Chapitre 6
Les décorateurs

1. Introduction

Un décorateur permet d'ajouter un comportement à un élément lors de l'exécution du code. Les décorateurs prennent la forme de fonctions qui peuvent être par la suite exécutées en utilisant une expression. Celle-ci commence avec le caractère @, suivi du nom du décorateur à exécuter.

Syntaxe :

```
@DecoratorName
```

Remarque

Il n'existe pas de convention de nommage actée pour les décorateurs. Toutefois, la convention de nommage la plus communément utilisée est celle dite "PascalCase". Dans ce chapitre, c'est la convention qui sera utilisée dans les exemples.

Les décorateurs sont souvent utilisés pour appliquer des aspects techniques sur les objets, comme par exemple :

– Logger l'exécution d'une méthode.
– Injecter des dépendances.
– Notifier le changement de la valeur d'une propriété.

L'autre intérêt des décorateurs est qu'ils permettent d'ajouter des métadonnées. Celles-ci permettent de préciser des informations supplémentaires sur un élément. Elles offrent aussi la possibilité aux développeurs d'écrire du code de manière plus déclarative. Cette utilisation des décorateurs est très courante et il existe de nombreux Frameworks et bibliothèques les mettant en œuvre pour différents besoins, comme par exemple :

– Définir le verbe HTTP lié à une action d'un contrôleur (Framework/Bibliothèque de type Web MVC).
– Définir la structure d'une table en base de données (Framework/Bibliothèque de type Object-relational Mapping, ou Mapping objet-relationnel en français).
– Définir les noms de propriétés lors de la conversion d'un objet en JSON (Framework/Bibliothèque de type Parser).

Remarque

La plupart des exemples cités ci-dessus seront mis en œuvre dans le code de ce chapitre et lors du TP (cf. chapitre Un premier projet avec Node.js).

L'implémentation des décorateurs dans JavaScript est une proposition de longue date qui n'est pas encore standardisée par ECMAScript. Le concept a été introduit dans la version 1.5 de TypeScript. Cependant, les décorateurs étant considérés comme expérimentaux, le langage ne permet pas de les utiliser par défaut. Il est donc nécessaire d'activer les options de compilation dédiées lors de l'initialisation du fichier `tsconfig.json`. Lors de l'utilisation de la commande `tsc --init`, il faut ajouter les arguments `--experimentalDecorators` (qui permet d'activer l'utilisation des décorateurs dans TypeScript) et `--emitDecoratorMetadata` (qui permet d'injecter dans le code, lors de la compilation, les métadonnées autour des types. cf. section Métadonnées) pour activer le support complet des décorateurs dans TypeScript.

Exemple :

```
tsc --init --experimentalDecorators --emitDecoratorMetadata
```

Les deux options de compilation portent le même nom que les arguments dans le fichier `tsconfig.json`.

Exemple :

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

Remarque

Par défaut, lors de l'initialisation, ces deux options sont disponibles dans le fichier `tsconfig.json`, mais commentées.

Les décorateurs sont fortement liés à la programmation orientée objet et peuvent être appliqués aux :

– Classes
– Méthodes
– Propriétés
– Accesseurs
– Paramètres

Les décorateurs ont un ordre d'exécution qui s'applique de la dernière déclaration vers la première (on parle d'exécution *bottom to top*).

Remarque

Attention, un décorateur ne doit pas dépendre d'un ordre d'exécution. La dépendance d'un décorateur à l'exécution d'un autre est considérée comme une mauvaise pratique, car cela complexifie la maintenance d'un programme.

