LIFE CYCLE HOOKS

Component Initialization

Angular Components and Directives Lifecycle: Notes with Definitions and Examples

1. Component and Directive Creation

- **Definition**: When Angular encounters a component or directive selector in the template, it creates (instantiates) a new instance of the corresponding class.
- **Example**:

```
@Component({
  selector: 'app-demo',
  template: '<p>Demo Component</p>'
})
export class DemoComponent { }
```

- When `<app-demo>` is encountered in the DOM, Angular creates an instance of `DemoComponent`.

2. Constructor Call

- **Definition**: When a component or directive is instantiated, its constructor is called to initialize its properties.
- **Key Point**: If a constructor is not explicitly defined, a default parameterless constructor is provided by TypeScript.
- **Example**:

```
export class DemoComponent {
  constructor() {
    console.log('DemoComponent Constructor called');
  }
}
```

- When `<app-demo>` is encountered, "DemoComponent Constructor called" will be logged.

```
})
export class InitializationComponent {

  title = "Demo-Component";
  @Input() message = "Hello";

  // Component Initialization

  constructor() {
    console.log("Components constructor is called");
    console.log(this.title);
    console.log(this.message);
  }
```

app.component.html M X

LifeCycle_Hooks > src > app > app.component.html > ...

Go to component

```
1  <app-initialization [message]="'HelloWorld'"></app-initialization>
```

```
})
export class InitializationComponent {

  title = "Demo-Component";
  @Input() message = "Hello";

  // Component Initialization

  constructor() {
    console.log("Components constructor is called");
    console.log(this.title);
    console.log(this.message);
  }
```

```
})
export class InitializationComponent {

  title = "Demo-Component";
  @Input() message = "Hello";

  // Component Initialization

  constructor() {
    console.log("Components constructor is called");
    console.log(this.title);
    console.log(this.message);
```

```
App constructor is called
Components constructor is called
Demo-Component
Hello
```

3. **Multiple Instances of Components**

- **Definition**: If the selector for a component is used multiple times in a template, Angular will create a new instance for each occurrence.
- **Example**: Using `<app-demo>` multiple times creates multiple instances.

html Copy co

```
<app-demo></app-demo>
<app-demo></app-demo>
<app-demo></app-demo>
```

- The constructor will be called three times, as each `<app-demo>` creates a new instance of `DemoComponent`.

4. **Order of Component Initialization**

- **Definition**: Parent components are instantiated before their child components.
- **Example**:

```
@Component({ selector: 'app-root' })
export class AppComponent {
  constructor() { console.log('AppComponent Constructor called'); }
}

@Component({ selector: 'app-demo' })
export class DemoComponent {
  constructor() { console.log('DemoComponent Constructor called'); }
}
```

- The `AppComponent` constructor is called first, followed by the `DemoComponent` constructor.

5. Input Properties and Initialization

- **Definition**: Input properties are not available when the constructor is called. They are initialized later.
- **Example**:

```
@Component({
  selector: 'app-demo',
  template: '<p>{{ message }}</p>'
})
export class DemoComponent {
  @Input() message: string = 'Initial Value';
  constructor() {
    console.log(this.message); // Logs 'Initial Value', not 'Hello World'
  }
}
```

```
// In parent component
<app-demo [message]="'Hello World'"></app-demo>
```

- The constructor logs the initial value, not the value passed by the parent.

6. Content Projection (ng-content)

- **Definition**: Content projection allows a parent component to project content into a child component using `<ng-content>`.
- **Example**:

```
@Component({
  selector: 'app-demo',
  template: '<ng-content></ng-content>'
})
export class DemoComponent { }

// In parent component
<app-demo><p>Projected Content</p></app-demo>
```

- `<ng-content>` is replaced by "Projected Content" from the parent component.

7. Lifecycle of a Component

- **Definition**: Angular components go through different phases in their lifecycle: creation, rendering, data binding, and destruction.
- **Key Phase**: The root component is created first, followed by child components in a hierarchical order.

- **Definition**: Lifecycle hooks are special methods that Angular calls during specific phases of a component's lifecycle.
- **Common Hooks**:
  - `ngOnChanges()`: Called when input properties change.
  - `ngOnInit()`: Called once after the first change detection.
  - `ngDoCheck()`: Detects changes beyond the default detection.
  - `ngAfterContentInit()`: After content (ng-content) is projected into the component.

- `ngAfterContentChecked()` : After content projection is checked.
- `ngAfterViewInit()` : After component's view (and child views) has been initialized.
- `ngAfterViewChecked()` : After the component's view (and child views) has been checked.
- `ngOnDestroy()` : Before the component is destroyed.

9. **Destruction of a Component**

- **Definition**: When a component is removed from the DOM, its instance is destroyed and the `ngOnDestroy` hook is called.
- **Example**:

```
@Component({ selector: 'app-demo' })
export class DemoComponent {
  ngOnDestroy() {
    console.log('DemoComponent destroyed');
  }
}
```

This summary covers the key points of component and directive lifecycle creation, including constructor calls, input properties, and lifecycle hooks.

When a constructor is called, by that time, none of its input properties are updated and available to use.

```
<app-demo></app-demo>
```

```
@Component({
    selector: ‘app-demo’,
    template: ‘<p>This is a paragraph</p>’
})
export class DemoComponent{
    constructor(){
        //Some code
    }

    title: string = ‘App Demo’;
    show: boolean = false;
    @input() myVar: string;
}
```

```
export class DemoComponent {
  title: string = 'Demo Component';
  @Input() message: string = 'Hello';

  constructor(){
    console.log('Demo component constructor called');
    console.log(this.title);
    console.log(this.message);
  }
}
```

When a constructor is called, by that time, none of its input properties are updated and available to use.

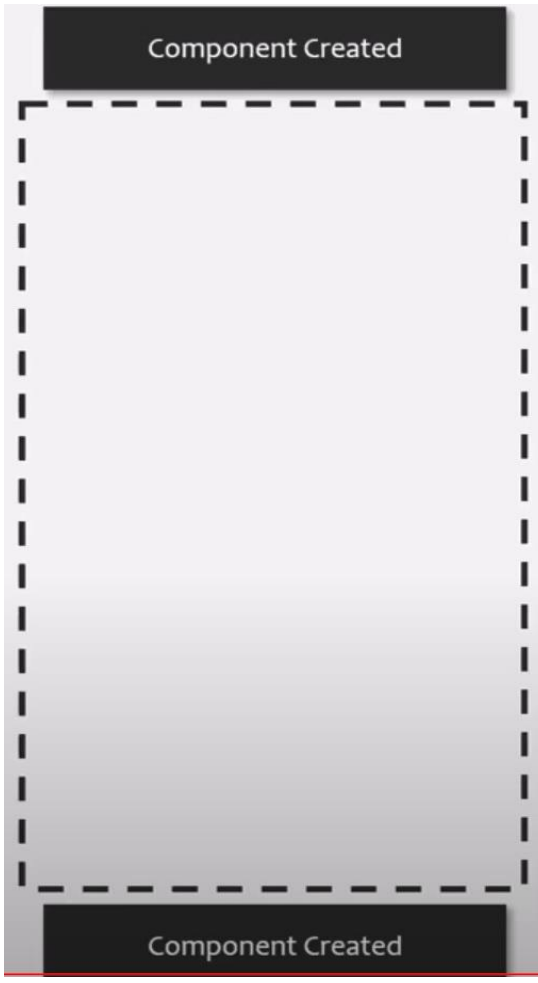
When a constructor is called, by that time, the child components of that component are not yet constructed.

Projected contents are also not available by the time the constructor of a component is called.

Once the component is removed from the DOM, we can say component is destroyed.

```
<app-demo>
    <p>This content will be projected.<p>
</app-demo>
```

```
@Component({
    selector: ‘app-demo’,
    template: ’<h2>Content Projection</h2>
                <ng-content></ng-content>
                <p>This is a paragraph</p>’

})
export class DemoComponent{
    constructor(){
        //Some code
    }

    title: string = ‘App Demo’;
    show: boolean = false;
    @input() myVar: string;
```

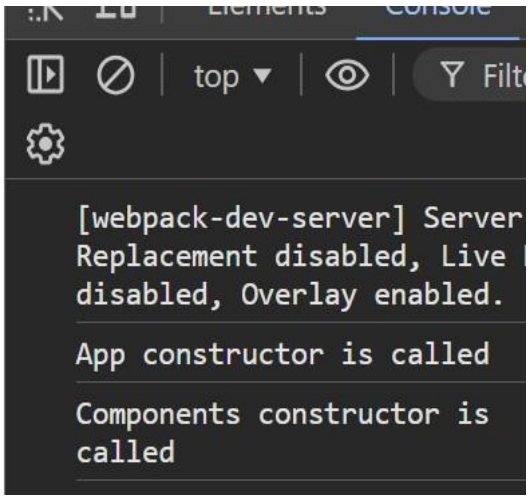


- When the Angular application starts, it first creates and renders the root component.
- Then it creates and renders its children and their children. In this way, it forms a tree of components.
- Once Angular loads the component, it starts the process of rendering the view. To do that, it needs to check the input properties, evaluate the data bindings & expressions, render the projected content etc.
- Angular then also removes the component from the DOM when it is no longer needed.
- And Angular lets us know when these events happen, using Angular lifecycle hooks.

The Angular life cycle hooks are the methods that angular invokes on a directive or a component, as it creates, changes and destroys them.

initialization works!

