Enhance Web Development

Angular 2

Les composants fondamentaux

- Les classes
- Les imports/exports
- Les décorateurs
- L'amélioration (composition)
- La répétition
- Le cycle de vie Angular 2

Class !== inheritance

Classes

- Créer un composant (classe ES6)
- Les propriétés et les méthodes de notre classe seront disponibles dans la vue

```
export class UsersComponent() { }
```

Notre composant

Import

- Importer les dépendances angular
- Importer les dépendances 3rd party
- Importer les dépendances du projet
- Très utile d'ordonner ses imports pour améliorer la lecture du code source

```
import (Component) from 'angular/core';

export class UsersComponent() { }
```

Les import

Decorate

- Pour transformer une classes ES6 en un artefact angular 2, on la décore avec les metadata spécifiques Angular 2.
- On utilise la syntaxe @<decorator>
- Les plus utilisés sont : @Component, @Injectable, @Directive et @Pipe
- On peut également décorer les propriétés : @Input et @Output

```
import (Component) from 'angular/core';
@Component({
    selector: 'users',
    templateUrl: './users.component.html'
})
export class UsersComponent() { }
```

Le décorateur @Component

Ameliorer (Enhance)

- L'idée est d'écrire une version minimale du composant puis de l'améliorer de façon itérative
- Améliorer par composition = ajout de méthodes, d'input, d'output, injecter des services, etc.
- un composant = simple et spécifique

```
import (Component) from 'angular/core';

export interface User {
    firstname: string
    lastname: string
}

@Component({
    selector: 'users',
    templateUrl: './users.component.html'
})
export class UsersComponent() {
    users: User[]
    defaultUser: User = {
        firstname: 'Default Firstname',
        lastname: 'Default Lastname',
    }
    constructor(
        private _userService: UserService
    ) {
    updateUser(index:number, user: User) {
        this.users = this._userService.update(index, user)
    }
`
```

Ajout de fonctionnalité à notre composant

Répéter

- La plateforme Angular 2 permet de créer très facilement un composant
- Découper un composant en plusieurs composant est très vivement recommandé

```
import (Component) from 'angular/core';

import {UserComponent} from './user/user.component'

export interface User {
    firstname: string
    lastname: string
}

@Component({
    selector: 'users',
    templateUrl: './users.component.html',
    directives: [UserComponent]
})
export class UsersComponent() {
    users: User[]
    defaultUser: User = {
        firstname: 'Default Firstname',
        lastname: 'Default Lastname',
    }
    constructor(
        private _userService: UserService
    ) {
    updateUser(index:number, user: User) {
        this.users = this._userService.update(index, user)
    }
}
```

Inclusion d'un sous-composant

Cycle de vie Angular 2

Les hooks

- Ils permettent d'exécuter de la logique personnalisé à chaque étape de vie d'un composant
- Pourquoi ? Les données ne sont pas toujours disponible immédiatement dans le constructeur
- Utilisable seulement avec TypeScript
- Les interfaces des hooks sont optionnels (comme tout typage) mais fortement recommandé
- Hooks = methodes de notre composant

Liste des hooks

- **ngOnInit** - Appeler directement après la mise en place du binding
- **ngOnChanges(changes)** - Appeler quand un Input change
- **ngDoCheck** - Permet d'effectuer du code personnalisé
- **ngAfterContentInit** - Après que le contenu soit initialisé
- **ngAfterContentChecked** - Après chaque vérification du composant par Angular
- **ngAfterViewInit** - Après que la vue soit initialisé
- **ngAfterViewChecked** - Après vérification de la vue
- **ngOnDestroy** - Juste avant la suppression de ce composant

