Angular

Components life cycle

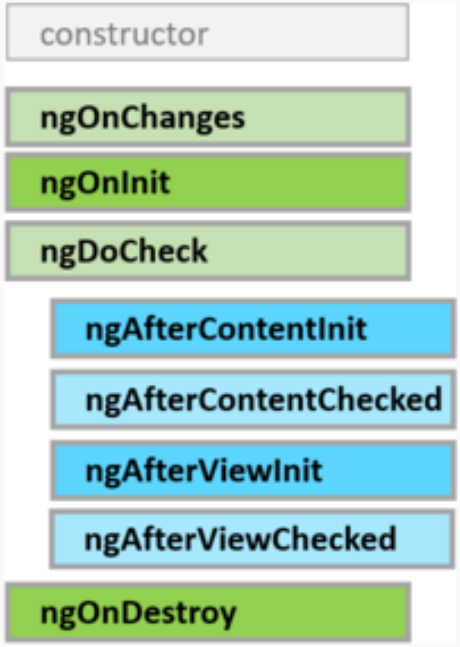
&

Components interaction

ITI – Assiut Branch
Eng. Hany Saad

Component lifecycle

Component Lifecycle



Lifecycle Hooks

A component has a lifecycle managed by Angular.

Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.

https://angular.io/guide/lifecycle-hooks

Component lifecycle (Cont.)

Component Lifecycle (Cont.)

`ngOnChanges()`	Respond when Angular (re)sets data-bound input properties. The method receives a `SimpleChanges` object of current and previous property values. Called before `ngOnInit()` and whenever one or more data-bound input properties change.
`ngOnInit()`	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called *once*, after the *first* `ngOnChanges()`.
`ngDoCheck()`	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after `ngOnChanges()` and `ngOnInit()`.
`ngAfterContentInit()`	Respond after Angular projects external content into the component's view. Called *once* after the first `ngDoCheck()`. *A component-only hook.*

Component lifecycle (Cont.)

Component Lifecycle (Cont.)

`ngAfterContentChecked()`	Respond after Angular checks the content projected into the component. Called after the `ngAfterContentInit()` and every subsequent `ngDoCheck()`. *A component-only hook.*
`ngAfterViewInit()`	Respond after Angular initializes the component's views and child views. Called *once* after the first `ngAfterContentChecked()`. *A component-only hook.*
`ngAfterViewChecked()`	Respond after Angular checks the component's views and child views. Called after the `ngAfterViewInit` and every subsequent `ngAfterContentChecked()`. *A component-only hook.*
`ngOnDestroy()`	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called *just before* Angular destroys the directive/component.

Component lifecycle (Cont.)

ngOnChanges:

- *It's called before* **ngOnInit** *and whenever one or more data-bound input properties change.*

- *You can use ngOnChanges() which is* ***called every time an @Input() is updated*** *by change detection.*

- *You can check within ngOnChanges whether all input values are already available and then execute your code.*

```
import { OnChanges } from '@angular/core';

  @Component({selector: 'my-cmp', template: `...`})
  class MyComponent implements OnChanges {
    @Input()
    prop: number;

    ngOnChanges(changes: SimpleChanges) {
      // changes.prop contains the old and the new value...
    }
  }
```

Component lifecycle (Cont.)

When to use constructor and use onInit?

- Normally we use **constructor** to define/initialize some variables, and for proper initialization of fields in the class, also it's used for Dependency Injection in Angular.
- when we have tasks related to Angular's bindings we move to Angular's **ngOnInit** life cycle hook, ngOnInit() is better place to "start" - it's where/when components' bindings are resolved.
- **ngOnInit** is called just after the constructor call. We can also do the same work in the constructor but its preferable to use **ngOnInit** to start Angular's binding.

in order to use `ngOnInit` we have to import this hook from the core library:

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

Then we implement this interface with exported class (this is not compulsory to implement this interface but generally we did).

Example of using both:

```
export class App implements OnInit{
  constructor(){
     //called first time before the ngOnInit()
  }

  ngOnInit(){
     //called after the constructor and called  after the first ngOnChanges()
  }
}
```

Component lifecycle (Cont.)

ngOnDestroy:

- It is called for cleanup logic when a component, directive, pipe or service is destroyed.