When a component is rendered It's constructor get's called if there is a constructor defined Else it will create an instance to that component with parameter less constructor, Only After Parent component is rendered in the dom child components gets rendered in Dom & in this phace component values is not initialized



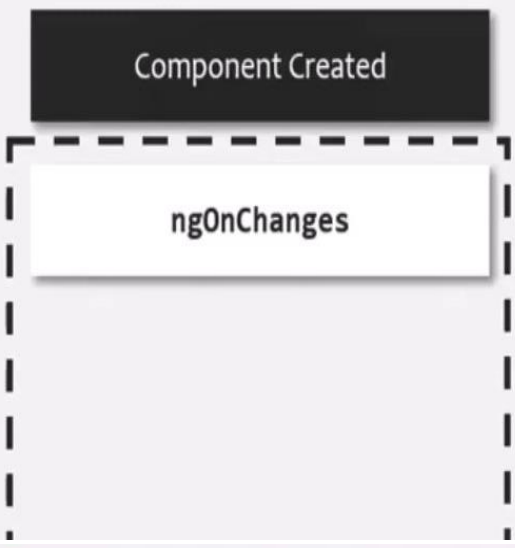
ngOnChanges Lifecycle Hook

The Angular life cycle hooks are the methods that angular invokes on a directive or a component, as it creates, changes and destroys them.

Change detection in Angular is a mechanism by which, Angular keeps the view template in sync with the component class.

```
<div>Hello {{ name }}</div>
```

- Whenever the @input property of a component changes
- Whenever a DOM event happens. E.g. Click or Change
- Whenever a timer events happens using `setTimeOut()` / `setInterval()`.
- Whenever an HTTP request is made.



The ngOnChanges hook get executed at the start, when a new component is created and its input bound properties are updated.

The ngOnChanges hook also gets executes everytime the input bound properties of the component changes

The ngOnChanges hook get executed at the start, when a new component is created and its input bound properties are updated.

The ngOnChanges hook also gets executes everytime the input bound properties of the component changes

The ngOnChanges hook is not raised if the change detection cycle does not find any changes in the input propertie's value.

Notes: Angular `ngOnChanges` Lifecycle Hook

1. **Definition:**
   - `ngOnChanges` is an Angular lifecycle hook that gets triggered when Angular detects changes to the input properties of a component or directive. It allows developers to respond when a property bound to the component changes.

2. **Change Detection Cycle:**
   - Angular keeps the component class and view in sync through the change detection cycle.
   - Events that trigger the cycle include:
     - Input property updates.
     - DOM events (e.g., `click`, `change`).
     - Timer events (`setTimeout`, `setInterval`).
     - HTTP requests.

3. **Component Lifecycle:**
   - The lifecycle begins when Angular instantiates a component (calls its constructor) and ends when the component is destroyed.
   - The constructor initializes the class but does **not** have access to the input properties yet.

4. `ngOnChanges` **Hook:**
   - It is called when:
     1. A component is created, and its input property is first initialized.
     2. An input property of the component changes.
   - The method is triggered when the change detection cycle detects differences between the current and previous values of an input property.

5. **Implementing** `ngOnChanges`**:**
   - To use `ngOnChanges`, a component can implement the `OnChanges` interface, although it is not mandatory.
   - `ngOnChanges` takes a `SimpleChanges` parameter, which contains information about the previous and current values of the input properties.

```typescript
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core

@Component({
  selector: 'app-demo',
  template: `<p>{{ message }}</p>`
})
```

```typescript
export class DemoComponent implements OnChanges {
  @Input() message: string;

  ngOnChanges(changes: SimpleChanges) {
    console.log('ngOnChanges called');
    console.log('Changes:', changes);
  }
```

6. **Example of `ngOnChanges`:**

   - Suppose a parent component passes a value to a child component:

```html
<app-demo [message]="parentMessage"></app-demo>
```

- In this case, `ngOnChanges` will be triggered when:

  1. The child component is first rendered, and the `message` input property is assigned.
  2. Any time the `parentMessage` value changes.

7. **Key Observations:**

   - **First Execution:** When the component is first rendered, `ngOnChanges` will log the input properties.
   - **Subsequent Executions:** If the input property changes, `ngOnChanges` logs the new values.
   - **No Change Detection:** `ngOnChanges` will not be triggered if the input property's current and previous values are the same.

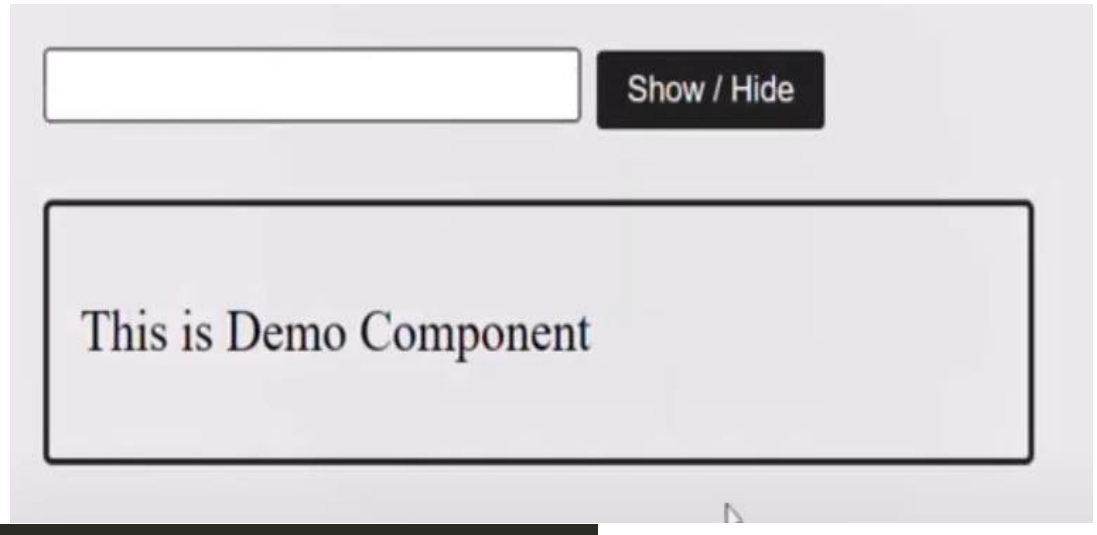
8. Practical Use Cases:

- Use `ngOnChanges` to react to input property updates, such as recalculating derived data, updating visual elements, or triggering specific behavior when an input value changes.

Example Output in Console:

text

```
ngOnChanges called
Changes: {
  message: {
    currentValue: 'Hello World',
    previousValue: undefined,
    firstChange: true
  }
}
```




```
})
export class DemoComponent {
  title: string = 'Demo Component';
  @Input() message: string;

  constructor(){
    console.log('Demo component constructor called');
    console.log(this.title);
    console.log(this.message);
  }
```

demo.component.html | demo.component.ts | app.component.html

src > app > app.component.html > div.container > app-demo

Go to component

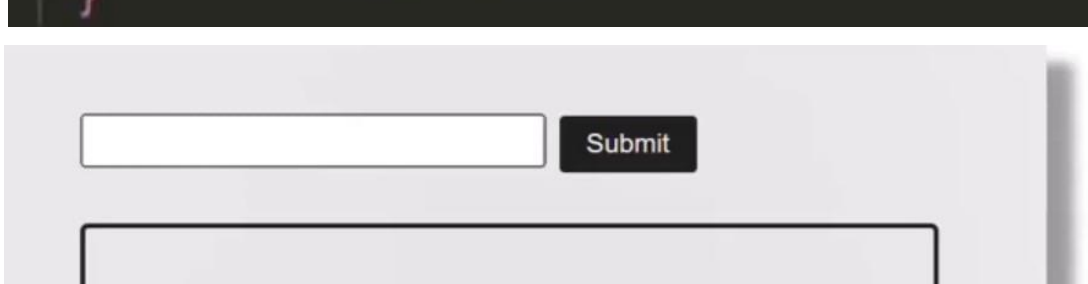
```
1
2   <div class="container">
3       <input type="text">
4       <button>Show / Hide</button>
5       <app-demo [message]="'Hello, World'"></app-demo>
6   </div>
7
```

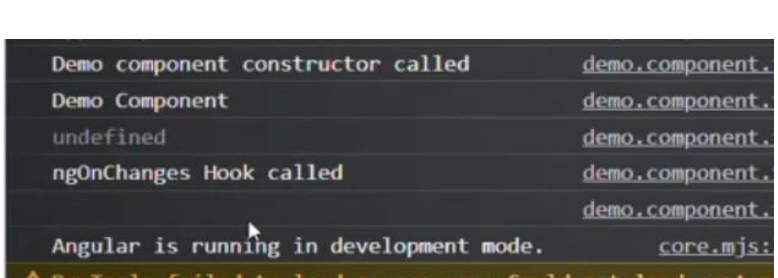
```ts
export class AppComponent {
  title = 'angular-lifecycle-hook';
  inputVal: string = '';
  constructor(){
    console.log('App Component constructor called!');
  }

  onBtnClicked(inputEl: HTMLInputElement){
    this.inputVal = inputEl.value;
  }
}
```

