TypeScript

PLAN

- Introduction
- Fichier de configuration
- Variable
- Constante
- Fonction
- Concept de décomposition (spread)
- Module
- Classe
- Héritage
- Classes abstraites
- Interfaces

INTRODUCTION

ECMAScript

ensemble de normes sur les langages de programmation de type script (JavaScript, ActionScript...)
standardisée par Ecma International (European Computer Manufacturers Association)
depuis 1994

Quelques versions

ECMAScript version 5 (ES5) ou ES 2009
ECMAScript version 6 (ES6) ou ES 2015 (compatible avec les navigateurs modernes)

TypeScript

- langage de programmation
  - procédural et orienté-objet
  - supportant le typage statique, dynamique et générique
- open-source créé par Anders Hejlsberg (inventeur de C#) de **MicroSoft**
- **Utilisé par Angular (Google)**

INTRODUCTION

Le navigateur ne comprend pas **TypeScript**

Il faut le transcompiler (ou transpiler) en **JavaScript**

Comment va t-on procéder dans ce cours?

INTRODUCTION

- De quoi on a besoin?
  - **Node.js** pour exécuter la commande node
  - **TypeScript** pour exécuter la commande tsc

- Pour **Node.js**, il faut
  - aller sur https://nodejs.org/en/
  - choisir la dernière version, télécharger et installer

- Pour **TypeScript**, il faut
  - ouvrir une console (invite de commandes)
  - lancer la commande npm install -g typescript
  - vérifier la version avec la commande tsc –v

- Quel IDE pour TypeScript :**Microsoft** recommande **Visual Studio Code**

Exercice

1- Créer un répertoire "exercices-ts ",

2- Dans « exercices-ts », créer deux sous répertoires nommés « pubic » et « src »:

- le premier pour mettre les fichiers html, css et js (javascript) générés,
- le deuxième pour placer les fichiers ts (Typsescript)

3- Dans src, créez un fichier main.ts .

Ajouter le contenu suivant :`console.log("Hello world");`

4- Pour compiler, lancez la commande *tsc main.ts*

5- pour exécuter, lancer la commande *node main.js*

Fichier de configuration

Fichier de configuration : tsconfig.json

- Pouvant être généré avec la commande **tsc --init**
- Placé à la racine d'un projet **TypeScript**
- Consulté par le compilateur à chaque exécution de la commande **tsc**
- Si le fichier n'existe pas, des valeurs par défaut seront utilisées

Dans tsconfig.json, modifiez les propriétés suivantes

- `"outDir": "./public,`
- `"rootDir": "./src"`

Variable - Déclaration

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x :number
```

Initialiser une variable

```
x = 2;
```

Déclarer et initialiser une variable

```
var x: number = 2;
```

Cependant, ceci génère une erreur car une varaible ne change pas de type

```
x = "bonjour";
```


8

Variable - Déclaration

Quels types pour les variables en **TypeScript** ?

- **number** pour les nombres (entiers, réels, binaires, décimaux, hexadécimaux...)
- **string** pour les chaînes de caractère
- **boolean** pour les booléens
- **array** pour les tableaux non-statiques (taille variable)
- **tuple** pour les tableaux statiques (taille et type fixes)
- **object** pour les objets
- **any** pour les variables pouvant changer de type dans le programme
- **enum** pour les énumérations (tableau de constantes)

Les types undefined et null du **JavaScript** sont aussi disponibles.

Variable - Déclaration

Pour les chaînes de caractères, on peut faire

```
var str1: string = "ben saleh";
var str2: string = 'meriem';
```

On peut aussi utiliser template strings

```
var str3: string = `Bonjour ${ str2 } ${ str1 }
Que pensez-vous de TypeScript ? `;
console.log(str3);

// affiche Bonjour meriem ben saleh
//Que pensez-vous de TypeScript ?
```

L'équivalent de faire

```
var str3: string = "Bonjour " + str2 + " " + str1 +
"\nQue pensez-vous de TypeScript ?";
```

Variable - Déclaration

Une première déclaration pour les tableaux

```
var list: number[] = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

Une deuxième déclaration

```
var list: Array<number> = new Array(1, 2, 3);
console.log(list);
// affiche [ 1, 2, 3 ]
```

Ou encore plus simple

```
var list: Array<number> = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

Variable - Déclaration

Pour les **tuples**, on initialise toutes les valeurs à la déclaration

```
var t: [number, string, string] = [100,"ben salah",'meriem'];
```

Pour accéder à un élément d'un tuple en lecture ou en écriture console.log(t[0]);

```
// affiche 100

t[2] = "travolta";
console.log(t);
// affiche [ 100, 'ben salah', ‘travolta' ]
```

Cependant, ceci génère une erreur

