

# CUSTOM DIRECTIVES

Ori Calvo, 2017

[oric@trainologic.com](mailto:oric@trainologic.com)

<https://trainologic.com>

# Directives

2

- Directives are JavaScript classes
- Angular has 3 types of directives
  - ▣ Component Directives
  - ▣ Structural Directives
  - ▣ Attribute Directives

# Attribute directives

3

- Each attribute can be defined as directive
- Angular finds all matching attributes and instantiates a directive instance
- Attribute directive can extend multiple unrelated components
  - ▣ Think about a tooltip directive
- Usually an attribute directive changes the look and behavior of an element

# myClass

4

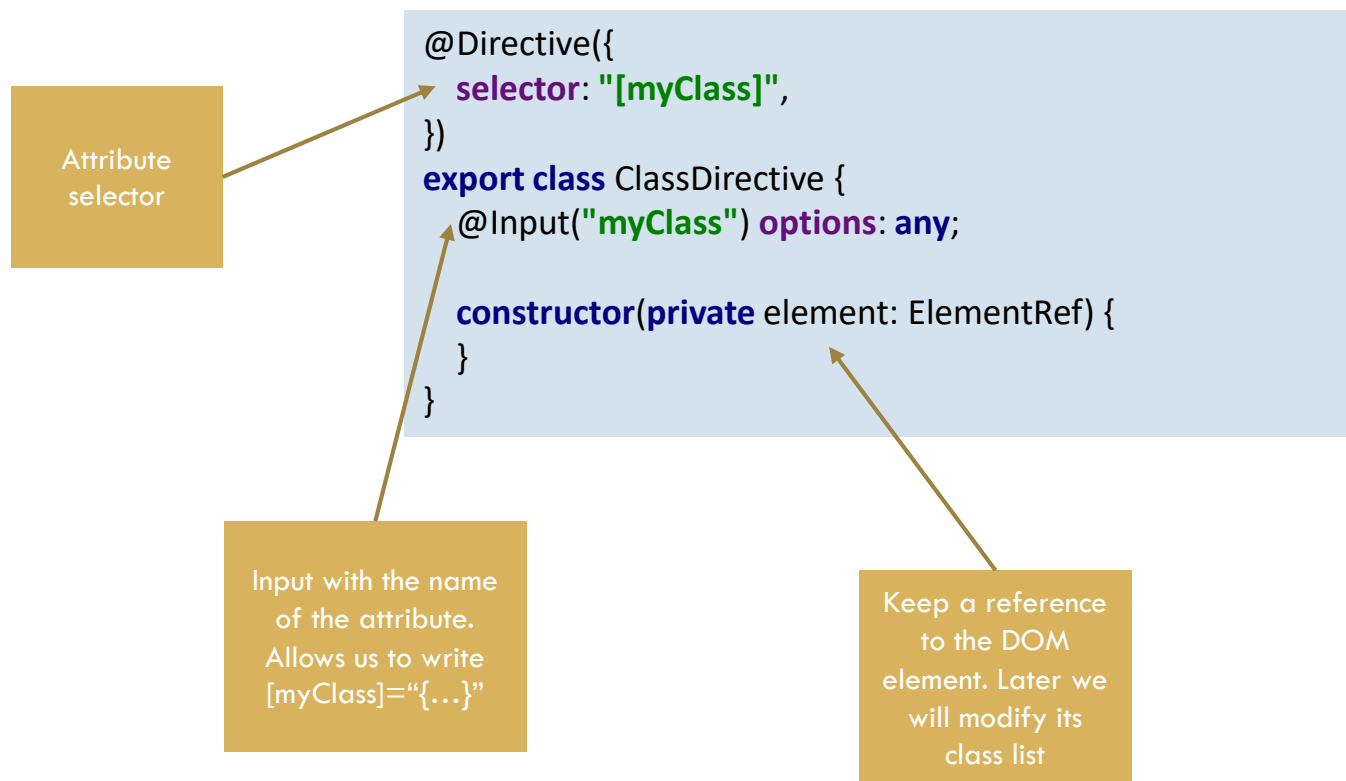
- Lets write our own version of **ngClass**

```
<li *ngFor="let contact of contacts" [myClass]="{selected: contact.selected}">  
  <span>{{ contact.name }}</span>  
  <button (click)="select(contact)">Select</button>  
</li>
```

- **myClass** gets a option object where each key is a class name and a value that is a Boolean indicator

# myClass

5



# Apply Changes

6

- Each time the options input changes we need to add/remove all CSS classes

```
private applyChanges() {  
  for(let className in this.options) {  
    if(this.options[className]) {  
      this.element.nativeElement.classList.add(className);  
    }  
    else {  
      this.element.nativeElement.classList.remove(className);  
    }  
  }  
}
```

- When should we execute **applyChanges** ?

# ngOnChanges

7

- Once a directive has one or more **@Input** the **ngOnChanges** is invoked at least once
- No need to use **ngOnInit** in that case

```
export class ClassDirective {  
  ngOnChanges() {  
    this.applyChanges();  
  }  
}
```

- What is wrong with the ngOnChanges approach ?