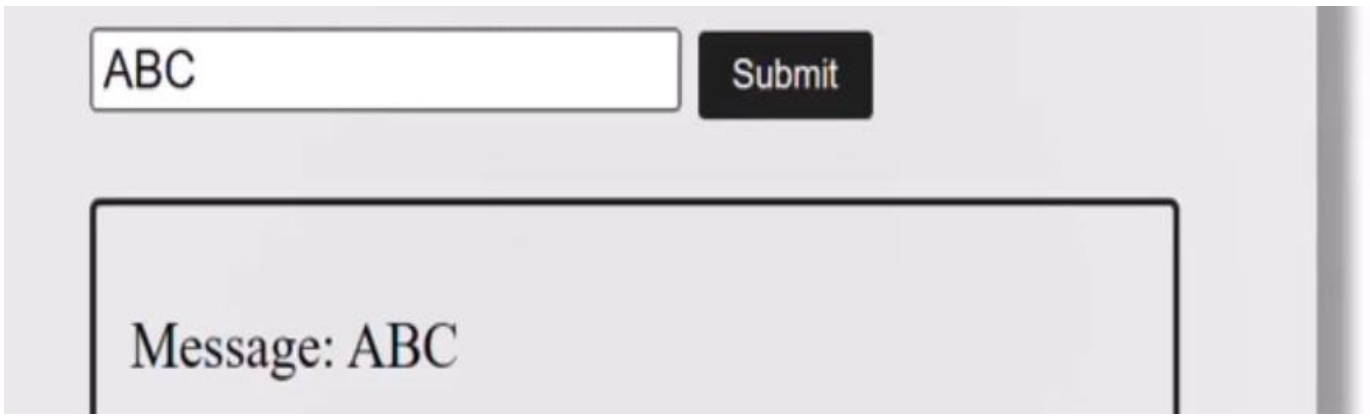
```html
<div class="container">
    <input type="text" #inputEl>
    <button (click)="onBtnClicked(inputEl)">Submit</button>
    <app-demo [message]="inputVal"></app-demo>
</div>
```

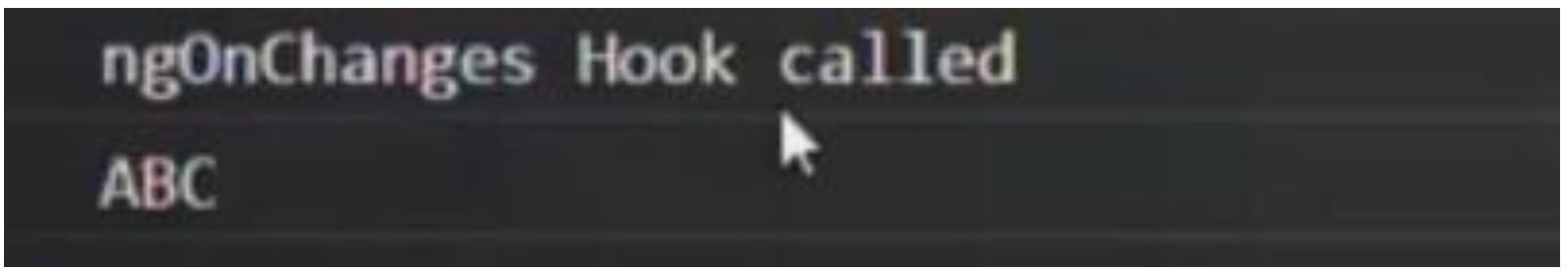
```ts
export class DemoComponent implements OnChanges{
  title: string = 'Demo Component';
  @Input() message: string;

  constructor(){
    console.log('Demo component constructor called');
    console.log(this.title);
    console.log(this.message);
  }

  ngOnChanges(){
    console.log('ngOnChanges Hook called');
    console.log(this.message);
  }
```

```ts
ngOnChanges(changes: SimpleChanges){
  console.log('ngOnChanges Hook called');
  console.log(this.message);
}
```

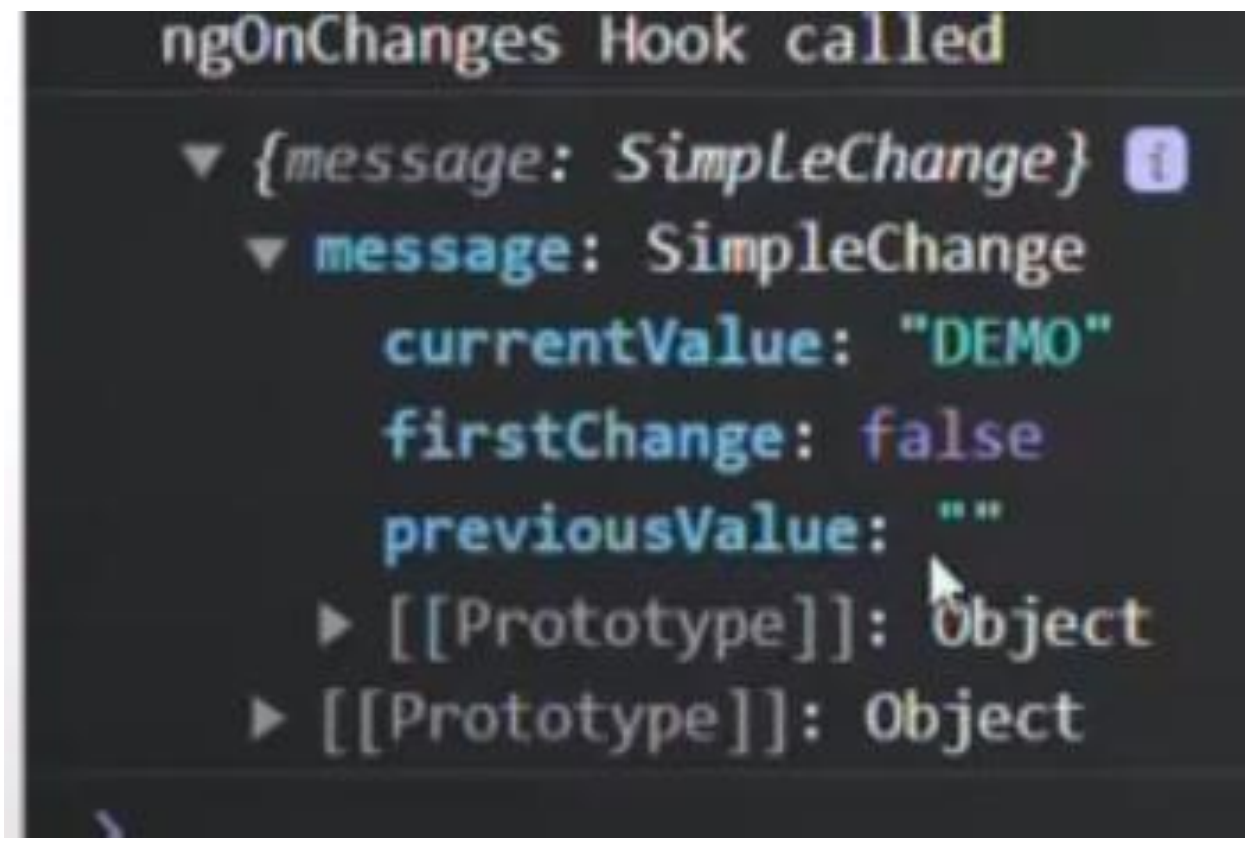




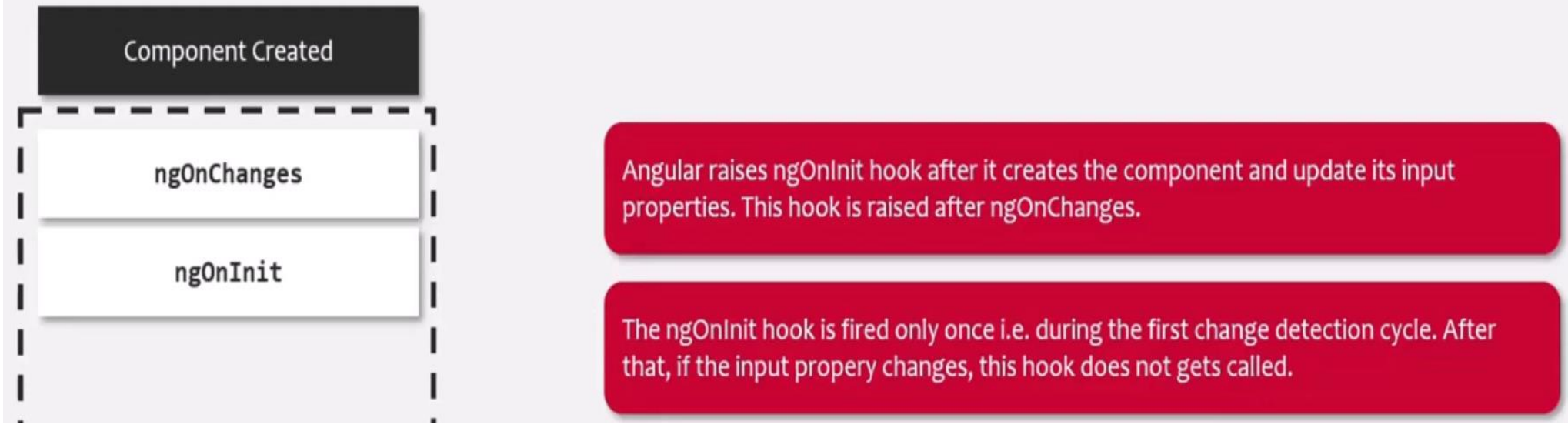

ABC
Submit
Message: ABC


ngOnChanges Hook called
ABC


DEMO
Submit
Message: DEMO


ngOnChanges Hook called
{message: SimpleChange}
message: SimpleChange
currentValue: "DEMO"
firstChange: false
previousValue: ""
[[Prototype]]: Object
[[Prototype]]: Object

ngOnInit Lifecycle Hook



Notes on `ngOnInit` Lifecycle Hook in Angular

Definition:

`ngOnInit` is a lifecycle hook in Angular that is called after Angular initializes the component's input properties. It occurs after the `ngOnChanges` hook but is called only **once** during the first change detection cycle.

