Introduction

C'est quoi Typescript ?

TypeScript est un **langage de programmation libre et open source** développé par Microsoft qui a pour but **d'améliorer et de sécuriser** la production de code JavaScript. C'est un **sur-ensemble** de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.

TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-contraignante de JavaScript. Il supporte la spécification ECMAScript 6.

Installer Typescript

Typescript n'est pas un langage directement disponible sur le système, il n'est pas non plus interprété directement au sein du navigateur, comme Javascript. Pour l'utiliser, nous avons besoin de l'installer sur la machine.

Il existe deux façons d'installer Typescript:

- En installant **Visual Studio** et le plugin Typescript
- En passant par le gestionnaire de paquets de node.js, **npm**

```
#permet d'installer typescript au sein du dossier courant
npm install typescript

#ou


#permet d'installer typescript de façon globale sur la machine
npm install -g typescript


#puis taper la commande suivante pour obtenir le numéro de version
tsc -v
```

Un Hello World en Typescript

Voici un programme de base écrit en typescript, un 'hello world'. La syntaxe ressemble énormément à Javascript mais vous pouvez noter que l'on utilise un type de données précis pour notre variable, le type 'string' qui correspond à une chaîne de caractères.

```
// on déclare une variable str contenant le chaîne "hello world"
let str:string = "hello world";

// puis on l'affiche
console.log(str);
```

Notez bien que l'on ne peut se servir directement de ce fichier source, en effet, aucun navigateur ou programme connu n'interprète le typescript de façon 'native'. Il faut donc passer par une étape supplémentaire nommée **transpiling** en anglais. Ce terme peut être traduit par **transcompilation**.

Transpiling

Le Typescript n'est pas (encore) un langage destiné à être directement interprété par le navigateur. Son but est d'offrir une **extension** à Javascript, un **sur-ensemble de fonctionnalités**

Il faut donc passer par une étape supplémentaire nommée **transpiling** en anglais. Ce terme peut être traduit par **transcompilation**.

```
#ici on "transcompile" le fichier "monfichier.ts" et l'on
#obtient ainsi un fichier équivalent, traduit en javascript

tsc monfichier.ts
```

Le fichier tsconfig.json

Le fichier tsconfig.json est le cœur de la configuration TypeScript. Il permet de définir comment le compilateur tsc doit transformer le code TypeScript en JavaScript.

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

Les options du compilateur

Dans compilerOptions, on trouve des paramètres importants :

- **target**: Version JS de sortie (ES5, ES6, ES2020, etc.)
- **module**: Système de modules (commonjs, esnext, etc.)
- **strict**: Active toutes les vérifications strictes (true recommandé)
- **outDir**: Dossier de sortie des fichiers compilés
- **rootDir**: Dossier racine des sources TypeScript
- **sourceMap**: Génère un .map pour faciliter le débogage dans le navigateur
- **noImplicitAny**: Interdit les variables sans type explicite
- **esModuleInterop**: Améliore la compatibilité avec les modules CommonJS
- **skipLibCheck**: Ignore les erreurs dans les fichiers .d.ts
- **include**: Fichiers ou dossiers à compiler
- **exclude**: Fichiers ou dossiers ignorés
- **files (opt)**: Si défini, alors seuls les fichiers mentionnés sont compilés

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "strict": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "sourceMap": true,
    "esModuleInterop": true,
    "include": ["src/**/*.ts"],
    "exclude": ["node_modules", "dist"]
  }
}
```

Créer un build

Une fois tsconfig.json configuré :

```
# Compiler le projet
npx tsc

# Compiler en watch mode
npx tsc --watch
```

Et pour se simplifier la vie, on peut ajouter les commandes suivantes au fichier package.json :

```
{
  "scripts": {
    "build": "tsc",
    "build:watch": "tsc --watch"
  }
}
```

Stratégies de projets à configuration multiples

Pour les gros projets ou les monorepos, il est fréquent de découper en plusieurs tsconfig. On regroupe ainsi toutes les options communes dans un fichier **tsconfig.base.json** et les options spécifiques à chaque sous projet sont contenues dans des fichiers spécifiques à chaque sous projet:

```
/monorepo
  ├── tsconfig.base.json
  ├── packages/
  │    ├── frontend/tsconfig.json
  │    └── backend/tsconfig.json
```

tsconfig.base.json

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "strict": true
  }
}
```

packages/frontend/tsconfig.base.json

```
{
  "extends": "../../tsconfig.base.json",
  "compilerOptions": {
    "outDir": "./dist"
  },
  "include": ["src"]
}
```

