TypeScript

Table des matières

Introduction

Prérequis

- La connaissance des fondamentaux Javascript
- Les bases en programmation orienté objet(type,classes, interfaces,..) ne sont pas requises mais seront un grand plus pour comprendre les concepts objets présents dans TypeScript

Historique

TypeScript est un langage de programmation libre et open source mis au point par Microsoft, notamment par Anders Hejlsberg le principal inventeur de C#, pour qui le JavaScript n'est pas fait pour les projets à grandes échelles.
Cependant, les navigateurs ne l'acceptant pas nativement, il est transformer en JS...

En 2012, après 2 ans de développement en interne, une première version de TypeScript est rendue publique...
Malgré tout, il faudra attendre 2017, voir 2019 pour voir la popularité de TS exploser

Présentation – C’est quoi ?

- Dérivé du JavaScript???
- Trans-compilation en JavaScript
- Facile à Intégrer dans un projet Angular, React, VueJs
- Intégration Js/Typescript possible

Présentation – Pourquoi l'utiliser

- Typage statique
- Programmation Orienté Objet
- Fonctionnalités récentes de JS
- Syntaxe proche de C#/C++/Java

Présentation - Comment ça marche ?

Les outils

Les outils à utiliser

- Node.Js (version LTS)
- Visual Studio Code (ou un autre IDE)
- Un navigateur internet (Chrome, firefox, edge...)
- TypeScript (dernière version)

Installation des Outils – Node.js

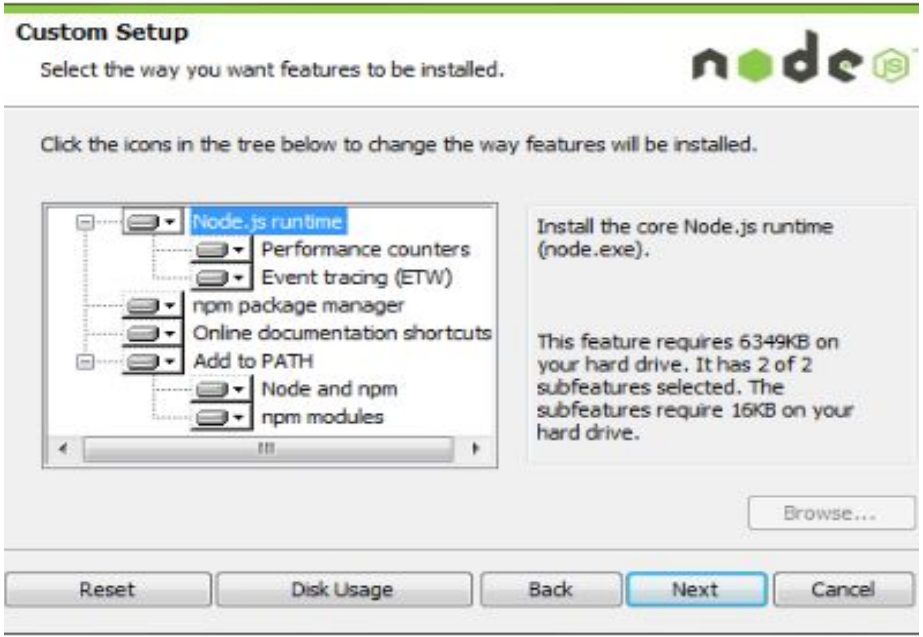
Pour Node.JS rien de bien compliqué

https://nodejs.org/en/

Installation des Outils – Node.js

- Exécutez le fichier, acceptez la licence, choisissez le dossier "Program files", ce qui installera :
  - **L'exécuteur node.js** qui permet d'exécuter les fichier JS
  - **Le module npm** qui permet d'ajouter / retirer les libraires que nous utiliserons dans nos application
  - **Un raccourci vers la documentation en ligne**.
  - **Des variables d'environnements** pour pouvoir utiliser des commandes node en invite de commande

Installation des Outils – Node.js



Pour vérifier que tout c'est bien passer il suffit d'ouvrir une invite de commande et de faire

node –v

La réponse en console doit être le numéro de la version nodejs installée.

Installation des outils

TypeScript s'installe via l'invite de commande

La ligne de commande à utiliser pour une installation globale est:

```
npm install –g typescript
```

Si je veux installer en local uniquement

```
npm install --save typescript
```

Installation des outils

Global vs Local

Local : installés dans le répertoire où vous exécutez npm install <nom-package>, et ils sont placés dans le dossier node_modules sous ce répertoire

Global : Placés dans un seul endroit de votre système (dépend de votre configuration), quel que soit l'endroit où vous exécutez npm install -g <nom-package>

Attention : Dans votre code, vous pouvez faire un *require* que pour les packages locaux.

En général, **tous les packages doivent être installés localement** pour permettre la réutilisation de nos modules et éviter les conflits de version
Un package sera installé **globalement lorsqu'il fournit une commande exécutable** que vous exécutez à partir du shell (CLI) et qu'il est ré-utilisé dans tous les projets.

