
TS

# SOMMAIRE

# TypeScript

## INTERFACE

### un exemple dans lequel on utilise 2 interfaces

```
interface Books {
	title: string;
}

interface Author {
	name: string;
	firstname: string;
	collection: Books[]
}



function getMessage (author: Author): string {
	const message = `${author.name} ${author.firstname} possède ${author.collection.length}
livres`;
	return message;
}

const books1 = [{ title: 'livre1' }, { title: 'livre3' }];

const author1 = {
   name: 'Dupond',
   firstname: 'Jean',
   collection: books1
}

const msg = getMessage(author1);

console.log(msg);
```

> **Note:** le retour sera Dupond Jean possède 2 livres

# PARAMETRE OPTIONNEL

**En JavaScript nous avons les paramètres optionnels. Si on ne les passe pas à l'appel de la fonction, leur valeur est undefined. Mais en TypeScript, si on déclares une fonction avec des paramètres typés, le compilateur générera une erreur :**

```
function addPointsToScore(player: Player, point: number): void {
    player.score += point;
}

addPointsToScore(player);
```

> **Note:** error TS2346 Supplied parameters do not match any signature of call target

## Pour montrer qu'un paramètre est optionnel dans une fonction (ou une propriété dans une interface), on ajoute un ? après le paramètre

```
interface Player {
    name: string;
    score: number;
}

function addPointsToScore(player: Player, point?: number): void {
    point = point || 0;
    player.score += point;
}

const p = {
    name: 'Durand',
    score: 12
}

addPointsToScore(p);
addPointsToScore(p, 4);

console.log(p);
```

> **Note:** le résultat est {name: "georegs", score: 16}
> **Astuce:** le paramètre optionnel peut-être signalé par une valeur par défaut function addPointsToScore(player: Player, **point: number = 0**): void { player.score += point; }

# FONCTION EN PROPRIETE

### On peut décrire un paramètre comme devant posséder une fonction spécifique plutôt qu'une propriété

```
interface CanRun {
		run(meters: number): void;
}
function startRunning(dog: CanRun): void {
		dog.run(10);
}

const dog = {
   run: (meters: number) => console.log(`dog runs ${meters}m`)
};

startRunning(dog);
```

> **Note:** le résultat est dog runs 10m

# CLASS

### une classe peut implémenter une interface

```
interface Dog {
   name: string;
   eating: boolean;
}

interface CanEat {
   onCanEat(dog: Dog): void;
}

class Dogs implements CanEat {
   onCanEat(dog: Dog) {
      if (dog.eating) {
         console.log(`Mon chien ${dog.name} est en train de manger`);
      } else {
         console.log(`Mon chien ${dog.name} ne mange pas`);
      }
```

```
    }
}

const myDog = {
    name: 'wolf',
    eating: true
}

const dogs = new Dogs()

dogs.onCanEat(myDog);
```

**Note:** le résultat est Mon chien wolf est en train de manger

### une classe peut implémenter une interface qui étend elle-même de plusieurs interface

```
interface Dog {
    name: string;
    eating: boolean;
    sleeping: boolean;
}

interface CanEat {
    onCanEat(dog: Dog): void;
}

interface CanSleep {
    onCanSleep(dog: Dog): void;
}

interface Animal extends CanEat, CanSleep { }

class Dogs implements Animal {
    onCanEat(dog: Dog) {
        if (dog.eating) {
            console.log(`Mon chien ${dog.name} est en train de manger`);
        } else {
            console.log(`Mon chien ${dog.name} ne mange pas`);
        }
    }

    onCanSleep(dog: Dog) {
```

```
        if (dog.sleeping) {
            console.log(`Mon chien ${dog.name} est en train de dormir`);
        } else {
            console.log(`Mon chien ${dog.name} ne dort pas`);
        }
    }
}

const myDog = {
    name: 'wolf',
    eating: true,
    sleeping: false
}

const dogs = new Dogs()

dogs.onCanSleep(myDog);
```

> **Note:** le résultat est Mon chien wolf ne dort pas

# PRIVATE / PUBLIC

### ajouter public ou private à un paramètre de constructeur permet de créer et d'initialiser un membre privé ou public. On évite ainsi son équivalent plus verbeux.

```
class Student {
    constructor(private eleve: string) { }

    getEleve() {
        console.log(this.eleve);
    }
}

const eleve = new Student('Charles');

eleve.getEleve();
```

> **Note:** le résultat est Charles

# DECORATEUR

**Un décorateur est une façon de faire de la méta-programmation. Ils ressemblent beaucoup aux annotations utilisées en Java**

Leur rôle est assez simple: ils ajoutent des métadonnées à nos classes, propriétés ou paramètres pour par exemple indiquer "cette classe est un composant", "cette dépendance est optionnelle", "ceci est une propriété spéciale du composant", etc...

Nous allons construire un décoratuer très simple : @Log(). Il va écrire le nom de la méthode à chaque fois qu'elle sera appelée.

```
class RaceService {
		@Log()
		getRaces() {
			const description = "Lister l'ensemble des courses";
		}

		@Log()
		getRace(raceId: number) {
			const description = "Lister la course #${raceId}";
		}
}
```

Pour définir le décorateur, nous devons écrire une méthode renvoyant une fonction comme celle-ci :

```
const Log = function () {
   return (target: any, name: string, descriptor: any) => {
      const originalMethod = descriptor.value;
      console.log(`méthode appelée depuis ${name}`);
      console.log(`méthode originale contient ${originalMethod}`);

			return descriptor;
		};
};
```

**Note:** Selon l'emplacement du décorateur, la fonction n'aura pas exactement les mêmes arguments.

> Ici nous avons un décorateur de méthode, qui prend 3 paramètres :
> • **target** : la méthode ciblée par notre décorateur • **name** : le nom

> de la méthode ciblée • **descriptor** : le descripteur (le contenu) de la méthode ciblée

Pour tester

```
const raceService = new RaceService();
```

> ** Note:** le résultat est : méthode appelée depuis getRaces méthode originale contient getRaces() { const description = "Lister l'ensemble des courses"; } méthode appelée depuis getRace méthode originale contient getRace(raceId) { const description = "Lister la course #${raceId}"; }