Key Points:

1. **Called After `ngOnChanges`:** `ngOnInit` is triggered after Angular calls the `ngOnChanges` lifecycle hook.
2. **Called Once:** It is called only once, after the first change detection, and does not get called again if the input properties change later.
3. **Not Available for Child Components:** Child components, projected content, or elements decorated with `@ViewChild`, `@ViewChildren`, `@ContentChild`, or `@ContentChildren` are **not available** at the time `ngOnInit` is called.

4. **Initialization Logic:** `ngOnInit` is a good place to write any **initialization logic** for the component as it runs after input properties are set.

Example:

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

@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html'
})
export class DemoComponent implements OnInit {
  @Input() inputVal: string;  // Input property

  constructor() {
    console.log('Constructor called');
  }

  ngOnInit() {
    console.log('ngOnInit called');
  }

  ngOnChanges() {
    console.log('ngOnChanges called');
  }
}
```

Explanation:

- **Constructor Called First:** The constructor is invoked first when the component is created.
- `ngOnChanges` **Called Next:** If any input properties change, `ngOnChanges` is called.
- `ngOnInit` **After** `ngOnChanges`: Once Angular updates the input properties, it calls `ngOnInit` during the first change detection cycle. If the input properties change again, `ngOnChanges` will be called, but not `ngOnInit`.

Scenario:

- **First Load:** When the component is loaded, both `ngOnChanges` and `ngOnInit` are called. After that, if input properties change, only `ngOnChanges` is called, not `ngOnInit`.

Accessing Child Elements:

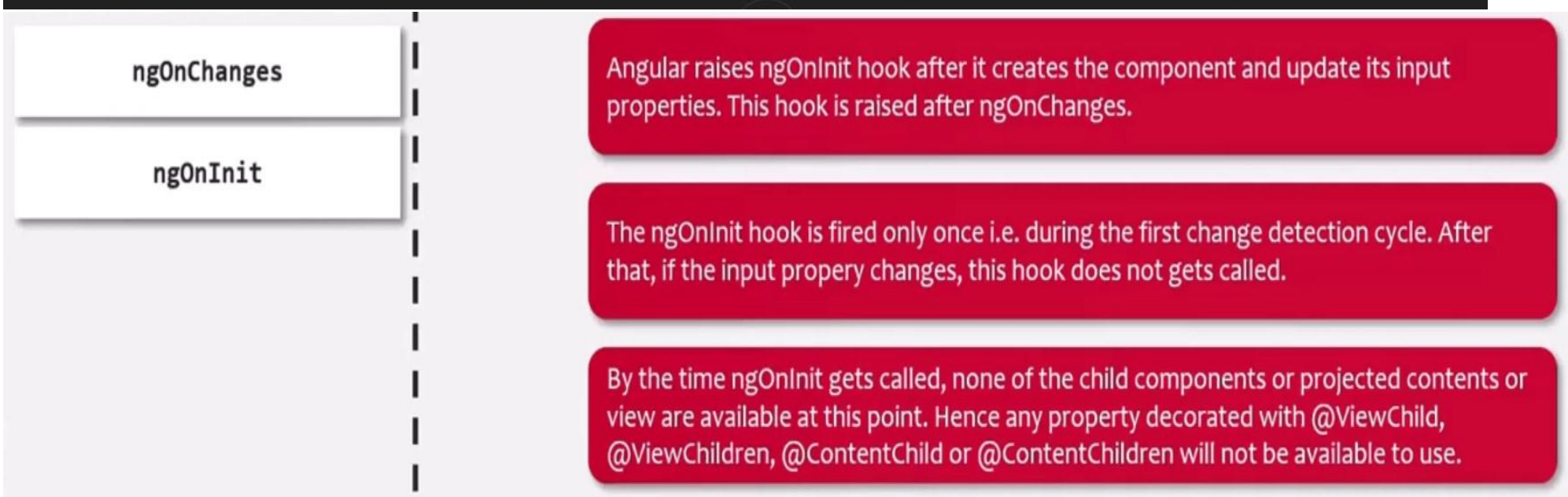
If you try to access a child element using `@ViewChild` inside `ngOnInit`, it might return `undefined` because the child view is not rendered at that point.

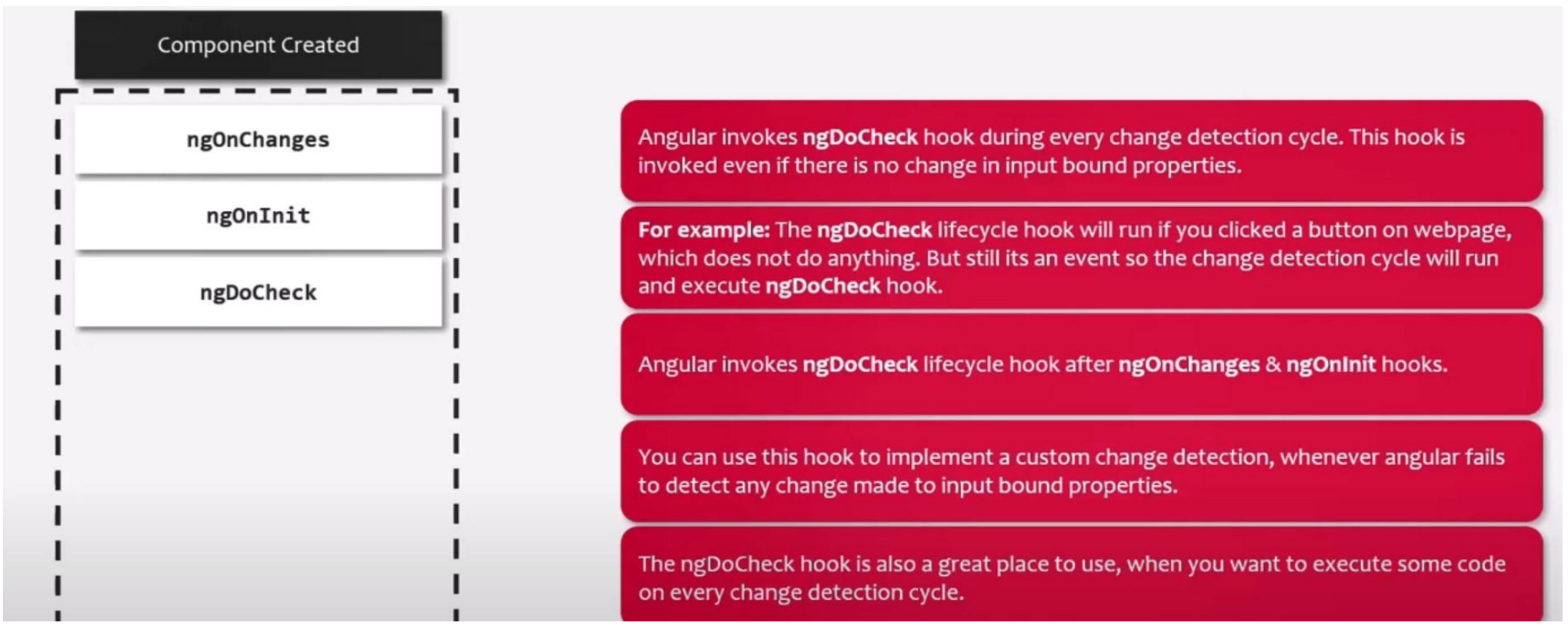
Example:

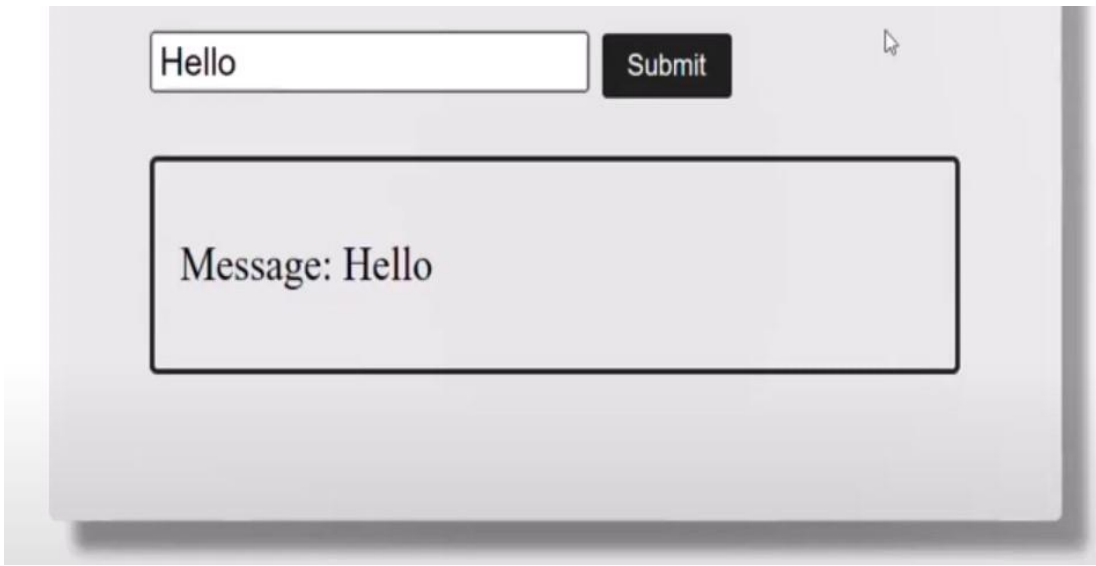
typescript

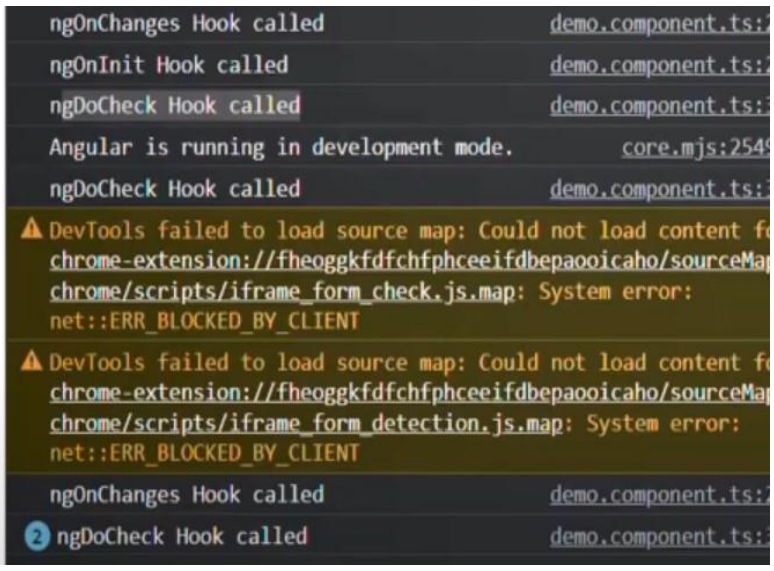
```typescript
@ViewChild('temp') tempPara: ElementRef;

