

12/12/2022

Version 3

The Angular logo, featuring the word "ANGULAR" in a bold, dark blue, sans-serif font. The text is centered within a white, scalloped-edged circular shape. The background is a solid teal color.

ANGULAR

A decorative wavy line in light blue and white, running vertically along the left side of the slide.

INTRODUCTION

INTRODUCTION À ANGULAR

INTRODUCTION

- Utilisation d'un logiciel de traitement de texte type VSCode, Sublime, Atom, Notepad++, Eclipse ou autre
- Prise en main d'un terminal
- Prise en main de navigateurs et de leur console de debug
- Préparer le répertoire Projet
- Installer Node.JS & NPM

INTRODUCTION

- Description HTML5 / CSS3
- Langage TypeScript
- Framework créé par Google
 - AngularJS est la première version
- Patterns
 - MVC (Model-View-Controller)
 - MVVM (Model-View-ViewModel) pour le binding
 - IoC (Inversion of Control) pour l'injection de dépendances

INTRODUCTION

- AngularJS a été écrit et pensé pour JavaScript
 - AngularJS désigne en réalité la version 1.x de Angular
- Angular a été écrit et pensé pour TypeScript
 - Angular désigne les versions 2.x et supérieures
- Toute la document est accessible sur <https://angular.io/docs>



TYPESCRIPT

INTRODUCTION À TYPESCRIPT

GESTIONNAIRE DE PAQUETS

- NPM est un gestionnaire de paquets
- Commandes utiles
 - Pour installer un paquet de façon globale sur la machine

```
npm install -g package
```

- Pour installer une dépendance dans un projet

```
npm install package
```

- Initialiser le projet (prépare le fichier *package.json* dans le répertoire projet)

```
npm init
```

- Pour récupérer les dépendances d'un projet (s'appuie sur le fichier *package.json*)

```
npm install
```

TYPESCRIPT

- Bienvenue dans TypeScript !
 - Du JavaScript mais ...Typé
 - Qui ne peut pas être interprété par les navigateurs
 - Il est donc « compilé » en JavaScript, qu'on appelle transpilation
- L'extension des fichiers TypeScript est TS
- Il ajoute de nouvelles fonctionnalités à JS
 - Les décorateurs
 - Le typage de variables
 - Les classes & interfaces (mais depuis ES6, c'est déjà le cas en JS)
 - L'import / export (mais depuis ES6, c'est déjà le cas en JS)
 - La signature des méthodes
 - La généricité

TYPESCRIPT

- Déclaration de variables

```
let maVar: number = 0;  
const maConstant: string = "Jérémy";  
let myClient: IClient = new Client();  
let myPersonnes: Array<Personne> = new Array<Personne>();
```

- | | |
|--------------------------------|-----------------------|
| – number | Entier ou flottant |
| – string | Chaîne de caractères |
| – boolean | Vrai / Faux |
| – IClient | Objet de type IClient |
| – Array<Personne> / Personne[] | Tableau de personnes |

TYPESCRIPT

- Déclaration de variables

```
let maVar!: number;  
let maVarNullable?: string;  
let maVarMultiTypes!: number | string | null;  
let myObject!: any;
```

- | | |
|------------|--|
| – (pipe) | Plusieurs types possibles |
| – any | Valeur dont on ignore le type concret |
| – unknow | Valeur dont on ignore le type concret, mais qui sera « type-safe » |

Le « ! » permet de gérer le cas où la variable n'est pas instanciée, mais le sera prochainement.
Le « ? » permet de gérer le cas où la variable pourra être null.

TYPESCRIPT

- Déclaration d'une classe

```
class Personne {  
  private nom: string;  
  public prenom?: string = "Jérémy";  
  protected age: number = 20;  
  
  constructor(nom: string, prenom?: string) {  
    this.nom = nom;  
    this.prenom = prenom;  
  }  
  
  public getNom(): string {  
    return `${this.prenom} ${this.nom}`;  
  }  
}
```

TYPESCRIPT

- Déclaration d'une classe

```
class Personne {  
  constructor(private nom: string, private prenom?: string) { }  
  
  public getNom(): string {  
    return `${ this.prenom } ${ this.nom }`;  
  }  
}
```

TYPESCRIPT

- Déclaration d'une classe

```
class Personne {  
    public get nom(): string {  
        return this._nom;  
    }  
  
    public set nom(value: string) {  
        this._nom = value;  
    }  
  
    public get prenom(): string {  
        return this._prenom;  
    }  
  
    public set prenom(value: string) {  
        this._prenom = value;  
    }  
  
    constructor(private _nom: string, private _prenom?: string) { }  
  
    public getNom(): string {  
        return `${ this._prenom } ${ this._nom }`;  
    }  
}
```

- Dans un contexte **Template Angular** –
 - > Déconseillé d'utiliser les getter pour lire
 - > Déconseillé d'utiliser les méthodes

Car les appels seront infinis
> Plusieurs par secondes ...