# Inline Changes

8

- The options object might change inline

```
<li *ngFor="let contact of contacts" [myClass]="contact.options">
  <span>{{contact.name}}</span>
  <button (click)="select(contact)">Select</button>
</li>
```

The options  
is changes  
inline

```
select(contact) {
  contact.options.selected = !contact.options.selected;
}
```

The whole  
options  
object is send

- In that case ngOnChanges is not executed ☹️



# ngDoCheck

9

- Is always executed by Angular
- The directive should compare old state with current
  - ▣ Must hold a copy of the old
- Angular offers special utility classes named **differ**

```
export interface KeyValueDiffer<K, V> {  
  diff(object: Map<K, V>): KeyValueChanges<K, V>;  
  diff(object: {  
    [key: string]: V;  
  }): KeyValueChanges<string, V>;  
}
```

```
export interface IterableDiffer<V> {  
  diff(object: NgIterable<V>): IterableChanges<V> | null;  
}
```

# ngDoCheck using KeyValueDiffer

10

## □ Create a differ object

```
ngOnChanges() {  
  this.differ = this.differs.find(this.options).create();  
}
```

## □ Use it

```
ngDoCheck() {  
  const changes = this.differ.diff(this.options);  
  
  if(changes) {  
    changes.forEachChangedItem(item => {  
      if(item.currentValue) {  
        this.element.nativeElement.classList.add(item.key);  
      }  
      else {  
        this.element.nativeElement.classList.remove(item.key);  
      }  
    });  
  }  
}
```

# Respond to user events

11

- A directive has no template
- Therefore it need to register user event through code

```
export class ClickDirective {  
  @Output("myClick") ev: EventEmitter<any> = new EventEmitter<any>();  
  
  constructor(private element: ElementRef) {  
  }  
  
  ngOnInit() {  
    this.element.nativeElement.addEventListener("click", () => {  
      console.log("Click detected");  
    });  
  }  
}
```

- Don't forget to unregister the event !!!
  - ▣ Usually during **ngOnDestory**

# @HostListener

12

- The Angular way to “addEventListener” without accessing the DOM
- Automatically unregisters during ngOnDestroy
- There are two different syntaxes

```
export class ClickDirective {  
  @HostListener("click", ["$event"])  
  private onClick($event) {  
  }  
}
```

```
@Directive({  
  host: {  
    "(click)": "onClick($event)"  
  }  
})  
export class ClickDirective {  
  private onClick($event) {  
  }  
}
```

# @HostBinding

13

- Same as HostListener but for attribute binding instead of event handling

```
export class ClickDirective {  
  @HostBinding("disabled")  
  private disabled: boolean = true;  
  
  ngOnInit() {  
    setTimeout(() => {  
      this.disabled = false;  
    }, 2500);  
  }  
}
```

```
@Directive({  
  host: {  
    "[disabled]": "disabled"  
  }  
})  
export class ClickDirective {  
  private disabled: boolean = true;  
  
  ngOnInit() {  
    setTimeout(() => {  
      this.disabled = false;  
    }, 2500);  
  }  
}
```

# Structural directives

14

- Structural directives are meant to change the DOM structure
- For example **ngIf** will change the DOM's element structure according to the respective condition
- To understand Structural directive we must first learn how to work with **ngTemplate**

# Implementing Tooltip

15

- A tooltip can be attached to any component
- Therefore cannot be implemented as component

```
@Directive({  
  selector: "[myTooltip]",  
})  
export class TooltipDirective {  
  constructor(private element: ElementRef) {  
  }  
  
  @HostListener("mouseenter")  
  private onMouseEnter() {  
  }  
}
```

- How can we support template based tooltip ?

# ngTemplate

16

- Every component can define many ng-template(s)
- The template is automatically removed from the DOM
- However, it can be referenced and restored on demand

```
<button myTooltip [myTooltipTemplate]="tooltipTemplate">
  A button with tooltip
</button>

<ng-template #tooltipTemplate>
  This is a custom tooltip
  <button (click)="close()">Close</button>
</ng-template>
```

Is  
automatically  
removed



# TemplateRef

17

- Each ng-template element is associated with **TemplateRef** instance
- Use **@ViewChild** to get a reference

```
export class AppComponent {  
  @ViewChild("tooltipTemplate") tooltipTemplate: TemplateRef;  
  
  ngOnInit() {  
    console.log(this.tooltipTemplate);  
  }  
}
```

# How to use TemplateRef

18

- TemplateRef can be injected to any component using a **ViewContainerRef** object

```
export class TooltipComponent {  
  @ViewChild("marker", {read: ViewContainerRef}) marker: ViewContainerRef;  
  private viewRef: ViewRef;  
  
  show(template: TemplateRef<any>) {  
    this.viewRef = this.marker.createEmbeddedView(template);  
  }  
}
```

marker is a  
simple  
div/span

This is a live view  
which is bound to  
the template  
parent !!!

