Cours d'Angular 11

Partie I: JAVASCRIPT ET TYPESCRIPT

Birane Baila Wane
Email:douvewane85@gmail.com

PARTIE I: JAVASCRIPT ET TYPESCRIPT

I) JavaScript

- Bases de JavaScript
- Fonctions
  - nommées
  - anonymes
  - Fléchées
  - Callbacks
- Tableaux
- Quelques exemples de Fonctions
  - filter()
  - map()
  - foreach()
- Objets

PARTIE I: JAVASCRIPT ET TYPESCRIPT

II) **ECMAScript 5 & 6**

- Mode strict
- var, let et const
- Fonctions fléchées
- POO
  - Classe
    - Attributs Instances
    - Attributs Static
    - Surcharge
  - Heritage
    - Redefinition
  - Module
    - export
    - import

PARTIE I: JAVASCRIPT ET TYPESCRIPT

III) **TypeScript**

- Transpilation
- Types
- Fonctions fléchées
  - Statique
  - Dynamique
  - Générique
- Les Promises
- POO
  - Classe
  - Interface
  - Module
- COLLECTIONS
  - Map
  - Set

Rappels JavaScript, les Bases

Définition, histoire et rôle

- est un langage de programmation de scripts orienté objet (à prototype)
- créé par Brendan Eich
- présenté par **Netscape** et **Sun Microsystems** en décembre 1995
- standardisé par **ECMAScript** en juin 1997 pour
- permettant de
  - compléter l'aspect algorithmique manquant à **HTML** (et **CSS**)
  - rendre plus vivant le site web avec notamment des animations, effets et de l'interaction avec l'internaute (le visiteur, le client...).

Rappels JavaScript, les Bases

Spécificités du **JavaScript**

- langage faiblement typé
- syntaxe assez proche de celle de Java, C++, C...
- possibilité d'écrire plusieurs instructions sur une seule ligne à condition de les séparer par `;`
- terminer une instruction par `;` est fortement recommandée même si cette dernière est la seule sur la ligne

Rappels JavaScript, les Bases

Autres langages de programmation de scripts

- AppleScript
- JScript, VBScript et TypeScript (Microsoft)
- LiveScript (Netscape) puis JavaScript
- ActionScript (MacroMedia)
- CoffeeScript (Open-source)
- ...

Bookmarks
chrome://bookmarks

Rappels JavaScript, Exemple

```
<!DOCTYPE html>
<html>
<head>
<title>First JS Page</title>
<script type="text/javascript">
  function maFonction() {
    alert("Hello World !");
  }
</script>
</head>
<body>
  <button onclick="maFonction()">
    Cliquer ici
  </button>
</body>
</html>
```

Remarque

La console, pourquoi ?

- permet de contrôler l'avancement de l'exécution d'un programme
  - en affichant le contenu de variables (déboguer)
  - en vérifiant les blocs du code visites (tracer)

Rappels JavaScript, les Bases

Quelques types de variable selon la valeur affectée

- number
- string
- boolean
- object
- undefined

Rappels JavaScript, les String

Méthodes utiles pour les chaînes de caractères

- `length` : la longueur de la chaîne
- `toUpperCase()` : pour convertir une chaîne de caractères en majuscule
- `toLowerCase()` : pour convertir une chaîne de caractères en minuscule
- `trim()` : pour supprimer les espaces au début et à la fin
- `substr()` : pour extraire une sous-chaîne de caractères
- `indexOf()` : pour retourner la position d'une sous-chaîne dans une chaîne, -1 sinon.
- ...

Rappels JavaScript, les Fonctions

Déclarer une fonction

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

Exemple

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

Déclarer une fonction anonym[illegible]ffecter a‘ une variable

Bookmarks
chrome://bookmarks

```
var nomVariable = function ([les arguments]) {
  les instructions de la fonction
}
```

Exemple

```
var somme2 = function (a, b){
  return a + b;
}
```

Déclarer une fonction en utilisant un constructeur de fonction

```
var nomFunction = new Function (["param1", ... , "
paramN",] "instructions");
```

Exemple

```
var somme4 = new Function ("a", "b", "return a + b")
   ;
```

Appeler une fonction anonyme

```
var resultat4 = somme4 (1, 3);
```

On peut affecter une valeur par défaut aux paramètres

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

Et si on veut que la fonction somme retourne la somme quel que soit le nombre de paramètres

```
function somme () {
  result = 0;
  for(var i = 0; i < arguments.length ; i++) {
    result += arguments[i];
  }
  return result;
}
```

