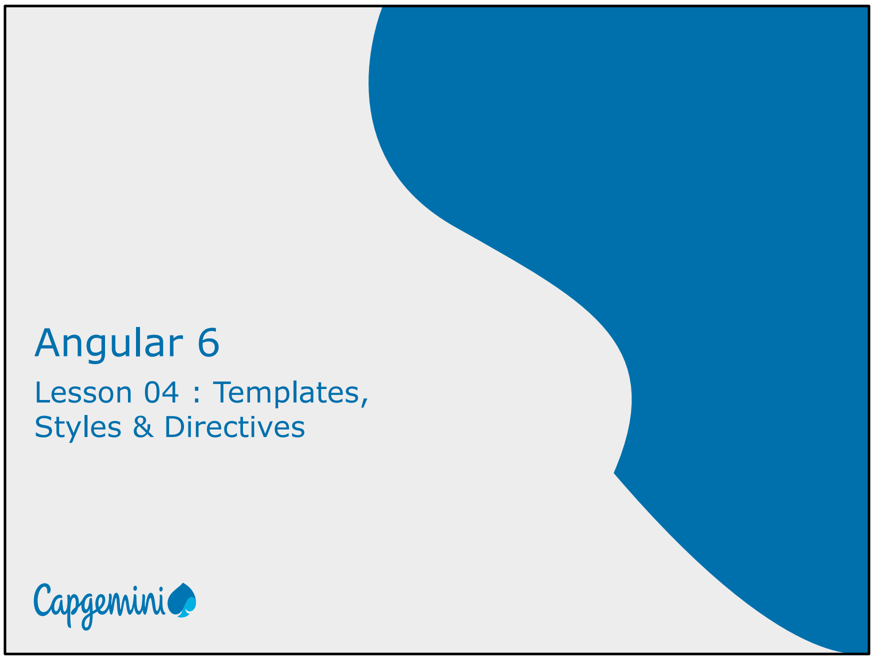
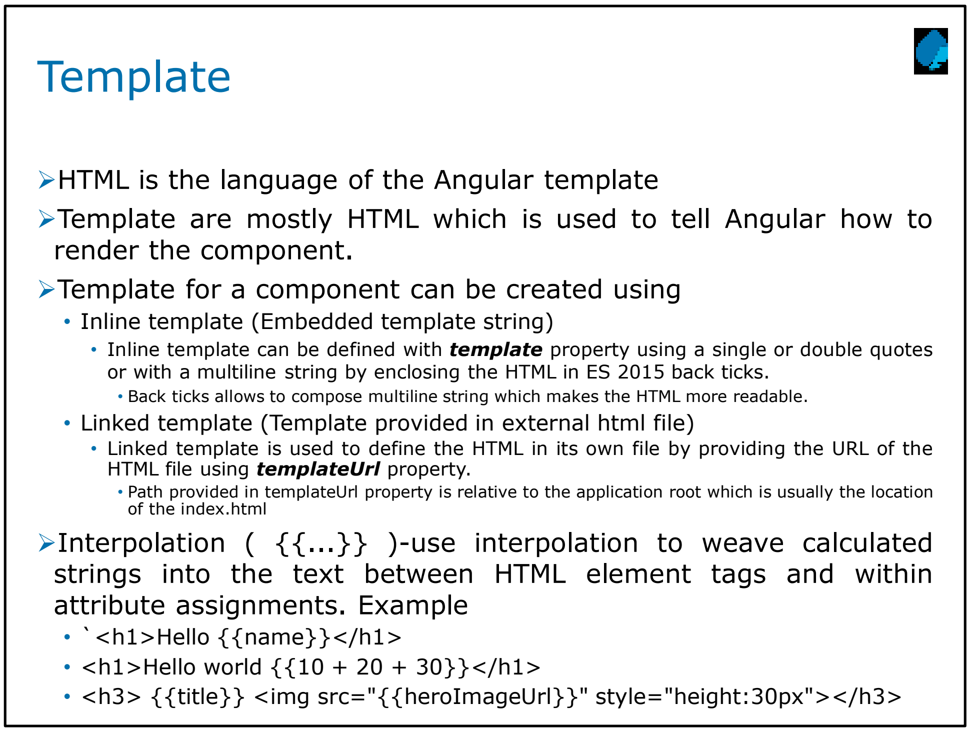