# ng-template Content

19

- The content of an ng-template is bound to the component which defined the template
- Not to the component where the template was injected to

```
<ng-template #tooltipTemplate>  
  This is a custom tooltip  
  
  <button (click)="close()">Close</button>  
</ng-template>
```

close function  
need to be  
defined inside the  
parent component

# exportAs

20

- A directive may choose to publish itself to the parent component
  - ▣ Usually in order to provide a richer API

```
@Directive({  
  selector: "[myTooltip]",  
  exportAs: "tooltip"  
})  
export class TooltipDirective {  
}
```

```
<button #buttonTooltip="tooltip"  
  myTooltip  
  [myTooltipTemplate]="tooltipTemplate">A button with  
  tooltip</button>
```

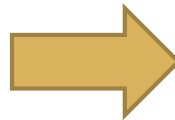
Requesting a  
reference to the  
directive

# Structural directives

21

- A directive is considered structural when it is used with the `*` syntax
  - ▣ Every directive can be used that way
- Angular creates an implicit `ng-template`
- The directive is expected to use the template and “play” with it

```
<div *myIf>  
  Dynamic content  
</div>
```



```
<div>  
  <ng-template myIf>  
    Dynamic content  
  </ng-template>  
</div>
```

# Accessing the template

22

```

@Directive({
  selector: "[myIf]"
})
export class IfDirective {
  @Input("myIf") condition: boolean;
  private viewRef: ViewRef;

  constructor(private template: TemplateRef<any>, private viewContainerRef: ViewContainerRef) {}

  ngOnChanges() {
    if(this.condition) {
      if(!this.viewRef) {
        this.viewRef = this.viewContainerRef.createEmbeddedView(this.template);
      }
    }
    else {
      if(this.viewRef) {
        this.viewRef.destroy();
        this.viewRef = null;
      }
    }
  }
}

```

The template  
is injectable



```
<button (click)="show=!show">Toggle</button>
```

```

<div *myIf="show">
  Dynamic content
</div>

```

# Multiple Structural Directives

23

- Angular does not allow multiple structural directives on the same element
- → Cannot use `ngIf` & `ngFor` on the same `<li>` ☹
  - ▣ Not like Angular 1
- Can fix that by using the `ng-template` explicit syntax

What the hell is going on here ?

```
<ul>  
  <ng-template ngFor [ngForOf]="[1,2,3]" let-num>  
    <li *ngIf="num % 2 == 0">  
      <span>{{num}}</span>  
    </li>  
  </ng-template>  
</ul>
```

# myFor

24

- Lets build our own simplified version of ngFor

```
@Directive({  
  selector: "[myFor]"  
})  
export class ForDirective {  
  @Input("myForOf") items: any[];  
  
  constructor(private template: TemplateRef<any>, private viewContainerRef: ViewContainerRef) {  
  }  
}
```

This allow us to  
use the "of"  
syntax



# Template input variable

25

- The client is using the following syntax

```
<ul>
  <li *myFor="let item of items; let i=index">
    <span>{{item}}, {{i}}</span>
  </li>
</ul>
```

- The myFor directive is responsible for defining the contextual **item** variable
- Please note that myFor does not control the name of the variable
  - ▣ As opposed to the **index** variable

# Template input variable

26

```
ngOnChanges() {  
  this.viewContainerRef.clear();  
  
  if(this.items) {  
    for(let i=0; i<this.items.length; i++) {  
      this.viewContainerRef.createEmbeddedView(this.template, {  
        index: i,  
        $implicit: this.items[i],  
      });  
    }  
  }  
}
```

\$implicit will be  
set as the value  
of item  
variable

index variable  
defined by  
myFor

clear the view  
before rendering  
new items.  
Inefficient  
implementation !!!

# Cooperating Directives

27

- Think about **myTab** and **myPage** directives
- Each page need to notify the tab parent of its existence

Out of the box support. Parent directive can be injected without any additional configuration

```
@Directive({  
  selector: "[myPage]"  
})  
export class PageDirective {  
  constructor(private tab: TabDirective) {  
  }  
  
  ngOnInit() {  
    this.tab.addPage(this);  
  }  
}
```

# Host Component

28

- A directive may interact with its host component
- The directive may be hosted by different component types
- Thus the component type cannot be used as provider
- Solution
  - ▣ Let the component publish itself using a token defined by the directive

# Accessing host component

29

```
export abstract class HostComponentWithTooltip {  
  }  
  
@Directive({  
  selector: '[appTooltip]'  
})  
export class TooltipDirective {  
  
  constructor(component: HostComponentWithTooltip) {  
    console.log("TooltipDirective.ctor", component);  
  }  
}  
  
@Component({  
  selector: 'app-clock',  
  templateUrl: './clock.component.html',  
  styleUrls: ['./clock.component.css'],  
  providers: [  
    {provide: HostComponentWithTooltip, useExisting: ClockComponent}  
  ]  
})  
export class ClockComponent {  
}
```

# Summary

30

- ❑ We usually build components not directives
- ❑ In case where you need close control over the DOM you will use directive
- ❑ A directive can extend multiple unrelated components
- ❑ Directives can talk to each other