



TypeScript – Fondamentaux

Animé par Mazen Gharbi

Two horizontal bars are positioned above the number '3'. The left bar is composed of a light blue segment followed by a dark blue segment. The right bar is composed of an orange segment followed by a red segment.

3

Types

Définition d'un type

▷ Différents types possibles :

- › String : Type primitif javascript
- › Number : Type primitif représentant les integer / flottants
- › Boolean : true / false
- › Any : Tout type de variable
- › Any[] : Tableau contenant tout type de données
- › Void : L'absence de valeur (undefined)
- › () => any : Type représentant une fonction renvoyant n'importe quoi

▷ Application :

```
let num: number = 1;  
let str: string = 'Chaine';  
let arr: string[] = ['Ch1', 'Ch2'];
```

Test d'un callback

▷ Voici une fonction qui attend un callback en paramètre :

```
function attendCallback (cb: (error: Error) => any) {  
    return cb(null);  
}
```

```
attendCallback(function (err) {  
    // Réaction  
});
```

Typage des Interfaces

```
interface MonInterface {  
    value: string; // Possède une propriété de type `string`.  
    method(): number; // De type `number`.  
    (): boolean; // C'est une fonction retournant un `boolean`.  
}
```

▷ Les interfaces peuvent également utiliser des types d'index, qui peuvent être des chaînes ou des nombres.

```
interface Dictionnaire {  
    [index: string]: string;  
}
```

Le transpileur vous fait confiance..

▷ TypeScript vous permet également de remplacer le type inféré / analysé par ce dont vous avez réellement besoin. Ceci est utilisé uniquement pour dire au compilateur que vous connaissez le type mieux que lui et il ne devrait pas deviner lui-même.

```
interface Foo {  
    x: number;  
    y: number;  
}
```

// Cette ligne passera sans problème

```
const foo = {} as Foo;  
foo.x = 10; // OK !  
foo.y = 20; // OK !  
foo.z = 'hello :)'; // Whoops, erreur :/ Property z does not exist...
```

Interface

▷ Ici, on aura évidemment une erreur car le contrat n'est pas respecté

```
interface Foo {  
  x: number;  
  y: number;  
}
```

```
const foo: Foo = {}; // Erreur !
```

Tableau multi-types

▷ Comment on fait si on veut un tableau contenant à la fois des String / Number et Boolean? ...

```
let mixedArray: Array<string | number | boolean>;  
  
mixedArray = [42, 'Tout est', 'OK', true];  
  
console.log(mixedArray);
```


Types abrégés

▷ Les types peuvent également être utilisés avec une syntaxe abrégée; par exemple, dans les paramètres, on peut directement décrire une interface. Il est également possible d'utiliser l'alias de type pour créer nos propres types :

```
type Toto = string;  
type MultiArray = Array<string | number | boolean>  
  
function salut(): Toto {  
    return 'Hello :);'  
}  
  
const mixedArray: MultiArray = [42, 'Tout est', 'OK!', true];  
  
// On déclare une interface directement  
function getMessage(obj: { message: string }): string {  
    return obj.message;  
}
```

Guard types

▷ Typescript par défaut considère qu'un type « any » passe la validation de n'importe quel interface. Les « Guard Types » sont une vérification supplémentaire permettant d'éviter que n'importe quel « any » puisse passer au runtime

```
interface Foo {  
    foo: string;  
}
```

```
function typeGuardFoo(arg: any): arg is Foo {  
    return arg.foo !== undefined;  
}
```

```
function doStuff(arg: Foo) {  
    if (typeGuardFoo(arg)) {  
        console.log(arg.foo); // OK  
    }  
}
```

Retourne false au runtime si les propriétés ne sont pas conformes à l'interface

Generics

- ▷ Vous vous souvenez des Generics ?
- ▷ On peut également appliquer le Generic actuel aux paramètres des méthodes

```
function toto<T>(param: T): T {  
    return param;  
}  
  
const result = toto<number>(1);  
const error = toto<string>(2); // ERREUR !
```