Installation des outils

Et ensuite tsc –v afin de vérifier que tout c'est bien passé.

Comme avec nodejs, la réponse doit être le numéro de la version installée.

```
PS E:\Documents\Cognitic\Web\TypeScript\TS\Sample> tsc -v
Version 4.1.3
```

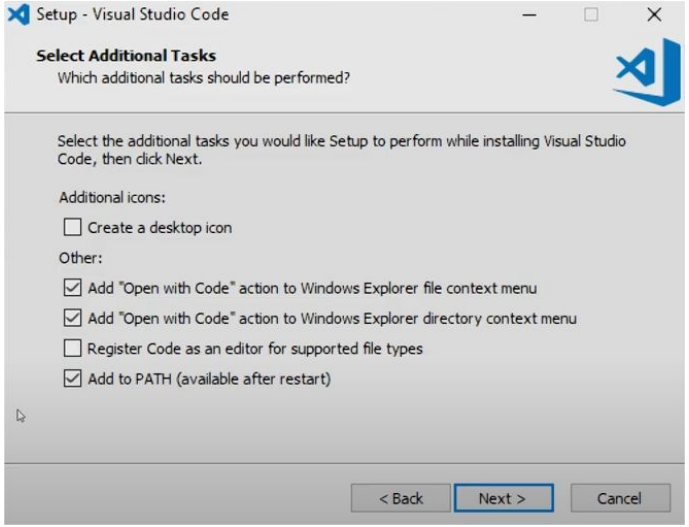
Installation des outils

Enfin Visual Studio Code

https://code.visualstudio.com/

Lors de cette installation je vous invite à cocher les 2 « add open with Code » action.

Ca permettra entre autre de pouvoir faire un clic droit sur un fichier, ouvrir avec et de pouvoir choisir vs code.



Démarrage du compilateur

Pour démarrer le compilateur il suffira de taper dans la console npx tsc –watch

Variables, types et typage

Les variables

Qu'est ce qu'une variable?

Au travers de nos programmes , nous allons devoir stocker temporairement des informations afin de les manipuler, que ce soit sous forme de calculs, de texte,...

Pour ce faire, il faut déclarer un conteneur, une sorte de tiroir dans la mémoire vive qui contiendra ces informations. C'est ce qu'on appel une variable

Les types

Le typage c'est quoi?

Il faut déjà comprendre ce qu'est un type de donnée. C'est ce qui définit la nature des valeurs que pourra recevoir une variable ainsi que les opérateurs qui pourront lui être appliqué.

Le typage c'est ce qui permet d'assigner un type à une variable. En Typescript, il y a deux méthodes pour y parvenir détaillée plus loin, mais d'abord regardons au différents type possible.

Les types

- boolean, le type plus basique qui soit, il retourne simplement vrai ou faux.

```
let isDone: boolean = false;
```

- number, en Typescript tous les types de nombre se retrouvent ici, que ce soit entier, réel, hexadécimal, binaire...

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

Les types

- string, c'est le type qui s'occupe de toutes les données textuels comme un nom ou un prénom.
- array, ce sont des tableaux de valeur, ils peuvent être déclarer de deux façons différente :
  type[] ou Array<type>

```
let color: string = "blue";
```

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

Les types

- unknow, c'est un type utilisé lorsque nous déclarons une variable dont nous ignorons encore le type lors de la conceptions de l'application.

```
let notSure: unknown = 4;
notSure = "maybe a string instead"

// OK, definitely a boolean
notSure = false;
```

- any, permet à une variable d'accepter n'importe quel type de valeur.

```
let looselyTyped: any = 4;
```

Les types

- void, c'est une absence de type, qu'on verra plus communément dans la signature d'une méthode pour dire qu'elle ne retourne rien.
- null et undefined, ce sont des sous-types des autres type, permettant d'assigner à un nombre une valeur null par exemple

```
function warnUser(): void {
  console.log("This is my warning message");
}
```

```
let u: undefined = undefined;
let n: null = null;
```

Les types

- never, type peu utilisé mais il est intéressant de savoir comment l'utiliser.

  C'est quelque chose qui n'arrive jamais, on va l'utiliser avec une fonction jette (throw) une erreur.

```
function stringOrNumber(x): boolean
{
    if(typeof x === "number"){
        return false;
    } else if (typeof x === "string"){
        return true;
    }
    return fail("ni un string, ni un nombre!");
}

function fail(message: string): never {
    throw new Error(message);
}
```

Les types

Il reste 3 types basique qui seront détaillé plus loin dans le cours:

enum, tuple et object qui auront leur chapitre propre.

Mais avant de les aborder, nous allons nous attarder sur le typage des variables.