- ngOnDestroy() is called just before component/directive is about to be destroyed by Angular. It can be used for following purposes.
    1. Stop interval timers.
    2. Unsubscribe Observables.
    3. Detach event handlers.
    4. Free resources that will not be garbage collected automatically.
    5. Unregister all callbacks.

```
import { OnDestroy } from '@angular/core';
@Component({selector: 'my-cmp', template: `...`})
class MyComponent implements OnDestroy {
  ngOnDestroy() {
    // ...
  }
}
```

Components interaction

Sharing Data Between Angular Components:

Parent to Child: Sharing Data via Input()

This is probably the most common and straightforward method of sharing data. It works by using the `Input()` decorator to allow data to be passed via the template.

Child to Parent: Sharing Data via Output() and EventEmitter

Another way to share data is to emit data from the child, which can be listed to by the parent. This approach is ideal when you want to share data changes that occur on things like button clicks, form entires, and other user events.

Child to Parent: Sharing Data via ViewChild()

ViewChild allows a one component to be injected into another, giving the parent access to its attributes and functions. One caveat, however, is that child won't be available until after the view has been initialized. This means we need to implement the AfterViewInit lifecycle hook to receive the data from the child.

Unrelated Components: Sharing Data with a Service

When passing data between components that lack a direct connection, such as siblings, grandchildren, etc, you should you a shared service. When you have data that should aways been in sync, I find the **RxJS BehaviorSubject** very useful in this situation.

You can also use a regular RxJS Subject for sharing data via the service, but here's why I prefer a BehaviorSubject.

More: https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods

Remember !

Decorator

A decorator is a function that is invoked with @ prefix and followed by class, method, property.

Ex:

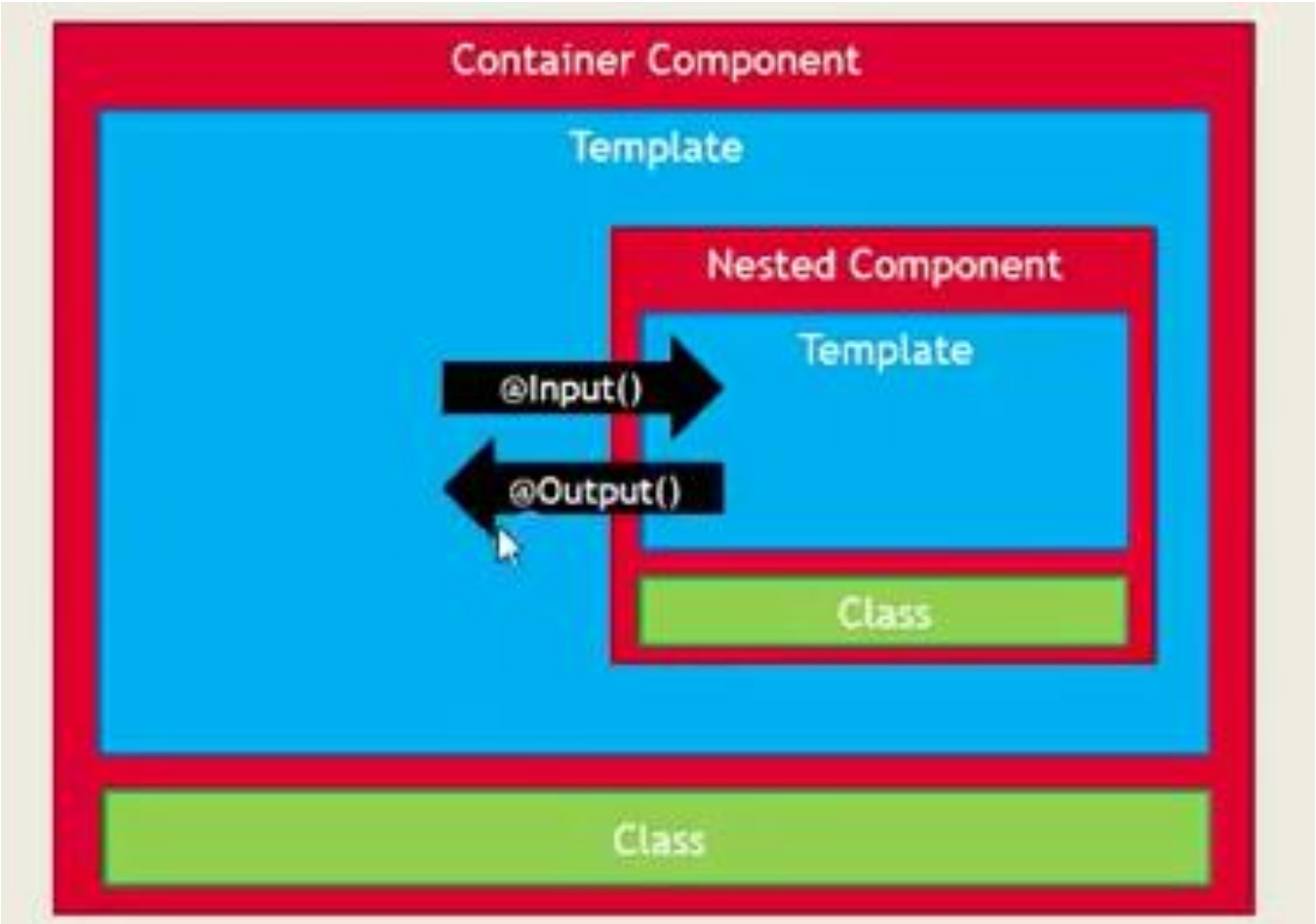
@Input()