```
t = [100, 200, ‘meriem'];
```

Variable - Déclaration

Avec TypeScript 3.0, on peut rendre certains éléments de tuple optionnels

```
var t: [number, string?, string?] = [100];
console.log(t);
// affiche [ 100 ]

console.log(t[1]);
// affiche undefined
```

Pour ajouter un élément

```
t[1] = 'meriem';
```

Ceci génère une erreur

```
t[2] = 100;
```

Et cette instruction aussi car on dépasse la taille du tuple `t[3] = 100;`

Variable - Déclaration

Exemple avec any

```
var x: any;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```

Une variable de type any peut être affectée à n'importe quel autre type de variable

```
var x: any;
x = "bonjour";  x = 5;
var y: number = x;
```

Variable - Déclaration

Le type unknown (TypeScript 3.0) fonctionne comme any mais ne peut être affecté qu'à une variable de type unknown ou any

```
var x: unknown;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```

Ceci génère donc une erreur

```
var x: unknown;  x = "bonjour";
x = 5;
var y: number = x;
```

Variable - Déclaration

Déclarons une énumération (dans file.ts)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

L'indice du premier élément est 0

```
console.log(mois.AVRIL)
// affiche 3
```

Pour modifier l'indice du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI, JUIN,
JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

En affichant maintenant, le résultat est

```
console.log(mois.AVRIL)
// affiche 4
```


16

Variable - Déclaration

Pour déclarer un objet

```
var obj: {  nom: string;
numero: number;
};
```

On peut initialiser les attributs de cet objet

```
obj = {
        nom: 'Mohamed',
        numero: 100
};

console.log(obj);
// affiche { nom: 'Mohamed', numero: 100 }

console.log(typeof obj);
// affiche object
```

Variable - Déclaration

On peut modifier les valeurs d'un objet ainsi

```
obj.nom = 'mohamed';
 obj['numero'] = 200;
 console.log(obj);
// affiche { nom: 'mohamed', numero: 200 }
```

Ceci est une erreur : `obj.nom = 125;`

Variable - Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable.
Déclarer une variable acceptant plusieurs types de valeur:

```
var y: number | boolean | string;
```

affecter des valeurs de type différent

```
y = 2;
y = "bonjour";
y = false;
```

Ceci génère une erreur

```
y = [2, 5];
```

Variable- variable locale

Le mot-clé let

permet de donner une visibilité locale à une variable déclarée dans un bloc.

Ceci génère une erreur car la variable x a une visibilité locale limitée au bloc if

```
if (5 > 2)
{
let x = 1;
}
console.log(x);

// affiche ReferenceError: x is not defined
```

Variable - cast

Premier exemple

```
let str: any = "bonjour";
let longueur: number = (<string>str).length;
console.log(longueur);
// affiche 7
```

Deuxième exemple

```
let str: any = "bonjour";
let longueur: number = (str as string).length;
console.log(longueur);
// affiche 7
```


21

Variable - Conversion

Pour convertir une chaîne de caractère en nombre

```
let x : string = "2";
let y: string = "3.5";
let a: number = Number(x);  let b: number = Number(y);
console.log(a);
// affiche 2
console.log(b);
// affiche 3.5
```

Il existe une fonction de conversion pour chaque type

22

Les constantes

se déclare avec le mot-clé const ,
permet à une variable de ne pas changer de valeur

Ceci génère une erreur car une constante ne peut changer de valeur

```
const X: any = 5;
X = "bonjour";
// affiche TypeError: Assignment to constant variable.
```

Les constantes

Avec TypeScript 3.4, on peut définir une constante avec une assertion sans préciser le type

```
let X = "bonjour" as const;
console.log(X);
// affiche bonjour

console.log(typeof X);
// affiche string

let Y: string = "bonjour";
console.log(X == y);
//affiche true
```

Ceci génère une erreur car une constante ne peut changer de valeur:

```
X = "hello";
```

Les constantes

Avec TypeScript 3.4, on peut aussi définir une constante ainsi

```
let X = <const>"bonjour";
console.log(X);
// affiche bonjour

console.log(typeof X);
// affiche string

let y: string = "bonjour";
console.log(X == y);
//affiche true
```

Ceci génère une erreur car une constante ne peut changer de valeur

```
X = "hello";
```

Fonction – déclaration et appel

Déclarer une fonction:

function nomFonction([les paramètres]){ les instructions de la fonction}

Exemple:

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

Appeler une fonction

```
let resultat: number = somme (1, 3);
 console.log(resultat);
// affiche 4
```

Fonction – déclaration et appel

Le code suivant génère une erreur

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

Celui-ci aussi

