passed in by the user template is used in the setStyle() method of renderer.

1. Now just add a similar block of code for the mouseleave event:

|  |
| --- |
| **ngOnInit(){**  **this.renderer.listen(this.elRef.nativeElement, "mouseenter", () => {**  **this.renderer.setStyle(this.elRef.nativeElement, 'color', this.appBoldNBlue);**  **this.renderer.setStyle(this.elRef.nativeElement, 'font-weight', 'bold');**  **});**  **this.renderer.listen(this.elRef.nativeElement, "mouseleave", () => {**  **this.renderer.setStyle(this.elRef.nativeElement, 'color', '');**  **this.renderer.setStyle(this.elRef.nativeElement, 'font-weight', '');**  **});**  **}** |

Note, the name of the property decorated by @Input is the same name as the custom directive.

1. Finally for this section, pass in the colour of choice when the directive is used. So this code is on the first-child-child.component.html file:

|  |
| --- |
| **<ng-content select=".helloMessage"></ng-content>**  **<p class="grandChild" [appBoldNBlue]="'green'">first-child-child (grand Child) works!</p>**  **<ng-content select=".byeMessage"></ng-content>** |

# part 06 – Structural Directives Limitations

There are situations where you may want to apply two or more structural directives. Obviously this is not allowed. In the example below we attempt to display a list of items using \*ngFor but then restrict what is displayed using \*ngIf.

1. We will use the first-child-child for this. Add an array called stuff that holds two types of data, either strings or numbers:

|  |
| --- |
| **export class FirstChildChildComponent implements OnInit, OnDestroy {**  **constructor() { }**  **stuff: (string | number)[] =**  **['Apple', 2, 'Orange', 3, 4, 'Banana', 5, "Peach", "Melon"];**  **ngOnInit(): void {**  **}** |

1. In the template we can display the elements of the stuff array using this code:

|  |
| --- |
| **<ng-content select=".byeMessage"></ng-content>**  **<ng-template ngFor [ngForOf]="stuff" let-item>**  **{{item}}**  **</ng-template>** |

Using ng-template is just one strategy, there are others. Note, the use of let here is only supported on ng-template.

1. What if we wanted to display only the string items of that array or maybe the numeric items. Well, first we need to add a method in the class to return true or false since we cannot do heavy logic in the template. In the component, add this method:

|  |
| --- |
| **isNumber(val:number|string): boolean {**  **return typeof val === 'number';**  **}** |

We can now call this method from the template.

1. The following code will NOT work. It appears logical to do this, but remember this is a structural angular directive

|  |
| --- |
| **<ng-template ngFor [ngForOf]="stuff" let-item \*ngIf="isNumber(item)">**  **{{item}}**  **</ng-template>** |

1. We would need to split up the directives, so add a pair of <div> tags and attach the \*ngIf to that div:

|  |
| --- |
| **<ng-template ngFor [ngForOf]="stuff" let-item>**  **<div \*ngIf="isNumber(item)">**  **{{item}}**  **</div>**  **</ng-template>** |

Of course, to get the string values from the array, use "!isNumber(item)

**Other Options**

1. Remembering that *let* is only supported on ng-template, if we used ng-container, the code will have to change a bit:

|  |
| --- |
| **<ng-container \*ngFor = "let item of stuff">**  **<div \*ngIf=!isNumber(item)>**  **{{item}}**  **</div>**  **</ng-container>** |

1. The simplest way to handle this situation of course is with two pairs of <div> tags:

|  |
| --- |
| **<div \*ngFor = "let item of stuff">**  **<div \*ngIf=!isNumber(item)>**  **{{item}}**  **</div>**  **</div>** |

If the display on the browser is picking up the div style, you can turn it off from app.component.ts by changing the property: encapsulation:ViewEncapsulation.Emulated (Part07)

1. As a reminder we could use ngStyle together with \*ngIf:

|  |
| --- |
| **<div \*ngFor = "let item of stuff">**  **<div [ngStyle]="{color: isNumber(item) ? 'blue' : 'brown'}">**  **{{item}}**  **</div>**  **</div>** |



# part 07 – View Encapsulation

CSS styles, have the ability to be applied globally. In Angular and other similar platforms, this may not be a desirable behavior. Angular in particular has given us the ability to apply individual styles to components. This is called *view encapsulation*. Of course we can override this.

