Introduction

C'est quoi Typescript ?

TypeScript est un **langage de programmation libre et open source** développé par Microsoft qui a pour but **d'améliorer et de sécuriser** la production de code JavaScript. C'est un **sur-ensemble** de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.

TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-contraignante de JavaScript. Il supporte la spécification ECMAScript 6.

Installer Typescript

Typescript n'est pas un langage directement disponible sur le système, il n'est pas non plus interprété directement au sein du navigateur, comme Javascript. Pour l'utiliser, nous avons besoin de l'installer sur la machine.

Il existe deux façons d'installer Typescript:

- En installant **Visual Studio** et le plugin Typescript
- En passant par le gestionnaire de paquets de node.js, **npm**

ts_install.js

```
#permet d'installer typescript au sein du dossier courant
npm install typescript

#ou


#permet d'installer typescript de façon globale sur la machine
npm install -g typescript


#puis taper la commande suivante pour obtenir le numéro de version
tsc -v
```

Un Hello World en Typescript

Voici un programme de base écrit en typescript, un 'hello world'. La syntaxe ressemble énormément à Javascript mais vous pouvez noter que l'on utilise un type de données précis pour notre variable, le type 'string' qui correspond à une chaîne de caractères.

ts_hello_world.ts

```
// on déclare une variable str contenant le chaîne "hello world"
let str:string = "hello world";

// puis on l'affiche
console.log(str);
```

Notez bien que l'on ne peut se servir directement de ce fichier source, en effet, aucun navigateur ou programme connu n'interprète le typescript de façon 'native'. Il faut donc passer par une étape supplémentaire nommée **transpiling** en anglais. Ce terme peut être traduit par **transcompilation**.

Transpiling

Le Typescript n'est pas (encore) un langage destiné à être directement interprété par le navigateur ou par tout autre programme. Son but est d'offrir une **extension** à Javascript, un **sur-ensemble de fonctionnalités**

Il faut donc passer par une étape supplémentaire nommée **transpiling** en anglais. Ce terme peut être traduit par **transcompilation**.

ts_compiling.js

```
#ici on "transcompile" le fichier "monfichier.ts" et l'on
#obtient ainsi un fichier équivalent, traduit en javascript

tsc monfichier.ts
```

Bases

Les types de données de bases en Typescript

- **Boolean**: Une simple valeur pouvant être vraie (true) ou fausse
- **Number**: Un nombre flottant (ex: 1.5)
- **String**: Une chaîne de caractères (ex: "Hello World")
- **Any**: Un type désignant "n'importe quel type de données", équivalent du type object en javascript.
- **Void**: Un type désignant du vide, rien, aucune valeur, souvent utilisé pour préciser qu'une fonction/méthode ne retourne pas de valeur (on attend du vide, donc rien)
- **Enum**: Un type de données permettant de créer son propre type de données customisé avec un choix prédéfini dans les valeurs (ex: enum Color {Red, Green, Blue}).
- **Array**: Un tableau pouvant contenir toutes sortes d'éléments (ex: [10,true,"google"]). On peut également définir le type de données d'un tableau (ex: let tab:Array;)
- **Null and Undefined**: Equivalent des types null et undefined Javascript.

Il en existe plein d'autres que vous pourrez retrouver sur la documentation en ligne du langage Typescript à www.typescriptlang.org

Templates Strings

Les "templates strings" permettent d'insérer plus facilement des valeurs de variables au sein d'une chaîne de caractère, ces dernières peuvent également être définies sur plusieurs lignes.

Il s'agit d'une nouveauté de la norme ES6, mais comme Typescript est un sur-ensemble de Javascript, elles sont supportées.

template_string.js

```
let jedi = {surname:"Obiwan", name:"Kenobi"};
let msg = `${jedi.name} ${jedi.surname} is the Jedi Master`;
console.log(msg);
```

Les variables

En Typescript, on n'emploie plus le mot clé **var** pour déclarer une variable mais le mot clé **let** et le mot clé **const**.

Le mot clé **const** est utilisé, comme son nom l'indique, pour déclarer une valeur **constante**, çàd une valeur ne pouvant être modifiée, si vous essayez de modifier une constante, une erreur est levée.