Angular 6

Lesson 04 : Templates, Styles & Directives

Lesson Objectives

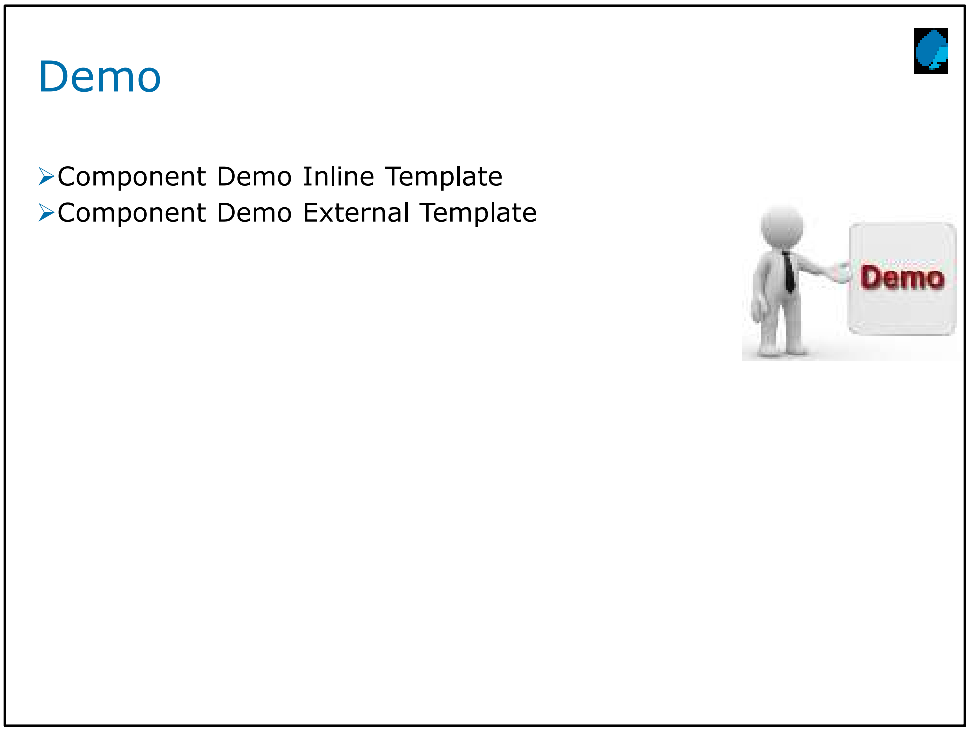
- ➢Using Templates & Styles in angular app
- ➢Built-in Directives
- ➢Creating Attribute Directive
- ➢Using Renderer to build attribute directive
- ➢Host Listener to listen to Host Events
- ➢Using Host Binding to bind to Host Properties
- ➢Building Structural Directives



Template

- ➢HTML is the language of the Angular template
- ➢Template are mostly HTML which is used to tell Angular how to render the component.
- ➢Template for a component can be created using
  - Inline template (Embedded template string)
    - Inline template can be defined with ***template*** property using a single or double quotes or with a multiline string by enclosing the HTML in ES 2015 back ticks.
      - Back ticks allows to compose multiline string which makes the HTML more readable.
  - Linked template (Template provided in external html file)
    - Linked template is used to define the HTML in its own file by providing the URL of the HTML file using ***templateUrl*** property.
      - Path provided in templateUrl property is relative to the application root which is usually the location of the index.html
- ➢Interpolation ({{...}})-use interpolation to weave calculated strings into the text between HTML element tags and within attribute assignments. Example
  - `<h1>Hello {{name}}</h1>
  - <h1>Hello world {{10 + 20 + 30}}</h1>
  - <h3> {{title}} <img src="{{heroImageUrl}}" style="height:30px"></h3>

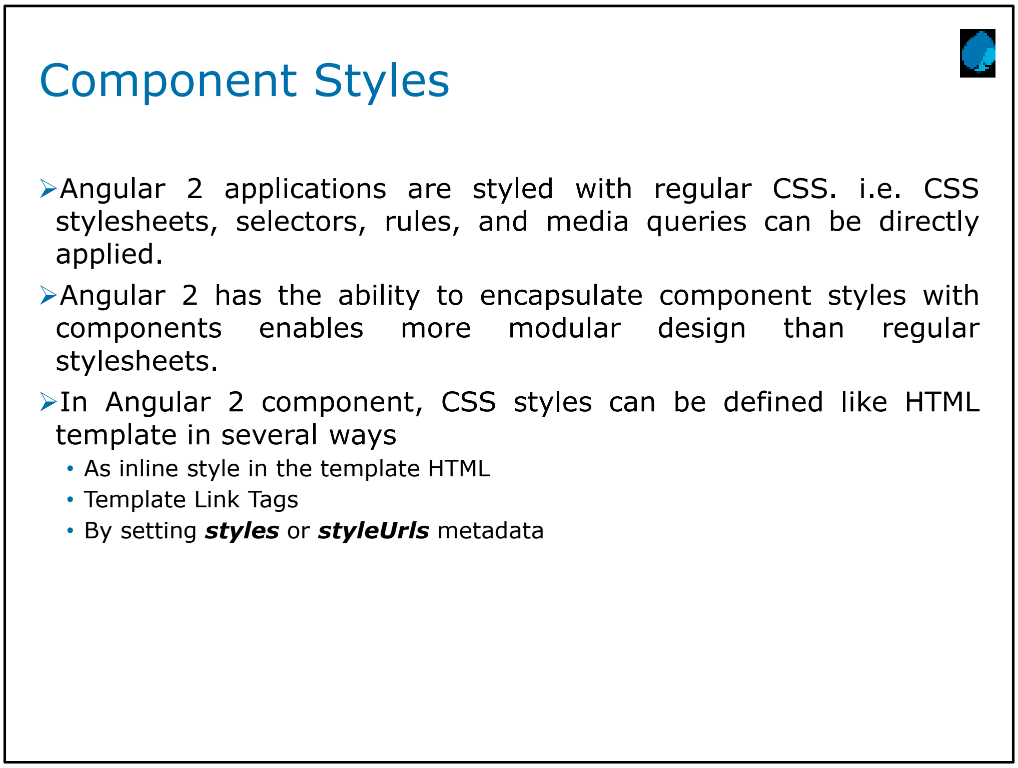
The expression can invoke methods of the host component such as getVal()
<!-- "The sum of 1 + 1 is not 4" --> <p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
A template **expression** produces a value. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive.
The interpolation braces in {{1 + 1}} surround the template expression 1 + 1. a template expression appears in quotes to the right of the = symbol as in [property]="expression".
We can write these template expressions in a language that looks like JavaScript. Many JavaScript expressions are legal template expressions, but not all.
JavaScript expressions that have or promote side effects are prohibited, including:
assignments (=, +=, -=, ...)
new
chaining expressions with ; or ,
increment and decrement operators (++ and --)



