TypeScript pour **Angular**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Plan

TypeScript

ECMAScript

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

TypeScript

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

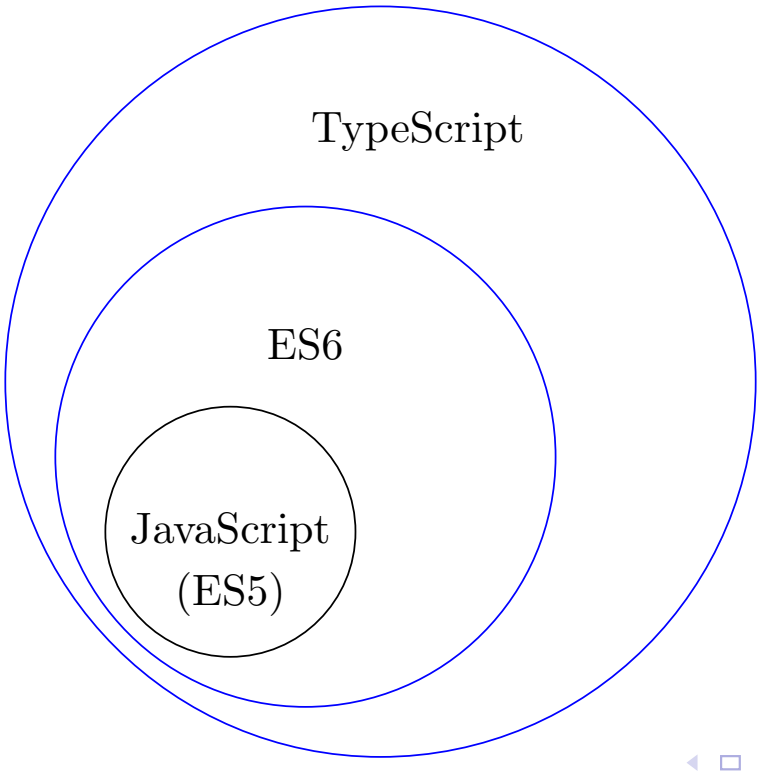
TypeScript

- 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**)

TypeScript

TypeScript : sur-couche de ES6 ajoutant

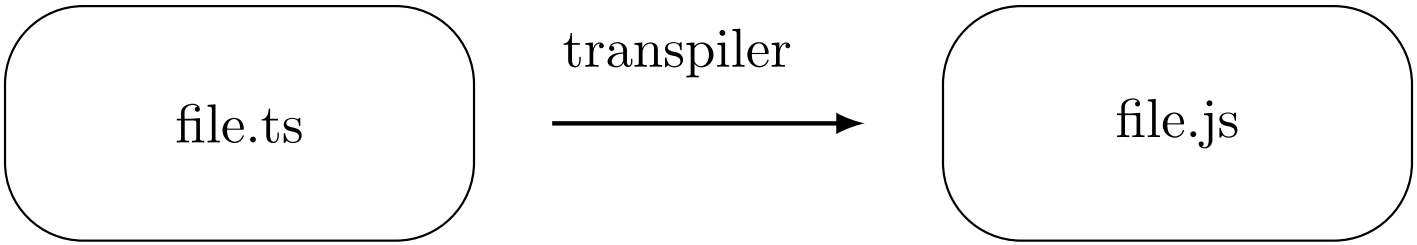
- typage
- meilleure gestion de module (à ne pas confondre avec les modules **Angular**)



TypeScript

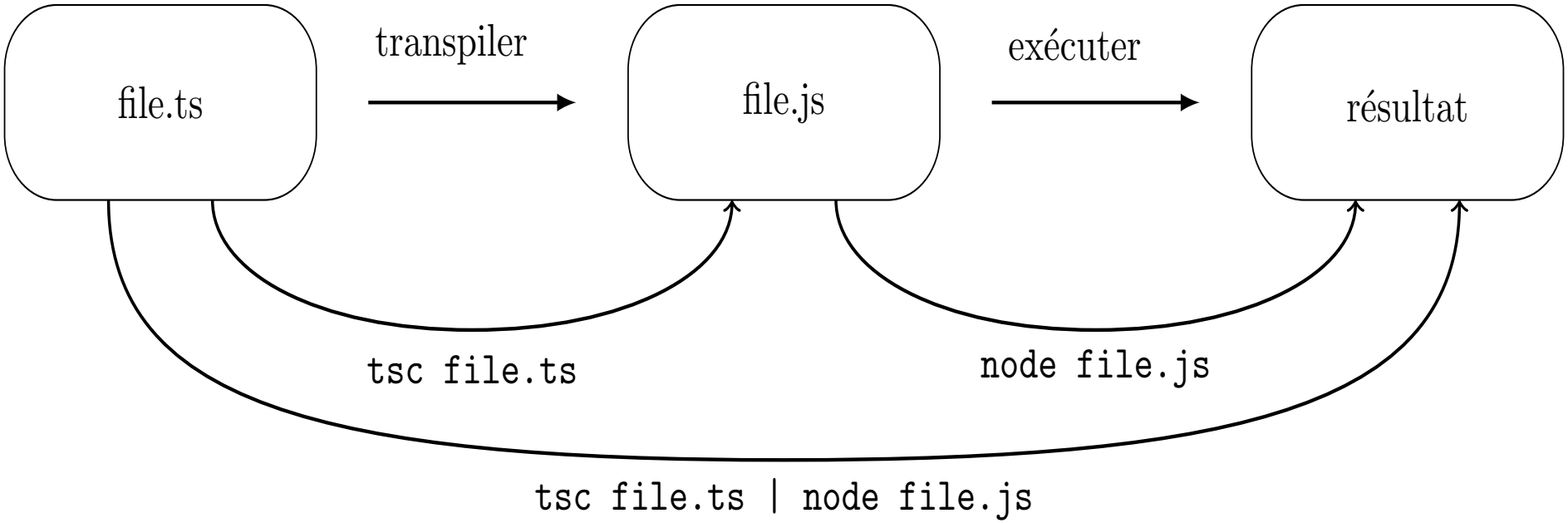
Le navigateur ne comprend pas le TypeScript

