

Template variables

- Dans le template, il est possible d'utiliser le symbole # pour déclarer une variable de template


```
<input #classe [value]="classeString" />
<input #style [value]="styleString" />
```

- La variable référence l'élément, et peut être utilisée de partout dans le template

```
<div>La valeur de mon input est {{ classe.value }}</div>
<button (click)="maj(classe.value, style.value)">Mise à jour</button>
```

- Le type de cette variable dépend d'où elle est déclarée
 - Pour un élément HTML standard, l'interface correspondante (HTMLInputElement pour un input)
 - Pour un composant, le type du composant
- https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API

Priorité et portée des variables

- Les variables de templates sont globales dans le template du composant, sauf celles à l'intérieur d'une directive structurelle
- Une variable de template peut être utilisée sur plusieurs éléments, dans ce cas la première sera prioritaire
- Les variables de template peuvent avoir le même nom que des attributs du composant
- Si plusieurs variables ont le même nom dans le template, la priorité suivante est appliquée
 - Variables définies dans une directive structurelle
 - Variable de template
 - Attribut du composant
-  Utiliser des noms différents pour les variables

Standalone components

```
@Component({
  standalone: true,
  imports: [CommonModule, OtherStandaloneComponent],

  selector: 'standalone-component',
  template: ' ... ',
  styles: [ ... ]
})
export class StandaloneComponent {
```

- Depuis Angular 14, il est possible de créer des composants standalone (ainsi que des directives et pipes), sans passer par des modules
- Dans le décorateur, il suffit d'ajouter le flag standalone à vrai
- Il faut également spécifier les dépendances directement dans le composant
- Si l'on souhaite réutiliser le composant standalone, on peut l'importer directement (dans un module ou directement dans un autre composant standalone)

Standalone components

```
bootstrapApplication(StandaloneComponent).catch(err => console.error(err));
```

- On peut bootstrap directement sur un composant standalone

Pipes

Pipes



- Un pipe est une classe Angular, qui possède une méthode de transformation
- Un pipe est utilisée dans le template pour transformer une expression en une autre
- Les pipes sont utilisés avec l'opérateur |

```
{{ entrée | pipe }}
```

Pipes



- Un pipe est une classe Angular, qui possède une méthode de transformation
- Un pipe est utilisée dans le template pour transformer une expression en une autre
- Les pipes sont utilisés avec l'opérateur |

```
{{ entrée | pipe | secondPipe }}
```

- Il est possible de faire des enchainements de pipes, où la sortie d'un pipe devient l'entrée de la suivante

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```


Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Décorateur @Pipe

- Indique à Angular que la classe est un pipe
- Utilisation de métadonnées pour paramétrer le pipe
- Un pipe peut être standalone

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Sélecteur CSS du pipe

```
...
{{ 12 | exponential:2 | exponential | exponential:3 }}
...
```

Créer un pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential',
  standalone: true
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent = 1): number {
    return Math.pow(value, exponent);
  }
}
```

Implémentation :

- Un pipe implémente l'interface PipeTransform
- La méthode transform est une fonction variadique, et a au moins un paramètre, l'expression à transformer
- ⚠ Attention au type d'entrée de sortie des pipes

Exercice

- Créer un pipe qui tronque une chaîne de caractères trop longue, et ajoute "..." à la fin
- La taille maximale de la chaîne en entrée est paramétrable, avec une valeur par défaut de 10

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'tronque',
  standalone: true
})
export class TronquePipe implements PipeTransform {

  ...

}
```

Pipes Angular

- Un certain nombre de pipes sont disponibles dans le package CommonModule :
- UpperCasePipe (uppercase) et LowerCasePipe (lowercase) : Convertit la valeur respectivement en majuscules ou minuscules
- DatePipe (date) : Formatte une date
- DecimalPipe (number), PercentPipe (percent), CurrencyPipe (currency) : Formatte un nombre
- JsonPipe (json) : Convertit un objet sous forme JSON (utile pour debug)
- AsyncPipe (async) : Souscrit à un Observable ou une Promise et retourne la dernière valeur émise
- <https://angular.io/api?type=pipe>

Pourquoi les pipes ?

Performance

- Angular execute le pipe uniquement lorsque l'expression d'entrée change

```
{{ toUpperCase(texte) }}
```

```
{{ texte | toUpperCase }}
```

Lisibilité

- En particulier lors de chaînage de pipes, l'expression finale est bien plus lisible

```
{{ toUpperCase(lowercase(format(transformation(texte)))) }}
```

```
{{ texte | transformation | format | lowercase | toUpperCase }}
```