Le mot clé **let** est quand à lui utilisé pour déclarer une valeur variable, çàd une valeur capable de changer au cours de la durée de vie de votre programme.

Contrairement à Javascript, qui n'est pas typé, le Typescript l'est lui, comme son nom l'indique. Il est donc utile de préciser le type de données que l'on souhaite stocker au sein de la variable (ou de la constante) à l'aide de la syntaxe suivante:

ts_variables.ts

```
// constante, ne peut être modifiée
const LIGHT_SPEED:string = "299 792 458 m / s";
// variable, peut être modifiée
let msg:string = "Hello World";
```

Les fonctions

En Typescript comme en Javascript, les fonctions sont incontournables. Il s'agit de blocs d'instructions répétables que l'on peut appeler (on dit invoquer) autant de fois que nécessaire.

Les fonctions Typescript, contrairement aux fonctions Javascript, précisent le type de données qu'elles renvoient, et si elles ne renvoient rien, alors le type de retour est **void** (du vide)

ts_functions.ts

```
// les paramètres sont typés, et le type de la donnée
// retournée également à l'aide de la syntaxe suivante:
function sum( a:number, b:number ):number{
    return a + b;
}

// quand une fonction ne retourne aucune donnée
// alors on le précise en utilsant le type "void"
function notif( msg:string ):void{
    alert(msg);
}

// les fonctions anonymes sont également utilisables, comme en JS
let anonymous:Function = function():void{
    console.log("I am an anonymous function");
};

// on peut invoquer une fonction anonyme comme en JS
anonymous();
```

Les fonctions fléchées

En Typescript comme en Javascript ES6, les fonctions fléchées sont supportées, bien entendu le typage des paramètres et de la donnée de retour est à ajouter au sein de la version Typescript. Il est également possible de spécifier des valeurs par défaut aux paramètres. Les fonctions fléchées ont également l'avantage de préserver le contexte dans lequel elles sont déclarées.

ts_arrow_functions.ts

```
// fonction fléchée anonyme avec valeur de parmètre par défaut
let hello = (param_user: string = "user"): void => {
    console.log("Hello", param_user);
};

hello(); // invoquons cette fonction

// les fonctions fléchées anonymes préservent le contexte
class Gandalf {
    name: string = "Gandalf";
    introduce = () => {
        console.log("Hello my name is", this.name);
    }
}

// testons notre code
new Gandalf().introduce();
```

Les boucles

En Typescript, les boucles fonctionnent comme en Javascript:

ts_loops.ts

```
let i:number = 10;

// une boucle for classique en Typescript
for( i = 0; i < 10; i ++){
    console.log(i);
}

// une boucle while classique en Typescript
i = 10;
while( --i > -1 ){
    console.log(i);
}

// une boucle do while classique en Typescript
i = -1;
do{
    if( i == -1 ){
        i = 10;
    }
}while( --i > -1)
```

Les tableaux

TypeScript, comme JavaScript, permet de travailler avec des **tableaux** de valeurs. A la différence qu'en Typescipt, les tableaux (si on le souhaite) peuvent être **typés** ,ce qui signifie qu'ils ne peuvent contenir **qu'un seul type de données**. La déclaration de type pour les tableaux peut s'écrire de deux façons :

ts_arrays.ts

```
// premiée façon de déclarer un tableau typé
let notes:number[] = [
    0,10,12,7,8,20,13,15
];

// seconde façon de déclarer un tableau typé
let notes2:Array<number> = [
    0,10,12,7,8,20,13,15
];
```

Les enums

TypeScript nous permet de créer et d'utiliser des **enums**. Les enums permettent au développeur de créer un lot de **constantes** et de les regrouper de façon à former un nouveau **type de données**.

ts_enums.ts

```ts
// créons un type de données "Direction"
// qui peut prendre 4 valeurs différentes
enum Direction { Up = 1, Down = 2, Left = 3, Right = 4 };

let haut: Direction = Direction.Up;
let bas: Direction = Direction.Down;
let gauche: Direction = Direction.Left;
let droite: Direction = Direction.Right;

console.log(haut, bas, gauche, droite);
```

Modules

A partir de ECMAScript 2015 (ES6), JavaScript introduit le concept de modules, et bien entendu, ce concept est supporté en Typescript

