



Ressourcesinformatiques

 + QUIZ

Version en ligne

OFFERTE !

pendant 1 an

TypeScript

Notions fondamentales

En téléchargement



le code source



Felix
BILLON
Sylvain
PONTOREAU





Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence ENI de l'ouvrage **RITYP** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1 Introduction

- 1. Un peu d'histoire. 9
- 2. ECMAScript 13
- 3. Pourquoi TypeScript? 17
 - 3.1 Transpilation 18
 - 3.2 Typage statique 21
- 4. Les dessous du typage. 23
 - 4.1 Les fichiers de définition 23
 - 4.2 Typage structurel 24
- 5. L'architecture de TypeScript 27
 - 5.1 Core TypeScript Compiler. 28
 - 5.2 Standalone TS Compiler 28
 - 5.3 Language Service. 28
 - 5.4 Options de compilation 29
- 6. Environnement de développement 30
 - 6.1 Node.js. 30
 - 6.2 Visual Studio Code 32
 - 6.3 Installer TypeScript 38

Chapitre 2

Types et instructions basiques

1. Variable et portée	41
2. Types basiques	45
2.1 Introduction	45
2.2 Types primitifs basiques	45
2.3 Types primitifs : undefined et null	49
2.4 Types par référence : array et tuples	52
2.5 Types par référence : object	54
2.6 Types spéciaux : any et void	56
2.7 Affirmation de type	58
2.8 Type spécial : unknown	59
3. Décomposition	62
4. Énumération	65
5. Conditionnelle	72
6. Opérateurs	74
6.1 Opérateurs arithmétiques	74
6.2 Opérateurs unaires	75
6.3 Opérateurs de comparaison	76
6.4 Opérateurs logiques combinatoires	76
7. Zoom sur les opérateurs	77
7.1 Opérateurs unaires + et -	77
7.2 Opérateur unaire !	78
7.3 Opérateurs == et ===	78
7.4 Opérateurs et &&	79
7.5 Opérateur de décomposition	81
8. Boucles	82
8.1 Boucles for...in	83
8.2 Boucle for...of	84
9. Symbol	85

10. Itération et collections	90
10.1 Iterator	90
10.2 Générateurs	92
10.3 Map et Set	92
10.3.1Map	92
10.3.2Set	93
10.3.3WeakMap et WeakSet	94
11. Fonctions	95
11.1 Les bases	95
11.2 Paramètre rest	97
11.3 Les closures	98
11.4 This et les fonctions fléchées	100
11.5 Type fonction	104
12. Prototype	105
13. Gestion des exceptions	106

Chapitre 3

La programmation orientée objet

1. Introduction	109
2. Les classes	110
3. Les propriétés	113
4. Les méthodes	116
5. Les constructeurs	122
6. Le statisme	124
7. L'accessibilité des membres	126
8. L'encapsulation	132
9. L'héritage	135
10. L'abstraction	142
11. Les interfaces	145

12. Le polymorphisme	156
13. Les principes SOLID	158
13.1 Introduction aux principes SOLID	158
13.2 Le principe de responsabilité unique	158
13.3 Le principe d'ouverture à l'extension, fermeture à la modification	161
13.4 Le principe de substitution de Liskov	164
13.5 Le principe de ségrégation des interfaces	169
13.6 Le principe d'inversion des dépendances	171

Chapitre 4

Les modules

1. Introduction	175
2. Historique	176
2.1 Pattern module	176
2.2 AMD et CommonJs	178
2.3 Standardisation	183
3. La norme ECMAScript 2015	183
3.1 Export	187
3.2 Import	191
4. Chargement et résolutions	194
4.1 Résolutions	194
4.2 Chargement	197

Chapitre 5

La généricité

1. Introduction	199
2. Déclaration de base	199
3. Classes et interfaces	202
4. Contraintes	205

5. Générique et constructeur 211

Chapitre 6
Les décorateurs

1. Introduction 213
2. Décorateur de classe 215
3. Décorateur de méthode 217
4. Décorateur de propriété 220
5. Décorateur de paramètre 225
6. Fabrique de décorateurs 227
7. Métadonnées..... 229