We will use the first-child-child for this demonstration then switch to the app component.

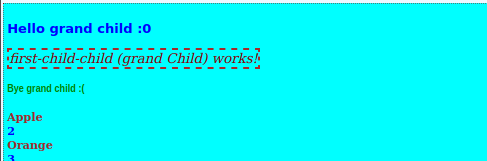
1. Here is a simple style being applied to a <span> tag, add it to the   
   first-child-child.component.css file:

|  |
| --- |
| **span{**  **font-style: oblique;**  **font-weight: 500;**  **font-size: larger;**  **border: 4px;**  **border-color: brown;**  **border-style: dashed;**  **}** |

1. Change the <p> tag to a <span> tag in the first-child-child.component.html file:

|  |
| --- |
| **<ng-content select=".helloMessage"></ng-content>**  **<span [appBoldNBlue]="'green'">first-child-child (grand Child) works!</span>**  **<ng-content select=".byeMessage"></ng-content>**  **<div \*ngFor = "let item of stuff">** |

1. When you refresh, the style gets applied **only** to the Span-tag inside of first-child-child component:



1. Now cut and paste the same styles into the app.component.css file:

As you can see from the image below, the style is being applied to those Span-tag elements present in the top parent template.

Recall also that if you click on the phrase “Blue is our colour!” , a new paragraph appears, identified by the downward point arrow.

You can toggle this to see the words “Hello Skillsoft” appear with the style that we just added:

1. Angular can achieve this kind of encapsulation by adding an *attribute* to the component’s elements. If you inspect your page (Inspector in Firefox) you will see a unique attribute added to blocks of HTML content. This is how Angular is able to tell one component apart from another:

A screenshot of a computer

Description automatically generated with medium confidence

1. By default, Angular used *emulated* view encapsulation. This is a property of the @Component class that can be changed. Go to app.component.ts file and specifically add the encapsulation property:

|  |
| --- |
| **import { SecondChildComponent } from "./second-child/second-child.component";**  **@Component({**  **selector: 'app-root',**  **templateUrl: './app.component.html',**  **styleUrls: ['./app.component.css'],  encapsulation:ViewEncapsulation.Emulated**  **})**  **export class AppComponent {** |

ViewEncapsulation package must be imported from @angular/core

1. Refresh the page and you will notice that nothing has changed, this is the default behavior. Now change the value from Emulated to None:

|  |
| --- |
| **import { SecondChildComponent } from "./second-child/second-child.component";**  **@Component({**  **selector: 'app-root',**  **templateUrl: './app.component.html',**  **styleUrls: ['./app.component.css'],  encapsulation:ViewEncapsulation.None**  **})**  **export class AppComponent {** |

This is the result on my system after I changed Emulated to None:



# Appendix A – Improving Part 06

It is possible to use @HostListener and @HostBinding to improve the code from Part06 above.

In this section we will insert a pair of Radio buttons and have the value of the clicked button sent to the parent component.

1. First import HostListener:

|  |
| --- |
| **import { Directive, ElementRef, HostListener, Input, OnInit, Renderer2 } from '@angular/core';**  **@Directive({** |

1. Then in the class, use the decorator function @HostListener():

|  |
| --- |
| **@HostListener('mouseenter') mouseover(someEvent:Event){**  **this.renderer.setStyle(this.elRef.nativeElement, 'color', this.appBoldNBlue);**  **this.renderer.setStyle(this.elRef.nativeElement, 'font-weight', 'bold');**  **};**  **@HostListener('mouseleave') mouseleave(someEvent:Event){**  **this.renderer.setStyle(this.elRef.nativeElement, 'color', '');**  **this.renderer.setStyle(this.elRef.nativeElement, 'font-weight', '');**  **};** |

Notice that the lines in the middle are the same as it was in Part06.