Tous ces appels sont corrects

```
console.log(somme(2, 3, 5));
console.log(somme(2, 3));
console.log(somme(2, 3, 5, 7));
console.log(somme(2));
```

Fonction de retour (callback)

- fonction appelée comme un paramètre d'une deuxième fonction
- très utilisée en **JavaScript** (**jQuery** et **nodeJS**) avec les fonctions asynchrones

Bookmarks
chrome://bookmarks

Considérons les deux fonctions suivantes

```
function somme(a, b) {
  return a + b;
}
function produit(a, b) {
  return a * b;
}
```

Utilisons les fonctions précédentes comme callback d'une fonction `operation()`

```
function operation(a,b,fonction) {
  console.log (fonction(a, b));
}

// appeler la fonction opération
operation (3, 5, somme);
// affiche 8
```

La fonction `setTimeout` accepte deux paramètres

WariCustomAthentificatorAuthenticator.php
todo: check the credentials inside F:\wamp64\...

- le premier : une fonction callback
- le deuxième : une durée (en millisecondes) qui précède l'exécution de la fonction callback

Déclarons une fonction qui affiche bonjour

```
function direBonjour() {
  alert("Bonjour");
}
```

Utilisons cette fonction comme callback dans `setTimeout`

```
function direBonjourApresXSecondes(x) {
  setTimeout(direBonjour, x * 1000);
}
```

Appelons `direBonjourApresXSecondes` et bonjour sera affiché après

```
direBonjourApresXSecondes(5);
```

todo: check the credentials inside F:\wamp64\...

Il est aussi d'annuler l'exécution de la fonction callback de `setTimeout` avec `clearTimeout` si cette dernière est appelée avant la fin de la durée de `setTimeout`

```
var timeout;
function direBonjourApresXSecondes(x) {
  timeout = setTimeout(function(){
    alert("Bonjour");
  },
  x * 1000);
}

direBonjourApresXSecondes(5);

clearTimeout(timeout);
// Bonjour ne sera jamais affiché
```

Rappels JavaScript, les Tableaux

Tableau (array)

- une variable pouvant avoir simultanément plusieurs valeurs
- l'accès à chaque valeur s'effectue via son indice et avec l'opérateur `[]` : `[ indice ]`
- la première valeur (on dit aussi élément) du tableau est d'indice 0.
- les valeurs peuvent avoir plusieurs types différents

Déclaration

```
var tab = new Array(value1, value2,... valueN);
```

Le raccourci

```
var tab = [value1, value2,... valueN];
```

Accès à l'élément du tableau d'indice `i`

```
tab[i]
```

Opérations sur les tableaux

- `push()` : ajoute un élément passé en paramètre au tableau
- `pop()` : supprime le dernier élément du tableau
- `shift()` : supprime le premier élément du tableau
- `indexOf()` : retourne la position de l'élément, passé en paramètre, dans le tableau, `-1` sinon.
- `reverse()` : inverse l'ordre des éléments du tableau
- `sort()` : trie un tableau
- `splice()` : permet d'extraire, ajouter ou supprimer un ou plusieurs éléments (selon les paramètres, voir slide suivante)
- `includes()` : retourne `true` si la valeur passée en paramètre est dans un tableau, `false` sinon.
- `...`

Autres opérations sur les tableaux (ES5)

- `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
- `...`

Exemple avec `map`, `filter`, `forEach`

```
var tab = [2, 3, 5];
tab.map(x => x + 3)
    .filter(x => x > 5)
    .forEach(
        function(a,b){
            console.log(a-3);
        }
    );
// affiche 3 et 5
```

On peut permuter map et filter selon le besoin
On peut aussi remplacer les fonctions fléchée par des fonctions anonymes

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

```
var tab =[2, 3, 5];
var somme = tab.map(x => x + 3)
    .filter(x => x > 5)
    .reduce(function(sum, elem){
        return sum + elem;
    });

console.log(somme);
// affiche 14
```

Rappels JavaScript, Objets

Comment sauvegarder toutes ces donnees dans une seule variable ?

nom	wick
prénom	john
⋮	⋮

© Achref

Comment faire ?