Demo

- Component Demo Inline Template
- Component Demo External Template

Add instructor notes here.



Component Styles

- Angular 2 applications are styled with regular CSS. i.e. CSS stylesheets, selectors, rules, and media queries can be directly applied.
- Angular 2 has the ability to encapsulate component styles with components enables more modular design than regular stylesheets.
- In Angular 2 component, CSS styles can be defined like HTML template in several ways
  - As inline style in the template HTML
  - Template Link Tags
  - By setting ***styles*** or ***styleUrls*** metadata

The URL is relative to the application root which is usually the location of the index.html web page that hosts the application. The style file URL is not relative to the component file.

Special selectors
Component styles have a few special selectors from the world of shadow DOM style scoping:

:host is a pseudo-class selector that applies styles in the element that hosts the component. It means if a component has a child component using component binding then child component will use :host selector that will target host element in parent component. :host selector can be used in component with styles metadata as well as with styleUrls metadata of @Component decorator.
@Component({ --- styles: [':host { position: absolute; top: 10%; }'] })

:host-context() : Looks for a CSS class in any ancestor of the component host element, all the way up to the document root. It's useful when combined with another selector. :host-context selector is used in the same way as :host selector but :host-context is used when we want to apply a style on host element on some condition outside of the component view. For the example a style could be applied on host element only if a given CSS class is found anywhere in parent tree up to the document root. In our example we have following components in parent-child relationship.
:host-context(.my-theme) h3 { background-color: green; font-style: normal; }

/deep/ : selector to force a style down through the child component tree into all the child component views. The /deep/ selector works to any depth of nested components, and it applies both to the view children and the content children of the component.
deep/ selector has alias as >>> . Component style normally applies only to the component's own template. Using /deep/ selector we can force a style down through the child component tree into all child component views. /deep/selector forces its style to its own component, child component, nested component, view children and content children.
:host /deep/ h3 { color: yellow; font-style: italic; } :host >>> p { color: white; font-style: Monospace; font-size: 20px; }

Add instructor notes here.

Component Styles (Contd…)

- Internal style

```
styles:[` p{font-weight:bold;background-color:red;}
        div{font-size: 20px;color:green}` ]
```

- External style

```
styleUrls:['./app.external.css']
```

The URL is relative to the application root which is usually the location of the index.html web page that hosts the application. The style file URL is not relative to the component file.

Special selectors
Component styles have a few special selectors from the world of shadow DOM style scoping:

:host is a pseudo-class selector that applies styles in the element that hosts the component. It means if a component has a child component using component binding then child component will use :host selector that will target host element in parent component. :host selector can be used in component with styles metadata as well as with styleUrls metadata of @Component decorator.
@Component({ --- styles: [':host { position: absolute; top: 10%; }'] })

:host-context() : Looks for a CSS class in any ancestor of the component host element, all the way up to the document root. It's useful when combined with another selector. :host-context selector is used in the same way as :host selector but :host-context is used when we want to apply a style on host element on some condition outside of the component view. For the example a style could be applied on host element only if a given CSS class is found anywhere in parent tree up to the document root. In our example we have following components in parent-child relationship.
:host-context(.my-theme) h3 { background-color: green; font-style: normal; }

/deep/ : selector to force a style down through the child component tree into all the child component views. The /deep/ selector works to any depth of nested components, and it applies both to the view children and the content children of the component.
deep/ selector has alias as >>> . Component style normally applies only to the component's own template. Using /deep/ selector we can force a style down through the child component tree into all child component views. /deep/selector forces its style to its own component, child component, nested component, view children and content children.
:host /deep/ h3 { color: yellow; font-style: italic; } :host >>> p { color: white; font-style: Monospace; font-size: 20px; }

Component Styles (Contd…)

- :host:host is a pseudo-class selector that applies styles in the element that hosts the component.
- It means if a component has a child component using component binding then child component will use :host selector that will target host element in parent component.
- :host selector can be used in component with styles metadata as well as with styleUrls metadata of @Component decorator.
- Example

```
@Component({
  ---
  styles: [ ':host { position: absolute; top: 10%; }' ]
})
```