1. At the import line, import the HostBinding class from @Angular/Core:

|  |
| --- |
| **import { Directive, ElementRef, HostBinding, HostListener, Input, OnInit, Renderer2 } from '@angular/core';**  **@Directive({** |

HostBinding will bind a property of the component class to a host element. If the property’s value change, HostBinding will update the host element.

1. Over in the class, add two properties decorated by the decorator function. Notice that the property we are binding to is the HTML element property:

|  |
| --- |
| **export class BoldNBlueDirective implements OnInit {**  **@Input() appBoldNBlue : string = "";**  **@HostBinding('style.color') color : string = "";**  **@HostBinding('style.fontWeight') bold : string = "";** |

1. Here is the final bold-nblue.directive.ts file:

|  |
| --- |
| **import { Directive, ElementRef, HostBinding, HostListener, Input, OnInit, Renderer2 } from '@angular/core';**  **@Directive({**  **selector: '[appBoldNBlue]'**  **})**  **export class BoldNBlueDirective implements OnInit {**  **@Input() appBoldNBlue : string = "";**  **@HostBinding('style.color') color : string = "";**  **@HostBinding('style.fontWeight') bold : string = "";**  **constructor(private elRef: ElementRef, private renderer:Renderer2) { }**  **ngOnInit(){ }**  **@HostListener('mouseenter') mouseover(someEvent:Event){**  **this.color = this.appBoldNBlue;**  **this.bold = "bold";**  **};**  **@HostListener('mouseleave') mouseleave(someEvent:Event){**  **this.color = "";**  **this.bold = "";**  **};**  **}** |

1. Think of HostBinding as [someValue]="myValue". But the HostBinding equivalent code will be HostBinding(someValue) myValue. In this case we are working with class properties.
2. Think of HostListener as (click)="someClick()". But the HostListener equivalent code will be HostListener('click') someClick (){ }. In this case we are working with class methods.

# Appendix B – TypeScript Introduction

Most of the folks on this call already know that TypeScript is a superset of JavaScript. This means that any JS code you see, is also TS code. The purpose of TS is to make you a better JS programmer. The main idea behind TS is static typing. This means that we must declare the data type in advance and it cannot (should not) be changed. Static typing means that we get IDE support, so we can eliminate certain bugs early in development.

1. The primitive types in TS is considered to be number, string and Boolean. Add a new .ts file to your project and add the following lines:

|  |
| --- |
| **let firstName : string;**  **let maxAllowed : number;**  **let isAuthorized : boolean;** |

Notice the types are all lower case to specify the type and not the object. Two other types which are considered primitive are *null* and *undefined*.

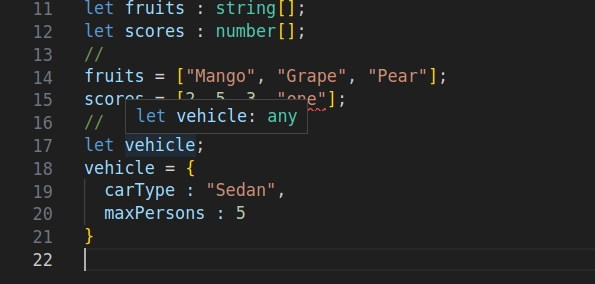
1. Complex types include arrays and objects.

|  |
| --- |
| **let fruits : string[];**  **let scores : number[];**  **fruits = ["Mango", "Grape", "Pear"]; scores = [2, 5, 3, "one"];//NOT possible, strings NOT allowed** |

1. The object type can be a composite of other types. Below we declare a vehicle type and then attempt to define what the type should look like:

|  |
| --- |
| **fruits = ["Mango", "Grape", "Pear"];**  **scores = [2, 5, 3, "one"];**  **//**  **let vehicle;**  **vehicle = {**  **carType : "Sedan",**  **maxPersons : 5**  **}** |

The problem with this approach is that TS is left to define the type, and it will define this vehicle type as any, see image below:



1. The advantage in this situation of using TS is that we can now employ Object Type Definition, in other words we describe what our object should look like:

|  |
| --- |
| **let vehicle : {**  **carType : string,**  **maxPersons : number**  **};**  **vehicle = {**  **carType : "Sedan",**  **maxPersons : 5**  **}** |

So now vehicle is valid.

