3

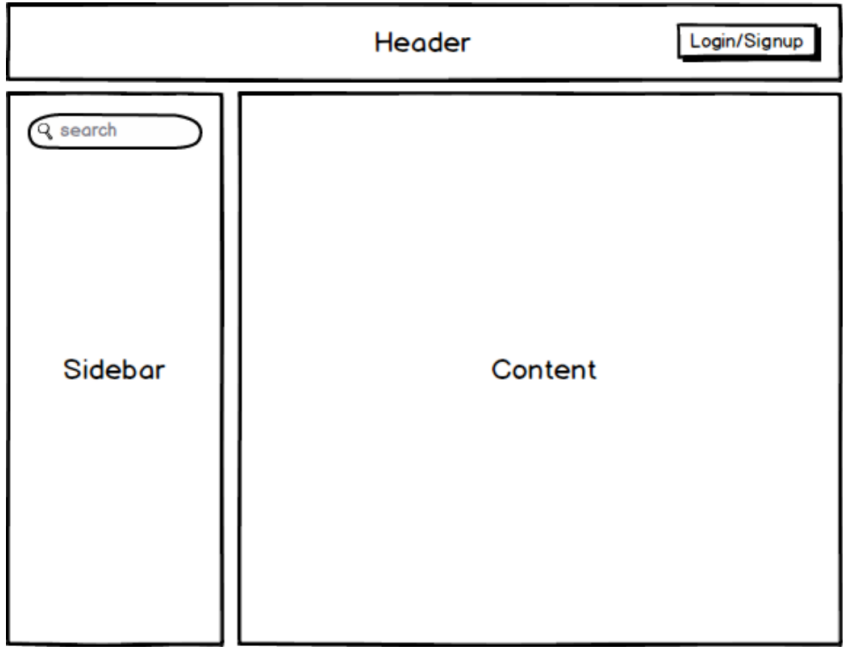
Components and Data binding

Work with components, data binding, lifecycle hooks, inputs and outputs

Architecting with component

When building an Angular app, we start by :

- Breaking down the application into separate, reusable components
- Describing the responsibility of each component
- Defining the component's **input** and **output** data



Architecting with component

Example

App component

```
<div>
  <h1>Title</h1>
  <app-header></app-header>
  <app-sidebar></app-sidebar>
  <app-content></app-content>
</div>
```

app-header component

```
<div>
  <ul>
     <li> Pizza </li>
     <li> Sign In </li>
     <li> … </li>
  </ul>
</div>
```

app-content component

```
<table *ngIf="pizzas.length > 0">
  <thead><tr><th>Name</th></tr>
  <th></th></thead>
  <tbody>
    <tr *ngFor="let pizza of pizzas">
      <td>{{pizza.name}}</td>
    </tr>
  </tbody>
</table>
```

Architecting with component

- When using multiple components, they have to be **DECLARED** in a module
  - To be available anywhere in the module
  - When importing a module, it's declarations become available in the current module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { AppHeaderComponent } from ‘./app-header.component';
import { AppSidebarComponent } from ‘./app-sidebar.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, AppSidebarComponent, AppHeaderComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

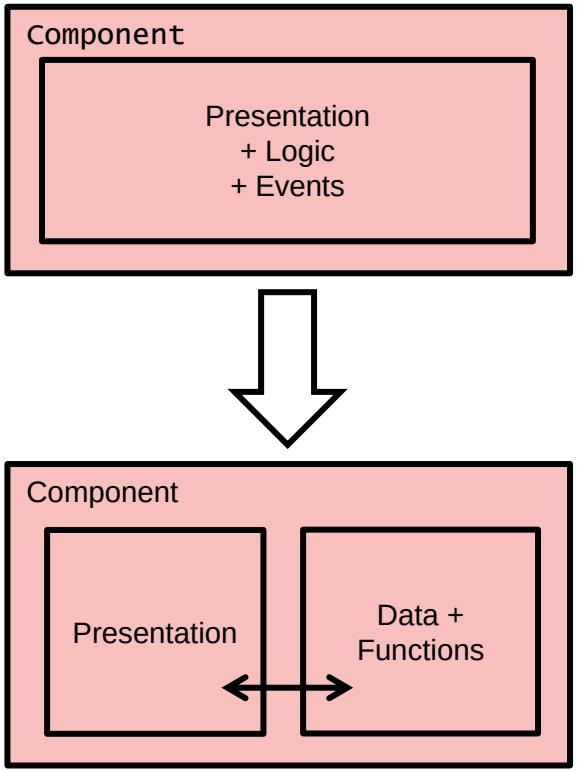
Data binding

WITHOUT Data Binding

- Load data properties into the DOM
- Synchronize data objects with DOM
- Casting from text to other types

