TypeScript

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

TypeScript

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction

Histoire TypeScript

- TypeScript est un **sur-ensemble syntaxique** de JavaScript créée en 2012 par Microsoft
- TypeScript ajoute des fonctionnalités de **typage statique**, **classes**, **interfaces** et autres fonctionnalités orientée objet
- Il a été rendu populaire grâce à son adoption par le framework Angular qui l'a rendu obligatoire

Objectif de TypeScript

- Le langage est conçu pour améliorer la productivité et offrant une meilleure vérification des erreurs et une prise en charge par les IDE
- Il est utilisé à la fois côté client et côté serveur
- La plupart des frameworks JS le prenne en charge par défaut

Installation

- `npm install -g typescript`

```
const greeter = (person) => `Hello, ${person}`;
let user = "Jane User";
document.body.textContent = greeter(user);
```

- `tsc greeter.ts`

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

tsconfig.json

Vue d'ensemble

- La présence d'un fichier `tsconfig.json` dans un répertoire indique que ce répertoire est la racine d'un projet TypeScript
- Le fichier `tsconfig.json` spécifie les fichiers racines et les options de compilation nécessaires pour compiler le projet

Composition du fichier de config

- compilerOptions : configuration du comportement de TypeScript
- include : L'emplacement des fichiers `.ts` à compiler
- exclude : L'emplacement des fichiers `.ts` à ne pas compiler

```
{
  "compilerOptions": {
    "module": "system"
  },
  "include": ["src/**/*"],
  "exclude": ["**/*.spec.ts"]
}
```

Les types

Les types primitifs

- string
- number
- boolean

```
let firstName: string = "Toto";
const age: number = 20;
let isAdult: boolean = number >= 18;
```

Les tableaux

- `string[]`
- `Array<number>`

```
const prenoms: string[] = ["toto", "titi"];
const ages = (Array<number> = [1, 5, 10]);
```

Le type any

- Le type `any` permet d'éviter les erreurs de vérification de type
- Ce type peut être pratique pour éviter d'écrire un long type spécifique pour convaincre TypeScript qu'une ligne de code est valide

Les fonctions

- Il est possible de typer les paramètres de fonction

```
function sayHello(name: string) {
  console.log(`Hello, ${name.toUpperCase()}`);
}
```

- Les retours de fonctions ont aussi leur propres types

```
function getRandomInt(min: number, max: number): number {
  return Math.floor(Math.random() * (max - min)) + min;
}
```

Retour de promesses

- Le type promesse est un type générique dans lequel on peut passer le type de retour attendu

```
async function getFavoriteNumber(): Promise<number> {
  return 26;
}
```

Le type objet

Pour typer un objet il est nécessaire d'ajouter un type à chacune de ses propriétés

```
function printCoord(pt: { x: number, y: number }) {}
function printCoord(user: { first: string, last?: string }) {}
```

Les types d'union

- Il est possible de combiner les types grâce au `union type`
- Le union type est formé d'au moins deux types, la valeur passée peut être l'un ou plusieurs de ces types
- Pour créer un type d'union on utilise le caractère `|`

```
let id: number | string;
const printId = (id: number | string) => {};
```

Particularité du type Union

- Les méthodes de l'un des types de l'union ne sera accessible qu'après vérification du type

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // id est de type string
    console.log(id.toUpperCase());
  } else {
    // id est de type number
    console.log(id);
  }
}
```

Les alias de type

- Un alias de type permet de nommer les différents types créés précédemment pour pouvoir les réutiliser
- Le type alias est juste un nom pour n'importe quel type

```
type Point = {
  x: number,
  y: number,
};

type ID = number | string;
```

Les interfaces

- Les interfaces sont une autre façon de nommer un type d'objet
- TypeScript ne s'intéresse qu'à la structure des valeurs passées dans une variable et non à son type

```
interface Point {
  x: number;
  y: number;
}
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}
printCoord({ x: 100, y: 100 });
```

Différences entre Type et Interface

- Les alias de type et les interfaces sont très similaires
- La principale distinction étant qu'un type ne peut pas être rouvert pour ajouter de nouvelles propriétés, contrairement à une interface qui est toujours **extensible**

Le type d'assertion

- Le type d'assertion permet de préciser un type spécifique que TypeScript ne peut pas connaître
- Ce type est supprimé à la compilation, ce qui veut dire qu'aucune exception ne sera levée si le type précisé est erroné
- Le mot clé à utiliser pour le type assertion est : `as`
- Ce type permet d'utiliser les méthodes et propriétés du type spécifié

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

Les types littéraux

- Les types littéraux, permettent d'obtenir une valeur exacte qu'une chaîne de caractères, un nombre ou un booléen doit avoir
- Les types littéraux de chaînes de caractères peuvent être combinés avec des types d'union, des gardes de type et des alias de type pour obtenir un comportement similaire à celui d'une énumération