1. The function below has two parameters that will be interpreted as type any:

|  |
| --- |
| **function modifiedType(p1, p2) {**  **return p1 + p2;**  **}** |

1. We can make it a bit clearer by declaring that the parameters are indeed of type any:

|  |
| --- |
| **function modifiedType(p1:any, p2:any): any {**  **return p1 + p2;**  **}** |

The return type is also any

1. The IDE does not complain if we call the function in #6 with any of these statements

|  |
| --- |
| **return p1 + p2;**  **}**  **modifiedType(10, "Hello");**  **modifiedType(true, "Hello");**  **modifiedType("Hello", {});** |

In some cases this is exactly what we need

1. With Generics, we can create our function and then decide at the time we call the function, exactly what type we want it to work with:

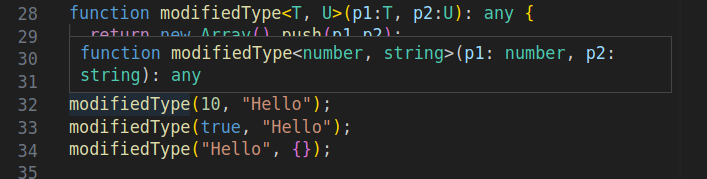
|  |
| --- |
| **function modifiedType<T, U>(p1:T, p2:U): any {**  **return p1 + p2;**  **}** |

Now, the return will not work. Typescript knows that it cannot add or concatenate certain types, for example object types.

1. Lets return an array instead, but one that contains both parameters:

|  |
| --- |
| **function modifiedType<T, U>(p1:T, p2:U): any {**  **return new Array().push(p1,p2);**  **}** |

Now, if you hover over the three calls to the function, you will see that the IDE has made a conclusion of what the types are based on when the function is called and what is being passed into the function. This is the beginnings of Generics.



In the image above notice that the interpretation of line 32 is that we are calling the modifiedType function with a number and a string. Not only that, we are guaranteed that the array being returned will have a number and a string.

# Appendix C – Angular Life-Cycle Hooks

Angular life-cycle hooks are well documented on their website located at: <https://angular.io/guide/lifecycle-hooks>

In this section I will just touch on the hooks that affect the Component and give a few examples of when each hook is executed. The class constructor is NOT a life-cycle hook. In the code below we will add a console.log() statement in the constructor to see when it gets executed, relative to other hooks. It should be noted though, that the constructor should not be used to perform heavy work. It is suggested that it be used to initialize simple variables local to the class itself. Just like any other programming language, the constructor gets called when the object is created in memory.

ngOnInit()

We have used this hook more than any other, so let’s start here. This method is called after any data-bound properties have been initialized and settled. For example in Day01 Part 08 #1, we had the following line in the TS file:

**messageFromParent : string = "";**

This property was set from the parent’s template, see the same part, but #4

**<app-first-child class = "firstChild" [messageFromParent]="messageToChildren"></app-first-child>**

In this case messageFromParent is a data-bound property and it is initialized when the value messageToChildren was applied to it.

This brings us to the point when this method is fired. It is fired once when the component is created and then once again after Angular runs it’s change detection procedure. Note, even though the component is created, it is NOT yet on the DOM.

Again, on Day01 Part10 #8 we used the ngOnInit() method since we wanted to fire the eventEmitter and therefore initialize the DOM with the Female value:

|  |
| --- |
| **…**  **selectedRBValue : string = "Female";**  **constructor() { }**  **ngOnInit(): void {**  **this.eventEmitter.emit(this.selectedRBValue);**  **}**  … |

As you can see, once the component is created, we immediately pass the selectedRBValue’s value to the DOM. Remember the DOM was two-way bounded to the selectedRBValue property.

