DIRECTIVES

Custom Attribute Directive

In Angular, **directives** are special markers or attributes on elements that tell Angular to do something with that element or its children. There are three types:

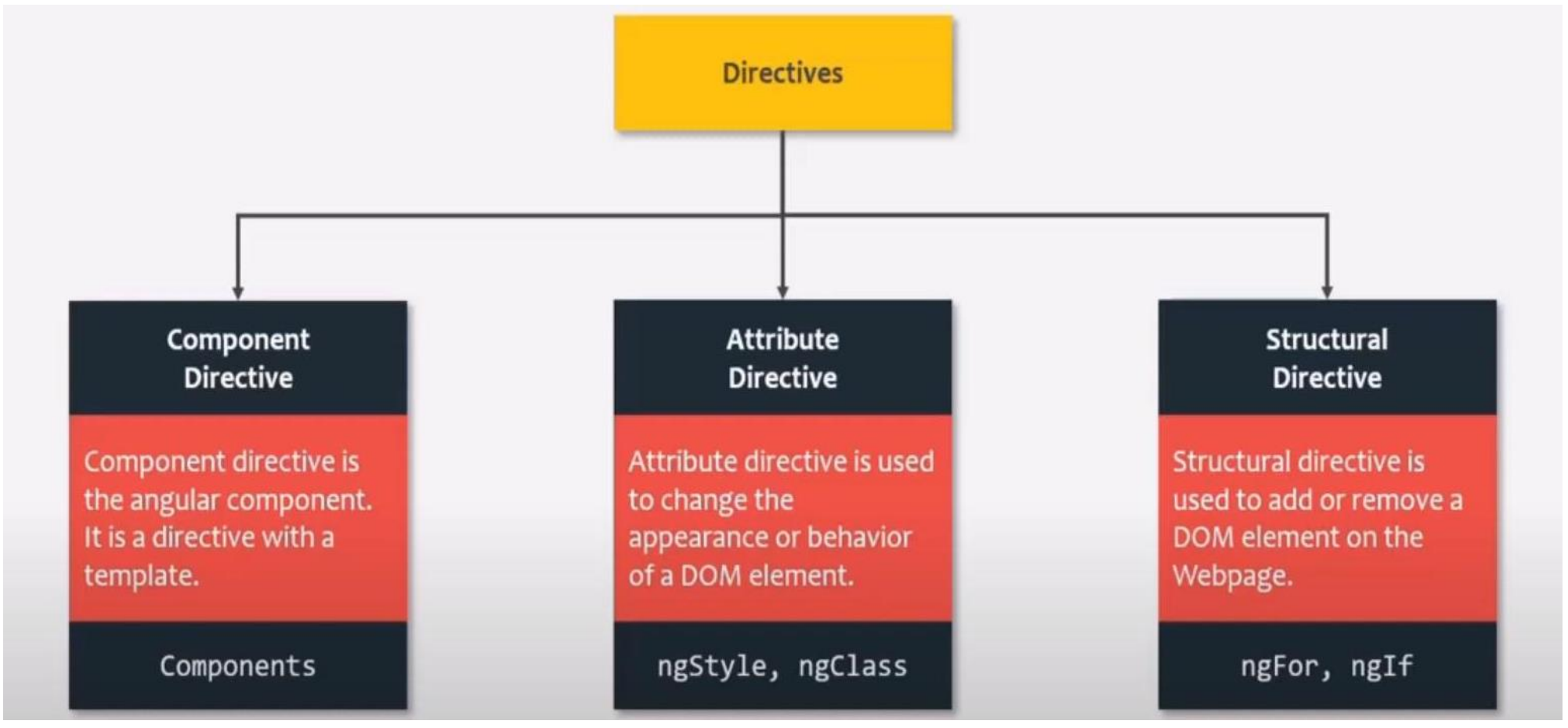
1. **Component Directives**: These are directives with a template (like your Angular components).
2. **Structural Directives**: They change the structure of the DOM, like `*ngIf` and `*ngFor`.
3. **Attribute Directives**: They change the appearance or behavior of an element, like `ngClass` or `ngStyle`.

In simple terms, directives add behavior or modify the DOM elements in an Angular application.

DIRECTIVES DEFINITION:

Yes, you can definitely say **directives are instructions to the DOM**. They tell Angular how to modify or manipulate the DOM elements, either by changing their structure (like adding or removing elements) or their behavior and appearance.

CUSTOM DIRECTIVE:



Notes on Angular Directives and Custom Attribute Directives

Types of Directives in Angular

1. **Component Directive**:
    - Angular components are considered directives with a template.
    - Example: `<app-root></app-root>` is a component directive in the root template.

2. **Attribute Directive**:
    - Changes the appearance or behavior of a DOM element.
    - Example: `ngClass` and `ngStyle` are attribute directives to modify styles dynamically.

```html
<div [ngStyle]="{'color': 'red'}">Hello</div>
```

3. **Structural Directive**:
    - Used to add or remove DOM elements.
    - Example: `ngIf` and `ngFor` modify the DOM based on conditions or iterations.

```html
<div *ngIf="isVisible">Visible</div>
<div *ngFor="let item of items">{{item}}</div>
```

Custom Attribute Directive

- **Goal**: Dynamically change the background and text color of an HTML element.
- **Steps to Create a Custom Attribute Directive**:

1. **Create the Directive**:
    - Create a TypeScript file for the directive (e.g., `set-background.directive.ts`).
    - Use `@Directive()` decorator to define a custom directive.

```typescript
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[setBackground]'
})
export class SetBackgroundDirective {
  constructor(private element: ElementRef) {
    this.element.nativeElement.style.backgroundColor = 'red';
  }
}
```

2. **Using the Custom Directive**:

   - Apply the directive to any HTML element by using its selector as an attribute.

```html
<div setBackground></div>
```

3. **Dependency Injection with ElementRef**:

   - Angular injects a reference to the element where the directive is applied using `ElementRef`.
   - Example:

```typescript
constructor(private el: ElementRef) {
  this.el.nativeElement.style.backgroundColor = 'yellow';
}
```

5. **Registering the Directive**:

   - The custom directive needs to be registered in the module's declarations array.

```typescript
import { SetBackgroundDirective } from './custom-directives/set-background.directi

@NgModule({
  declarations: [SetBackgroundDirective],
  // other configurations
})
```

4. **Setting Dynamic Styles:**

   - You can set dynamic styles using `nativeElement` property:

```typescript
this.el.nativeElement.style.color = 'white';
```

6. **Final Example in HTML:**

```html
<div setBackground>Content with custom background</div>
```

This covers the basics of Angular directives and how to create a custom attribute directive to modify the appearance of DOM elements dynamically.

Background.directive.ts U

src > app > DIRECTIVES_Lesson > direct > SetBackground.directive.ts > SetBackground

```typescript
import { Directive, ElementRef, OnInit } from "@angular/
core";

@Directive({
  selector: '[setBackground]'
})

export class SetBackground implements OnInit {
  constructor(private elem: ElementRef) {
    // this.elem = elem;
  }

  ngOnInit() {
    this.elem.nativeElement.style.backgroundColor = 'red';
    this.elem.nativeElement.style.color = 'white';
  }
}
```

direct.component.html U

angular-ekart > src > app > DIRECTIVES_Lesson > direct > direct.component.html > div.

