**Introduction to Attribute Directives**

Let’s learn about the foundations of attribute directives and how they work.

**We'll cover the following**

* [Attribute directives](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5535717331828736#Attribute-directives)
* [Build an attribute directive](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5535717331828736#Build-an-attribute-directive)
* [Apply the directive](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5535717331828736#Apply-the-directive)
* [Summary](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5535717331828736#Summary)

This chapter is entirely about **attribute directives** in Angular and the basic concepts surrounding them. In this lesson, we’ll cover these two things:

* Pass data
* Handle events

Finally, we’ll create our own attribute directive. Let’s go!

**Attribute directives**

Let’s review the three types of directives in Angular, which are as follows:

* **Components**
* **Attribute directives**
* **Structural directives**

Attribute directives can change the appearance or behavior of any existing element. So, we can use them on standard HTML elements in templates and other components, or even trigger other directives! They’re very flexible because of the possibilities given by the directive’s selector.

Right now, let’s start with examples of elementary directives to smoothly introduce the concept of attribute directives. We’ll cover very interesting use cases later on that can help implement larger projects, and avoid unnecessary duplications or unclear architecture.

In this lesson, we’ll use an example of a directive that can be used on images to set their size to an application’s standard avatar size, which is 64 x 64 px.

**Build an attribute directive**

To generate a directive file, we can use the **Angular CLI command**. Or, we can make a file of our own and write the code from scratch. Let’s use the recommended option of using the Angular CLI command. First, however, let’s see how it’s done step by step so that we can follow the same steps to make our own file.

Here’s the command that generates a directive using Angular CLI:

ng generate directive nameOfYourDirective

It also has a simplified version, which looks like this:

ng g d nameOfYourDirective

In our case, we use this:

ng generate directive avatar

By default, this command generates the two files below:

* The **directive class** is avatar.directive.ts
* The **directive test class** is avatar.spec.directive.ts We’ll focus on setting the directive class. Here’s what that looks like:

import { Directive } from '@angular/core';  
  
@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective {  
  constructor() { }  
}

We can use this initial template if we create the file by ourselves.

Angular, by default, generates directive code with the selector built as an attribute selector, but we can change it later to whatever suits us best. This is just the default case because it’s a very popular use case.

Let’s get back to our example. We want to use this directive on images. So, in our templates, we want to set the size of the <img> elements to 64px \* 64 px. How should we manipulate the size of an element in the DOM? Usually, we do so by setting the appropriate style, like this:

img {  
    width: 64px;  
    height: 64px;  
}

However, we want to set the style in our directive and not the stylesheet. We can still change the styles of the element, even without using a stylesheet. If we get the HTML element in our TypeScript code, we can interfere with its styles implicitly.

The question is, **how do we get the element?**

Let’s think of how we would approach this problem in the world of components. Let’s assume we need a reference to the component’s host element. In those cases, we inject ElementRef into the component’s constructor to get access to its host element, like this:

@Component({  
  selector: 'app-button',  
  templateUrl: './button.component.html',  
  styleUrls: ['./button.component.css']  
})  
export class ButtonComponent {  
    constructor(public elementRef: ElementRef){}  
}

Usually, if we find a solution to our problem that works for components, it also works for directives. This is because the components are simply specific kinds of directives.

So, for our directive, we simply inject the same reference:

import { Directive, ElementRef } from '@angular/core';  
  
@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective {  
  constructor(private elementRef: ElementRef) { }  
}

Because attribute directives are applied to some elements, they always have the reference to the exact DOM node where they’re set.

Since we have a reference to the actual HTML element, we can use it to modify its style. The ElementRef directive has a field called nativeElement that is the actual DOM node. This field contains all of its properties, like style, and attributes along with others.