ngOnInit() {
  console.log(this.tempPara.nativeElement.innerHTML);  // May return undefined
}
```

To avoid this, use the `ngAfterViewInit` hook to access child components or DOM elements.









Notes on `ngDoCheck` Lifecycle Hook in Angular

Definition:

- The `ngDoCheck` lifecycle hook in Angular is the third lifecycle hook after `ngOnChanges` and `ngOnInit`. It is invoked during every change detection cycle, regardless of whether there is an actual change in input-bound properties.

Key Points:

- `ngDoCheck` is called during every change detection cycle in Angular.
- It can run even if there are no changes in the input-bound properties.
- You can use this hook to implement custom change detection logic, especially when Angular's default change detection fails to detect changes.
- This hook is useful for executing logic that should run on every change detection cycle.

When Is `ngDoCheck` Invoked?

- Whenever Angular runs a change detection cycle (e.g., during a button click, input focus, or any DOM event), the `ngDoCheck` hook is invoked.
- **Example:** Even if a button click doesn't change any input property, the `ngDoCheck` hook will still run due to the click event triggering the change detection cycle.

Usage:

1. **Custom Change Detection:** If Angular fails to detect changes in input-bound properties (e.g., a property decorated with `@Input`), you can write custom change detection logic inside `ngDoCheck`.
2. **Executing Logic on Every Change Detection Cycle:** If there is logic that should be executed whenever the change detection cycle runs, this logic can be written inside the `ngDoCheck` hook.

Example:

1. **Component Class (`demo.component.ts`):**

typescript

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

@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html'
})
export class DemoComponent implements DoCheck {
  message: string = '';

  // Custom change detection
  ngDoCheck() {
    console.log('ngDoCheck called');
  }
}
```

2. **Component Template (`demo.component.html`):**

html

```html
<input [(ngModel)]="message" />
<button (click)="submit()">Submit</button>
<p>{{ message }}</p>
```

3. **Explanation:**

- When the user types something and clicks the submit button, the `ngDoCheck` hook is called during the change detection cycle.
- Even if the user clicks the button without changing the input, `ngDoCheck` still runs because the button click triggers the change detection cycle.

Additional Example:

- If an event like a focus event on an input element occurs, the `ngDoCheck` hook is called due to the change detection cycle triggered by the focus event.

html Copy code

```html
<input (focus)="noop()" />
```

Yes, exactly! In conclusion, `ngDoCheck` is called during every change detection cycle, even if:

- An input-bound property is not updated.
- No actual change happens in the component.
- Simply clicking a button or triggering a DOM event (like focus) initiates a change detection cycle.

So, this hook gets called every time Angular runs a change detection cycle, regardless of whether there's an actual change in the component's data or properties.

`ngDoCheck()` is called during every change detection cycle, **even if a property hasn't been updated** or **no actual change has occurred**. Simply interacting with the page, like clicking a button or focusing an input, triggers a change detection cycle, and `ngDoCheck()` gets called as a result.

So yes, this hook gets invoked **every time a change detection happens**, regardless of whether there were any changes to the data or properties.

ngAfterContentInit Lifecycle Hook

Angular Lifecycle Hook: `ngAfterContentInit`

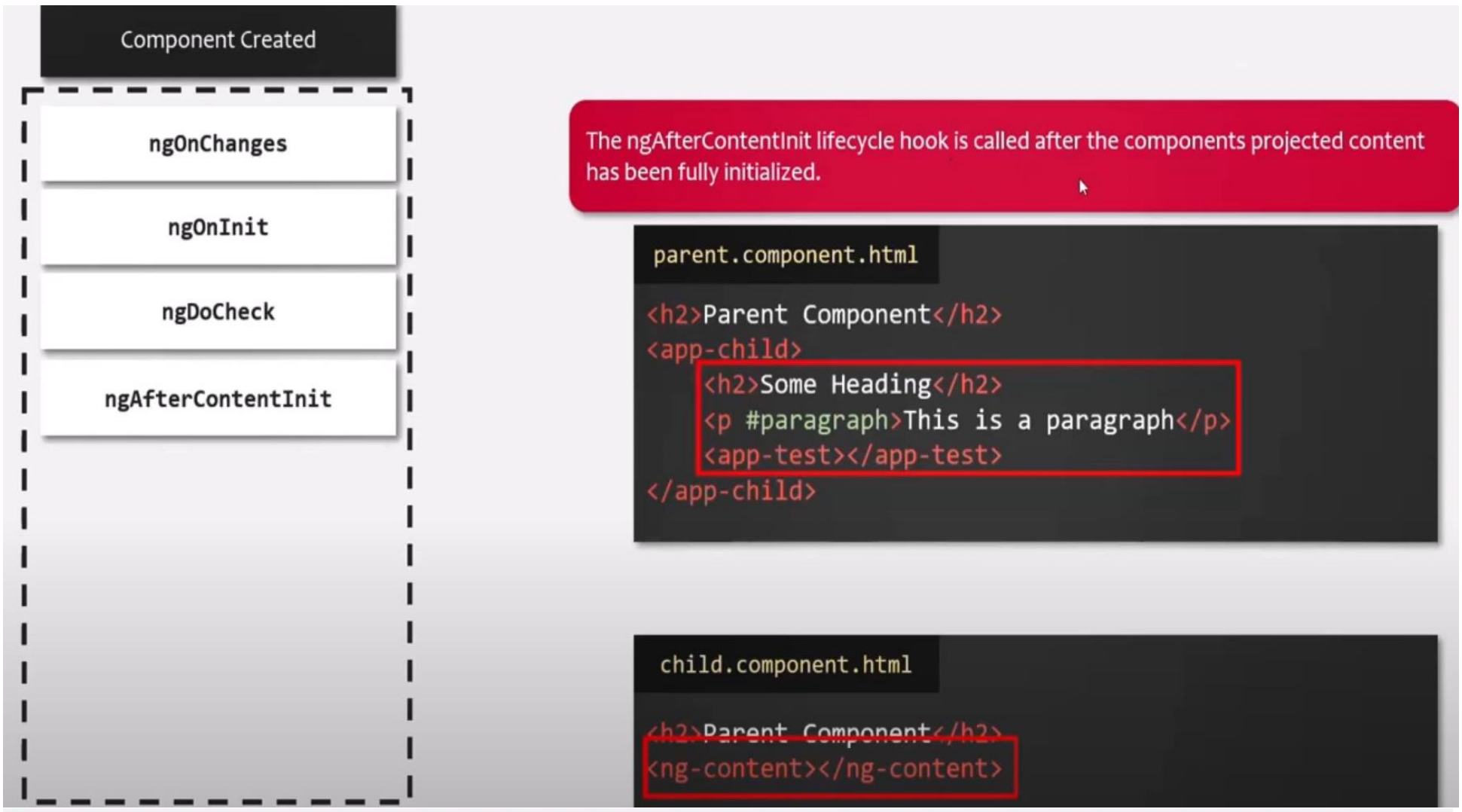
Definition:

The `ngAfterContentInit` is an Angular lifecycle hook that is triggered after the component's projected content (content passed from a parent component) has been fully initialized and injected into the component's view. This hook is called only once, during the first change detection cycle.

- **Purpose**: Called when Angular finishes initializing all projected content in the component.
- **Behavior**: It gets called regardless of whether projected content exists or not.

Key Points:

- Triggered after the projected content has been fully initialized and injected.
- If no content is projected, the hook is still called.
- It runs once, similar to `ngOnInit`, but specifically for projected content.



The ngAfterContentInit lifecycle hook is called after the components projected content has been fully initialized.

Angular updates the properties decorated with @ContentChild & @ContentChildren decorator just before this hook is raised.