Since this is the place where all initial properties are set, it is recommended that you perform API calls or any other HTTP requests from here.

This method is called AFTER ngOnChanges() and of course, the constructor method.

At the moment ngOnInit() executes, none of the child components or projected content is available. So if there are any properties in the class that were decorated with @ViewChild, @ViewChildren, @ContentChild and @ContentChildren, those properties will not be available and may produce null results.

In order to demonstrate these phases of the component, I added a log() statement in the constructor of second-child and in the ngOnInit() method:

|  |
| --- |
| **export class SecondChildComponent implements OnInit {**  **approved : boolean = false;**  **@ViewChild('fName') fName! : ElementRef;**  **constructor() {**  **console.log("Constructor of second-child executed");**  **}**  **ngOnInit(): void {**  **console.log("fName in second-child? " + this.fName);**  **}** |

If you did this yourself, the ngOnInit() method will show the value of fName as undefined. You can use any child component for this, here I am using second child.  
A screenshot of a computer error

Description automatically generated

Remember that @ViewChild gives the component class access to a DOM element, as an object. Here, since the DOM is not yet created, we get undefined.

ngOnChanges

As noted on the Angular website, this method fires on a regular basis. Once when the component is built and then once again anytime a property is changed. This method should be used with caution since it can slow down your App’s performance.

This is the only life cycle method that takes a parameter. Once executed, this method will look for a SimpleChanges object passed in as a parameter. This parameter holds current and previous values. Think of it as keeping state at least temporarily.

This method is called **before** ngOnInit() if the component has bound inputs. So basically, any property that has the @Input decorator in front is the trigger for this method. Whenever these properties change this method fires.

In order to demonstrate this method, we will use the first-child component since it has the messageFromParent as a property decorated with @Input.

Remove or comment out all of the log lines in second-child.component.ts file if you just want to see the ngOnChanges method only.

1. Import the various classes and interfaces from @angular/core into first-child.component.ts file:

|  |
| --- |
| **import { Component, OnInit, Input, Output, EventEmitter, SimpleChanges, OnChanges } from '@angular/core';** |

SimpleChanges needs to be passed into the ngOnChanges() method if used.

1. Implement the OnChanges interface:

|  |
| --- |
| **styleUrls: ['./first-child.component.css']**  **})**  **export class FirstChildComponent implements OnInit, OnChanges {**  **@Input() messageFromParent : string = "";** |

1. Add a method in the same file to track ngOnChanges() method:

|  |
| --- |
| **selectedRBValue : string = "Female";**  **constructor() { }**  **ngOnChanges(change: SimpleChanges){**  **console.log("ngOnChanges called");**  **}** |

1. Also add a log statement to see when ngOnInit() is called:

|  |
| --- |
| **ngOnInit(): void {**  **this.eventEmitter.emit(this.selectedRBValue);**  **console.log("ngOnInit called");**  **}** |

A screenshot of a computer error message

Description automatically generated

You may add a log statement to see what is inside the change property.

1. In order to demonstrate the ngOnChangs feature even furher, let’s continue with the first child and the parent app component. The current first-child.component.ts file does have a property decorated with @Input, but let’s create a new one:

|  |
| --- |
| **export class FirstChildComponent implements OnInit, OnChanges {**  **@Input() messageFromParent : string = "";**  **@Input() someValue : string = "some value";**  **messageFromFirstChild : string = "Message from first-child";  @Output() eventEmitter = new EventEmitter<string>();**  **isChecked: boolean = false;** |

Remember to implement the OnChanges interface.