We can use the nativeElement field like this to modify the width and the height:

import { Directive, ElementRef } from '@angular/core';  
  
@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnInit {  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnInit() {  
        const size = '64px';  
        this.elementRef.nativeElement.style.width = size;  
        this.elementRef.nativeElement.style.height = size;  
    }  
}

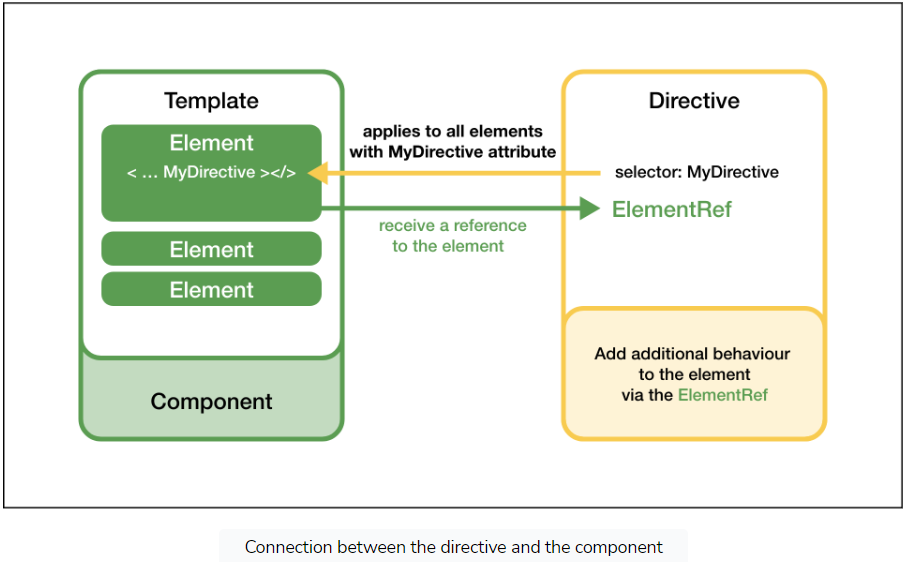
## Apply the directive

How can we use the directive that we created? Since we use a default attribute selector, wherever we put the attribute onto the element applies the directive, like this:

<img src="/assets/avatar.png" appAvatar>

Angular creates an instance of the directive for every usage in this straightforward way. In our case, it also takes the reference to the <img> element and injects that into the directive’s elementRef field from the constructor.

The diagram below explains the concept of connection between the directive and the component:



Here’s a live example with this directive implemented. We can take a closer look to find out an important rule. if we set styles directly using the nativeElement reference, they take precedence over styles set by the CSS. So, let’s keep this in mind.

## Summary

Attribute directives can be easily created using the the Angular CLI command ng g d <name>. Usually, it’s helpful to inject the instance of ElementRef that gives the reference to the host element of the directive. While implementing, we must remember that with our directive, different selectors help trigger specific elements.

**Exercise: Introduction to Attribute Directives**

Let’s practice implementing a custom attribute directive.

**We'll cover the following**

* [Objectives](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/6400872726921216#Objectives)
* [Task](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/6400872726921216#Task)
* [Result](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/6400872726921216#Result)

Let’s practice what we learned and improve our directive for avatars.

**Objectives**

We’ll create a similar directive to the one in the previous lesson, so that it matches the following criteria:

* The directive should set the size of the element to 72px x 72px.
* The directive should add a yellow, 1px-width border to the image.
* The directive should round the image so that it’s in the same shape as the circle.
* The directive should only work on <img> element and not any other.

**Hint:** Think about the selector’s possibilities

**Solution: Introduction to Attribute Directives**

Let’s compare the task result with the expected result and explanation.

**We'll cover the following**

* [Solution](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5137143242686464#Solution)
* [Explanation](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5137143242686464#Explanation)

**Solution**

Here’s an example of what the solution for this task may look like:

## Explanation

The actual directive code is very similar to what we learned in the lesson. We use ElementRef to get access to the element and set the styles on it directly, like this:

const size = '72px';  
this.elementRef.nativeElement.style.width = size;  
this.elementRef.nativeElement.style.height = size;  
this.elementRef.nativeElement.style.border = `1px solid gold`;  
this.elementRef.nativeElement.style.borderRadius = size;

The last thing is to trigger only the img elements. For that, we need to improve our selector so that it matches the rule:

@Directive({  
  selector: 'img[appAvatar]'  
})

# Pass Values to the Custom Attribute Directive

Let’s learn how to pass data into the directive.