WITH Data Binding

- Using **MVVM** pattern to separate view from data and logic
- Declarative synchronization between data and view



Model View ViewModel

MODEL

- An object model that represents the real state content
- Independent from presentation logic
  - Easy to test
  - Easy to re-use
- In Angular, the model typically comes from Services

Model View ViewModel

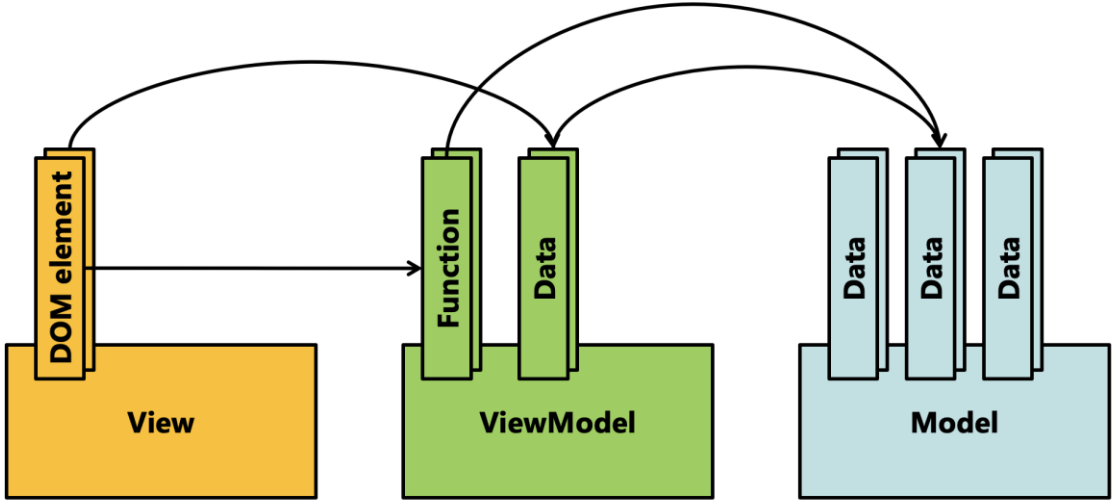
VIEW

- Refers to all elements displayed by the Browser such as buttons, inputs, selects, and other DOM Elements
- Pure HTML
- Relies heavily on data binding
  - To data
  - To functions

Model View ViewModel

ViewModel

- A ViewModel is an object that contains all (or some of) the **model** and functionality that a view binds to.
- In Angular, the component class serves as ViewModel for it's template (View)



One way binding (Component → View)

- One-way bindings update the view from the component
  - No side-effects are allowed
  - View cannot Component using this mechanism

- Types
  - Interpolation
  - Property binding
  - Attribute binding

One way binding (Component → View)

- **Interpolation**
  - Format : **`{{ property }}`**
  - Displays data from the component
- Template expression is often a model property

```
<p>{{user.firstName}}</p>
```

- Could be an expression as well
  - Insert result as string

```
<p>5+25={{5+25}}</p>
```

One way binding (Component → View)

- **Property binding**
  - Set a property of a view element
  - Format : **`[property]="expression"`**

```
<input type="button" [value]="orderText" />
```

- We can use interpolation for property binding

```
<input type="button" value="{{orderText}}" />
```

One way binding (Component → View)

- **Attribute binding**
  - Bind to attribute
  - Format : **`[attr.nameOfAttr]="expression"`**

```
<!-- Property Binding Fails -->
<tr><td colspan="{{1+1}}">Three-Four</td></tr>
<!-- Attribute Binding to the rescue! -->
<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>
```

View → Component

- Bindings to call the component from the view
  - Side-effects are allowed
  - ViewModel might be updated
  - Certain views will be re-rendered

- Types
  - Event handling
  - Two-way binding

View → Component

- **Event handling**
  - Respond to events
  - Format : **(event)="statement"**

```
<input (click)="add()" type="button" />
```

- $event : Object that contains information about the event

```
<input (keyup)="onKey($event)">

onKey(event: KeyboardEvent) {
  this.values.push((<HTMLInputElement>event.target).value);
}
```