description: string;

Nested components interaction

Parent to Child: Sharing Data via Input()

This is probably the most common and straightforward method of sharing data. It works by using the `Input()` decorator to allow data to be passed via the template.



Nested components interaction (Cont.)

On Child component:

```
import {Input} from '@angular/core';
@Input() selectedCatID:Number;
```

- Then you can use the selectedCatID as needed and to change the data in view according to it.

On Parent component:

- In HTML bind to the property that will be sent to the child:

```
<select [(ngModel)]="selectedCategoryID">
<option [value]="cat.ID" *ngFor='let cat of
catList'>{{cat.Name}}</option>
</select>
```

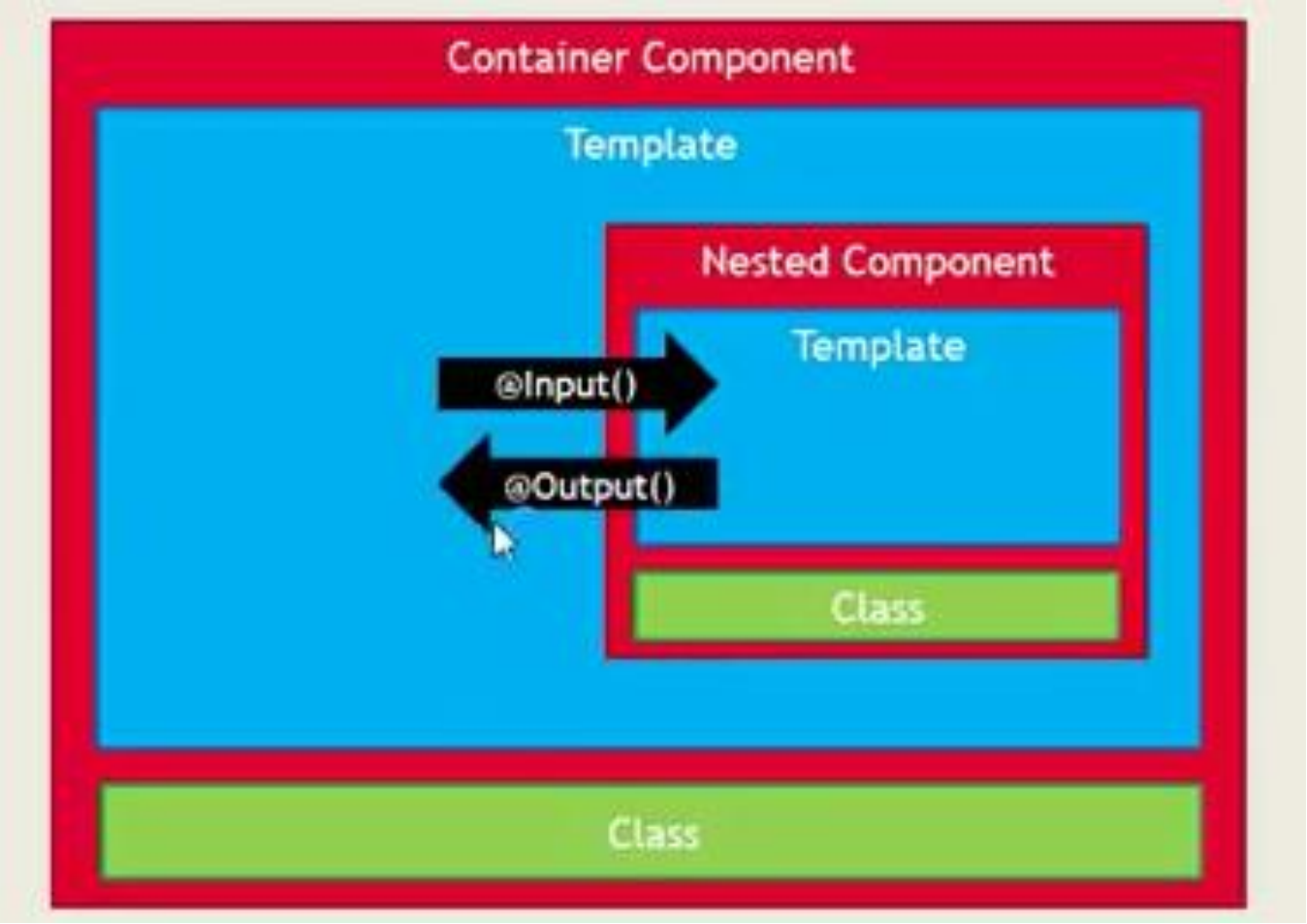
- In child component Directive in the parent:

```
<app-cart-details [selectedCatID]='selectedCategoryID'></app-cart-
details>
```

Nested components interaction (Cont.)

Child to Parent: Sharing Data via Output() and EventEmitter

Another way to share data is to emit data from the child, which can be listed to by the parent. This approach is ideal when you want to share data changes that occur on things like button clicks, form entires, and other user events.



Nested components interaction (Cont.)

On Child component:

- Create Event, and use @Output:

```
@Output() totalPriceChanged: EventEmitter<number> =new EventEmitter<number>();
```

- Emit (Fire the event, when total price change (Ex.when user clicks “Buy” button):