**We'll cover the following**

* [Pass values](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5076796469149696#Pass-values)
* [Pass dynamic values](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5076796469149696#Pass-dynamic-values)
* [Single template attribute syntax](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5076796469149696#Single-template-attribute-syntax)
* [Summary](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5076796469149696#Summary)

We learned how to create a basic attribute directive that changes the element’s appearance in the previous lesson. However, we often require the element’s appearance to work in a slightly more dynamic way. For instance, we may want the element to potentially change its appearance, but we may not want the same static values to always apply. Maybe we want to supply different values based on the actual use case. In cases like these, passing values to the directive is especially useful.

## Pass values

In this lesson, we’ll continue working on the example from the previous lesson–**the avatar directive**. This time we want to control the size of the avatar. Previously, it worked as shown below, each time setting the width and height to 64px:

@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnInit {  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnInit() {  
        const size = '64px';  
        this.elementRef.nativeElement.style.width = size;  
        this.elementRef.nativeElement.style.height = size;  
    }  
}

This time, let’s make the size value dynamic. To do that, we want to pass the value through app.component.html.

How do we usually pass the data between the component and its child components? We do it through **inputs**. Remember that most things we know from the components’ world can be used for directives, too. So, let’s use them!

We can add the input like this:

@Input() size: string;

Now, we can use its value, as shown below:

@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnInit {  
  
    @Input() size: string;  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnInit() {  
        this.elementRef.nativeElement.style.width = this.size;  
        this.elementRef.nativeElement.style.height = this.size;  
    }  
}

Now, the directive class looks complete. We just have to provide the size value to this directive. In the previous lesson, the following was enough:

<img src="/assets/avatar.png" appAvatar>

Now, we also need to insert the attribute for the Input property. Whenever we apply a directive to an element, it will have all of the directive public properties available such as Input, Output, other public methods, or other fields. All of that is available to us, and we can use it in our component.

In our case, we’ll add the binding to the size input property:

<img src="/assets/avatar.png" appAvatar size="120px">

We provide a value of 120px and bind it to the size property of the avatar directive. The directive can use it to initialize the element’s size, like before.

The application behaves nicely, but there’s one thing that we should improve. When we implemented the directive before, we didn’t provide a size, as shown below:

<img src="/assets/avatar.png" appAvatar>

This means the application may not work as expected. We can do a couple of things to prevent this. However, we need to know how to provide compatibility with the previous version so that it works even without the provided value.

We can set an initial value, which behaves like a default on in case the value isn’t provided through the Input:

@Input() size: string = '64px';

This always makes our avatars 64px x 64px by default. However, if the size is specified, it takes the specified value.

**Pass dynamic values**

In the previous example, we set a fixed size, which is fine. But, we can make the avatar even more **dynamic**. Let’s imagine that we have buttons that increase and decrease a value and can be used to populate the avatar’s size.

We start with the component code below:

@Component({  
  selector: 'app',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
    value: number = 64;  
  
    increase() {  
        this.value = this.value + 4;  
    }  
  
    decrease() {  
        this.value = this.value - 4;  
    }  
}

We have a size property that’s a number starting from 64, which changes while calling the increase() and decrease() methods. Here’s the template for that:

<img src="/assets/avatar.png" appAvatar size="120px">  
  
<button (click)="increase()">Increase + 4</button>  
<button (click)="decrease()">Decrease - 4</button>

Since the size property of the directive is an Input property, we can use the data binding to not only bind it with a plain string like 120px, but also to bind it with any other field. In our case, we want to bind it with the value component’s field.

<img src="/assets/avatar.png" appAvatar [size]="value">  
  
<button (click)="increase()">Increase + 4</button>  
<button (click)="decrease()">Decrease - 4</button>

To do this, we use a proper [] data-binding syntax.

To perform this data binding, there are two things we need to adjust in our directive:

1. The avatar directive takes size as a string, and we provide a number. These two different types of data aren’t compatible.
2. The avatar directive sets the styles in the ngOnInit hook. This means it only sets up these styles once we initialize the directive, and does not care about incoming changes.