Il faut le transcompiler (ou transpiler) en JavaScript



TypeScript

Comment va t-on procéder dans ce cours ?



Pour consulter la liste d'ptions pour la commande `tsc`

`https://www.typescriptlang.org/docs/handbook/compiler-options.html`

TypeScript

De quoi on a besoin ?

- **Node.js** pour exécuter la commande `node`
- **TypeScript** pour exécuter la commande `tsc`

TypeScript

Pour **Node.js**, il faut

- aller sur `https://nodejs.org/en/`
- choisir la dernière version, télécharger et installer

TypeScript

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`

TypeScript

Quel IDE pour TypeScript (et éventuellement Angular)

- **Microsoft** recommande **Visual Studio Code**
- `code.visualstudio.com/download`

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

Initialiser une variable

```
x = 2;
```

TypeScript

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

Initialiser une variable

```
x = 2;
```

Déclarer et initialiser une variable

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

TypeScript

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

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

TypeScript

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)

TypeScript

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.

TypeScript

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

TypeScript

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

On peut aussi utiliser `template strings`

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

TypeScript

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

On peut aussi utiliser `template strings`

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

L'équivalent de faire

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

TypeScript

Une première déclaration pour les tableaux

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

TypeScript

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 ]
```

TypeScript

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 ]
```

TypeScript

Remarques

- En **JavaScript**, il n'y a pas de méthode `range` pour générer un tableau contenant un intervalle de valeurs entières consécutives.
- Avec **EcmaScript 6**, on peut utiliser des méthodes comme `from` et `keys` pour générer un intervalle de valeurs entières consécutives.

TypeScript

Remarques

- En **JavaScript**, il n'y a pas de méthode `range` pour générer un tableau contenant un intervalle de valeurs entières consécutives.
- Avec **EcmaScript 6**, on peut utiliser des méthodes comme `from` et `keys` pour générer un intervalle de valeurs entières consécutives.

Exemple

```
var list: number[] = Array.from(Array(3).keys())
console.log(list);
// affiche [ 0, 1, 2 ]
```

TypeScript

Remarques

- En **JavaScript**, il n'y a pas de méthode `range` pour générer un tableau contenant un intervalle de valeurs entières consécutives.
- Avec **EcmaScript 6**, on peut utiliser des méthodes comme `from` et `keys` pour générer un intervalle de valeurs entières consécutives.

Exemple

```
var list: number[] = Array.from(Array(3).keys())
console.log(list);
// affiche [ 0, 1, 2 ]
```

On peut utiliser une fonction fléchée (à voir dans une prochaine section) pour modifier les valeurs générées

```
var list: number[] = Array.from({ length: 3 }, (v, k) => k + 1)
console.log(list);
// affiche [ 1, 2, 3 ]
```

TypeScript

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

```
var t: [number, string, string] = [ 100, "wick", 'john' ];
```

TypeScript

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

```
var t: [number, string, string] = [ 100, "wick", 'john' ];
```

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, 'wick', 'travolta' ]
```

TypeScript

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

```
var t: [number, string, string] = [ 100, "wick", 'john' ];
```

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, 'wick', 'travolta' ]
```

Cependant, ceci génère une erreur

```
t = [100, 200, 'john'];
```

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
```

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] = 'wick';
```

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] = 'wick';
```

Ceci génère une erreur

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

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] = 'wick';
```

Ceci génère une erreur

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

Et cette instruction aussi car on dépasse la taille du tuple

```
t[3] = 100;
```