Go to component

```html
<div class="container" *ngFor="let prod of product">

  <span setBackground>{{prod.gender}}</span><br>
  <span setBackground>{{prod.brand}}</span><br>
  <span setBackground>{{prod.category}}</span><br>

</div>
```

Don’t Forgot to Import it in the App Module Folder.

product-detail.component.html | product-detail.component.css | SetBackground.directive.ts

src > app > CustomDirectives > SetBackground.directive.ts > SetBackground > constructor

```typescript
import { Directive, ElementRef } from "@angular/core";

@Directive({
    selector: '[setBackground]'
})
export class SetBackground{
    constructor(element: ElementRef){
        element.nativeElement.style.backgroundColor = '#36454F';
        element.nativeElement.style.color = 'white';
    }
}
```

```html
<div class="ekart-product-detail-gbc">
    <span setBackground>{{ product.gender }}</span>
    <span setBackground>{{ product.brand }}</span>
    <span setBackground>{{ product.category }}</span>
</div>
```

product-detail.component.html ● | # product-detail.component.css | TS SetBackground.directive.ts ●

src > app > CustomDirectives > TS SetBackground.directive.ts > SetBackground > element

```ts
import { Directive, ElementRef, OnInit } from "@angular/core";

@Directive({
    selector: '[setBackground]'
})
export class SetBackground implements OnInit{
    private element: ElementRef;
    constructor(element: ElementRef){
        this.element = element;
    }

    ngOnInit(){
        this.element.nativeElement.style.backgroundColor = '#36454F';
        this.element.nativeElement.style.color = 'white';
    }
```

product-detail.component.html | # product-detail.component.css | TS SetBackground.directive.ts ●

src > app > CustomDirectives > TS SetBackground.directive.ts > SetBackground > constructor

```ts
import { Directive, ElementRef, OnInit } from "@angular/core";

@Directive({
    selector: '[setBackground]'
})
export class SetBackground implements OnInit{
    //private element: ElementRef;

    constructor(private element: ElementRef){
        this.element = element;
    }

    ngOnInit(){
        this.element.nativeElement.style.backgroundColor = '#36454F';
        this.element.nativeElement.style.color = 'white';
    }
```

```ts
ngOnInit(){
    this.element.nativeElement.style.backgroundColor = '#36454F';
    this.element.nativeElement.style.color = 'white';
}
```

2. Accessing DOM Elements Using `ElementRef`

- **ElementRef**: Provides a reference to the underlying DOM element.
- **Native Element**: The actual DOM object is accessed via `elementRef.nativeElement`.

Renderer2 In Angular

Notes on Angular Custom Attribute Directive and Renderer2

1. Custom Attribute Directive (Set Background Example)

- **Definition**: A directive that changes the behavior or appearance of the element it is applied to.
- **Example Directive**:
    - We create a directive called `setBackground` with the selector `[setBackground]`.
    - This selector is applied as an attribute on elements in the template.
    - The directive is used to change the element's **background color** and **text color**.

```
@Directive({
  selector: '[setBackground]'
})
export class SetBackgroundDirective {
  constructor(private element: ElementRef) {
    this.element.nativeElement.style.backgroundColor = 'gray';
    this.element.nativeElement.style.color = 'white';
  }
}
```

- **Explanation**: We access the DOM element using `ElementRef` and modify its style properties directly.

Why Direct DOM Manipulation is Not Recommended

- **Performance**: Angular's change detection and data binding are bypassed.
- **Cross-Platform Issues**: Direct DOM manipulation only works in browsers, not in environments like web workers or server-side rendering.
- **Security Risk**: DOM APIs do not sanitize data, potentially exposing the app to XSS attacks.

3. Using `Renderer2` to Manipulate the DOM

- **Definition**: `Renderer2` is a class that provides an abstraction layer for DOM manipulation, allowing safe and cross-platform DOM operations.
- **Why Use Renderer2**:
  - Prevents direct DOM manipulation.
  - Works across different platforms (browser, server-side, etc.).
  - Ensures security and prevents XSS attacks.

Example Using Renderer2 for Styles

Copy code

```
import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[setBackground]'
})
export class SetBackgroundDirective {
  constructor(private element: ElementRef, private renderer: Renderer2) {
    this.renderer.setStyle(this.element.nativeElement, 'background-color', 'gray');
    this.renderer.setStyle(this.element.nativeElement, 'color', 'white');
  }
}
```

- **Explanation**: Instead of accessing the DOM directly, `Renderer2` is used to set the background and text color.
- **Methods in `Renderer2`**:
  - `setStyle(element, styleName, value)`: Sets a CSS style on the element.
  - **Example**: Changes background color to gray and text color to white.

Output:

- Before using `Renderer2`:
  - Background: Gray, Text: White
- After modifying the text color to yellow:
  - Background: Gray, Text: Yellow

4. **Other Methods in** `Renderer2`

- **setAttribute()**: Dynamically sets attributes on an HTML element.
  - Example:

typescript Copy code

```typescript
this.renderer.setAttribute(this.element.nativeElement, 'title', 'Example Title');
```

  - **Output**: Hovering over the element will display the title "Example Title."

- **addClass()**: Adds a CSS class to an HTML element.

typescript

```typescript
this.renderer.addClass(this.element.nativeElement, 'active');
```

- **removeClass()**: Removes a CSS class from an HTML element.

typescript

```typescript
this.renderer.removeClass(this.element.nativeElement, 'active');
```

Key Points:

- **Avoid direct DOM access** through `ElementRef.nativeElement`. Use `Renderer2` instead for safer, platform-independent DOM manipulation.
- `Renderer2` provides useful methods like `setStyle()`, `setAttribute()`, `addClass()`, and `removeClass()` to interact with the DOM safely.

1. Angular keeps the component & view in sync using templates, data binding and change detection etc. All of them are bypassed when we update the DOM directly.

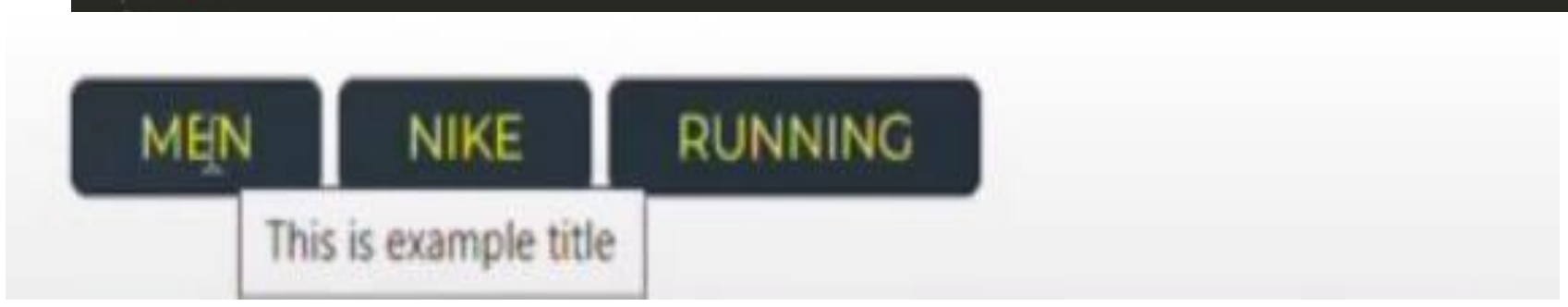
2. The DOM manipulation works only in browsers. You will not be able to use your app in other platforms like web workers, servers for server-side rendering, desktop or mobile apps etc. where there is no browser.

