Cycle de vie des composants

En Angular, les composants ont **un cycle de vie**.

Angular crée le composant, l'affiche, crée et affiche ses éventuels composants enfants, vérifie le changement de propriétés, et détruit le composant avant de le supprimer du DOM.

Angular offre la possibilité d'agir à chaque étape de ce cycle de vie grâce à des accroches (*hook*).

Agir lors d'une étape du cycle de vie

Pour chacune des étapes du cycle de vie, le principe est le même : si le développeur veut effectuer une action à un moment précis, il doit **importer l'interface correspondante depuis** `@angular/core`.

Ainsi, par exemple pour agir sur l'étape `OnInit` :

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


export class MaClasse implements OnInit {


 constructor() { }


  // implémente la méthode ngOnInit de OnInit :
  ngOnInit() {
  // fais quelque chose
 }


}
```

Les huit différentes accroches (*hooks*) sont les suivantes :

`ngOnChanges()` : Cette méthode est appelée par Angular à chaque fois qu'une ou plusieurs propriétés `input` liées à des données change. Elle est toujours également appelée une fois juste avant `ngOnInit()`. Cette méthode reçoit un objet spécifique qui contient les valeurs des propriétés avant et après leur modification.

`ngOnInit()` : Cette méthode **n'est appelée qu'une seule fois**. Elle initialise le composant juste après qu'Angular ait définit les propriétés liées à des données.

`ngDoCheck()` : Cette méthode est appelée à chaque changement et est invoquée une fois après `ngOnInit()`.

`ngAfterContentInit()` : Cette méthode est appelée **une seule fois** immédiatement après qu'Angular ait projeté le contenu externe dans la vue du composant. Elle est appelée après `ngDoCheck()` mais une seule fois.

`ngAfterContentChecked()` : Cette méthode est appelée à chaque fois que le contenu projeté dans le composant est vérifié. Elle est appelée après `ngAfterContentInit()` **mais aussi** après tous les `ngDoCheck()`.

`ngAfterViewInit()` : Cette méthode **n'est appelée qu'une seule fois**. Elle est appelée juste après qu'Angular ait initialisé les vues du composant et des éventuels composants enfants.

`ngAfterViewChecked()` : Cette méthode est appelée à chaque fois qu'Angular vérifie les vues du composant et des éventuels composants enfants. Elle est appelée après ngAfterViewInit() **mais aussi** après tous les `ngAfterContentChecked()`.
`ngOnDestroy()` : Cette méthode est appelée juste avant qu'Angular détruise le composant. C'est à ce moment que les souscriptions aux `Observable` et les gestionnaires d'événements seront supprimées (*nous étuiderons cela en détails*). Cette méthode est particulièrement importante car elle permet d'éviter les fuites mémoire.

Il faut savoir **qu'Angular n'appellera ces accroches seulement si elles sont définies sur la classe du composant**.

Il faut également savoir que les accroches `DoCheck`, `AfterContentChecked` et `AfterViewChecked` sont appelées très souvent : il est donc nécessaire de garder une logique très légère si vous les utilisez sinon vous risquez d'observer des baisses de performance.

Quand utiliser les principales acroches du cycle de vie ?

Nous allons vous donner des indications supplémentaires sur quand il faut utiliser telle ou telle accroche. Nous allons classer ces méthodes par ordre d'importance :

`OnInit()`

Cette méthode doit être utilisée pour **réaliser les initialisations complexes juste après la construction du composant** et pour **configurer le composant juste après qu'Angular ait définit les propriétés** `input`.

Cette méthode est principalement utilisée pour mettre la logique de récupération des données utilisées dans le composant.

Retenez que cette méthode est la plus utilisée sur les composants.

`OnDestroy()`

Cette méthode permet de **mettre la logique de nettoyage du composant qui doit avoir lieu avant la destruction du composant afin d'éviter les fuites mémoires**.