TypeScript

Exemple avec `any`

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

TypeScript

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;
```

TypeScript

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;
```

TypeScript

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;
```

Déclarons une énumération (dans `file.ts`)

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

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
```

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 };
```

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
```

TypeScript

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

TypeScript

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

En affichant, le résultat est

```
console.log(mois.MARS);
// affiche 3
console.log(mois.JUIN);
// affiche 12
console.log(mois.DECEMBRE);
// affiche 3
```

TypeScript

On peut aussi modifier plusieurs indices simultanément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL = 10, MAI, JUIN,
   JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE = 2, DECEMBRE };
```

En affichant, le résultat est

```
console.log(mois.MARS);
// affiche 3
console.log(mois.JUIN);
// affiche 12
console.log(mois.DECEMBRE);
// affiche 3
```

Ceci est une erreur, on ne peut modifier une constante

```
mois.JANVIER = 3;
```

TypeScript

Pour déclarer un objet

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

TypeScript

Pour déclarer un objet

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

On peut initialiser les attributs de cet objet

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

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

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

TypeScript

On peut modifier les valeurs d'un objet ainsi

```
obj.nom = 'abruzzi';
obj['numero'] = 200;

console.log(obj);
```

TypeScript

On peut modifier les valeurs d'un objet ainsi

```
obj.nom = 'abruzzi';
obj['numero'] = 200;

console.log(obj);
```

Ceci est une erreur

```
obj.nom = 125;
```

TypeScript

Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable

TypeScript

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;
```

TypeScript

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;
```

TypeScript

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];
```

TypeScript

Le mot-clé `let`

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

TypeScript

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
```

TypeScript

Premier exemple

```
let str: any = "bonjour";
let longueur: number = (<string>str).length;

console.log(longueur);
// affiche 7
```

TypeScript

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
```

TypeScript

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
```

TypeScript

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

TypeScript

Le mot-clé `type` permet de définir un alias de type

```
type maStructure = [number, string, string];
```

TypeScript

Le mot-clé `type` permet de définir un alias de type

```
type maStructure = [number, string, string];
```

Ensuite, on peut utiliser `maStructure` comme un type

```
let first: maStructure = [100, "wick", 'john' ];

console.log(first);
// affiche [ 100, 'wick', 'john' ]
```

TypeScript

L'opérateur `??` permet d'éviter d'affecter la valeur `null` ou `undefined` à une variable

```
var obj = {nom: null, prenom: 'john'};
let nom: string = obj.nom ?? 'doe';
console.log(nom);
// affiche doe
```

C'est équivalent à

```
var obj = {nom: null, prenom: 'john'};
let nom: string = (obj.nom !== null && obj.nom !==
   undefined) ? obj.nom : 'doe';
console.log(nom);
// affiche doe
```

TypeScript

Les constantes

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

TypeScript

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.
```

TypeScript

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
```

TypeScript

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";
```

TypeScript

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
```

TypeScript

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";
```

TypeScript

Déclarer une fonction

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

TypeScript

Déclarer une fonction

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

Exemple

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

TypeScript

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
```

TypeScript

Le code suivant génère une erreur

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

TypeScript

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);
```

TypeScript

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);
```

TypeScript

Une fonction qui ne retourne rien a le type `void`

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

TypeScript

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){

    }
}
```

TypeScript

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

TypeScript

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
```

TypeScript

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

TypeScript

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
```

TypeScript

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

TypeScript

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
```

TypeScript

Il est possible d'autoriser plusieurs types pour un paramètre

TypeScript

Il est possible d'autoriser plusieurs types pour un paramètre

```
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
```

TypeScript

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]++;
  }
}
```

TypeScript

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]++;
  }
}
```

TypeScript

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

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

TypeScript

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; }
```

TypeScript

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;
```

TypeScript

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);
```

TypeScript

Cas d'une fonction fléchée à un seul paramètre

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

TypeScript

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
```

TypeScript

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
```

TypeScript

Remarque

- Il est déconseillé d'utiliser les fonctions fléchées dans un objet
- Le mot-clé `this` est inutilisable dans les fonctions fléchées

TypeScript

Remarque

- Il est déconseillé d'utiliser les fonctions fléchées dans un objet
- Le mot-clé `this` est inutilisable dans les fonctions fléchées

Sans les fonctions fléchées

```
let obj = {
  nom: 'wick',
  afficherNom: function() {
    console.log(this.nom)
  }
}
obj.afficherNom();
// affiche wick
```

Avec les fonctions fléchées

```
let obj = {
  nom: 'wick',
  afficherNom: () => {
    console.log(this.nom)
  }
}
obj.afficherNom();
// affiche undefined
```

TypeScript

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
- `...`

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

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

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

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

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

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

function afficher(value) {
    console.log(value);
}
// affiche 2 3 5
```

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

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

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

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

function afficher(value) {
    console.log(value);
}
// affiche 2 3 5
```