- Dans un contexte **Angular API REST** –
 - > Les données seront des objets littéraux

Donc le typage n'est qu'indicatif dans ce cas

TYPESCRIPT

- Déclaration d'une classe avec héritage

```
class Client extends Personne {  
  constructor(nom: string, prenom: string, private ca: number) {  
    super(nom, prenom);  
    this.ca = ca;  
  }  
}
```

TYPESCRIPT

- Déclaration et implémentation d'une interface

```
interface IClient {  
    getCa(): number;  
}
```

```
class Client extends Personne implements IClient {  
    constructor(nom: string, prenom: string, private ca: number) {  
        super(nom, prenom);  
        this.ca = ca;  
    }  
  
    public getCa(): number {  
        return this.ca;  
    }  
}
```

TYPESCRIPT

- Déclaration et utilisation de la généricité

```
interface IClient<T> {  
    getCa(): T;  
}
```

```
class Client extends Personne implements IClient<number> {  
    constructor(nom: string, prenom: string, private ca: number) {  
        super(nom, prenom);  
        this.ca = ca;  
    }  
  
    public getCa(): number {  
        return this.ca;  
    }  
}
```


TYPESCRIPT

- Installation du transpileur TypeScript

```
npm install -g typescript
```

- Vérifier l'installation

```
tsc -v
```

- Transpilation

```
tsc fichier.ts
```

TYPESCRIPT

- On peut transpiler avec des options
 - Créer un fichier *tsconfig.json*

```
{  
  "compilerOptions":  
  {  
    "target": "ES2022",  
    "module": "commonjs",  
    "lib": [ "DOM", "ES2022" ]  
  }  
}
```

```
tsc --project tsconfig.json
```

EXERCICE

- Installer NPM (NodeJS)
- Créer un fichier TypeScript avec une classe
 - Personne
 - Qui a un nom, un prénom
 - Client qui hérite de Personne
 - Qui a un CA et une liste de produits
 - Fournisseur qui hérite de Personne
 - Qui a un nom de société
 - Produit
 - Qui a un nom, un prix, un fournisseur
 - Qui a une liste de clients
- Compiler en JS

TYPESCRIPT

- On peut utiliser des décorateurs
 - Pour ajouter du comportement à une classe, une méthode, une propriété
 - Joue le même rôle qu'un attribut en PHP et en C#, ou qu'une annotation en Java

```
const DemoDecorator = (options: any) => (target: Object) => {  
    console.log(options.description, target);  
};
```

```
@DemoDecorator({  
    description: "Démonstration"  
})  
class Personne {  
}
```

TYPESCRIPT

- Pour utiliser les décorateurs, il faut ajouter l'option *experimentalDecorators*

```
{
  "compilerOptions":
  {
    "target": "ES2022",
    "module": "commonjs",
    "lib": [ "DOM", "ES2022" ] ,
    "experimentalDecorators": true
  }
}
```

A decorative wavy line in light blue and white, flowing from the top left towards the bottom left of the slide.

PRÉSENTATION

PRÉSENTATION D'ANGULAR

PRÉSENTATION D'ANGULAR

- Modèles
 - TypeScript
- Vues
 - HTML5 / CSS3
- Contrôleur
 - TypeScript

PRÉSENTATION D'ANGULAR

- Quelques notions d'Angular
 - Import
 - Il faudra importer les éléments dont vous aurez besoin, un peu comme en Java
 - Module
 - Conteneur de directives, de composants, de services, ...
 - Directive
 - C'est une classe
 - C'est un Component sans vue
 - Component
 - Hérite de Directive
 - Composant Angular
 - Composé de vue (template) HTML, d'une classe et éventuellement d'un style CSS

PRÉSENTATION D'ANGULAR

- Implémentation du Design Pattern IoC
 - Un composant n'est plus responsable de sa dépendance
 - Le composant déclare sa dépendance
 - Le composant n'instancie pas sa dépendance
- Résolution d'une dépendance
 - Basé sur le type de la dépendance ou sur un identifiant, appelé « token »
 - Demandée en tant qu'argument dans un constructeur

```
constructor(private service: UnServiceInjectable) {}
```

```
constructor(@Inject('LeTokenDuService') private service: UnServiceInjectable) {}
```

```
constructor(@Inject('LeTokenDuService') private services: UnServiceInjectable[]) {}
```

PRÉSENTATION D'ANGULAR

- Règles de nommage
 - AppModule `app.module.ts`
 - AppComponent `app.component.ts`
 - MonComposantComponent `mon-composant.component.ts`
 - ProduitModule `produit.module.ts`
 - AscBoldComponent `asc-bold.component.ts`
 - ProduitDirective `produit.directive.ts`
 - Produit `produit.ts`
 - ProduitService `produit.service.ts`

PRÉSENTATION D'ANGULAR

- L'architecture sera la suivante

- Racine projet

- src

- index.html

Point d'entrée, fichier principal

- main.ts

Fichier principal Angular

- app

Sources du projet Angular

- assets

Ressources statiques (CSS, images, etc.)

- package.json

Configuration du projet NPM

- angular.json

Configuration de l'application Angular

- tsconfig.json

Configuration de TS

PRÉSENTATION D'ANGULAR

- Notre application Angular a besoin d'un « exécuteur » pour démarrer
 - On va utiliser Browser, puisque notre application s'exécutera dans un navigateur
- BrowserModule (platform-browser)
 - Contient le code partagé pour l'exécution au sein d'un navigateur (thread DOM entre autre)
- platformBrowserDynamic (platform-browser-dynamic)
 - Contient le code côté client qui permet
 - De générer et d'intégrer les templates HTML, avec le binding MVVM, les composants, les directives, ...
 - De gérer l'injection de dépendances (IoC)

PRÉSENTATION D'ANGULAR

- Point d'entrée de l'application
 - Fichier *app/main.ts* (qui sera traduit en fichier JS)
- Single Page Application (SPA)
 - *index.html*

A decorative wavy line in light blue and white, flowing vertically along the left side of the slide.