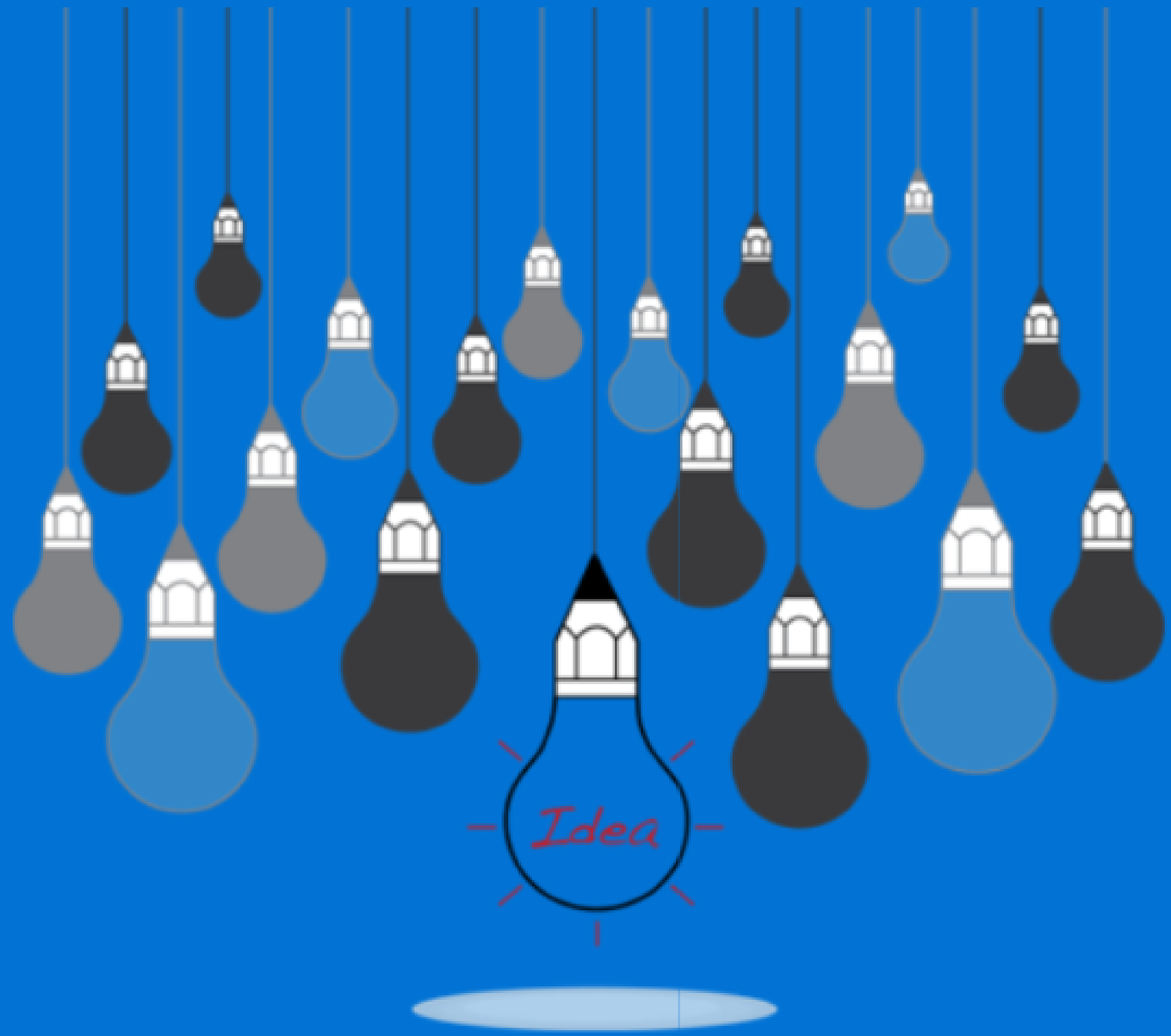
```
import (Component, OnInit) from 'angular/core';

import {UserComponent} from './user/user.component'

@Component({
    selector: 'users',
    templateUrl: './users.component.html',
    directives: [UserComponent]
})
export class UsersComponent implements OnInit () {
    users: User[]
    defaultUser: User = {
        firstname: 'Default Firstname',
        lastname: 'Default Lastname',
    }
    constructor(
        private _userService: UserService
    ) {
    ngOnInit() {
        this.users = this._userService.get()
    }
    updateUser(index:number, user: User) {
        this.users = this._userService.update(index, user)
    }
}
```

onInit hook

Demo time !



