Les bases TypeScript

Présentation

JavaScript

JavaScript a été créé en 1995 par Netscape puis repris par ECMA (European Computer Manufacturers Association) en 1996.

Le terme ECMAScript désigne donc le JavaScript, ECMAScript est actuellement (année 2020) à la version 11. La plupart des IDE vous propose ES6

Le JavaScript est un langage

- dynamique (contenu web)
- principalement côté client
- interprété
- orienté objet

TypeScript

TypeScript a été créé en 2012 par Microsoft sous licence Apache 2, TypeScript est actuellement (année 2020) à la version 3.9. TypeScript est une surcouche ou “superset” de JavaScript.

Par rapport à JavaScript, TypeScript apporte :

- Un typage obligatoire
- Une POO plus avancée
- une vérification du code avec TSLint (ou ESLint)

JavaScript permet aussi une vérification du code avec ESLint.

Par la suite, nous privilégions le TypeScript :)

Les outils

Pour développer avec TypeScript, vous devez installer les outils suivants:

- Node.JS : installez la dernière version LTS (Pour Debian/Ubuntu : utilisez plutôt le lien suivant : https://github.com/nodesource/distributions/blob/master/README.md). Vérifiez la version avec la commande `node --version`
- NPM : NPM est un gestionnaire de package. Déjà installé avec la commande précédente mais vous pouvez installer la dernière version de NPM avec la commande `npm install -g npm@latest`.

Les bases du langage

Pour le nommage nous utiliserons le camelCase (appelé aussi lowerCamelCase).

Pour déclarer une variable, vous devez utiliser les mots clés let ou const qui ont une portée bloc. Il faut bannir le mot clé var.

Les types de base

La déclaration se fait avec le format : {let | const} nomVariable: type [= valeur]; (voir Basic Types · TypeScript)

Exemple:

```
let isOpened: boolean = true;
let isClosed = false; // le type est déduit de la valeur
                      // cela s'appelle de l'inférence de type
const PI: number = 3.14;
let total: number; // valeur = undefined
const title = 'Mon titre';
let phrase = 'Ceci est';
phrase = `${phrase} aussi du texte; PI= ${PI}`;
let logins: string[] = ['john.doe', 'mr.smith', 'marcel.durand'];
enum Color {Red, Green, Blue}
let color: Color = Color.Green;
let notSure: any = 4; // peut recevoir tous les types
                      // à éviter si possible !
notSure = 'maybe a string instead';
notSure = false; // okay, definitely a boolean
```

Les opérateurs Égal et Différent

Il faut préférer les opérateurs strict aux opérateurs "faibles" :

- égalité: ===
- différence: !==

Les structures

Les structures classiques : if, switch, while, for

L'instruction foreach n'existe pas en TypeScript, mais elle est possible avec l'instruction for … of.

Attention :

- for … in fait une itération des clés
- for … of fait une itération des valeurs

Exemple :

```
let list = [4, 5, 6];
console.log('Interation in')
for (let i in list) {
   console.log(i); // outputs 0, 1, 2
}
console.log('Interation of')
for (let i of list) {
   console.log(i); // outputs 4, 5, 6
}
```

Les fonctions

Les fonctions sont “classiques” plus les apports de TypeScript.
Quelques exemples :

```
// fonction "classique"
function add1(x: number, y: number): number {
   return x + y;
}
// fonction avec paramètre optionnel indiqué par le ?
function add2(x: number, y: number, z?: number): number {
   if (z) {
       return x + y + z;
   }
   return x + y;
}
// fonction avec la construction d'arguments (paramètres) supplémentaires
function add(x: number, y: number, ...rest: number[]): number {
   let total = x + y;
   if (rest) {
       for(let val of rest) {
           total += val;
       }
   }
   return total;
}

// définition de fonction anonyme
const mult = (a: number, b: number): number => { return  a * b };
// affection de fonction
const autreNomDeMult = mult;


console.log(add(1, 2));
console.log(add(1, 2, 3));
console.log(add(1, 2, 3, 4, 5));
console.log(mult(4, 6));
console.log(autreNomDeMult(5, 6));
```

La déstructuration

Ce principe permet de séparer les valeurs selon l'ordre que vous donnez.