1. **Parent Component Template** (`app.component.html`):
   - The reference variable `#projectedContent` should be assigned to the projected content (e.g., the paragraph element):

html

```
<app-demo>
  <p #projectedContent>This is projected content</p>
</app-demo>
```

2. **Child Component** (`demo.component.html`):

html

```
<ng-content></ng-content> <!-- Placeholder for projected content -->
```

3. **Child Component Class** (`demo.component.ts`):

typescript

Copy cod

```typescript
import { Component, AfterContentInit, ContentChild, ElementRef } from '@angular/core';

@Component({
  selector: 'app-demo',
  templateUrl: './demo.component.html'
})
export class DemoComponent implements AfterContentInit {

  @ContentChild('projectedContent') paraContent!: ElementRef;

  ngAfterContentInit() {
    console.log('ngAfterContentInit called');
    console.log(this.paraContent.nativeElement);  // Access projected content
  }
}
```

4. **Result**:

- When the projected content (the paragraph) is fully initialized and inserted in place of the `ng-content` , the `ngAfterContentInit` hook gets called, logging the projected content in the console.

Behavior with No Projected Content:

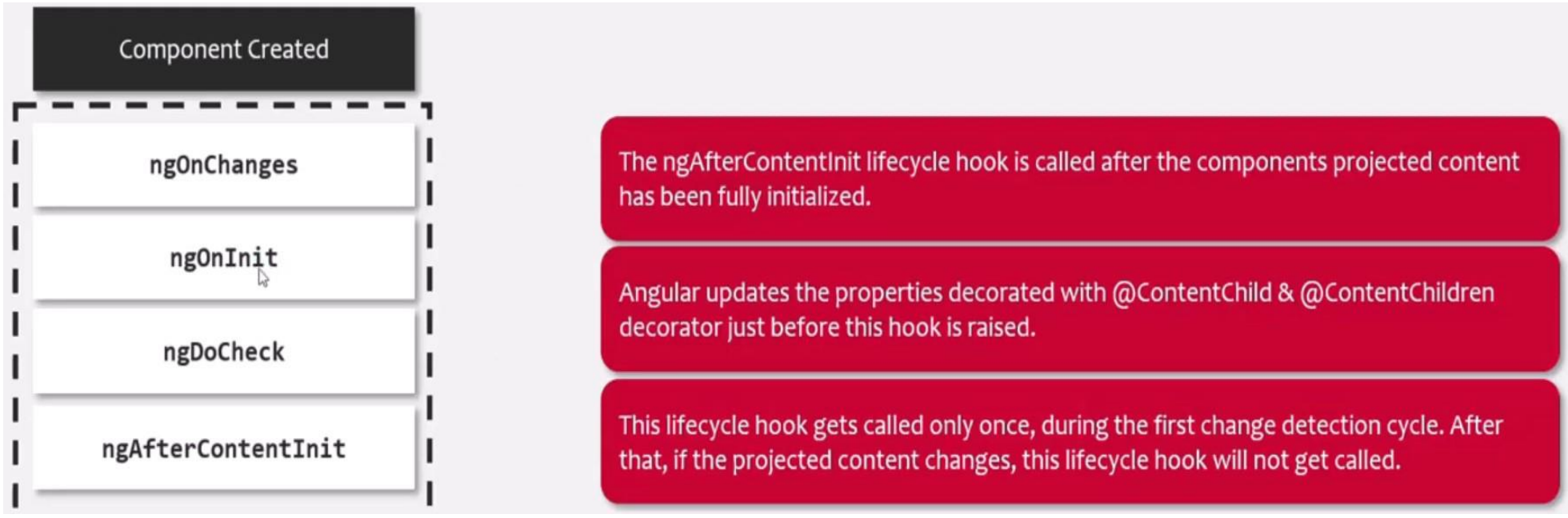
If no content is projected from the parent component, the `ngAfterContentInit` hook will still be called, but any references to projected content (like `paraContent`) will be `undefined` .

2. **Component Template (** `demo.component.html` **):**

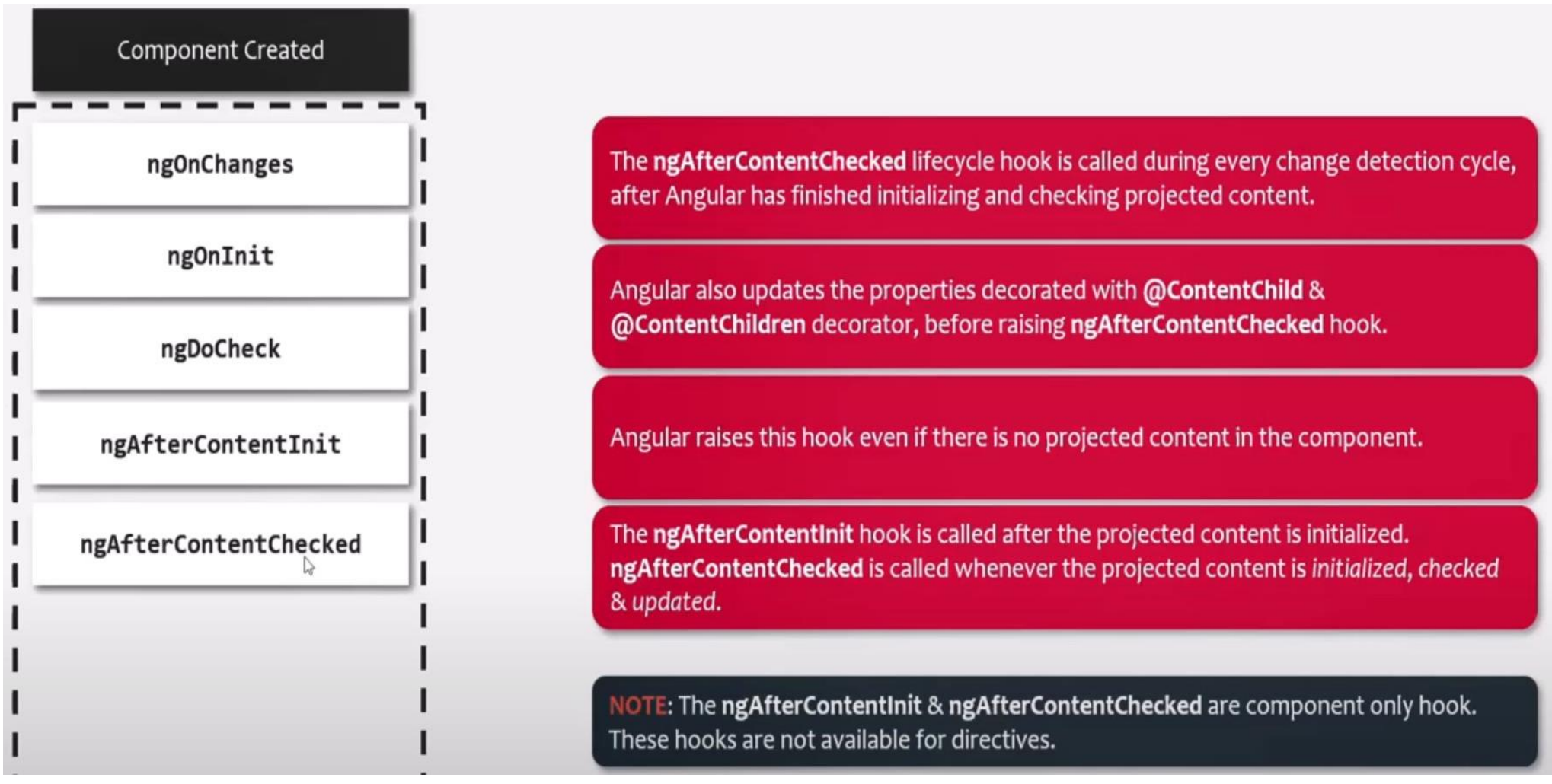
```html
<input [(ngModel)]="message" />
<button (click)="submit()">Submit</button>
<p>{{ message }}</p>
```

2. **Component Template (** `demo.component.html` **):**

```html
<input [(ngModel)]="message" />
<button (click)="submit()">Submit</button>
<p>{{ message }}</p>
```



ngAfterContentChecked Lifecycle Hook



Notes on `ngAfterContentChecked` Lifecycle Hook

Definition:

- `ngAfterContentChecked` is an Angular lifecycle hook that is invoked **after Angular has checked the projected content** during a change detection cycle.
- It is called after the `ngAfterContentInit` lifecycle hook.
- This hook is triggered during every change detection cycle **after projected content has been initialized**, checked, or updated.

Difference Between `ngAfterContentInit` and `ngAfterContentChecked`:

- `ngAfterContentInit`:
  - Called **once**, after the projected content is initialized, during the **first change detection cycle**.
- `ngAfterContentChecked`:
  - Called during the **first change detection cycle**, and **every time the projected content is checked** (whether it changes or not) in subsequent cycles.

When `ngAfterContentChecked` Is Called:

1. During the **first change detection cycle**, after projected content is initialized.
2. Each time the projected content is **updated or checked**, during any change detection cycle.
3. Even if the projected content does **not change**, `ngAfterContentChecked` is called whenever Angular runs a change detection cycle (e.g., on events like clicks or focus).