```
let x: "hello" = "hello";
function printText(s: string, alignment: "left" | "right" | "center");
```

Null et Undefined

- L'option `strictNullChecks` fait en sorte que lorsqu'une valeur est null ou undefined, il faut la tester avant d'utiliser des méthodes ou des propriétés sur cette valeur

```
function doSomething(x: string | null) {
  if (x === null) {
    // do nothing
  } else {
    console.log("Hello, " + x.toUpperCase());
  }
}
```

Les fonctions en détail

Typage des callbacks

- Le typage des callback se fait sous forme de fonctions fléchées
- Il est également possible de créer un type dédié pour les fonctions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

type GreetFunction = (a: string) => void;
function greeter(fn: GreetFunction) {}
```

Signature de constructeur

- Les fonctions peuvent être invoquées avec le mot clé `new`
- Pour gérer ce type il faut ajouter le mot clé `new` dans la déclaration de type

```
type SomeConstructor = {
  new(s: string): SomeObject,
};
function fn(ctor: SomeConstructor) {
  return new ctor("hello");
}
```

Les fonctions génériques

- Une fonction générique permet de manipuler différents types, c'est un type de polymorphisme en POO
- La déclaration d'une fonction générique se fait avec le type entre `<>`

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}
```

Contraintes de type générique

- Les contraintes sur les types génériques permet d'ajouter une limitation sur le type passé en paramètre
- Dans l'exemple suivant, le type doit implémenté la propriété length

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}
```

Surcharge de fonctions

- Certaines fonction JavaScript peuvent être appelées avec une multitudes d'argument (exemple `Date`)
- TypeScript permet de surcharger les signatures de fonctions, pour cela il faut ajouter les signatures de fonctions suivi de son corps

```
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
  // Détail de la fonction
}
```

Les types de retours

- **void** : indique aucune valeur de retour
- **object** : représente un type non primitif, différent de `Object` ou de `{}`
- **unknown** : type similaire à `any` mais beaucoup moins permissif
- **never** : indique qu'une fonction lève une exception ou termine le programme
- **Function** : représente une fonction qui a les propriétés `bind`, `call` , `apply` et qui peut être appelée

Destructuration de paramètres

- La destructuration de paramètres en TypeScript fonctionne de la même manière qu'en JavaScript en précisant les types

```
function sum({ a, b, c }: { a: number, b: number, c: number }) {
  console.log(a + b + c);
}
// OU
type ABC = { a: number, b: number, c: number };
function sum({ a, b, c }: ABC) {
  console.log(a + b + c);
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le type Objet

Les types d'objets

En TypeScript il existe différentes manières de regrouper les données via des objets via :

- Les objets anonymes
- Les interfaces
- Les alias

Les propriétés facultatives

La plupart des objets peuvent avoir des propriétés qui sont facultatives, dans ce cas un rajouter le caractère `?` après son type

```
interface PaintOptions {
  shape: Shape;
  xPos?: number;
  yPos?: number;
}
```

Propriétés en lecture seule

- Les propriétés en lecture seule permettent de protéger l'affectation sur ces propriétés
- On ne pourra les affecter que lors de la création de l'objet

Extension de type

L'extension de type en TS permet de spécialiser un type plus général sans avoir à se répéter

```
interface BasicAddress {
  name?: string;
  street: string;
  city: string;
  country: string;
  postalCode: string;
}

interface AddressWithUnit extends BasicAddress {
  unit: string;
}
```

Extensions multiples

Les interfaces en TS supporte l'extension de plusieurs interfaces :

```
interface Colorful {
  color: string;
}

interface Circle {
  radius: number;
}

interface ColorfulCircle extends Colorful, Circle {}
```

Intersection de types

Le caractère `&` permet de créer un combinaison de plusieurs types, de la même manière que l'union avec `|`

Ainsi le code de la slide précédente peut aussi s'écrire de la manière suivante :

```
interface Colorful {
  color: string;
}
interface Circle {
  radius: number;
}

type ColorfulCircle = Colorful & Circle;
```

Type ou Interface

- **Type primitif** : pour utiliser un alias d'un type primitif, le `type` est à privilégier
- **Union** : pour une union, le `type` est à privilégier
- **Objet** : une `interface` est plus appropriée pour une représenter un objet
- **Extension de type** : une `interface` est plus adaptée pour étendre un type

Les objets génériques

En TypeScript, un objet générique est un objet qui peut manipuler n'importe quel type

De la même façon que les fonctions génériques, on la déclare avec `<>`

```
interface Box<Type> {
  contents: Type;
}

let box: Box<string>;
```

Le type tuple

Le type tuple est une sorte de tableau qui connaît le nombre d'éléments qui le compose ainsi que les types qu'il contient