**How do we fix the incompatible types of the input?** In this case, we simply switch to numbers only and assume they’re always in px units:

@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnInit {  
  
    @Input() size: number = 64;  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnInit() {  
        this.elementRef.nativeElement.style.width = this.size + 'px';  
        this.elementRef.nativeElement.style.height = this.size + 'px';  
    }  
}

Done! Now, we need to take care of the changes.

Usually in Angular, we can solve problems like these by using an appropriate lifecycle hook. In our case, we shouldn’t use ngOnInit because it only works once. Rather, we should use ngOnChanges, which is called each time the Input property changes. In fact, it’s called each time size changes.

@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnChanges {  
  
    @Input() size: number = 64;  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnChanges() {  
        this.elementRef.nativeElement.style.width = this.size + 'px';  
        this.elementRef.nativeElement.style.height = this.size + 'px';  
    }  
}

## Single template attribute syntax

In a situation like ours, when the directive has only a single Input property, it makes sense to simplify the code, like this:

<img src="/assets/avatar.png" appAvatar [size]="value">

For this purpose, we can use a special naming convention for the properties.

If we name our Input property the same as the selector directive, we’ll get a single attribute that does these two jobs:

* It applies the specified directive once it’s called.
* It makes the directive’s appAvatar property available for binding.

So, let’s change the name this way:

@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnChanges {  
  
    @Input() appAvatar: number = 64;  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnChanges() {  
        this.elementRef.nativeElement.style.width = this.appAvatar + 'px';  
        this.elementRef.nativeElement.style.height = this.appAvatar + 'px';  
    }  
}

If we don’t want to change the property name and keep the original, we can always leave it as is and just specify the Input name, like this:

@Directive({  
  selector: '[appAvatar]'  
})  
export class AvatarDirective implements OnChanges {  
  
    @Input('appAvatar') size: number = 64;  
  
  constructor(private elementRef: ElementRef) { }  
  
    ngOnChanges() {  
        this.elementRef.nativeElement.style.width = this.size + 'px';  
        this.elementRef.nativeElement.style.height = this.size + 'px';  
    }  
}

Now, no matter which path we use, we can now use the directive as follows:

<img src="/assets/avatar.png" [appAvatar]="value">

This way, we can simultaneously apply the directive and bind its property in one attribute!

## Summary

* Directives can use @Input properties to get data the same way that the child components do.
* We can specify the Input property name to be the same as the directive selector name, which results in applying a single attribute to the template.
* It’s essential to use proper lifecycle hooks in our directives.

**Objectives**

We want to implement a directive that can be applied to change the apprearance of buttons while adhering to the following requirements:

* Create a directive that applies to the appButton selector.
* The directive should take one of these two color modes:
  + The primary mode should change the background color to blue and the font color to white.
  + The secondary mode should change the background color to yellow and the font color to black.

After the directive takes either primary or secondary, the color mode property should be able to bind using a single attribute with the appButton. `

* The directive should take another value which is the size provided in px. It should also follow these parameters:
  + By default, the button should have a 120px width.
  + If a value is provided, it should be set as a new width.

Sample:

import { Directive, ElementRef, Input, OnChanges } from '@angular/core';

@Directive({

  selector: '[appButton]'

})

export class ButtonDirective implements OnChanges {

  @Input('appButton') color!: 'primary' | 'secondary';

  @Input() size: number = 120;

  constructor(private elementRef: ElementRef) {

  }

  ngOnChanges() {

    this.elementRef.nativeElement.style.width = `${this.size}px`;

    if (this.color === 'primary') {

      this.elementRef.nativeElement.style.backgroundColor = 'deepskyblue';

      this.elementRef.nativeElement.style.color = 'white';

    }

    if (this.color === 'secondary') {

      this.elementRef.nativeElement.style.backgroundColor = 'gold';

      this.elementRef.nativeElement.style.color = 'black';

    }

  }

}

## Solution explained

We need to use Input properties in the directive to pass the color mode data as well as the optional size data.