▷ Un type d'intersection représente un mélange de plusieurs types ! Par exemple :

```
interface I1 { // Interface 1
  a: number;
}

const element1: I1 = {a: 1}; // Objet 1

interface I2 { // Interface 2
  b: number;
}

const element2: I2 = {b: 2}; // Objet 2

function melange<A, B>(a: A, b: B): A & B { // A & B => Le resultat sera un mélange des 2 types
  Object.keys(b).forEach(key => {
    a[key] = b[key];
  });

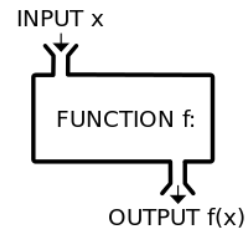
  return a as A & B;
}

// On attend en résultat un mélange des 2 types envoyés ;')
const result: I1 & I2 = melange<I1, I2>(element1, element2);
```

Définitions de tuples

▷ Un tuple est un ensemble de données organisés dans un ordre spécifique. Chaque donnée peut évidemment être typée

```
let notreVariable: [string, number];  
// Initialisation  
notreVariable = ['Yo', 42]; // OK  
notreVariable = [42, 'Yo']; // Erreur !!
```



Les Fonctions

Plusieurs manières de déclarer une fonction

```
function hello() {  
    console.log('Salut !');  
}
```

```
let hello = function() {  
    console.log('Salut !');  
};
```

```
let hello = () => {  
    console.log('Salut !');  
};
```

Mot-clé this

- ▷ Le mot clé **this** représente l'entité qui l'appelle et non pas l'entité qui la déclare !

```
class A {  
    public bonjour() {  
        setTimeout(function () {  
            this.disBonjour();  
        }, 1000);  
    }  
  
    private disBonjour() {  
        console.log('Bonjour à tous !');  
    }  
}  
  
let a = new A();  
a.bonjour(); // ERREUR ICI !
```

Le problème vient d'ici. Le **this** ne fait plus référence à la classe

Plusieurs solutions 😊

```
public bonjour() {  
    let self = this; // La solution la moins... honorable dirons nous  
    setTimeout(function () {  
        self.disBonjour();  
    }, 1000);  
}
```

```
public bonjour() {  
    // C'est déjà un peut mieux !  
    setTimeout(function () {  
        this.disBonjour();  
    }.bind(this), 1000);  
}
```

LA solution

```
class A {  
    public bonjour() {  
        setTimeout(() => { // Arrow function  
            this.disBonjour();  
        }, 1000);  
    }  
  
    private disBonjour() {  
        console.log('Bonjour à tous !');  
    }  
}  
  
let a = new A();  
a.bonjour();
```

Paramètres

```
function toto(a: number, b: number): number {  
    return a + b;  
}  
toto(1, 2);
```

Type de retour



Mais on peut également faire ainsi :

```
function toto({a, b}: {a: number, b: number}): number {  
    return a + b;  
}  
toto({a: 1, b: 2});
```

Typing a variable

- In typescript, you can also type a variable called « functional » :

```
let toto: (a: number, b: number) => number;
```

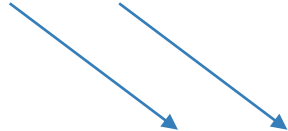


Déclaration du prototype de la fonction

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

Typing une variable : déstructuration objet

```
let toto: ({a, b}: { a: number, b: number }) => number;  
  
toto = function ({a, b}: { a: number, b: number }) {  
    return a + b;  
};
```



i *Les noms de paramètre doivent rester les mêmes entre la déclaration et la création de la fonction*

Paramètres optionnels et par défaut

```
function add(a: number, b?: number) {  
    if (b) {  
        return a + b;  
    } else {  
        return a;  
    }  
}
```



Indique que le paramètre est optionnel

Ou encore mieux :

```
function add(a: number, b?: number = 0) {  
    return a + b;  
}
```

Nombre de paramètres variable

Rest parameter

```
function add(...numbers) {  
    let result = 0;  
  
    for (const num of numbers) {  
        result += num;  
    }  
  
    return result;  
}  
  
add(1, 2, 4, 6);
```

Type de retour

- ▷ Une fonction peut évidemment signaler quel type de variable elle va renvoyer

```
function toto(): number {  
    return 1;  
}
```

```
function doNothing() {  
    // Rien :(  
}
```