3. The DOM APS's does not sanitize the data. Hence it is possible to inject a script, thereby, opening our app an easy target for the XSS injection attacks.

Renderer2 allows us to manipulate the DOM without accessing the DOM elements directly, by providing a layer of abstraction between the DOM element and the component code.

```
import { Directive, ElementRef, OnInit, Renderer2 } from "@angular/core";

@Directive({
    selector: '[setBackground]'
})
export class SetBackground implements OnInit{
    //private element: ElementRef;
    //private renderer

    constructor(privat
        //this.element
        //this.rendere
    }

    ngOnInit(){
        // this.elemen
        // this.elemen
        this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', '#36454F')
    }
```

```
(method) Renderer2.setStyle(el: any, style: string, value: any, flags?:
RendererStyleFlags2): void

Implement this callback to set a CSS style for an element in the DOM.

@param el — The element.

@param style — The name of the style.

@param value — The new value.

@param flags — Flags for style variations. No flags are set by default.
```



renderer2.directive.ts U

Complete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > renderer2.directive.ts > ...

```ts
import { Directive, ElementRef, OnInit, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appRenderer2]'
})
export class Renderer2Directive implements OnInit {

  //Now these properties with access modifier creates an instance
  // for these properties and injects that instance to these properties
  constructor(private elemRef: ElementRef, private render: Renderer2) { }

  ngOnInit() {

    this.render.setStyle(this.elemRef.nativeElement, 'backgroundColor', 'yellow');

    this.render.setStyle(this.elemRef.nativeElement, 'color', 'red');

    this.render.setAttribute(this.elemRef.nativeElement, 'title', 'This is some example')
```

direct.component.html U

Course > angular-ekart > src > app > DIRECTIVES_Lesson > direct > direct.component.html

Go to component

```html
<div class="container" *ngFor="let prod of product">

  <span appRenderer2>{{prod.gender}}</span><br>
  <span appRenderer2>{{prod.brand}}</span><br>
  <span appRenderer2>{{prod.category}}</span><br>

</div>
```

@HostListner In Angular

The `@HostListner` decorator listens to a DOM event on the host element and it reacts to that event by executing an event handler method.

Angular `@HostListener` Decorator

The `@HostListener` decorator in Angular allows you to listen to events on the host element (the element the directive is applied to) and execute an event handler when that event occurs. This is particularly useful for handling user interactions like mouse movements, key presses, and other DOM events directly from within your directive.

2. **Directive Code**: We will use `@HostListener` to listen to `mouseenter` and `mouseout` events on the host element and zoom in/out the element when these events occur.

typescript Copy code

typescript Copy code

```typescript
@HostListener(eventName: string, [args])
```

- `eventName` : The name of the event you want to listen for (e.g., `'click'` , `'mouseenter'`).
- `args` : (Optional) The arguments you want to pass to the event handler method.

Example

1. **Create a Custom Directive** using Angular CLI:

bash Copy c

```bash
ng g d custom-directives/highlight
```

This creates a `highlight.directive.ts` file where the directive logic will be implemented.

```typescript
import { Directive, ElementRef, Renderer2, HostListener } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {}
```

Output:

When you hover over the div with the `appHighlight` directive applied, the product will zoom in, and when the mouse leaves the div, it will zoom out.

```
  // Listen to mouseenter event
  @HostListener('mouseenter') onMouseEnter() {
    this.renderer.addClass(this.el.nativeElement, 'highlight-product');
  }

  // Listen to mouseout event
  @HostListener('mouseout') onMouseOut() {
    this.renderer.removeClass(this.el.nativeElement, 'highlight-product');
  }
}
```

3. **CSS to Add the Zoom Effect**: Define the `.highlight-product` class to scale the product when the mouse hovers over it.

```css
.highlight-product {
  transform: scale(1.1);
  transition: transform 0.3s ease-in-out;
}
```

4. **Usage in HTML**: Apply the `appHighlight` directive to any HTML element, for example:

```html
<div appHighlight>
  <!-- Product details here -->
</div>
```

Example Output:

Before Hover:

```css
[ Product ]
```

After Hover (with zoom effect):

CSS

```
[ Zoomed Product ]
```

highlight.directive.ts U

mplete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > highlight.directive.ts > ...

```ts
import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';

@Directive({
  selector: '[Highlight]'
})
export class HighlightDirective {

  constructor(private element: ElementRef, private render: Renderer2) { }

  @HostListener('mouseenter') onMouseEnter() {
    this.render.addClass(this.element.nativeElement, 'highlight-product')
  }

  @HostListener('mouseout') onMouseOut() {
    this.render.removeClass(this.element.nativeElement, 'highlight-product')
  }

}
```

highlight.directive.ts U | host.component.html U | host.component.css U

Complete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > host > host.component.css > .highligl

```css
.container {
  width: 300px;
  height: 400px;
  margin-top: 30px;
  margin: 20px;
  padding: 20px;
  background-Color: rgb(255, 0, 200);
}

.highlight-product {
  -webkit-box-shadow: 0 0 1px 3px #000 inset;
  -moz-box-shadow: 0 0 1px 3px #000 inset;
  border: #efefef 2px solid;
  box-shadow: 10px 10px 18px #cccccc;
  transform: scale(1.05);
}
```

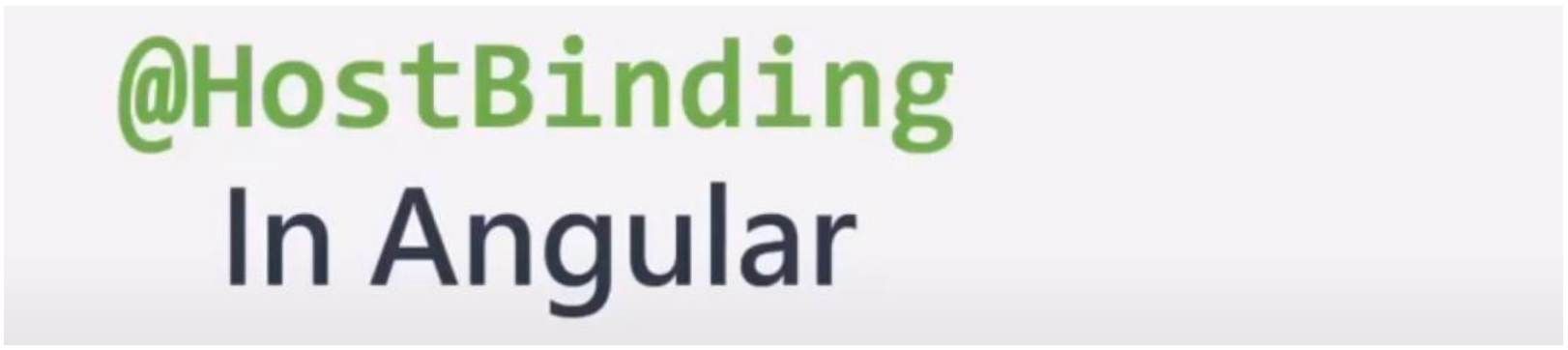
highlight.directive.ts U | host.component.html U

Complete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > host >

Go to component

```
<div class="container">

  <div class="card" Highlight>
    <div class="card-body">

      <div class="card-title">
        <h1>First Card</h1>
      </div>

      <p>Price : 800000</p>
    </div>
  </div>

</div>
```

When mouse enters into card it will be Zoomed In &when mouse leaves from card it will be Zoom Out.