```
type StringNumberPair = [string, number];
const myStringNumber: StringNumberPair = ["user", 12];
```

Ici le type à l'index 0 est de type string tandis que celui à l'index 1 est de type number

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les classes

Brief

- TypeScript prend en charge les classes introduites par JavaScript en 2015 avec l'arrivée de ES6.
- Cette prise en charge permet de typer les membres, méthodes et types de retours tout en ajoutant de nouveaux comportements.

Visibilité

Les membres d'une classes peuvent avoir 3 visibilités:

- `public` : les membres sont accessibles en dehors de la classe
- `protected` : les membres sont accessibles uniquement dans la classe et ses classes dérivées
- `private` : les membres sont accessibles uniquement dans la classe

```
class user {
  private id: int;
  protected username: string;
  public lastConnexion: date;
}
```

Membres

Les membres d'une classe fonctionne comme le reste des types en TypeScript

```
class User {
  id = 0; // le type sera number
  readonly username; // le membre est initialisé dans le constructeur
  age; // type any
}
```

Le constructeur

Comme pour les fonctions, il est possible de surcharger le constructeur pour avoir plusieurs signatures

```
class Point {
  // Overloads
  constructor(x: number, y: string);
  constructor(s: string);
  constructor(xs: any, y?: any) {
    // A implémenter
  }
}
```

Les méthodes

Les méthodes fonctionnent de la même manière que les fonctions en TypeScript

```
class Point {
  x = 10;
  y = 10;

  scale(n: number): void {
    this.x *= n;
    this.y *= n;
  }
}
```

Les classes génériques

- Les classes comme les interfaces peuvent être génériques
- Le système de contrainte est le même que pour les interfaces

```
class Box<Type> {
  contents: Type;
  constructor(value: Type) {
    this.contents = value;
  }
}

const b = new Box("hello!");
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Manipulation de types

Opérateur keyof

- L'opérateur `keyof` prend un type d'objet et produit une chaîne ou une union numérique littérale de ses clés
- Cet opérateur est généralement utilisé avec les **mapped types**

```
type Point = { x: number; y: number };
type P = keyof Point;
```

Opérateur typeof

- `typeof` est un mot clé du JavaScript qui permet de vérifier le type d'une variable
- Avec TypeScript ce mot clé permet d'affecter le type d'une variable à un nouveau type

```
type Predicate = (x: unknown) => boolean;
type K = ReturnType<Predicate>;
```

Types conditionnels

Il est possible d'affecter un type selon une condition à l'aide d'une ternaire sous la forme:

```
SomeType extends OtherType ? TrueType : FalseType;
```

```
interface Animal {
  live(): void;
}
interface Dog extends Animal {
  woof(): void;
}

type Example1 = Dog extends Animal ? number : string;
```

Mapped Types

- Un type mappé est un type générique qui utilise une union de PropertyKeys (souvent créée via un `keyof`) pour itérer à travers les clés afin de créer un type
- Ici le type OptionsFlags va transformer tous les propriétés passées en paramètres en booléen

```
type OptionsFlags<Type> = {
  [Property in keyof Type]: boolean;
};
```

```
type Features = {
  darkMode: () => void,
  newUserProfile: () => void,
};

type FeatureOptions = OptionsFlags<Features>;
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les types utilitaires

Un type utilitaire

- TypeScript fournit plusieurs types d'utilitaires pour faciliter les transformations de type courants
- Ces utilitaires sont disponibles n'importe où dans le langage

Partial<Type>

- Construit un type dont toutes les propriétés sont définies comme facultatives

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}
```

Required<Type>

- Construit un type composé de toutes les propriétés de Type définies comme obligatoires. Le contraire de Partial

```
interface Props {
  a?: number;
  b?: string;
}

const obj: Props = { a: 5 };

const obj2: Required<Props> = { a: 5 };
```

Omit<Type>

Construit un type en sélectionnant toutes les propriétés du type et en supprimant les clés (chaîne littérale ou union de chaînes littérales). Le contraire de Pick.

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
  createdAt: number;
}
```

```
type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
  title: "Clean room",
  completed: false,
  createdAt: 1615544252770,
};
```

Les décorateurs

Définition

Le design pattern Decorator permet d'affecter dynamiquement de nouveaux comportements à des classes, méthodes et propriétés en les plaçant dans des wrappers qui implémentent ces comportements
La prise en charge des décorateurs nécessite l'ajout d'une option dans le fichier `tsconfig.json` :

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

Fonctionnement

- Les décorateurs utilisent la forme `@expression`, où l'expression doit évaluer une fonction qui sera appelée au moment de l'exécution avec des informations sur l'élément décoré
- Par exemple, étant donné le décorateur `@sealed`, nous pourrions écrire la fonction sealed comme suit :