Création d'un workflow personnalisé

On peut créer un flux de travail complet pour TypeScript avec compilation, nettoyage et exécution.

```
{
  "scripts": {
    "clean": "rm -rf dist",
    "build": "npm run clean && tsc",
    "start": "node dist/index.js",
    "dev": "tsc --watch & nodemon dist/index.js"
  }
}
```

- **clean :** supprime les fichiers compilés
- **build :** nettoie puis compile
- **start :** lance le programme compilé
- **dev :** compile en continu et redémarre automatiquement

TypeScript Basics & Basic Types

Les types de données de bases en Typescript

- **Boolean**: Une simple valeur pouvant être vraie (true) ou fausse
- **Number**: Un nombre flottant (ex: 1.5)
- **String**: Une chaîne de caractères (ex: "Hello World")
- **Any**: Un type désignant "n'importe quel type de données", équivalent du type object en javascript.
- **Void**: Un type désignant du vide, rien, aucune valeur, souvent utilisé pour préciser qu'une fonction/méthode ne retourne pas de valeur (on attend du vide, donc rien)
- **Enum**: Un type de données permettant de créer son propre type de données customisé avec un choix prédéfini dans les valeurs (ex: enum Color {Red, Green, Blue}).
- **Array**: Un tableau pouvant contenir toutes sortes d'éléments (ex: [10,true,"google"]). On peut également définir le type de données d'un tableau (ex: let tab:Array;)
- **Never**: Un type qui ne retourne jamais rien (boucle infinie ou erreur)
- **Null and Undefined**: Equivalent des types null et undefined Javascript.

Il en existe plein d'autres que vous pourrez retrouver sur la documentation en ligne du langage Typescript à www.typescriptlang.org

Templates Strings

Les "templates strings" permettent d'insérer plus facilement des valeurs de variables au sein d'une chaîne de caractère, ces dernières peuvent également être définies sur plusieurs lignes.

Il s'agit d'une nouveauté de la norme ES6, mais comme Typescript est un sur-ensemble de Javascript, elles sont supportées.

```
let jedi = {surname:"Obiwan", name:"Kenobi"};
let msg = `${jedi.name} ${jedi.surname} is the Jedi Master`;
console.log(msg);
```

Les variables

En Typescript, on n'emploie plus le mot clé **var** pour déclarer une variable mais le mot clé **let** et le mot clé **const**.

Le mot clé **const** est utilisé, comme son nom l'indique, pour déclarer une valeur **constante**, çàd une valeur ne pouvant être modifiée, si vous essayez de modifier une constante, une erreur est levée.

Le mot clé **let** est quand à lui utilisé pour déclarer une valeur variable, çàd une valeur capable de changer au cours de la durée de vie de votre programme.

Contrairement à Javascript, qui n'est pas typé, le Typescript l'est lui, comme son nom l'indique. Il est donc utile de préciser le type de données que l'on souhaite stocker au sein de la variable (ou de la constante) à l'aide de la syntaxe suivante:

```
// constante, ne peut être modifiée
const LIGHT_SPEED:string = "299 792 458 m / s";
// variable, peut être modifiée
let msg:string = "Hello World";
```

Les fonctions

En Typescript comme en Javascript, les fonctions sont incontournables. Il s'agit de blocs d'instructions répétables que l'on peut appeler (on dit invoquer) autant de fois que nécessaire.

Les fonctions Typescript, contrairement aux fonctions Javascript, précisent le type de données qu'elles renvoient, et si elles ne renvoient rien, alors le type de retour est **void** (du vide)

```
// les paramètres sont typés, et le type de la donnée
// retournée également à l'aide de la syntaxe suivante:
function sum( a:number, b:number ):number{
    return a + b;
}

// quand une fonction ne retourne aucune donnée
// alors on le précise en utilsant le type "void"
function notif( msg:string ):void{
    alert(msg);
}

// les fonctions anonymes sont également utilisables, comme en JS
let anonymous:Function = function():void{
    console.log("I am an anonymous function");
};

// on peut invoquer une fonction anonyme comme en JS
anonymous();
```