- Les tableaux indexés ne le permettent pas
- Les tableaux associatifs ou les objets

Objet ?

un ensemble de

Bookmarks
chrome://bookmarks

- attributs (variables, champs) : clé + valeur[s]
- méthodes : fonctions

Création d’un objet

```
var obj = {
  nom: "wick",
  prenom: "john"
};
```

Bookmarks
chrome://bookmarks

Accès à un attribut de l’objet

```
console.log(obj.nom);
console.log(obj["prenom"]);
```

Un objet est non-itérable avec `for ... of`

```
for(var elt of obj)
  console.log(elt);
```

Il est itérable avec `for ... in`

```
for(var key in obj)
  console.log(key + " : " + obj[key]);
```

Pour transformer un objet en chaîne de caractère

```
var perso = { nom: 'wick', prenom: 'john' };
var str = JSON.stringify(perso);
console.log(str);
// affiche {"nom":"wick","prenom":"john"}
```

Bookmarks
chrome://bookmarks

Pour transformer une chaîne de caractère en objet

```
var p = JSON.parse(str);
console.log(p.nom + " " + p.prenom);
// affiche wick john
```

Un objet avec attributs et méthodes

```
var obj = {
  nom: "wick",
  prenom: "john",
  direBonjour: function (){
    console.log("bonjour");
  }
};
```

Pour appeler la méthode `direBonjour`

```
obj.direBonjour();
```

Ceci génère une erreur

```
var obj = {
  nom: "wick",
  prenom: "john",
  direBonjour: function (){
    console.log("bonjour " + nom);
  }
};
obj.direBonjour();
```

Il faut utiliser l'opérateur `this`

```
var obj = {
  nom: "wick",
  prenom: "john",
  direBonjour: function (){
    console.log("bonjour " + this.nom);
  }
};
obj.direBonjour();
```

Déclarer un constructeur contenant une méthode

```
var Personne = function(nom,prenom){
  this.nom = nom;
  this.prenom = prenom;
  this.direBonjour = function (){
    console.log("bonjour " + this.nom);
  }
}
```

Pour appeler cette méthode

```
var perso = new Personne("wick","john");
perso.direBonjour();
```

Une fois le constructeur déclaré, impossible de lui ajouter de nouvel attribut de cette manière

```
Personne.age = 0;
```

Pour cela, il faut utiliser le prototype

```
Personne.prototype.age = 0;
```

Pareillement pour les méthodes

```
Personne.prototype.getAge = function(){
  return this.age;
}

Personne.prototype.setAge = function(age){
  this.age = age;
}
```

II) ECMAScript 5 & 6

- Mode strict
- var, let et const
- Fonctions fléchées
- POO
  - Classe
    - Attributs Instances
    - Attributs Static
    - Surcharge
  - Heritage
    - Redefinition
  - Module
    - export
    - import

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 (compatible avec les navigateurs modernes)
- ECMAScript version 6 (ES6) ou ES 2015 (nécessite, selon le navigateur, un transcompilateur vers ES5)

Le mode strict

- permet d'éviter d'utiliser une variable non-déclarée
- s'utilise comme une chaîne de caractère `"use strict"`

Pour utiliser le mode strict

```
"use strict";
```

Ceci déclenche donc une erreur

```
var x = 2;
y = 3;
```

Le mot-clé `let`

Bookmarks
chrome://bookmarks

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)
{
  var x = 1;
}
console.log(x);
// affiche ReferenceError: x is not defined
```

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 = 5;
x = "bonjour";
// affiche TypeError: Assignment to constant
   variable.
```

Il est devenu possible d'attribuer une valeur par défaut pour les paramètres d'une fonction

```
function division(x, y = 1)
{
  return x / y;
}

console.log(division(10));
// affiche 10

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

Il est aussi possible de déclarer une fonction en utilisant les expressions Lambda (les fonctions fléchées)

```
var nomFonction = ([les arguments]) => {
  les instructions de la fonction
}
```

Exemple

```
var somme = (a,b) => { return a + b; }
```

Appeler une fonction fléchée

```
var resultat = somme (1,3);
```

Les classes comme en programmation objet

- Créer des classes en utilisant la fonction constructeur
- Appliquer le principe d'encapsulation
- Faire de l'héritage
- ...

Pour déclarer une classe

```
class Point {
  constructor(abs, ord) {
    this.abs = abs;
    this.ord = ord;
  }
}
```

Instancier la classe ≡ créer un objet de cette classe

```
var point = new Point(2,4);
console.log(point); // affiche Point { abs: 2, ord: 4 }
```

Pour accéder à un attribut de cet objet

```
point.abs = 3;
console.log(point); // affiche Point { abs: 3, ord: 4 }
```

Pour rendre un attribut prive, on lui ajoute le préfixe # et on le déclare avant