Example:

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

@Component({
  selector: 'app-demo',
  template: `
    <div>
      <ng-content></ng-content>
    </div>
```

```typescript
export class DemoComponent implements AfterContentChecked {

  ngAfterContentChecked() {
    console.log('ngAfterContentChecked called');
  }
}
```

- **Explanation**: The `ngAfterContentChecked` method logs a message every time a change detection cycle is run, even if the projected content has not changed.

Projected Content Example:

typescript

```typescript
@Component({
  selector: 'app-parent',
  template: `
    <app-demo>
      <p>User has entered: {{ inputVal }}</p>
    </app-demo>
    <input [(ngModel)]="inputVal" />
  `
})
```

```typescript
export class ParentComponent {
  inputVal: string = '';
}
```

- In this case, the `<p>` element is projected into the `app-demo` component, and the projected content is the paragraph showing the user's input.

Key Points:

- `ngAfterContentChecked` :
  - Is called **for every change detection cycle**.
  - Is triggered even when **projected content is not updated**, but the change detection cycle runs.
  - Updates the properties decorated with `@ContentChild` or `@ContentChildren` before it is called.
- **Component-only Hook**:
  - `ngAfterContentChecked` (and `ngAfterContentInit`) is **not available for directives**, it is a component-only hook.

Conclusion:

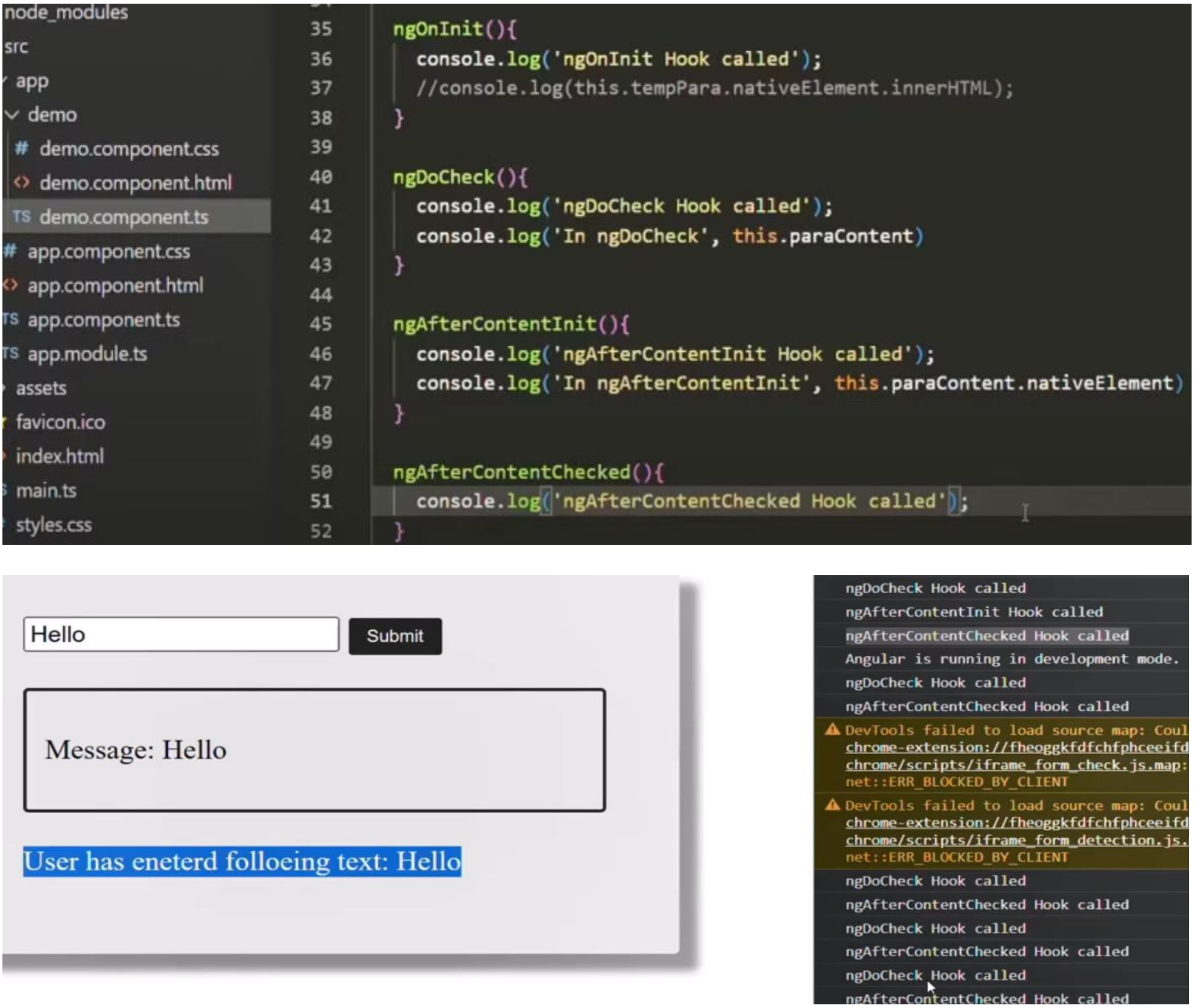
- The `ngAfterContentChecked` hook is very similar to `ngDoCheck`, but it specifically checks **projected content** and is called after Angular has checked or updated that content during every change detection cycle.

demo.component.html | demo.component.ts | app.componen

src > app > app.component.ts > AppComponent > inputVal

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'angular-lifecycle-hook';
  inputVal: string = '';
  constructor(){
    console.log('App Component constructor called!');
  }

  onBtnClicked(inputEl: HTMLInputElement){
      this.inputVal = inputEl.value;
  }
}
```

```html
<div class="container">
    <input type="text" #inputEl (focus)="0">
    <button (click)="onBtnClicked(inputEl)">Submit</button>
    <app-demo [message]="inputVal">
        <p #temp>User has eneterd folloeing text: {{ inputVal }}</p>
    </app-demo>
</div>
```



ngAfterViewInit Lifecycle Hook

ngOnChanges

ngOnInit

ngDoCheck

ngAfterContentInit

ngAfterContentChecked

ngAfterViewInit

The **ngAfterViewInit** is called after the components View template and all its child components view templates are fully initialized.

Angular also updates the properties decorated with **@ViewChild** and **@ViewChildren** decorator before raising this hook.

This hook is called during the first change detection cycle, when Angular initializes the view for the first time.

By the time this hook get's called for a component, all the lifecycle hook methods of child components and directives are completely processed and child components are completely ready.

NOTE:

ngAfterViewInit( ) is called on only components And

They are Not available on Directives.

Notes on `ngAfterViewInit` Lifecycle Hook

Definition:

- `ngAfterViewInit` is an Angular lifecycle hook that is called after the component's view and all its child components' views are fully initialized. It is triggered after the `ngAfterContentChecked` hook.

Key Characteristics:

- Called **once** during the first change detection cycle when the component is created.
- Does not get called again if the view changes after initialization.
- Angular updates properties decorated with `@ViewChild` and `@ViewChildren` **before** this hook is called.

Example:

1. **Basic Implementation:**

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

@Component({
  selector: 'app-demo',
  template: `
    <div>
      <p #tempura>Paragraph Element</p>
    </div>
```

```
export class DemoComponent implements AfterViewInit {
  @ViewChild('tempura') tempura: ElementRef;

  ngAfterViewInit() {
    console.log('ngAfterViewInit hook called');
    console.log(this.tempura); // Logs the ElementRef of the paragraph element
  }
}
```

2. **Behavior with Child Components**:
   - If `DemoComponent` has child components, the `ngAfterViewInit` hook will only be called after the lifecycle hooks of those child components have executed. For example, if `DemoComponent` uses `ComponentB` and `ComponentC`, `ngAfterViewInit` will be called after all lifecycle hooks of `ComponentB` and `ComponentC` are complete.

3. **Console Log Output**:
   - When navigating to the web page, you will see logs indicating the lifecycle hooks called in order:

csharp — Copy code