Cela permet notamment d' `unsubscribe` aux `Observables`, de signaler à d'autres éléments Angular (composants, directives ou services) que le composant va être supprimé. Il faut également penser à stopper les éventuels timer (`interval` par exemple).

En résumé c'est le lieu pour éviter les fuites mémoires pour tout ce qui ne peut pas être automatiquement supprimé par Angular.

`OnChanges()`

La méthode `ngOnChanges()` est appelée à chaque fois que des modifications de propriétés `input` surviennent.

Prenons un exemple pour bien comprendre `ngOnChanges()` :

Nous allons créer un composant parent contenant deux propriétés : `prenom` qui est une chaîne de caractères et `monObj` qui est un objet contenant une propriété `prenom`.

Sur le template du composant parent, nous allons créer deux `input` avec une double liaison de données (*en utilisant la directive native `ngModel`, sans oublier d'importer le module `FormsModule`*).

Ces `inputs` permettent de modifier les deux propriétés que nous avons définis.

Nous allons ensuite passer ces propriétés à un composant enfant grâce à deux propriétés `@Input()`.

Sur le template du composant parent il suffit d'ajouter la liaison de propriété : `<app-enfant [prenom]="prenom" [monObj]="monObj"></app-enfant>`.

Sur la classe du composant enfant, il suffit de déclarer ces liaisons grâce au décorateur `@Input()` : `@Input() prenom;` pour la première propriété par exemple.

Nous allons maintenant implémenter l'accroche `ngOnChanges()` en important `OnChanges` depuis `@angular/core`.

Nous allons ensuite créer une boucle pour afficher à chaque fois que la méthode `ngOnChanges()` est déclenchée, la propriété `@Input()` qui a été modifiée.

Nous avons pour cela accès à un objet passé par `ngOnChanges()` qui comporte pour chaque propriété modifiée un objet avec notamment une propriété `currentValue` qui contient la nouvelle valeur de la propriété, et une propriété `previousValue` qui contient l'ancienne valeur.

Nous affichons sur le template du composant enfant ces modifications grâce à un `*ngFor` et à une interpolation de chaîne de caractères.

Nous obtenons ce résultat :

https://stackblitz.com/edit/angular-c4-l9-1

Tapez dans les deux `input`. Et surprise ! Lorsque vous modifiez le premier `input`, tout a l'air de fonctionner : la modification de propriété est correctement affichée. Mais lorsque vous modifiez le second `input` rien ne se passe !

Quelle est la raison de cette situation ?

`ngOnChanges()` ne catch pas les changements sur `monObj.prenom`.

La raison est liée à Javascript : en javascript les objets sont passés par référence et non par valeur. Ce qui signifie que la référence à `mon.Obj` ne change pas lorsque l'on modifie la valeur de sa propriété `prenom`.

Donc de la perspective de `ngOnChanges()`, `mon.Obj` n'a pas changé et la méthode n'est donc pas appelée.

Il est important de comprendre cette situation pour ne pas se retrouver avec des bugs qui paraissent insolubles !

`DoCheck()`

Comment résoudre ce problème ? La réponse est l'utilisation de la méthode `DoCheck()`.

Rappelons que cette méthode est appelée à chaque changement d'une propriété. Il faut donc l'utiliser en dernier recours car elle entraîne une charge très importante pour l'application : cette méthode sera exécutée un énorme nombre de fois.

Voici l'exemple simplifié en utilisant `DoCheck()` :

https://stackblitz.com/edit/angular-c4-l9-2

Nous avons enlevé la propriété qui contenait uniquement une chaîne de caractères pour se concentrer sur l'objet.

Nous observons maintenant que les changement à la propriété `prenom` de `monObj` sont bien pris en compte.

Nous attirons une nouvelle fois votre attention sur les problèmes de performance induits par `DoCheck()` : cette méthode est appelée une quantité impressionnante de fois (par exemple cliquer sur un `input` suffit pour l'invoquer. Il faut donc l'utiliser de manière parcimonieuse et ne mettre que des logiques légères.

Exemple de la vidéo

Voici l’exemple de la vidéo :

https://stackblitz.com/edit/angular-chap4-l9