```
class Point {
  #abs;
  #ord;
  constructor(abs, ord) {
    this.#abs = abs;
    this.#ord = ord;
  }
}
```

Ceci n'est plus possible et génère une erreur

```
point.#abs = 3;
console.log(point);
```

Tester en utilisant le navigateur (n'utilisez pas la commande node)

Nous pouvons définir un getter/setter pour chaque attribut qui auront le nom de l'attribut sans le préfixe #

Bookmarks
chrome://bookmarks

```
class Point {
  #abs;
  #ord;
  constructor(abs, ord) {
    this.#abs = abs;
    this.#ord = ord;
  }
  get abs() {
    return this.#abs;
  }
  set abs(abs) {
    this.#abs = abs;
  }
}
```

Ainsi, nous pouvons modifier et récupérer la valeur d'un attribut

```
point.abs = 3;
console.log(point.abs); // affiche 3
```

Pour l'héritage, on utilise le mot-clé `extends`

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
}

var personne = new Personne ("wick","john");
console.log(personne);
// affiche Personne {nom: "wick", prenom: "john"}

class Etudiant extends Personne {
  constructor(nom, prenom, bourse) {
    super(nom,prenom);
    // pour appeler le constructeur de la classe mère
    this.bourse = bourse;
  }
}

var etudiant = new Etudiant ("wick","john",500);
console.log(etudiant);
// affiche Etudiant {nom: "wick", prenom: "john", bourse: 500}
```

Supposant que l'on a une méthode qui retourne tous les détails de la classe `Personne`

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  afficheDetails() {
    return this.prenom + " " + this.nom;
  }
}
var personne = new Personne ("wick","john");
console.log(personne.afficheDetails());
// affiche john wick
```

Cette méthode `afficheDetails()` peut être redéfinie dans les classes filles

```
class Etudiant extends Personne {
  constructor(nom, prenom, bourse) {
    super(nom,prenom);
    this.bourse = bourse;
  }

  afficheDetails() {
    return this.prenom + " " + this.nom + " " + this.
       bourse;
  }
}

var etudiant = new Etudiant ("wick","john",500);
console.log(etudiant.afficheDetails());
// affiche john wick 500
```

La surcharge est aussi possible

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  afficheDetails() {
    return this.prenom + " " + this.nom;
  }

  afficheDetails(maj) {
    if (maj == false)
      return this.prenom + " " + this.nom;
    return this.prenom.toUpperCase() + " " + this.nom.toUpperCase();
  }
}

var personne = new Personne ("wick","john");
console.log(personne.afficheDetails(true));
// affiche JOHN WICK

console.log(personne.afficheDetails(false));
// affiche john wick
```

Il est possible de définir des attributs et/ou méthodes statiques (avec `static`)

```
class Personne {
  static nbrPersonnes = 0;
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
    Personne.nbrPersonnes++;
  }
}
console.log(Personne.nbrPersonnes);
// affiche 0

var personne = new Personne ("wick","john");
console.log(Personne.nbrPersonnes);
// affiche 1

var personne = new Personne ("abruzzi","john");
console.log(Personne.nbrPersonnes);
// affiche 2
```

Considérons la classe `Personne` définie dans le fichier `personne.js`

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
};
```

Pour l'utiliser dans un autre fichier, il faut

- l'exporter là où elle est définie
- l'importer là où on veut l'utiliser

Première méthode d’exportation

```
export class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
};
```

Deuxième méthode d’exportation

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
};
export { Personne };
```

Pour l'importer et l'utiliser dans un fichier `file.js`

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

var personne = new Personne ("wick","john");
console.log (personne);
// affiche Personne {nom: "wick", prenom: "john"}
```

Pour pouvoir utiliser l'importation d'un autre module JS, il faut ajouter l'attribut suivant dans la page HTML

```
<script src="file.js" type="module"></script>
```

On peut renommer l'élément exporté

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }
};
export { Personne as FirstClass };
```

On peut aussi renommer l'élément importé

```
import { FirstClass as Person } from "./personne.js";

var personne = new Person ("wick","john");
console.log (personne);
// affiche Person {nom: "wick", prenom: "john"}
```

On peut exporter et importer

- une classe
- un objet
- une fonction
- une variable
- une constante
- ...