```
ngAfterContentChecked from DemoComponent
ngAfterViewInit from DemoComponent
ngAfterViewInit from AppComponent
```

Important Notes:

- `ngAfterViewInit` is a **component-only hook** and cannot be used in directives.
- This hook is useful for executing code that depends on the initialization of child views, such as manipulating the DOM or accessing child components.

Feel free to ask if you have any questions about this lifecycle hook!

ngAfterViewChecked Lifecycle Hook

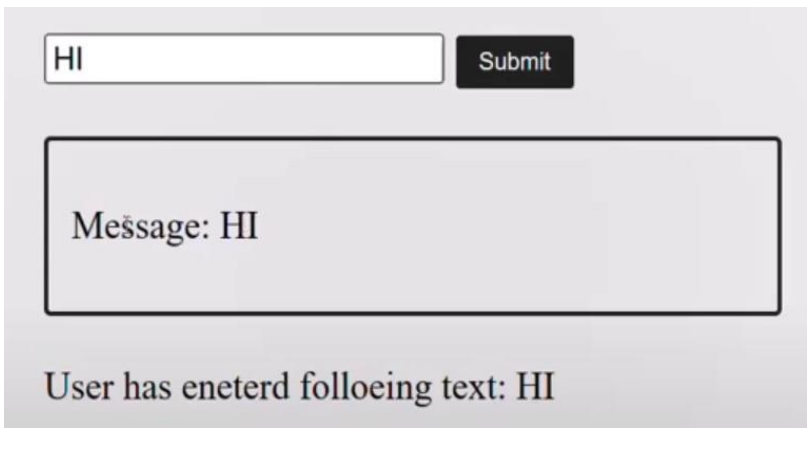
- ngOnChanges
- ngOnInit
- ngDoCheck
- ngAfterContentInit
- ngAfterContentChecked
- ngAfterViewInit
- ngAfterViewChecked

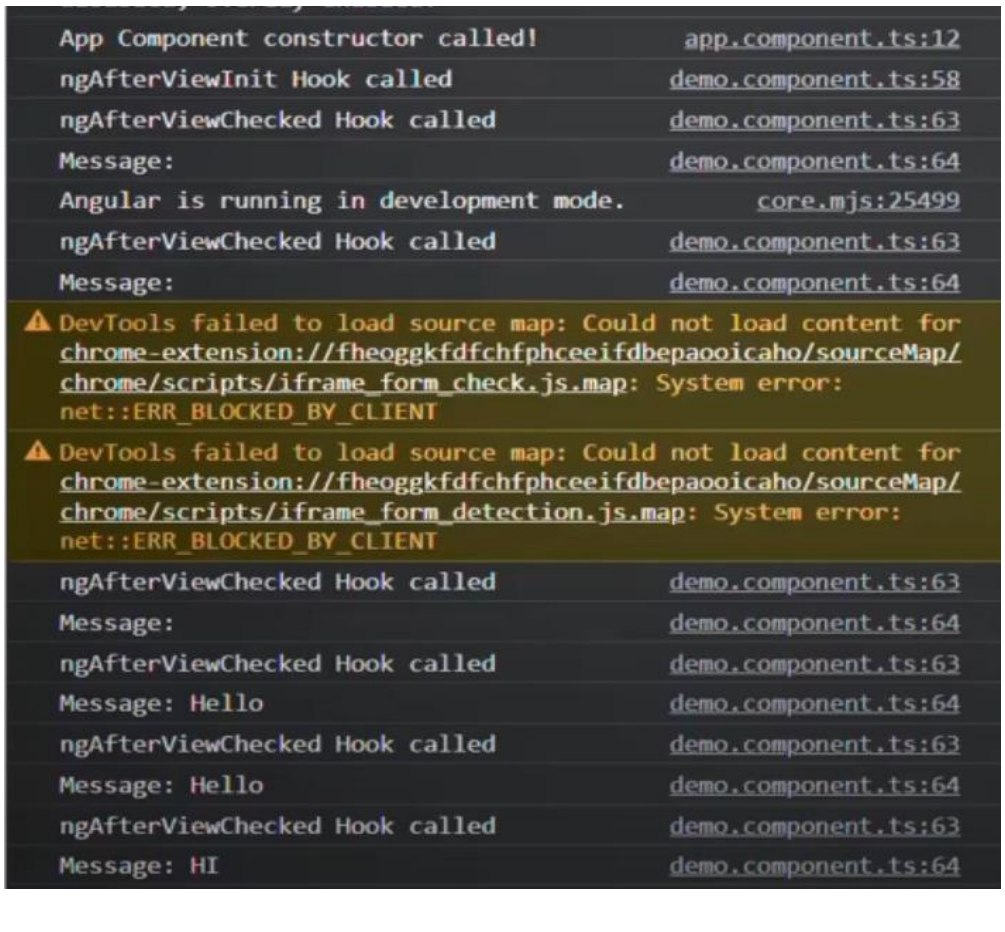
Angular fires **ngAfterViewChecked** hook after it checks and updates the components View template and all its child components view templates.

This hook is called during the first change detection cycle, after **ngAfterViewInit** hook has executed. And after that during every change detection cycle.

Angular also updates the properties decorated with **@ViewChild** and **@ViewChildren** decorator before raising this hook.

NOTE: The **ngAfterViewChecked** hook is a component only hook. It is not available for directives.





Notes on `ngAfterViewChecked` Lifecycle Hook

Definition:

- `ngAfterViewChecked` is an Angular lifecycle hook that is called after the `ngAfterViewInit` lifecycle hook. It is triggered after Angular checks and updates the component's view template and all its child components' views.

Key Characteristics:

- Called **once** during the first change detection cycle after the view is initialized.
- Called **every time** the view of the component or its child components changes.
- Also triggered during each change detection cycle, even if the view has not changed.
- It is a **component-only hook** and not available for directives.

Example:

1. **Basic Implementation:**

typescript Copy code

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

@Component({
  selector: 'app-demo',
  template: `
    <div>
      <p #tempura>{{ message }}</p>
      <input [(ngModel)]="message" placeholder="Type a message" />
      <button (click)="submit()">Submit</button>
    </div>
  `,
})
```

Copy code

```typescript
export class DemoComponent implements AfterViewChecked {
  @ViewChild('tempura') tempura: ElementRef;
  message: string = '';

  ngAfterViewChecked() {
    console.log('ngAfterViewChecked hook called');
    console.log(this.tempura.nativeElement.textContent); // Logs the updated text cont
  }
```

```typescript
  submit() {
    // Logic to handle submission
  }
```

2. **Behavior with View Changes**:

   - When the value of `message` changes (e.g., through input), the content of the paragraph updates, and `ngAfterViewChecked` is called.
   - If you click the submit button, the `ngAfterViewChecked` hook will log the updated text content.

3. **Multiple Calls**:

   - The `ngAfterViewChecked` hook can be called multiple times during a single change detection cycle. For instance, if you have an input field that triggers a change detection cycle on focus, `ngAfterViewChecked` will be invoked even if the view hasn't changed.

Important Notes:

- Before `ngAfterViewChecked` is called, all lifecycle hooks of the child components will have executed.
- It is useful for responding to changes in the view and for performing tasks that need the view to be fully initialized.

Comparison with Other Hooks:

- **ngDoCheck**: Runs for every change detection cycle; used for custom change detection.
- **ngAfterContentChecked**: Runs after the projected content is initialized; cannot compare previous and current projected content.
- **ngAfterViewChecked**: Runs after the component and child views are initialized; suitable for comparing previous and current views.

Feel free to ask if you have any questions about this lifecycle hook!

ngOnDestroy Lifecycle Hook

Notes on `ngOnDestroy` Lifecycle Hook

Definition:

- `ngOnDestroy` is the last lifecycle hook of an Angular component or directive. It is called just before the component or directive is removed from the DOM.

Key Characteristics:

- Fired before the component is destroyed.
- Useful for performing cleanup tasks, such as unsubscribing from observables and detaching event handlers, to prevent memory leaks.
- It is the final opportunity to execute code in the lifecycle of a component or directive.

Example:

1. **Basic Implementation**:

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

@Component({
  selector: 'app-demo',
  template: `<div>Demo Component</div>`,
})
export class DemoComponent implements OnDestroy {
  ngOnDestroy() {
    console.log('ngOnDestroy hook called');
  }
}
```

2. **Using with `ngIf`:**

typescript

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

@Component({
  selector: 'app-root',
  template: `
    <button (click)="toggleComponent()">Show/Hide Component</button>
    <app-demo *ngIf="!toDestroy"></app-demo>
  `,
})
export class AppComponent {
  toDestroy: boolean = false;

  toggleComponent() {
    this.toDestroy = !this.toDestroy;
  }
}
```

3. **Behavior:**

- When the button is clicked, `toggleComponent()` is called, toggling the `toDestroy` property.
- If `toDestroy` is `true`, the `ngIf` directive removes the `DemoComponent` from the DOM.
- Before removal, `ngOnDestroy` is triggered, logging "ngOnDestroy hook called".

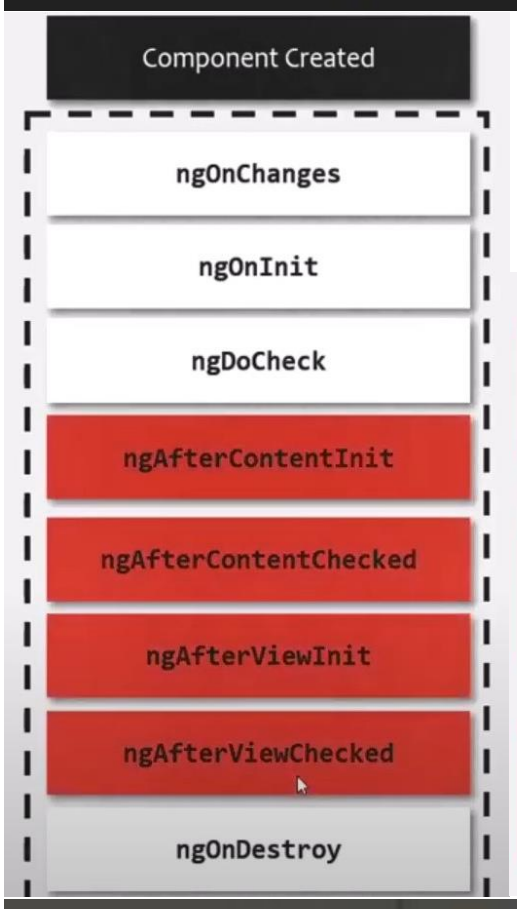
Cleanup Tasks:

- Use `ngOnDestroy` for tasks like:
  - Unsubscribing from services or observables.
  - Removing event listeners.
  - Clearing intervals or timeouts.

Important Notes:

- This hook is not available for directives.
- `ngOnDestroy` provides a clean way to manage resources and avoid memory leaks, making it crucial in Angular applications.

Feel free to ask if you have any questions about the `ngOnDestroy` lifecycle hook!



Angular fires **ngOnDestroy** lifecycle hook just before the component or the directive is destroyed i.e. removed from the DOM.

This hook is a great place to do some cleanup work like unsubscribe from an observable or detach event handler etc., as this hook is called right before the component is destroyed.

This **ngOnDestroy** is the last lifecycle hook of a component & a directive.

component.html | demo.component.ts | app.component.ts

app.component.html > div.container > button

Go to component

```
<div class="container">
    <input type="text" #inputEl (focus)="0">
    <button (click)="onBtnClicked(inputEl)">Submit</button>
    <app-demo [message]="inputVal" *ngIf="!toDestroy">
        <p #temp>User has eneterd folloeing text: {{ inputVal }}</p>
    </app-demo>
    <br><br>
    <button (click)="DestroyComponent()">Show / Hide</button>
</div>
```

Before Calling Ng Destroy Method

After Calling NgDestroy( ) Method.