Les fonctions fléchées

En Typescript comme en Javascript ES6, les fonctions fléchées sont supportées, bien entendu le typage des paramètres et de la donnée de retour est à ajouter au sein de la version Typescript. Il est également possible de spécifier des valeurs par défaut aux paramètres. Les fonctions fléchées ont également l'avantage de préserver le contexte dans lequel elles sont déclarées.

```
// fonction fléchée anonyme avec valeur de parmètre par défaut
let hello = (param_user: string = "user"): void => {
    console.log("Hello", param_user);
};

hello(); // invoquons cette fonction

// les fonctions fléchées anonymes préservent le contexte
class Gandalf {
    name: string = "Gandalf";
    introduce = () => {
        console.log("Hello my name is", this.name);
    }
}

// testons notre code
new Gandalf().introduce();
```

Les boucles

En Typescript, les boucles fonctionnent comme en Javascript:

```
let i:number = 10;

// une boucle for classique en Typescript
for( i = 0; i < 10; i ++){
    console.log(i);
}

// une boucle while classique en Typescript
i = 10;
while( --i > -1 ){
    console.log(i);
}

// une boucle do while classique en Typescript
i = -1;
do{
    if( i == -1 ){
        i = 10;
    }
}while( --i > -1)
```

Les tableaux

TypeScript, comme JavaScript, permet de travailler avec des **tableaux** de valeurs. A la différence qu'en Typescipt, les tableaux (si on le souhaite) peuvent être **typés** ,ce qui signifie qu'ils ne peuvent contenir **qu'un seul type de données**. La déclaration de type pour les tableaux peut s'écrire de deux façons :

```
// premièe façon de déclarer un tableau typé
let notes:number[] = [
    0,10,12,7,8,20,13,15
];

// seconde façon de déclarer un tableau typé
let notes2:Array<number> = [
    0,10,12,7,8,20,13,15
];
```

Les enums

TypeScript nous permet de créer et d'utiliser des **enums**. Les enums permettent au développeur de créer un lot de **constantes** et de les regrouper de façon à former un nouveau **type de données**.

```
// créons un type de données "Direction"
// qui peut prendre 4 valeurs différentes
enum Direction { Up = 1, Down = 2, Left = 3, Right = 4 };

let haut: Direction = Direction.Up;
let bas: Direction = Direction.Down;
let gauche: Direction = Direction.Left;
let droite: Direction = Direction.Right;

console.log(haut, bas, gauche, droite);
```

Les Tuples

TypeScript nous permet de créer et d'utiliser des **tuples**. Les tuples permettent au développeur de créer des tableaux ordonnés avec des types prédéfinis.

```
// tableau fixe et ordonné
let utilisateur: [string, number];
utilisateur = ["Alice", 30]; // ✅
utilisateur = [30, "Alice"]; // ❌ Ordre incorrect
```

Next-generation JavaScript

Impact sur la syntaxe

TypeScript est un sur-ensemble de JavaScript :

- Tout code JavaScript valide est aussi du TypeScript.
- On ajoute des annotations de types, des interfaces, des enums, etc.
- Ces ajouts n'existent plus dans le code JavaScript compilé.

```
// TypeScript
function addition(a: number, b: number): number {
  return a + b;
}


// JavaScript généré
function addition(a, b) {
  return a + b;
}
```

Conclusion : L'impact est surtout au niveau du développement et de la compilation, pas à l'exécution.

Automatiser la documentation

Le typage explicite permet de générer automatiquement une documentation à partir du code.

Voici un exemple avec **typedoc**

```
npm install typedoc --save-dev
npx typedoc --entryPoints src/index.ts --out docs


/**
 * Additionne deux nombres.
 * @param a Premier nombre
 * @param b Deuxième nombre
 */
function addition(a: number, b: number): number {
  return a + b;
}
```

Utiliser TypeScript avec du code JavaScript standard

On peut intégrer TypeScript dans un projet existant JavaScript :

1. Renommer petit à petit les fichiers .js en .ts ou .tsx
2. Activer allowJs dans tsconfig.json

```
{
  "compilerOptions": {
    "allowJs": true,
    "checkJs": true
  }
}


// JavaScript avec vérification TypeScript
/**
 * @param {number} a
 * @param {number} b
 */
function addition(a, b) {
  return a + b;
}
```

Inclure/générer des fichiers de définition pour la compatibilité

Les fichiers .d.ts contiennent uniquement des définitions de types, sans implémentation.