Défi

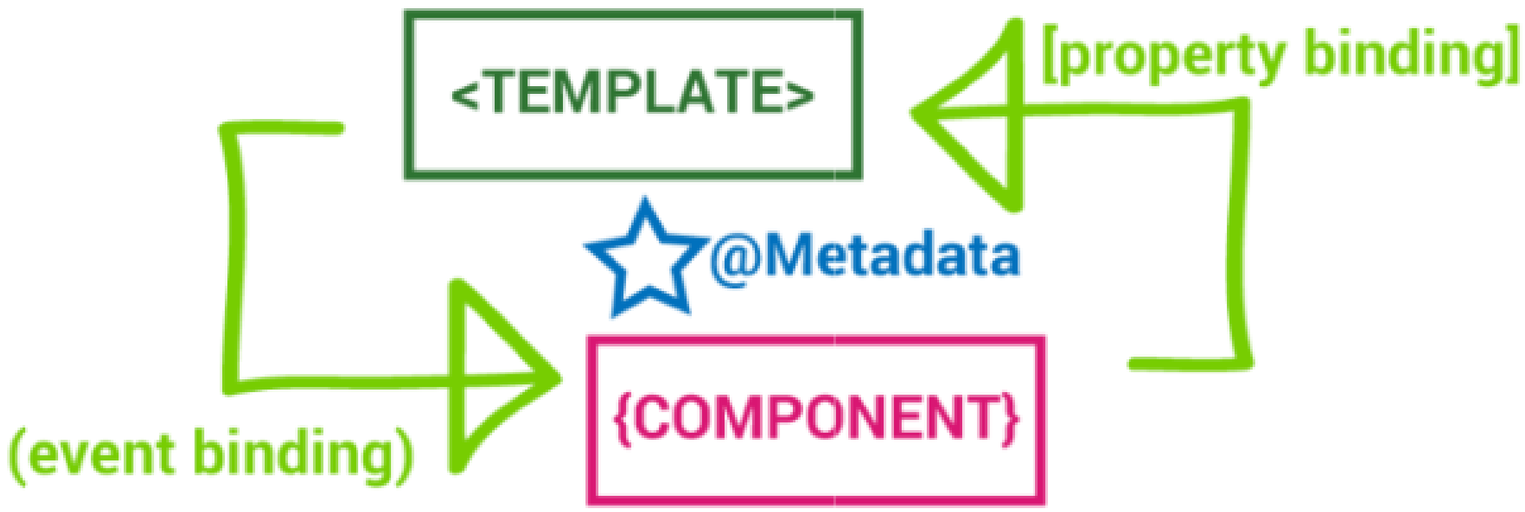
- Créer la structure de fichier pour un composant **widget**
- Créer la classe ES6 **widget**
- Importer les dépendances
- Décorer la classe **widget** pour utiliser un template & un sélecteur
- Afficher le composant **widget** dans le composant home
- BONUS : Créer une simple route pour se rendre sur le composant **widget**

Templates

[angular.io]

- Interpolation
- Method binding
- Property Binding
- Two-way bingind
- Hashtag Operator (#)
- Asterisk Operator (*)
- Elvis Operator (?)
- Web component & Native element
- Pourquoi ce fonctionnement ?

Templates


<TEMPLATE>
[property binding]
@Metadata
{COMPONENT}
(event binding)

<TEMPLATE>

{{value}}

[property] = “value”

(event) = “handler”

[(ngModel)] = “property”

{COMPONENT}

Interpolation

- Nous permet d'afficher une variable de notre composant, dans la vue associée
- Utilise la syntaxe {{ expression }}
- On peut appeler des méthodes, des propriétés, faire des maths, etc.

```
<span>{{interpolatedValue + getValue() + 4}}</span>
```

Interpolation

Property binding

- Permet de propager des données du composant vers l'élément
- Utilise la syntaxe ****
- Autre syntaxe : bind-attribute ****

Property binding

Ne jamais utiliser les [] si :

- La cible accepte les strings
- La valeur est fixe
- Cette valeur initiale ne change jamais

```
<img [src]="logo.jpg" />
<img [attr.src]="logo.jpg" />
<span [style.color]="componentStyle">Some colored text!</span>
```

Property bindings

Event binding

- Permet de propager des données de l'élément vers le composant
- Utilise la syntaxe **<button (click)="foo()"></button>**
- Autre syntaxe : on-attribute **<button on-src="foo()"></button>**
- Récupérer l'évènement avec $event : **<button (click)="foo($event)"></button>**

```
<button (click)="alertTheWorld()">Click me!</button>
```

Event bindings

Two-way binding

- La combinaison d'un event binding et d'un property binding
- Utiliser avec **ngModel** : [(ngModel)]

```
<md-input-container>
<label>The awesome input</label>
<input md-input [(ngModel)]="dynamicValue"
placeholder="Watch the text update!" type="text">  </md-input-container>
<br>
<span>{{dynamicValue}}</span>
```

Two-way bindings

Asterisk operator

- L'asterisk indique une directive qui modifie le code HTML
- C'est du "syntactic sugar" pour éviter de devoir utiliser un template

```
<div *ngIf=”userIsVisible”>{{user.name}}</div>
<template [ngIf]="userIsVisible">
    <div>{{user.name}}</div>
</template>
```

Asterisk operator

Hashtag operator

- Le hashtag (#) initialise une variable locale dans notre template
- Cette variable est disponible sur les éléments frère et enfant de l'élément qui porte la déclaration
- Pour l'utiliser : comme une variable venant du composant !

```
<img src="logo.jpg" #myImage />
{{myImage.src}}
```