On peut simplifier l'écriture précédente en utilisant les callback

```
tab.forEach(afficher);

function afficher(value) {
    console.log(value);
}
// affiche 2 3 5
```

TypeScript

La fonction `afficher` peut accepter deux paramètres : le premier est la valeur de l'itération courante et le deuxième est son indice dans le tableau

```
tab.forEach(afficher);

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

/* affiche
0 2
1 3
2 5
*/
```

TypeScript

La fonction `afficher` peut accepter un troisième paramètre qui correspond au tableau

```
tab.forEach(afficher);

function afficher(value, key, t) {
    console.log(key, value, t);
}

/* affiche
0 2 [ 2, 3, 5 ]
1 3 [ 2, 3, 5 ]
2 5 [ 2, 3, 5 ]
*/
```

TypeScript

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
```

TypeScript

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
```

TypeScript

Remarque

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

TypeScript

Remarque

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

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
```

TypeScript

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
```

TypeScript

Remarques

- 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

TypeScript

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
```

TypeScript

Fonctions fléchées : pourquoi ?

- Simplicité d'écriture du code $\Rightarrow$ meilleure lisibilité
- Pas de binding avec les objets prédéfinis : `arguments`, `this`...
- ...

Considérons la fonction `somme` suivante

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

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
```

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));
```

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));
```

On peut utiliser partiellement la décomposition

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

TypeScript

Considérons les deux objets suivants

```
let obj = { nom: 'wick', prenom: 'john'};
let obj2 = obj;
```

TypeScript

Considérons les deux objets suivants

```
let obj = { nom: 'wick', prenom: 'john'};
let obj2 = obj;
```

Modifier l'un $\Rightarrow$ modifier l'autre

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

console.log(obj2);
// affiche { nom: 'abruzzi', prenom: 'john' }
```

TypeScript

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

```
let obj = { nom: 'wick', prenom: 'john'};
let obj2 = { ...obj };
obj2.nom = 'abruzzi';

console.log(obj);
// affiche { nom: 'wick', prenom: 'john' }

console.log(obj2);
// affiche { nom: 'abruzzi', prenom: 'john' }
```

TypeScript

Particularité

- Avec **TypeScript**, on peut 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.

TypeScript

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;
}
```

TypeScript

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;
}
```

TypeScript

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 };
```

TypeScript

Pour importer et utiliser une fonction

```
import { somme } from './fonctions';

console.log(somme(2, 5));
// affiche 7
```

TypeScript

Pour importer et utiliser une fonction

```
import { somme } from './fonctions';

console.log(somme(2, 5));
// affiche 7
```

On peut aussi utiliser des alias

```
import  { somme as s } from './fonctions';

console.log(s(2, 5));
// affiche 7
```

TypeScript

Pour importer plusieurs éléments

```
import { somme, produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

TypeScript

Pour importer plusieurs éléments

```
import { somme, produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

Pour importer plusieurs éléments

```
import  * as f from './fonctions';

console.log(f.somme(2, 5));
// affiche 7

console.log(f.produit(2, 5));
// affiche 10
```

TypeScript

On peut aussi donner des alias pendant l'export

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

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

TypeScript

On peut aussi donner des alias pendant l'export

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

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

Pour importer

```
import * as f from './fonctions';

console.log(f.s(2, 5));
// affiche 7

console.log(f.p(2, 5));
// affiche 10
```

TypeScript

On peut aussi utiliser le `export default` (un seul par fichier)

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

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

TypeScript

On peut aussi utiliser le `export default` (un seul par fichier)

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

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

Pour importer, pas besoin de { } pour les éléments exporter par défaut

```
import somme from './fonctions';
import { produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

TypeScript

Attention, ici on a importé `somme` avec deux alias différents

```
import s from './fonctions';
import produit from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 7
```

TypeScript

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)

TypeScript

Qu'est ce qu'une classe en POO ?

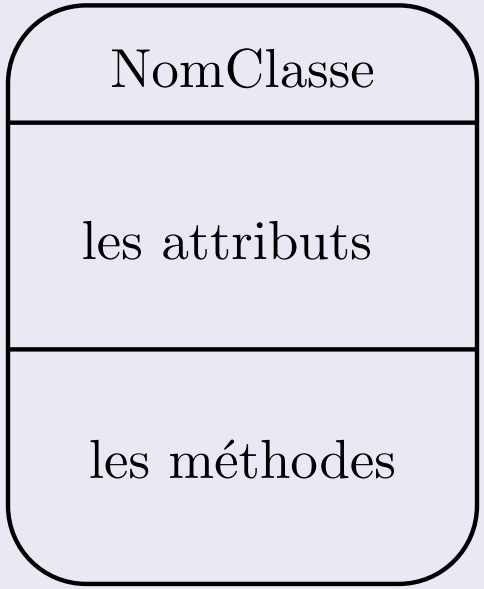
- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

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
- instance $\equiv$ objet

TypeScript

De quoi est composé une classe ?



- Attribut : [visibilité] + nom + type
- Méthode : [visibilité] + nom + arguments + valeur de retour ≡ signature : exactement comme les fonctions en procédurale

TypeScript

Considérons la classe `Personne` définie dans `personne.ts`

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

TypeScript

Considérons la classe `Personne` définie dans `personne.ts`

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

En TypeScript

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

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

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

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

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

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

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

Étape 2 : déclarons un objet (objet non créé)