TypeScript

- Transpilation
- Types
- Typage
  - Statique
  - Dynamique
  - Générique
- POO
  - Classe
  - Interface
  - Module
  - Enumeration

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**

Frameworks Frontend utilisant **TypeScript**

- **Angular** depuis la version 2
- **React** (voir `https://www.typescriptlang.org/docs/handbook/react.html`) et **Vue.js** (voir `https://fr.vuejs.org/v2/guide/typescript.html`) qui intègrent un support **TypeScript**

Achref EL Mo

Frameworks Backend utilisant **TypeScript**

- **ExpressJS** avec l'extension **TypeScript Node Starter** (voir `https://github.com/microsoft/TypeScript-Node-Starter`)
- **nest** Framework backend de **TypeScript**

Librairie écrite en **TypeScript**

RxJS (voir `https://github.com/ReactiveX/rxjs`)

API écrite en **TypeScript**

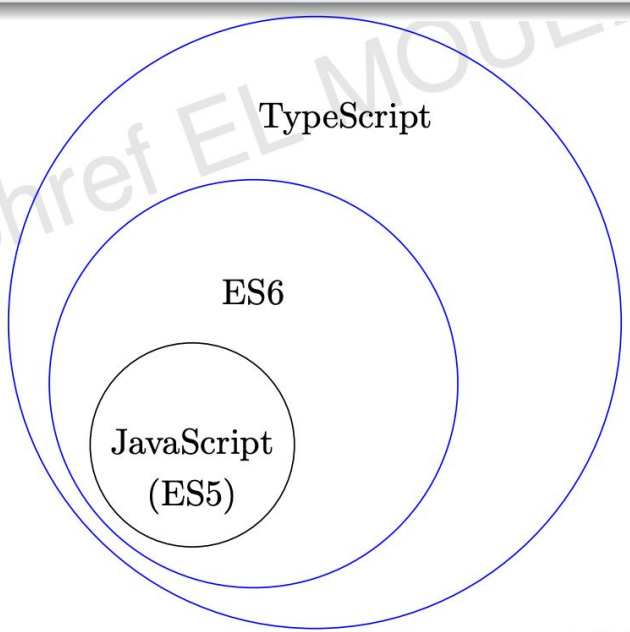
Firebase (voir `https://github.com/firebase/`)