INITIALISATION

INITIALISER UNE APPLICATION

INITIALISATION

- Installation de Angular CLI

```
npm install -g @angular/cli@latest
```

- Création d'un nouveau projet

```
ng new nom_projet
```

- Démarrage du projet

```
ng serve
```

```
ng s
```

- Transpilation des fichiers TS en fichiers JS
- Cette commande écoute la modification des fichiers
 - Lorsque les fichiers sont modifiés, ils seront retranspilés, et la page du navigateur sera rafraichie automatiquement
- Chargement des fichiers
 - index.html > main.ts > app.module.ts > app.component.ts

INITIALISATION

- Angular CLI permet d'initialiser rapidement une nouvelle application
- Mais permet aussi de générer des nouveaux composants, services, ...
- Toute la documentation : <https://github.com/angular/angular-cli/wiki>

A decorative wavy line in light blue and white, flowing vertically along the left edge of the slide.

FONDAMENTAUX

LES FONDAMENTAUX

FONDAMENTAUX

- Contenu du fichier *app.module.ts* (Module principal)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
```

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

On utilise le composant
AppComponent

Le module a besoin de
BrowserModule

On démarre avec
AppComponent

Définition de la classe
principale

FONDAMENTAUX

- Contenu du fichier *app.component.ts* (Composant principal)

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'demo';
}
```

Sélecteur HTML qu'on utilisera
pour appeler ce composant

Fichier HTML de la vue du
composant

Fichier(s) CSS de la vue du
composant

FONDAMENTAUX


- Dans **@NgModule**

- import Liste des modules requis pour le module en question
- declarations Liste des composants, directives, etc. utilisés par le module
- providers Liste des services utilisés par le module
- bootstrap Liste des composants utilisés pour démarrer l'application
 - On peut démarrer l'application de multiples façons ; au sein d'un navigateur, on va utiliser le bootstrapping Browser

FONDAMENTAUX

- Quant au contenu du body du fichier HTML *index.html*

```
<body>  
  <app-root></app-root>  
</body>
```



Sélecteur HTML déclaré dans
le composant

FONDAMENTAUX

- Contenu du fichier *main.ts*

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
  
import { AppModule } from './app/app.module';  
  
platformBrowserDynamic().bootstrapModule(AppModule)  
  .catch(err => console.error(err));
```

- Dans ce fichier, on indique qu'on exécute notre module principal, avec platform-browser-dynamic
- C'est pour cette raison que AppModule a besoin d'importer BrowserModule

EXERCICE

- Initialiser et démarrer le projet (avec Angular CLI)
- Tester et vérifier que le module s'exécute bien
 - Le terminal reste en attente d'une modification de fichier
 - L'adresse <http://localhost:4200/> est disponible

FONDAMENTAUX

- Angular mappe des données au HTML en utilisant des interpolations (Expression Language)
 - {{ expression }}
- Exemples

```
<p>{{ 5 + 5 }}</p>
```

```
<p>Le client est {{ prenom }} {{ nom }}</p>
```

```
<p>Le client est {{ client.prenom }} {{ client.nom }}</p>
```


EXERCICE

- Afficher un prénom dans le paragraphe
 - Le prénom est un attribut de classe du composant

EXERCICE

- Il est possible de lire une donnée avec les expressions (binding read)
- Dans la vue
 - Ajouter un input qui fait référence à *prenom*

```
<input type="text" [value]="prenom" />
```

- Quel est le résultat ?

EXERCICE

- Dans la vue
 - Ajouter un bouton qui appelle une fonction qui change la valeur de *prenom*
 - Créer cette fonction dans la classe AppComponent

```
<button (click)="resetPrenom()">CHANGER</button>
```

- Quel est le résultat ?

EXERCICE

- Dans la vue
 - Utiliser l'évènement « change » pour capter le changement de l'input
 - Envoyer « \$event » comme argument
 - Dans le composant, créer la fonction qui attend *event* (en type « any »)
 - `event.target.value` donnera la valeur de l'input
 - Changer la valeur de *prenom* par la valeur de l'input
- Quel est le résultat ?

FONDAMENTAUX

- Il est possible de lire une donnée avec les expressions (binding read)
- Il est également possible de les modifier (binding write)
 - Utilisation d'une directive *ngModel*
 - On aura besoin du module **FormsModule** dans le module principal

```
import { FormsModule } from '@angular/forms';
```

```
imports: [  
  BrowserModule,  
  FormsModule  
]
```

EXERCICE

- Tester ce code : dans le template HTML

```
<input type="text" [(ngModel)]="prenom" />
<p>{{ prenom }}</p>
```

dans le composant TS

```
export class AppComponent {
  prenom: string = "Jérémy";
}
```

- Que fait ngModel ?

FONDAMENTAUX

- *ngModel* lie les données du modèle à la vue, et inversement !
 - C'est ce qu'on appelle un *data-binding Two-Way*