Typage

Il existe deux façon de typer les variables

La méthode automatique, par inférence, en assignant une valeur à la variable, celle-ci prend le type de la valeur qu'on lui passe.

```
let age: number
Type 'string' is not assignable to type 'number'. ts(2322)
Peek Problem (Alt+F8)    No quick fixes available
```

```
let variable;
variable = "Geoffroy";
variable = 40;
variable = true;

let age = 40;
age = 34;
age = "Jérôme";

let prenom = "Thierry";
prenom = "Michaël";
prenom = true;
```

Typage

La méthode manuel, de manière explicite, je précise à la variable son type lorsque je la déclare.

```
let uneVariable: any;
let nbr : number;
let uneChaine: string;
let flag: boolean;
```

```
uneVariable = 42;
uneVariable = "Steve";
uneVariable = false;

nbr = 42;
nbr = "Aurélien";
nbr = true;

uneChaine = 42;
uneChaine = "Flavian";
uneChaine = false;

flag = 42;
flag = "Mélanie";
flag = true;
```

Les Arrays

Les Arrays

Le typage d'un array se fait comme pour les variables peut se faire de manière explicite

```
//Il est possibles d'attribuer à un array le type que l'on souhaite
let tab: any[] = ["Thierry", "Michaël", "Steve"];
tab = [42, 40];
tab = [true, false];
tab = ["Jérôme", 42, true];

let tabNombre: number[] = [42, 23, 31];
tabNombre = ["Geoffroy", "Mélanie"];

//le tableau étant défini comme number il ne peut recevoir autre chose
//Cependant il y a un moyen de combiner plusieurs type dans un tableau sans passer par any

let tabMulti: (number | string)[] = ["Aurélien", 29];
tabMulti = ["Flavian", 35, true] //ne passe pas puisque défini uniquement sur string et number
```

Les Arrays

Il peut également se faire par inférence

```
//ça fonctionne aussi par inférence
let tab2 = ["Constance", "Raphaël", "Aurélie"];
tab2 = [4, 1, 34]; //ça ne marche pas, le type du tableau défini par inférence est string

let tabMulti2 = ["Caroline", true];
//c'est comme si j'avais dis de manière explicite
//let tabMulti2: (string | boolean)[] = ["Caroline", true];
```

Les enums

Les Enums

Une enum est une collection valeur ayant un lien entre elles comme par exemple les jours de la semaine. Comme pour un array, on utilisera un index pour atteindre la valeur à afficher.

La différence se trouve dans la valeur de départ de l'index, en Typescript on peut préciser cette valeur, qui par défaut est à 0.

```
enum WeekDays {
    Lundi = 1,
    Mardi,
    Mercredi
}

console.log(WeekDays[1]);
console.log(WeekDays[2]);
```

Les Enums

Une enum peut être parcouru au moyen d'une itération, d'une boucle dans laquelle pour chaque valeur numérique, j'afficherai l'élément associé.

```
for (const value in Object.keys(WeekDays)){
    console.log(WeekDays[Number(value)]);
}
```

Les Enums

L'index d'une enum n'est pas forcément au format numérique, il peut être de type chaîne de caractère.

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}

console.log(Direction.Up);
console.log(Direction["Down"]);
```

Direction.Up renverra UP en console.
Direction[« Down »] renverra DOWN.

Les Tuples

Les tuples

Un tuple est similaire à un array qui ne peut contenir qu'une seule valeurs de chaques types spécifiés.

```
let me: [string, number] = ["Geoffroy", 40];
console.log(me);
```

Si je souhaite avoir par exemple deux éléments de type identique je dois le spécifier dans la déclaration du tuple

```
let me: [string, number, string] = ["Geoffroy", 40, "du texte"];
console.log(me);
```

Les objets

Les objets

Comme les variables et les arrays, le type objet peut être défini par Inférence

```
//Les objets c'est plus ou moins la même chose
//cependant attention à bien respecter les "key" mise au départ
let userData = {
    name: "Geoffroy", //ne pourra être que du string
    age: 40, //ne pourra être qu'un nombre
    isMajeur: true //ne pourra être qu'un boolean
};

userData = {
    a: "Aurélie", //pas la bonne key, il faut mettre name
    b: 34, //pas la bonne key, il faut mettre age
    c: true //pas la bonne key, il faut mettre isMajeur
}
```

Les objets

Ou de manière explicite

```
//je spécifie directement le type de chaque Key
let userData2: { name: string, age: number, isMajeur: boolean } = {
    name: "Geoffroy", //ne pourra être que du string
    age: 40, //ne pourra être qu'un nombre
    isMajeur: true //ne pourra être qu'un boolean
};
```

Méthodes

Quand je déclare une méthode, à l'instar de langage comme Java ou C#, je peux préciser le type de retour.

Une méthode qui prend void en type de retour ne renvoie rien, elle exécute juste du code, on parlera alors de procédure.

```
function helloWorld(): void {
    console.log("salut bande de dev !");
}

helloWorld();
```

Méthodes

Une méthode qui prend autre chose que void (number, string...) et le mot clé return est une fonction. Les méthodes peuvent recevoir des paramètres dont je peux préciser le type.