The `@HostBinding` decorator binds a host elements property to a property of a directive or a component class.

Angular HostBinding and HostListener Decorators

1. HostListener Decorator

- **Definition**: The `HostListener` decorator listens to events on the host element (the element to which the directive is applied). When the event occurs, it executes a specified method in the directive class.
- **Example**:

```
@Directive({
  selector: '[appHover]'
})
export class HoverDirective {
  @HostListener('mouseenter') onMouseEnter() {
    console.log('Mouse entered');
  }

  @HostListener('mouseleave') onMouseLeave() {
    console.log('Mouse left');
  }
}
```

Usage in Template:

```html
<button appHover>Hover over me</button>
```

Output:

- When the mouse enters the button, "Mouse entered" is logged.
- When the mouse leaves the button, "Mouse left" is logged.

2. HostBinding Decorator

- **Definition**: The `HostBinding` decorator binds a property of the host element (such as style or class) to a property of the directive or component class.
- **Example**:

```ts
@Directive({
  selector: '[appHover]'
})
export class HoverDirective {
  @HostBinding('style.backgroundColor') backgroundColor: string = 'black';

  @HostListener('mouseenter') onMouseEnter() {
    this.backgroundColor = 'white';
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.backgroundColor = 'black';
  }
}
```

Usage in Template:

```html
<button appHover>Hover over me</button>
```

Output:

- Initially, the button has a black background.
- On mouse enter, the background changes to white.
- On mouse leave, the background reverts to black.

3. Difference Between HostListener and HostBinding

- **HostListener**: Listens to events (like `click`, `mouseenter`) on the host element and executes logic accordingly.
- **HostBinding**: Binds properties (like `style`, `class`) of the host element to the directive's properties.

```ts
export class HostBindingDirective {

  constructor(private elem: ElementRef) { }

  @HostBinding('style.backgroundColor') backgroundColor = 'black';

  @HostBinding('style.color') textColor = 'white';

  @HostBinding('style.border') border = 'none';

  @HostListener('mouseenter') mouseEnter() {
    this.backgroundColor = 'white';
    this.textColor = 'black';
    this.border = 'orange 3px solid';
  }

  @HostListener('mouseout') mouseOut() {
    this.backgroundColor = 'black';
    this.textColor = 'white';
    this.border = 'none';
  }
```

```html
<div class="Host-Binding">

  <button appHostBinding> Add To Cart</button>

</div>
```

Add To Cart

Output:

- When the mouse enters the button, "Mouse entered" is logged.
- When the mouse leaves the button, "Mouse left" is logged.

Add To Cart

Property Binding vs @HostBinding

Property Binding in Components

- **Definition**: Property binding allows you to bind a property of a DOM element to a property in the component. This is useful when you want to dynamically change the DOM element based on the component's data.
- **Example**:
  In a component, you can bind a `textValue` property to the `value` of an input element using property binding syntax (square brackets).

```html
<input [value]="textValue">
```

Here, `textValue` is a property in the component's TypeScript file:

```typescript
export class DemoComponent {
  textValue: string = "Hello World";
}
```

This will render "Hello World" in the input field.

Host Binding in Directives

- **Definition**: Host binding is used to bind properties in directives to the host element's property. This is helpful when the directive needs to modify or control the behavior of the host element.
- **Example**:
  In a directive, you can bind the `inputValue` property to the `value` property of the host element using `@HostBinding`.

```typescript
@Directive({
  selector: '[appSample]'
})
export class SampleDirective {
  @HostBinding('value') inputValue: string = 'Hi there';
}
```

This will render "Hi there" in any input element where the directive is applied.

Event Binding in Components

- **Definition**: Event binding allows you to listen to DOM events and execute a method in the component when the event occurs.
- **Example**:
  To handle the `focus` event in a component, use parentheses to bind the event to a method.

```html
<input (focus)="logValue()">
```

In the component class, define the method:

```typescript
export class DemoComponent {
  logValue() {
    console.log('Input has been focused');
  }
}
```

Every time the input element is focused, the message "Input has been focused" will be logged in the console.

Host Listener in Directives

- **Definition**: Host listener is used in directives to listen for events on the host element and run a method in the directive. This is useful for event handling in directives.
- **Example**:
  To handle the `focus` event in a directive, use `@HostListener` to listen for the event on the host element.

When the input element is focused, the message "Input has been focused from sample directive" will be logged.

```
@Directive({
  selector: '[appSample]'
})
export class SampleDirective {
  @HostListener('focus') logMessage() {
    console.log('Input has been focused from sample directive');
  }
}
```

Key Differences:

1. **Property Binding vs Host Binding**:
   - Property binding is used in **components** to bind a component property to a DOM element property using square brackets (`[]`).
   - Host binding is used in **directives** to bind a directive property to the host element's property using `@HostBinding`.

2. **Event Binding vs Host Listener**:
   - Event binding in **components** listens for events on DOM elements using parentheses (`()`), and calls methods in the component class.
   - Host listener in **directives** listens for events on the host element using `@HostListener` and runs methods in the directive class.


```
TS demo.component.ts   <> demo.component.html ●   TS sample.directive.ts ×
src > app > CustomDirectives > TS sample.directive.ts > SampleDirective > inputValue
1   import { Directive, HostBinding } from '@angular/core';
2
3   @Directive({
4     selector: '[appSample]'
5   })
6   export class SampleDirective {
7
8     @HostBinding('value') inputValue: string = "Hi there!";
9
10    constructor() { }
11
12  }
```

```
TS demo.component.ts   <> demo.component.html ●   TS sample.directive.ts
src > app > demo > <> demo.component.html > input
      Go to component
1   <input type="text" appSample>
2   <button>Click Me</button>
```

Event Binding in Components VS Host Listener Directive.

```
TS demo.component.ts   <> demo.component.html 1 ●   TS sample.directive.ts
src > app > demo > <> demo.component.html > input
      Go to component
1   <input type="text" [value]="textValue" (focus)="logValue()">
2   <button>Click Me</button>
3
```

demo.component.ts ● | demo.component.html ● | sample.directive.ts

src > app > demo > demo.component.ts > DemoComponent > logValue

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html',
  styleUrls: ['./demo.component.css']
})
export class DemoComponent {
  textValue: string = 'Hellow, World!';

  logValue(){
    console.log('Input has been focused!');
  }
}
```

```html
<div class="data">

  <input type="text" [value]="inputData">

  <button>Click</button>

</div>
```

```ts
@Component({
  selector: 'app-host',
  templateUrl: './host.component.html',
  styleUrls: ['./host.component.css']
})
export class HostComponent {
  inputData = 'Hi i am this components input';
}
```

eb-vs-hlistener.directive.ts U

e_Course > angular-ekart > src > app > DIRECTIVES_Lesson > eb-vs-hlistener.directive.ts > EBVSHLIstenerDirective

```ts
import { Directive, HostBinding } from '@angular/core';

@Directive({
  selector: '[appEBVSHLIstener]'
})
export class EBVSHLIstenerDirective {

