Directives

Attribute vs Structural Directives

- **Attribute** Directives
  - Look like a normal HTML Attribute (possibly with databinding or event binding)
  - Only affect/change the element they're added to
- **Structural** Directives
  - Look like a normal HTML Attribute but have a leading ***** (for desugaring)
  - Affect a whole area in the DOM (addition/removal of elements)

ngFor and ngIf Recap

- Can't place two **Structural Directives** (template bindings) in one element
  - For instance, it's impermissible to use ***ngFor** and ***ngIf** on one element
  - Can restructure this by using an ***ngIf** directive on a div that houses an ***ngFor** directive

ngClass and ngStyle Recap

- **ngClass**
  - Syntax → **[ngClass] = { <class-name>: <expression> }**
    - If the expression evaluates to **True**, then the class is added
  - Square brackets **([ ])** indicate that we're binding to some property on our **ngClass** directive
- **ngStyle**
  - **Syntax**
    - Simple → **[ngStyle] = { <css-property>: <css-value> }**
    - Ternary → **[ngStyle] = { <css-property>: <condition> ? <true-result> : <false-result> }**
  - Allows us to pass a property on the same directive

Creating a Basic Attribute Directive

- **First**, create a basic directive file
  - Syntax→ **<directive-name>.directive.ts**
  - Can be done with **ng generate directive <directive-name>**
- **Second**, export a class of that very name
- **Third**, make it a directive by adding the **@Directive Decorator**
  - Takes in an object containing a **selector** field
    - Syntax → **@Decorator({ selector: '[sampleDirective]' })**
    - Directive's name written in camelCase
- **Fourth**, access the element to modify its properties via **injection**
  - Done by declaring an **ElementRef** in the constructor and touching its properties
  - Simple, but not-quite-right approach
    - **this.elementRef.nativeElement.style.backgroundColor = 'green'**
    - This approach is bad because there are cases when Angular renders templates without a DOM
- **Fifth**, write the directive's **selector** in the element's head

Using the Renderer to Build a Better Attribute Directive

- Why using **Renderer2** is better
  - Angular isn't limited to the browser, meaning it may function without a DOM → Unlike before, this survives without a DOM
  - It encapsulates data by using accessor methods
- First declare **Renderer2** and **ElementRef** in the constructor
  - This property contains several built-in methods to alter element characteristics
  - Example → **this.<renderer-object>.setStyle(this.<element-ref-object>, <css-property>, <css-value>, <optional-flags>)**

Using HostListener to Listen to Host Events

- One way of making directives listen to events is by adding the **@HostListener Decorator** with a method to execute
  - Takes the argument's name as input → Such as "mouseenter"
  - Use in conjunction with **Renderer2**
- Example → **@HostListener(<argument-name>) <method-name> (event-data) { renderer-call-from-above }**

Using HostBinding to Bind to Host Properties

- Another way of making directives listen to events is by using the **HostBinding Decorator**
  - **Example → @HostBinding(<property>.<sub-property>) <variable-name>: <variable-type> = <initial-value?;**
- This method doesn't require us to use **Renderer2**
- We change values by reassigning them in **@HostListener**
  - Example → **this.<host-binding-variable> = <new-value>**

Binding to Directive Properties

- We can use **Custom Property Binding** to bind data from other components
  - We use **@Input** in the directive component to input data, and use its corresponding variable appropriately
    - Example → **@Input() <variable-name>: <variable-type> = <default-value>;**
      - Better to put the assignment in **ngOnInit** as the page will have obtained that data before rendering
  - The template holding the element assigns values
    - Example → **<element <directive> [<property-bind>]="<value>"><element>**
- Angular intuits if we want to bind to a property of the element or the directive
- If you're passing down a string, you can omit the **[ ]** and **" "** for the directive in the template

What Happens Behind-the-Scenes on Structural Directives?

- The ***** is required because it indicates to Angular that it represents a **Structural Directive**
- Behind the scenes, Angular transforms them into an **ng-template**
  - **ng-template** isn't rendered, but provides a template for Angular to use later when a condition is true for rendering
  - Allows us to use **ngIf** property binding
- Example
  - **<div *ngIf = <condition>> . . . **
  - **<ng-template [ngIf] = <condition>> . . . </ng-template>**

Building a Structural Directive

- Must get the condition as an input via **@Input**
- Whenever the condition (like an input parameter) changes, we want to execute a method
  - Here, we make it a method by using the **set** keyword
  - The method takes the condition as input
- This ultimately sits at the **ng-template** component as that's where the ***** would put it
- We can access the template and where it's positioned in the document via **injection** in the directive's constructor
  - **TemplateRef<type>** → Gives us access to a template, like **ElementRef**
  - **ViewContainerRef** → Tells us where the directive is stored in the document
- If the method's condition is **true**, we **this.vcRef.createEmbeddedView(this.templateRef)**; else, we **this.vcRef.clear()**
- We can replace the original directive with the new one in the holding template → Don't forget the *****

Understanding ngSwitch

- Specifies an expression to match against, allowing for highly variable output
- Done by binding a **value** to **ngSwitch** via **[property-binding]**
- All elements inside the **ngSwitch** are given an **ngSwitchCase** attribute, with the last likely getting **ngSwitchDefault**
- Example
    - <div [ngSwitch] = "value">
        <p *ngSwitchCase="5">Value is 5</p>
        <p *ngSwitchDefault>Value is Default</p>