1. In the template file, so first-child.component.html, display this new property via string interpolation:

|  |
| --- |
| **<p>first-child works!</p>**  **<p>{{messageFromParent}}</p>**  **<p>{{someValue}}</p>**  **<div>** |

1. In the parent template, so app.component.html add a mechanism to change this value in the child. Basically add an input control to accept a value from the user and then pass the entire element to a function in the class (coded in the next section):

|  |
| --- |
| **<span class = "topComponent">{{ title }} app is running!</span> <div>**  **<input type="text" #pInput />**  **<button (click)="OnSubmit(pInput)">Click Me!</button>**  **<div>**  **<app-first-child**  **class = "firstChild" [messageFromParent]="messageToChildren()"**  **(eventEmitter) = "messageFromChildren($event)"**  **[someValue]="someText"**  **>**  **</app-first-child>** |

Notice that when the button (above) is clicked, the entire <input> element is passed to the class function. When first-child is instantiated via this parent template, we bind *someValue* to *someText*. The value of someText is set in the parent, but passed into the child. This is done in the OnSubmit() method of the parent. The pair of wrapper <div> tags is just to the input box is not jammed up next to other elements.

1. Also in in app.component.ts file, add a new property to hold the value that the user will type into the form control we added in #7 above:

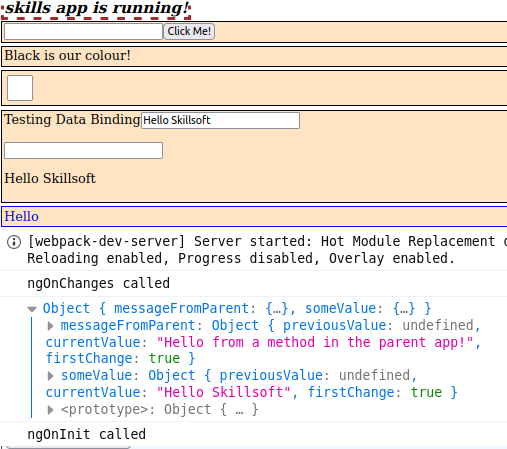
|  |
| --- |
| **divColor : string = "Black";**  **approved : boolean = false;**  **userInput : string = "Hello Skillsoft";**  **messages : string[] = ["Hello", "from", "Skillsoft"];**  **@ViewChild(SecondChildComponent) secondChild!:SecondChildComponent;** |

Here I just initialized userInput with a string value.

1. Now lets use custom property binding to pass the value the user types into the input control to the userInput property on the parent component:

|  |
| --- |
| **divColor : string = "Black";**  **approved : boolean = false;**  **userInput : string = "Hello Skillsoft";**  **messages : string[] = ["Hello", "from", "Skillsoft"];**  **@ViewChild(SecondChildComponent) secondChild!:SecondChildComponent;** |

Here I just initialized userInput with a string value.

1. When the app first runs, we see the log from the ngOnChanges() method first. If we expand the SimpleChanges object , we see that someValue initially is undefined. But then remember that the EventEmitter will fire from the parent template. This in turn triggers someValue to be initialized with whatever is assigned to userInput, which in this case is “Hello Skillsoft”.  
     
   
2. Now, add something inside of the input text box and click the **Click Me** button:A screenshot of a computer program

   Description automatically generated

Notice now that the previousValue is not undefined but it holds the value before the user entered something into the input control. The user entered Test and it is now the currentValue.

ngDoCheck

Use the DoCheck hook to detect and react to changes that Angular doesn't catch on its own. In other words force Angular to run certain code. This method is very similar to ngOnChange(). It will run every time an event is fired whether a value got changed, or not.

This method is Called immediately after ngOnChanges() on every change detection cycle, and just after ngOnInit() the first time the component is created.

1. In the first-child.component.ts file, add the method and log an appropriate message:

|  |
| --- |
| **generateGenderMessage(event : Event) {**  **this.selectedRBValue = (<HTMLInputElement>event.target).value ;**  **this.eventEmitter.emit(this.selectedRBValue);**  **}**  **ngDoCheck(){**  **console.log("ngDoCheck was run");**  **}** |

1. This is what the console window looked like on my machine:

A screenshot of a computer

Description automatically generated

Notice that the method was ran two times. The reason is that Angular loaded core.mjs file and that was a trigger for ngDoCheck().