  @HostBinding('value') userData = 'Hi there';

  constructor() { }

}
```

```html
<div class="data">

  <input type="text" appEBVSHLIstener [value]="inputData">

  <button>Click</button>

</div>
```

component.ts U

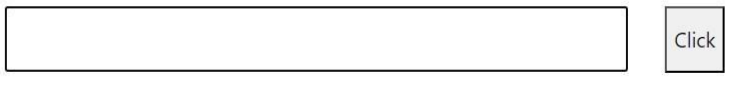
angular-ekart > src > app > DIRECTIVES_Lesson > host > host.component.ts > ...

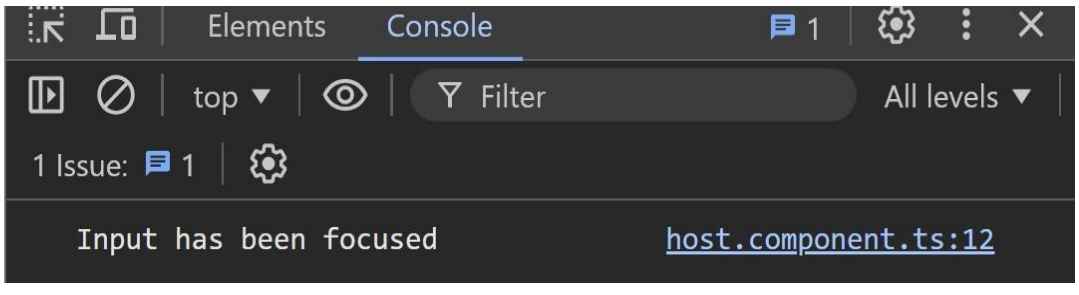
```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-host',
  templateUrl: './host.component.html',
  styleUrls: ['./host.component.css']
})
export class HostComponent {
  inputData = 'Hi i am this components input';

  logValue() {
    console.log("Input has been focused");
  }
}
```

host.component.html U

Complete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > host > host.compo

```html
<!-- Event Binding in Component VS Host Listener
Directive -->

<div class="data">

  <input type="text" (focus)="logValue()">

  <button>Click</button>

</div>
```





vs-hlistener.directive.ts U

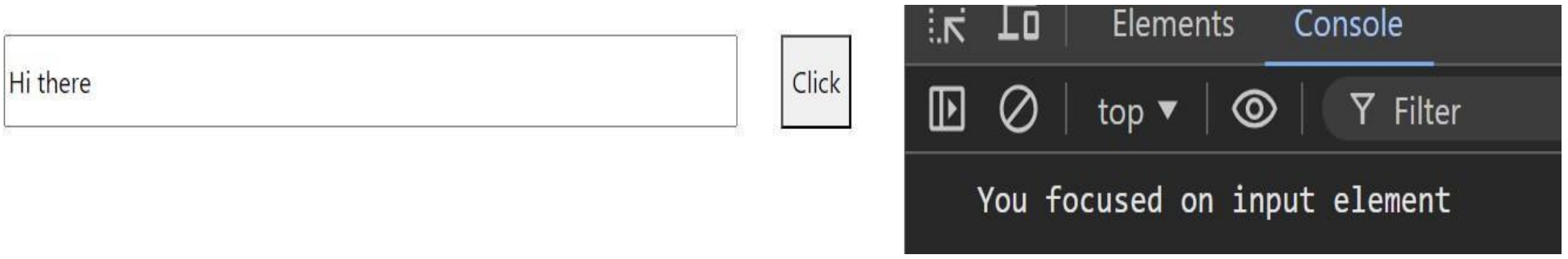
ete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > eb-vs-hlistener.directive.ts > ...

```ts
import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[appEBVSHLIstener]'
})
export class EBVSHLIstenerDirective {

  @HostBinding('value') userData = 'Hi there';


  constructor() { }

  @HostListener('focus') logFocusMessage() {
    console.log("You focused on input element")
  }

}
```

host.component.html U

omplete_Course > angular-ekart > src > app > DIRECTIVES_Lesson > host

```html
<!-- Event Binding in Component VS Host
Listener Directive -->

<div class="data">

  <input type="text" appEBVSHLIstener>

  <button>Click</button>

</div>
```



Property Binding in Directives

Notes on Custom Property Binding in Angular Directives

1. Custom Property Binding Overview

- **Definition**: Custom property binding allows dynamic assignment of values to properties of a directive.
- **Example**: Instead of hardcoding background color or text color, we can bind values provided by the user to directive properties.

2. Host Element

- **Definition**: The element where the directive is applied is called the host element.
- **Example**: In `<span appSetBackground>`, the `span` is the host element for the `appSetBackground` directive.

3. Creating Custom Properties

- **Step 1**: Define properties in the directive.

typescript Copy code

```typescript
@Input() backColor: string = 'gray';
@Input() textColor: string = 'white';
```

- **Step 2**: Use these properties inside the directive logic instead of hardcoded values.

4. Input Decorator

- **Definition**: The `@Input()` decorator is used to bind values from the component to the directive properties.
- **Example**:

```typescript
@Input() backColor: string;
@Input() textColor: string;
```

7. Default Property Binding

- **Scenario**: If no value is provided for a bound property, the default value specified in the directive will be used.
- **Example**: If no `textColor` is provided, the default value (`white`) will be applied.

5. Binding Directive Properties in the Component

- **Step 1**: Apply the directive on a component or HTML element.
- **Step 2**: Bind values using square brackets in the component's template.

```html
<span [appSetBackground]="backColor" [textColor]="white"></span>
```

6. Aliasing Properties

- **Definition**: You can alias a directive's property with the directive's selector name to make it more intuitive to bind the property directly.
- **Example**:

```typescript
@Input('appSetBackground') backColor: string;
```

- **Usage**:

```html
<span [appSetBackground]="'red'"></span>
```

8. Handling Multiple Properties

- If a directive has multiple properties, ensure each property is bound separately in the component's template.

```html
<span [appSetBackground]="'red'" [textColor]="'yellow'"></span>
```

9. Conflict with DOM Properties

- **Scenario**: If a directive and a DOM element share a property (e.g., `title`), Angular first checks if the property exists in the directive. If not, it binds it to the DOM element.
- **Example**:

```typescript
@Input() title: string = 'Directive Title';
```

```html
<span [title]="'My Title'"></span>
```

10. Key Points

- You must always use the directive for property binding if the properties are defined in the directive.
- Angular resolves property binding conflicts by prioritizing directive properties over DOM properties.

```
//private renderer: Renderer2;
backColor: String = '#36454F';
textColor: string = 'White';

constructor(private element: ElementRef, private renderer: Renderer2){
    //this.element = element;
    //this.renderer = renderer;
}

ngOnInit(){
    // this.element.nativeElement.style.backgroundColor = '#36454F';
    // this.element.nativeElement.style.color = 'white';
    this.renderer.setStyle(this.element.nativeElement, 'backgroundColor', this.backColor);
    this.renderer.setStyle(this.element.nativeElement, 'color', this.textColor);
    //this.renderer.setAttribute(this.element.nativeElement, 'title', 'This is example title');
}
```

```
@Directive({
    selector: '[setBackground]'
})
export class SetBackground implements OnInit{
    //private element: ElementRef;
    //private renderer: Renderer2;
    @Input() backColor: String = '#36454F';
    @Input() textColor: string = 'White';
```

Creating a Conditional Attribute Directive

```
<div class="ekart-product-detail-gbc">
    <span setBackground [backColor]="'red'" [textColor]="'white'">{{ product.gender }}</span>
    <span setBackground [backColor]="'yellow'" [textColor]="'black'">{{ product.brand }}</span>
    <span setBackground [backColor]="'black'" [textColor]="'white'">{{ product.category }}</span>
</div>
```