```
function addition(x: number, y: number): number{
    return x + y;
}

console.log(addition(5,5));
```

Attention si je n'indique pas le type du paramètre, il se fera par inférence et je pourrais me retrouver avec une chaîne de caractère au lieu d'une nombre...

Créer un type

Si je veux créer deux objets « personne » , je déclare mon objet et précise le type des key.

```
const personne: {
    name: string;
    age: number;
    isMajeur: boolean;
    hobbies: string[];
} = {
    name: "Geoffroy",
    age: 40,
    isMajeur: true,
    hobbies: ["Volley", "Jeux Videos", "Jeux de rôles"]
};
```

```
const personne2: {
    name: string;
    age: number;
    isMajeur: boolean;
    hobbies: string[];
} = {
    name: "Aurélie",
    age: 36,
    isMajeur: true,
    hobbies: ["Course à pied", "Lire", "Cinéma"]
}
```

Je peux observer une redondance d'informations... Et il existe un moyen de l'éviter

Créer un type

Il me suffit de créer moi-même un type dans lequel je précise le type des key, je pourrai ensuite créer un ou plusieurs objets du type créé.

```
type Personne = {
    name: string;
    age: number;
    isMajeur: boolean;
    hobbies: string[];
};
```

```
const personne: Personne = {
    name: "Geoffroy",
    age: 40,
    isMajeur: true,
    hobbies: ["Volley",
              "Jeux Videos",
              "Jeux de rôles"]
};

const personne2: Personne = {
    name: "Aurélie",
    age: 36,
    isMajeur: true,
    hobbies: ["Course à pied",
              "Lire",
              "Cinéma"]
};
```

Typeof

Typeof permet d'aller récupérer le type d'une variable au format chaîne de caractère. Il va permettre de vérifier que le type de la variable qu'on reçoit est celui qu'on attend.

```
let value = 15;

if(typeof value == "number"){
    console.log("value est un nombre");
}
```

Fichier config

On peut depuis la console créer un fichier tsconfig.json en utilisant la commande :

tsc –init

Ce fichier reprend toute une série d'options pour le compilateur qu'on peut ajouter.

Ou pourrait, par exemple, mettre en plus de ce qui est présent un compileOnSave qui recompilera automatiquement à chaque sauvegarde.

```
"compileOnSave":true,

"compilerOptions": {
```

Il y a toute un série d'options déjà présente

target -> version d'EcmaScript utilisée par le compilateur

module -> le module utilisé par défaut est commonjs

soEmitOnError -> à true, pas de création de fichier JS en cas d'erreur à la compialtion

strict -> à true, fait les verifications des types

noImplicitAny -> à true, bloque la possibilité des créer des variables de type any

```
"compilerOptions": {
  "target" : "es5",
  "module" : "CommonJS",
  "noEmitOnError": true,
  "strict" : true,
  "noImplicitAny" : true
}
```

Les interfaces

C’est quoi?

Pour rappel une interface doit être vue comme un contrat que nos objets doivent obligatoirement implémenter.

Ce faisant les objets respecteront un certain nombre de règle.

De prime à bord, on pourrait dire qu’une interface fonctionne comme un type que je créerais moi-même.

```
interface PersonneId {
    name: string;
}

// type PersoneId = {
//     name: string;
// };

function direBonjour(person: PersonneId){
    console.log("Bonjour mon cher ${person.name}.");
}

const person = {
    name: "Jérôme",
    age: "36"
};

direBonjour(person);
```

Key optionnel, key readonly

```ts
interface IUserProfile {
    readonly firstName: string;
    nickName: string;
    age?: number;
    password: string;
}

const user1: IUserProfile = {
    firstName: "Geoffroy",
    nickName: "Jof",
    //age: 40,
    password: "azerty"
}

console.log(user1);
user1.firstName = "Michaël";
```

Si je place devant une key de l'interface le mot clé readonly, ça signifie qu'elle est uniquement disponible en lecture, je ne peux pas la modifier.

```
(property) IUserProfile.firstName: string
Cannot assign to 'firstName' because it is a read-only property. ts(2540)
Peek Problem (Alt+F8)    No quick fixes available
```

Si je place un ? Après une key, je la rend optionnel, elle n'est pas obligatoire.

Héritage

Concept de base de l'orienté objet, ça signifie que l'enfant hérite des caractéristiques du parent, ce concept s'applique également aux interfaces.

```
interface IProfile {
    readonly firstName: string;
    nickName: string;
    age?: number;
}

interface IAdvancedProfile extends IProfile{
    hobbies: string[];
    color: string;
    isMajeur: boolean;
}

const user2: IAdvancedProfile = {
    firstName: "Geoffroy",
    nickName: "Jof",
    //age: 40
    hobbies:  ["Volley", "jeux vidéos", "jeux de rôles"],
    color: "Purple",
    isMajeur: true
}
```

Héritage

L'objet créé avec le type de l'interface enfant doit implémenter toutes les key des deux interfaces (à l'exception des key optionnel) pour qu'il puisse être instancié sans erreur.

Si une seule (ou plus manque à l'appel), le contrat n'est pas rempli et l'objet ne peut être instancié.