```
let resultat: number = somme ("1", 3);
```

Et même celui-ci

```
let resultat: string = somme(1, 3);
```

Une fonction qui ne retourne rien a le type ***void*** function

```
direBonjour(): void
{  console.log("bonjour");}
```

Une fonction qui n'atteint jamais sa fin a le type never

```
function boucleInfinie(): never {
while (true){
}
```

}

Fonction – Paramètres par defaut

Il est possible d'attribuer une valeur par défaut aux paramètres d'une fonction

```
function division(x: number, y: number = 1) : number
{
return x / y;
}
console.log(division(10));
// affiche 10
console.log(division(10, 2));
// affiche 5
```


28

Fonction – Paramètres optionnels

Il est possible de rendre certains paramètres d'une fonction optionnels

```
function division(x: number, y?: number): number {
if(y)
return x / y;
return x;
}
console.log(division(10));
// affiche 10
console.log(division(10, 2));
// affiche 5
```


29

Fonction – paramètres restants

Il est possible de définir une fonction prenant un nombre indéfini de paramètres

```
function somme(x: number, ...tab: number[]): number {
for (let elt of tab)
x += elt;
return x;
}
console.log(somme(10));
// affiche 10

console.log(somme(10, 5));
// affiche 15

console.log(somme(10, 1, 6));
// affiche 17
```

Fonction – paramètres à plusieurs types autorisés

Il est possible d'autoriser plusieurs types pour un paramètres

```
function stringOrNumber(param1: string | number,param2: number): number
{
if (typeof param1 == "string")  return param1.length +
param2;
 return param1 + param2;
}

console.log(stringOrNumber("bonjour", 3));
// affiche 10

console.log(stringOrNumber(5, 3));
// affiche 8
```

Fonction – paramètres en lecture seule

Le mot-clé ReadonlyArray (TypeScript 3.4) indique qu'un paramètre de type tableau est en lecture seule (non-modifiable)

```
function incrementAll(tab: ReadonlyArray<number>): void {
for (let i = 0; i < tab.length; i++){
// la ligne suivante génère une erreur
tab[i]++;
}
}
```

On peut aussi utiliser le mot-clé readonly qui s'applique sur les tableaux et les tuples

```
function incrementAll(tab: readonly number[]): void {
for (let i = 0; i < tab.length; i++){
// la ligne suivante génère une erreur
tab[i]++;
}
}
```

Fonction – Fonctions fléchées (arrow function)

Il est possible de déclarer une fonction en utilisant les expressions fléchées

```
let nomFonction = ([les paramètres]): typeValeurRetour => {
les instructions de la fonction
}
```

Exemple

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

Ou en plus simple

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

Appeler une fonction fléchée

```
let resultat: number = somme (1, 3);
```

Fonction – Fonctions fléchées (arrow function)

Cas d'une fonction fléchée à un seul paramètre

```
let carre = (a: number): number => a * a;
console.log(carre(2));
// affiche 4
```

Sans typage, la fonction peut être écrite ainsi

```
let carre = a => a * a;
console.log(carre(2));
// affiche 4
```

Déclaration d'une fonction fléchée sans paramètre

```
let sayHello = (): void => console.log('Hello');
sayHello();
// affiche Hello
```

Remarque

Le mot-clé **this** est inutilisable dans les fonctions fléchées

Fonction – Fonctions fléchées (arrow function)

Les fonctions fléchées sont utilisées pour réaliser les opérations suivant sur les tableaux

- **forEach**() : pour parcourir un tableau
- **map**() : pour appliquer une fonction sur les éléments d'un tableau
- **filter**() : pour filtrer les éléments d'un tableau selon un critère défini sous forme d'une fonction anonyme ou fléchée
- **reduce**() : pour réduire tous les éléments d'un tableau en un seul selon une règle définie dans une fonction anonyme ou fléchée
- **…**

Fonction – Fonctions fléchées (arrow function)

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];

tab.forEach(elt => console.log(elt));

//affiche 235
```

Dans `forEach`, on peut aussi appeler une fonction `afficher`

```
tab.forEach(elt => afficher(elt));

function afficher(value) { console.log(value);}

//affiche 235
```

On peut simplifier l'écriture précédente en utilisant les callback

```
tab.forEach(afficher);

function afficher(value) {  console.log(value);}

//affiche 235
```

Fonction – Fonctions fléchées (arrow function)

On peut utiliser *map* pour effecteur un traitement sur chaque élément du tableau puis *forEach* pour afficher le nouveau tableau

```
tab.map(elt => elt + 3)
.forEach(elt => console.log(elt));
// affiche 5 6 8
```