The color mode consists of two values—primary and secondary. So, we need to include these in our code. Also, we have to make sure this property is implemented in a way that allows a single-bind syntax, as follows:

<button appButton="primary"> Click me </button>

To fulfill all these requirements for color, let’s implement it this way:

@Input('appButton') color!: 'primary' | 'secondary';

The property for size should be optional and have a default value of 120, so:

@Input() size: number = 120;

**And that’s most of the task!** The rest is just about using the values to apply specific styles, which we already know how to do.

# Handle Events with a Custom Attribute Directive

Let’s learn how to handle events in the directive.

**We'll cover the following**

* [Receive events](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/4826443605606400#Receive-events)
  + [The HostListener declaration](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/4826443605606400#The-HostListener-declaration)
* [Send custom events](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/4826443605606400#Send-custom-events)
* [Summary](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/4826443605606400#Summary)

We recently learned how to create a custom attribute directive and pass data to it. Now, we’ll learn how to handle events. When we say “handle events,” we mean these two actions:

* Receiving events, such as handling clicks on elements appropriately.
* Sending events to others, like parent components.

This ability is instrumental to cases in which our directive behaves differently based on user interaction, or if we want to trigger external logic based on a specific condition.

## Receive events

In Angular, receiving events subscribe to the DOM events, and can be implemented in different ways. However, the most popular way is to use event binding syntax on elements in the template, like this:

<button (click)="doSomething()"> Click me! </button>

Here, we bind a component’s function doSomething to an event whose target name is click. That’s fine if we have a template and elements inside it, like we have in components. However, we’re currently in the world of directives, so we have no template.

There’s another way, which is to register an **event listener** function. We can do this by using the HostListener decorator provided by Angular. We’ll focus on implementing it inside the directive, but let’s keep in mind that this approach works similarly for components too.

We start with something straightforward, such as a click event. Then, we create a directive that logs something in the console. That’s it–just a basic implementation!

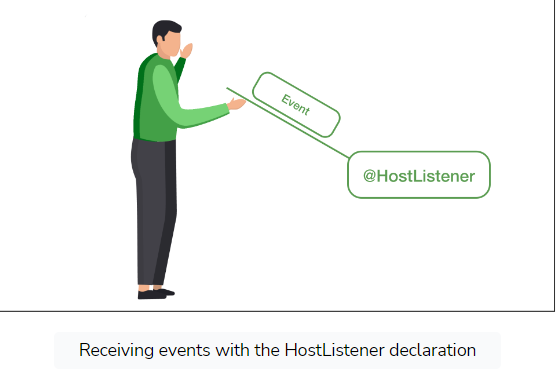
### The HostListener declaration

The HostListener declaration is one of the Angular decorators. It should be provided for a function, just like Input is provided for a variable. The decorator runs the function called the event handler if the event occurs. The event and optional arguments are specified in the HostListener declaration. Let’s look at an example:

@Directive({  
  selector: '[appMy]'  
})  
export class MyDirective {  
  
    @HostListener('click', ['$event'])  
    onClick(event: Event) {  
        *// ... do something on click*  
    }  
  
}

There are two essential blocks of code detailed below:

* The first one is the HostListener declaration, which has the event name and the optional array of arguments. In this case, we register for the click event and want to receive the whole event object.
* The second one is the actual onClick() function. This is the regular function with one argument—the event object. We can do whatever we want inside this function.



Let’s see how the directive logs something into the console through a click.

This is very similar to the example:

@Directive({  
  selector: '[appClickLog]'  
})  
export class ClickLogDirective {  
  
    @HostListener('click')  
    onClick(){  
        console.log('clicked');  
    }  
}

If we set this up on an element, this happens:

<button (click)="doSomething()" appClickLog> Click me! </button>

Each time the user clicks this button, two things occur:

* The doSomething function is called in the template using event binding.
* The clicked string is logged in the console using the directive’s event notifier.

**Send custom events**

Many times, handling logic internally is not enough. For example, sometimes, we want to trigger something in the parent components or at least inform them about something. To do that, we need to establish a way to communicate between the directive and the component that uses the directive.