Inclure : pour utiliser une lib JS dans TS

```
npm install @types/express
```

Générer pour votre propre lib :

```
{
  "compilerOptions": {
    "declaration": true,
    "outDir": "./dist"
  }
}
```

Bénéfice des getters/setters implicites

En TypeScript, on peut créer des accesseurs (get / set) qui :

1. Permettent de contrôler la lecture/écriture d'une propriété
2. Conservent la même syntaxe que les attributs standards

```
class Utilisateur {
  private _nom: string;

  constructor(nom: string) {
    this._nom = nom;
  }

  get nom(): string {
    return this._nom.toUpperCase();
  }

  set nom(val: string) {
    if (val.length < 2) throw new Error("Nom trop court");
    this._nom = val;
  }
}

const u = new Utilisateur("Alice");
console.log(u.nom); // ALICE
u.nom = "Bob";      // passe par le setter
```

Modules

A partir de ECMAScript 2015 (ES6), JavaScript introduit le concept de modules, et bien entendu, ce concept est supporté en Typescript

Les modules sont éxécutés au sein de leur propre portée, et non pas au sein de la portée globale. Cela signifie que les variables, fonctions, classes, etc... Déclarées au sein d'un module ne sont pas visibles en dehors du module à moins qu'elles ne soient explicitement exportées à l'aide du mot-clé **export**. De fait, pour utiliser un élément exporté, au sein d'un module différent, il faut utiliser le mot-clé **import**.

```
export function toto(){
    console.log("toto est beau");
}

import {toto} from './ts_modules_1';

toto();
```

Namespaces

Utilisés pour regrouper du code sans système de modules

Moins courant aujourd'hui, mais utile pour code global ou libs anciennes

```
namespace Geometrie {
  export function aireCercle(rayon: number): number {
    return Math.PI * rayon * rayon;
  }
}

console.log(Geometrie.aireCercle(10));
```

Typescript & POO

Programmation orientée objet (POO)

La programmation orientée objet est un style de programmation nous permettant de représenter des concepts informatiques sous forme d'objets. Pour conceptualiser un objet, nous avons besoin de plusieurs outils directement intégrés au langage. Il existe également plusieurs façons de "coder objet", Javascript est un langage **orienté objet par prototypage**, mais **Typescript** lui émule le comportement d'un **langage orienté objet par classe**. Il est important de retenir qu'un objet possède plusieurs caractéristiques:

- Un objet possède des **propriétés**, çàd des variables qui lui appartiennent en propre, la portée de ces propriétés peut être, publique, protégée ou privée.
- Un objet possède également des **méthodes**, çàd des fonctionnalités (fonctions) qui lui appartiennent en propre, la portée de ces méthodes peut être, publique, protégée ou privée.
- Dans un langage orienté objet par classe, un objet peut **hériter** d'un autre objet.
- Un objet peut prendre **plusieurs formes** et rédéfinir ses propriétés héritées

Et il ne s'agit ici que des caractéristiques minimum, çàd celles qui définissent la base de la base d'un langage orienté objet, on parle des principes** d'encapsulation, d'héritage et de polymorphisme**.

Les classes

```
class Hero{
    // une propriété peut être publique, protégée ou privée
    public name:string;
    public power:string;

    // la fonction constructrice est invoquée automatiquement
    // à la création d'un nouvel objet
    constructor(
        param_name:string,
        param_power:string
    ){
        // on attribue à nos propriétés les valeurs passées
        // en paramètre
        this.name = param_name;
        this.power = param_power;
    }

    // une méthode peut être publique, protégée ou privée
    public sayMyName():void{
        console.log(this.name);
    }

    public sayMyPower():void{
        console.log(this.power);
    }
}

let myHero:Hero = new Hero("Batman", "Being rich");
myHero.sayMyName();
```

Le Singleton

Le design pattern Singleton permet de s'assurer qu'un object n'est instancié qu'une seule fois à travers toute l'application. Il procure également un accès unifié à cette instance (ici via la méthode **getInstance**.

```
class MySingleton{
    private static _instance:MySingleton;
    public static getInstance():MySingleton{
        if( !MySingleton._instance ){
            MySingleton._instance = new MySingleton();
        }
        return MySingleton._instance;
    }

    private constructor(){}
}

// pour obtenir l'instance:
console.log( MySingleton.getInstance() == MySingleton.getInstance() );
```

Le pattern Factory

Centralise la création d'objets, utile quand la logique d'instanciation est complexe.

```
interface Vehicule {
  drive(): void;
}

class Voiture implements Vehicule {
  drive() { console.log("La voiture roule"); }
}

class Moto implements Vehicule {
  drive() { console.log("La moto roule"); }
}

class VehicleFactory {
  static create(type: "voiture" | "moto"): Vehicule {
    if (type === "voiture") return new Voiture();
    if (type === "moto") return new Moto();
    throw new Error("Type de véhicule inconnu");
  }
}

const v = VehicleFactory.create("voiture");
v.drive();
```