On peut aussi utiliser *filter* pour filtrer des éléments

```
tab.map(elt => elt + 3)
.filter(elt => elt > 5)
.forEach(elt => console.log(elt));
// affiche 6 8
```

Remarque: Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer

Fonction – Fonctions fléchées (arrow function)

Exemple avec *reduce* : permet de réduire les éléments d'un tableau en une seule valeur

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
.filter(elt => elt > 5)
.reduce((sum, elt) => sum + elt);
console.log(somme);
// affiche 14
```

Si on a plusieurs instructions, on doit ajouter les accolades

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
.filter(elt => elt > 5)
.reduce((sum, elt) => {
return sum + elt;
}
);
console.log(somme);
// affiche 14
```

- Le premier paramètre de **reduce** correspond au résultat de l'itération précédente
- Le deuxième correspond à l'élément du tableau de l'itération courante
- Le premier paramètre est initialisé par la valeur du premier élément du tableau
- On peut changer la valeur initiale du premier paramètre en l'ajoutant à la fin de la méthode

Fonction – Fonctions fléchées (arrow function)

Dans cet exemple, on initialise le premier paramètre de reduce par la valeur 0

```
var somme = tab.map(elt => elt + 3)
.filter(elt => elt > 5)
.reduce((sum, elt) => sum + elt, 0);
console.log(somme);
// affiche 14
```

Fonctions fléchées : pourquoi?

- Simplicité d'écriture du code
- meilleure lisibilité

Concept de décomposition (spread)

Considérons la fonction somme suivante

```
function somme(a?: number, b?: number, c?: number): number {
return a + b + c;
}
```

Pour appeler la fonction somme, il faut lui passer trois paramètres number

```
console.log(somme (1, 3, 5));
// affiche 9
```

Et si les valeurs se trouvent dans un tableau, on peut utiliser la décomposition

```
let t: Array<number> = [1, 3, 5];
console.log(somme(...t));
```

Concept de décomposition (spread)

Considérons les deux objets suivants

```
let obj = { nom: 'Ben Salem', prenom: 'Anis'};
  let obj2 = obj;
```

Modifier l'un ➔ modifier l'autre

```
obj2.nom = 'Trabelsi';  console.log(obj);
// affiche { nom: 'Trabelsi', prenom: 'Anis' }
 console.log(obj2);
// affiche { nom: 'Trabelsi', prenom: 'Anis' }
```

Pour que les deux objets soient indépendants, on peut utiliser la décomposition pour faire le clonage

```
let obj = { nom: 'Ben Salem', prenom: 'Anis'};
let obj2 = { ...obj };
obj2.nom = 'Trabelsi';
 console.log(obj);
// affiche { nom: 'Ben Salem', prenom: 'Anis' }
 console.log(obj2);
// affiche { nom: 'Trabelsi', prenom: 'Anis' }
```

Module

Module

- Introduit dans ES6
- Un fichier pouvant contenir des variables ; fonctions, classes, interfaces...

Propriétés

- Il est possible d'utiliser des éléments définis dans un autre fichier : une variable, une fonction, une classe, une interface...
- Pour cela, il faut l'importer là où on a besoin de l'utiliser
- Pour importer un élément, il faut l'exporter dans le fichier source
- En transpilant le fichier contenant les import, les fichiers contenant les éléments importés seront aussi transpilés.

étant donné le fichier fonctions.ts dont le contenu est:

```
  function  somme(a: number = 0, b: number = 0) {
      return a + b;}
function produit(a: number = 0, b: number = 1) {
      return a * b;
}
```


42

Module

Pour exporter les deux fonctions somme et produit de fonction.ts

```
export function somme(a: number = 0, b: number = 0)
{  return a + b;}
export function produit(a: number = 0, b: number = 1)
{return a * b;}
```

Ou aussi

```
function somme(a:number = 0, b:number = 0) {
return a + b;}
function produit(a: number = 0, b: number = 1) {
return a * b;}
export { somme, produit };
```


43

Module

Pour importer et utiliser une fonction
```
import { somme } from './fonctions';
console.log(somme(2, 5));
// affiche 7
```
Pour importer plusieurs éléments
```
import { somme, produit } from './fonctions';
console.log(somme(2, 5));
// affiche 7
console.log(produit(2, 5));
// affiche 10
```

On peut aussi utiliser des alias
```
import { somme as s, produit as p } from './fonctions';
console.log(s(2, 5));
// affiche 7
console.log(p(2, 5));
// affiche 10
```
Ou aussi
```
import * as f from './fonctions';
console.log(f.somme(2, 5));
// affiche 7
console.log(f.produit(2, 5));

// affiche 10
```

Exercice 1,2 et 3

Classes

Qu'est ce qu'une classe en POO?

- ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Instance?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe

composé une classe?

- **Attribut** : [visibilité] + nom + type
- **Méthode** : [visibilité] + nom + arguments + valeur de retour

Classes

Considérons la classe Personne définie dans personne.ts

```
export class Personne {
num: number;  nom: string;  prenom: string;
}
```

Avant de compiler, vérifiez dans tsconfig.json les propriétés suivantes

- "target": "es6 "

En TypeScript

- Toute classe a un constructeur par défaut sans paramètre.
- Par défaut, la visibilité des attributs est public.

Classes

Hypothèse

Si on voulait créer un objet de la classe Personne avec les valeurs 1, Trabelsi et Anis

Etape 1 : Commençons par importer la classe Personne dans file.ts

```
import { Personne } from './personne';
```

Etape 2 : déclarons un objet (objet non créé) `let personne: Personne;`

Etape 3 : créons l'objet (instanciation) de type Personne (objet créé)

```
personne = new Personne();
```

On peut faire déclaration + instanciation

```
let personne: Personne = new Personne();
```

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "Trabelsi";
personne.prenom = "Anis";
```

Pour être sûr que les valeurs ont bien été affectées aux attributs, on affiche

```
console.log(personne);
// affiche Personne { num: 1, nom: 'Trabelsi', prenom: 'Anis' }
```

Classes - setter

Hypothèse

Supposant que l'on n'accepte pas de valeur négative pour l'attribut **num** de la classe **Personne**

Démarche

- Bloquer l'accès directe aux attributs (mettre la visibilité à **private**)

- Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les **setter**)

Convention

- Mettre la visibilité **private** ou **protected** pour tous les attributs

- Mettre la visibilité **public** pour toutes les méthodes

Classes - setter

Mettons la visibilité private pour tous les attributs de la classe Personne

```
export class Personne {
private num: number;
private nom: string;
private prenom: string;
}
```

Dans le fichier file.ts, les trois lignes suivantes sont soulignées en rouge

```
personne.num = 1;
personne.nom = "Trabelsi";
 personne.prenom = "Anis";
```

Explication

- Les attributs sont privés, donc aucun accès direct n'est autorisé

Solution : les setters

Des méthodes qui contrôlent l'affectation de valeurs aux attributs

Classes - setter

Conventions TypeScript

- Le setter est une méthode déclarée avec le mot-clé set
- Il porte le nom de l'attribut
- On l'utilise comme un attribut
- Pour éviter l'ambiguïté, on ajoute un underscore pour l'attribut

Nouveau contenu de la classe Personne après ajout des setters

```
export class Personne {
private _num: number;  private _nom: string;
private _prenom: string;
public set num(_num : number) {
this._num = (_num >= 0 ? _num : 0);
}
public set nom(_nom: string) {  this._nom = _nom;}
public set prenom(_prenom: string) {
this._prenom = _prenom;}
}
```

Classes - setter

Pour tester, rien à changer dans file.ts

```
import { Personne } from './personne';
let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "Trabelsi";
personne.prenom = "Anis";
console.log(personne);
```

Le résultat est :

Personne { _num: 1, _nom: Trabelsi', _prenom: 'Anis' }

Testons avec une valeur négative pour l'attribut numero

```
import { Personne } from './personne';
let personne: Personne = new Personne();
personne.num = -1;  personne.nom = "Trabelsi";
personne.prenom = "Anis";  console.log(personne);
```

Le résultat est :

Personne { _num: 0, _nom: 'Trabelsi', _prenom: 'Anis' }

Classes -getter

Question

Comment récupérer les attributs (privés) de la classe Personne?

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les getter)

Conventions TypeScript

- Le getter est une méthode déclarée avec le mot-clé get
- Il porte le nom de l'attribut
- On l'utilise comme un attribut

Ajoutons les getters dans la classe Personne

```
public get num() : number {
return this._num;
}
public get nom(): string {
return this._nom;
}
public get prenom(): string {
return this._prenom;
}
```

Pour tester

```
import { Personne } from './personne';
let personne: Personne = new Personne();
 personne.num = 1;
 personne.nom = "Trabelsi";
personne.prenom = "Anis";
 console.log(personne.num);
// affiche 1
console.log(personne.nom);
// affiche Trabelsi
console.log(personne.prenom);
// affiche Anis
```

Classes - constructeur

Contructeur

- Par défaut, toute classe en TypeScript a un constructeur par défaut sans paramètre
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe

Les constructeurs avec TypeScript

- On le déclare avec le mot-clé **constructor**
- Il peut contenir la visibilité des attributs si on veut simplifier la déclaration

Classes - constructeur

Le constructeur de la classe Personne prenant trois paramètres