Deno

Le nouveau projet de Ryan Dahl pour remplacer **NodeJS** écrit en **TypeScript**

TypeScript : sur-couche de ES6 ajoutant

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

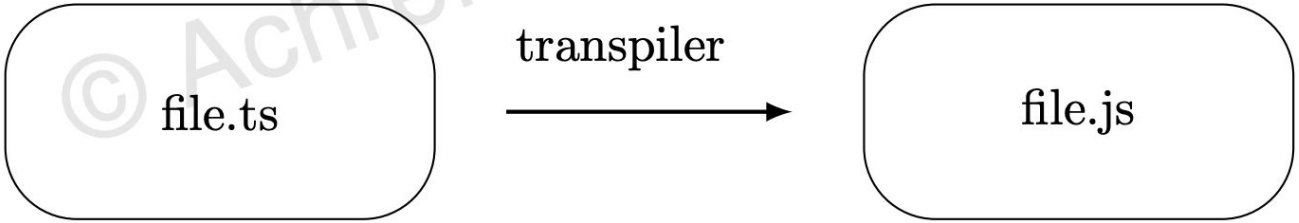


© Achref EL MOUL

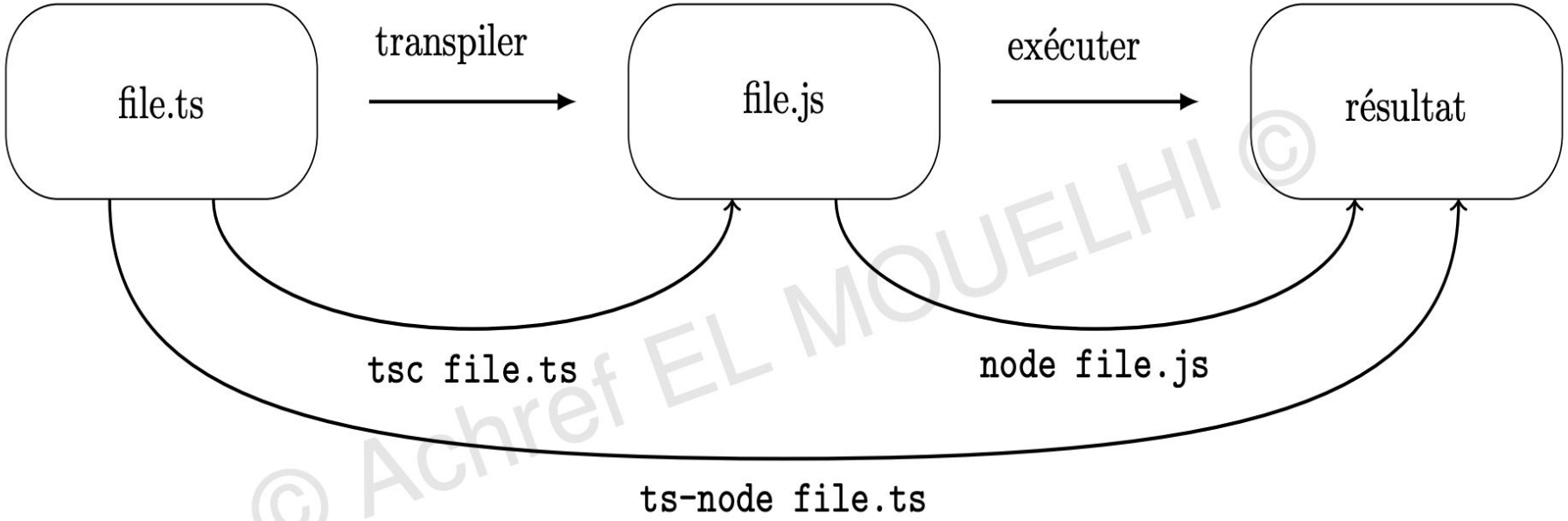
Le navigateur ne comprend pas **TypeScript**

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

© Achref EL MOUL



Comment va t-on procéder dans ce cours ?



© Achref EL MOUELHI ©

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

De quoi on a besoin ?

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

Pour **Node.js**, il faut

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

Bookmarks
chrome://bookmarks

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`

Pour **ts-node**, il faut

- lancer la commande `npm install -g ts-node`
- vérifier la version avec la commande `ts-node -v`

Les règles de nommage

- Pour les classes et interfaces : **Pascal case**
- Pour les variables, fonctions et méthodes : **Camel case**
- Pour les fichiers : **Kebab case**

Pour plus de détails

`https://wprock.fr/blog/conventions-nommage-programmation/`

Trois modes de typage en **TypeScript**

- Statique : le développeur précise le type à la déclaration de la variable
- Dynamique : le type d'une variable est déterminé à l'exécution (comme en JS)
- Générique : il permet au développeur de paramétrer un type complexe

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
- `symbol` pour les symboles [**ES6**]
- `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)

Bookmarks
chrome://bookmarks

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

Initialiser une variable

```
x = 2;
```

Déclarer et initialiser une variable

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

© Ashref EL MESSELHI ©

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 ?";
```

Une première déclaration de tableau

```
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 ]
```

MOUELHI ©

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
```

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;
```

© Achref EL MOUELHI ©

chrome://bookmarks

Exemple avec `any`

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

© Ashref EL MO

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;
```

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;
```

© Achref EL

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 };
```

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
```

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
```

On peut modifier les valeurs d'un objet ainsi

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

console.log(obj);
// affiche { nom: 'abruzzi', numero: 200 }
```

© Achre

Ceci est une erreur

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

Pour déclarer un symbole

```
const sym = Symbol(2);

console.log(sym);
// affiche Symbol(2)
```

Les symboles sont uniques

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

console.log(sym2 === sym3);
// affiche false
```

© Achref EL MOU

Un symbole peut-être utilisé comme une propriété d'objet (comme les chaînes de caractères)

```
const sym = Symbol(2);

let obj = {
    [sym]: "value"
};

console.log(obj);
// affiche { [Symbol(2)]: 'value' }

console.log(obj[sym]);
// affiche value

console.log(sym);
// affiche Symbol(2)
```

Union de type

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

WariCustomAthentificatorAuthenticator.php
todo: check the credentials inside F:\wamp64\...