```
let  personne: Personne;
```

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

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

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

Étape 2 : déclarons un objet (objet non créé)

```
let  personne: Personne;
```

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

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

TypeScript

Hypothèse

Si on voulait créer un objet de la classe `Personne` avec les valeurs `1`, `wick` et `john`

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

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

Étape 2 : déclarons un objet (objet non créé)

```
let  personne: Personne;
```

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

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

On peut faire déclaration + instanciation

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

TypeScript

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

TypeScript

Affectons les valeurs aux différents attributs

```
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";
```

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

```
Console.log(personne)
// affiche Personne { num: 1, nom: 'wick', prenom: '
   john' }
```

TypeScript

Hypothèse

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

TypeScript

Hypothèse

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

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

TypeScript

Hypothèse

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

Démarche

1. Bloquer l'accès directe aux attributs (mettre la visibilité à `private`)
2. 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

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

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

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 = "wick";
personne.prenom = "john";
```

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 = "wick";
personne.prenom = "john";
```

Explication

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

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 = "wick";
personne.prenom = "john";
```

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

TypeScript

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

TypeScript

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;
    }
}
```

TypeScript

Pour tester, rien à changer dans `file.ts`

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

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

TypeScript

Pour tester, rien à changer dans `file.ts`

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

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

Pour transpiler, ajouter l'option `-t es5`, le résultat est :

```
Personne { _num: 1, _nom: 'wick', _prenom: 'john' }
```

TypeScript

Testons avec une valeur négative pour l'attribut `numero`

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

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

TypeScript

Testons avec une valeur négative pour l'attribut `numero`

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

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

Le résultat est :

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }
```

TypeScript

Question

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

TypeScript

Question

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

Démarche

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

TypeScript

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

TypeScript

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;
}
```

TypeScript

Pour tester

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

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne.num);
// affiche 1

console.log(personne.nom);
// affiche wick

console.log(personne.prenom);
// affiche john
```

TypeScript

Remarques

- 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

TypeScript

Remarques

- 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

TypeScript

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;
}
```

TypeScript

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;
}
```

TypeScript

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;
}
```

TypeScript

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

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

TypeScript

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

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

Explication

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

TypeScript

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

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

Explication

Le constructeur par défaut a 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

TypeScript

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;
}
```

TypeScript

Pour tester

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

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";
console.log(personne);

let personne2: Personne = new Personne(2, 'bob', 'mike');
console.log(personne2);
```

TypeScript

Pour tester

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

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";
console.log(personne);

let personne2: Personne = new Personne(2, 'bob', 'mike');
console.log(personne2);
```

En exécutant, le résultat est :

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }
Personne { _num: 2, _nom: 'bob', _prenom: 'mike' }
```

TypeScript

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) {

}
```

TypeScript

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: 'wick', _prenom: 'john' }
Personne { _num: 2, _nom: 'bob', _prenom: 'mike' }
```

TypeScript

Récapitulatif

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

TypeScript

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`)

TypeScript

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.

TypeScript

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

TypeScript

Ajoutons un attribut statique `_nbrPersonnes` à la liste d'attributs de la classe `Personne`

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

TypeScript

Ajoutons un attribut statique `_nbrPersonnes` à la liste d'attributs de la classe `Personne`

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

Incrémentons notre compteur de personnes dans les constructeurs

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

TypeScript

Créons un `getter` pour l'attribut static `_nbrPersonnes`

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

TypeScript

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 = "wick";
personne.prenom = "john";
console.log(Personne.nbrPersonnes);
// affiche 1
let personne2: Personne = new Personne(2, 'bob', 'mike');
console.log(Personne.nbrPersonnes);
// affiche 2
```

TypeScript

Exercice

- Définir une classe `Adresse` avec trois attributs privés `rue`, `codePostal` et `ville` de type chaîne de caractère
- Définir un constructeur avec trois paramètres, les getters et setters
- Dans la classe `Personne`, ajouter un attribut `adresse` (de type `Adresse`) et définir un nouveau constructeur à quatre paramètres et le getter et le setter de ce nouvel attribut
- Dans `file.ts`, créer deux objets : un objet `adresse` (de type `Adresse`) et `personne` (de type `Personne`) prenant comme adresse l'objet `adresse`
- Afficher tous les attributs de l'objet `personne`

TypeScript

L'héritage, quand ?