```
//private renderer: Renderer2;
@Input('setBackground') backColor: String = '#36454F';
@Input() textColor: string = 'White';
```

```
<div class="ekart-product-detail-gbc">
    <span [setBackground]="'red'" [textColor]="'white'" [ngClass]="">{{ product.gender }}</span>
    <span [setBackground]="'yellow'" [textColor]="'black'">{{ product.brand }}</span>
    <span [setBackground]="'black'" [textColor]="'white'">{{ product.category }}</span>
</div>
```

Using Both properties using a single reference

```
@Directive({
    selector: '[setBackground]'
})
export class SetBackground implements OnInit{
    //private element: ElementRef;
    //private renderer: Renderer2;
    // @Input('setBackground') backColor: String = '#36454F';
    // @Input() textColor: string = 'White';

    @Input('setBackground') changeTextAndBackColor: {backColor: string, textColor: string};
```

```
<div class="ekart-product-detail-gbc">
    <span [setBackground]="{backColor: 'red', textColor: 'white'}">{{ product.gender }}</span>
    <span [setBackground]="{backColor: 'black', textColor: 'white'}">{{ product.brand }}</span>
    <span [setBackground]="{backColor: 'blue', textColor: 'white'}">{{ product.category }}</span>
</div>
```

Attribute Directives in Angular

Definition:

An **attribute directive** is a type of directive in Angular that can modify the behavior or appearance of the DOM elements to which they are applied. Unlike structural directives, which modify the DOM structure, attribute directives are used to change styling, classes, or other attributes on the host elements.

Example:

Below is a simple example of how to create an attribute directive that disables a product and changes its appearance if it is out of stock.

1. **Creating the Directive:** Use Angular CLI to generate a directive called `disableProduct`.

bash — Copy code

```
ng generate directive disableProduct
```

2. **Directive Class Implementation:** In the directive's TypeScript file (`disable-product.directive.ts`), you can inject the `ElementRef` to access the element and `Renderer2` to manipulate its appearance.

```
import { Directive, ElementRef, Renderer2, Input } from '@angular/core';

@Directive({
  selector: '[disableProduct]'
})
```

```
  constructor(private el: ElementRef, private renderer: Renderer2) {}
}

export class DisableProductDirective {
  @Input() set disableProduct(isDisabled: boolean) {
    if (isDisabled) {
      this.renderer.setStyle(this.el.nativeElement, 'opacity', '0.5');
      this.renderer.setStyle(this.el.nativeElement, 'pointer-events', 'none');
    } else {
      this.renderer.removeStyle(this.el.nativeElement, 'opacity');
      this.renderer.removeStyle(this.el.nativeElement, 'pointer-events');
    }
  }
```

Copy code

3. **Using the Directive:** Apply this directive to a product element in your component's template. For example, in `product.component.html`:

html

```
<div *ngFor="let product of products" [disableProduct]="!product.isInStock">
  {{ product.name }}
</div>
```

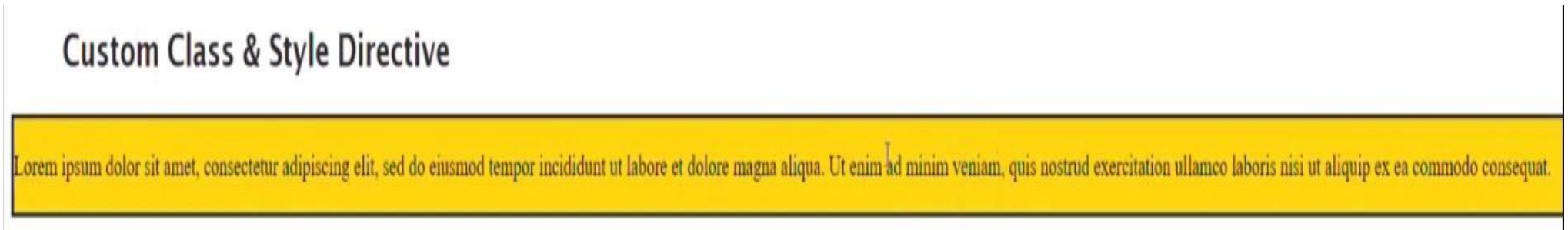
Here, the `disableProduct` directive will disable and gray out products that are out of stock (`isInStock` is `false`).

Key Points:

- **ElementRef** provides access to the DOM element.
- **Renderer2** is used to manipulate styles, classes, and attributes in a way that ensures compatibility across different environments.
- The directive uses an `@Input` property to conditionally apply styles based on whether the product is in stock.

This directive is reusable and can be applied to any DOM element where conditional styling or functionality is required.

Creating a Custom Class Directive



app.component.css | app.component.html

src > app > app.component.html > div

Go to component

```
<h2>Custom Class & Style Directive</h2>
<div [ngClass]="{appHighlight: 10 > 7}">
  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
    ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
    nisi ut aliquip ex ea commodo consequat.
  </p>
</div>
```

```
.app-container{
    margin: 20px 50px;
    padding: 10px 20px;
}
.appHighlight{
    border: #28282B 3px solid;
    background-color: gold;
}
```

```
# app.component.css    <> app.component.html X
src > app > <> app.component.html > div > p
      Go to component
1   <h2>Custom Class & Style Directive</h2>
2   <div [ngClass]="{appHighlight: 10 > 7, 'app-container': true}">
3     <p>
4       Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
5       ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco lat
6       nisi ut aliquip ex ea commodo consequat.
7     </p>
8   </div>
```

Creating a Custom Style Directive

Notes on Custom Style Directive in Angular

A **custom style directive** in Angular is a type of directive that allows you to dynamically change the styles of an element. Directives are one of the core features in Angular, and they provide a way to extend the behavior of HTML elements. A **style directive** allows you to apply CSS styles dynamically based on certain conditions or actions in your Angular application.

In Angular, there are three types of directives:

1. **Structural Directives**: Change the DOM layout (e.g., `*ngFor`, `*ngIf`).
2. **Attribute Directives**: Change the appearance or behavior of an element (e.g., `ngClass`, `ngStyle`).
3. **Component Directives**: A special kind of directive that uses a template and a view (e.g., Angular components).

Custom style directives belong to the **attribute directives** category. They allow you to manipulate the DOM by adding or modifying styles dynamically.

1. Definition of Custom Style Directive

A **custom style directive** is used to dynamically manipulate the **CSS styles** of an HTML element when certain conditions or events occur. This is useful for scenarios like changing the background color of an element when a user interacts with it or applying a set of styles conditionally.

2. **Creating a Custom Style Directive**

To create a custom style directive, follow these steps:

2.1 **Step 1: Define the Directive**

Create a new directive using the Angular CLI:

```
ng generate directive highlight
```

This command will create a new directive called `highlight`. It will automatically create the directive file `highlight.directive.ts`.

2.2 **Step 2: Implement the Directive**

In the directive class, you can inject `ElementRef` to access the DOM element and `Renderer2` to safely manipulate the styles.

Here's an example of a custom directive that changes the background color of an element when the mouse hovers over it.

Code for `highlight.directive.ts`:

```
import { Directive, ElementRef, Renderer2, HostListener } from '@angular/core';

@Directive({
  selector: '[appHighlight]' // Selector to apply the directive in HTML
})
export class HighlightDirective {
```