```
const user2: IAdvancedProfile

Property 'isMajeur' is missing in type '{ firstName: string; nickName: string; hobbies:
string[]; color: string; }' but required in type 'IAdvancedProfile'. ts(2741)

08_héritage.ts(17, 5): 'isMajeur' is declared here.

Peek Problem (Alt+F8)    No quick fixes available
```

Les fonctions

Petit rappel, une fonction est définie par sa signature, c'est à dire son nom, son type et ses paramètres. Si je veux placer en placer une dans une interface, je dois y précer les paramètres et le type de retour qu'elle aura.

```
interface IBonjour {
    (name: string, age?:number) : void;
}

const bonjour: IBonjour = (name, age) => {
    console.log(`Bonjour petit dev ${name}, tu as ${age} ans.`);
};

bonjour("Jérémy", 30);
```

Les fonctions fléchées

Ce sont des fonctions qui possèdent une syntaxe courte qui les rend rapide à écrire, elles utilisent le signe => ressemblant à une flèche.

Cette fonction

```
const ami: IBonjour = name => {
    return name;
}
```

Va devenir

```
const ami: IBonjour = name => name;
```

Les Arrays associatifs

C'est un tableau dans lequel on associe à un ensemble de clés un ensemble de valeurs.

Chaque clés ne peut recevoir qu'une seule valeur.

Pour déclarer un array associatif dans une interface,

elle doit hériter de Array <...> afin d'avoir accès aux

fonctions des array (pop, push,..),

ensuite je peux lui donner le type d'index que

je vais utiliser et lui préciser de quel type sera l'array

```
interface IRepertory extends Array<number> {
    [index: number]: number;
}

const days: IRepertory= [1,2,3,4,5];
console.log(days);

days.push(6);
console.log(days);
```

Propriétés illimitées des objets

En reprenant le principe de l'array associatif, si je lui donne un index de type string et que je donne le type any à l'array, chaque nouvelles propriétés (key) que j'ajouterai pourra être du type que je souhaite.

On peut voir aussi que pour placer une fonction nommée dans une interface il suffit d'y placer sa signature complète

```
interface IObject {
    [index: string]: any;
    sayHello: {(name: string): void};
}

const myObject: IObject = {
    title: "Dominique",
    color: "black",
    sayHello(name: string): void {
        console.log(`Hello ${name} !`);
    }
};

console.log(myObject.title);
myObject.sayHello("David");
```

Les classes

Qu’est ce qu’une classe?

C’est une structure qui permet de définir un objet, un modèle à partir du quel je peux créer un objet.

Elle a pour but de définir les données et les différentes fonctionnalités que chaque instanciation d’un objet qui lui est lié contiendra.

Une classe possède des attributs tel que nom, prénom... Ainsi que le type de chacun d'entre eux. Je dois lui donner un constructeur, c'est une méthode particulière, exécutée à la l'instanciation d'un objet. Elle permet de donner à se propriété les paramètres que je lui passe à la création.

```
class UserProfile {
    firstName: string;
    nickName: string;
    age: number;

    constructor(firstName: string, nickName: string, age: number){
        this.firstName = firstName;
        this.nickName = nickName;
        this.age =age;
    }
}

const user3 = new UserProfile("Geoffroy", "jof", 40);
console.log(user3.firstName);
```

Un classe peut aussi comprendre une ou plusieurs méthodes qui lui sont propres.

Pour les appelé il me suffit de faire:

nomDeMonObjet.laMéthode

```
class UserProfile {
    firstName: string;
    nickName: string;
    age: number;

    constructor(firstName: string, nickName: string, age: number){
        this.firstName = firstName;
        this.nickName = nickName;
        this.age =age;
    }

    getInfo(){
        console.log(this.firstName);
        console.log(this.nickName);
        console.log(this.age);
    }
}

const user3 = new UserProfile("Geoffroy", "jof", 40);
console.log(user3.firstName);
user3.getInfo;
```

Héritage des classes

Comme nous l'avons vu avec les interfaces, pour qu'une classe hérite d'une autre, je vais utiliser le mon clé extends.

Les petites choses qui changent se trouve dans le constructeur, j'en ai un dans chaque classes et je dois amener dans le les propriété du constructeur de la première classe dans celui du second.

Mais ce sera pas suffisant, je vais aussi devoir utilisé la fonction super(propriétés de la classe parent) afin de fusionner les propriétés des deux classes.