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

TypeScript

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
};
```

TypeScript

Exemple

- Un enseignant a un numéro, un nom, un prénom et un salaire

TypeScript

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

TypeScript

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

TypeScript

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`

TypeScript

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 peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom` et `prénom`

TypeScript

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 peut utiliser la classe `Personne` puisqu'elle contient tous les attributs `numéro`, `nom` et `prénom`
- Les classes `Étudiant` et `Enseignant` hériteront donc (`extends`) de la classe `Personne`

TypeScript

Particularité du langage **TypeScript**

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

TypeScript

Préparons la classe `Enseignant`

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

export class Enseignant extends Personne {

}
```

TypeScript

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

TypeScript

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

TypeScript

Pour créer un objet de type `Enseignant`

```
import { Enseignant } from './enseignant';

let enseignant: Enseignant  = new Enseignant();
enseignant.num = 3;
enseignant.nom = "green";
enseignant.prenom = "jonas";
enseignant.salaire = 1700;
console.log(enseignant);
```

TypeScript

Pour créer un objet de type `Enseignant`

```
import { Enseignant } from './enseignant';

let enseignant: Enseignant  = new Enseignant();
enseignant.num = 3;
enseignant.nom = "green";
enseignant.prenom = "jonas";
enseignant.salaire = 1700;
console.log(enseignant);
```

En exécutant, le résultat est :

```
Enseignant { _num: 3, _nom: 'green', _prenom: 'jonas
   ', _salaire: 1700 }
```

TypeScript

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);
}
```

TypeScript

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

TypeScript

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, "green", "jonas"
   , 1700);
```

TypeScript

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, "green", "jonas"
   , 1700);
```

Refaire la même chose pour `Etudiant`

TypeScript

À 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`

TypeScript

On peut créer un objet de la classe `Enseignant` ainsi

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
```

TypeScript

On peut créer un objet de la classe `Enseignant` ainsi

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
```

Ou ainsi

```
let enseignant: Personne = new Enseignant(3, "green"
   , "jonas", 1700);
```

TypeScript

On peut créer un objet de la classe `Enseignant` ainsi

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
```

Ou ainsi

```
let enseignant: Personne = new Enseignant(3, "green"
   , "jonas", 1700);
```

Ceci est faux

```
let enseignant: Enseignant = new Personne(3, "green"
   , "jonas");
```

TypeScript

Remarque

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

TypeScript

Remarque

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

Exemple

```
let enseignant: Personne = new Enseignant(3, "green", "jonas",
   1700);

console.log(enseignant instanceof Enseignant);
// affiche true

console.log(enseignant instanceof Personne);
// affiche true

console.log(personne instanceof Enseignant);
// affiche false
```

TypeScript

Exercice

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

TypeScript

Exercice

1. Créer un objet de type `Etudiant`, un deuxième de type `Enseignant` et un dernier de type `Personne` stocker les tous dans un seul tableau.
2. Parcourir le tableau et afficher pour chacun soit le `numero` 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) {

}
```

TypeScript

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);
}
```

TypeScript

Classe abstraite

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

TypeScript

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 {
    ...
}
```

TypeScript

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, 'bob', 'mike');
```

TypeScript

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

TypeScript

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

Déclarons une méthode abstraite `afficherDetails()` **dans** `Personne`

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

TypeScript

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

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`

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`

Le code généré

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

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`

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);
}
```

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`

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);
}
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherDetails();
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherDetails();
```

En exécutant, le résultat est :

```
green jonas 1700
```

TypeScript

En TypeScript

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

TypeScript

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

TypeScript

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

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

TypeScript

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 {
    ...
}
```

TypeScript

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`

TypeScript

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.");
}
```

TypeScript

Modifions le code de deux méthodes générées

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

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

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```

TypeScript

Pour tester

```
let enseignant: Enseignant = new Enseignant(3, "
   green", "jonas", 1700);
enseignant.afficherNomMajuscule();
enseignant.afficherPrenomMajuscule();
```

En exécutant, le résultat est :

```
GREEN
JONAS
```

TypeScript

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.