Les modules sont éxécutés au sein de leur propre portée, et non pas au sein de la portée globale. Cela signifie que les variables, fonctions, classes, etc... Déclarées au sein d'un module ne sont pas visibles en dehors du module à moins qu'elles ne soient explicitement exportées à l'aide du mot-clé **export**. De fait, pour utiliser un élément exporté, au sein d'un module différent, il faut utiliser le mot-clé **import**.

ts_modules_1.ts

```
export function toto(){
    console.log("toto est beau");
}
```

ts_modules_2.ts

```
import {toto} from './ts_modules_1';

toto();
```

Typescript & POO

Programmation orientée objet (POO)

La programmation orientée objet est un style de programmation nous permettant de représenter des concepts informatiques sous forme d'objets. Pour conceptualiser un objet, nous avons besoin de plusieurs outils directement intégrés au langage. Il existe également plusieurs façons de "coder objet", Javascript est un langage **orienté objet par prototypage**, mais **Typescript** lui émule le comportement d'un **langage orienté objet par classe**. Il est important de retenir qu'un objet possède plusieurs caractéristiques:

- Un objet possède des **propriétés**, çàd des variables qui lui appartiennent en propre, la portée de ces propriétés peut être, publique, protégée ou privée.

- Un objet possède également des **méthodes**, çàd des fonctionnalités (fonctions) qui lui appartiennent en propre, la portée de ces méthodes peut être, publique, protégée ou privée.

- Dans un langage orienté objet par classe, un objet peut **hériter** d'un autre objet.

- Un objet peut prendre **plusieurs formes** et rédéfinir ses propriétés héritées

Et il ne s'agit ici que des caractéristiques minimum, çàd celles qui définissent la base de la base d'un langage orienté objet, on parle des principes **d'encapsulation, d'héritage et de polymorphisme**.

Les classes

ts_classes.ts

```
class Hero{
    // une propriété peut être publique, protégée ou privée
    public name:string;
    public power:string;

    // la fonction constructrice est invoquée automatiquement
    // à la création d'un nouvel objet
    constructor(
        param_name:string,
        param_power:string
    ){
        // on attribue à nos propriétés les valeurs passées
        // en paramètre
        this.name = param_name;
        this.power = param_power;
    }

    // une méthode peut être publique, protégée ou privée
    public sayMyName():void{
        console.log(this.name);
    }

    public sayMyPower():void{
        console.log(this.power);
    }
}

let myHero:Hero = new Hero("Batman", "Being rich");
myHero.sayMyName();
```

Héritage

ts_inheritance.ts

```
class Personnage {
    // on veut transmettre cette propriété à nos enfants, on utilise donc protected
    protected name: string;
    // on veut transmettre cette propriété à nos enfants, on utilise donc protected
    protected lifepoint: number;

    constructor(param_name: string, param_lifepoint: number) {
        this.name = param_name;
        this.lifepoint = param_lifepoint;
    }

    //  on doit pouvoir demander à un personnage s'il est mort
    // sans être soi-même un personnage, la portée est donc publique
    public isDead(): boolean {
        // si le nombre de points de vie est inférieur
        // ou égal à 0 alors le personnage est mort
        if (this.lifepoint <= 0) {
            return true;
        }
        else {
            return false;
        }
    }

    public sayMyName():void{
        console.log(this.name);
    }
}
```

Héritage

ts_inheritance2.ts

```
class Wizard extends Personnage{
    constructor( ){
        // on peut invoquer le constructor de la classe parente à l'aide de "super"
        super("Gandalf", 100);

        //... maintenant un nouvel objet de type wizard se nommera toujours
        // Gandalf et aura 100 points de vie
    }

    // on réécrit la méthode sayMyName définit par le parent
    // et héritée de celui-ci.
    public sayMyName():void{
        // mais on peut toujours invoquer l'ancienne "version" de la méthode
        // toujours à l'aide de "super"
        super.sayMyName();

        // si on veut, on peut ajouter des opérations supplémentaires
        console.log("I am a super magician !");
    }
}

// on crée un nouveau sorcier
let gandalf:Wizard = new Wizard();
// on crée un personnage Gollum qui possède 2 points de vie
let gollum:Personnage = new Personnage("Gollum", 2);

gandalf.sayMyName();
gollum.sayMyName();
```