So, how would we approach this problem in the world of components? How would the child and parent components communicate? Of course, there are a few options, but we’ll focus on the most direct one.

We’ll use the Output decorator along with EventEmitter. We do this as follows:

* Register EventEmitter as an Output in the child component:

@Component({  
  selector: 'app-hello',  
  template: `<button (click)="onClick()"> Hello! </button>`,  
})  
export class HelloComponent {  
  @Output() hello = EventEmitter<void>;  
  
    onClick() {  
        this.hello.emit();  
    }  
}

* Receiving and handling event in the parent component:

<app-hello (hello)="onHello()"></app-hello>

* Assuming that the parent component has the onHello function, the code will look like this:

@Component({  
  selector: 'app',  
  template: `<app-hello (hello)="onHello()"></app-hello>`,  
})  
export class AppComponent {  
    onHello() {  
        this.console.log('clicked: on hello');  
    }  
}

Okay, let’s get back to directives and see how this helps us send custom events. We can think of the child component as just a directive, and parent component as an ordinary component that includes some elements with that directive applied to them.

The directive has the EventEmitter marked as Output and sends the events that a parent can handle. For instance, let’s imagine we want to build a directive that changes its appearance when clicked, but also sends that information to a component.

We create a directive that zooms on the element when clicked:

@Directive({  
  selector: '[appZoom]'  
})  
export class ZoomDirective {  
      
    @Output() zoom = new EventEmitter<boolean>();  
  
    constructor(private elementRef: ElementRef) { }  
  
    @HostListener('click')  
    onClick(){  
        this.elementRef.nativeElement.style.transform = 'scale(1.1)';  
        this.zoom.emit(true);  
    }  
}

The directive can be used as follows:

<img src="/assets/avatar.png" appZoom (zoom)="onZoom($event)">

Let’s briefly discuss how this works.

* First, we have a directive that performs an action when the click event occurs. For that, we use a well-known technique known as the HostListener.
* We also have an Output variable that sends events externally through EventEmitter. This time, we define the events as boolean values.

The reason we do this is that this directive only reacts to mouse click events but doesn’t do anything to reset the zoom applied to the element. It would make sense to implement the second method, reset the zoom, and pass the event with EventEmitter. In order to avoid creating another Output for that, we can simply pass information in a single event by doing the following:

* Send true when the mouse is clicked for the first time.
* Send false to reset it when the mouse is clicked for the second time.

**Summary**

Attribute directives work very well with DOM events and can also send any custom events to others. These directives give us the ability to trigger external logic by giving us information from the directive.

* We can use the HostListener technique to register for DOM events in the directive.
* We can use @Output variables to emit events from the directive.

# Exercise: Handle Events with a Custom Attribute Directive

Let’s practice handing events in directives.

**We'll cover the following**

* [Objectives](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5847979925962752#Objectives)
* [Task](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5847979925962752#Task)
* [Result](https://www.educative.io/module/page/El5jyzfkAngPpgpAB/10370001/4816453486510080/5847979925962752#Result)

## Objectives

Our task is to extend our avatar directive with the following features:

* The image should be slightly scaled up when the mouse cursor, or pointer, hovers over it.
* The image scale should be reset when the pointer is removed from the image.
* An event should be emitted each time the pointer is placed on or removed from an image.
* A component that uses the directive should receive information when zooming in or out on an image. It should also log image zoom in and image zoom out into the console accordingly.

## Explanation

Key points to consider while implementing this exercise are:

* How to react and change the style on mouse events.
* How to emit events from the directive.

Reacting on mouse events are solved using HostListener, like so:

@HostListener('mouseenter')  
onMouseEnter() {  
    this.elementRef.nativeElement.style.transform = 'scale(1.1)';  
}

Now, we need a way of informing listeners about our actions. For this, we need to use the Output property, like this:

@Output() zoom = new EventEmitter<boolean>();

We can emit values when changes occur, as shown below:

@HostListener('mouseenter')  
onMouseEnter() {  
    this.zoom.emit(true);  
}  
  
@HostListener('mouseleave')  
onMouseLeave() {  
    this.zoom.emit(false);  
}

**That’s all there is to it! !**