

# Typescript

## Table des matières

Typescript.....	1
1. Qu'est-ce que c'est ?.....	1
2. Principaux typages en Typescript.....	1
3. Les variables.....	1
4. Tableaux.....	2
5. Les objets.....	2
5.1 Héritage avancé.....	5
6. Filtrer un tableau en typescript.....	6
7. Les Guid.....	7
8. Les Promises.....	8

### 1. Qu'est-ce que c'est ?

Il s'agit d'un langage open source basé sur le Javascript. Cependant, il est plus « intransigeant » sur son typage, c'est-à-dire que Typescript, contrairement à Javascript, va vérifier constamment que vous assignez bien le bon type à une variable.

Le code typescript est transformé en Javascript par le compiler, et on obtient un code Javascript « propre », utilisable dans n'importe quelle application.

Une application Angular suit le TSLint, un outil qui analyse votre code pour le rendre « propre », améliorant sa lisibilité et aussi pour d'éventuelles erreurs.

### 2. Principaux typages en Typescript

En Typescript vous avez les types :

- string
- number
- boolean
- undefined
- array

Et bien d'autres, mais il s'agit là des plus utilisés.

Vous pouvez utiliser le **any** pour indiquer à Typescript, que l'on attend n'importe quel type, **mais je ne veux absolument pas voir ça !**

### 3. Les variables

On déclare des variables avec les mots clés :

- **let** : si l'on souhaite réassigner celle-ci
- **const** : s'il s'agit d'une variable qui ne changera pas

Et avec son typage on arrive à ça :

**let name : string ;** // déclaration d'une variable de nom name, de type string étant réassignable

**const email : string ;** // déclaration d'une variable de nom email, de type string, n'étant pas réassignable

## 4. Tableaux

Il existe plusieurs manières de déclarer un tableau :

- Via les []
- Via le mot clé Array<Type>

Et une fois déclaré, on arrive à ça :

- array: string[]
- array: Array<string>

Array<string> a un avantage, c'est qu'il peut être instancier via un new et permet d'utiliser des méthodes pour ajouter des éléments au tableau, les supprimer, etc.

## 5. Les objets

En Javascript il existe plusieurs manières de déclarer un objet, par exemple :

```
const user = { name : 'John', firstName : 'Doe' } ;
```

Peut s'écrire dans une interface :

```
Interface User {  
    name : string ;  
    firstName : string ;  
}
```

Et on la réutilisera de cette manière :

```
const user : User = { name : 'John', firstName : 'Doe' } ;
```

OU

```
const user : User ;  
user.name = 'John' ;  
user.firstName = 'Doe' ;
```

Il est aussi possible d'utiliser une classe :

```
Class User {  
  
    name : string ;  
  
    firstName : string ;  
  
}
```

Et quelle est la différence entre une classe et une interface ?

Une classe peut avoir un constructeur et ainsi avoir des paramètres définis dès sa création, une interface non.

Une classe peut aussi utiliser des propriétés.

Schéma d'une classe :

```
export class Energy {  
    private _id: number;  
    private _name: string;  
  
    get id(): number {  
        return this._id;  
    }  
  
    get name(): string {  
        return this._name;  
    }  
  
    set name(value: string) {  
        this._name = value;  
    }  
  
    constructor(id: number, name: string) {  
        this._id = id;  
        this._name = name;  
    }  
  
    methodThatDoNothingSpecial(): void { }  
}
```

Déclaration d'une classe de nom Energy (création d'un nouveau type)

Déclaration des **attributs** de la classe, c'est ce qui la caractérise  
On les déclare **private** afin qu'ils ne soient utilisables que dans la classe en cours, et il faut les typer avec les deux points suivis du type

Déclaration des **propriétés** de la classe, c'est ce qui permet d'accéder aux attributs en dehors de la classe

On peut les utiliser via le '.', par exemple :  
`const energy = new Energy(1, 'Diesel');`  
`energy.name` (=> on accède à la propriété name, car energy est une variable de type Energy, via le **new Energy** plus haut)

Déclaration d'un constructeur rempli, c'est à dire qu'il prend en paramètre des valeurs à **set** à ses attributs (via les propriétés)  
Si on ne renseigne pas de constructeur, par défaut il est vide, on peut tout de même faire un `new Energy()`, juste qu'il faudra affecter ses attributs différemment

Déclaration d'une méthode pour la classe Energy, void signifie qu'elle ne renvoie aucune valeur. On y accède au même titre qu'une propriété sauf que l'on met des parenthèses à la fin :  
`energy.methodThatDoNothingSpecial();`  
On met les éventuels paramètres de celle-ci entre les parenthèses

Déclaration d'une interface :

```
export interface InterfaceEnergy {  
    id: number;  
    name: string;  
  
    displayName(name: string): string;  
}
```

Déclaration d'une interface de nom InterfaceEnergy

Déclaration des **attributs** de l'interface, ici on leur laisse leur visibilité par défaut (**public**)

> Une interface est aussi une déclaration d'un nouveau type pour enrichir votre application

> La différence entre une interface et une classe est que l'interface ne s'instancie pas, vous n'avez pas besoin de faire un `new InterfaceEnergy` pour accéder aux méthodes / attributs

Déclaration d'une méthode pour notre interface, cependant on ne déclare que le synopsis de celle-ci : **uniquement son nom, ses éventuels paramètres et sa valeur de retour.**  
(Une valeur de retour ou **void** si celle-ci est amenée à ne pas renvoyer de valeur)

## Déclaration d'une classe abstraite :

```
export abstract class AbstractNameProperty {  
    // tslint:disable-next-line:variable-name  
    protected _name: string;  
  
    get name(): string {  
        return this._name;  
    }  
  
    set name(value: string) {  
        this._name = value;  
    }  
}
```

← Déclaration de classe abstraite (c'est à dire qui ne s'instancie pas, comme une interface, sauf que l'on peut y décrire des comportements de méthodes et des attributs avec getters/setters), pour cela on utilise le mot clé **abstract** pour la décrire comme classe abstraite

Il existe 3 valeurs de visibilité des attributs / méthodes de classe :

- public : on peut accéder à l'attribut/méthode dans la classe et en dehors
- private : on peut accéder à l'attribut/méthode uniquement dans la classe mais pas en dehors de celle-ci
- protected : on peut accéder à l'attribut/méthode dans la classe et ses classes filles (héritage) mais pas en dehors de celle-ci

```
export abstract class Human {  
    ...  
}
```

Ici, Human est la classe mère

```
export class Homme extends Human {  
    ...  
}
```

Ici, Homme est classe fille, car elle étend (**extends**) la classe Human, on peut donc dire, qu'un Homme est aussi de type Human

## Exemple de déclaration d'une classe :

```
/////////////////////////////////////////  
// déclaration d'une classe //  
/////////////////////////////////////////  
class Brand {  
  
    ///////////////////////////////////////  
    // déclaration des attributs //  
    ///////////////////////////////////////  
  
    // Ils sont toujours en privé dans le cas d'une classe métier, afin d'encapsuler  
    // son comportement  
    // C'est la classe qui décide comment on accède à ses attributs  
    private _id: number;  
    private _name: string;  
  
    ///////////////////////////////////////  
    // déclaration des propriétés //  
    ///////////////////////////////////////  
  
    // Les propriétés correspondent aux getters/setters des attributs de la classe  
    // La différence entre une propriété et une méthode c'est le mot clé get/set devant  
    // Ainsi que l'utilisation de celle-ci : avec ou sans parenthèses  
  
    // Seulement la propriété get a été défini pour l'id  
    // car il n'y a aucun intérêt à vouloir le modifier  
    get id(): number {  
        return this._id;  
    }  
  
    // get : on récupère quelque chose  
    get name(): string {  
        return this._name;  
    }  
  
    // set : on modifie la valeur de notre attribut  
    set name(value: string) {  
        this._name = value;  
    }  
  
    ///////////////////////////////////////  
    // déclaration d'un constructeur //  
    ///////////////////////////////////////  
}
```

```
// Déclaration d'un constructeur pour instancier la classe, celui-ci est rempli,
// c'est à dire qu'il initialise ses attributs via les paramètres qu'il a
// Si un constructeur n'est pas défini dans une classe, il existe implicitement,
// il est juste vide
constructor(id: number, name: string) {
    this._id = id;
    this._name = name;
}

//////////
// déclaration des méthodes //
//////////

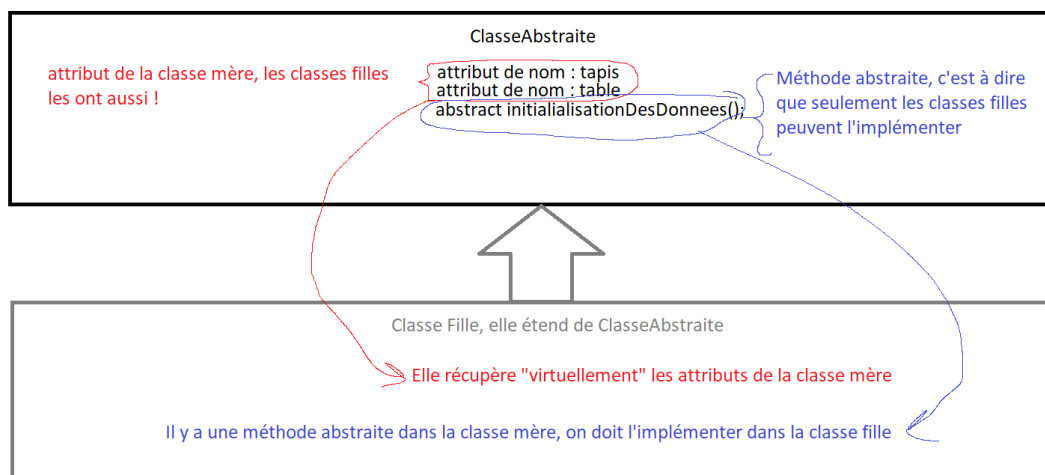
// Déclaration d'une méthode/fonction
toString(): string {
    return '(' + this.id + ') ' + this.name;
}
}
```

Exemple de réutilisation d'une classe :

```
constructor() {
    // Création d'une variable de nom arrayObject qui est une instance de Array<string>
    // Initialisation de la variable classe de nom arrayObject
    // qui est une instance de Array<string>
    const arrayObject = new Array<string>();
    // this : représente la classe dans laquelle on se situe, ainsi on peut
    // accéder aux attributs ou aux méthodes de cette classe via le mot clé "this."
    this.arrayObject = new Array<string>();
    // Cette ligne permet de dire que this.brand est une instance de la classe Brand
    this.brand = new Brand(1, 'Opel');
    // Ici on utilise la méthode toString, donc il y a les ()
    this.brand.toString();
    // Ici on utilise la propriété name donc il n'y a pas de ()
    this.brand.name;
}
```

## 5.1 Héritage avancé

Schéma explicatif de l'héritage avancé



***Il ne faut pas perdre de vue, qu'une classe fille bénéficie des attributs (protected/public) de la classe mère, il en va de même pour les mfonctions/méthodes de la classe mère !***

Exemple d'une méthode **abstraite**, on ne peut la déclarer que dans une **classe abstraite**, vous noterez qu'elle n'a pas de comportement, le principe est qu'elle impose aux classes filles de lui en définir un.

Ainsi, on peut « cadrer » des choses, à savoir que toutes les classes filles vont avoir la même méthode/fonction, et on peut donc la réutiliser.

Méthode abstraite : on déclare juste son synopsis (visibilité, nom et retour) sans lui définir de traitement  
On laisse les classes filles décider de ce qu'elles feront, mais toutes les classes fille l'auront  
`protected abstract initializeDatas(): void;`

Dans notre exemple, le constructeur de la classe mère appelle la méthode **initializeDatas**, afin que lors de l'instanciation d'une classe fille, **initializeDatas** sera appelée aussi.

```
protected constructor() {  
    this.arrayAbstractAttributes = new Array<AbstractAttributes>();  
    this.initializeDatas();  
}
```

Ainsi, on peut ne pas définir de constructeur dans une classe fille et laisser le constructeur de la classe mère travailler.

Les classes filles, par défaut, utilisent le constructeur de la classe mère, il peut être nécessaire dans certains cas de figure de le redéfinir ou l'améliorer afin qu'il correspondent mieux à nos besoins, dans ces cas-là, il faut utiliser le mot clé « *super()* », qui permet de rappeler le constructeur de la classe mère.

Exemple :

```
constructor(private factionService: FactionService, private weaponService:  
WeaponService) {  
    super();  
}
```

Ici, je définis un constructeur pour ma classe fille, qui a deux injections de dépendances, la classe mère ne les a pas, donc je devais redéfinir le constructeur. ***J'ajoute le « super() » afin de rappeler le constructeur de la classe mère, car lors de la redéfinition de constructeur, il faut rappeler celui de la classe mère.***

## 6. Filtrer un tableau en typescript

Syntaxe « allongée » via un « foreach » (attention, il est implicite !) :

```
let abstractGeoApiTmp: AbstractGeoApi;  
for (let abstractGeoApi of this.arrayAbstractGeoApi) {  
    if(abstractGeoApi.code === code) {  
        abstractGeoApiTmp = abstractGeoApi;  
    }  
}
```

Syntaxe « classique » via for « normal » :

```
for(let i = 0; i < this.arrayAbstractGeoApi.length; i++) {  
    if (this.arrayAbstractGeoApi[i].code === code) {
```

```
}  
}
```

Syntaxe « via filter de la classe Array » (celle-ci se base sur la méthode *forEach()*, utilisable sur un Array):

```
const abstractGeoApi = this.arrayAbstractGeoApi.filter(ga => ga.code === code);
```

En français : « Je déclare une variable, qui correspond à l'itération de mon tableau, de nom *ga*, pour chaque « *ga* » qui valide la condition (ici « *ga.code === code* ») alors, je rempli un autre tableau avec le *ga* correspondant à la condition »

(Lien vers la documentation des boucles : <https://www.zendevs.xyz/les-boucles-for-foreach-each-en-javascript/#javascript-for-in>)

## 7. Les Guid

*Guid* ou *Global Unique IDentifier* représente un identifiant unique pour nos classes, il a l'avantage d'être plus sécurisé qu'un entier classique ou chaîne de caractère, car il est généré aléatoirement à chaque fois que vous en créez un (fonction *Guid.create()*)

```
// Création d'un Guid - ex: 544fc1f7-d989-3e25-8dc9-2d1472e5c863  
this.guid = Guid.create();  
console.log('Objet GUID : ' + this.guid);  
// Pour passer le type Guid en type string, il faut faire un toString()  
const guidString = this.guid.toString();  
console.log('GUID string : ' + guidString);  
// Afin de récupérer un objet de type Guid depuis une chaîne de caractère, on  
fait un Guid.parse  
this.guid = Guid.parse(guidString);  
console.log('Objet Guid récupère depuis Guid.parse : ' + this.guid);  
// Pour effectuer une condition entre deux Guid, il faut utiliser la méthode  
equals()  
if (this.guid.equals(Guid.create())) {  
    // traitement en cas d'égalité  
}
```

Pour utiliser le *Guid* en paramètre de routes ou de component (via Input) il faut le convertir en chaîne de caractère via la méthode *toString()*.

Et de dans le component où l'on récupère le *Guid*, sous forme de chaîne de caractères, il faut le « parser » pour le retransformer en objet de type *Guid* via la méthode *Guid.parse()*.

L'objet *Guid* ne se « teste » pas de la même manière qu'un type générique (comme string ou number) via '===' ou '!==', il faut utiliser la méthode *equals()*.

## 8. Les Promises

Une promise est un traitement asynchrone, c'est-à-dire que le programme continuera de s'exécuter pendant que la Promise continuera le sien.

Une promise est un type, c'est-à-dire qu'une fonction peut renvoyer une promise, cela se représente de cette manière :

*Promise<string>*

Prenons par exemple cette méthode :

```
async getAsyncNumber(): Promise<number> {  
    return new Promise((resolve, reject) => {  
        setTimeout(() => {  
            resolve(42) ;  
        }, 2000);  
    });  
}
```

En typescript on déclare les méthodes travaillant de manière asynchrone via le mot clé **async**, ici cette méthode va simplement renvoyer le nombre 42 dans 2ms.

Pour réutiliser cette méthode, on l'appelle avec le mot clé **await** de cette manière :

```
const theReponse = await this.getAsyncNumber() ;
```