```
<input type="text" [ngModel]="prenom" (ngModelChange)="prenom = $event" />
<p>{{ prenom }}</p>
```

- Angular s'appuie sur le pattern *MVVM* pour le binding
 - Comme vu dans l'exercice précédent
 - Chaque modification d'un côté entraîne une mise à jour de l'autre côté

FONDAMENTAUX

- Dans la vue des exercices précédents
 - Utilisation de
 - `{{ ... }}`
 - `[(...)]="..."`
 - `(...)= "..."`
 - Il en existe d'autres
 - `*...="..."`
 - `#...`
 - `[...] = "..."`
 - `bind-...="..."`
 - `bindon- ...="..."`

FONDAMENTAUX

Syntaxe	Direction de l'information	Type	Exemples
{{ expression }}	Sens unique vers la vue	Interpolation	{{ couleur }}
[attr]="expression" bind-attr="expression"	Sens unique vers la vue	Propriété	[style.color]="couleur" bind-style.color="couleur"
(evenement)="traitement" on-evenement="traitement"	Sens unique vers la source de données	Evènement	(click)="fonction()" on-dblclick="fonction()" (mouseenter)="fonction()"
[(attr)]="expression" bindon-attr="expression"	Dans les deux sens	Propriété Evènement	[(ngModel)]="couleur" bindon-ngModel="couleur"
*directive	Permet de modifier la structure HTML	-	*ngIf="true" *ngFor="let e of list" *ngSwitchCase="valeur"
#nom	Permet de nommer un élément HTML	-	#zoneDemo

EXERCICE

- Ajouter une liste de choix à la vue (select HTML)
 - Quelques couleurs (noir, vert, jaune, bleu)
- Ajouter un input de type « color »
- Lorsque l'utilisateur change la couleur
 - La couleur du paragraphe change en fonction de la sélection !
 - Utiliser uniquement ngModel

FONDAMENTAUX

- Dans un template, on peut nommer dynamiquement un élément HTML
 - En utilisant « # »

```
<input type="text" #prenom />  
<p>{{ prenom.value }}</p>
```



CLASSES

LES CLASSES ANGULAR

CLASSES

- Il est possible de fabriquer autant de classes que nécessaires
 - **Component**
 - **Directive**
 - **Pipe**
 - **Service**
 - Autres classes
 - Et pour ces dernières, il n'y a rien de spécial à faire, si ce n'est de créer le fichier ... ou de le générer

```
ng generate class produit
```

```
ng g class produit
```

EXERCICE

- Créer une classe « Todo »
 - id number
 - title string
 - completed boolean
 - userId number

EXERCICE

- Dans la vue
 - Ajouter des input / select pour la saisie des informations id, title, et completed
 - La paragraphe affiche le title
 - Et ne doit s'afficher que lorsque le title est rempli, et que l'id est différent de 0 !
 - Si le todo n'est pas complété, le background du paragraphe applique une classe CSS qui fait devenir le paragraphe rouge
 - `[class.nom-classe]` ou `[ngClass]="{ 'nom-class': true }"`



DIRECTIVES

LES DIRECTIVES ANGULAR

DIRECTIVES

- Une directive est une fonctionnalité front-end
 - Intégrée à un module
 - Il est possible de faire tout ce que peut faire un composant
 - Simplement, la directive n'a pas de vue associée
 - Certaines sont natives à Angular et sont préfixées par « ng »
 - **ngModel**, **ngClass**, **ngIf**, **ngFor**, **ngSwitch**, **ngSwitchCase**, ...

COMPONENTS

- Utilisation de Angular CLI pour générer une directive

```
ng generate directive helloWorld
```

```
ng g d helloWorld
```

- Angular CLI va
 - Générer les fichiers TS et SPEC.TS (pour les tests unitaires)
 - Ajouter dans le module principal la déclaration de la nouvelle directive

<https://angular.io/cli/generate#directive>

DIRECTIVES

- Chaque directive doit être dans un module, il faut donc
 - Importer cette directive dans le module
 - La déclarer dans la liste des déclarations
 - C'est déjà ce qu'a fait Angular CLI, si tout s'est bien passé et que le fichier du module n'était pas en cours de modification manuelle

DIRECTIVES

- Trois types de directives
 - Les composants
 - Les directives d'attributs
 - Modifient le comportement des éléments HTML et/ou de leurs attributs
 - Représentée généralement sous forme d'attribut HTML

```
<p hello-world>Le contenu</p>
```

- Les directives structurelles
 - Modifient la structure HTML des éléments sur lesquels elles s'appliquent
 - **ngIf** et **ngFor** par exemple

DIRECTIVES

- **ngClass**

- Permet d'appliquer une classe en fonction d'une condition

```
<p ng-class="{ 'une-classe': nom, 'attention': prix < 0 }">...</p>
```

DIRECTIVES

- **ngIf**
 - Afficher un élément en fonction d'une condition vérifiée

```
<p *ngIf="condition_vrai">  
  C'est vrai  
</p>
```

```
<p *ngIf="condition_vrai; else sinon">C'est vrai</p>  
<ng-template #sinon>  
  <p>C'est faux</p>  
</ng-template>
```

DIRECTIVES

- **ngFor**
 - Effectue une boucle for dans la vue
 - Se place sur l'élément à répéter

```
<ul>  
  <li *ngFor="let element of liste">{{ element }}</li>  
</ul>
```

DIRECTIVES

- **ngSwitch / ngSwitchCase / ngSwitchDefault**
 - Afficher un contenu selon la valeur d'une donnée