Interfaces & Duck Typing

Le Duck Typing : 'Si ça marche comme un canard et que ça cancane, alors c'est un canard'. En TypeScript, une interface n'impose pas la classe exacte, juste la forme (structure) attendue.

```
interface Flying {
  fly(): void;
}

function makeMeFly(obj:Flying) {
  obj.fly();
}

// avec un objet
const bird = { fly: () => console.log("L'oiseau vole") };

// avec une classe

class Boeing implements Flying{
    fly(): void {
        console.log("Le Boeing vole");
    }
}


makeMeFly(bird); // ✅ Pas besoin que `bird` implémente explicitement `Flying`
makeMeFly( new Boeing());
```

Héritage

```
class Personnage {
    // on veut transmettre cette propriété à nos enfants, on utilise donc protected
    protected name: string;
    // on veut transmettre cette propriété à nos enfants, on utilise donc protected
    protected lifepoint: number;

    constructor(param_name: string, param_lifepoint: number) {
        this.name = param_name;
        this.lifepoint = param_lifepoint;
    }

    //  on doit pouvoir demander à un personnage s'il est mort
    // sans être soi-même un personnage, la portée est donc publique
    public isDead(): boolean {
        // si le nombre de points de vie est inférieur
        // ou égal à 0 alors le personnage est mort
        if (this.lifepoint <= 0) {
            return true;
        }
        else {
            return false;
        }
    }

    public sayMyName():void{
        console.log(this.name);
    }
}
```

Héritage

```
class Wizard extends Personnage{
    constructor( ){
        // on peut invoquer le constructor de la classe parente à l'aide de "super"
        super("Gandalf", 100);

        //... maintenant un nouvel objet de type wizard se nommera toujours
        // Gandalf et aura 100 points de vie
    }

    // on réécrit la méthode sayMyName définit par le parent
    // et héritée de celui-ci.
    public sayMyName():void{
        // mais on peut toujours invoquer l'ancienne "version" de la méthode
        // toujours à l'aide de "super"
        super.sayMyName();

        // si on veut, on peut ajouter des opérations supplémentaires
        console.log("I am a super magician !");
    }
}

// on crée un nouveau sorcier
let gandalf:Wizard = new Wizard();
// on crée un personnage Gollum qui possède 2 points de vie
let gollum:Personnage = new Personnage("Gollum", 2);

gandalf.sayMyName();
gollum.sayMyName();
```

Héritage multiple via Mixins

TypeScript n’autorise pas l’héritage multiple direct, mais on peut combiner plusieurs classes avec des mixins.

```
type Ctor<T = {}> = new (...args: any[]) => T;

function Warrior<TBase extends Ctor>(Base: TBase) {
  return class extends Base {
    fight() { console.log("Baston !"); }
  };
}

function GameCharacter<TBase extends Ctor>(Base: TBase) {
  return class extends Base {
    introduce() { console.log("Chui un barbare !"); }
  };
}

class CharacterBase {}
class Barbarian extends GameCharacter(Warrior(CharacterBase)) {}

const crom = new Barbarian();
crom.fight();
crom.introduce();
```

Advanced types

Gestion des types personnalisés

En TypeScript, on peut créer ses propres types pour :

- Simplifier le code
- Documenter l'intention
- Réutiliser une définition

```
type ID = string | number;
type Point = { x: number; y: number };

const identifiant: ID = 42;
const position: Point = { x: 10, y: 20 };

// ou

interface Utilisateur {
  nom: string;
  email: string;
}

const u: Utilisateur = { nom: "Alice", email: "alice@mail.com" };

// type est plus flexible (permet unions, intersections), interface est extensible.
```

Les génériques

Les génériques permettent de créer des structures ou fonctions qui fonctionnent avec n'importe quel type, tout en conservant la vérification.

```
// fonction générique
function identity<T>(value: T): T {
  return value;
}

identity<string>("Hello"); // T = string
identity(42); // T est inféré comme number


// classe générique
class Box<T> {
  private content: T;
  constructor(value: T) {
    this.content = value;
  }
  getContent(): T {
    return this.content;
  }
}

const box = new Box<number>(123);
console.log(box.getContent()); // number

const boiteTexte = new Box("Bonjour"); // T inféré comme string
```