```
function sealed(target) {
  // do something with 'target' ...
}
```

Exemple de décorateur

- Voici un exemple de décorateur qui rajoute un système de log à une fonction

```
function log(target: any, name: string, descriptor: PropertyDescriptor) {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${name} with`, args);
    const result = original.apply(this, args);
    console.log(`Result of ${name} is`, result);
    return result;
  };
  return descriptor;
}
```

Decorator Factory

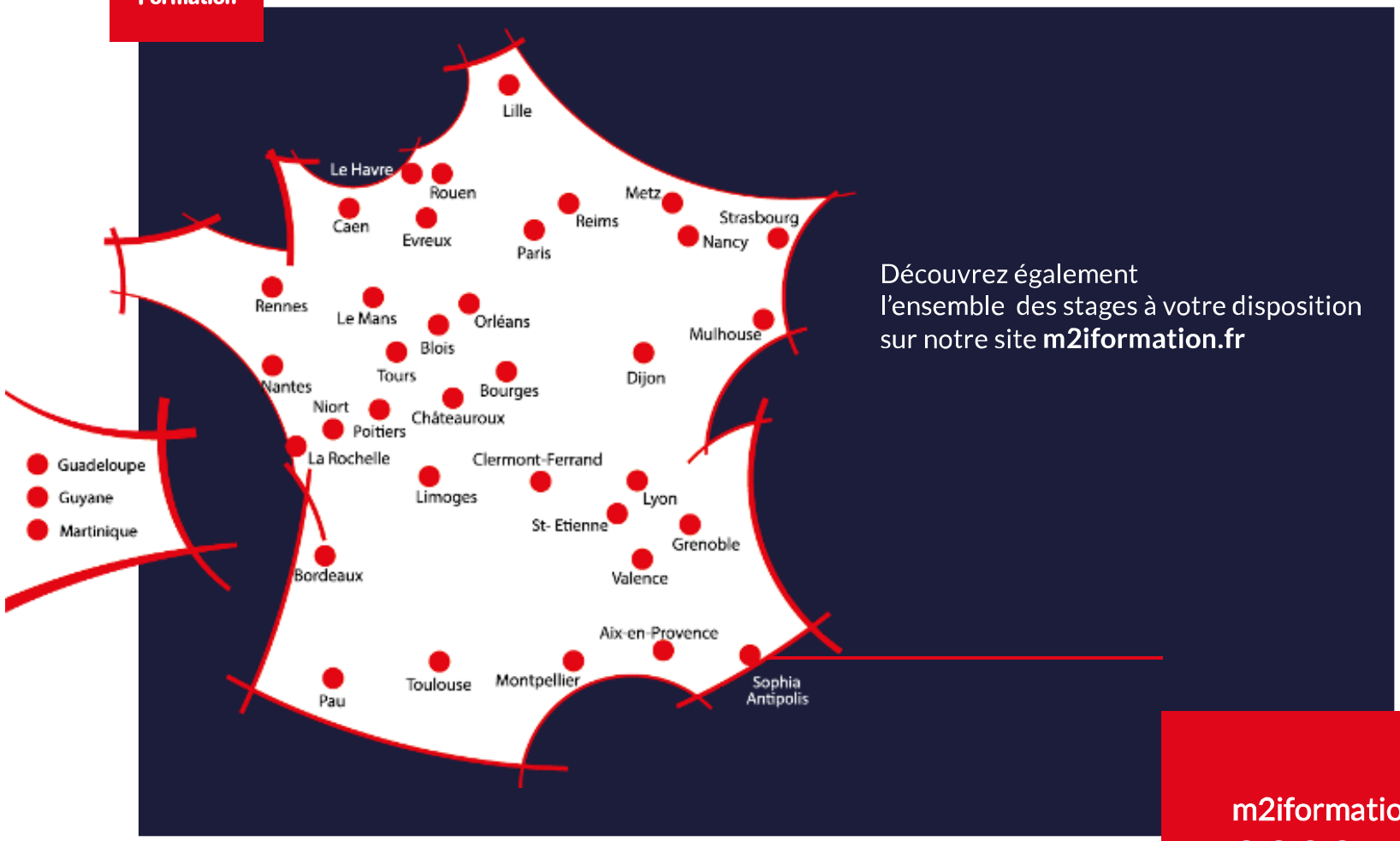
- Pour personnaliser la manière dont un décorateur est appliqué à une déclaration, il faut créer un decorator factory
- Techniquement, c'est une fonction qui renvoie l'expression qui sera appelée par le décorateur au moment de l'exécution

```
function first() {
  console.log("Le décorateur first() est exécuté");
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    console.log("first(): called");
  };
}
```

```
class ExampleClass {
  @first()
  method() {}
}
```

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr