

# TYPESCRIPT

---



# HISTOIRE EN BREF

Typescript est sortie en 2012

- Open source (Microsoft)
- Cocrée par Anders Hejlsberg (un des créateurs de c#!)
- Il introduit le concept de class dans l'ecosystem js, et c'est ensuite seulement que le TC39\* à ajouter les class à spec!

# TYPESCRIPT ?

- C'est un langage de programmation
- on parle de "sur-ensemble" syntaxique à javascript
  - on parle de transpilation du code ts vers js
- Syntaxe plus souple et évolutif (ne suis pas le TC39)
- Sécurise le code (limite les erreurs de typage)

# TYPESCRIPT ?

- Fortement orienté objet
- Typage fort (mais peut être désactivé)
  - d'où son nom :)
- Du code js fonctionnera dans un projet ts
  - /!\ l'inverse n'est pas vrai et c'est pas le but..

# TYPESCRIPT ?

En résumé les avantages sont :

- code plus lisible (= plus besoin de 20k ligne de jsdoc)
- maintenable
- évolutif
- multiplateforme (= tournera sur toutes les plateformes js)

# INSTALLATION

Rien de méchant :



```
npm install typescript
```

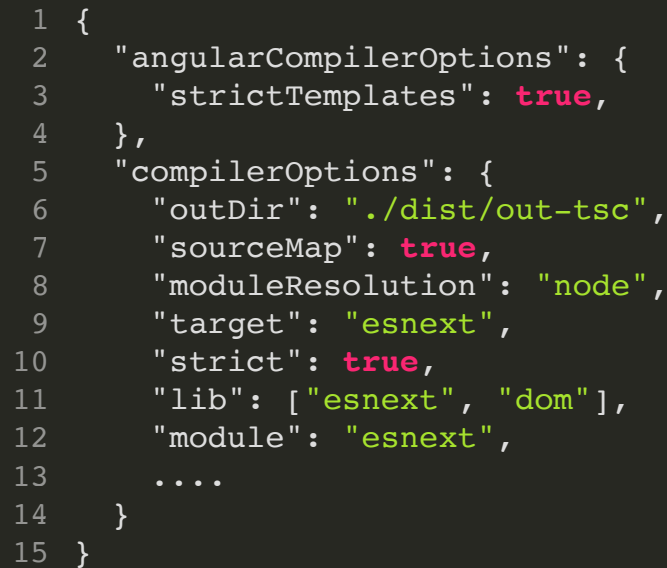
et pour transpiler (!= compiler):



```
tsc
```

# CONFIGURATION

Un fichier unique: **tsconfig.json**



```
1 {
2   "angularCompilerOptions": {
3     "strictTemplates": true,
4   },
5   "compilerOptions": {
6     "outDir": "./dist/out-tsc",
7     "sourceMap": true,
8     "moduleResolution": "node",
9     "target": "esnext",
10    "strict": true,
11    "lib": ["esnext", "dom"],
12    "module": "esnext",
13    ....
14  }
15 }
```


# LES TYPES

---



# LES VARIABLES

Format suivant: <type déclaration> <nom>: <type> = <valeur>;



```
1 const abc: string = "abc";  
2 let abx: string = "abx";  
3 var abz: string = "abz";
```

# TYPE DE BASE

- Les nombres: **number**
  - `ex:const age: number = 28;`
  - entier, décimale,...
- Les chaînes de caractère: **string**
  - `ex:const nom: string = "Daniel";`
  - Attention il existe aussi **String** (objet vs type primaire)
- Les tableaux: `<type>[]` ou `Array<type>`
  - `ex:const ages: number[] = [28, 29]`
  - `ex:const ages: Array<number> = [28, 29]`

# TYPE DE BASE

- Les booléen: **boolean**
  - ex: `isTrue: boolean = true;`
  - ex: `isTrue: true = true;`
  - ex: `isFalse: false = false;`
- les objets: **object**
  - ex: `const player: object = {}`
  - Tout objet hos objet primitif (number,string,boolean)
- "wildcard": **any**
  - ex: `objet: any = ....`

# TYPE DE BASE

- type inconnu: **unknown**
  - ex: `const retourAPI: unknown = getData();`
- retour "vide": `undefined`
  - `ex: const cVide = undefined`
  - valeur par défaut des variables
    - Si j'écris: `const cVideAussi;`
    - alors `cVide` vaut `"undefined"`
- retour "vide": `null`
  - `ex: const cVide2 = null;`
  - même comportement que `undefined`
  - préférer `undefined` (reco typescript)

# TYPE DE BASE

- aucun retour: **void**
  - quand une fonction ne retourne rien
  - (elle retourne en vrai undefined)
- aucun retour: **never**
  - similaire à void mais indique qu'aucun retour n'est fait
    - suite au lancement d'une exception par exemple


# LES FONCTIONS

---

# LES FONCTIONS


Ils existent plusieurs façon de crée une fonction

function "classique"




```
function maFonction(a: number, b: number): number {  
    return a + b;  
}
```

arrow function (depuis es6)



```
const maFonction: (a: number, b: number) => number =  
    (a: number, b: number): number => {  
        return a + b;  
    }
```



```
const maFonction = (a: number, b: number): number => a + b;
```

# LET'S GO

---

- Créer un nouveau projet typescript
  - pour initialiser le projet: `npx tsc --init`
- transformer ce code js en typescript
  - **<https://gist.github.com/danielpayetdev>**
    - récupérer le fichier index.js




# LES INTERFACES

---

# INTERFACES

Une interface défini le typage strict d'un objet.

Elle ce défini sous le format suivant:



```
1 interface MonInterface {  
2     proprieteA: MonTypeA,  
3     proprieteB: MonTypeB,  
4     ....  
5 }
```


# INTERFACES

Il est important de comprendre que les interfaces (et les types en générale) ne sont présent que lors de la phase d'écriture/transpilation, au runtime (= à l'exécution), elles n'existent plus !

# INTERFACES

Sur le tp vous avez du utiliser "object", c'est ok pour ce tp 1.


Pour être plus précis on aurait du écrit comme ça:




```
1 interface Contrat {  
2     reference: string;  
3     id: number;  
4 }  
5  
6 function getContrat(): Contrat {  
7     return {  
8         reference: getReferenceContrat(),  
9         id: getIdContrat()  
10    }  
11 }  
12  
13 const monContrat: Contrat = getContrat();
```

# INTERFACES

Imaginons qu'on veut ajouter une propriété sur Contrat mais qu'on sait qu'elle peut ne pas être définie: dans ce cas on ajoute ? devant le type :




```
1 interface Contrat {  
2     reference: string;  
3     id: number;  
4     referenceClient?: string;  
5 }
```



```
1 interface Contrat {  
2     reference: string;  
3     id: number;  
4     referenceClient: string | undefined;  
5 }
```

# INTERFACES

Imaginons maintenant qu'on ne veut pas autoriser la modification de la référence: on va alors utiliser readonly




```
1 interface Contrat {  
2     readonly reference: string;  
3     id: number;  
4     referenceClient?: string;  
5 }
```

**error** TS2540: Cannot assign to 'reference' because it is a read-only property.

# INTERFACES

L'héritage existe aussi sur les interfaces :



```
1 interface Vehicule {  
2     couleur: string  
3 }  
4  
5 interface Voiture extends Vehicule {  
6     clim: boolean  
7 }
```

# LET'S GO

---

- Créer l'interface Contrat du TP 1
- y ajouter une referenceClient (optionnel)
- marquer la reference en lecture seul.