- ▷ Sachez que l'on peut spécifier plusieurs valeurs de retour...

```
function multiRetour(a: boolean): number | string {  
    if (a) {  
        return 1; // Je renvoie un nombre  
    } else {  
        return 'Chaine'; // Ou une chaine !  
    }  
}
```




Classes

Une classe simple

```
export class Personne {  
  prenom: string;  
  
  constructor(prenom: string) {  
    this.prenom = prenom;  
  }  
  
  sePresenter(): void {  
    console.log('Bonjour, je m\'appelle ' + this.prenom);  
  }  
}
```

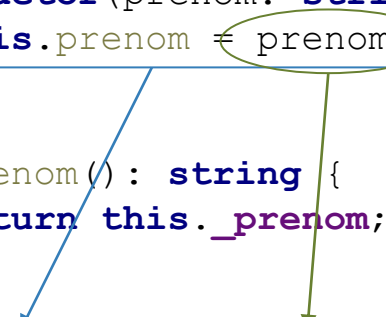
Principe d'encapsulation
non respecté



personne.class.ts

Une classe simple

```
export class Personne {  
  private _prenom: string;  
  
  constructor(prenom: string) {  
    this.prenom = prenom;  
  }  
  
  get prenom(): string {  
    return this._prenom;  
  }  
  
  set prenom(nouveauPrenom: string): void {  
    this._prenom = nouveauPrenom;  
  }  
}
```



C'est mieux comme ça !

personne.class.ts



Tous vos attributs doivent être private par défaut, puis si vous avez besoin de les récupérer de l'extérieur, faites un accesseur

Créer une instance

```
let toto: Personne; // Custom Type

toto = new Personne('Toto');
toto.sePresenter();

console.log(toto._prenom); // ERREUR ICI !
console.log(toto.prenom()); // Erreur aussi
console.log(toto.prenom); // affiche 'Toto'
```

main.ts

 La surcharge de constructeur n'existe pas en TypeScript

Portée des propriétés

```
export class Personne {  
    private _prenom: string;  
    protected _age: number;  
    private _taille: number;  
    readonly nom: string;
```

} Même principe pour les méthodes

Public : Accessible partout à la lecture et à la modification

Private : Accessible seulement au sein de la classe actuelle

Protected : Accessible à la classe actuelle et aux enfants seulement

Readonly : Accessible partout mais en lecture seulement

Héritage

- ▷ L'un des modèles les plus fondamentaux de la programmation basée sur les classes est la possibilité d'étendre les classes existantes pour en créer de nouvelles en utilisant l'héritage.
- ▷ Une classe B qui hérite d'une classe A récupèrera ainsi l'ensemble des propriétés et méthodes déclarées par la classe mère. La portée des entités héritées doit être « protected » ou « public »
- ▷ Inheritance is evil. Et vous, qu'en pensez-vous ?

Héritage

```
class A {  
  public a: string = 'A';  
  protected b: string = 'B';  
  private c: string = 'C';  
}
```

```
class B extends A {  
  constructor() {  
    super();  
    console.log(this.a);  
    console.log(this.b);  
    console.log(this.c); // ERREUR ICI  
  }  
}
```

Ce mot-clé est intéressant! Il permet une entité de la classe mère, ici le constructeur !

error TS2341: Property 'c' is private and only accessible within class 'A'.

```
let b = new B();
```

Héritage - super

```
class A {  
    public methode() {  
        console.log('parent');  
    }  
}  
  
class B extends A {  
    constructor() {  
        super();  
        super.methode(); // On appelle la méthode parent  
        this.methode(); // Méthode enfant  
    }  
  
    methode() {  
        console.log('enfant!');  
    }  
}  
  
let b = new B();
```


Héritage - super

```
class A {  
    public methode() {  
        console.log('parent');  
    }  
}  
  
class B extends A {  
    constructor() {  
        super();  
        super.methode(); // On appelle la méthode parent  
        this.methode(); // Méthode parent aussi  
    }  
}  
  
let b = new B();
```

Classe abstraite

- Les classes abstraites sont des classes de base à partir desquelles d'autres classes peuvent être dérivées. Ils ne peuvent pas être instanciés directement.

```
abstract class EtreHumain {  
    abstract respirer(): void;  
  
    vivre(): void {  
        console.log('Je vis!');  
    }  
}
```

etre-humain.class.ts

- Contrairement à une interface, une classe abstraite peut contenir des détails d'implémentation pour ses membres. Le mot-clé `abstract` est utilisé pour définir des classes abstraites ainsi que des méthodes abstraites au sein d'une classe abstraite.