1. One interesting thing about ngDoCheck() is, if you click any of the buttons on the view, this method runs. So regardless of whether a property is changed or not, as long as it detects an event, the method runs. Because of this, use this method only if absolutely necessary. Another important trigger for this method is when an API returns with data.

ngAfterContentInit, ngAfterContentChecked, ngAfterViewInit, ngAfterViewChecked

1. Here is the code for the rest of the lifecycle hooks, except ngOnDestroy:

|  |
| --- |
| **export class FirstChildComponent implements OnInit, OnChanges, DoCheck {**  **@Input() messageFromParent : string = "";**  **@Input() someValue : string = "some value";**  **messageFromFirstChild : string = "Message from first-child";**  **@Output() eventEmitter = new EventEmitter<string>();**  **isChecked: boolean = false;**  **selectedRBValue : string = "Female";**  **constructor() { }**  **ngOnChanges(change: SimpleChanges){**  **console.log(change);**  **}**  **ngOnInit(): void {**  **this.eventEmitter.emit(this.selectedRBValue);**  **}**  **generateChildMessage() {**  **this.eventEmitter.emit(this.messageFromFirstChild);**  **}**  **generateGenderMessage(event : Event) {**  **this.selectedRBValue = (<HTMLInputElement>event.target).value ;**  **this.eventEmitter.emit(this.selectedRBValue);**  **}**  **ngDoCheck(){**  **console.log("ngDoCheck was run");**  **}**  **ngAfterContentInit(){**  **console.log("ngAfterContentInit was run");**  **}**  **ngAfterContentChecked(){**  **console.log("ngAfterContentChecked was run");**  **}**  **ngAfterViewInit(){**  **console.log("ngAfterViewInit was run");**  **}**  **ngAfterViewChecked(){**  **console.log("ngAfterViewChecked was run");**  **}**  **}** |

ngOnDestroy

This method is fired just before the component, directive, pipe, or service is destroyed. The method must be configured on the object that has to be destroyed, but can be called from a parent object.

Just like the *finally* block, in a try/catch enclosure, this method is used to perform cleanup work:

Unsubscribe from Observables and/or DOM events

Stop any interval timers

Unregister any callbacks that the directive registered in the global space

Notify some other part of the application that the component will no longer be a part of the application

In the example below we will destroy the grand-child, so the child component of first-child. Note, I am using existing methods, so the method naming will probably not make much sense, but the mechanics of destroying a component is what I am trying to demonstrate here.

1. In the first-child.component.ts file, add a class property that will be of the Boolean type:

|  |
| --- |
| **messageFromFirstChild : string = "Message from first-child";**  **@Output() eventEmitter = new EventEmitter<string>();**  **isChecked: boolean = false;**  **destroyMe: boolean = true;**  **selectedRBValue : string = "Female";** |

1. This component already has a method called generateChildMessage(). We will use this method to turn on/off the destoyMe property:

|  |
| --- |
| **generateChildMessage() {**  **this.eventEmitter.emit(this.messageFromFirstChild);**  **this.destroyMe=false;**  **}** |

1. Remember that the first-child component creates the first-child-child component via the first-child’s template. We could now bind this class property to that first-child-child component creation code:

|  |
| --- |
| **<app-first-child-child \*ngIf="destroyMe">**  **<h3 class="helloMessage">Hello grand child :0</h3>**  **<h4 class="byeMessage">Bye grand child</h4>**  **</app-first-child-child>** |

1. There is nothing to do on the first-child-child.component.ts file except if we wanted to have a message display in the console window. Since this is the component that the OnDestroy() method is called, the interface must be imported and implemented:

|  |
| --- |
| **export class FirstChildChildComponent implements OnInit, OnDestroy {**  **constructor() { }**  **ngOnInit(): void { }**  **ngOnDestroy(){**  **console.log("grandchild destroyed");**  **}**  **}** |

1. By clicking the “Send message to parent” button on the first-child component, the grand child will be removed from the dom.  
     
   A white paper with black lines

   Description automatically generated