```
this.totalPriceChanged.emit(this.orderTotalPrice);
```

On Parent component:

- In child component Directive in the parent:

```
<app-product-list [receivedCatID]="selectedCatID"
      (totalPriceChanged)="onTotalPriceChanged($event)">
 </app-product-list>
```

Nested components interaction (Cont.)

Using @viewchild() to access Dom elements:

We can access native DOM elements that have a template reference variable. Let's say we have this in our template with the `someInput` reference variable:

```
<input #someInput placeholder="Your favorite pizza toping">
```

We can access the input itself with **ViewChild** like this:

```
import { Component,
         ViewChild,
         AfterViewInit,
         ElementRef } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements AfterViewInit {
  @ViewChild('someInput') someInput: ElementRef;

  ngAfterViewInit() {
    this.someInput.nativeElement.value = "Anchovies! 🍕🍕";
  }
}
```

And the value of our input will be set to `Anchovies! 🍕🍕` when **ngAfterViewInit** fires.

Nested components interaction (Cont.)

Using @viewchild() to access child component:

Child to Parent: Sharing Data via ViewChild()

ViewChild allows a one component to be injected into another, giving the parent access to its attributes and functions. One caveat, however, is that child won't be available until after the view has been initialized. This means we need to implement the AfterViewInit lifecycle hook to receive the data from the child.