Component Styles (Contd…)

- :host-context selector is used in the same way as :host selector but :host-context is used when we want to apply a style on host element on some condition outside of the component view.
- Example

```
:host-context(.my-theme) h3 {
    background-color: green;
    font-style: normal;
}
```

Demo

- Component Demo Style

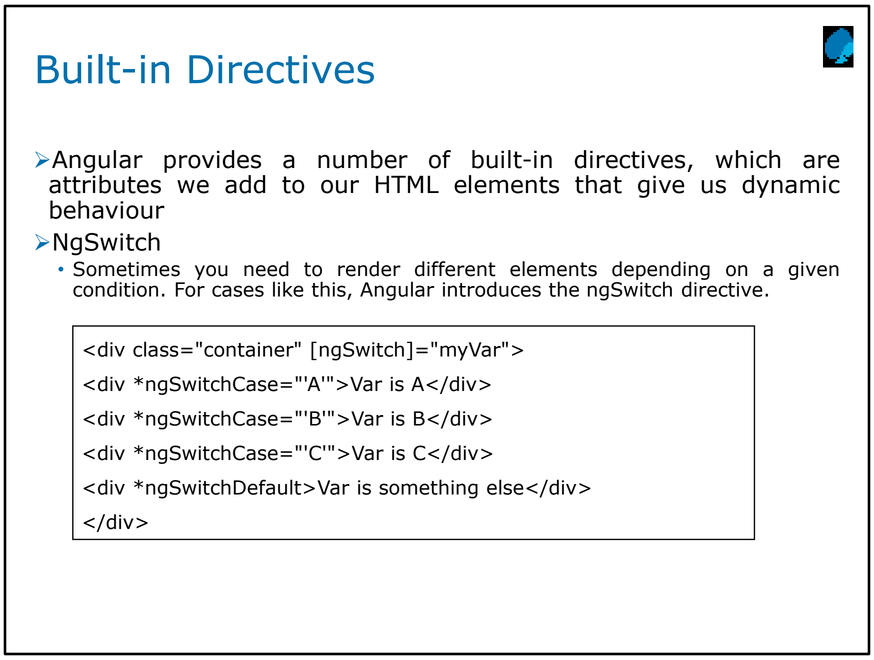
Directives

- Directives are instructions to the DOM.
- "Components" are such kind of instructions in the DOM.
- Once we place our selector of our component somewhere in out template, we are instructing angular to add content of our component template and business logic in our typescript code in that place where we use the selector.
- "Components" are directives with templates, But there are also directives without template.
- Two Type of Directives:
  - Built in directives
  - Custom directives

Built-in Directives

- Angular provides a number of built-in directives, which are attributes we add to our HTML elements that give us dynamic behaviour
- NGIF
  - The ngIf directive is used when you want to display or hide an element based on a condition.
  - The condition is determined by the result of the expression that you pass into the directive.

```
<div *ngIf="false"></div>      <!-- never displayed -->
<div *ngIf="a > b"></div>      <!-- displayed if a is more than b -->
<div *ngIf="str == 'yes'"></div> <!-- displayed if str is the string "yes"
<div *ngIf="myFunc()"></div>      <!-- displayed if myFunc returns truthy -->
```



Built-in Directives

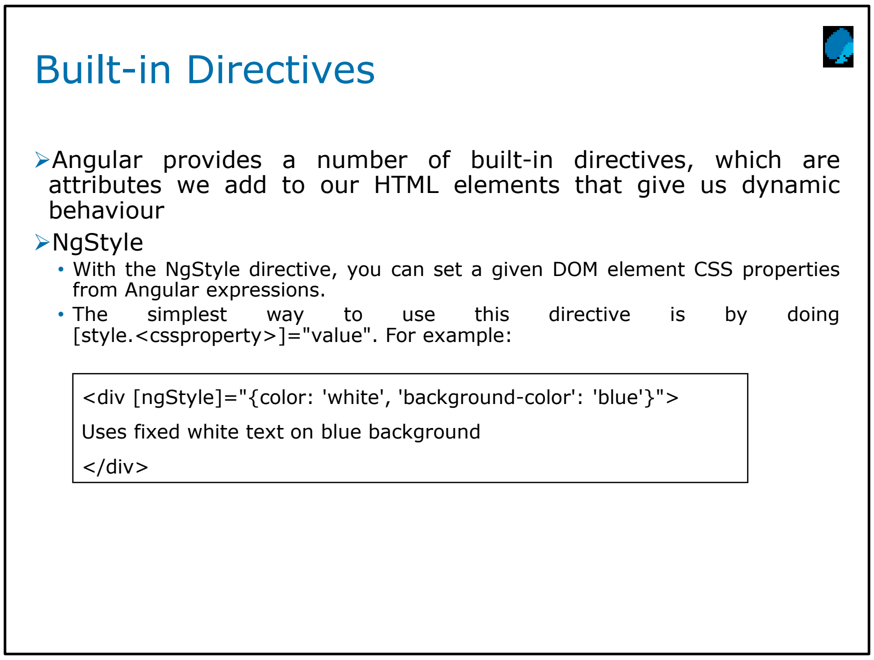
- Angular provides a number of built-in directives, which are attributes we add to our HTML elements that give us dynamic behaviour
- NgSwitch
  - Sometimes you need to render different elements depending on a given condition. For cases like this, Angular introduces the ngSwitch directive.

```
<div class="container" [ngSwitch]="myVar">
<div *ngSwitchCase="'A'">Var is A</div>
<div *ngSwitchCase="'B'">Var is B</div>
<div *ngSwitchCase="'C'">Var is C</div>
<div *ngSwitchDefault>Var is something else</div>
</div>
```