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];
```

Najref EL MOUTAHI ©

Croisement de type

Il est possible qu'une variable ait les propriétés de plusieurs types différents

Déclarer une variable ayant les propriétés de plusieurs types différents

```
var enseignant: {
    nom: string;
    salaire: number;
};

var etudiant: {
    niveau: string;
};

var doctorant: typeof etudiant &  typeof enseignant = {
    nom: 'wick',
    salaire: 1700,
    niveau: 'master'
};
console.log(doctorant);
// affiche { nom: 'wick', salaire: 1700, niveau: 'master' }
```

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
```

© Achref EL MO

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
```

Déstructuration (ES6)

permet d'extraire les données d'un objet ou un tableau dans des variables.

bref EL MOUEL

Exemple

```
var personne = { nom: 'wick', prenom: 'john' };
var { nom, prenom } = personne;

console.log(nom, prenom);
// affiche wick john
```

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
```

MOUELHI ©

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);
```

© Mohamed EL MOUELHI

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){

    }
}
```

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
```

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
```

Bookmarks
chrome://bookmarks

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
```

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
```

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);
```

© Achret EL MOUELHI ©

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
```

© Pr. M. MOUELHI ©

Remarque

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

Prof EL MOUEL

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
```

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
- `some()` : pour vérifier s'il existe au moins un élément qui respect une condition
- `every()` : pour vérifier si tous les éléments respectent une condition
- `...`

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: number) {
    console.log(value);
}
// affiche 2 3 5
```

On peut simplifier l'écriture précédente en utilisant les callback

```
tab.forEach(afficher);

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

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.

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
```

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

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

WariCustomAthentificatorAuthenticator.php
todo: check the credentials inside F:\wamp64\...

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

console.log(somme);
// affiche 14
```

Dans cet exemple, on vérifie s'il existe un élément pair dans le tableau, après modification

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

let result = tab.map(elt => elt + 3)
    .some(elt => elt % 2 == 0);

console.log(result);
// affiche true
```

Dans cet exemple, on vérifie si tous les éléments du tableau sont pairs, après modification

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

let result = tab.map(elt => elt + 3)
    .every(elt => elt % 2 == 0);

console.log(result);
// affiche false
```

Fonctions fléchées : pourquoi ?

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

Bookmarks
chrome://bookmarks

Remarque

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

Prof EL MOUELHI ©

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

Considérons les deux objets suivants

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

Modifier l'un ⇒ modifier l'autre

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

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

TypeScript

Promesse

- introduite dans ES6
- utilisée souvent pour réaliser des traitements sur un résultat suite à une opération asynchrone
- disposant d'une première méthode `then()` permettant de traiter le résultat une fois l'opération accomplie
- disposant d'une deuxième méthode `catch()` exécutée en cas d'échec de l'opération
- composée de deux parties : déclaration et utilisation

Considérons la déclaration suivante d'une promesse

```
var test = true;
var promesse = new Promise<void>((resolve, reject) => {
  if(test)
    resolve();
  else
    reject();
});
```

Dans la partie utilisation, on doit indiquer ce qu'il faut faire dans les deux cas : réussite (`resolve`) ou échec (`reject`)

```
promesse.then(() => console.log("test réussi") )
        .catch(() => console.log("erreur") );
// affiche test réussi car test = true
```

Le `catch()` n'est pas obligatoire

Une promesse peut être déclarée avec une fonction fléchée

```
var test = true;
var promesse = () => {
  return new Promise<void>((resolve, reject) => {
    if(test)
      resolve();
    else
      reject();
  });
};
```

Pour l'utilisation, il faut appeler promesse comme une fonction

```
promesse().then(() => console.log("test réussi") )
          .catch(() => console.log("erreur") );
```

Exemple : déclaration

```
var division = (a: number, b: number) => {
    return new Promise((resolve, reject) => {
        if (b != 0)
            resolve(a / b);
        else
            reject("erreur : division par zéro");
    });
};
```

© Achref EL M

L'utilisation

```
division(10, 2).then((res) => console.log("résultat : " + res))
                .catch((error) => console.log(error));
// affiche résultat : 5

division(5, 0).then((res) => console.log("résultat : " + res))
               .catch((error) => console.log(error));
// affiche erreur : division par zéro
```

Remarque : les promesses s'exécutent d'une manière asynchrone

```
var division = (a: number, b: number) => {
  return new Promise((resolve, reject) => {
    if(b != 0)
      resolve(a / b);
    else
      reject("erreur : division par zéro");
  });
};

division(10, 2).then((res) => console.log("résultat : " + res))
               .catch((error) => console.log(error));

division(5, 0).then((res) => console.log("résultat : " + res))
              .catch((error) => console.log(error));

console.log("fin");
```

Le résultat est :

```
fin
résultat : 5
erreur : division par zéro
```

Le mot-clé `async`

Il permet de transformer une fonction en promesse

Considérons la fonction `somme()` suivante

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

Le résultat est :