View → Component

- **Two-way binding**
  - A combination between one-way binding with event handling
  - Typically used with **ngModel**
  - Format : **`[(ngModel])="property"`**

Sets value in view

Respond to change

```
<input [value]="current.name"
       (change)="current.name=$event">
```

simplified

```
<input [(ngModel)]="current.name">
```

Data binding (Example)

HTML

```
<div>
  <p>Welcome {{user.username}}</p>
  <input [(ngModel)]="user.username" />
  <input type="password" [(ngModel)]="user.password" />
  <input type="button" (click)="login()" value="login" />
</div>
```

TS

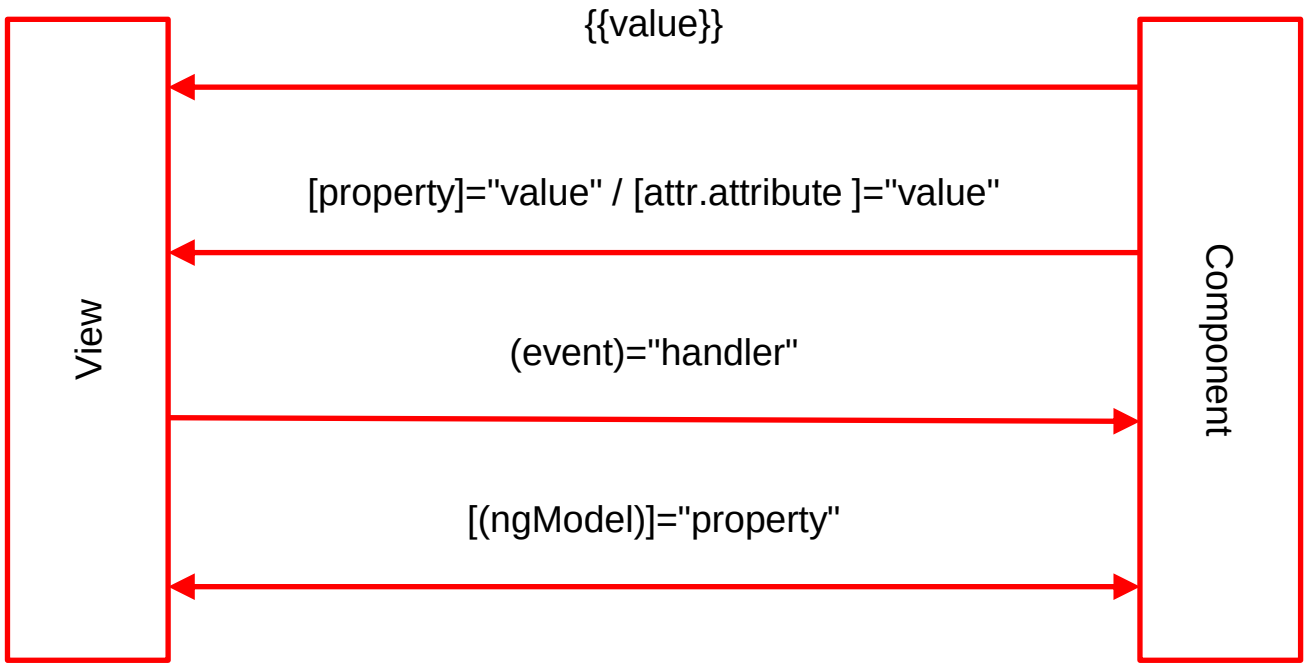
```
@Component(…)
export class LoginComponent {

  user: User

  constructor() {
  }

  login() {
    //
  }

}
```

One-Way binding

Two-Way binding

Event-Handling

Data binding (Summary)