```
<div [ngSwitch]="type">  
  <p *ngSwitchCase="CLIENT">C'est un client</p>  
  <p *ngSwitchCase="FOURNISSEUR">C'est un fournisseur</p>  
  <p *ngSwitchDefault>- N'est pas pris en charge -</p>  
</div>
```


EXERCICE

- Créer une liste de Todos
 - Afficher cette liste dans un tableau HTML
 - Colonnes ID, Titre, Complet
 - Dans la colonne Complet
 - Si le todo n'est pas complet, afficher « Non terminé »
 - Si le todo est complet, afficher « Terminé ! »
- Créer un formulaire pour ajouter un nouveau todo
 - Ajouter un todo dans la liste des todos en cliquant sur un bouton « ajouter »
 - Utiliser `Array.prototype.push(object)`
 - Le tableau doit se mettre à jour automatiquement

EXERCICE

- Ajouter la possibilité de filtrer les todos sur leur title
 - Le filtre se traite dans le composant
 - Utiliser la syntaxe suivante pour vous aider

```
this.todos.filter(t => t.title.indexOf("valeur") !== -1);
```

- Après l'ajout d'un todo, le filtre doit continuer de s'appliquer

On notera que l'appel d'une fonction dans un Template n'est pas recommandé
> Même si la stratégie de détection des chargements améliore la situation



PIPES

TRANSFORMATION DES DONNÉES

PIPES

- Un pipe (ou filtre) est une fonctionnalité front
 - Intégré à un module
 - Réutilisable
- Permet d'afficher une donnée transformée, formatée
- Certains sont prévus par Angular
 - Transformer une date en date « Jour n mois année »
 - Transformer un chiffre en monnaie (euros, dollars, ...)
 - Mettre tout en majuscule / minuscule
 - ...

PIPES

- Utilisation de Angular CLI pour générer un composant

```
ng generate pipe monFiltre
```

```
ng g p monFiltre
```

- Angular CLI va
 - Générer les fichiers TS et SPEC.TS (pour les tests unitaires)
 - Ajouter dans le module principal la déclaration du nouveau pipe

<https://angular.io/cli/generate#pipe>

PIPES

- Chaque pipe doit être dans un module, il faut donc
 - Importer ce pipe dans le module
 - Le déclarer dans la liste des déclarations
 - C'est déjà ce qu'a fait Angular CLI, si tout s'est bien passé et que le fichier du module n'était pas en cours de modification manuelle

PIPES

- Les pipes sont à utiliser dans les templates
 - Il faut utiliser le caractère pipe « | » pour appliquer un filtre

```
<p>{{ prenom | uppercase }}</p>
```

- On peut ajouter des paramètres en utilisant les deux points « : »

```
<p>{{ prix | currency:'EUR' }}</p>
```

- Il est possible d'en enchaîner plusieurs à la suite : chaining pipes
 - La transformation s'applique dans l'ordre de lecture (gauche vers la droite)

```
<p>{{ dateAchat | date:"EEEE dd/MM/yyyy" | uppercase }}</p>
```

PIPES

- On peut aussi créer son propre Pipe ; dans le fichier *mon-filtre.pipe.ts*
 - On déclare le filtre, avec une classe, en le décorant de **@Pipe**

```
@Pipe({  
  name: 'monFiltre'  
})  
export class MonFiltrePipe implements PipeTransform {  
  transform(value: string, ...args: string[]): string {  
    return "valeur";  
  }  
}
```

Nom du pipe à utiliser dans les templates

PIPES

- Dans la classe du pipe
 - Il faut une méthode transform, qui attend au moins un argument
 - La donnée qui sera utilisée pour le formatage
 - Selon les besoins, on peut y ajouter plusieurs autres arguments
 - Qui seront les paramètres du pipe

```
@Pipe({
  name: 'monFiltre'
})
export class MonFiltrePipe implements PipeTransform {
  transform(value: number, ...args: string[]): string {
    return "valeur";
  }
}
```

```
<p>{{ prix | monFiltre:"valeur de argument 1" }}</p>
```

EXERCICE

- Créer un pipe **TodoFilterPipe** qui permet
 - De filtrer sur un champ de recherche
 - Attend la liste à filtrer, et la valeur de la recherche

EXERCICE

- Créer un pipe **TodoStatePipe** qui permet
 - D'afficher en couleur ou une chaîne de caractères

Toto complet	Paramètre	Valeur
Non	string	« Non terminé »
Oui	string	« Terminé ! »
Non	–	Couleur rouge hsl(341 79% 53%)
Oui	–	Couleur verte hsl(153 48% 49%)

- Si le paramètre reçu est "string", alors c'est le nom de la catégorie qui est retourné
- Sinon, c'est la couleur

```
<p [style.color]="todo.completed | todoState">{{ todo.completed | todoState:"string" }}</p>
```



COMPOSANTS

LES COMPOSANTS ANGULAR

COMPONENTS

- Un composant est une fonctionnalité front-end
 - Intégré à un module
 - Sépare la partie logique de la partie manipulation DOM
 - Réutilisable
- Quelques questions à se poser avant de se lancer
 - Quel est son rôle ?
 - Doit-il prendre place dans le vue HTML ? (avoir une structure HTML, une vue)
 - Quels sont ses paramètres / attributs d'entrée ?

COMPONENTS

- Utilisation de Angular CLI pour générer un composant