```
class UserProfile5{
    name: string;
    age: number;

    constructor(name: string, age: number){
        this.name = name;
        this.age = age;
    }
}

class AdvancedUserProfile5 extends UserProfile5 {
    hobbies: string[];
    color: string;

    constructor(name: string, age: number, hobbies: string[], color: string){
        super(name, age);
        this.hobbies = hobbies;
        this.color = color;
    }
}

const user6 = new AdvancedUserProfile5("Geoffroy", 40, ["volley, jeux vidéos"], "purple");
```

En poussant un peut plus loin il y a une façon plus moderne, plus rapide décrire les classes en TypeScript. Cependant, Attention, cette façon de faire rend la surcharge des constructeur impossible.
Je déclare ma classe et dedans , je ne met pas de propriétés, juste un constructeur qui prendra les paramètres précédés du mot clé public, ça aura pour effet de les rendre accessible depuis l'extérieur de la classe.

```
class userProfile7 {
    constructor(public name: string, public age: number){}
}
```

Dans ma classe enfant, je n'ai pas besoin de répéter le mot clé public sur les propriétés de son parent.

```
class AdvencedUserProfile7 extends userProfile7 {
    constructor(name: string, age: number, public hobbies: string[], public color:string){
        super(name,age);
    }
}

const user7 = new AdvancedUserProfile5("Geoffroy", 40, ["volley, jeux vidéos"], "purple");
console.log(user7.name, user7.color);
```

Accessibilités

Par défaut, les propriétés que je crée dans une classe sont public, ce qui l'a rend accessible en dehors de la classe.

A contrario, une propriété privée n'est accessible qu'au sein de la classe où elle est déclarée.

Par convention d'écriture une variable privée commence toujours par un underscore

```
private _maVariablePrivee : boolean = true;
```

Pour y accéder je devrai mettre en places des méthodes get (récupérer la valeur) et set (modifier la valeur) .

```
class UserProfile {
    firstName: string;
    nickName: string;
    age: number;
    private _password : string = "azerty";

    constructor(firstName: string, nickName: string, age: number){
        this.firstName = firstName;
        this.nickName = nickName;
        this.age = age;
    }

    get password(): string {
        return this._password;
    }

    set password(newPsw: string){
        this._password = newPsw;
    }
}
```

```
const user3 = new UserProfile("Geoffroy", "jof", 40);
console.log(user3.password);
user3.password="nouveau";
console.log(user3.password);
```

Il existe aussi un mot clé d'accessibilité appelé protected, qui va dire l'accès est restreint à la classe où l'élément est déclaré et à ses types dérivés, ses enfants.

```
class UserProfile2 {
    constructor(public name: string, public age: number, protected _firstName: string ="Michel"){}
}

class AdvancedUserProfile2 extends UserProfile2 {
    constructor(name:string, age:number, public hobbies: string[], color:string){
        super(name,age);
    }
    get firstName(){
        return this._firstName;
    }
}
const user4 = new AdvancedUserProfile2("Aurélie", 34, ["lire", "course à pied"], "blue");
console.log(user4.firstName, user4.hobbies);
```

Static

Le mot clé static permet de dire que l'élément qu'il précède appartient au type et pas à son instanciation, on ne peut accédé à un élément static qu'en passant par le nom du type.

```
class UserProfile3 {
    public myName: string = "Jérémy";
    public static myNewName: string ="Axel";

    static sayHello(){
        console.log("Hello");
    }

    sayWorld(){
        console.log("World");
    }
}

const uneClass = new UserProfile3();
uneClass.sayWorld();
console.log(uneClass.myName);
console.log("--------------");
UserProfile3.sayHello();
console.log(UserProfile3.myNewName);
```

Abstract

Lorsque les fonctionnalité d'une classe ne sont pas suffisante seules pour créer un objet utilisable, on va la faire précéder un mot clé abstract.

En déclarant une classe abstract, j'empêche qu'on puisse créer une instance de celle-ci.

Une classe abstract ne peut être

utilisée que dans le cadre de l'héritage.

```
abstract class UserProfile4 {
    constructor(public name: string, public age: number){}
}

class AdvancedUserProfile4 extends UserProfile4 {
    constructor(name:string, age:number, color:string){
        super(name,age);
    }
}

const user5 = new AdvancedUserProfile4("Michaël", 38, "red");
```

Interface et classe

C'est via le mot clé implements que je vais spécifier qu'une classe implément une interface.

Une fois que c'est fait, la classe passe un contrat avec l'interface et doit obligatoirement en implémenter toutes les propriétés.

```
interface UserProfile8 {
    name: string;
    age: number;
}

class AdvancedUserProfile8 implements UserProfile8{
    constructor(public name: string, public age: number, public color: string){}
}

const user8 = new AdvancedUserProfile8("Axel", 30, "pink");
console.log(user8.name, user8.color);
```

Côté serveur

Structure d’un projet

Il est maintenant temps de passer à une structure de projet plus conventionnel, c’est en général de cette manière qu’un projet sera organiser.

Un projet doit contenir un dossier dist et un dossier src.

Dans le terminal je peux lancer la commande npm init –y qui va me créer un fichier package.json .

Dans le fichier package.json, je vais modifier certains éléments pour qu'il corresponde à ce dont j'ai besoin comme le main et le script, le définir de cette façon permet de lancer le code JS côté serveur.