```
public constructor(_num: number, _nom: string, _prenom: string)
{
 this._num = _num;
 this._nom = _nom;
 this._prenom = _prenom;
}
```

Pour préserver la cohérence, il faut que le constructeur contrôle la valeur de l'attribut num

```
public constructor(_num: number, _nom: string, _prenom: string) {
this._num = (_num >= 0 ? _num : 0);
this._nom = _nom;
this._prenom = _prenom;
}
```

On peut aussi appelé le setter dans le constructeur

```
public constructor(_num: number, _nom: string, _prenom: string) {
this.num = _num;  this._nom = _nom;
this._prenom = _prenom;
}
```

Classes - constructeur

Dans file.ts, la ligne suivante est soulignée en rouge

```
let personne: Personne = new Personne();
```

Explication

Le constructeur par défaut à été écrasé (il n'existe plus)

Comment faire?

- TypeScript n'autorise pas la présence de plusieurs constructeurs (la surcharge)
- On peut utiliser soit les valeurs par défaut, soit les paramètres optionnels

Le nouveau constructeur avec les paramètres optionnels

```
public constructor(_num?: number, _nom?: string, _prenom?: string) {
if(_num)
    this.num = _num;
if (_nom)
    this._nom = _nom;
if(_prenom)
    this._prenom = _prenom;
}
```

Classes - constructeur

Pour tester

```
import { Personne } from './personne';
let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "Trabelsi";
personne.prenom = "Anis";
console.log(personne);
let personne2: Personne = new Personne(2, 'Trabelsi', 'ali');
console.log(personne2);
```

En exécutant, le résultat est :

Personne { _num: 0, _nom: ’Trabelsi’, _prenom: ’Anis’ }
Personne { _num: 2, _nom: 'Trabelsi', _prenom: 'ali' }

TypeScript nous offre la possibilité de fusionner la déclaration des attributs et le constructeur

```
public constructor(private _num?: number,
 private _nom?: string, private _prenom?:  string) {
}
```

En exécutant, le résultat est le même

Personne { _num: 0, _nom: ’Trabelsi’, _prenom: ’Anis’ }
Personne { _num: 2, _nom: 'Trabelsi', _prenom: 'ali' }

Classes – attributs et méthodes statiques

Récapitulatif

Les instances d'une même classe ont toutes les mêmes attributs mais pas les mêmes valeurs

Hypothèse

Et si nous voulions qu'un attribut ait une valeur partagée par toutes les instances (par exemple, le nombre d'objets instanciés de la classe Personne)

Solution : attribut statique ou attribut de classe

Un attribut dont la valeur est partagée par toutes les instances de la classe.

Exemple

- Si on voulait créer un attribut contenant le nombre d'objets créés à partir de la classe **Personne**
- Notre attribut doit être déclaré **static**, sinon chaque objet pourrait avoir sa propre valeur pour cet attribut

Ajoutons un attribut statique nbrPersonnes **à la liste d'attributs de la classe** Personne

```
private static _nbrPersonnes: number = 0;
```

Classes – attributs et méthodes statiques

Incrémentons notre compteur de personnes dans les constructeurs

```
public constructor(private _num?: number,
private _nom?: string,  private _prenom?: string) {
Personne._nbrPersonnes++;
}
```

Créons un getter pour l'attribut static nbrPersonnes

```
public static get nbrPersonnes() {
  return Personne._nbrPersonnes;
}
```

Testons cela dans file.ts

```
import { Personne } from './personne';
console.log(Personne.nbrPersonnes);
// affiche 0
let personne: Personne = new Personne();  personne.num = -1;
personne.nom = "Trabelsi"; personne.prenom = "Anis";
console.log(Personne.nbrPersonnes);
// affiche 1
let personne2: Personne = new Personne(2, 'Trabelsi', 'ali');
console.log(Personne.nbrPersonnes);
// affiche 2
```

héritage

L'héritage, quand?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une Classe1 est (une sorte de) Classe2

Forme générale

```
class ClasseFille extends ClasseMère
{
// code
};
```

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire
- Un étudiant a aussi un numéro, un nom, un prénom et un niveau
- Sémantiquement, enseignant et étudiant sont une sorte de personne
- En plus, les deux partagent plusieurs attributs tels que numéro, nom et prénom
- Donc, on peut utiliser la classe Personne puisqu'elle contient tous les attributs numéro, nom et prénom
- Les classes Etudiant et Enseignant hériteront donc (extends) de la classe Personne

héritage

Particularité du langage TypeScript

- Une classe ne peut hériter que d'une seule classe
- L'héritage multiple est donc non-autorisé.

Préparons la classe Enseignant