TypeScript

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)

TypeScript

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;
}
```

TypeScript

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

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

TypeScript

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

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

On peut rendre les attributs optionnels

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

TypeScript

Ainsi on peut faire

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

TypeScript

Ainsi on peut faire

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

Pour la suite, gardons l'attribut `nom` obligatoire

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

TypeScript

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.

TypeScript

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.

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

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

TypeScript

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

```
afficherNom(person);
// affiche turing

afficherNom(personne);
// affiche wick
```

TypeScript

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

```
afficherNom(person);
// affiche turing

afficherNom(personne);
// affiche wick
```

Ceci est aussi correcte car `alien` a un attribut `nom`

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

TypeScript

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

```
afficherNom(person);
// affiche turing

afficherNom(personne);
// affiche wick
```

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: 'ford', modele: 'fiesta', num: 100000};
afficherNom(voiture);
```

TypeScript

Décorateur

- L'équivalent d'annotations en **Java** et **php**
- Méta-programmation (modification des informations/comportement sur un objet/classe)
- Utilisé avec le préfixe `@`

TypeScript

Décorateur

- L'équivalent d'annotations en **Java** et **php**
- Méta-programmation (modification des informations/comportement sur un objet/classe)
- Utilisé avec le préfixe `@`

Ajoutons le décorateur `@f()` à `afficherDetails()` dans `Enseignant`

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

TypeScript

Le décorateur `@f()` n'existe pas en TypeScript, il faut donc le définir (en lui associant une fonction)

```
export function f() {
    return function(a,b,c){
        console.log('before afficherDetails()');
    }
}
```

TypeScript

Le décorateur `@f()` n'existe pas en TypeScript, il faut donc le définir (en lui associant une fonction)

```
export function f() {
    return function(a,b,c){
        console.log('before afficherDetails()');
    }
}
```

Testons maintenant le code suivant et vérifions que la fonction associée au décorateur a bien été exécutée (ajouter l'option `--experimentalDecorators` à la commande de transpilation)

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherDetails();
```

TypeScript

Le décorateur `@f()` n'existe pas en TypeScript, il faut donc le définir (en lui associant une fonction)

```
export function f() {
    return function(a,b,c){
        console.log('before afficherDetails()');
    }
}
```

Testons maintenant le code suivant et vérifions que la fonction associée au décorateur a bien été exécutée (ajouter l'option `--experimentalDecorators` à la commande de transpilation)

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas", 1700);
enseignant.afficherDetails();
```

Le résultat est :

```
before afficherDetails()
green jonas 1700
```

TypeScript

Généricité

- Un concept défini dans tous les LOO avec $<\ \ldots\ >$
- Elle permet de définir des fonctions, classes, interfaces qui s'adaptent avec plusieurs types

TypeScript

Exemple

- si on a besoin d'une classe dont les méthodes effectuent les mêmes opérations quel que soit le type d'attributs
  - **somme** pour **entiers** ou **réels**,
  - **concaténation** pour **chaînes de caractères**,
  - **ou logique** pour **booléens**...
  - ...
- **Impossible sans définir plusieurs classes (une pour chaque type)**

TypeScript

Solution avec la généricité

```
export class Operation<T>{

  constructor(private var1: T, private var2: T) { }

  public plus() {
    if (typeof this.var1 == 'string') {
      return this.var1 + this.var2;
    }
    else if (typeof this.var1 == 'number' && typeof this.var2 == '
       number') {
      return this.var1 + this.var2;
    }
    else if (typeof this.var1 == 'boolean' && typeof this.var2 == '
       boolean') {
      return this.var1 || this.var2;
    }
    else {
      throw "error"
    }
  }
}
```

TypeScript

Nous pouvons donc utiliser la même méthode qui fait la même chose pour des types différents

```
let operation1: Operation<number>  = new Operation(5, 3);
console.log(operation1.plus());
// affiche 8

let operation2: Operation<string> = new Operation("bon", "jour");
console.log(operation2.plus());
// affiche bonjour

let operation3: Operation<number> = new Operation(5.2, 3.8);
console.log(operation3.plus());
// affiche 9

let operation4: Operation<boolean> = new Operation(true, false);
console.log(operation4.plus());
// affiche true
```

TypeScript

`Map` (dictionnaire)

- Type fonctionnant avec un couple (clé,valeur)
- La clé doit être unique
- Chaque élément est appelé entrée (`entry`)
- Les éléments sont stockés et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

TypeScript

`Map` (dictionnaire)

- Type fonctionnant avec un couple (clé,valeur)
- La clé doit être unique
- Chaque élément est appelé entrée (`entry`)
- Les éléments sont stockés et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

Pour tester, utiliser l'option `-t ES6`

TypeScript

Pour créer un `Map`

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche  Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

TypeScript

Pour créer un `Map`

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche  Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

Ajouter un élément à un `Map`

```
map.set('html', 18);
```

TypeScript

Pour créer un `Map`

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche  Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

Ajouter un élément à un `Map`

```
map.set('html', 18);
```

Pour ajouter plusieurs éléments à la fois

```
map.set('html', 18)
    .set('css', 12);
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

Pour vérifier l'existence d'une clé

```
console.log(map.has('php'));
// affiche true
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

Pour vérifier l'existence d'une clé

```
console.log(map.has('php'));
// affiche true
```