```
{
  "name": "monprojet",
  "version": "1.0.0",
  "description": "",
  "main": "./dist/main.js",
  ▷ Debug
  "scripts": {
    "start": "node ./dist/main.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Le dissier src recevra tout les fichier TypeScript que j'ai écris, le dossier dist recevra tous les fichier Javascript qui auront été transcompilé.

Il me faudra modifier je fichier config.json afin de définir les chemins d'accès à ces deux dossier. Pour rappel je crée ce fichier via la commande tsc –init.

Je vais modifier la ligne outDir `"outDir": "./dist",` , mes fichiers transcompilé iront maintenant dans ce dossier.

Namespace

Un Namespace permet d'organiser le code.

Je vais pouvoir, par exemple, y déclarer plusieurs fonctions précédée du mot clé export.

```
namespace MathFunctions {
    export function addition(nbr1: number, nbr2: number){
        return nbr1 + nbr2;
    }

    export function soustraction(nbr1: number, nbr2: number){
        return nbr1 - nbr2;
    }

    export function multiplication(nbr1: number, nbr2: number){
        return nbr1 * nbr2;
    }

    export function division(nbr1: number, nbr2: number){
        return nbr1 / nbr2;
    }
}
```

Pour l'importation d'un Namespace dans un autre fichier, il me suffit d'y faire référence comme ceci `/// <reference path="math.ts" />` , je lui passe le chemin du fichier TS.

Je peux ensuite appeler les fonctions définie dans le Namespace facilement.

```
console.log(MathFunctions.addition(5,6));
console.log(MathFunctions.multiplication(5,6));
console.log(MathFunctions.soustraction(4,3));
console.log(MathFunctions.division(12,4));
```

ATTENTION ceci nécessite de modifier le fichier ts.config afin de faire en sorte que tous les fichiers TS soit exporté dans un seul et même fichier JS, ce qui peut engendrer des conflit
« outfile » devient ./dist/main.js
« module » devient system
il suffit de relancer visual studio code pour ne plus avoir d'erreur après ces modifications

PROBLEME, on ne voit pas ce qu'on importe, c'est difficile à gérer sur des gros projet et ce n'est pas une fonctionnalité ES6 donc risque de compatibilité avec les navigateurs.

SOLUTION, il est préférable d'utiliser les modules

Les modules

A l'image d'un namespace j'écris mes fonctions dans un fichier TS.

```
export function addition(nbr1: number, nbr2: number){
    return nbr1 + nbr2;
}

export function soustraction(nbr1: number, nbr2: number){
    return nbr1 - nbr2;
}

export function multiplication(nbr1: number, nbr2: number){
    return nbr1 * nbr2;
}
```

Je peux soit importer tout le contenu du fichier

```
import * as MathFunctions from "./18_math";
```

Soit importer la fonction qui m'est nécessaire

```
import { addition } from "./18_math";
```

Leur utilisation reste la même

```
console.log(addition(4,5));
console.log(MathFunctions.addition(5,6));
console.log(MathFunctions.multiplication(5,6));
console.log(MathFunctions.soustraction(4,3));
```

Si je ne veux pas répéter le mot clé export, je peux déclarer toute mes fonctions puis l'export en fin de fichier et y placer les fonction que je souhaite exporter. Je peux y placer aussi l'alias que je souhaite donner à la fonction exportée.

```
function addition(nbr1: number, nbr2: number){
    return nbr1 + nbr2;
}

function soustraction(nbr1: number, nbr2: number){
    return nbr1 - nbr2;
}

function multiplication(nbr1: number, nbr2: number){
    return nbr1 * nbr2;
}

export {addition as add, soustraction, multiplication};
```

La généricité

Sur les fonctions

La généricité permet de travailler avec plusieurs type de données, ça ouvre à une plus grande réutilisabilité du code.

Jusqu'ici pour ça on utilisait le type any pour ça, cependant avec ce type on a une perte d'information sur le type renvoyé, il m'est impossible de savoir de quel type sont les paramètres passés.

```
function identity(data: any){
    return data;
}
            function identity(data: any): any
console.log(identity("Jof"));
console.log(identity(40));
console.log(identity(true));
```

La version générique utilise les chevrons et une lettre majuscule (par convention T) <T> que ce soit pour le type à retourner ou pour le(s) paramètre(s) à recevoir. Attention que le paramètre doit être du type à retourner sinon il y aura une erreur.

```
function generics<T>(data: T){
    return data;
}

console.log(generics<string>("Jof"));
console.log(generics<number>(40));
console.log(generics<boolean>(true));
```

Sur les Array

La généricité va également nous permettre de limité à un seul type les valeurs qu'un array, c'est le meilleurs moyen que nous avons d'être sur du type d'une valeur que nous allons recevoir d'un array.

```
const myArray: Array<number> = [1,5,9];
myArray.push(7);
```