Annotations

Définition

- ▷ Un décorateur est une déclaration qui peut être appliquée sur différents types de données :
 - › Classes ;
 - › Méthodes ;
 - › Accesseurs ;
 - › Propriétés ;
 - › Paramètres.

- ▷ Toujours préfixé d'un « @ » ;

- ▷ Exécuté au moment du « runtime »

Création & Mise en place

▷ Création :

```
function f(param) {  
    return function (target, prop) {  
        console.log('Appelé :');  
    };  
}
```

▷ Mise en place :

```
@f(1)  
methode() {  
    console.log('Hello');  
}
```



TypeScript – Aller plus loin

Animé par Mazen Gharbi



Surcharger une fonction

Plusieurs prototypes

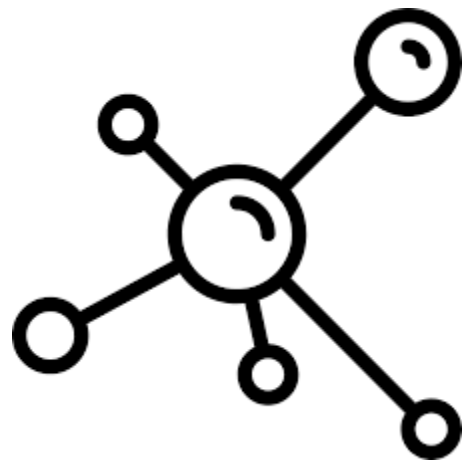
- ▷ En TypeScript, impossible d'avoir plusieurs fonctions portant le même nom
 - › Mais il existe une astuce.
- ▷ On va créer plusieurs prototypes pour un même fonction
- ▷ Nous traiterons les paramètres dans la fonction finale pour repérer le cas de figure dans lequel nous sommes

Action

```
function f(a: number): void;  
function f(a: string): void;
```

```
// Et ainsi de suite ..
```

```
function f(param: any): void {  
    if (typeof param === 'number') {  
        console.log('J\'ai un nombre!!');  
    } else if (typeof param === 'string') {  
        console.log('J\'ai une chaine de caractères!!');  
    }  
    // etc...  
}
```



Mixins (héritage multiple)

Héritage multiple

- ▷ L'**héritage multiple** est le principe qui dit qu'une classe peut hériter de plusieurs autres classes à la fois. Ainsi, elle récupère toutes leurs méthodes ;
- ▷ En **Typescript**, comme dans beaucoup d'autres langages, **cela n'est pas possible** nativement.. On va malgré tout y arriver avec quelques petites manipulations techniques !
- ▷ C'est le principe de **mixer** plusieurs classes dans une seule

Etape 1

▷ Créons nos 2 classes mères :

```
class Rhinoceros {  
    coupDeCorne(): void {  
        console.log('Coup de corne!');  
    }  
}
```

rhinoceros.ts

```
class Cheval {  
    galoper(): void {  
        console.log('Je galope');  
    }  
}
```

cheval.ts

Etape 2

▷ Puis on crée notre classe fille qui **implémente** les 2 classes et qui déclare 2 propriétés correspondants aux fonctions dont elle a besoin :

licorne.ts

```
class Licorne implements Rhinoceros, Cheval {  
    public coupDeCorne: () => void;  
    public galoper: () => void;  
}
```

Dernière étape

▷ Il faut maintenant appeler une fonction (fournie par TypeScript) pour appeler le mixin et lier les fonctions des parents à l'enfant :

main.ts

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {  
    baseCtors.forEach(baseCtor => {  
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {  
            derivedCtor.prototype[name] = baseCtor.prototype[name];  
        });  
    });  
}
```

```
applyMixins(Licorne, [Rhinoceros, Cheval]);
```

Questions