

Pipe pur et impur

- La fonction de transformation du pipe doit être pure
 - La valeur de sortie doit dépendre uniquement des valeurs d'entrée
 - La fonction ne modifie pas de valeurs en dehors de ses variables locales
- ⚠ Utilisation du pipe avec les tableaux et les objets
- Un pipe est pur par défaut, mais il est possible de le rendre 'impur'

```
@Pipe({  
  name: 'impurePipe',  
  pure: false  
})
```

- ⚠ Angular ne 'devine' pas si le pipe est pure ou impur, c'est à vous de lui indiquer selon l'usage
- Un pipe impur n'a plus l'optimisation du pipe pur, et sera exécuté aussi souvent qu'une méthode

Pipes et objets

0 Incrément index 0

1 Incrément index 1

2 Incrément index 2

3 Incrément index 3

4 Incrément index 4

Ajouter Supprimer Somme des éléments : 10

- Ce composant affiche les éléments d'un tableau
- Il est possible d'incrémenter les éléments, et d'ajouter ou supprimer un élément
- La somme totale des éléments est affichée, en utilisant un pipe
- Problème : La somme n'est pas mise à jour si l'utilisateur modifie les éléments

Solution 1

- Utiliser un pipe impure

```
@Pipe({  
  standalone: true,  
  name: 'sumPipe',  
  pure: false  
})
```

- ✓ Solution simple
- ✗ Peut impacter les performances si l'opération est complexe (tri, ...)

- Exercice : trouver une solution sans passer par un pipe impure

Solution 2

- Forcer le changement de référence manuellement (le tableau devient immuable)

```
increment(index: number): void {  
    this.tableau = [...this.tableau]  
    this.tableau[index] += 1  
}  
  
addElement(): void {  
    this.tableau = [...this.tableau]  
    this.tableau.push(this.tableau.length)  
}  
  
removeElement(): void {  
    this.tableau = [...this.tableau]  
    this.tableau.pop()  
}
```

✓ Meilleures performances : le pipe est appelé uniquement lorsque la valeur change

✗ Il faut faire la modification si l'on ajoute une opération, ou si on réutilise le pipe ailleurs

✗ Le tableau est recréé à chaque fois, et il est parcouru entièrement pour calculer la somme

Solution 3

- Déléguer la logique au composant, le pipe n'est plus utilisé

Somme des éléments : {{ somme }}

```
somme: number = this.tableau.reduce((a, b) => a + b, 0)
```

```
...
```

```
increment(index: number): void {  
  this.tableau[index] += 1  
  this.somme += 1  
}
```

```
addElement(): void {  
  this.somme += this.tableau.length  
  this.tableau.push(this.tableau.length)  
}
```

```
removeElement(): void {  
  let n = this.tableau.pop()  
  if(typeof n === 'number') {  
    this.somme -= n  
  }  
}
```

✓ Meilleures performances

✗ Il faut implémenter la logique de la somme dans le composant

- Il n'y a pas de méthode meilleure dans tous les cas de figure
- C'est à vous de vous adapter selon l'usage

Découpage en composants

@Input

```
@Input({
  alias: 'alias',
  required: true,
  transform: (input: number) => input * 10
})
attribut: number = 0
```

- Le décorateur @Input lie l'attribut d'un composant à une propriété de son élément

```
alias?: string
```

- Par défaut, la propriété liée à le même nom que l'attribut. L'alias permet de définir une propriété différente (ici alias)

```
<app-fils [alias]="value" > </app-fils>
```

@Input

```
@Input({
  alias: 'alias',
  required: true,
  transform: (input: number) => input * 10
})
attribut: number = 0
```

- Le décorateur @Input lie l'attribut d'un composant à une propriété de son élément

```
required?: boolean
```

- Depuis Angular 16, il est possible d'obliger le binding de la propriété
- Si required est à true, une erreur sera renvoyée à la compilation si l'on essayer de créer le composant sans lier la propriété

```
<app-fils></app-fils> <!-- ici erreur de compilation -->
```


@Input

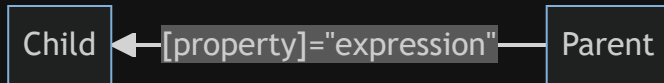
```
@Input({
  alias: 'alias',
  required: true,
  transform: (input: number) => input * 10
})
attribut: number = 0
```

- Le décorateur @Input lie l'attribut d'un composant à une propriété de son élément

```
transform?: (value: any) => any
```

- Appelle un callback pour transformer la propriété en entrée avant de l'assigner à l'attribut
- Ici la fonction multiplie simplement l'entrée par 10, mais il est possible de changer le type d'entrée
- ⚠ Comme souvent avec les décorateurs, le compilateur ne vérifie pas que le type de retour de votre callback corresponde à celui de l'attribut à lier

Input



```
<app-fils [alias]="value" > </app-fils>
```

- La syntaxe dans le template du père est la même que pour se à lier une propriété
- ⚠ Comme pour le property binding, un changement de la propriété chez le fils n'est pas répercuté chez le père

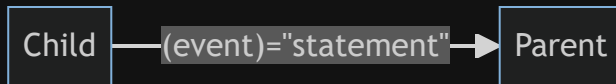
@Output

```
@Output({
  alias: 'alias',
})
event: EventEmitter<number> = new EventEmitter()
...

...
event.emit(value)
...
```

- Le décorateur @Output permet au composant fils d'émettre des évènements
- Il est possible de définir un alias comme pour l'input
- La variable doit être du type EventEmitter<T>, avec T le type de valeurs à émettre
- Chaque appel à emit() dans le composant fils envoie un évènement dans le composant père

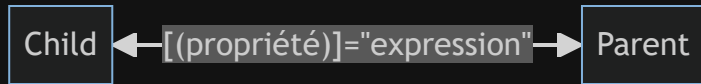
Output



```
<app-fils (event)="value=$event" > </app-fils>
```

- La syntaxe dans le template du père est la même que pour se lier à un évènement d'un élément HTML
- La variable \$event permet de récupérer directement la valeur renvoyée par le composant fils

Two-way binding



- La donnée peut être modifiée par le père et le fils
- Combinaison d'une property et d'un event binding, avec une syntaxe raccourcie

```
<app-fils [(value)]="value"> ... </app-fils>  
<app-fils [value]="value" (valueChange)="value=$event"> ... </app-fils>
```


- C'est ce qui est fait lors de l'utilisation de la directive ngModel

```
<input [(ngModel)]="data" />  
<input [ngModel]="data" (ngModelChange)="data=$event" />
```

@Input et @Output

- Dans le composant fils, une variable @Input x doit avoir comme @Output xChange

```
...  
@Input()  
value: number = 0  
  
@Output()  
valueChange: EventEmitter<number> = new EventEmitter()  
...
```

-  Il est nécessaire d'appeler manuellement la méthode emit() dans le composant fils

```
...  
addValue() {  
  this.value++  
  this.valueChange.emit(this.value)  
}  
...
```

Exercices (1/2)

- Refactoriser l'application de démo, en créant un composant pour la liste des topics, et un composant pour la description

Exercices (2/2)

- Partant de la classe suivante :

```
class Tree {  
  constructor(public value: string, public children: Tree[] = []) {}  
}
```

- Créer un composant permettant d'afficher une arborescence :
- On peut afficher ou cacher les fils d'un élément de l'arbre en cliquant dessus

Recursive Component

root

folder

readme