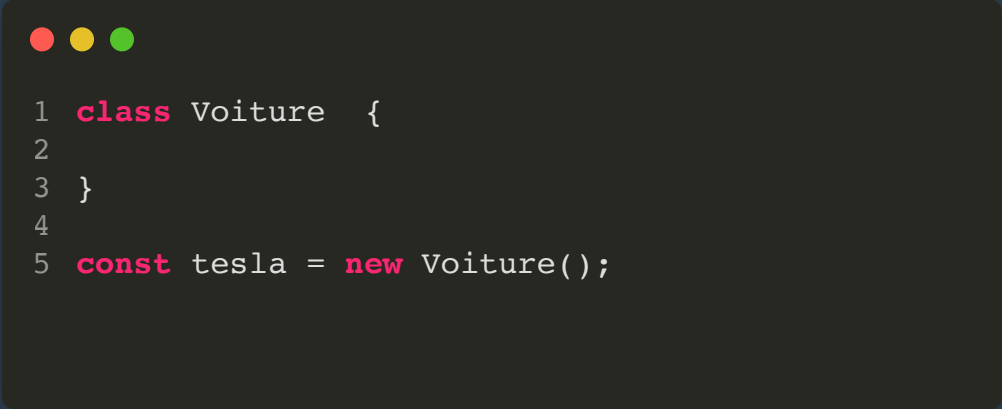
# LES CLASSES

---

# CLASSES

Les classes typescript sont très similaire à ce qui existe dans le monde de la programmation.

On va pouvoir y déclarer des propriétés, des méthodes,...




```
1 class Voiture {  
2  
3 }  
4  
5 const tesla = new Voiture();
```

# CLASSES

Le constructeur:

On va ajouter une méthode nommée.. **constructor()**

Il est optionnel si pas de paramètre en entrée (ou code d'init)



```
1  class Voiture {
2
3      constructor() {
4
5      }
6
7  }
8
9  const tesla = new Voiture();
```

# CLASSES

Les propriétés :

La syntaxe est quasiment la même qu'une variable.

On y accède avec le mot clé "this" (qui représente l'instance de la classe) depuis la classe

Ici on ajoute un paramètre au constructor pour initialiser la couleur:


Pour récupérer les propriétés en dehors de la classe on utiliser  
<monInstance>.<maPropriete>

```
1  class Voiture {
2    couleur: string;
3
4    constructor(couleur: string) {
5      this.couleur = couleur;
6    }
7
8  }
9
10 const tesla = new Voiture('blanc');
11 console.log(tesla.couleur);
```

# CLASSES

On peut aussi définir la portée de cette propriété :


- **public** => tout le programme y a accès (par défaut)
- **privé** => seules les méthodes internes de la classe ont le droit de voir
  - /!\ à l'exécution le contrôle n'existe pas), pour avoir une vraie propriété privée en js il faut préfixer la propriété par #
- **protected** => seules les classes filles ont accès



```
1 class Voiture {
2   public couleur: string;
3   private km: number = 0;
4   ...
5 }
6
7 const tesla = new Voiture('blanc');
8 console.log(tesla.couleur) // OK;
9 console.log(tesla.km) // ERREUR
```

# CLASSES

Comme sur les interfaces, on peut marquer une propriété en lecture seul avec readonly



```
1 class Voiture {
2     public readonly couleur: string = 'rouge';
3 }
4
5 const v = new Voiture();
6
7 v.couleur = "vert" // ERREUR !
```

# CLASSES

TS permet de définir une propriété directement depuis le constructeur en ajoutant `public/private/protected` (très utilisé en Angular)



```
1 class Voiture {  
2   public couleur: string;  
3  
4   constructor(couleur: string) {  
5     this.couleur = couleur;  
6   }  
7 }
```



```
1 class Voiture {  
2   constructor(public couleur: string) { }  
3 }
```

# CLASSES

Forcément, l'héritage existe aussi sur les classes.


Si un constructeur est ajouté sur la classe fille, il faut appeler `super()` en premier lieu de celui-ci

```
1 class Voiture {
2     constructor(public couleur: string) {
3     }
4 }
5
6 class TeslaModel3 extends Voiture {
7     constructor(couleur: string, public autonomie: string) {
8         super(couleur);
9         // Mon code spécifique ici
10    }
11 }
```