Si j'essaye d'y ajouter autre chose que le type défini, j'ai une erreur

```
const myArray
myArray.push(
myArray.push("7");
```

Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)

Peek Problem (Alt+F8) No quick fixes available

Sur les classes

Bien évidemment la généricité fonctionne aussi avec les classes mais implique de modifier le fichier tsconfig.json et de désactivé le mode strict car il impose que chaque propriété de la classe soit initialisé et qu'elle possède un constructeur.

```
class Mathematique<T> {
    add: (x: T, y:T) => T;
    sub: (x: T, y:T) => T;
}

let myTotal = new Mathematique<number>()
myTotal.add = (x, y) => x + y;
myTotal.add = (x, y) => x - y;

console.log(myTotal.add(2,3));
console.log(myTotal.sub(3,2));
```

Les contraintes et le multi-type

Si j'impose à mon paramètre générique un type, il ne pourra être d'aucun autre.
Il est également bon de savoir que je peux avoir plusieurs type générique

```
class Mathematique<T extends number, U extends number> {
    add: (x: T, y:U) => T;
    sub: (x: T, y:U) => T;
}

let myTotal = new Mathematique<number, number>();
myTotal.add = (x, y) => x + y;
myTotal.add = (x, y) => x - y;

console.log(myTotal.add(2,3));
console.log(myTotal.sub(3,2));
```

Les décorateurs

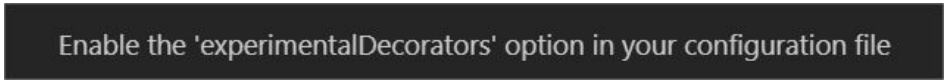
C’est quoi?

Un décorateur est un outil qui permet d’annoter ou de modifier une classe ou une propriété de cette classe.

Comme ça n’existe pas en JS, c’est encore au stade expérimental en TS, de ce fait la première fois qu’on en met un en place dans un projet un message d’erreur s’affiche

```
Experimental support for decorators is a feature that is subject to change in a future
release. Set the 'experimentalDecorators' option in your 'tsconfig' or 'jsconfig' to remove
this warning. ts(1219)
Peek Problem (Alt+F8)    Quick Fix... (Ctrl+;)
```

Il suffit de cliquer sur quick fix et ensuite sur



Ce qui ajoutera dans le fichier tsconfig.json la ligne suivante

```
"experimentalDecorators": true,
```

Il ne reste plus qu'a sauvegarder le fichier tsconfig.json et je peux utiliser les décorateurs dans mon projet.

Un décorateur se présente sous la forme d'une fonction que j'attache à la classe où je veux l'utiliser via un **@** , dans ce cas ci je cible le contenu d'un constructeur

```
function logConstruct(constructorFunction: Function){
    console.log(constructorFunction);
}

@logConstruct
class Identity {
    constructor(){
        console.log("Geoffroy");
    }
}
```

Avec un décorateur je peux également cibler les propriété d' une classe.

Cependant il y a un petit bug dans TypeScript, pour pouvoir accéder à ma fonction print, je dois caster mon objet en type any.

```
function printProperty(properties: Function){
    properties.prototype.print = function() {
        console.log(this);
    };
}

@printProperty
class Games {
    rts = "StarCraft";
    rpg = "Final Fantasy XIV";
}

const game = new Games();
(<any>game).print()
```

Annexes

Surcharge du constructeur

Un constructeur est une méthode, et comme toutes méthodes il est possible de la surcharger, de lui donner plusieurs façon de fonctionner.

Si je veux, par exemple, pouvoir créer un objet vide dont je définirai les paramètre plus tard j'ai besoin d'un constructeur vide.

Je vais donc définir une méthode constructor() sans paramètre, à la suite de laquelle je viens placer une méthode constructor(param1 : type, ...) et ensuite une troisième méthode où je rendrai les paramètre optionnel constructor(param1? : type).

```
class UserProfileTest {
    firstName: string;
    nickName: string;
    age: number;

    constructor()
    constructor(firstName: string, nickName: string, age: number)
    constructor(firstName?: string, nickName?: string, age?: number){
        this.firstName = firstName || "";
        this.nickName = nickName || "";
        this.age = age || 0;
    }

    getInfo(){
        console.log(this.firstName);
        console.log(this.nickName);
        console.log(this.age);
    }
}
```

Je n'ai ensuite plus qu'a placer un coalesce dans l'attribution des valeurs au sein de mon constructeur pour les initialiser avec des valeurs par défaut.

```
const test = new UserProfileTest();
test.firstName = "toto";
test.nickName = "tutu";
test.age = 25;
test.getInfo();

const test2 = new UserProfileTest("tintin", "milou", 59);
test2.getInfo();
```

Merci pour votre attention.

CogniTIC

Building Together Better IT Solutions