And we don’t have to touch the default (i.e. fallback) condition. Having the ngSwitchDefault element is optional. If we leave it out, nothing will be rendered when myVar fails to match any of the expected values.
You can also declare the same *ngSwitchCase value for different elements, so you’re not limited to matching only a single time. Here’s an example:

```
<h4>
Current choice is {{ choice }}
</h4>
<div>
<ul [ngSwitch]="choice">
<li *ngSwitchCase="1">First choice</li>
<li *ngSwitchCase="2">Second choice</li> 9
<li *ngSwitchCase="3">Third choice</li>
<li *ngSwitchCase="4">Fourth choice</li>
<li *ngSwitchCase="2">Second choice, again</li>
<li *ngSwitchDefault>Default choice</li>
</ul>
</div>
<div style="margin-top: 20px;">
```

```
<button class="ui primary button" (click)="nextChoice()">
Next choice
</button>
</div>
```



Built-in Directives

- Angular provides a number of built-in directives, which are attributes we add to our HTML elements that give us dynamic behaviour
- NgStyle
  - With the NgStyle directive, you can set a given DOM element CSS properties from Angular expressions.
  - The simplest way to use this directive is by doing [style.<cssproperty>]="value". For example:

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
Uses fixed white text on blue background
</div>
```

```
<div>
<input type="text" name="color" value="{{color}}" #colorinput>
</div>
<div>
<input type="text" name="fontSize" value="{{fontSize}}" #fontinput>
</div>
<button (click)="apply(colorinput.value, fontinput.value)">
Apply settings
</button>
<div>
<span [ngStyle]="{color: 'red'}" [style.font-size.px]="fontSize">
red text
</span>
</div>
apply(color: string, fontSize: number): void {
this.color = color;
this.fontSize = fontSize;
}
```

Built-in Directives

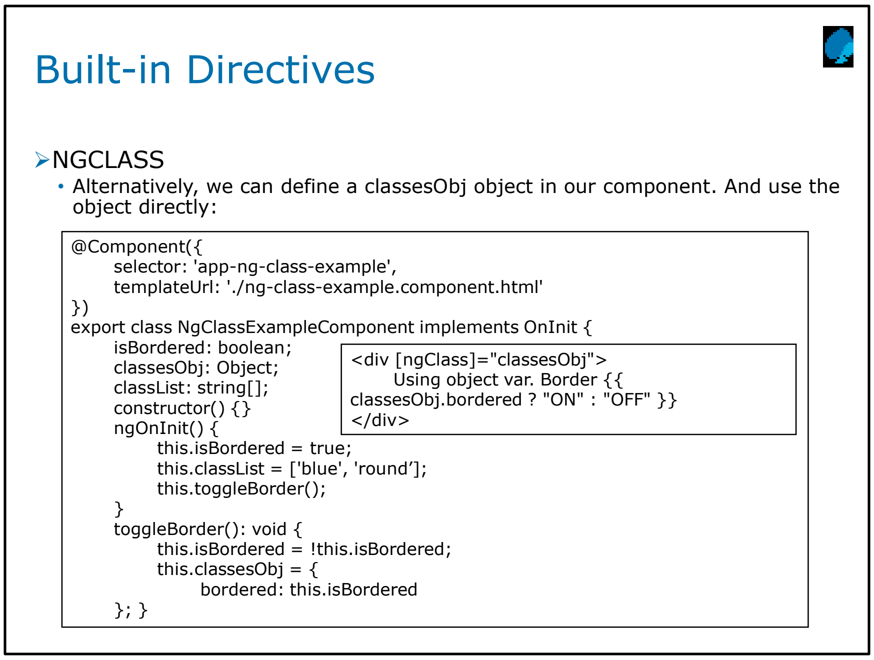
- NGCLASS
  - The NgClass directive, represented by a ngClass attribute in your HTML template, allows you to dynamically set and change the CSS classes for a given DOM element.
  - The first way to use this directive is by passing in an object literal. The object is expected to have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

code/built-in-directives/src/styles.css

```
bordered {
      border: 1px dashed black;
      background-color: #eee;
}
```

code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html

```
<div [ngClass]="{bordered: false}">This is never bordered</div>
<div [ngClass]="{bordered: true}">This is always bordered</div>
```



Built-in Directives

➢NGCLASS

- Alternatively, we can define a classesObj object in our component. And use the object directly:

```
@Component({
        selector: 'app-ng-class-example',
        templateUrl: './ng-class-example.component.html'
})
export class NgClassExampleComponent implements OnInit {
        isBordered: boolean;
        classesObj: Object;
        classList: string[];
        constructor() {}
        ngOnInit() {
                this.isBordered = true;
                this.classList = ['blue', 'round'];
                this.toggleBorder();
        }
        toggleBorder(): void {
                this.isBordered = !this.isBordered;
                this.classesObj = {
                        bordered: this.isBordered
        }; }
```

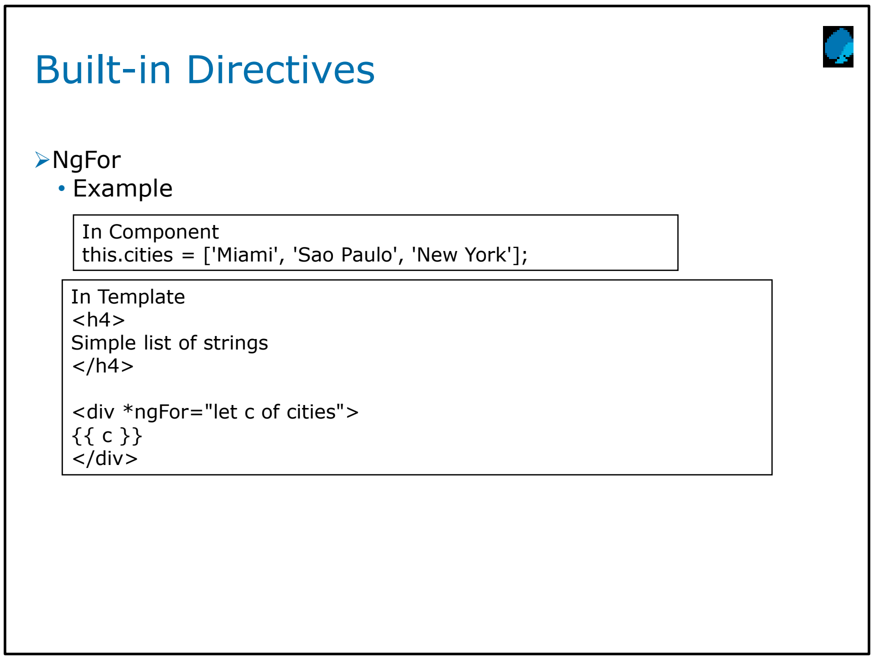
```
<div [ngClass]="classesObj">
        Using object var. Border {{
classesObj.bordered ? "ON" : "OFF" }}
</div>
```

We can also use a list of class names to specify which class names should be added to the element. For that, we can either pass in an array literal:
<div class="base" [ngClass]="['blue', 'round']">
This will always have a blue background and
round corners
</div>
Or assign an array of values to a property in our component:
this.classList = ['blue', 'round'];
And pass it in:
code/built-in-directives/src/app/ng-class-example/ng-class-example.component.html
<div class="base" [ngClass]="classList">
This is {{ classList.indexOf('blue') > -1 ? "" : "NOT" }} blue
and {{ classList.indexOf('round') > -1 ? "" : "NOT" }} round
</div>
In this last example, the [ngClass] assignment works alongside existing values assigned by the HTML class attribute.
The resulting classes added to the element will always be the set of the classes provided by usual class HTML attribute and the result of the evaluation of the [class]

directive. In this example:

Built-in Directives

- NgFor
- The role of this directive is to repeat a given DOM element (or a collection of DOM elements) and pass an element of the array on each iteration.
- The syntax is
  - *ngFor="let item of items"
  - The let item syntax specifies a (template) variable that's receiving each element of the items array;
  - The items is the collection of items from your controller



Built-in Directives

- NgFor
  - Example

```
In Component
this.cities = ['Miami', 'Sao Paulo', 'New York'];
```

```
In Template
<h4>
Simple list of strings
</h4>

<div *ngFor="let c of cities">
{{ c }}
</div>
```

We can also iterate through an array of objects like these:

```
this.people = [
{ name: 'Anderson', age: 35, city: 'Sao Paulo' },
{ name: 'John', age: 12, city: 'Miami' },
{ name: 'Peter', age: 22, city: 'New York' }
];
```

And then render a table based on each row of data

```
<h4>
List of objects
</h4>
<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>City</th>
</tr>
</thead>
<tr *ngFor="let p of people">
```