```
ng generate component helloWorld
```

```
ng g c helloWorld
```

- Angular CLI va
 - Générer les fichiers TS, HTML, CSS, et SPEC.TS (pour les tests unitaires)
 - Ajouter dans le module principal la déclaration du nouveau composant

<https://angular.io/cli/generate#component-command>

COMPONENTS

- Chaque composant doit être dans un module, il faut donc
 - Importer ce composant dans le module
 - Le déclarer dans la liste des déclarations
 - C'est déjà ce qu'a fait Angular CLI, si tout s'est bien passé et que le fichier du module n'était pas en cours de modification manuelle

COMPONENTS

- Dans le fichier *nom-composant.component.ts*
 - On déclare le composant, avec une classe, en la décorant de **@Component**

```
@Component({  
  selector: 'hello-world,[hello-world]',  
  templateUrl: './hello-world.component.html',  
  styleUrls: ['./hello-world.component.css']  
})  
export class HelloWorldComponent {  
  
}
```

On précise les sélecteurs qu'on utilisera en HTML
'nom-sélecteur' balise HTML
'[nom-sélecteur]' attribut HTML
Ici, on utilise 2 sélecteurs

```
<hello-world>...</hello-world>  
<p hello-world>...</p>
```


EXERCICE

- Créer un composant « asc-bold » :
 - Son rôle sera de mettre en gras un texte

COMPONENTS

- Dans le template du composant (*nom-composant.component.html*)
 - On définit la structure HTML
 - On peut inclure un contenu en utilisant la balise ng-content

```
<div> <!-- Conteneur du composant -->
  <p>Le contenu du composant</p>

  <!-- Contenu issu de la vue qui a appelé ce composant -->
  <ng-content></ng-content>
</div>
```

- « Démo » sera positionné à l'endroit où se trouve ng-content dans le composant

```
<hello-world>Démo</hello-world>
```

COMPONENTS

- C'est ce qu'on appelle le « Content Projection »
 - On peut déclarer plusieurs zones dans le composant et les remplacer par un sélecteur

```
<ng-content select=".classname"></ng-content>  
<ng-content select="[attrname]"></ng-content>
```

```
<div class="classname">Démo contenu nommé avec sélecteur classe</div>  
<div attrname>Démo contenu nommé avec sélecteur attribut</div>
```

```
<ng-container ngProjectAs=".classname">  
  Démo contenu nommé avec sélecteur classe ... projeté.  
</ng-container>
```

EXERCICE

- Modifier le composant « asc-bold »
 - Définir deux zones : une zone en gras, une autre en normal
 - Les templates sont envoyées depuis le parent vers l'enfant

COMPONENTS

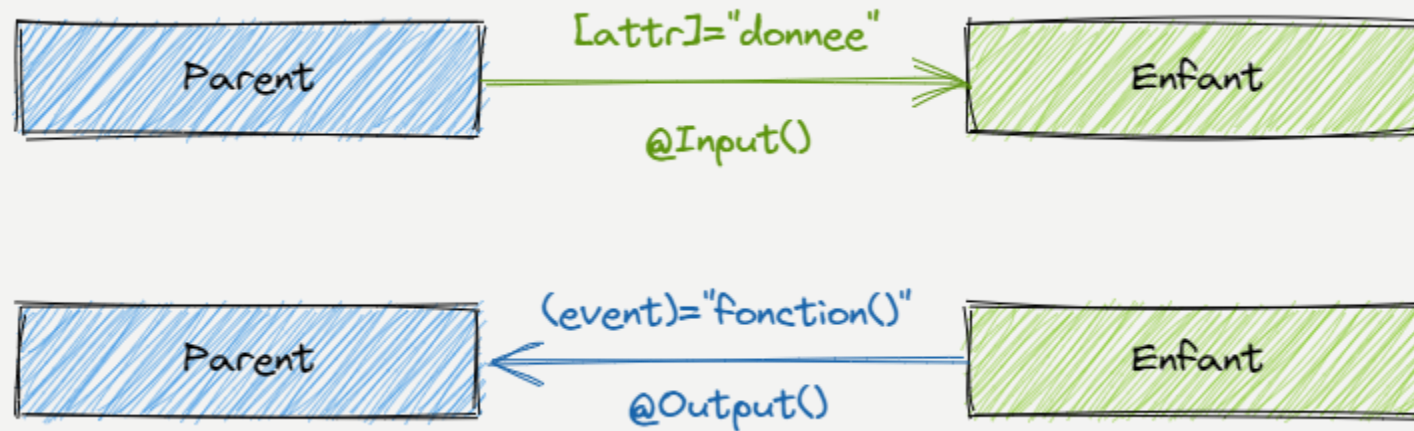
- Dans la classe du composant, pour binder une action utilisateur sur l'ensemble du composant
 - On peut utiliser (evenement) dans la vue
 - Ou utiliser le décorateur **@HostListener** sur les méthodes
 - Pour ça, il nous faut importer **HostListener**, de **@angular/core**

```
@HostListener('click')
onClick(): void {
  // TODO
}
```

EXERCICE

- Modifier le composant « asc-bold » :
 - Au clique, tenir à jour un compteur (initialisé à 0)
 - L'afficher dans la console

COMPONENTS



COMPONENTS

- Dans la classe du composant
 - On peut récupérer des informations en entrée, brutes ou bindées sur le modèle de données
 - Utiliser le décorateur **@Input**, de `@angular/core`
 - On place ce décorateur sur l'attribut à récupérer

```
@Input('titre') titre: string = "";  
@Input() text: string = "";
```

- En utilisant les principes de binding dans la vue
 - Sans les crochets Texte brute
 - Avec les crochets Binding vers une donnée existante dans le modèle

```
<hello-world titre="prenom" [text]="prenom">Démo</hello-world>
```


EXERCICE

- Créer un composant « asc-text-field » :
 - Un label
 - Un input de type text
 - L'inclure dans le composant principal par élément HTML
 - Le label est une information en entrée
 - La valeur de l'input est une information en entrée

COMPONENTS

- Dans la classe du composant
 - On peut déclencher des évènements
 - Utiliser le décorateur **@Output**, de `@angular/core`
 - On place ce décorateur sur l'attribut évènement qui pourra être émit

```
@Output() clicked = new EventEmitter<number>();
```

- Et au moment voulu, on émet l'évènement

```
this.clicked.emit(10);
```

COMPONENTS

- Dans la classe du composant
 - On peut intercepter un évènement
 - En utilisant les principes de binding dans la vue, avec les parenthèses

```
<hello-world (clicked)="onClicked($event)">Démo</hello-world>
```

```
onClicked(count: number): void {  
  alert(count)  
}
```

EXERCICE

- Modifier le composant « asc-text-field » :
 - Lorsque la valeur a été modifiée
 - Déclencher un évènement permettant de prévenir le parent du changement
 - Rappel : attrChange permettra un binding two-ways éventuel

COMPONENTS

- On peut aller encore plus loin sur les Content Projection avec **ngContainer** et **ngTemplate**

```
<ng-container [ngTemplateOutlet]="header"></ng-container>
```

```
@ContentChild('header') header!: TemplateRef<unknown>;
```

```
<ng-template #header>  
  <h1>Header de la page</h1>  
</ng-template>
```

COMPONENTS

- Et également envoyer des informations de l'enfant vers le parent

```
<ng-container [ngTemplateOutlet]="slotDemo"  
               [ngTemplateOutletContext]="{ keyDemo: demo }">  
</ng-container>
```

```
@ContentChild('slotDemo') slotDemo!: TemplateRef<unknown>;
```

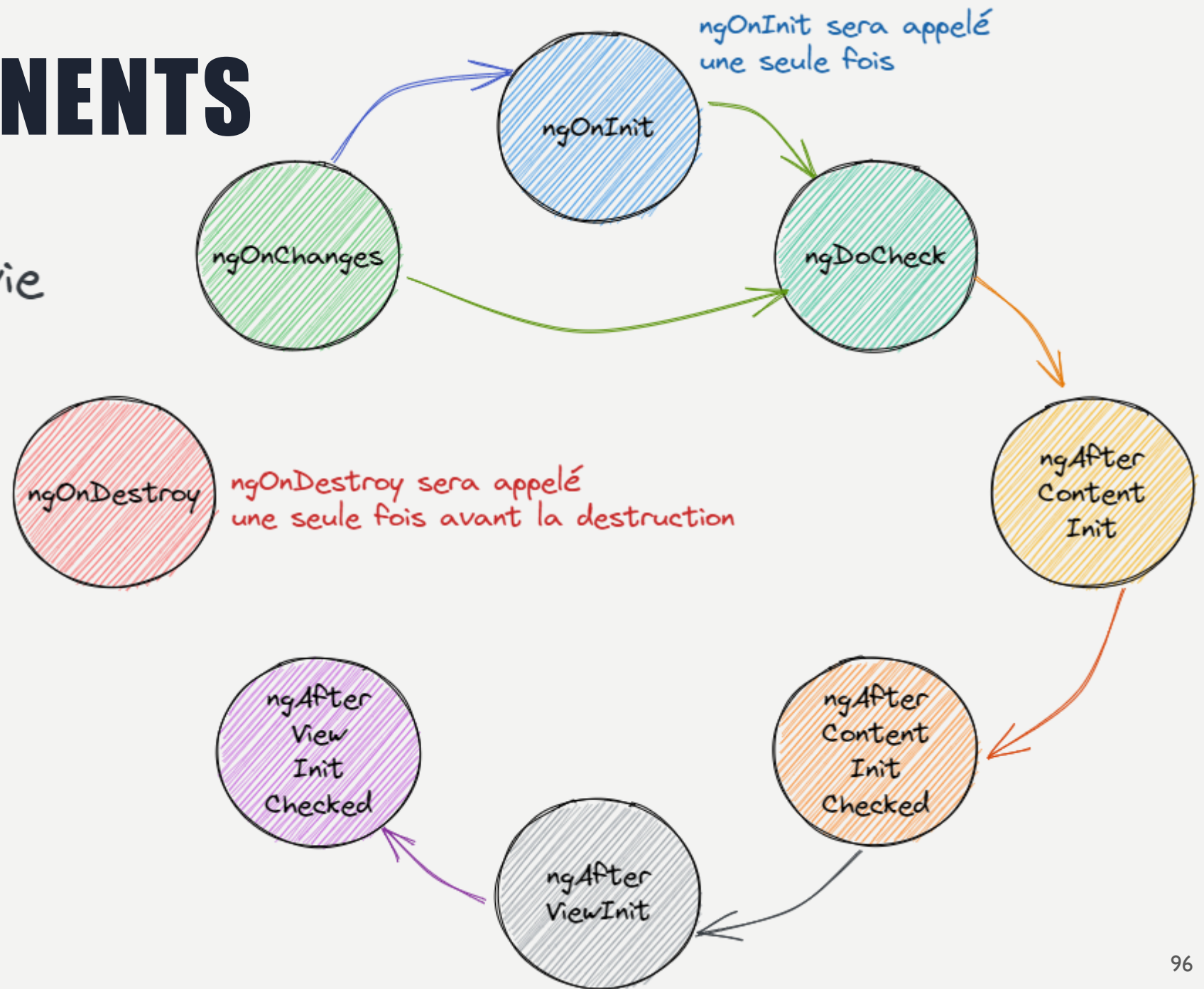
```
<ng-template #slotDemo let-demo="keyDemo">  
  <div>  
    {{ demo }}  
  </div>  
</ng-template>
```

EXERCICE

- Créer le composant « asc-list »
 - Attendre une liste de todos
 - Pour chaque todo, attendre un template, et renvoyer le todo à ce template

COMPONENTS

Cycle de vie



COMPONENTS

- Ces éléments de cycle de vie sont des *hook*
 - OnInit
 - OnChanges
 - OnDestroy
 - DoCheck
 - AfterViewInit / AfterViewChecked
 - AfterContentInit / AfterContentChecked

```
export class AppComponent implements OnInit {  
  ngOnInit(): void {  
    // TODO  
  }  
}
```

EXERCICE

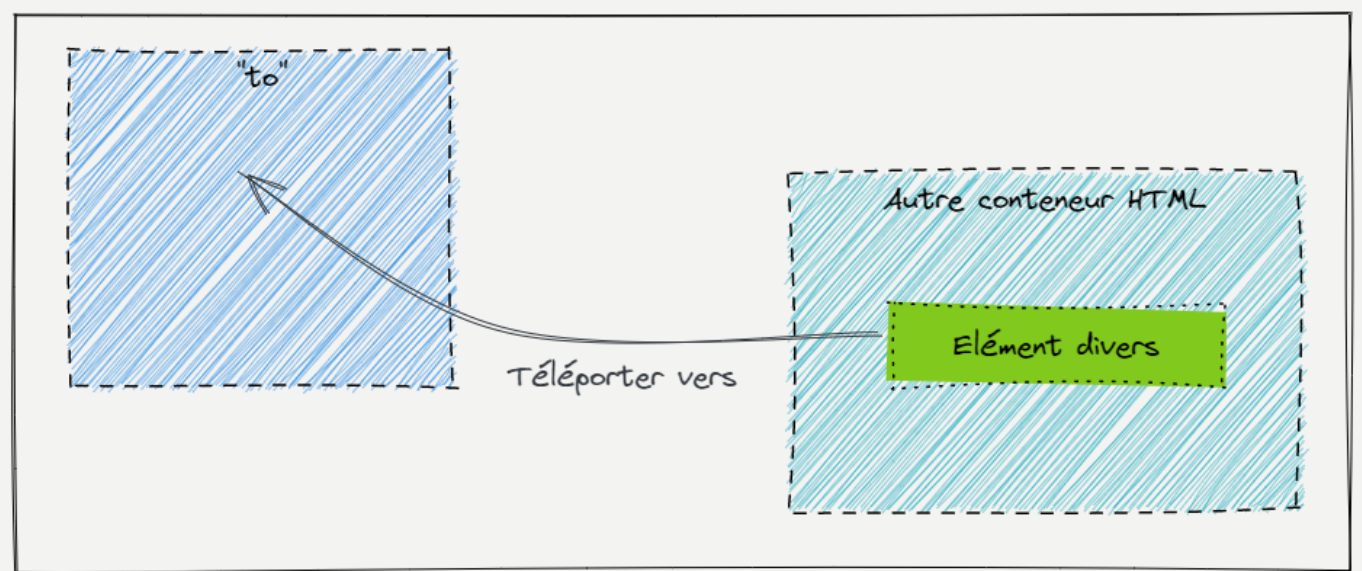
- Créer un composant « navigation » dont le but sera d'afficher la navigation de l'application
 - Lien Accueil
 - Lien Liste des todos
 - A l'initialisation, afficher un message dans la console

EXERCICE

- Créer un composant qui affiche un tooltip au survol de la souris
 - CSS
 - bordure arrondie 4px
 - couleur rose #FF5588
 - padding 10px 20px

```
<p asc-tooltip titre="Et voilà un tooltip !">Voilà un message !</p>
```

EXERCICE



- Créer une directive « teleport »
 - Injecter **ElementRef** et **Renderer2**
 - Attendre une valeur « to », le sélecteur vers lequel téléporter l'élément
 - Au chargement
 - Récupérer l'élément HTML « to »
 - Insérer l'élément cible vers l'élément « to »

```
to?.append(this.elementRef.nativeElement);
```

- Au déchargement
 - Supprimer le contenu téléporté

```
this.renderer.setProperty(to, 'innerHTML', "");
```