Quelques exemples :

```
let input = [1, 2];      // Déclare un tableau
let [first, second] = input;      // Assigne les valeurs du tableau aux variables : déstructuration
console.log(first); // outputs 1
console.log(second); // outputs 2

let [one, ...rest] = [1, 2, 3, 4]; // déstructuration vers 2 variables
      // les 3 points indique de prendre tout ce qui suit donc le reste
console.log(one); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

Plus d'exemples sur [Variable Declarations · TypeScript]

Un autre exemple que nous ne verrons pas forcément par la suite mais cela vous donne une idée de la déstructuration et des [classes "utilitaires"] comme Partial<T>:

```
interface Todo {
   `title: string;
    description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
    return { ...todo, ...fieldsToUpdate };
}

const todo1 = {
    title: 'organize desk',
    description: 'clear clutter',
};

const todo2 = updateTodo(todo1, {
    description: 'throw out trash',
});
```

Les classes

Grâce à TypeScript nous pouvons utiliser le mot clé "class" comme dans tous les langages orientés objets.
Par défaut, un membre de classe sans étiquette d'accès est public.

```
List<String> list = []; // Liste vide
List<String> list2 = ['one', 'two'];
```

Prenons l'exemple d'une voiture. Une voiture possède les caractéristiques suivantes:

- immatriculation
- nombre de porte (3 ou 5)
- catégorie: familiale, break, SUV
- et bien d'autre …

Ainsi que :

- un constructeur
- des accesseurs
- des méthodes

Première version

En s'inspirant d'un langage objet, la classe Voiture pourrait s'écrire ainsi:

```
export class Voiture {
   immatriculation: string;
   nombreDePortes: number;
   categorie: string;

   constructor(immatriculation: string, nombreDePortes: number, categorie:
string) {
       this.immatriculation = immatriculation;
       this.nombreDePortes = nombreDePortes;
       this.categorie = categorie;
   }

   description(): string {
       return `Voiture de type ${this.categorie} avec
${this.nombreDePortes} portes, immatriculation : ${this.immatriculation}`;
   }
}
```

Le mot clé `export` permet d'exporter (comme le mot clé public dans d'autres langages) la classe.
Le problème ici est que les attributs sont publics … pas très POO mais souvent utilisé en JS ou TS.
Exemple d'utilisation:

```
let audi = new Voiture('AA 123 AA', 4, 'SUV');
console.log(audi.description());
console.log(audi.immatriculation);
```

Deuxième version

En s'inspirant de C# (avec les propriétés et attributs privés), la classe Voiture pourrait s'écrire ainsi:

```
export class Voiture2 {
   private _immatriculation: string;
   private _nombreDePortes: number;
   private _categorie: string;


   get immatriculation(): string
   {
       return this._immatriculation;
   }
   get nombreDePortes(): number
   {
       return this._nombreDePortes;
   }
```

```
    get categorie(): string
    {
        return this._categorie;
    }

    constructor(immatriculation: string, nombreDePortes: number, categorie:
string) {
        this._immatriculation = immatriculation;
        this._nombreDePortes = nombreDePortes;
        this._categorie = categorie;
    }

    description(): string {
        return `Voiture de type ${this.categorie} avec
${this.nombreDePortes} portes, immatriculation : ${this.immatriculation}`;
    }
}
```

Remarque: Typescript “recommande” de mettre un underscore devant les membres privés

Troisième version

En utilisant les apports de TypeScript nous pouvons réduire le code pour les attributs:

```
export class Voiture3 {

    get immatriculation(): string
    {
        return this._immatriculation;
    }
    get nombreDePortes(): number
    {
        return this._nombreDePortes;
    }
    get categorie(): string
    {
        return this._categorie;
    }

    constructor(private _immatriculation: string, private _nombreDePortes:
number, private _categorie: string) {
    }

    description(): string {
        return `Voiture de type ${this.categorie} avec
${this.nombreDePortes} portes, immatriculation : ${this.immatriculation}`;
    }
}
```

Quatrième version

En JavaScript ou TypeScript, il est rare d'utiliser les membres privés. En utilisant des membres publics, le code pourrait s’écrire ainsi :