# CLASSES

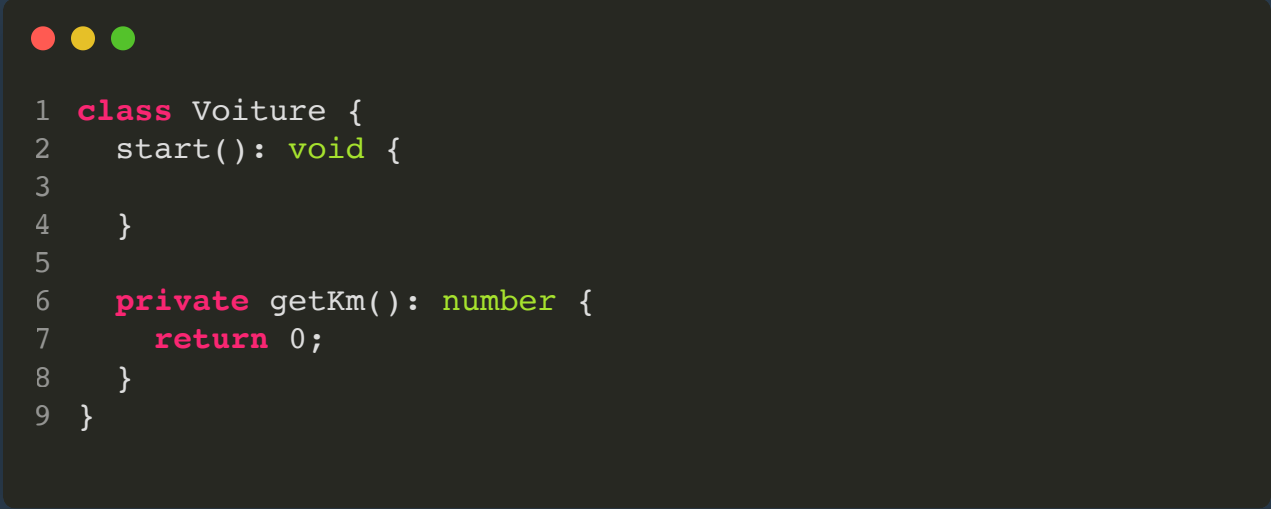
On peut aussi "implémenter" les interfaces :



```
1 interface Vehicule {  
2     const couleur: string;  
3 }  
4  
5 class Voiture implements Vehicule {  
6     const couleur: string;  
7     ...  
8 }
```

# CLASSES

Les méthodes: les mêmes info que les propriétés mais avec la syntaxe suivante (comme les fonctions au final, sans le mot clé function) :



```
1 class Voiture {  
2     start(): void {  
3  
4     }  
5  
6     private getKm(): number {  
7         return 0;  
8     }  
9 }
```

# LET'S GO

- Créer une classe Personne, avec:
  - nom
  - prenom
  - age
  - une méthode qui retourne nom prénom
    - pour concaténer: `\${this.nom} \${this.prenom}`
- une classe Developpeur qui hérite de personne:
  - avec une liste de language de programmation
- une classe Prof qui hérite de personne:
  - avec une liste de cours

# LES TYPES COMPLEXE


---

# INTERSECTIONS

Les intersections permettent de faire combiner plusieurs type en un seul.

ex: je veux une variable avec les propriétés A ET B.

On utilise alors le symbole & :



```
1 interface Voiture {
2     couleur: string
3 }
4
5 interface Camion {
6     poids: number;
7 }
8
9 const CyberTruck: Voiture & Camion;
10 // Equivaut à
11 // {
12 //     couleur: string;
13 //     poids: number;
14 // }
```

# UNION

Quand je veux que mon type soit l'un ou l'autre

Alors on va utiliser le symbole |

```
1 interface Voiture {  
2   couleur: string  
3 }  
4  
5 interface Camion {  
6   poids: number;  
7 }  
8  
9 const CyberTruck: Voiture | Camion;
```

ici CyberTruck sera soit une voiture, soit un camion mais pas les deux

# UNION

Pour distinguer le type on peut utiliser une fonction qui va permettre de choisir un type ou l'autre à l'aide de retour "in" + "is"

```
1 interface Voiture {  
2   couleur: string  
3 }  
4  
5 interface Camion {  
6   poids: number;  
7 }  
8  
9 const CyberTruck: Voiture | Camion;  
10  
11 function isCamion(tesla: Voiture | Camion): tesla is Camion {  
12   return "poids" in tesla;  
13 }
```