2. Décorateur de classe

Une fonction peut être utilisée en tant que décorateur de classe à partir du moment où son type correspond à celui de `ClassDecorator`. Ce type est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
declare type ClassDecorator = <TFunction extends Function>(
  target: Tfunction
) => TFunction | void;
```

Remarque

Le mot-clé `type` est utilisé en TypeScript pour définir des alias de type. Il sera abordé en détail dans la suite de cet ouvrage (cf. chapitre Système de types avancés). Le mot-clé `declare` est quant à lui utilisé dans les fichiers de définitions pour définir et typer un élément afin qu'il puisse être utilisé par la suite dans du code TypeScript.

Ce type définit que la fonction décoratrice accepte un paramètre dont le type est contraint, via l'utilisation d'un générique, à étendre celui de `Function`. Le paramètre `target` permet de récupérer le constructeur de la classe.

Exemple :

```
const LogClassName: ClassDecorator = target => {
  console.log(target.name);
};

// Log: Person
@LogClassName
class Person {
  constructor(
    public readonly firstName: string,
    public readonly lastName: string
  ) {}
}
```

Un décorateur étant une fonction, il est possible d'exécuter celle-ci directement avec la classe en paramètre.

Exemple :

```
const LogClassName: ClassDecorator = target => {
  console.log(target.name);
};

// Log: Person
LogClassName(
  class Person {
    constructor(
      public readonly firstName: string,
      public readonly lastName: string
    ) {}
  }
);
```

Remarque

Tous les types décorateurs peuvent être exécutés de cette manière. Dans la suite de ce chapitre, seule la syntaxe avec expression sera utilisée.

Lorsqu'une classe est décorée, il faut ensuite l'importer pour que le décorateur s'exécute (cf. chapitre Les modules). Si un élément est décoré, mais qu'il n'est importé dans aucun module, il est tout de même possible d'exécuter le décorateur en effectuant un import direct du module. Ce type d'import est utilisé pour déclencher un effet de bord dans le programme sans pour autant récupérer les éléments exportés par le module.

Exemple (person.ts) :

```
import "./person";
```

3. Décorateur de méthode

Une fonction peut être utilisée en tant que décorateur de méthode à partir du moment où son type correspond à celui de `MethodDecorator`. Ce type est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
declare type MethodDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> | void;
```

Ce type définit que le décorateur accepte trois paramètres en entrée :

– `target` : l'objet dans lequel la propriété est contenue.

– `propertyKey` : la méthode qui a été décorée.

– `descriptor` : le descripteur de propriété sous la forme d'une instance du type `TypedPropertyDescriptor<T>`.

Remarque

Les descripteurs de propriété ont déjà été abordés lors de l'explication sur les boucles (cf. chapitre Types et instructions basiques).

Le type `TypedPropertyDescriptor<T>` est défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
interface TypedPropertyDescriptor<T> {
  enumerable?: boolean;
  configurable?: boolean;
  writable?: boolean;
  value?: T;
  get?: () => T;
  set?: (value: T) => void;
}
```

Il existe une version simplifiée de ce type qui utilise `any` à la place du type générique `T : PropertyDescriptor`. Ce type est lui aussi défini dans le fichier de déclaration de base de TypeScript (`lib.d.ts`) :

```
interface PropertyDescriptor {
    configurable?: boolean;
    enumerable?: boolean;
    value?: any;
    writable?: boolean;
    get?(): any;
    set?(v: any): void;
}
```

Remarque

Dans la section sur les boucles du chapitre Types et instructions basiques, le premier exemple montre comment définir une propriété sur un objet via l'utilisation de : `Object.defineProperty`. Le troisième paramètre attendu par la méthode `defineProperty` est typé avec l'interface `PropertyDescriptor`. Via la propriété `value` définie par cette interface, il est aussi possible de définir une méthode. C'est pour cela que les décorateurs de méthode s'appuient sur les descripteurs de propriété. De par leur nature, les décorateurs de méthode peuvent donc aussi être appliqués aux accesseurs.

Les décorateurs de méthode sont utiles pour encapsuler l'exécution d'une méthode afin de lui ajouter d'autres comportements (pouvant être appliqués avant ou après l'exécution de la méthode d'origine). Pour cela, il est nécessaire de redéfinir la méthode d'origine contenue dans la propriété `value` du paramètre `descriptor` avec une nouvelle fonction.