ngDoCheck

`ngDoCheck` is a lifecycle hook in Angular that provides an opportunity to detect and act on changes that Angular doesn't catch on its own. It's called during every change detection run, and it gives you a chance to implement custom change detection logic. Unlike other lifecycle hooks, `ngDoCheck` is not automatically triggered by Angular; you have to explicitly implement it.

Here's a detailed explanation of `ngDoCheck`:

Definition:

- **Name:** `ngDoCheck`
- **When is it called:** During every change detection run, after `ngOnChanges` and `ngOnInit`.
- **Use Case:** Implement custom change detection logic.

How to Implement `ngDoCheck`:

1. **Import the Necessary Modules:**

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

2. **Implement the DoCheck Interface:**

```
export class YourComponent implements DoCheck {
  // Component properties and methods

  ngDoCheck() {
    // Custom change detection logic
  }
}
```

Example Usage:

Let's consider an example where you want to track changes in a component property manually:

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

@Component({
  selector: 'app-custom-change-detection',
  template: `
    <p>{{ data }}</p>
    <button (click)="updateData()">Update Data</button>
  `,
})
```

```
export class CustomChangeDetectionComponent implements DoCheck {
  @Input() data: string;

  previousData: string;

  ngDoCheck() {
    // Custom change detection logic
    if (this.data !== this.previousData) {
      console.log(`'data' property changed from ${this.previousData} to
${this.data}`);
      this.previousData = this.data;
    }
  }

  updateData() {
    this.data = 'Updated Data';
  }
}
```

In this example:

- The `CustomChangeDetectionComponent` implements `DoCheck`.
- The `ngDoCheck` method is implemented with custom change detection logic.
- It compares the current value of the `data` property with the previous value stored in `previousData`.
- If a change is detected, it logs a message to the console.

Key Points:

1. **Manual Change Detection:**
    - `ngDoCheck` is often used for scenarios where you need to perform manual or custom change detection logic that Angular's default change detection mechanism might not catch.
2. **Use with Caution:**
    - Be cautious when using `ngDoCheck` as it can lead to performance issues if misused. Angular's default change detection is highly optimized, and manual checks should be used judiciously.
3. **Access to Component State:**
    - You have access to the component's state and properties within `ngDoCheck`, allowing you to implement custom checks based on your application's requirements.
4. **Not Automatically Triggered:**
    - Unlike some other lifecycle hooks, `ngDoCheck` is not automatically triggered by Angular. It's your responsibility to implement and use it appropriately.
5. **Combination with `ngOnChanges`:**

- In some cases, `ngDoCheck` is used in combination with `ngOnChanges` to cover a broader range of change detection scenarios.

In summary, `ngDoCheck` provides a way to implement custom change detection logic when necessary. It should be used judiciously, and most of the time, Angular's default change detection is sufficient for handling changes in your application.

The `ngDoCheck` lifecycle hook in Angular is used for implementing custom change detection logic. It's called during every change detection run and provides developers with an opportunity to perform their own checks for changes that Angular might not automatically detect. Here are some scenarios in which you might want to use `ngDoCheck`:

1. **Custom Change Detection:**
   - Use `ngDoCheck` when you need to implement custom change detection logic for specific properties or conditions.

```
ngDoCheck() {
  // Custom change detection logic
  if (this.shouldUpdate()) {
    // Perform actions when changes are detected
  }
}
```

2. **Performance Optimization:**
   - If certain calculations or operations are resource-intensive and don't need to be performed on every change detection cycle, you can use `ngDoCheck` to optimize performance.

```
ngDoCheck() {
  // Custom change detection logic
  if (this.isDirty) {
    // Perform heavy calculations or operations
    this.calculateData();
    this.isDirty = false;
  }
}
```

3. **Checking External Dependencies:**
   - If your component relies on external dependencies that might change asynchronously, such as data fetched from an external service, you can use `ngDoCheck` to react to those changes.

```
ngDoCheck() {
  // Custom change detection logic
  if (this.externalService.hasDataChanged()) {
    // React to changes in external dependencies
    this.loadData();
  }
}
```

4. **Form Input Value Changes:**
   - In situations where you're working with forms and need to detect changes in the input values, you might use `ngDoCheck` to perform custom checks.

```
ngDoCheck() {
  // Custom change detection logic for form input values
  if (this.formInputValue !== this.previousInputValue) {
    // Perform actions on input value changes
  }
}
```

5. **Integration with External Libraries:**
   - When integrating with third-party libraries that have their own state changes, you might use `ngDoCheck` to synchronize your component state.

```
ngDoCheck() {
  // Custom change detection logic for third-party library state
  if (this.thirdPartyLibrary.hasStateChanged()) {
    // Synchronize component state with third-party library
    this.syncWithLibraryState();
  }
}
```

6. **Combining with `ngOnChanges`:**
   - In some scenarios, you might use both `ngDoCheck` and `ngOnChanges` to cover different aspects of change detection. `ngDoCheck` provides a more fine-grained control over changes.

```
ngDoCheck() {
  // Custom change detection logic
  if (this.shouldUpdate()) {
    // Perform actions when changes are detected
  }
}
```

```
  ngOnChanges(changes: SimpleChanges) {
    // React to changes in input properties
    if (changes.data) {
      // Perform actions based on input property changes
    }
  }
```

Keep in mind that while `ngDoCheck` provides flexibility, it should be used judiciously. Angular's default change detection mechanism is highly optimized, and unnecessary use of `ngDoCheck` can lead to performance issues. It's typically used in scenarios where fine-grained control over change detection is necessary.