# ALIAS

Pour simplifier le code, on peut créer des alias avec la syntaxe suivante:

```
1 interface Voiture {  
2   couleur: string  
3 }  
4  
5 interface Camion {  
6   poids: number;  
7 }  
8  
9 type TrucMoche = Voiture | Camion; // <=  
10  
11 const CyberTruck: TrucMoche  
12  
13 function isCamion(tesla: TrucMoche): tesla is Camion {  
14   return "poids" in tesla;  
15 }
```

On peut faire des alias avec des types primitifs, des interfaces, d'autres alias, des unions, des intersections, etc..



# LES ENUMS

Permet de créer un ensemble de valeur:

```
1 enum MonEnum {  
2     VALEUR_1,  
3     VALEUR_2,  
4     VALEUR_3,  
5     ...  
6 }
```

Par défaut, les valeurs commencent à 0. Mais on peut changer, par exemple pour commencer à 1 :

```
1 enum MonEnum {  
2     VALEUR_1 = 1,  
3     VALEUR_2,  
4     VALEUR_3,  
5     ...  
6 }
```

# LET'S GO

---

En reprenant le tp précédant:

- Transformer les listes (language et cours) en enum
- ajouter le sexe sur personne ('masculin' ou 'feminin' ou 'autre')
- Créer un type Vacataire qui signifie être prof **et** developpeur

# LES GÉNÉRIQUES

# GENERIQUE

Permet la réutilisation de type,

un type générique qu'on a déjà vu: les tableaux !


- `const monTableau: Array<string> = [];`

Pour créer des types génériques on utilise la syntaxe suivante `<T>`

Sachant que `T` ici equivant à un `"any"` à la différence que le type peut être vérifié à la compilation, alors que `any` va cacher souvent les problèmes..

# GENERIQUE


Exemple de fonction générique:



```
1  function compteur<T>(valeur: T[]): number {  
2      return valeur.length;  
3  }  
4  
5  compteur([1,2,3]);  
6  compteur(['1', '2', '3']);  
7  compteur([true, false, true]);  
8  ...
```

# GENÉRIQUE

Exemple de classe générique:



```
1 class MaClasse<T> {  
2     maValeur: T;  
3  
4     getMaValeur(): T {  
5         return this.maValeur;  
6     }  
7 }  
8  
9 const toto = new MaClasse<string>();
```

# GENERIQUE

On peut restreindre le type générique:

ici seul les objets de type voiture (ou classe fille) sont accepté

```
1 class MaClasse<T extends Voiture> {  
2     maValeur: T;  
3  
4     getMaValeur(): T {  
5         return this.maValeur;  
6     }  
7 }
```

# GENERIQUE


On peut aussi rendre le type optionnel en lui donnant une valeur par défaut ! `<T = MonTypeParDefaut>`

```
1 class MaClasse<T = Voiture> {  
2     maValeur: T;  
3  
4     getMaValeur(): T {  
5         return this.maValeur;  
6     }  
7 }
```



# GENERIQUE

On peut lui passer plusieurs types génériques et le nom que l'on veut



```
1 class MaClasse<TypeDeMaValeur, TypeDeMaValeur2> {  
2     maValeur: TypeDeMaValeur;  
3     maValeur2: TypeDeMaValeur2;  
4 }
```

# LET'S GO

- Créer une interface **Article** (nom, prix)
- Créer plusieurs objets (Pomme, ...) qui étendent **Article**
- Créer une classe générique **Panier** avec une propriété qui stock les articles, une méthode pour ajouter un article, une pour supprimer, et une pour avoir le nombre d'article du panier
- Créer un Panier


# EXPRESS

Typescript génère des fichiers de définition de type : \*.d.ts

Ils sont partagé en plus des fichiers js.

On a aussi les "source map" qui permettent de faire le lien entre js et ts pour les IDE et le debug

Express n'a pas de type (full js), on doit alors installer une lib en plus:



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