Chapitre 7
Asynchronisme

1. Introduction 239
2. Callback hell 240
3. Promesse 242
 3.1 Historique 242
 3.2 Promesses ECMAScript 2015..... 243
 3.2.1 Résolution des promesses..... 243
 3.2.2 Then, catch et finally 245
 3.2.3 Promise.all et Promise.race..... 248
 3.2.4 Conclusion 249
4. Async/await 250

Chapitre 8

Système de types avancés

1. Introduction	255
2. Alias de type	255
3. Type never	259
4. Type union et intersection	261
4.1 Union	261
4.2 Intersection	263
5. Type guards	266
5.1 Introduction	266
5.2 Opérateur typeof	266
5.3 Opérateur instanceof	268
5.4 Opérateur in	270
5.5 Union discriminante	271
5.6 Type guards définis par l'utilisateur	273
5.7 Gestion de null et undefined	276
5.8 Cas impossible	279
6. Types littéraux	280
7. Type index	282
8. Mapped type	285
8.1 Avant TypeScript 2.1	285
8.2 Les opérateurs	287
8.3 Mapped type natif	289
9. Affirmation constante	292
10. Type conditionnel	294
10.1 Les bases	294
10.2 Distributivité	296
10.3 Opérateur infer	298
10.4 Type conditionnel natif	299

Chapitre 9
TypeScript et la programmation fonctionnelle

- 1. Introduction 303
- 2. Fonction pure et impure. 306
- 3. Immutabilité. 309
- 4. Itération. 316
- 5. Conditions. 320
- 6. Fonction partielle 321
- 7. Currying 322
- 8. Pattern Matching 324
- 9. Composition de fonctions 328
- 10. Pour aller plus loin... 333

Chapitre 10
Un premier projet avec Node.js

- 1. Introduction 335
- 2. Préparation de la base de données 338
- 3. Mise en place du projet 342
 - 3.1 Création du fichier package.json 343
 - 3.2 Mise en place de TypeScript 344
 - 3.2.1 Installation 344
 - 3.2.2 Configuration de TypeScript 345
 - 3.3 Amélioration de l'environnement de développement 349
 - 3.3.1 Exécution de l'application 349
 - 3.3.2 Débogage 351
 - 3.4 Une première application. 353
 - 3.4.1 Mise en place de la bibliothèque Express.js. 353
 - 3.4.2 Gestion des variables d'environnement. 356
 - 3.4.3 Une première requête SQL. 357

4.	Création du Framework MVC	359
4.1	Registre des routes	360
4.2	Décorateurs de routes.	366
4.2.1	Le décorateur Controller	366
4.2.2	Le décorateur Route	368
4.3	Mise en œuvre du Framework MVC	370
4.3.1	Création du contrôleur.	370
4.3.2	Création des routes.	372
4.4	Middleware Express.js	374
5.	Accès aux données	377
5.1	Entité métier	378
5.2	Dépôt de données abstrait	379
5.2.1	Dépôt.	379
5.2.2	Type d'opération.	380
5.2.3	Dépôt de données abstrait	380
5.3	Définition des modèles métiers	388
5.4	Un premier dépôt de données : EmployeeRepository	389
5.5	Utilisation du dépôt de données EmployeeRepository	391
6.	Injection des dépendances	393
6.1	Service de gestion des dépendances	393
6.2	Registre de clé	396
6.3	Décorateurs de gestion des dépendances	397
6.4	Résolution des dépendances lors de la création des contrôleurs	401
6.5	Enregistrement d'une dépendance.	403
7.	Finalisation de l'API	405
7.1	Ajout d'un employé	405
7.2	Gestion des équipes	414
7.3	Changer l'équipe d'un employé	420
8.	Conclusion de l'exercice	426
	Index	427



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.