```
constructor(private el: ElementRef, private renderer: Renderer2) {}

// Event listener for mouse enter
@HostListener('mouseenter') onMouseEnter() {
  this.changeBackgroundColor('yellow'); // Change color on hover
}
```

```
// Event listener for mouse leave
@HostListener('mouseleave') onMouseLeave() {
  this.changeBackgroundColor('transparent'); // Reset color when mouse leaves
}

private changeBackgroundColor(color: string) {
  // Set the background color using Renderer2 for better security and cross-browser comp
  this.renderer.setStyle(this.el.nativeElement, 'background-color', color);
}
```

Explanation:

- `ElementRef`: A reference to the host DOM element (the element the directive is applied to).
- `Renderer2`: A safer way to manipulate DOM properties, ensuring that changes are rendered in a platform-independent way.
- `@HostListener`: Listens to events (e.g., `mouseenter` and `mouseleave`) on the host element and triggers the method that changes the styles.

2.3 **Step 3: Declare the Directive in the Module**

In order for Angular to recognize and use the directive, you need to declare it in the module (`app.module.ts`).

Code for `app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { HighlightDirective } from './highlight.directive'; // Import the directive

@NgModule({
  declarations: [
    AppComponent,
    HighlightDirective // Declare the directive
  ],
```

2.4 **Step 4: Apply the Directive in HTML**

Once the directive is created and declared, you can use it in your component's template by adding the directive's selector as an attribute to an element.

Example HTML (`app.component.html`):

```
<p appHighlight>Hover over this text to change the background color!</p>
```

Here, the directive `appHighlight` is applied to the `<p>` element. When you hover over the text, the background color will change to yellow.

3. Using Custom Style Directive with Input Parameters

You can also pass input parameters to customize the behavior of your directive. For example, let's create a directive that allows you to change the background color dynamically based on a provided color value.

3.1 Directive with Input Property

Add an `@Input()` property to the directive to accept values.

Code for `highlight.directive.ts` with `@Input()`:

```
import { Directive, ElementRef, Renderer2, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
```

```
  @Input() highlightColor: string = 'yellow'; // Input property to accept color value

  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.changeBackgroundColor(this.highlightColor); // Apply the color passed via Input
  }
```

```
  @HostListener('mouseleave') onMouseLeave() {
    this.changeBackgroundColor('transparent'); // Reset color
  }

  private changeBackgroundColor(color: string) {
    this.renderer.setStyle(this.el.nativeElement, 'background-color', color);
  }
```

3.2 Using the Directive with an Input Parameter in HTML

You can now pass a custom color to the directive in your HTML.

Code for `app.component.html`:

```
<p [appHighlight]="'lightblue'">Hover over this text to change the background color to lightblue!</p>
```

4. Use Case Example: Changing Text Color Based on Condition

Let's consider a scenario where we want to change the text color based on a condition (e.g., if the value is positive, make it green; if negative, make it red).

Code for `text-color.directive.ts`:

```
import { Directive, ElementRef, Renderer2, Input, OnChanges, SimpleChanges } from '@angular

@Directive({
  selector: '[appTextColor]'
})
export class TextColorDirective implements OnChanges {
  @Input() appTextColor: number = 0; // Input property to pass a number

  constructor(private el: ElementRef, private renderer: Renderer2) {}

  ngOnChanges(changes: SimpleChanges): void {
    if (changes['appTextColor']) {
      this.updateTextColor();
    }
  }

  private updateTextColor() {
    const color = this.appTextColor > 0 ? 'green' : (this.appTextColor < 0 ? 'red' : 'blacl
    this.renderer.setStyle(this.el.nativeElement, 'color', color); // Set text color
  }
```

Example HTML for Conditional Text Color:

html Copy

```
<p [appTextColor]="100">This text is green because the number is positive.</p>
<p [appTextColor]="-50">This text is red because the number is negative.</p>
```

In this example, the text color changes dynamically based on the numeric value passed to the `appTextColor` directive.

5. Summary of Key Concepts:

- **Custom Style Directives** allow you to modify the styles of an element based on user interaction or dynamic conditions.
- `ElementRef` provides direct access to the DOM element, and `Renderer2` ensures platform-independent and secure DOM manipulation.
- You can use `@HostListener` to listen for events like mouseenter and mouseleave to trigger style changes.
- `@Input()` allows passing dynamic values to the directive, enabling customizable behavior.

Conclusion:

Custom style directives in Angular are powerful tools that allow developers to dynamically change the appearance of elements based on events or input values. They enhance the flexibility and reusability of components while maintaining separation of concerns by encapsulating style logic within a directive.

How a Structural Directive Works

Notes on Structural Directives in Angular

Custom Attribute Directive Recap

- **Custom Attribute Directives**: Used to modify the appearance or behavior of DOM elements.
    - Example: Adding or removing classes based on some condition.
- **Data Binding in Attribute Directives**: Custom attribute directives can accept data inputs via `@Input`.

Introduction to Structural Directives

- **Structural Directives** modify the DOM structure by adding or removing elements.
    - Example: `*ngIf`, `*ngFor`, and `*ngSwitch`.

Understanding Structural Directives (`ngIf`)

1. `ngIf` **Example:**

html Copy code

```
<div *ngIf="display">
  <!-- Content rendered if display is true -->
</div>
```

- **ngIf** is used with an asterisk (`*`) to indicate it's a structural directive.
- Based on the condition (`display`), it either renders or removes elements from the DOM.

2. **Component Setup:**
    - HTML Structure:

html Copy code

```
<div>
  <button (click)="toggleDisplay()">Toggle Content</button>
</div>
<div *ngIf="display">
  <h1>Terms of Service</h1>
  <p>Some content about terms...</p>
</div>
```

```
export class AppComponent {
  display = false;

  toggleDisplay() {
    this.display = !this.display;
  }
}
```

How `ngIf` Works Behind the Scenes

1. **Template Wrapping**: Angular wraps elements using structural directives inside an `<ng-template>`.
    - Example:

```
<ng-template [ngIf]="display">
  <div>
    <h1>Terms of Service</h1>
  </div>
</ng-template>
```

2. **Moving the Directive**: Angular moves the directive onto the `<ng-template>` and removes the asterisk (`*`).
    - It converts:

html Copy code

```
<div *ngIf="display">...</div>
```

Into:

html Copy code

```
<ng-template [ngIf]="display">
  <div>...</div>
</ng-template>
```

`ngIf` with `else`

1. **Using `else` in `ngIf`:**
   - HTML:

- HTML:

```html
<div *ngIf="display; else noDisplay">Terms of Service</div>
<ng-template #noDisplay>
  <div>Content not available</div>
</ng-template>
```

- This renders `noDisplay` template if the condition in `ngIf` is false.

2. **Behind the Scenes**: When using `else`, Angular creates two templates:
   - One for `ngIf` and another for the `else` block.
   - Converted Code:

```
<ng-template [ngIf]="display" [ngIfElse]="noDisplay">
  <div>Terms of Service</div>
</ng-template>
<ng-template #noDisplay>
  <div>Content not available</div>
</ng-template>
```

Summary of Angular Structural Directive Mechanism

- Angular structural directives (like `ngIf`) modify DOM by:
  1. Wrapping content in an `<ng-template>`.
  2. Removing the asterisk (`*`).
  3. Treating the directive like an attribute directive with conditions.
  4. Conditionally rendering the wrapped content based on the directive's logic.