```
export class Voiture4 {
```

```
    constructor(public immatriculation: string, public nombreDePortes:
number, public categorie: string) {
    }

    description(): string {
        return `Voiture de type ${this.categorie} avec
${this.nombreDePortes} portes, immatriculation : ${this.immatriculation}`;
    }
}
```

A noter que l'on pourrait aussi mettre les attributs en privé.

Compléments

Avec typeScript nous pouvons utiliser :

- Les types d'accès private, protected, public
- Les classes et les méthodes abstraites

Interface

Nous pouvons donc utiliser les interfaces comme ceci:

```
interface Exemple {
    monAttribut: number;
    unAutreAttribut: string;
}
```

Nous pouvons aussi définir des attributs optionnels avec le point d'interrogation :

```
interface Exemple {
    monAttribut: number;
    unAutreAttribut: string;
    unAutreAttributOptionnel?: string;
}
```

Avec TypeScript, si nous voulons créer une classe de type DTO (aussi appelée POJO en Java ou POCO en C#), il faut créer un interface comme "modèle de données".
Exemple avec la classe Voiture5 avec un constructeur automobile :

```
export class Voiture5 {
    constructor(public immatriculation: string, public nombreDePortes:
number, public categorie: string, public constructeur: Constructeur) {
    }
    description(): string {
        return `${this.constructeur.name} de type ${this.categorie} avec
${this.nombreDePortes} portes, immatriculation : ${this.immatriculation}`;
    }
}
```

Version "longue" :

```
const audi: Constructeur = { id: 1, name: "Audi"};
let audiA5 = new Voiture5('AA 123 AA', 4, 'SUV', audi);
```

Version "courte" :

```
let audiA5 = new Voiture5('AA 123 AA', 4, 'SUV', { id: 1, name: "Audi"});
```

Manipulation de liste

Nous allons définir une interface pour les véhicules:

```
interface Vehicule {
   immatriculation: string;
   constructeur: string;
   modele: string;
   couleur: string;
   prix: number;
}
```

Puis nous allons l'alimenter :

```
const vehicules: Vehicule[] = [
   { immatriculation: 'AA-111-AA', constructeur: 'Peugeot', modele: '208',
couleur: 'Gris', prix: 20000},
   { immatriculation: 'AA-222-AA', constructeur: 'Peugeot', modele: '308',
couleur: 'Blanc', prix: 32000},
   { immatriculation: 'AA-333-AA', constructeur: 'Peugeot', modele: '3008',
couleur: 'Blanc', prix: 40000},
   { immatriculation: 'CC-111-CC', constructeur: 'Audi', modele: 'A4',
couleur: 'Gris', prix: 42000},
   { immatriculation: 'DD-111-DD', constructeur: 'Audi', modele: 'SQ5',
couleur: 'Noir', prix: 72000},
   { immatriculation: 'DD-444-DD', constructeur: 'Dacia', modele: 'Logan',
couleur: 'Noir', prix: 8200},
   { immatriculation: 'FO-777-RD', constructeur: 'Ford', modele: 'Mustang',
couleur: 'Blanc glacier', prix: 48000},
];
```

Et nous allons faire des manipulations de listes.

Parcourir la liste

La méthode forEach

Permet d'itérer la liste (équivalent de for of). forEach ne renvoie rien.
Exemple:

```
vehicules.forEach( vehicule => {
   console.log(`Véhicule : ${vehicule.immatriculation});

});
```

La méthode map

Permet d'itérer la liste mais aussi d'enchaîner les méthodes. map renvoie le résultat du traitement.
Exemple:

```
vehicules.map( vehicule => {
   console.log('Véhicule : ' + vehicule.immatriculation)
});
```

Autre exemple:

```
const list = [1, 2, 3, 4, 5];
let resultat = list.map(value => value*10).filter(value => value >= 30);
console.log(resultat)
```

Rechercher un élément

La méthode find permet de rechercher un élément dans la liste.
Exemple:

```
let vehiculeFord = vehicules.find( vehicule => vehicule.immatriculation === FO-777-RD'));
```

Ajouter un élément

La méthode push permet d'ajouter un élément dans la liste.
Exemple:

```
const citroen2cv: Vehicule = { immatriculation: 'CI-222-CV', constructeur: 'Citroën', modele: '2CV', couleur: 'Blanc', prix: 12000};
vehicules.push(citroen2cv);
```

Supprimer un élément

La méthode splice permet de supprimer un élément dans la liste.
Exemple:

```
// Suppression de la 308
const index = vehicules.findIndex(value => value.immatriculation === 'AA-222-AA');
if (index >= 0) {
   vehicules.splice(index, 1);
}
```

Filtrer les éléments

La méthode filter permet de filtrer la liste.
Exemple, filtre d'après le prix :

```
const vehiculePrixSup40ke: Vehicule[] = vehicules.filter(vehicule => vehicule.prix >= 40000);
console.log('Véhicules prix >= 40.000 : ', vehiculePrixSup40ke);
```

Exemple, filtre d'après le constructeur :

```
console.log('Véhicules Audi:', vehicules.filter(vehicule => vehicule.constructeur.toUpperCase() === 'AUDI'));
```

Trier les éléments

La méthode sort permet de trier une liste.
Exemple :

```
const vehiculePrixSorted = vehicules.sort((a, b) => a.prix - b.prix);
console.log('Véhicules prix croissant : ', vehiculePrixSorted);
```

Faire des calculs sur une liste

La méthode reduce permet de faire un calcul sur l'ensemble de la liste.
Exemple, calcul du prix total des véhicules de la liste :

```
const valeurInitiale = 0;
let totalVehicule = vehicules.reduce( (accumulateur, vehicule) => accumulateur + vehicule.prix
   , valeurInitiale
```

```
);
console.log('Véhicules total prix=' + totalVehicule);
```

Créer un premier projet

Création du projet

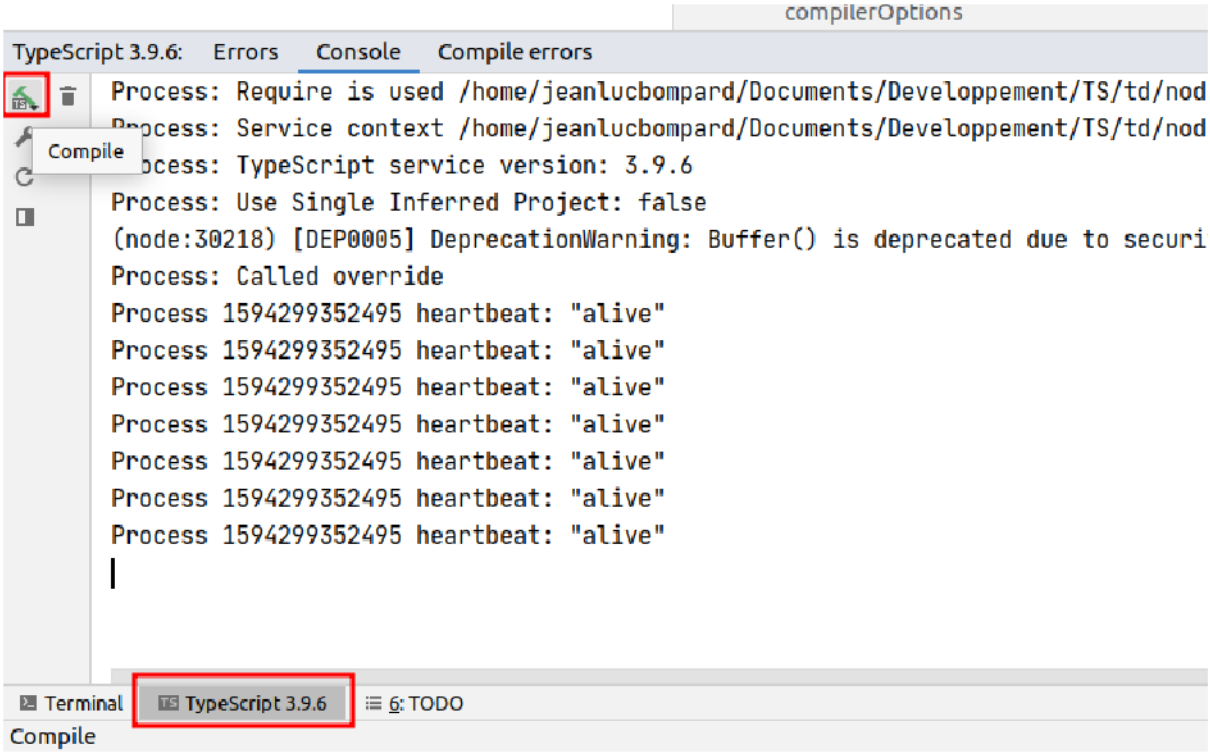
Créez un répertoire nommé “td” et ouvrez une console dans ce répertoire.

Pour créer le projet, vous devez :

- faire l’initialisation, commande : `npm init` (entrez juste la description et acceptez tout le reste
- installez TypeScript, commande : `npm install typescript --save-dev`. La commande crée un répertoire “node_modules” contenant tous les modules importés. Ce répertoire ne doit pas être listé par Git et peut être supprimé puis recréé avec la commande `npm install`
- ouvrir le projet avec WebStorm.
- créer un répertoire “src” et dans ce répertoire créer un fichier TypeScript nommé main.ts
- ajouter le fichier de configuration de TypeScript : clic droit sur le projet : New | tsconfig.json
- dans le fichier “tsconfig.json”, ajouter le chemin des fichiers sources et mettez es6:

```
{
 "compilerOptions": {
   "module": "commonjs",
   "target": "es6",
   "sourceRoot": "src",
   "sourceMap": true
 },
 "exclude": [
   "node_modules"
 ]
}
```

- compiler le projet : dans la console TypeScript (en bas), compiler le fichier tsconfig.json :



La compilation devrait réussir. Il est à noter que la compilation crée 2 fichiers:

- main.js : le fichier “transpiler” de main.ts
- main.js.map : information sur la génération du fichier main.js

La transpilation

Le langage TypeScript n’est pas interprété, il faut d’abord le traduire en JavaScript (ce n’est pas de la compilation). Le processus de traduction s’appelle la transpilation.

Chaque fichier d’extension .ts est transpiler en fichier .js et .js.map. Nous n’avons pas besoin de travailler sur ces fichiers; Nous allons donc les mettre dans un répertoire appelé “dist”.

A la racine du projet, créez un répertoire nommé “dist” pour la transpilation des sources.

Supprimez les 2 fichiers : “main.js” et “main.js.map” puis indiquer le répertoire de sortie dans le fichier tsconfig.json :

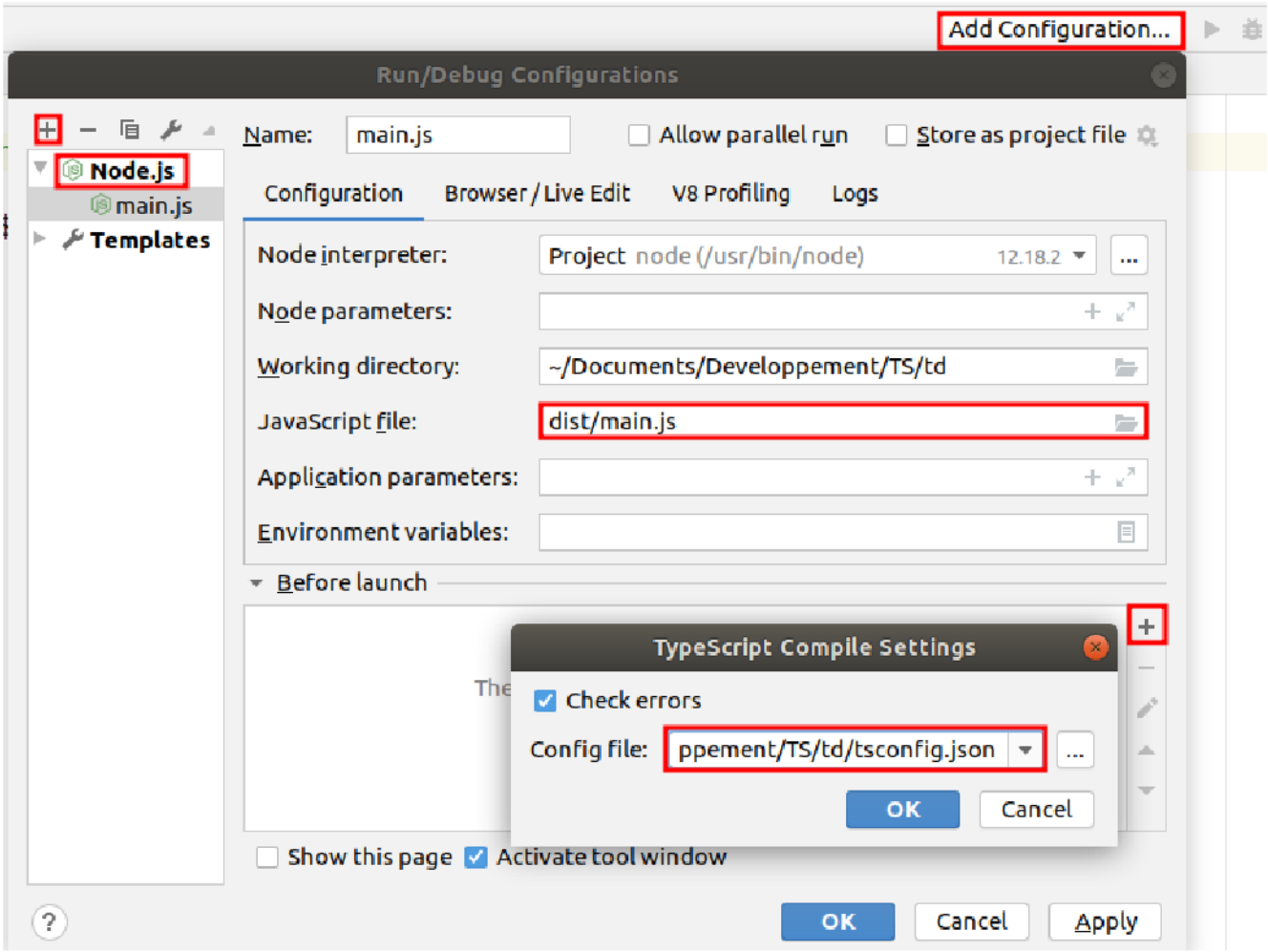
```
{
 "compilerOptions": {
   "module": "commonjs",
   "target": "es6",
   "sourceRoot": "src",
   "outDir": "dist",
   "sourceMap": true
 },
 "exclude": [
   "node_modules",
   "dist"
 ]
}
```

Recompilez le projet. Vous devez voir voir apparaître les fichiers main.js et main.js.map dans le répertoire “dist” (vous pouvez supprimer les fichier .js et .map dans le répertoire src).

Exécution du projet

Pour exécuter le projet, il faut

- ajouter une configuration indiquant le fichier principal de démarrage. Nous allons donc indiquer le fichier “main.js” qui est transpiler dans le répertoire “dist”
- ajouter une tâche avant l’exécution (de type “Compile TypeScript”):



La tâche permet de compiler le projet avant de l’exécuter.

Premier test

Pour varier un peu, nous allons faire un “Hello from Angular”.
Dans votre fichier “main.ts”, entrez le code suivant :

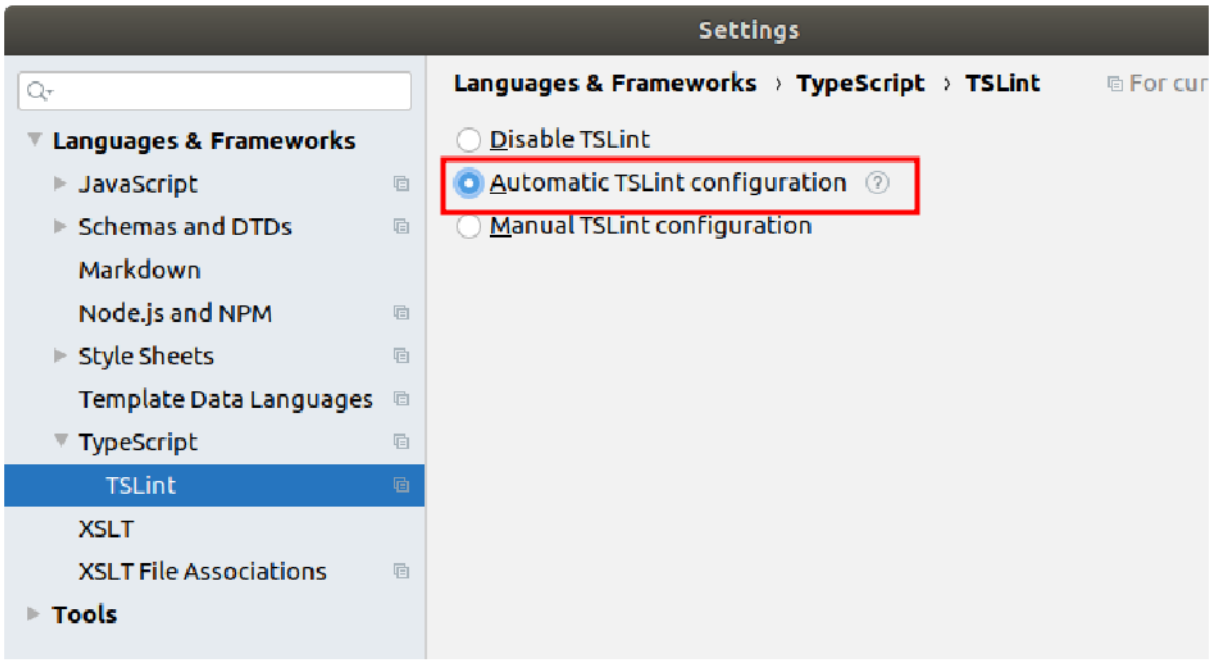
```
let message: string = "Angular";