```
console.log(somme(2, 3));
// affiche 5
```

© Achi EL MOUELHI ©

Pour transformer la fonction `somme()` en promesse, on ajoute le mot-clé `async` à sa déclaration

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

Maintenant, on peut l'utiliser comme une promesse

```
somme(2, 3).then(result => console.log(result));
// affiche 5
```

Appeler `somme()` comme une simple fonction JavaScript n'affiche pas le résultat

```
console.log(somme(2, 3));
// affiche Promise { 5 }
```

Le mot-clé `await`

- utilisable seulement dans des environnements asynchrones
- permettant d'interrompre l'exécution de la fonction asynchrone et attendre la résolution de la promesse

Considérons la promesse `somme()` qui attend 2 secondes pour retourner un résultat

```
let somme = (a: number, b: number) => {
    return new Promise((resolve) => {
        setTimeout(() => { resolve(a + b) }, 2000);
    });
};
```

On veut implémenter une promesse `sommeCarre()` qui utilise la promesse `somme()`

```
let sommeCarre = async (a: number, b: number) => {
    let s: number = 0;
    somme(a, b).then(result => s = result as number);
    let result = Math.pow(s, 2);
    return result;
};
```

Pour utiliser la promesse avec les valeurs `2` et `3`

```
sommeCarre(2, 3).then(result => console.log(result));
// affiche 0 car on n'a pas obtenu le résultat de somme lorsqu'
   on a calculé le carré
```

Solution, utiliser `await` pour attendre la fin de la première promesse

```
let sommeCarre = async (a: number, b: number) => {
    let s: number = 0;
    await somme(a, b).then(result => s = result as number);
    let result = Math.pow(s, 2);
    return result;
};
```

Si on teste maintenant

```
sommeCarre(2, 3).then(result => console.log(result));
// affiche 25
```

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.

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 };
```

Pour importer et utiliser une fonction

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

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

Karel EL MOUELH

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
```

Les alias peuvent être attribués à l'exportation

```
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 } ;
```

© Achref EL

Pour importer

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

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

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

Le mot-clé import peut être utilisé comme une fonction afin d'importer dynamiquement un module (import conditionnel...). Lorsqu'il est utilisé ainsi, il renvoie une promesse :

```
import('./fonctions')
    .then((module) => {
        console.log(module.somme(2, 5));
        // affiche 7

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

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

```
export default function somme(a:  WariCustomAthentificatorAuthenticator.php
                                  todo: check the credentials inside F:\wamp64\...
                                  ber = 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
```

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

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

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

© Achref EL M

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

- `"target": "es6"`
- `"strictPropertyInitialization": false`

En TypeScript

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

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();
```

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' }
```

© Achraf EL MOUDNI

Bookmarks
chrome://bookmarks

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;
}
```

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

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 préfixe l'attribut par un underscore.

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;
    }
}
```

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);
```

© Achr

Le résultat est :

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

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);
```

© Achr

Le résultat est :

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

Question

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

Démarche

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

© Achref El Mouelhi ©

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

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

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

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

Remarques

- Par défaut, toute classe **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.

© Achref E

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.

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;
}
```

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

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;
}
```

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

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

}
```

© Achret Ed

En exécutant, le résultat est le même

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

Bookmarks
chrome://bookmarks

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`

L'héritage, quand ?

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

(c) Achref EL M

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 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`

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

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");
```

© MOUELHI ©

Remarque

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

OUELHI ©

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
```

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.

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');
```

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);
}
```

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

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

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

© Aebref EL M

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

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

Modifions le code de deux méthodes générées

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

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

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)

© Achref EL M

Exemple

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

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' }
```

© Achrei

On peut rendre les attributs optionnels

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

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' }
```

© Achrer

On peut rendre les attributs optionnels

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

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.

© Achref

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 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);
```

Décorateur

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

© Achret EL

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

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

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

```
export function f() {
    return function (target: any, propertyKey: string, descriptor:
       PropertyDescriptor){
         console.log('before afficherDetails()');
    }
}
```

Testons maintenant le code suivant et vérifions que la fonction associée au décorateur a bien été exécutée

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

MOUELHI ©

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

- `"experimentalDecorators": true`
- `"emitDecoratorMetadata": true`

Généricité

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

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)**

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"
    }
  }
}
```

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
```

`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 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);
```

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');
```

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
*/
```

Pour afficher les clés et les valeurs

```
map.forEach(afficher);

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

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

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 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');
```

© Achraf EL MOUELHI

Pour ajouter plusieurs éléments à la fois

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

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' }
```

Pour parcourir un `Set`

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