```
import { Personne } from "./personne";
export class Enseignant extends Personne {
}
```

Préparons la classe Etudiant

```
import { Personne } from "./personne";
export class Etudiant extends Personne {
}
```

extends est le mot-clé à utiliser pour définir une relation d'héritage entre deux classes

héritage

Ensuite

- Créer un attribut niveau dans la classe Etudiant ainsi que ses getter et setter
- Créer un attribut salaire dans la classe Enseignant ainsi que ses getter et setter

Pour créer un objet de type Enseignant

```
import { Enseignant } from './enseignant';
let enseignant: Enseignant = new Enseignant();
enseignant.num = 3;
enseignant.nom = "Trabelsi";
enseignant.prenom = "Anis";
enseignant.salaire = 5000;
console.log(enseignant);
```

En exécutant, le résultat est :

Enseignant { _num: 3, _nom: 'Trabelsi', _prenom: 'Anis', _salaire: 5000 }

héritage

TypeScript autorise la redéfinition : on peut définir un constructeur, même s'il existe dans la classe mère, qui prend plusieurs paramètres et qui utilise le constructeur de la classe mère

```
constructor(_num?: number,_nom?: string,_prenom?: string,
private _salaire?: number) {
super(_num, _nom, _prenom);
}
```

super() fait appel au constructeur de la classe mère

Maintenant, on peut créer un enseignant ainsi
let enseignant: Enseignant = new Enseignant(3, "Trabelsi", "Anis", 5000);

Refaire la même chose pour Etudiant

héritage

A partir de la classe Enseignant

- On ne peut avoir accès direct à un attribut de la classe mère
- C'est-à-dire, on ne peut faire this._num car les attributs ont une visibilité private
- Pour modifier la valeur d'un attribut privé de la classe mère, il faut :
  - ✓ soit utiliser les getters/setters
  - ✓ soit mettre la visibilité des attributs de la classe mère à protected

On peut créer un objet de la classe Enseignant ainsi

```
let enseignant: Enseignant = new Enseignant(3, "Trabelsi", "Anis",
5000);
```

Ou ainsi

```
let enseignant: Personne = new Enseignant(3, "Trabelsi", "Anis", 5000);
```

Ceci est faux

```
let enseignant: Enseignant = new Personne(3, "Trabelsi" , "Anis");
```

héritage

Remarque

Pour connaître la classe d'un objet, on peut utiliser le mot-clé instanceof

Exemple

```
let enseignant: Personne = new Enseignant(3, "Trabelsi", "Anis", 5000);
console.log(enseignant instanceof Enseignant);
// affiche true
console.log(enseignant instanceof Personne);
// affiche true
console.log(personne instanceof Enseignant);
// affiche false
```

Exercice

- Créer un objet de type Etudiant, un deuxième de type Enseignant et un dernier de type Personne stocker les tous dans un seul tableau.
- Parcourir le tableau et afficher pour chacun soit le numéro s'il est personne, soit le salaire s'il est enseignant ou soit le niveau s'il est étudiant.

Pour parcourir un tableau, on peut faire

```
let personnes: Array<Personne> = [personne, enseignant,etudiant];
for(let p of personnes) {      }
```

héritage

Solution

```
let personnes: Array<Personne> = [personne, enseignant, etudiant];
for(let p of personnes) {
  if(p instanceof Enseignant)
     console.log(p.salaire);
  else if (p instanceof Etudiant)
    console.log(p.niveau)  else
 console.log(p.num);
}
```

Classe et méthodes abstraites

Classe abstraite

- C'est une classe qu'on ne peut instancier
- On la déclare avec le mot-clé abstract

Si on déclare la classe Personne abstraite

```
export abstract class Personne {     ...    }
```

Tout ce code sera souligné en rouge

```
let personne: Personne = new Personne();
```

…

```
let personne2: Personne = new Personne(2, 'Trabelsi', 'ali');
```

Méthode abstraite

- C'est une méthode non implémentée (sans code)
- Une méthode abstraite doit être déclarée dans une classe abstraite
- Une méthode abstraite doit être implémentée par les classes filles de la classe abstraite

Classe et méthodes abstraites

Déclarons une méthode abstraite afficherDetails() dans Personne