Pour supprimer un élément selon la clé

```
map.delete('php');
```

TypeScript

Pour récupérer la liste des clés d'un `Map`

```
console.log(map.keys());
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

TypeScript

Pour récupérer la liste des clés d'un `Map`

```
console.log(map.keys());
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

Pour récupérer la liste des valeurs d'un `Map`

```
console.log(map.values());
// affiche [Map Iterator] { 10, 12, 20, 12 }
```

TypeScript

Pour récupérer la liste des clés d'un `Map`

```
console.log(map.keys());
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

Pour récupérer la liste des valeurs d'un `Map`

```
console.log(map.values());
// affiche [Map Iterator] { 10, 12, 20, 12 }
```

Pour récupérer la liste des entrées d'un `Map`

```
console.log(map.entries());
/* affiche
[Map Entries] {
  [ 'java', 10 ],
  [ 'c', 12 ],
  [ 'html', 20 ],
  [ 'css', 12 ]
}
*/
```

TypeScript

Pour parcourir un `Map`, on peut utiliser `entries()` (solution ES5)

```
for (let elt of map.entries())
    console.log(elt[0] + " " + elt[1]);
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

Pour parcourir un `Map`, on peut utiliser `entries()` (solution ES5)

```
for (let elt of map.entries())
    console.log(elt[0] + " " + elt[1]);
/* affiche
java 10
c 12
html 20
css 12
*/
```

Depuis ES6, on peut faire

```
for (let [key, value] of map) {
    console.log(key, value);
}
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

On peut le faire aussi avec `keys()`

```
for (let key of map.keys()) {
    console.log(key + " " + map.get(key));
}
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

Une deuxième solution consiste à utiliser `forEach` (affiche seulement les valeurs)

```
map.forEach(elt => console.log(elt));

/* affiche
10
12
20
12
*/
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach` (affiche seulement les valeurs)

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

function afficher(elt) {
    console.log(elt)
}
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach` (affiche seulement les valeurs)

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

function afficher(elt) {
    console.log(elt)
}
```

On peut encore simplifier l'appel de la fonction avec les callback (affiche seulement les valeurs)

```
map.forEach(afficher);

function afficher(elt){
    console.log(elt)
}
```

TypeScript

Pour afficher les clés et les valeurs

```
map.forEach(afficher);

function afficher(value, key) {
    console.log(value, key)
}

/* affiche
10 java
12 c
20 html
12 css
*/
```

TypeScript

Set

- Une collection ne contenant pas de doublons
- Acceptant les types simples et objets
- Les éléments sont stockées et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

TypeScript

`Set`

- Une collection ne contenant pas de doublons
- Acceptant les types simples et objets
- Les éléments sont stockées et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

Pour tester, utiliser l'option `-t ES6`

TypeScript

Pour créer un `Set`

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche  Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

TypeScript

Pour créer un `Set`

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche  Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

Ajouter un élément à un `Set`

```
marques.add('citroen');
```

TypeScript

Pour créer un `Set`

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche  Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

Ajouter un élément à un `Set`

```
marques.add('citroen');
```

Pour ajouter plusieurs éléments à la fois

```
marques.add('citroen')
       .add('renault');
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', '
   citroen', 'renault' }
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', '
   citroen', 'renault' }
```

Pour vérifier l'existence d'un élément

```
console.log(marques.has('fiat'));
// affiche true
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', '
   citroen', 'renault' }
```

Pour vérifier l'existence d'un élément

```
console.log(marques.has('fiat'));
// affiche true
```

Pour supprimer un élément

```
marques.delete('ford');
console.log(marques);
// affiche Set { 'peugeot', 'fiat', 'mercedes', 'citroen', '
   renault' }
```

TypeScript

Autres méthodes sur les `Set`

- `A.subSet(B)` : retourne `true` si `A` est un sous-ensemble de `B`, `false` sinon.
- `A.union(B)` : retourne un **Set** regroupant les éléments de `A` et de `B`.
- `A.intersection(B)` : retourne un **Set** contenant les éléments de `A` qui sont dans `B`.
- `A.difference(B)` : retourne un **Set** contenant les éléments de `A` qui ne sont pas dans `B`.

TypeScript

Pour parcourir un `Set`

```
for(let marque of marques) {
    console.log(marque)
}
/* affiche
peugeot
fiat
mercedes
citroen
renault
*/
```

TypeScript

Une deuxième solution consiste à utiliser `forEach`

```
marques.forEach(elt => console.log(elt));

/* affiche
peugeot
fiat
mercedes
citroen
renault
*/
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach`

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

function afficher(elt){
    console.log(elt)
}
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach`

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

function afficher(elt){
    console.log(elt)
}
```

On peut encore simplifier l'appel de la fonction avec les callback

```
marques.forEach(afficher);

function afficher(elt){
    console.log(elt)
}
```