Chapitre 4 Introduction à la notion de décorateurs

Nous allons tout d'abord faire un peu de configuration. Créez un dossier chap4. Créez un fichier tsconfig.json, le dossier App et dist puis définissez les paramètres suivants dans le fichier tsconfig.json :

```
{
    "compilerOptions": {
      "outDir": "./dist", // dossier de compilation
      "rootDir": "./App", // définit le dossier des fichiers ts
      "experimentalDecorators": true, // définit le support expérimental des décorateurs
      "target": "es5", // cible
      "module" : "commonjs"
    },
    "exclude" : ["node_modules"] // exclure le dossier node_modules de la compilation
  }
```

Vous l'avez deviné ce fichier est un fichier propre à TypeScript et sa configuration.

Exercice 7
Mettez en place cette configuration et créez un fichier **app.ts.** Puis testez la compilation à l'aide de la commande suivante : **tsc**
TypeScript va utiliser le fichier **tsconfig.json** pour récupérer les paramètres de compilation. Notez que votre fichier compilé se trouve dans le dossier « dist ».

Le paramètre **module** permet de préciser que l'on souhaite utiliser ce support pour la gestion des modules.

Le paramètre experimentalDecorators à true permet l'utilisation des décorateurs en TypeScript.

Exécutez la commande ci-dessous. L'option -w permet de watcher le code en continu et de compiler toutes vos modifications sans relancer celle-ci :

```
tsc -w
```

Remarques : pour la suite des exercices/applications utilisez cette manière de vous organiser avec ce fichier de configuration TypeScript et dossiers.

Les décorateurs
Les décorateurs sont implémentés dans le TypeScript directement, nous les verrons également dans Angular. Pour le standard ES6, ils sont en mode « expérimental », mais très certainement dans la version ES7. Ils seront intégrés de manière native. Quoiqu'il en soit on ne peut pas faire l'impasse de ce dernier concept car Angular l'utilise vraiment partout !

Rôle d'un décorateur
Un décorateur permet de modifier le comportement d'une classe et/ou de ses propriétés.
Dans le code suivant le décorateur **@Component** déclare la classe **AppComponent** comme étant un « composant » d'Angular et modifie les propriétés selector, templateUrl et styleUrls de la classe AppComponent :

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

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

Exercice/Application 8.1

Vous testerez ces exemples dans un dossier Ex8

Exemple de surcharge d'une propriété en écriture d'une méthode. La méthode speed() de la classe Bike est protégée en écriture par un décorateur (writable vaut « false ») :

```
// Exercice 8.1
function readonly(target, key, descriptor) {
    descriptor.writable = false; // modifier la visibilité à true si vous voulez redéfinir la méthode

    return descriptor;
  }

  class Bike {
    @readonly
    speed () {
      return 8;
    }
  }

  let bike = new Bike;

  bike.speed = () => 5 ; // on tente de redéfinir la méthode impossible car writable à false

  // la méthode retournera 8 car elle n'est accessible qu'en lecture
  console.log(bike.speed());
```

On peut également modifier le comportement d'une classe de ses propriétés et méthodes.

Exercice/Application 8.2

Voici un exemple de décorateur callable. Il redéfinit la méthode model de la classe ci-dessous :

```
function readonly(target, key, descriptor) {
    descriptor.writable = false; // modifier la visibilité à true si vous voulez redéfinir la méthode

    return descriptor;
}

// decorateur callable avec passage de paramètres
function modify(model: string) {
    let newModel = model;

    return function (target, key, descriptor) {
        descriptor.value = () => newModel;
    };
}

class Bike {
    @readonly
    speed() {
        return 8;
    }
    @modify('electric')
    model() { return 'normal'; }
}

let bike = new Bike;
bike.speed = () => 5; // on tente de redéfinir la méthode impossible car writable à false

// la méthode retournera 8 car elle n'est accessible qu'en lecture
console.log(bike.speed());
console.log(bike.model()); // modification du modèle
```

Décorateur de classe avec paramètres

Imaginons un code comme suit avec un décorateur qui modifie le comportement de la classe Duck en lui ajoutant une méthode swim() :

```
@Feature({
    action: "Nage comme un canard"
})
class Duck {
    say() { return "Je suis un oiseau"; }
}

let duck = new Duck;
console.log(duck.swim());
```

Voyez le code à écrire pour définir ce comportement :

```
function Feature(config) {

    return function (target) {
        // Permet de définir une nouvelle propriété
        Object.defineProperty(
            target.prototype, // Nom de la classe Duck
            'swim', // nom de la fonction que l'on ajoute
            // le descripteur on définit ici la méthode elle-même
            {
                value: () => config.action,
            }
        );
    }
}
```

Les décorateurs sont un sujet assez technique en soi. Il faudra pour Angular retenir l'idée qu'ils permettent de « modifier » ou « surchager » le comportement de la classe initiale.

On le voit également un décorateur permet d'agir sur des méthodes ou attributs de la classe elle-même.