```
<td>{{ p.name }}</td>
<td>{{ p.age }}</td>
<td>{{ p.city }}</td>
</tr>
</table>
```

Built-in Directives

- NgFor
  - Example
    - Getting an index
  - There are times that we need the index of each item when we're iterating an array.
  - We can get the index by appending the syntax let idx = index to the value of our ngFor directive, separated by a semi-colon.
  - When we do this, ng will assign the current index into the variable we provide (in this case, the variable idx).
  - Note that, like JavaScript, the index is always zero based. So the index for first element is 0, 1 for the second and so on.

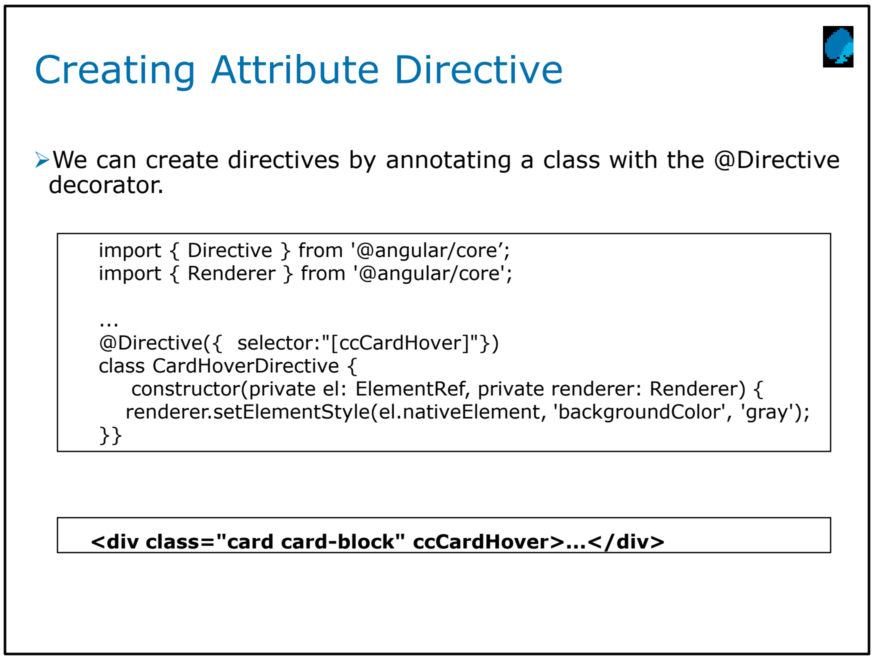
```
<div class="ui list" *ngFor="let c of cities; let num = index">
<div class="item">{{ num+1 }} - {{ c }}</div>
</div>
```

Built-in Directives

➢NGNONBINDABLE

- We use ngNonBindable when we want tell Angular not to compile or bind a particular section of our page.
- Let’s say we want to have a div that renders the contents of that content variable and right after we want to point that out by outputting <- this is what {{ content }} rendered next to the actual value of the variable.

```
<div class='ngNonBindableDemo'>
<span class="bordered">{{ content }}</span>
<span class="pre" ngNonBindable>
&larr; This is what {{ content }} rendered
</span>
</div>
```



Creating Attribute Directive

- We can create directives by annotating a class with the @Directive decorator.

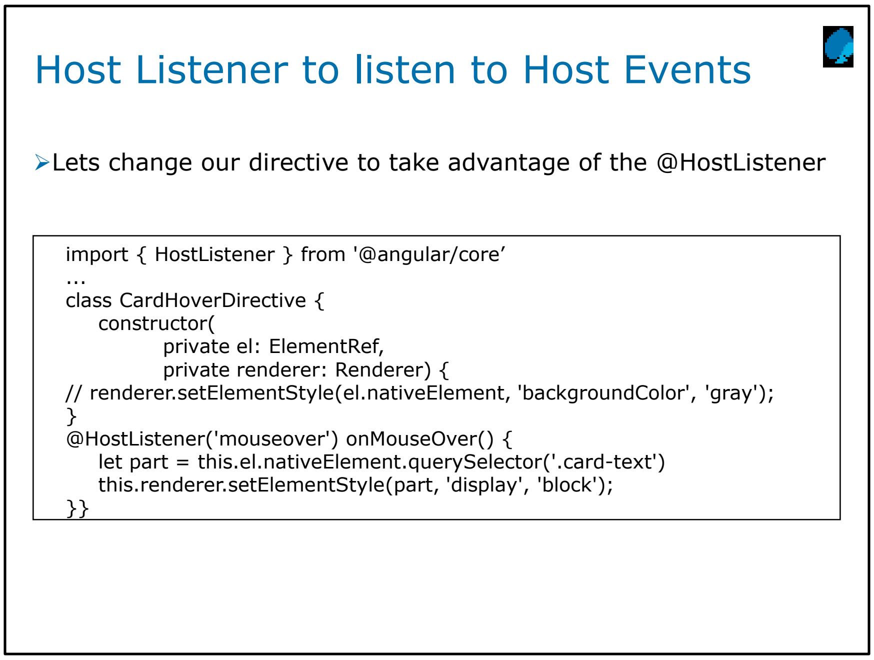
```
import { Directive } from '@angular/core’;
import { Renderer } from '@angular/core';

...
@Directive({  selector:"[ccCardHover]"})
class CardHoverDirective {
     constructor(private el: ElementRef, private renderer: Renderer) {
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
}}
```

```
<div class="card card-block" ccCardHover>...</div>
```

Host Listener to listen to Host Events

- @HostListener decorator is a function decorator that accepts an event name as an argument. When that event gets fired on the host element it calls the associated function