```
abstract afficherDetails(): void ;
```

Remarque

- La méthode afficherDetails() dans Personne est soulignée en rouge car la classe doit être déclarée abstraite

- En déclarant la classe Personne abstraite, les deux classes Etudiant et Enseignant sont soulignées en rouge car elles doivent implémenter les méthodes abstraites de Personne

Pour implémenter la méthode abstraite

- Placer le curseur sur le nom de la classe
- Dans le menu afficher, cliquer sur Quick Fix puis Add inherited abstract class

Classe et méthodes abstraites

Le code généré

```
afficherDetails(): void {
throw new Error("Method not implemented.");
}
```

Remplaçons le code généré dans Etudiant par

```
afficherDetails(): void {
console.log(this.nom + " " + this.prenom + " " + this.niveau);
}
```

Et dans Enseignant par

```
afficherDetails(): void {
console.log(this.nom + " " + this.prenom + " " + this.salaire);
}
```

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "Trabelsi", "Anis", 5000);
enseignant.afficherDetails();
```

En exécutant, le résultat est :

Trabelsi Anis 5000

Interface

En TypeScript

- Une classe ne peut hériter que d'une seule classe
- Mais elle peut hériter de plusieurs interfaces

Une interface

- déclarée avec le mot-clé interface
- comme une classe complètement abstraite (impossible de l'instancier) dont : toutes les méthodes sont abstraites
- un protocole, un contrat : toute classe qui hérite d'une interface doit implémenter toutes ses méthodes

Interface

Définissons l'interface IMiseEnForme dans i-mise-en-forme.ts

```
export interface IMiseEnForme {
afficherNomMajuscule(): void;
afficherPrenomMajuscule() : void;
}
```

Pour hériter d'une interface, on utilise le mot-clé implements

```
export abstract class Personne implements IMiseEnForme {
...
}
```

La classe Personne est soulignée en rouge

- Placer le curseur sur la classe Personne
- Dans le menu afficher, cliquer sur Quick Fix puis Add inherited abstract class

Le code généré

```
afficherNomMajuscule(): void {
throw new Error("Method not implemented.");
}
afficherPrenomMajuscule(): void {
throw new Error("Method not implemented.");
        }
```

Interface

Modifions le code de deux méthodes générées

```
  afficherNomMajuscule(): void {
console.log(this.nom.toUpperCase());}
afficherPrenomMajuscule(): void
{console.log(this.prenom.toUpperCase());}
```

Pour tester

```
let enseignant: Enseignant =
 new Enseignant(3, " Trabelsi", "Anis", 5000);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```

En exécutant, le résultat est :

TRABELSI ANIS

Interface

Remarque

- Une interface peut hériter de plusieurs autres interfaces (mais pas d'une classe)
- Pour cela, il faut utiliser le mot-clé extends et pas implements car une interface n'implémente jamais de méthodes.

Une deuxième utilisation

- En TypeScript, une interface peut être utilisée comme une classe Model de plusieurs autres interfaces (mais pas d'une classe)
- Elle contient des attributs (qui sont par définition publiques) et des méthodes (abstraites)

Exemple

```
export interface Person {
num: number;
nom: string;
prenom: string;
}
```

Interface

Impossible d'instancier cette interface avec l'opérateur new, mais on peut utiliser les objets JavaScript

```
let person: Person = {
num: 1000,  nom: 'boudaga',
prenom: 'ali'
};
console.log(person);
// affiche { num: 1000, nom: 'boudaga', prenom: 'ali' }
```

On peut rendre les attributs optionnels

```
export interface Person {
num?: number;
nom?: string;
prenom?: string;
}
```

Interface

Ainsi on peut faire let person:

```
Person = {  nom: 'boudaga',};
console.log(person)
// affiche { nom: 'boudaga' }
```

Pour la suite, gardons l'attribut nom obligatoire

```
export interface Person {
 num?: number;
 nom: string;
 prenom?: string;
}
```

Interface

Duck typing

- Un concept un peu proche du polymorphisme
- Il se base sur une série d'attributs et de méthodes attendus
- L'objet est considéré valide quel que soit sa classe s'il respecte les attributs et les méthodes attendus.

Interface

Exemple : considérons la fonction afficherNom() définie dans file.ts

```
function afficherNom(p: Person){
console.log(p.nom)
}
```

Si l'objet passé en paramètre contient un attribut nom, alors ce dernier sera affiché

```
afficherNom(person);
// affiche boudaga
   afficherNom(personne);
// affiche Trabelsi
```

Ceci est aussi correcte car alien a un attribut nom

```
 let alien = { couleur: 'blanc', nom: 'white' };
afficherNom(alien);
// affiche white
```

Ceci génère une erreur car voiture n'a pas d'attribut nom

```
let voiture = { marque: 'citroen', modele: 'c5',
 num: 100000};
 afficherNom(voiture);
```

Exercice 4