Hashtag operator

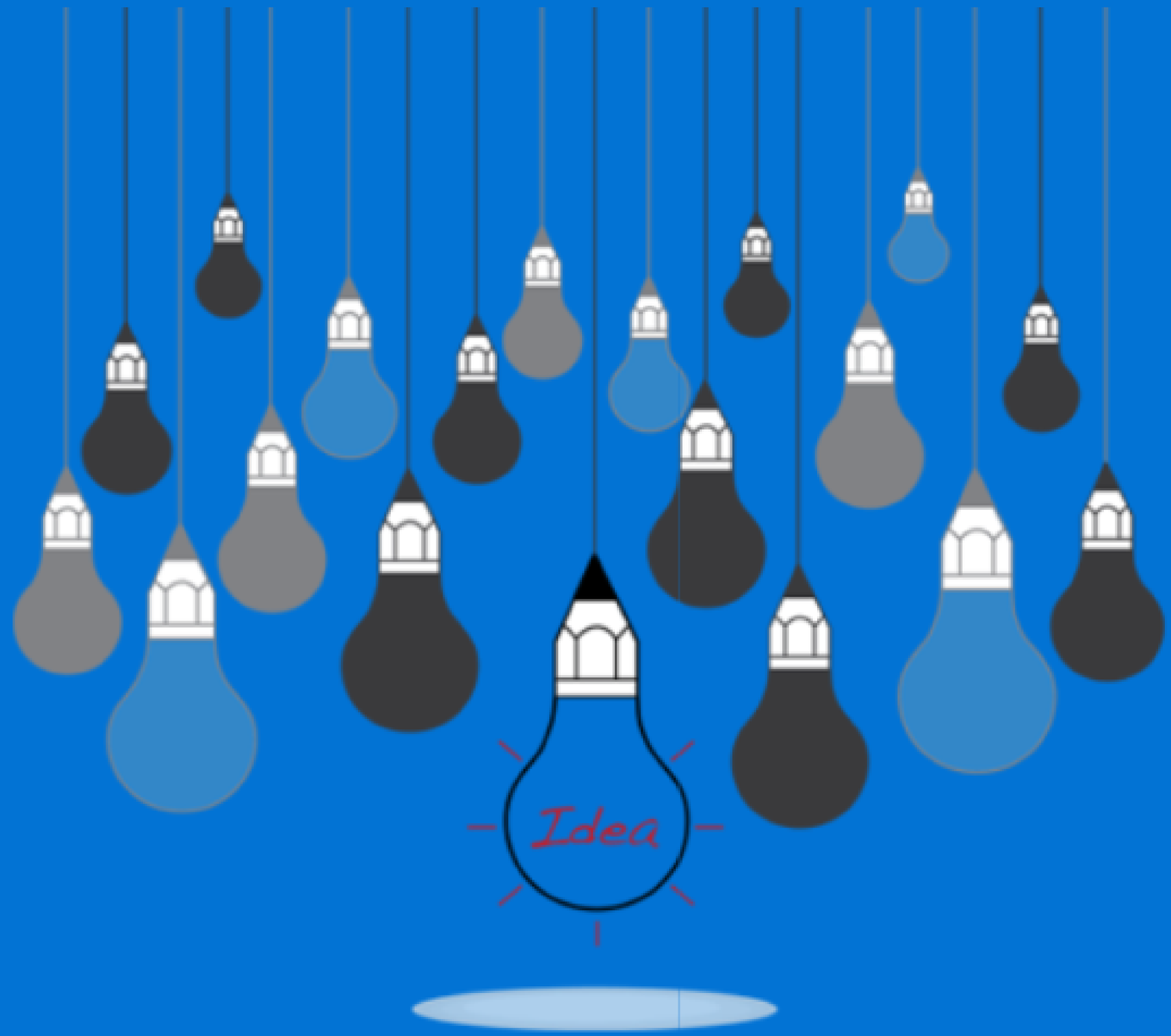
Elvis operator

- Le elvis operator (**?.**) est écrit avec un point d'interrogation directement suivit d'un point
- Evite de lever une exception si la valeur demandé n'existe pas
- Défenseur avéré contre les *null* et *undefined*

```
<md-input-container>
<label>Type to see the value</label>  <input md-input type="text" #input />
</md-input-container>
<strong>{{input?.value}}</strong>
```

Elvis operator

Demo time !



Défi

- Reprendre le **widget** et rajouter :
  - Une interpolation
  - Une property binding
  - Un event binding
  - Un two-way binding
- **BONUS** : Utiliser un hashtag (#), une directive native (*ngFor, *ngIf, etc.) et un elvis operator (?.) avec un **setTimeout**