console.log(`Hello from ${message}`);
```

Exécutez le projet et le message s’affiche !

Ecrire du code propre

TypeScript est typé et le langage recommande de bonnes pratiques. Pour mettre en oeuvre ces bonnes pratiques, nous allons utiliser TSLint.
Pour ce faire ouvrez les settings : “File | Setting | Languages & Frameworks | TypeScript | TSLint” et activez le :



TD Bibliothèque

Manipulation du langage avec la POO

Vous devez réaliser la gestion minimale d'une bibliothèque (voir document td-bibliotheque.pdf).

Les observables

(voir PowerPoint Angular)

Exemple d'utilisation d'observable

Le code suivant permet de créer et d'utiliser des observables :

```
// aaa
const subject = new Subject<number>();

// aaa
subject.subscribe({
   next: (v) => console.log(`observerA: ${v}`)
});
subject.subscribe({
   next: (v) => console.log(`observerB: ${v}`)
});

// aaa
subject.next(1);
subject.next(2);
```

Mise en œuvre pour la bibliothèque

Vous devez pouvoir dans le code de test vous abonner à un observable de la classe Bibliothèque. Cet observable émet l'emprunt qui vient d'être fait.

Faire des tests

Créez un répertoire nommé “tests” à la racine du projet et marqué le comme répertoire de test, clic-droit : “Mark Directory as | Test Source Root”.

Les dépendances

Nous allons utiliser les modules

- “chai” pour vérifier les résultats (avec expect … equal…)
- “ts-node” pour faciliter les tests
- “mocha” pour faire nos tests (avec assert(test, message))
- @types/mocha pour l’intégration de mocha avec TypeScript

Ouvrez le terminal dans WebStorm et installez les modules :

```
npm install chai --save-dev
npm install mocha --save-dev
npm install @types/mocha --save-dev
npm install ts-node --save-dev
```

Tests pour la classe Voiture

Modifiez le fichier “package.json” pour inclure les tests :

```
...
"scripts": {
 "test": "mocha -r ts-node/register tests/**/*.test.ts"
},
...
```

Tous les fichiers de tests devront donc se terminer par “test.ts”.
Dans le répertoire tests, créez le fichier “Voiture.test.ts” avec le code suivant :

```
import { expect } from 'chai';
import * as assert from "assert"

import {Voiture} from "../src/Voiture"

describe("Voiture", () => {

   it ("Test pour le nombre de portes", () => {

       let voiture: Voiture = new Voiture("immat", 2, "categorie");
       // Version avec chai
       expect(voiture.nombreDePortes).equal(2, "voiture.nombreDePortes
failed");
       // version avec types/node
       assert(voiture.nombreDePortes == 2, "voiture.nombreDePortes failed")
   });
});
```

Lancez les tests