Les restrictions des types génériques

On peut restreindre un générique avec extends pour n’accepter que certains types.

```
// Restriction à un type
function displayLength<T extends { length: number }>(valeur: T) {
  console.log(valeur.length);
}

displayLength("Hello"); // ✅ string a length
displayLength([1, 2, 3]); // ✅ tableau a length
displayLength(42); // ❌ number n'a pas length

// Restriction à une interface
interface Nameable { myName: string; }

function displayMyName<T extends Nameable>(obj: T) {
  console.log(obj.myName);
}

displayMyName({ myName: "Alice", age: 30 }); // ✅
```

Les décorateurs

Les décorateurs et les 'MetaData'

En TypeScript, un décorateur est une fonction spéciale qui peut être attachée à :

- Une classe
- Une propriété
- Une méthode
- Un paramètre

Important : il faut installer le package **reflect-metadata** et activer les décorateurs dans tsconfig.json :

- Les décorateurs enrichissent le code sans le modifier directement
- Les factories permettent de les paramétrer
- reflect-metadata sert à stocker et lire des données attachées à des éléments
- On peut cibler classes, propriétés, méthodes, paramètres
- Idéal pour des patterns avancés : IoC, validation, logging, sécurité

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}


import "reflect-metadata";

// target étant l'objet auquel vous souhaitez ajouter des metadata
Reflect.defineMetadata("role", "admin", target);

// target étant l'objet duquel vous souhaitez tirer des metadata
const role = Reflect.getMetadata("role", target);
```

Les décorateurs Factories

Un décorateur **simple** est une fonction, en revanche, un décorateur factory est une fonction qui retourne un décorateur, utile pour passer des paramètres.

```
function Logger(message: string) {
  return function (constructor: Function) {
    console.log(`${message} - ${constructor.name} créé`);
  };
}

@Logger("Création de la classe")
class Test {}
// Affiche : "Création de la classe - Test créé"
```

Les différents types de décorateurs

```
// décorateur de propriété
function Readonly(target: any, propertyKey: string) {
  Object.defineProperty(target, propertyKey, {writable: false});
}
 // décorateur de paramètre
function LogParam(target: any, methodName: string, paramIndex: number) {
  console.log(`Paramètre #${paramIndex} de ${methodName}`);
}

// décorateur de fonction/méthode
function LogMethodCall(target: any,propertyKey: string,descriptor: PropertyDescriptor) {
  const func = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Appel de ${propertyKey} avec`, args);
    return func.apply(this, args);
  };
}

class TestClass {
    @Readonly
    nom = "Fixe";

    @LogMethodCall
    helloWorld( @LogParam param:string){ console.log(param)}
}

const e = new TestClass();
(e as any).nom = "Test"; // Ignoré en mode strict
```

Créer des décorateurs personnalisés

```
// Exemple avec reflect-metadata pour stocker des infos sur une méthode

import "reflect-metadata";

function Role(role: string) {
  return function (target: any, key?: string) {
    Reflect.defineMetadata("role", role, target, key!);
  };
}

class AdminService {
  @Role("admin")
  supprimer() {}
}

// Lecture
const role = Reflect.getMetadata("role", AdminService.prototype, "supprimer");
console.log(role); // "admin"
```

TypeScript en pratique

Usage avec Node.js

TypeScript s'intègre parfaitement avec Node.js pour du développement serveur.

```
# installation
npm init -y
npm install typescript ts-node @types/node --save-dev
npx tsc --init


# en développement
npx ts-node src/index.ts

# en production
npx tsc
node dist/index.js

# Utiliser ts-node-dev pour recharger automatiquement en développement.
npm install ts-node-dev --save-dev
npx ts-node-dev src/index.ts
```

Exemple de tsconfig pour node js:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "strict": true,
    "esModuleInterop": true
  }
}
```

Livrer du code compatible JavaScript/Typescript

Si vous développez une librairie en TypeScript destinée à être utilisée par des projets JavaScript et TypeScript, il faut livrer :

Le Javascript compilé

Les définitions de types **.d.ts**

Exemple de configuration tsconfig.json pour une lib

```
{
  "compilerOptions": {
    "declaration": true,       // Génère les fichiers .d.ts
    "declarationMap": true,    // Génère les maps pour navigation
    "outDir": "./dist",
    "target": "ES2019",
    "module": "commonjs",
    "esModuleInterop": true,
    "strict": true
  },
  "include": ["src/**/*"]
}
```

Arborescence du projet:

```
/src
  index.ts
/dist
  index.js      // Code JS compilé
  index.d.ts    // Définitions de types
```

Et on peut consommer la librairie normalement:

```
const { addition } = require("mylib");
console.log(addition(2, 3));
```