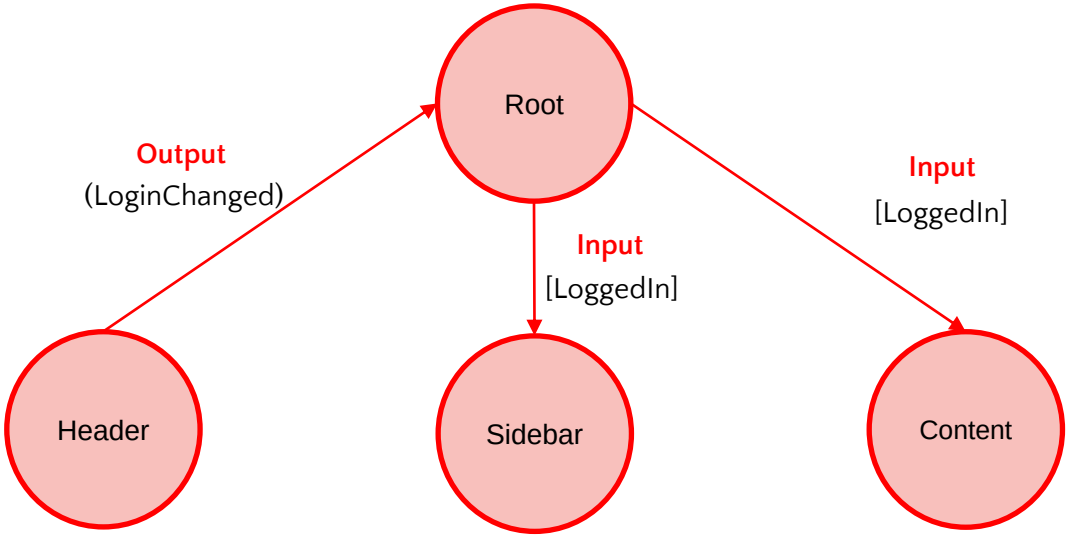


LAB 2

Use Data Binding

Component communication

- How do components communicate with each other ?



Inputs

- Parent to child communication
- Define inputs for the component using **@Input** decorator
- Use property binding to send data

```
export class ContentComponent {
  @Input()  loggedIn: Boolean;
}
```

```
<app-content [loggedIn]="isLoggedIn" ></app-content>
```

Outputs

- Child to parent communication
- Define outputs for the component using **@Output** decorator
- Use event handling using **EventEmitter** class

```
export class HeaderComponent {
        @Output() loginChanged = new EventEmitter<Boolean>();
}
```

```
<app-header  (loginChanged)="onLoginChanged($event)" ></app-header>
```

EventEmitter

- When a parent needs to respond to changes of a child, you can use an EventEmitter
  - Outputs properties are always EventEmitters

```
@Output() loginChanged = new EventEmitter<boolean>();
```

Type of $event

- EventEmitter<T> class
  - Use emit(value) to raise an event

Inputs, Outputs (Example)

```
@Component({
    selector: 'my-app',
    template: `
              <div>
                <h1>App</h1>
                <app-header (loginChanged)="onLoggedIn($event)"></app-header>
                <app-content [loggedIn]="isLoggedIn">   </app-content >
              </div>
      `,
  })
  export class AppComponent {
    isLoggedIn = false

    onLoggedIn(loggedIn: boolean): void {
      this.isLoggedIn = loggedIn
    }
  }
```

```
login() {
    ...
    this.loginChanged.emit(true)
}
```

Content projection

- Content projection is to provide an HTML content to use inside another component
- Use **ng-content** tag placeholder to define where the content should be projected

```
<!-- app.component.html -->
<app-card>
    <div>
      <h1> This is a title </h1>
    </div>
</app-card>
```

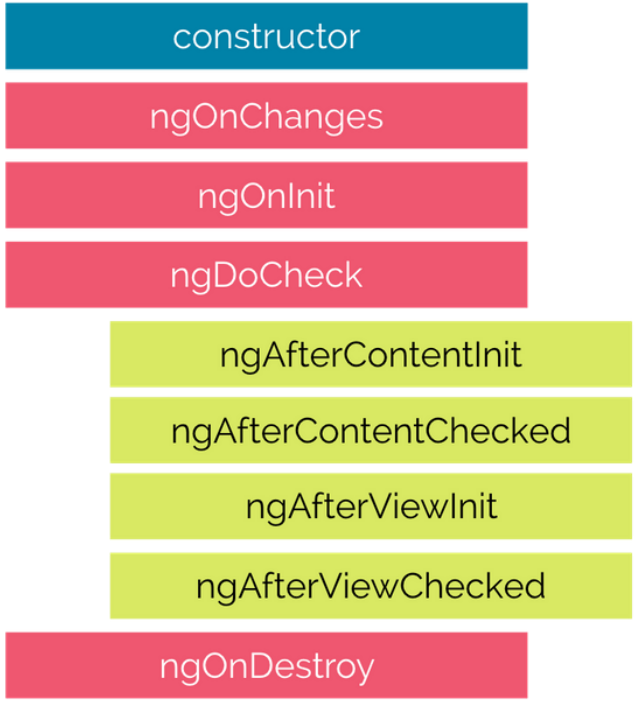
Projection

```
<!-- app-card.component.html -->
<div>
    <ng-content> </ng-content>
    <div>
        some stuff here
    </div>
    <buttton>
        Submit
    </buttton>
</div>
```

Component lifecycle

Every component goes through a life cycle

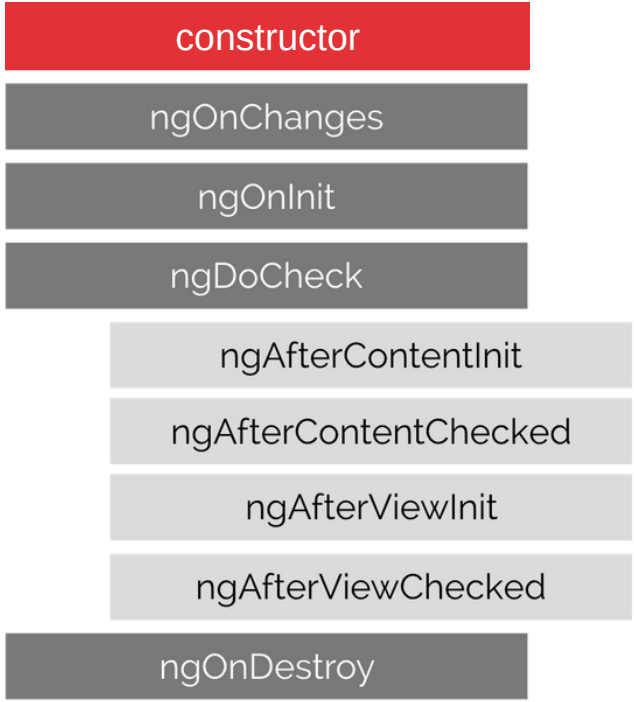
- Managed by Angular
- Provides hooks to inject code at various times in the lifecycle



Component lifecycle

constructor

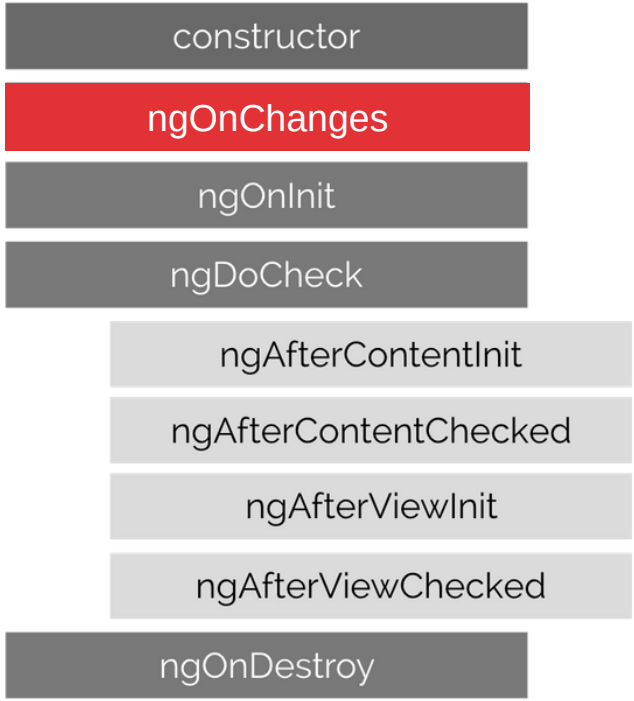
- Parent constructor is called before child constructors
  - Child can relies on parent data
- Best not to call async code here



Component lifecycle

ngOnChanges

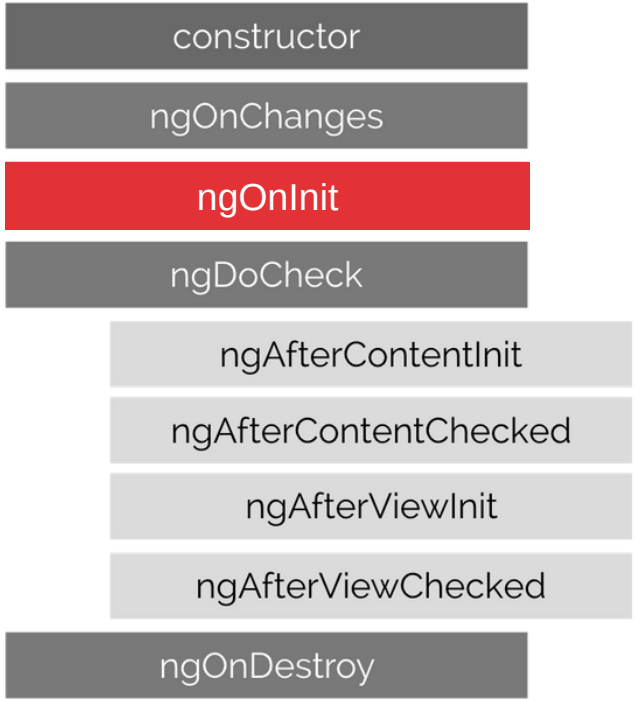
- Called when input binding value changes
- Provides info about:
  - Previous value
  - Current value
  - Is first change