In the next step, you can create a **custom structural directive** by understanding this process.

Creating a Custom Structural Directive

```
<h2>Parent Component</h2>
<div *ngIf="display">
    <h2>Some Heading</h2>
    <p>This is a paragraph</p>
    <p>This is another paragraph</p>
</div>
```

```
<h2>Parent Component</h2>
<ng-template [ngIf]="display">
    <div>
        <h2>Some Heading</h2>
        <p>This is a paragraph</p>
        <p>This is another paragraph</p>
    </div>
</ng-template>
```

Notes on Creating a Custom Structural Directive in Angular

1. Overview of Structural Directives:

- Structural directives in Angular change the layout of the DOM by adding or removing elements. A common example is `ngIf`, which conditionally renders elements.
- Structural directives are typically prefixed with an asterisk `*`, signaling that Angular will rewrite the DOM structure behind the scenes.

2. What Happens Behind the Scenes?

- When Angular processes a structural directive (e.g., `*ngIf`), it automatically:
    - Wraps the element inside an `ng-template`.
    - Moves the directive from the host element to the `ng-template`.
    - Conditionally renders or removes the element based on the directive's logic.

3. Steps to Create a Custom Structural Directive (Similar to `ngIf`)

- Create a new directory and a directive file, e.g., `if.directive.ts`.
- In the directive file:
    - Use the `@Directive` decorator from Angular.
    - Define a selector that corresponds to the directive's name (e.g., `appIf`).

4. Important Imports and Setup:

- **Template Reference**: To manipulate the view, you need `TemplateRef` to reference the DOM element (view).
- **View Container**: You use `ViewContainerRef` to add or remove the view from the DOM.
- You need to import `TemplateRef` and `ViewContainerRef` from `@angular/core`.

5. Example of Directive Class:

typescript

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

@Directive({
  selector: '[appIf]'
})
export class IfDirective {
  constructor(private template: TemplateRef<any>, private viewContainer: ViewContainerRef)

  @Input() set appIf(condition: boolean) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.template);
    } else {
      this.viewContainer.clear();
    }
```

6. Explanation of the Code:

- **Constructor**: Receives `TemplateRef` (view reference) and `ViewContainerRef` (where to insert/remove the view).
- **Input Property (`appIf`)**:
  - This property controls whether the element should be displayed based on the condition.
  - If the condition is `true`, the content inside the directive is added to the DOM using `createEmbeddedView()`.
- If the condition is `false`, it removes the content using `clear()`.

7. Modifying the `app.module.ts`:

- The custom directive needs to be declared in the `declarations` array of your `AppModule`.

```
import { IfDirective } from './custom-directives/if.directive';

@NgModule({
  declarations: [IfDirective, /* other components */],
  imports: [/* other modules */],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

8. Usage of Custom Structural Directive in HTML:

- In your HTML, apply the custom directive with the asterisk `*` syntax:

```
html                                                    Copy code

<div *appIf="condition">This div will be conditionally rendered.</div>
```

- The `condition` can be a boolean expression (e.g., `true` or `false`), and based on the condition, the content will either be displayed or removed.

9. Handling Conditions Dynamically:

- You can bind the `appIf` directive to a property in your component, and when the value of that property changes, Angular will update the DOM.

10. Final Result:

- The directive conditionally renders the DOM elements when `true` and removes them when `false`.

11. Example in Action:

1. Component Template:

```html
<button (click)="toggleDisplay()">Toggle Display</button>
<div *appIf="isVisible">This is a conditional div.</div>
```

2. Component Logic:

```typescript
export class AppComponent {
  isVisible = false;

  toggleDisplay() {
    this.isVisible = !this.isVisible;
  }
```

12. Summary:

- You created a custom structural directive `appIf` to conditionally render elements.
- The directive manipulates the DOM by embedding views into the `ViewContainerRef` based on a boolean input condition.
- You learned how to bind the directive to a DOM element and toggle visibility using Angular's directive system.

ngSwitch Directives

Notes on `ngSwitch` Directive in Angular

Overview:

- `ngSwitch` is a **structural directive** that manipulates the DOM by adding or removing elements based on a condition.
- It functions similarly to a `switch` statement in programming languages, allowing you to display specific content based on the value of an expression.

Key Points:

1. **Usage of `ngSwitch`:**
   - `ngSwitch` directive is used on a parent container element.
   - Its associated case directives (`ngSwitchCase` and `ngSwitchDefault`) determine which child elements should be rendered based on the value of an expression.

- Corresponding functions in the component class:

typescript

```typescript
onInfoClick() {
  this.tab = 'info';
}
```

2. **Binding `ngSwitch`:**
   - The `ngSwitch` directive is assigned an expression that is evaluated, e.g., a string or a property.
   - It does not manipulate the DOM directly, but works with `ngSwitchCase` to determine which content to render.

```html
<div [ngSwitch]="tab">
  <!-- Different divs for each case -->
  <div *ngSwitchCase="'info'">Info Content</div>
  <div *ngSwitchCase="'service'">Service Content</div>
  <div *ngSwitchCase="'privacy'">Privacy Content</div>
  <div *ngSwitchCase="'user'">User Agreement</div>
  <div *ngSwitchDefault>Default Content</div>
</div>
```

3. **Handling Events to Update `ngSwitch`:**
   - A `click` event is bound to buttons, which updates the value of the `tab` variable, controlling which content is shown.

html

```html
<button (click)="onInfoClick()">Info</button>
<button (click)="onServiceClick()">Service</button>
<button (click)="onPrivacyClick()">Privacy</button>
<button (click)="onUserClick()">User Agreement</button>
```

```
onServiceClick() {
  this.tab = 'service';
}

onPrivacyClick() {
  this.tab = 'privacy';
}
```

```
onUserClick() {
  this.tab = 'user';
}
```

4. `ngSwitchCase`:

   - `ngSwitchCase` is used on the child elements to display content when the value matches.
   - Example:

```html
<div *ngSwitchCase="'service'">Service Content</div>
```

5. `ngSwitchDefault`:

   - `ngSwitchDefault` is used to specify default content when no `ngSwitchCase` matches.

```html
<div *ngSwitchDefault>Default Content</div>
```

Example Implementation:

```html
<div [ngSwitch]="tab">
  <div *ngSwitchCase="'info'">Information Section</div>
  <div *ngSwitchCase="'service'">Service Section</div>
  <div *ngSwitchCase="'privacy'">Privacy Policy</div>
  <div *ngSwitchCase="'user'">User Agreement</div>
  <div *ngSwitchDefault>Select a tab to display content</div>
</div>
```

```html
<!-- Buttons to trigger content switch -->
<button (click)="tab = 'info'">Info</button>
<button (click)="tab = 'service'">Service</button>
<button (click)="tab = 'privacy'">Privacy</button>
<button (click)="tab = 'user'">User Agreement</button>
```

Comparison with `ngIf`:

- Unlike `ngIf`, which conditionally renders one view, `ngSwitch` can render one of several views based on multiple possible conditions.

Key Concept:

- Similar to the `switch` statement in JavaScript:

```
switch(expression) {
  case 'info':
    // render info content
    break;
  case 'service':
    // render service content
    break;
  default:
    // render default content
}
```

Summary:

- `ngSwitch` is useful when you need to show content based on multiple conditions and can handle default rendering when no match is found.