It's just as easy to access a child component and call methods or access instance variables that are available on the child. Let's say we have a child component with a `whoAmI` method like this:

```
whoAmI() {
  return '👶 I am a child!!';
}
```

Nested components interaction (Cont.)

Using @viewchild() to access child component (Cont.):

app.component.ts

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

import { ChildComponent } from './child.component';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements AfterViewInit {
  @ViewChild(ChildComponent) child: ChildComponent;

  ngAfterViewInit() {
    console.log(this.child.whoAmI()); // 👶 I am a child!
  }
}
```

Components interaction (Cont.)

Unrelated Components: Sharing Data with a Service

When passing data between components that lack a direct connection, such as siblings, grandchildren, etc, you should you a shared service. When you have data that should aways been in sync, I find the **RxJS BehaviorSubject** very useful in this situation.

You can also use a regular RxJS Subject for sharing data via the service, but here's why I prefer a BehaviorSubject.

- It will always return the current value on subscription – there is no need to call `onnext`
- It has a `getValue()` function to extract the last value as raw data.
- It ensures that the component always receives the most recent data.

In the service, we create a private BehaviorSubject that will hold the current value of the message. We define a currentMessage variable handle this data stream as an observable that will be used by the components. Lastly, we create function that calls next on the BehaviorSubject to change its value.

The parent, child, and sibling components all receive the same treatment. We inject the DataService in the constructor, then subscribe to the currentMessage observable and set its value equal to the message variable.

Example: https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/#Unrelated-Components-Sharing-Data-with-a-Service

Components interaction (Cont.)

More about component interaction:

- More examples for @Viewchild()
  https://alligator.io/angular/viewchild-access-component/
  https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/
  https://ng2.codecraft.tv/components/viewchildren-and-contentchildren
  https://www.concretepage.com/angular-2/angular-2-viewchild-example

- Component Interaction:
  https://angular.io/guide/component-interaction
- https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/
-
- Components inheritance:
  https://scotch.io/tutorials/component-inheritance-in-angular-2

- Parent and children communicate via a service:
  https://angular.io/guide/component-interaction#parent-and-children-communicate-via-a-service

- Redux with Angular:
  http://blog.ng-book.com/introduction-to-redux-with-typescript-and-angular-2
  https://www.pluralsight.com/guides/front-end-javascript/building-a-redux-application-with-angular-2-part-1
  http://onehungrymind.com/build-better-angular-2-application-redux-ngrx
  http://blog.rangle.io/getting-started-with-redux-and-angular-2

Safe Navigation Operator & non-null assertion operator

The Safe Navigation Operator (optional chaining operator) **?**. permits reading the value of a property located deep within a chain of connected objects without having to expressly validate that each reference in the chain is valid.
The ?. operator functions similarly to the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.

```
obj?.prop // Accessing object's property
obj?.[expr] // Optional chaining with expressions
arr?.[index] // Array item access with optional chaining
func?.(args) // Optional chaining with function calls
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining

! is non-null assertion operator (post-fix expression) – it's saying to type checker that **you're** sure that a is not null or undefined.

https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator

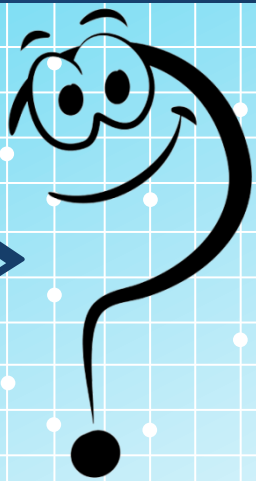
More details: https://docs.angular.lat/guide/template-expression-operators

Using Non null assertion operator in object declaration


```
// clientNameInputObj:ElementRef | null = null;
// clientNameInputObj:ElementRef= {} as ElementRef;
// clientNameInputObj?:ElementRef;

//Non-null assertion operator
clientNameInputObj!:ElementRef;
```

<SCRIPT >



</SCRIPT>

```
<script>document.writeln("Thank
You!")</script>
```