```
@HostListener('mouseover') onHover() {
    window.alert("hover");
}
```



Host Listener to listen to Host Events

- Lets change our directive to take advantage of the @HostListener

```
    import { HostListener } from '@angular/core’
    ...
    class CardHoverDirective {
        constructor(
                private el: ElementRef,
                private renderer: Renderer) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
    }
    @HostListener('mouseover') onMouseOver() {
        let part = this.el.nativeElement.querySelector('.card-text')
        this.renderer.setElementStyle(part, 'display', 'block');
    }}
```

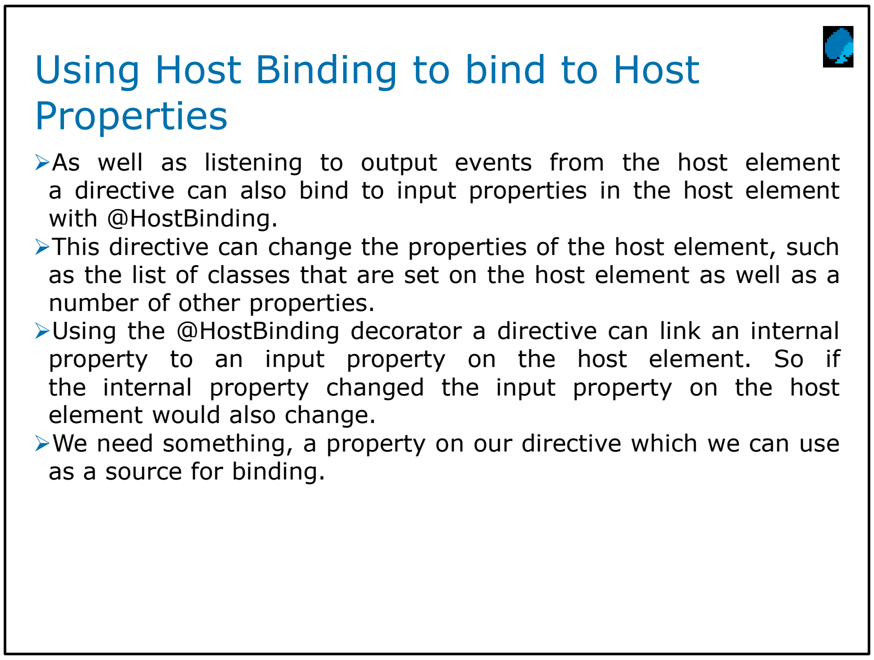
We’ve removed the code to render the background color to gray.
We decorate a class method with @HostListener configuring it to call the function on every mouseover events.
We get a reference to the DOM element that holds the text.

We set the display to block so that element is shown.
For the above to work we need to change our Component template so the joke is hidden using the display style property, like so:

```
<div class="card card-block" ccCardHover>
<h4 class="card-title">{{data.setup}}</h4>
<p class="card-text"     [style.display]="'none'">{{data.punchline}}</p>   </div>
```

As well as showing the punchline on a mouseover event we also want to *hide* the punchline on a mouseout event, like so:

```
class CardHoverDirective {
        constructor(private el: ElementRef, private renderer: Renderer) {
        // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
        }
        @HostListener('mouseover') onMouseOver() {
                let part = this.el.nativeElement.querySelector('.card-text');
                this.renderer.setElementStyle(part, 'display', 'block');
        }
        @HostListener('mouseout') onMouseOut() {
                let part = this.el.nativeElement.querySelector('.card-text');
                this.renderer.setElementStyle(part, 'display', 'none');
        }
}
```



Using Host Binding to bind to Host Properties

- As well as listening to output events from the host element a directive can also bind to input properties in the host element with @HostBinding.
- This directive can change the properties of the host element, such as the list of classes that are set on the host element as well as a number of other properties.
- Using the @HostBinding decorator a directive can link an internal property to an input property on the host element. So if the internal property changed the input property on the host element would also change.
- We need something, a property on our directive which we can use as a source for binding.

Using Host Binding to bind to Host Properties

```
import { HostBinding } from '@angular/core'
. . .
class CardHoverDirective {
  @HostBinding('class.card-outline-primary') private ishovering: boolean;
  constructor(private el: ElementRef,private renderer: Renderer) {  }
  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setElementStyle(part, 'display', 'none');
    this.ishovering = false;
  }
}
```

Add instructor notes here.

Demo

- Demo Custom Directive

Add the notes here.

Creating own structural directive

- We can create custom attribute directives and custom structural directives using @Directive decorator.
- Structural directives are responsible for HTML layout.
- We can add and remove elements in DOM layout dynamically.
- The HTML element using directive is called host element for that directive.
- To add and remove host elements from DOM layout we can use TemplateRef and ViewContainerRef classes in our structural directive.
- To change DOM layout we should use TemplateRef and ViewContainerRef in our structural directive.
- **TemplateRef** : It represents an embedded template that can be used to instantiate embedded views.
- **ViewContainerRef :** It represents a container where one or more views can be attached.

Demo

- Demo Own structural Directive