Component lifecycle

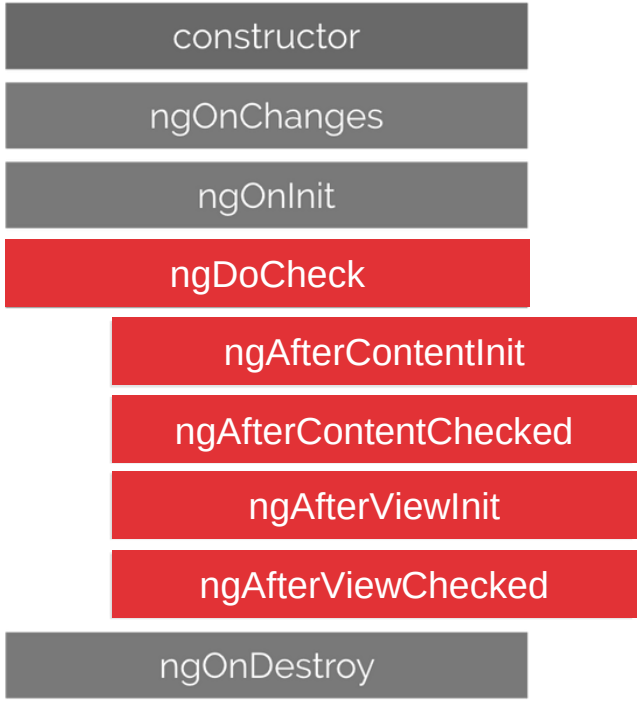
ngOnInit

- Called before view initialization
- Initialize state for component
- Child component initialize first
- Can deal with async code



Component lifecycle

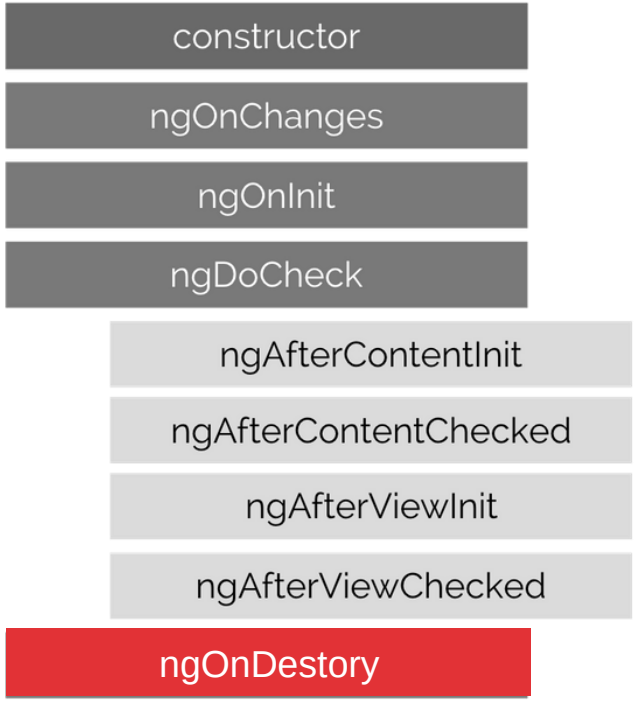
- **ngDoCheck** : Developer's custom change detection
- **ngAfterContentInit**: Runs after content projection
- **ngAfterContentChecked**: Runs after every ngDoCheck
- **ngAfterViewInit**: Called after view initialization
- **ngAfterViewChecked**: Called after every ngAfterContentChecked



Component lifecycle

ngOnDestroy

- Called before destroying the component
- Clean up component data



Implement a hook (Example)

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

@Component({...})
  export class AppComponent implements OnInit {

    constructor() {

    }

    ngOnInit(): void {

      }
}
```

LAB 3

Use multiple components